Enumeration Algorithms for Real-World Networks: Efficiency and Beyond

Alessio Conte
Ph.D. Thesis

Università di Pisa
October 2017
Enumeration Algorithms for Real-World Networks: Efficiency and Beyond

Alessio Conte

Supervisor
Roberto Grossi
Università di Pisa

Supervisor
Maurizio Patrignani
Università Roma Tre

Referee
Etsuji Tomita
The University of Electro-Communications

Referee
Nadia Creignou
Aix-Marseille Université

October 2017
Abstract

Graphs are ubiquitous. They spawn from domains such as the world wide web, social networks, biological phenomena, roads and transportation, and are found in arguably all scientific fields. Graphs that model real-world phenomena can be extremely large, with up to billions of vertices and edges, and can store extensive amounts of information, both structural and semantic. A huge body of literature is dedicated to analyzing such networks to discover information, but this is a challenging task, due to their linked nature and their size.

Basic forms of graph analysis study some properties of graphs such as density or diameter. This can provide important information on the gross shape of a graph, but may not capture important details of substructures in a graph: while two similar graphs usually have similar properties, two graphs with similar properties may not be similar at all.

In this thesis we will consider the enumeration of subgraphs with desired properties in the context of real-world networks.

The analysis of subgraphs is a way to uncover this high quality information, which allows deeper understanding of the data and is a starting point for operations such as knowledge discovery, clustering and classification. Enumeration, when efficient, can also be a more flexible alternative to optimization algorithms, which comes in handy when the goal of the optimization is not clear or changes over time.

The goal of this thesis is on one hand to provide useful tools for network analysis and, on the other hand, to give a deeper understanding of algorithm design techniques that are reliable in this context, and how they can be applied effectively.

Finally, we take a step beyond enumeration, and consider the problem of dealing with the potentially large number of solutions of an enumeration algorithm, reducing their redundancy and improving their significance in some real-world case studies.
# Contents

1 Introduction  
1.1 Why enumerate?  
1.1.1 Application examples  
1.1.2 The challenge of real-world networks  
1.1.3 Beyond enumeration  
1.2 Content of the thesis  
1.2.1 Purpose  
1.2.2 Structure  
1.2.3 Published material  

2 Graph enumeration  
2.1 Preliminaries  
2.1.1 Graphs  
2.1.2 Set systems  
2.2 Enumeration and efficiency  
2.2.1 Definition  
2.2.2 Space efficiency  
2.2.3 Time efficiency  
2.3 Main techniques  
2.3.1 Recursion and iteration  
2.3.2 Recursion trees  
2.3.3 Binary partition  
2.3.4 Reverse search  
2.3.5 General algorithms for connected hereditary properties  
2.3.6 General algorithms for closed sets  
2.3.7 Enumeration by reduction  

2.4 Maximality  

2.5 Real-world networks  
2.5.1 Common features  
2.5.2 Common dynamics  
2.5.3 Exploiting the real-world network structure: some examples  

2.6 Limits of enumeration  
2.6.1 Structural complexity of listing: $\mathcal{L}$-P-completeness
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Listing strongly connected maximal cliques in directed graphs</td>
<td>97</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>97</td>
</tr>
<tr>
<td>6.2 Preliminaries</td>
<td>98</td>
</tr>
<tr>
<td>6.3 Problem characterization</td>
<td>99</td>
</tr>
<tr>
<td>6.3.1 Relaxed accessibility of strongly connected cliques</td>
<td>99</td>
</tr>
<tr>
<td>6.3.2 Maximal undirected and strongly connected cliques</td>
<td>100</td>
</tr>
<tr>
<td>6.3.3 Bounding the number of maximal SCQs</td>
<td>100</td>
</tr>
<tr>
<td>6.4 Listing maximal SCQs</td>
<td>102</td>
</tr>
<tr>
<td>6.4.1 Algorithm description</td>
<td>103</td>
</tr>
<tr>
<td>6.4.2 Correctness</td>
<td>104</td>
</tr>
<tr>
<td>6.5 Complexity analysis</td>
<td>105</td>
</tr>
<tr>
<td>6.6 Achieving polynomial space</td>
<td>106</td>
</tr>
<tr>
<td>6.6.1 Revised Lexicographical order</td>
<td>106</td>
</tr>
<tr>
<td>6.6.2 Parent-child relationship</td>
<td>107</td>
</tr>
<tr>
<td>6.6.3 Finding child solutions</td>
<td>108</td>
</tr>
<tr>
<td>6.6.4 Final algorithm</td>
<td>110</td>
</tr>
<tr>
<td>6.6.5 Correctness</td>
<td>111</td>
</tr>
<tr>
<td>6.6.6 Complexity analysis</td>
<td>111</td>
</tr>
<tr>
<td>6.7 Final remarks</td>
<td>113</td>
</tr>
<tr>
<td>7 Listing graph orientations with constraints on sources and cycles</td>
<td>115</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>115</td>
</tr>
<tr>
<td>7.2 Preliminaries</td>
<td>119</td>
</tr>
<tr>
<td>7.3 Single source orientations (ssso)</td>
<td>120</td>
</tr>
<tr>
<td>7.4 Single source acyclic orientations (ssao)</td>
<td>122</td>
</tr>
<tr>
<td>7.4.1 Iterating over valid direction assignments</td>
<td>122</td>
</tr>
<tr>
<td>7.5 Acyclic orientations (AO)</td>
<td>124</td>
</tr>
<tr>
<td>7.6 Cyclic orientations (CO)</td>
<td>126</td>
</tr>
<tr>
<td>7.6.1 Algorithm setup</td>
<td>126</td>
</tr>
<tr>
<td>7.6.2 Enumerating cyclic orientations</td>
<td>128</td>
</tr>
<tr>
<td>7.6.3 Finding extended orientations</td>
<td>129</td>
</tr>
<tr>
<td>7.6.4 Putting back the log-hole</td>
<td>129</td>
</tr>
<tr>
<td>7.7 Single source cyclic orientations (ssco)</td>
<td>132</td>
</tr>
<tr>
<td>7.7.1 Finding and removing a cycle not involving ( s )</td>
<td>133</td>
</tr>
<tr>
<td>7.7.2 Enumerating special orientations</td>
<td>133</td>
</tr>
<tr>
<td>7.7.3 Putting back the hole</td>
<td>134</td>
</tr>
<tr>
<td>7.8 Variations of the enumeration problems</td>
<td>135</td>
</tr>
<tr>
<td>7.9 From one-way streets to strong orientations</td>
<td>137</td>
</tr>
<tr>
<td>7.10 Finding strong orientations (SO)</td>
<td>139</td>
</tr>
<tr>
<td>7.10.1 Algorithm description</td>
<td>140</td>
</tr>
<tr>
<td>7.10.2 Correctness</td>
<td>141</td>
</tr>
<tr>
<td>7.10.3 Analysis</td>
<td>142</td>
</tr>
<tr>
<td>7.11 Final remarks</td>
<td>143</td>
</tr>
<tr>
<td>8 Listing maximal ( k )-degenerate subgraphs in a chordal graph</td>
<td>145</td>
</tr>
<tr>
<td>8.1 Introduction</td>
<td>145</td>
</tr>
<tr>
<td>8.2 Preliminaries</td>
<td>146</td>
</tr>
<tr>
<td>8.2.1 Chordal graphs and clique trees</td>
<td>147</td>
</tr>
<tr>
<td>8.2.2 ( k )-degenerate graphs</td>
<td>147</td>
</tr>
<tr>
<td>8.3 Hardness of the extension problem</td>
<td>147</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Polynomiality of restricted cases</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Hardness proof</td>
</tr>
<tr>
<td>8.4</td>
<td>Enumeration algorithm</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Greedy filling strategy</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Binary partition method</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Correctness</td>
</tr>
<tr>
<td>8.5</td>
<td>Complexity analysis</td>
</tr>
<tr>
<td>8.6</td>
<td>Final remarks</td>
</tr>
<tr>
<td>II</td>
<td>Beyond Enumeration</td>
</tr>
<tr>
<td>9</td>
<td>Finding diversified common substructures in protein networks</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>9.2</td>
<td>Output sensitive $T$-MCCIs's enumeration</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Problem transformation: implicit product graph</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Finding BC-cliques</td>
</tr>
<tr>
<td>9.3</td>
<td>Improving the usability of the result</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Filtering</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Recombining</td>
</tr>
<tr>
<td>9.4</td>
<td>Experimental evaluation</td>
</tr>
<tr>
<td>9.4.1</td>
<td>From proteins to graphs</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Experimental results</td>
</tr>
<tr>
<td>9.5</td>
<td>Final remarks</td>
</tr>
<tr>
<td>10</td>
<td>Finding a cover of diversified cliques</td>
</tr>
<tr>
<td>10.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>10.2</td>
<td>Clique covering</td>
</tr>
<tr>
<td>10.2.1</td>
<td>Clique cover graph</td>
</tr>
<tr>
<td>10.3</td>
<td>Framework and variants</td>
</tr>
<tr>
<td>10.3.1</td>
<td>Operation SELECT_UNCOVERED_EDGE</td>
</tr>
<tr>
<td>10.3.2</td>
<td>Operation FIND_CLIQUE_OF</td>
</tr>
<tr>
<td>10.4</td>
<td>Implementation and complexity</td>
</tr>
<tr>
<td>10.5</td>
<td>Experimental evaluation</td>
</tr>
<tr>
<td>10.6</td>
<td>Experiments with real-world and synthetic networks</td>
</tr>
<tr>
<td>10.6.1</td>
<td>Data sets and experimental guidelines</td>
</tr>
<tr>
<td>10.6.2</td>
<td>Analyzing clique size distribution</td>
</tr>
<tr>
<td>10.6.3</td>
<td>Vertex covering index</td>
</tr>
<tr>
<td>10.6.4</td>
<td>Edge covering index</td>
</tr>
<tr>
<td>10.6.5</td>
<td>Time performance and scalability</td>
</tr>
<tr>
<td>10.7</td>
<td>Conclusions</td>
</tr>
<tr>
<td>11</td>
<td>Finding significant itemsets</td>
</tr>
<tr>
<td>11.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>11.2</td>
<td>FIM as a problem on bipartite graphs</td>
</tr>
<tr>
<td>11.3</td>
<td>Our goal</td>
</tr>
<tr>
<td>11.4</td>
<td>I-GRAPH</td>
</tr>
<tr>
<td>11.4.1</td>
<td>Building the I-GRAPH</td>
</tr>
<tr>
<td>11.4.2</td>
<td>Complexity</td>
</tr>
<tr>
<td>11.5</td>
<td>Problem decomposition</td>
</tr>
</tbody>
</table>
### 11 Finding an item clustering
- 11.5.1 Finding an item clustering ................................................. 198
- 11.6 Dynamic threshold decomposition ........................................... 199
- 11.7 Experimental results .......................................................... 199
- 11.8 Final remarks ...................................................................... 202

### 12 Listing large k-plexes
- 12.1 Introduction ........................................................................... 203
- 12.2 Overview .............................................................................. 204
- 12.3 Finding maximum k-plexes
  - 12.3.1 Criteria ........................................................................... 207
  - 12.3.2 Algorithms for maximum k-plexes ................................. 208
- 12.4 Finding large k-plexes .......................................................... 210
- 12.5 Theoretical basis ................................................................. 212
- 12.6 Experiments
  - 12.6.1 Results .......................................................................... 214
- 12.7 Related works ..................................................................... 219
- 12.8 Final remarks .................................................................... 220
1. Introduction

1.1 Why enumerate?

Graphs can store extensive amounts of information, either in their structure or in the semantic information embedded in them, and a huge body of literature is devoted to analyzing and mining this information. Extracting information from graphs, especially large graphs, is a hard task due to their linked nature. Algorithms on graphs are rarely data-friendly, that is, they usually require access to random points in the data without a specific order or logic other than the topology of the graph at hand.

The most basic form of graph analysis which is to analyze graph properties, which correspond to values which summarize some dynamic of a graph. Some examples are the density, which measures the interconnectedness of the graph, the diameter, or effective diameter, which measure the distance among vertices in the worst or average case, and the clique number, which corresponds to the size of the largest complete subgraph in the graph.

Analyzing properties is often efficient, as they usually are simple numeric values that are either easy to compute or can be approximated, and provides important information on the gross shape of a graph. On the other hand, studying properties does not provide high quality information. While we can say that two graphs whose properties have values that differ greatly are indeed different, when the properties have similar values we have no information on the graphs’ similarity. Take as an example the graphs shown in Figure 1.1. It can be seen at first sight how these graphs differ in their structure. One has well defined dense groups (or communities) which are loosely connected to each other. In the other graph, vertices are connected randomly with other vertices and no clear community can be identified. These two graphs, however, have exactly the same number of vertices and edges, hence the same density, and both have the same diameter, that is the maximum distance between two vertices. This is an example of how studying the average shape of a graph can fail to provide accurate information on its structure.

The analysis of patterns in networked data such as graphs can yield important information on the structure being modeled, which allows deeper understanding of the data and is a starting point for operations such as high quality clustering and classification. A pattern is defined as any set of elements (e.g., vertices or edges) which respects some given property (e.g., being a triangle). In particular, enumerating patterns and structures is a way to discover important features in graphs, and to perform custom and high quality analysis, as the presence of different patterns can reflect the different dynamics involved in the structure of a network. To give an example, if we enumerate the cliques of the graphs in Figure 1.1, we will find out that the left one has larger cliques and quasi-cliques than the one on the right; this gives us some tangible measure that supports our intuitive judgment, i.e., that the graph on the left contains communities while the
Algorithms for enumerating subgraphs have a long history, especially in the context of enumerative combinatorics and computational complexity [Gol91, LLRK80, Rea81, SW86, Val79], and their influence has spread to a wide number of other fields, massively involving algorithm design techniques. A comprehensive list of enumeration algorithm can be found in [Was16].

Another reason for enumerating is that it provides a flexible alternative to optimization algorithms. Optimization problems on graphs are fundamental in computer science. Researchers have spent great effort in the formulation of such problems and in producing fast algorithms to solve them. Given a pattern or in general a query, optimization finds the one result of maximum value according to some objective function.

This kind of result, however, is not always suitable to solve real-world problems. These problems often have fuzzy and loosely defined objective functions that are easier to understand for humans than they are to express in a formal language. Enumeration algorithms are an excellent tool for such cases, as they will enumerate all possible solutions to the problem, to then allow further analysis or filtering by a different source. On one hand, this is likely to be slower than an ad-hoc approach for cases where efficient ad-hoc approaches have been designed, as the
number of solution may be large; on the other hand, this strategy eliminates the need to design
a brand new algorithm for each variation of the goal function. Furthermore, an enumeration
algorithm may be combined with more than one goal function, so that the user may obtain from
the algorithm the best solution (or, e.g., the $k$ best solutions) for each goal function and finally
choose the most suitable among this restricted pool of possible solutions.

1.1.1 Application examples

Road networks. If one has ever looked at a map, it should appear evident that roads can
be conveniently modeled as graphs. Not surprisingly, many works do so, and some of these use
directed graphs to model the problem of assigning the right direction to one-way roads: when
driving on a road, the very least we could demand from the network is that from our position
we may reach all other places in the area, and vice versa; otherwise we may become stuck in a
“sink” or simply be unable to reach our destination, both of which are things we do not want
to happen. When representing a road network as a graph, satisfying this requirement is called
giving a strong orientation to the graph, and it is also known as the one-way street problem.
Several variations of the one-way street problem have been studied from an optimization point
of view, with the purpose of minimizing some goal function that determines the goodness of
the resulting road network. Some have tried to minimize the average distance between any
two points [DOW04], some the maximum distance [FMR04] [FMPR04] [Gut94] [LK12] or the
maximum weighted distance [BFKW99], i.e., maximum distance where each edge has a custom
weight. Others tried to minimize the distance stretch for each pair of vertices [HM89], maximize
some degree-based constraints [BAGN15], or partially solve the problem for only some pairs of
vertices [AH02]. While each of this algorithms is likely the most efficient known way of solving
the specific problem they consider, an issue becomes evident: every time one wishes to consider
a slight variation of our goal function, they need to develop a brand new ad-hoc algorithm.
Furthermore, usually these algorithms are only able to evaluate a single property of the desired
solution, while the techniques may not adapt to combinations of desired properties. This is
not always ideal in real-world situations, where for example we may want the average distance
between two points to be small, and add some additional constraint to promote load balancing
of the traffic. An efficient enumeration algorithm may offer a convenient trade-off in this kind
of situation: indeed, given an algorithm that lists all and only strong orientations of a graph,
in a reasonable amount of time, it is enough to write any arbitrary evaluation function for the
solutions, and simply evaluate every solution to find the most suitable.

Database queries. Querying a database is one of the most basic and fundamental operation
in informatics and, in fact, this is an enumeration problem. This problem is often simple under a
theoretical point of view, such as in relational databases, and effort is devoted in optimizing its
performance with clever implementations and sophisticated caching techniques.

In other cases, however, the query itself may involve some complexity. This can be the case
for example when querying structured and linked data, such as graph databases. In these cases,
a query can sometimes be modeled as a problem of subgraph enumeration [CKS08].

Furthermore, subgraph enumeration techniques may be the key to dealing with real data,
which is often incomplete or faulty, as they can be exploited to retrieve approximated answers to
queries that have no exact result. [CS05]

Maximal cliques. The problem of community detection in graphs has been extensively studied.
In undirected graphs, dense subgraphs are often used to detect communities, with applications
in areas such as social network analysis [Sco12] [WF94], biology [JS11], and more [For10].
Several definitions of dense subgraph have been proposed to model communities [PYB12, WF94]. The earliest, and perhaps the most widely studied is that of the maximal clique: interest in the problem of finding maximal cliques started several decades ago [Akk73, BK73, MM65] and effort to produce efficient algorithms can still be seen in recent works [CRL15, Akk73, CGMV16, ELS13].

**Frequent Itemset Mining.** Frequent Itemset Mining (FIM) is a fundamental data mining technique, extensively used in transaction-based domains such as marketing [UKA04]. The aim of FIM is finding sets of items that are frequently chosen together. The instance of the problem is usually represented as a collection of *baskets*, which are sets or multisets of items. The classical setting that gives this naming is that of observing customer behavior in stores, as the knowledge that product $X$ is frequently bought by people that buy product $Y$ may enable sellers to make intelligent marketing strategy to promote one product or the other.

The principle that drives FIM is to find itemset with large *support*, which correspond to groups of items frequently chosen together.

Given an itemset (set of items) $A$, the support is the set of baskets containing all items in $A$.

An itemset is called frequent if its support has size larger than a given threshold $\tau$, closed if no item can be added to it without decreasing the size of its support, and maximal if no item can be added to it without decreasing its support below $\tau$. While techniques for this kind of frequent patterns exist (see Section 2.3.6), FIM has an interesting combinatorial property: closed itemsets correspond to maximal *bipartite cliques* on a bipartite graph suitably defined on the baskets and items.

Algorithms for listing bipartite cliques have thus been used to give an efficient solution to this problem. In particular, the algorithm LCM by Uno et al. [UAUA03], which is based on bipartite clique enumeration, was the winner of the FIMI 2004 challenge for frequent itemset mining algorithms. [04]

### 1.1.2 The challenge of real-world networks

A great number of enumeration algorithms can be found in literature: an ever growing list of enumeration algorithms can be found in [Was16]; as of October 2017 it lists solutions for over 400 enumeration problems, referencing close to 300 scientific publications. When dealing with a well known pattern or structure it is likely that someone already designed an enumeration algorithm for it. However, different context sometimes require different kinds of algorithms: for example, traditional graph algorithms are sometimes not tailored to deal with very large networks, and may incur in issues deriving from networks size such as insufficient memory or simply an enormous running time. These problems, however, may be avoided by handling the problem differently. This motivates the need of new techniques ad-hoc for real-world networks, which may be very large, but whose common properties may be taken advantage of to produce efficient algorithms.

Real-world networks, on one hand, can be extremely large, but on the other hand are sometimes relatively easy to process with respect to their size. For example, while the number of maximal cliques in a graph can be exponential, Schmidt et al. [SSTP09] show how real-world networks contain number of maximal cliques which is sometimes even linear in the size of the graph, and in all cases much smaller than the theoretical worst case of $3^n/3$. This suggests that we should look at real-world graphs through different lenses than those used for generic ones. In Section 2.5 we attempt to give a more accurate view of real-world graphs, and to show how this information can be used to drive algorithm design.
1.1.3 Beyond enumeration

Enumeration is usually not the end of the story. Simply counting the occurrences of some pattern (e.g., triangles) gives a definitive answer to a question, and while enumeration implicitly solves the simpler problem of counting the patterns which have been enumerated, the output is far more complex. Sometimes, the user is only interested in finding one optimal solution (e.g., the best team to solve a task), and needs to apply some complex measure in order to evaluate each one. In other cases, the user may be interested in finding more than one solution (e.g., in community detection, where the user may want a set of communities representative of the social structures in a network), but may need fewer solutions that those found by the enumeration algorithm, which can often be exponential. In both cases, we could say that the usability of the result should be improved. Some solutions to this problem have been proposed: one example is top – \( k \) approaches which return the \( k \) best solution according to some score, as well as other approaches that aim at reducing the amount of enumerated solutions. However, no definitive answer, or widely accepted strategy has been given so far. In the final part of this thesis we will show some study cases, in which we attempt to improve usability of the result of enumeration problems. As there is no definitive way to do this, we will attempt case-to-case to identify a suitable solution for the problem at hand.

1.2 Content of the thesis

1.2.1 Purpose

In this thesis we will consider the enumeration of some well known and some lesser known patterns in the context of real-world networks: we will tailor our algorithms to properly handle the available resources, and to make the most out of the structure of the input and the problem at hand to be efficient. We will later give a more formal definition of what efficient means.

In this thesis we address problems related to enumeration in the context of real-world networks. We use different techniques for different problems and showcase the solutions found, explaining how the techniques involved are applied. The goal of the thesis is two-fold:

Firstly, we aim at identifying which techniques are the most suitable and reliable for enumeration in real-world graphs. This is to serve as inspiration for the reader wishing to solve new problems in the area, who may gain a deeper understanding of the techniques considered and evaluate their strengths and weakness by studying the proposed application examples.

Secondly, the presented results themselves constitute a collection algorithms that address well known problems, and can be valuable tools for analyzing networks. Significant effort has been put into producing algorithm which have a simple structure and are easy to implement, while the technical complexity is segregated as much as possible in their analysis. This way, even the reader just interested in using the algorithms should be able to effectively implement and apply them with minimum effort.

1.2.2 Structure

The thesis is structured as follows:

Chapter 2 gives an introduction to the area, introducing fundamental concepts such as the terminology and notions of efficiency, some of the most important techniques used to design output-sensitive algorithms, that are those considered in this thesis, and makes an overview of the challenges posed by real-world networks and enumeration in general.
Part I

This part contains the main body of the thesis, that is a collection of enumeration algorithms for different problems which find applications in the context of real world networks. As efficient enumeration requires a careful analysis of the problem at hand, and optimization techniques derived from its structure, we address each problem individually. Due to their technical similarity, we are able to make an exception for listing problems related to graphs orientations, which are described together in Chapter 7. The problems addressed and results obtained are summarized in Table 1.1 (top half).

For the reader interested in the algorithmic techniques, Chapters 3 to 6 uses the framework of reverse search introduced in Section 2.3.4 (Chapters 5 and 6 also extend the technique by Cohen et al. described in Section 2.3.5). Chapters 7 and 8 are instead based on the simpler, yet effective, binary partition technique, introduced in Section 2.3.3.

Part II

This part concerns the usability of the result of enumeration problems. In particular, we consider practical scenarios in which enumeration can be used to find useful information, but in which the potentially exponential number of solutions makes the result difficult to process.

For each problem, we devise techniques which have good practical performance and which let us restrict our attention to a smaller subset of interesting solutions, which can be processed more easily by the user. The problems addressed and results obtained are summarized Table 1.1 (bottom half).

1.2.3 Published material

The contributions contained in this thesis have been partially or completely described in the following publications (authors are in alphabetical order):


<table>
<thead>
<tr>
<th>CHP</th>
<th>PROBLEM</th>
<th>RESULT</th>
<th>PUBLICATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>maximal cliques</td>
<td>sublinear space, minimal space P-Delay</td>
<td>[CGMV16]</td>
</tr>
<tr>
<td>4</td>
<td>maximal independent sets</td>
<td>minimal space, P-DELAY</td>
<td>[CGM+17b]</td>
</tr>
<tr>
<td>5</td>
<td>maximal solutions of strongly accessible set systems</td>
<td>minimal space, P-DELAY for some cases</td>
<td>[CGMV18] (preprint)</td>
</tr>
<tr>
<td>6</td>
<td>maximal strongly connected cliques</td>
<td>linear space, P-Delay</td>
<td>[CKUW17a]</td>
</tr>
<tr>
<td>7</td>
<td>graph orientations (cyclic, acyclic, single/multiple sources, strongly connected)</td>
<td>linear space, P-Delay, sos for some cases</td>
<td>[CGMR17] [CGM+16b] [CGMR16] [CGMR15]</td>
</tr>
<tr>
<td>8</td>
<td>$k$-degenerate subgraphs (in chordal graphs)</td>
<td>polynomial space, P-Delay</td>
<td>[CKO+17]</td>
</tr>
</tbody>
</table>

**Contributions of Part II (beyond enumeration)**

<table>
<thead>
<tr>
<th>CHP</th>
<th>PROBLEM</th>
<th>RESULT</th>
<th>PUBLICATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>finding large common biological substructures</td>
<td>heuristic, outperforms state of the art in quality and performance</td>
<td>[CGM+17a]</td>
</tr>
<tr>
<td>10</td>
<td>finding an edge clique cover</td>
<td>heuristic, outperforms state of the art in quality and performance and scales to larger networks</td>
<td>[CGM16a]</td>
</tr>
<tr>
<td>11</td>
<td>finding frequent itemsets</td>
<td>heuristic, outperforms state of the art in finding frequent itemsets with low support</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>finding large $k$-plexes</td>
<td>exact, outperforms state of the art</td>
<td>[CFM+17]</td>
</tr>
</tbody>
</table>

Table 1.1: Problems concerning enumeration (Part I) and improving the usability of enumeration results (Part II) considered in this thesis.


7. **[CGMV16]** Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Sublinear-space bounded-delay enumeration for massive network analytics: Maximal cliques. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July*


Other publications the author worked on during the PhD studies include:


Some of the publications were supported by the following grants:
PRIN 2012C4E3KT national research project AMANDA—Algorithmics for Massive and Networked Data. By the Italian Ministry of Education, University, and Research (MIUR).

UNIPI PRA 2015 0058 research project Computational Methods for Personalized Medicine. By the University of Pisa.

JPMJCR1401 By the Japan Science and Technology Agency (JST), CREST program.
2.1 Preliminaries

In this section we introduce the basic concepts that will be used in the remainder of the thesis. First we will define the most commonly used graph types, then present \textit{set systems}, a generic formalism which can be used to model and classify enumeration problems. The terminology is consistent with that in Diestel’s well known graph theory book [Die05].

2.1.1 Graphs

A graph $G = (V(G), E(G))$ consists of a set of \textit{vertices} $V(G)$, and a set of \textit{edges} $E(G) \subseteq (V(G) \times V(G))$, which represent relationships (or links) between two vertices.

For simplicity, we will omit the subscript $G$ when this is clear from the context. We will thus often simply use notation such as $V$ and $E$ instead of $V(G)$, $E(G)$.

We will often refer to the number of vertices ($|V(G)|$) as $n$, and to the number of edges ($|E(G)|$) as $m$. Vertices of a graph are assumed to be in a linear ordering, \textit{i.e.}, $V(G) = \{v_1, v_2, \ldots, v_n\}$, may be arbitrary or have specific properties. For two vertices $v_i$ and $v_j$, we say that $v_i$ is \textit{lexicographically} smaller (or simply smaller) than $v_j$ if $i < j$. Using this ordering we define $V_{\leq v_i}(G)$ as $\{v_1, v_2, \ldots, v_{i-1}\}$, and $V_{< v_i}(G)$ as $V_{\leq v_i}(G) \cup \{v_i\}$. If $A$ is a set of vertices, $A_{\leq v_i}$ is $A \cap V_{\leq v_i}(G)$, $A_{< v_i}$ is $A \cap V_{< v_i}(G)$, $V_{\geq v_i}(G)$, $V_{> v_i}(G)$, $A_{\geq v_i}$ and $A_{> v_i}$ are similarly defined.

In \textit{undirected} graphs an edge in $E(G)$ between two vertices $x$ and $y$ is an unordered pair, that is a set $\{x, y\}$. Note that $\{y, x\}$ represents exactly the same set and edge. $x$ and $y$ are called the \textit{extremes} of the edge. The \textit{neighborhood} of the vertex $v$ in $G$ is the set $N_G(v) = \{w : \{v, w\} \in E(G)\}$, the size $|N_G(v)|$ of its neighborhood is the \textit{degree} of $v$. We denote by $\Delta(G)$ the highest degree of a vertex in $G$. Any vertex with degree 0 is called \textit{isolated}, and any vertex with degree $|V| - 1$, \textit{i.e.}, adjacent to all other vertices, is called \textit{universal}.

With respect to the given ordering, we define the \textit{forward} of vertex $v$ as its neighbors which come later in the ordering, \textit{i.e.}, $N_>(v) = N(v) \cap V_{> v}$, and similarly the \textit{backward} neighbors as $N_<(v) = N(v) \cap V_{< v}$.

All graphs considered in this thesis are \textit{finite}. Unless stated otherwise, graphs are \textit{simple} graphs, which means they do not contain self-loops, that are edge whose extremes are the same vertex, nor multiple edges between the same two extremes. Graphs which allow self-loops and multiple edges are called \textit{multigraphs}.

\textit{Directed} graphs (also called digraphs) have directed edges, thus $E(G)$ contains \textit{ordered} pairs where one vertex is the \textit{tail} and one the \textit{head}. $(x, y)$ represents an edge whose tail is $x$ and head $y$, and we say that the edge is \textit{from} $x$ \textit{towards} $y$, as that is the direction the edge is meant.
to be navigated. The neighborhood of a vertex $v$ is the set $N_G(v) = \{w : (v, w) \in E(G)\}$ or $(w, v) \in E(G))$. We also define the out-neighborhood of $v$ as the set of vertices with an edge from $v$, i.e., $N_G^+(v) = \{w : (v, w) \in E(G)\}$, and the in-neighborhood as that of vertices with an edge towards $v$, i.e., $N_G^-(v) = \{w : (w, v) \in E(G)\}$. Note that the directed edges $(x, y)$ and $(y, x)$ are two different edges, called symmetric edges and may appear in the same graph even though it is not a multigraph. A graph where symmetric edges are not allowed to co-exist is called an oriented graph, as it can be obtained by orienting (i.e., giving a direction to) the edges of an undirected graph.

Furthermore, we say that a vertex $x$ is a sink w.r.t. a set of nodes $S$ if $N_G^+(x) \cap S = \emptyset$, and a source w.r.t. $S$ if $N_G^-(x) \cap S = \emptyset$. When $x$ is a sink (source) w.r.t. $V(G)$ we simply say that $x$ is a sink (source).

In situations where ambiguity among directed and undirected edges may arise, we refer to directed edges as arcs, we denote sets of arcs with above arrows, e.g., $E(G)$.

If $G(V(G), E(G))$ is a directed graph, we denote by $\overline{G}(G)$, called underlying (undirected) graph of $G$, the undirected graph with vertex set $V(G)$ and edge set $\{(x, y) : (x, y) \in E(G)\}$ or $(y, x) \in E(G))$.

An induced subgraph of $G$, given a set of vertices $X \subseteq V(G)$, is the subgraph $G[X] = (X, E[X])$, where $E[X] = E(G) \cap (X \times X)$. In other words, $G[X]$ is the subgraph obtained by removing all vertices in $V(G) \setminus X$ and all edges incident to those vertices from $G$. An edge subgraph of $G$, given a set of edges $F \subseteq E(G)$, is the subgraph $G[Y] = (V[Y], Y)$, where $V[Y]$ is the set of vertices incident to an edge in $Y$, i.e., $V[Y] = \{x : \{x, y\} \in Y\}$. In other words, $G[Y]$ is the subgraph obtained by removing all edges in $E(G) \setminus Y$ and all vertices that became isolated vertices in the process.

A path $P$ of length $k$ is a sequence of $k$ distinct vertices $p_1, p_2, \ldots, p_k$, such that $\{p_i, p_{i+1}\} \in E(G)$ for each $1 \leq i < j$. A cycle of length $k \leq 3$ is defined similarly, with the additional constraint that $\{p_k, p_1\}$ is in $E(G)$.

Two vertices $x$ and $y$ are connected if there is a path that connects them, i.e., such that $p_1 = x$ and $p_k = y$. The length of the shortest such path is the distance between $x$ and $y$, and a graph is connected if all pairs of vertices are connected. Furthermore, a graph is called acyclic if it does not contain any cycle, and cyclic otherwise.

A tree is an acyclic connected graph, and it is rooted when one vertex is named the root. For a rooted tree $T$ and two vertices $u$ and $v$ of $T$, we call $v$ an ancestor of $u$, and $u$ a descendant of $v$, if $v$ is on the unique path from the root to $u$; $u$ and $v$ are incomparable if $v$ is neither an ancestor nor descendant of $u$. A node that has no descendants is called a leaf.

A clique of a graph $G$ is a subset $C$ of $G$ that induces a complete graph, and a maximal clique is a clique $C$ of $G$ such that $C \cup \{x\}$ is not a clique for any $x \in V(G) \setminus C$. Depending on the context, $C$ may refer to its set of vertices or the induced subgraph $G[C]$. The size of a clique is given by the amount of vertices in it. We denote by $\omega(G)$, or simply $q$, the largest size of a clique in $G$.

A graph $G$ is $d$-degenerate if for any induced subgraph $H$ in $G$, $H$ has a vertex whose degree is at most $d$. The degeneracy of a graph is the minimum value $d$ for which the graph is $d$-degenerate. A degeneracy ordering of a graph $G$ of degeneracy $d$ is a sequence $v_1, v_2, \ldots, v_n$ of its vertex set such that the degree of each $v_i$ in $G[\{v_i, \ldots, v_n\}]$ is at most $d$: a $d$-degenerate graph always admits a degeneracy ordering w.r.t. $d$. A property of this ordering which will prove useful several times in this thesis is that it implies $|N^v_G(v)| \leq d$.

Note that $q - 1 \leq d \leq \Delta \leq n - 1 \leq m$.

Given a total ordering on the vertices in $V$, represented by increasing labels $v_1, \ldots, v_n$, the associated lexicographic ordering on $2^V$, denoted by $\preceq$, is such that $A \preceq B$ if $A$ contains the lexicographically smallest element not in common, i.e., $\min((A \cup B) \setminus (B \cap A)) \in A$. 12
Other more specific concepts which are only required in some sections of this thesis will be defined when necessary. The basic and more used notation is summarized in Table 2.1.

<table>
<thead>
<tr>
<th>symbol</th>
<th>name</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V(G))</td>
<td>Vertices</td>
<td>Set of vertices in (G)</td>
</tr>
<tr>
<td>(E(G))</td>
<td>Edges</td>
<td>Set of edges in (G)</td>
</tr>
<tr>
<td>(n)</td>
<td>Cardinality</td>
<td>Number of vertices in (G) (also (</td>
</tr>
<tr>
<td>(m)</td>
<td>Size</td>
<td>Number of edges in (G) (also (</td>
</tr>
<tr>
<td>(q)</td>
<td>Clique number</td>
<td>Number of vertices in the largest clique of (G) (also (\omega(G)))</td>
</tr>
<tr>
<td>(\Delta)</td>
<td>Maximum degree</td>
<td>Highest number of edges incident to a vertex in (G) (also (\Delta(G)))</td>
</tr>
<tr>
<td>(d)</td>
<td>Degeneracy</td>
<td>Measure of sparsity, small on sparse graphs (see Section 2.5.3)</td>
</tr>
<tr>
<td>(N(v))</td>
<td>Neighborhood of (v)</td>
<td>({u : {u,v} \in E(G)})</td>
</tr>
<tr>
<td>(N_&gt;(v))</td>
<td>Forward neighborhood of (v)</td>
<td>({u &gt; v : {u,v} \in E(G)})</td>
</tr>
</tbody>
</table>

Table 2.1: Glossary of the most common graph terms encountered in this thesis

2.1.2 Set systems

Another formalism that we will use in this thesis is that of set systems. Set systems are a general framework that can be used to formalize many enumeration problems, including graph enumeration.

A set system \((\mathcal{U}, \mathcal{F})\) is a unifying model which can be used to formulate listing problems on graphs by specifying a ground set \(\mathcal{U}\) of objects (e.g., the vertices of the graph) and indicating which combination of these objects meets the wanted property, expressed as the collection \(\mathcal{F} \subseteq 2^{\mathcal{U}}\) of “good” subsets that fulfill the property \([LLRK80]\) (By convention, it is always assumed that \(\emptyset \in \mathcal{F}\)). This formalism can highlight the fundamental structure of the problem at hand and help recognize similarities between problems.

Consider for instance one of the main problems addressed in this thesis, that of listing the maximal cliques in a graph \(G = (V,E)\).

We can see this problem as defined on a ground set \(\mathcal{U} = V\), where each solution (clique) is represented as a set of vertices \(X \subseteq \mathcal{U}\) that satisfies the condition \(X \in \mathcal{F}\): this condition, i.e., being a clique, corresponds to say that for each distinct pair \(u,v \in X\) we have \(\{u,v\} \in E\). This
means that \( \mathcal{F} \) can be defined algorithmically, i.e., we never actually store the \( \mathcal{F} \) set but we are able to check membership to it by checking some (usually simple) property of \( X \). This concept is extremely important: the content of the \( \mathcal{F} \) set is not computed a priori, as its computation time and space usage could both be exponential. While \( \mathcal{F} \) is referred to as a set in the commonly used terminology, e.g., by writing \( X \in \mathcal{F} \), what we really mean is that \( X \) satisfies the property identified by \( \mathcal{F} \). Every time we ask whether \( X \in \mathcal{F} \) what we really do is run \( X \) through an algorithmic procedure which tells us the answer (e.g., we can check whether \( X \) is a clique by checking that all vertices in \( X \) are pairwise adjacent).

For this reason, in the remainder of the paper we will use the term “property” as a synonym of set system, meaning the set system induced by the given property (e.g., being a clique) in a graph. Moreover, we assume in the rest of the paper that \( \mathcal{F} \) is given algorithmically, i.e., we can check whether \( X \in \mathcal{F} \) in time \( \mathcal{O}_T = poly(|U|) \) and space \( \mathcal{O}_S = \Omega(q) \), where \( q \) is the maximum size among the sets \( X \in \mathcal{F} \).

Set systems are often characterized by their accessibility, a term which refers to how easy it is to modify a solution by adding or removing elements.

**Independence Set Systems (ISS).** This class of set systems/properties corresponds to set systems \( (U, \mathcal{F}) \) for which \( X \in \mathcal{F} \) implies that every \( Y \subseteq X \) is also in \( \mathcal{F} \). In other words, a subset of a solution is always a solution. For this reason properties associated with these set systems are also known as hereditary properties (they are “inherited” by subsets). This class contains several well known properties, such as cliques, independent sets, k-plexes and forests.

**Connected Hereditary Properties (CH).** This class has been defined for graph properties by Cohen et al. [CKS08]. A property is connected hereditary if \( X \in \mathcal{F} \) implies that every connected \( Y \subseteq X \) is also in \( \mathcal{F} \), i.e., connected subsets of solutions are solutions. Well known properties in this class are connected k-plexes, trees (i.e., connected forests), and in general every hereditary property with the addition of a connectivity constraint. It should be noted that while most properties in CH are not in ISS, some can be: example, every subset of a clique happens to be a connected subset, thus cliques are in both CH and ISS.

**Strongly Accessible Set Systems (SA).** This class corresponds to properties which allow the following property: \( X \in \mathcal{F}, Y \in \mathcal{F}, \) and \( X \subset Y \) imply that \( \exists z \in Y \setminus X : X \cup \{z\} \in \mathcal{F} \). While not intuitive in this form, the property corresponds to a very basic principle. In other words, every solution \( X \) that is non-maximal, i.e., contained in a larger one \( Y \), can be extended into a larger solution that contains another element (of \( Y \)). This is especially significant when dealing with maximal solutions, as this is the most generic set system which guarantees that we can determine in polynomial time whether a solution is maximal: if for every \( x \in U \setminus X \) we have that \( (X \cup \{x\}) \notin \mathcal{F} \), then \( X \) must indeed be maximal.

**Accessible Set Systems (WA).** This class, also known as Weakly Accessible Set Systems, corresponds to properties which allow an even weaker property: given a non empty \( X \in \mathcal{F} \), \( \exists y \in X : X \setminus \{y\} \in \mathcal{F} \). An example of weak accessible property are dense subgraphs, i.e., subgraphs whose density is at least a certain value [Uno08].

Figure [2.1] gives a visual representation of the relationship among these set systems.
Figure 2.1: Hierarchy of set systems/properties. For the definition of Commutable set systems see Chapter 5.

2.2 Enumeration and efficiency

In this section we give a formal definition of what we call enumeration problem, and discuss more in detail the meaning of efficient enumeration.

2.2.1 Definition

We recall from Section 2.1.2 that a property is defined as the set \( \mathcal{F} \) of all elements which satisfy said property, and that we assume to have an algorithmic procedure which can check membership to \( \mathcal{F} \).

Problem 2.1 (Enumeration problem). Given a set system \((U, \mathcal{F})\), where \( U \) is a ground set and \( \mathcal{F} \) a property, return all \( X \subseteq U \) such that \( X \in \mathcal{F} \).

A large amount of enumeration problems can be modeled this way, and in particular all enumeration problems for induced and edge subgraphs, by setting \( U = V(G) \) or \( U = E(G) \) respectively, as long as membership to \( \mathcal{F} \) can be checked.

This includes well known properties such as cliques, dense subgraphs, (induced) paths, (induced) acyclic subgraphs, (induced) trees or spanning trees.

Furthermore, we remark that this formalism may be applied to a different kind of enumeration problem, considered in Chapter 7 of this thesis, that is the enumeration of certain graph orientations: given an undirected graph \( G \), an orientation \( \overrightarrow{G} \) of \( G \) is any directed graph that may be obtained by assigning a direction to each edge of \( G \), i.e., replacing the undirected edge \( \{x, y\} \) with either the arc \( (x, y) \) or \( (y, x) \)\(^2\). While \( \overrightarrow{G} \) is not a subgraph of \( G \), consider the graph \( G^* \) obtained by replacing every edge \( \{x, y\} \) of \( G \) with both arcs \( (x, y) \) and \( (y, x) \): every orientation \( \overrightarrow{G} \) of \( G \) is a specific edge subgraph of \( G^* \) obtained by removing exactly one among each pair of symmetric edges. As removing one of this edges immediately implies that the other one in the pair should be used, this is conceptually equivalent to giving a direction to the undirected edge, and thus the problem will have a similar structure to those modeled with set systems.

The formalism of set systems can also be applied to the enumeration of frequent structures, such as frequent subgraphs. This in general requires different techniques w.r.t. subgraph enumeration, as the solutions are not proper subgraphs of the given graph \( G \), but structures which are isomorphic to many subgraphs of \( G \). More detail can be found in Section 2.3.6.

\(^2\)This implies that \( \overrightarrow{G} \) may not contain two symmetric edges, and one may recall that directed graphs with no symmetric edges are indeed called oriented graphs.
When dealing with enumeration problems, a common issue is the size of the output, which can be exponential. Removing redundant and non significant solutions from the output is an important task. For this reason, many algorithms focus on enumerating only maximal solution, that is solutions that are not subsets of any other (larger) solution. The definition changes as follows:

**Problem 2.2** (Enumeration problem with maximality). Given a set system \((\mathcal{U}, \mathcal{F})\), where \(\mathcal{U}\) is a ground set and \(\mathcal{F}\) a property, return all \(X \subseteq \mathcal{U}\) such that \(X \in \mathcal{F}\) and there is no \(Y \in \mathcal{F}\) such that \(X \subset Y\).

It is natural to assume that if a solution \(Y\) includes a smaller one \(X\), then \(X\) does not contain more information than \(Y\), thus \(X\) does not need to be returned to the user. For this reason, the algorithms proposed in this thesis will focus on finding only maximal solutions.

It is also straightforward to see that the result of the enumeration problem with a maximality constraint is a subset of the generic enumeration one, and thus an algorithm which solves the former may solve the latter (assuming that maximal solutions can be recognized). However, as we will discuss in Section 2.4, designing efficient algorithms for finding maximal solutions is in general more difficult.

2.2.2 Space efficiency

An algorithm is said to use linear space if its space requirement is \(O(m)\), polynomial space if it is \(poly(m, n)\) (e.g., \(O(n^3)\), \(O(m \cdot n)\)), and exponential space otherwise (e.g., \(O(2^n)\)).

When dealing with large graphs, a careful and parsimonious management of the space usage is essential. For example, storing a graph as an adjacency matrix yields the best time-wise performance for checking adjacency, and is a popular choice in traditional graph algorithms. In large networks, however, the associated \(O(n^2)\) space requirement is often not sustainable, and data structure that use just linear space \((O(m))\) should be used instead. If an algorithm is meant for large networks, using more than linear space should be very carefully evaluated.

In this thesis, we shall always consider that the input graph is loaded in main memory. Indeed, external memory approaches exist, which store data in the slower, larger, secondary memory, but due to the linked nature of graphs these may suffer from random and frequent access to this slower memory. For this reason, we will consider the space that is used by the algorithm on top of the read-only input data, which we call additional space.

Algorithms which use small additional space have several benefits:

Larger graphs. By minimizing the additional space, we increase the space that can be dedicated to storing the input graph. Consider an algorithm \(A\) which uses \(O(m)\) additional space, and \(B\) which uses just \(O(1)\). Both algorithms use \(O(m)\) space in total as they store the input graph. However, while \(B\) will dedicate approximately all of the available memory to the input graph, \(A\) may be using, say, 50% of the total space to store the additional data structures and will only dedicate 50% to the input data. This makes \(A\) able to process graphs twice as large as those that \(B\) can process on the same hardware.

Parallelism. As the input data is read-only, it may be shared by multiple algorithms running concurrently. If an algorithm is suitable to be made parallel (as are, e.g., those in Chapters 3, 4 and 5), the extra cost for each additional thread of the algorithm on top of the first one will be just the additional space, which is small.

---

3Furthermore note that graph orientations always involve all edges of a graph, thus non-maximal solutions do not exist in that case.

4Or even more, as the \(O(m)\) notation masks constant factors.
Exploiting low-level cache. Modern CPUs have multiple cores, and each core has access to a fast, but small, private cache (usually few hundreds of kylobytes), as well as a slower but larger shared cache (few megabytes). If the additional space is small enough to fit in the private cache, this can reduce the update times due to writing as the algorithm will not need to load different memory blocks from the shared cache, and the frequency of propagation of the write operations in upper levels of cache will decrease. Furthermore, this may create synergy if different threads are running on separate cores of the processor, as smaller working space means that more of the shared cache may be dedicated to storing shared, read-only parts of the input graph.

Some examples of the strategies involved to design space-efficient algorithms are shown in Section 2.5.3.

In particular, we say that an algorithm uses sublinear space if this is \( o(m) \) (little-o of \( m \)), or in other words, if the additional space is bounded by some function \( f(m) \) such that \( \lim_{m \to \infty} \frac{m}{f(m)} = 0 \). Furthermore, we say that an algorithm uses minimal space if this is \( O(s) \), where \( s \) is the size of the largest solution to the problem. Note that achieving minimal space in some cases implies a sublinear additional space usage (e.g., Chapter 3), while in others it may not (e.g., Chapter 4), depending on the problem at hand.

2.2.3 Time efficiency

It is important to remark that enumeration is not to be confused with brute-force approaches: while it is possible, says, to enumerate every solution of a set system \((U, F)\) by simply iterating over the \( 2^{|U|} \) possible subsets of \( U \), the time (and possibly space usage) involved is unsustainable unless \( U \) is very small.

While efficient enumeration algorithms may still require exponential time to run, they employ techniques which aim at cutting the search space as much as possible, to try and minimize the amount of non necessary operations. This concept of efficiency leads to two opposite types of complexity analysis: input sensitive and output sensitive analysis.

While any algorithm can be analyzed under both types of analysis, designing algorithm meant to have good input sensitive bounds typically involves different techniques from those used to obtain good output-sensitive ones. For this reason we will refer to the earlier as input sensitive algorithms and the latter as output-sensitive algorithms.

**input sensitive algorithms.** Input sensitive efficiency aims at bounding the total running time of an algorithm with respect to the size of the input. The cost of such algorithms is commonly given in a form similar to \( O(k^n) \), where \( k \) is a constant.\(^5\)

An important property of input sensitive algorithms is that their complexity can be linked directly to the maximum number of possible solutions. For example, Tomita et al.’s algorithm for enumerating maximal cliques [TTT06] has a running time of \( O(3^{n/3}) \), which implies that no more than \( O(3^{n/3}) \) cliques may exist. At the same time, this makes the algorithm worst-case optimal as the highest number of maximal cliques a graph may have is indeed \( 3^{n/3} \) [MM65]. Such bounds may be also parameterized: Eppstein et al. [ELS13] for example, improved the bound for the number of maximal cliques on sparse graphs, by showing that \( d \)-degenerate graphs may have up to \( (n - d)3^{d/3} \) maximal cliques, using an algorithm which enumerates them in \( O(d(n - d)3^{d/3}) \).

More in general, an algorithm is called a *Fixed Parameter Tractable* (FPT) algorithm for some parameter \( p \) if its running time is \( f(p) \cdot n^k \).

\(^5\)Unless the algorithm is in \( \text{ptime} \), in which case it is in a form similar to \( O(n^k) \).
This type of analysis is particularly convenient for algorithms such as optimization and counting ones, whose output size is bounded and independent from the number of solutions that exist.

**Output-sensitive algorithms.** Although many input-sensitive enumeration algorithms exist (see, e.g., [FK10]), when dealing with an enumeration problem one may want guarantees on the running time with respect to the size of both the input and the output.

For example, when performing a database query, such techniques allow to explore the data with well designed fast queries that retrieve only few elements, while we may be willing to wait a longer time for the result of larger queries.

Furthermore, for many enumeration problems we cannot give bounds polynomial in the size of the input as the number of solution may be exponential. On the other hand, we can sometimes give bounds polynomial in the size of the output.

Running time of output-sensitive algorithms usually includes the number of solutions that are reported, which we refer to as \( \alpha \) in the remainder of the thesis, as well as the size of the input, which we refer to as \( N \).

We refer to the complexity classes introduced by Johnson, Yannakakis and Papadimitriou [JYP88], and Valiant [Val79]. We will refer to algorithms as output-sensitive if their complexity falls into one of the following classes.

**PTT** Quoting Johnson et al. [JYP88], “The least that we could ask [an enumeration algorithm] is that the time required to output all configurations be bounded by a polynomial in \( N \) (the size of the input) and \( \alpha \) (the number of configurations).”\(^6\) This class is referred to as *Polynomial Total Time* (PTT), or sometimes as *output polynomial* [Pap97]. More formally, the running time of algorithms in this class is bounded by \( O(\alpha^h \cdot N^k) \), where \( h \) and \( k \) are two constants. Such algorithms can be efficient for problems with a limited number of solution, but their running time may become unfeasible when \( \alpha \) approaches exponential, or even superlinear, size with respect to \( N \).

**INC-P** A slight refinement of PTT is the class called *Incrementally Polynomial* (INC-P). An algorithm runs in incrementally polynomial time if the time required to output the \( i \)-th solution is bounded by a polynomial in \( i \) and \( N \), i.e., \( O(i^h \cdot N^k) \). Note that the total running time of INC-P algorithms can be the same as that of PTT ones; however, INC-P algorithms will immediately start outputting solutions, whereas PTT ones may output all solutions towards the end of the execution.

**P-Delay** When running an enumeration algorithm, one may set up a pipeline which processes the solutions as soon as they are found. For this reason, it may be desired that solutions are produced at a constant rate, so that the pipeline does not suffer from starvation. This class is called *Polynomial Time Delay* (P-Delay), and it guarantees that the delay, i.e., the time elapsed between the output of the \( i \)-th and the \((i + 1)\)-th solution, is bounded by a polynomial in \( N \). The delay has been already used by many, e.g., [CYQ13, CN85, CR15, JYP88, MU04, TIAS77], as a worst-case measure. The cost of these algorithms is usually given by just stating their delay, which will be bounded by \( O(N^k) \). Note that the total running time will be \( O(\alpha \cdot N^k) \). To give a more rigorous definition, it should be imagined that a dummy solution is output just before the beginning of the computation and just after the end to account for the time elapsed before

---

6Where “configurations” refers to the solutions to the problem, and their number is referred to as \( C \) rather than \( \alpha \).
the output of the first solution and after the last. However, these are sometimes considered separately as pre-processing/post-processing costs.

**P-Enum** Closely related to P-Delay, the less used but nonetheless interesting class P-Enumerable has been defined by Leslie G. Valiant [Val79]: this refers to algorithms with a running time of $O(\alpha \cdot N^k)$. The class corresponds to PTT algorithms ($O(\alpha^h \cdot N^k)$) for the case $h = 1$. The relationship with the P-Delay class comes from the two classes having the same bound on the total running time. P-Enum algorithms, however, do not give guarantees on the time elapsed between two outputs. Hence, rather than a polynomial delay, we say that such algorithms exhibit a polynomial amortized cost per solution. It should be noted that, for algorithms in P-Delay, the amortized cost per solution cannot be higher than the delay, but may very well be lower: an example is the algorithm proposed in Section 7.10 which runs in time $O(\alpha \cdot m)$ but has delay $O(m^2)$.

**Sos** Refining the definition of P-Enum, we may ask that the total running time of the algorithm is just as much as the time needed to output the solutions. We will refer to this class as Strongly Output Sensitive (sos)\(^7\). In other words, if an algorithm enumerates solutions of size, say, $\Theta(S)$, the algorithm is sos if its running time is bounded by $O(\alpha \cdot S)$. Some algorithm manage to achieve an even stronger form of bound: if $S = \{s_1, s_2, \ldots, s_\alpha\}$ is the set of solutions, the cost of these algorithms is just $\sum_{s_i \in S}(|s_i|)$, where $|s_i|$ is the size of the $i$-th solutions. This is sometimes referred to as optimal running time for an enumeration algorithm, as it is impossible to achieve a lower running time if we do output each solution found. Some examples are the algorithm for listing cycles and st-paths by Birmele et al. [BFG\(+13\)], and several of the algorithms shown in Chapter 7. Note that sos algorithms are in P-Enum, but may or may not be in P-Delay.

**Cat** Finally, there are some enumeration algorithms whose running time is bounded by the number of solutions, but not by their size. Some do not output the solutions found (e.g., if we use an enumeration algorithm for just counting), other output the result in a compressed form (e.g., TTT06), other efficiently iterate over solutions without returning them, in such a way that two consecutive solutions only differ slightly (e.g., using Grey Codes [Sav97]). The complexity of these algorithms can go even below the “optimal” bound of sos algorithms. Some of these algorithms fit under the class of Constant Amortized Time, described in Rus03. The cost of such algorithms is just $O(1)$, or constant, time per each solution found. In other words the complexity of an algorithm in this class is $O(\alpha)$. As the names suggest, the cost of these algorithms is usually obtained by amortized analysis, which essentially consists in smartly scapegoating the cost of operations onto the solutions. Some examples are algorithms found in Rus03, and those which use the push-out amortization technique Uno15.

Figure 2.2 gives a graphical representation of the classes discussed and their inclusion relationships, while their definitions are recalled in Table 2.2. For some classes this is trivial as their definition is obtained by adding constraint to that of their super-class (e.g., Cat is a subset of P-Enum). For others, this is less intuitive.

In order to give a better view of Figure 2.2 we give some artificial but simple examples which aim at clarifying some of the relationships. Consider an enumeration problem with $\alpha = x^N$ for some positive $x$, and an algorithm $A$ which finds and returns each solution in $O(1)$ time from the last one, but pauses for $O(x^{N/2})$ time before finding and returning the last solution. This algorithm is clearly in Cat, as its total running time is $O(x^N)$, and in Inc-P as the $i$-th

\(^7\)Sometimes this class is called simply output-sensitive, e.g., in FGR11. However, the term output-sensitive is also used more broadly, as in this thesis, to refer to all algorithms whose running time is influenced by the size of the output. We thus add the term strongly to strengthen the definition.
solution is always found within $O(i)$ time, including the last one (for which $i = x^N$). However the algorithm is not in P-Delay due to the long pause before the last solution is found. On the other hand, assume that the pause occurs between the output of the first and second solutions: then the algorithm would still be in CAT, but not in INC-P, since the second solution ($i = 2$) would be found after $O(x^{N/2})$ time, which is not bounded by $O(i^h \cdot N^k)$.

<table>
<thead>
<tr>
<th>symbol</th>
<th>name</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTT</td>
<td>Polynomial Total Time</td>
<td>$O(\alpha^h \cdot N^k)$</td>
</tr>
<tr>
<td>INC-P</td>
<td>Incrementally Polynomial</td>
<td>$i$-th solution found in $O(i^h \cdot N^k)$</td>
</tr>
<tr>
<td>P-ENUM</td>
<td>P-Enumerable</td>
<td>$O(\alpha \cdot N^k)$</td>
</tr>
<tr>
<td>P-Delay</td>
<td>Polynomial Delay</td>
<td>$i$-th solution found in $O(N^k)$ after the $(i-1)$-th</td>
</tr>
<tr>
<td>SOS</td>
<td>Strongly Output-Sensitive</td>
<td>$O(\alpha \cdot S)$</td>
</tr>
<tr>
<td>CAT</td>
<td>Constant Amortized Time</td>
<td>$O(\alpha)$</td>
</tr>
</tbody>
</table>

Table 2.2: Glossary of complexity classes for output-sensitive algorithms. $N$: size of the input. $\alpha$: number of solutions. $S$: tight bound on size of the largest solution. $h, k$: positive constants.

The first part of this thesis is focused on designing efficient algorithms meant for large real-world networks. For this reason, we will only propose algorithms with complexity bounds at least in P-Enum, with additional focus to bound the delay and achieve Strongly Output-Sensitive bounds whenever possible.

In particular, all algorithms proposed in Chapters 3, 4, 8, 7, 6 are in P-Delay.
suitable conditions, this also holds for those in Chapter 5. Furthermore, two of the algorithms in Chapter 7 achieve SOS bounds, namely those presented in Sections 7.5 and 7.10.

As for space usage, the algorithm in Chapter 3 uses space proportional to the size of the output, which can be exponential unless (as often happens in real-world networks) the graph is suitably sparse. All the remaining approaches above use polynomial space. In particular, those in Chapter 7 use linear space, and those in Chapters 3, 4, and 5 use minimal additional space, that is, only the space required to store one solution, on top of the input graph. For the algorithm in Chapter 3, this additional space is also proved to be sublinear.

The second part of the thesis is focused on problems which do not concern just enumeration, thus it is difficult to bound the complexity with respect to the size of the approaches’ final results. However, we remark that all these approaches are build upon efficient algorithms, which all have polynomial delay, except for the algorithm in Chapter 10 which runs in polynomial time.

2.3 Main techniques

2.3.1 Recursion and iteration

It is assumed that the reader already had a basic knowledge of recursive and iterative algorithms. In particular, we will call recursion tree the computational structure associated with the nested calls of a recursive algorithm, and recursive node a recursive call. For any recursive node \( X \), the nested recursive calls correspond to the children nodes of \( X \) in the recursion tree, while the call to which the computation is returned after \( X \) is completed is the parent node of \( X \) in the tree. This representation is often convenient for giving complexity bounds of both time and space to recursive algorithms. Indeed, in most algorithms presented we will assume that each leaf node of the recursion tree will correspond to the output of a solution: when this is true, the delay of the recursive algorithms is bounded by the sum of the costs of the nodes along a root-to-leaf path, and the space usage by the information stored by such nodes.

In order to improve the space complexity, we will in some cases (e.g., Chapter 3) transform a recursive algorithm in an equivalent iterative one, i.e., that does not involve nested calls but simply modifies the value of some static variables to “simulate” them. While it is usual for iterative algorithms to keep the state of previous simulated calls using a stack, a key feature of our iterative algorithm is that they will use suitably defined rules to avoid storing this potentially large stack. While this changes the implementation details, the structure of the algorithm, i.e., the order in which operations are performed, will remain essentially the same, thus we will sometimes use terminology meant for recursive algorithms (such as recursive node or parent node) for iterative versions too.

2.3.2 Recursion trees

Looking at an algorithm’s recursion tree is often a powerful technique to give complexity bounds to the algorithm. For example, the Measure and Conquer technique [FK10] can bound the total running time of an algorithm by computing the number of leaves in the recursion tree. In output-sensitive analysis, rather than the tree’s size, we are interested in its topology.

In the following, let \( h \) be the height of the considered recursion tree, i.e., the maximum distance between a leaf and the root, and let \( X \) be an upper bound on the computational cost of a recursive node. We assume \( X \) to be polynomial in the size of the input \( N \).

We describe three commonly found types of topology and their properties with respect to output-sensitive analysis. Due to their properties, we name them after the main characters from the famous spaghetti western by Sergio Leone, “The Good, the Bad and the Ugly”.

21
A graphical representation of some recursion trees is given in Figure 2.3. Nodes marked in green correspond to recursive calls in which a solution is output, and unless stated otherwise the recursive calls corresponding to the children of a node are executed in order from left to right. Note that the execution order of the nodes corresponds to a depth-first search (DFS) of the recursion tree.

The Bad. Bad trees refer to those that contain recursion nodes which are leaves but will not output a solution. These nodes, which we will sometimes call dead leaves, put a significant burden on the complexity analysis, as their cost is hard to link to that of any solution.

While algorithms that exhibit this type of recursion tree may still be fast in practice, they usually do not offer any guarantee from an output-sensitive perspective.

An example of bad tree is that induced by the Bron-Kerbosch algorithm for maximal clique enumeration [BK73], which can be fast in practice but has no guarantee on the cost per solution. A graphical representation can be seen in Figure 2.3 (left).

The Ugly. Ugly trees are those characterized by the absence of dead leaves. In other words, every leaf in the recursion tree will output a solution. It is not relevant whether other internal nodes (nodes which are not leaves) output solutions as well.

These trees do offer output-sensitive bounds, as long as $h$ is polynomial in $N$: indeed, for each leaf of the tree, consider the nodes on the path that connects it to the root. This path is made of at most $h + 1$ nodes, thus the total computational cost of the nodes will be $O(h \cdot X)$.

It is straightforward to see that every internal node will be on at least one root-to-leaf path, and thus by summing the cost of all root-to-leaf paths we will obtain an upper bound of the total cost of the algorithm, that is $O(\alpha \cdot h \cdot X)$ time, or in other words $O(h \cdot X)$ time per solution.

Algorithms with an ugly tree are in $\text{P-Enum}$. Furthermore, if the algorithm is a recursive one, its execution will follow a DFS of the tree. As the distance between two leaves in such a traversal is bounded by $2 \cdot h$, the algorithm will also be in $\text{P-Delay}$ with a delay of $O(h \cdot X)$.

While this type of tree can already provide output-sensitive algorithms, it is called ugly as this type of analysis will count the cost of the same recursive nodes multiple times if they have multiple descendant leaves, thus providing a bound that may not be tight. Furthermore, the cost per solution will be dependent from the height of the tree $h$, making it hard to obtain more efficient bounds (e.g., sos or cat).

An example of algorithm which exhibits an ugly recursion tree is the algorithm for listing Single Source Acyclic Orientations shown in Section 7.4. A graphical representation can be seen in Figure 2.3 (center).

The Good. Finally, we say that a recursion tree is “Good” when two conditions apply: firstly, as for ugly trees, each leaf outputs a solution. Secondly, each internal node must have at least two children.

Good trees have an important property, that is, the number of internal nodes is smaller than the number of leaves. This is easily proven by induction: remove an internal node $x$ with only leaves as children (one such node always exists), and replace it with one of its children (removing the other children of $x$ too); this new tree contains 1 fewer internal node and at least 1 fewer leaf. By iterating this process we remove at most as many internal nodes as leaves, and we are left with just one leaf in the end.

The leaves are at most $\alpha$ as every leaf corresponds to a solution, thus the total number of nodes is at most $2 \cdot \alpha$. This means that the total running time of the algorithm is bounded by $2 \cdot \alpha \cdot X = O(\alpha \cdot X)$, or $O(X)$ time per solution. This is a direct improvement of the cost of ugly trees, as it does not feature the height $h$ in the amortized cost per solution.
Figure 2.3: Different types of recursion trees. From left to right: a bad, ugly, and good tree. Recursive nodes in which a solution is output are marked in green.

It should be noted that an ugly recursion tree can always be turned into a good one by conceptually compressing chains of unary nodes (nodes with only one child) into a single node: this way every node will have two children and the tree will be good. However, the cost of a recursion node will increase by up to a factor $h$ due to the compression, meaning that the cost of $O(h \cdot X)$ per solution remains unchanged.

An example of algorithm which exhibits a Good recursion tree is the algorithm for listing Strong Orientations shown in Section 7.10. A graphical representation can be seen in Figure 2.3 (right).

As for the delay, the same conditions as for ugly trees apply ($O(h \cdot X)$): note for example the time needed to find the first solution in the computational tree in Figure 2.3 (right). Sometimes, however, additional techniques may be applied to reduce the delay. An example is the algorithm for listing Acyclic Orientations in Section 7.5.

Better than Good. Finally, one more case should be considered: in some algorithms, each recursive node will output a solution. We do not give a name to this case as the topology of the recursion tree becomes irrelevant: the cost per solution will always be bounded by the cost of a recursive node. An example is algorithms based on the framework of Reverse Search [AP96], that will be introduced in Section 2.3.4.

2.3.3 Binary partition

When it comes to algorithm design, and enumeration in particular, a powerful and widely used yet simple technique is that of binary partition, which consists in recursively dividing the set of solutions into two (or sometimes more) non-overlapping partitions: a classical strategy is to (i) first find all solutions which contain a given element $v$, then (ii) finding those that do not contain $v$, as detailed in Algorithm 1.

This very simple structure is enough to enumerate any induced subgraph that respects the
Algorithm 1: Structure of a binary partition algorithm for graphs

Input:
A graph \(G = (V(G), E(G))\), and a property \(\mathcal{F}\)

Output:
All \(H \in \mathcal{F}\)

1. Binary-Partition \((G, \mathcal{F}, \emptyset, \emptyset)\)

2. Function Binary-Partition \((G, \mathcal{F}, S, X)\)

   3. if \(S \in \mathcal{F}\) then output \(S\)

   4. \(v \leftarrow\) arbitrary vertex in \(V(G) \setminus (S \cup X)\)

   5. Binary-Partition \((G, \mathcal{F}, S \cup \{v\}, X)\) // Find solutions containing \(v\)

   6. Binary-Partition \((G, \mathcal{F}, S, X \cup \{v\})\) // Find solutions not containing \(v\)

property \(\mathcal{F}\), as long as we are able to check whether \(H \in \mathcal{F}\). Note that, by choosing edges rather than vertices, this same structure can also be used to enumerate edge subgraphs rather induced ones.

Note, however, that unless some halting conditions are applied, the associated computation tree is going to have dead leaves, i.e., recursive calls not leading to a solution. More in detail, if no halting condition is given at all, this procedure will iterate over all \(2^n\) possible subsets of \(V(G)\). Ideally, we want to halt the computation as soon as we are certain that no solution can be found: e.g., if enumerating trees, we can halt the computation as soon as a cycle appears in the solution.

This implicitly induces a question which is sometimes referred to as extension problem. Recalling the basic notation of properties and set systems from Section 2.1.2, the problem can be defined as follows.

**Problem 2.3** (Extension problem for \(\mathcal{F}\)). Given a graph \(G(V(G), E(G))\), a property \(\mathcal{P}\), and two disjoint sets of vertices \(S, X \subseteq V(G)\), is there a solution \(H \in \mathcal{F}\) such that \(S \subseteq H\) and \(X \cap H = \emptyset\)?

When using binary partition, this problem is the key to obtain output-sensitive algorithms: indeed, if we can answer the question (in polynomial time), we can skip the computation of children that will not lead to a solution. In other words, we can say we maintain a certificate of the existence of a solution in the subtree that we are exploring. This strategy will thus never incur in dead leaves, and the recursion tree obtained this way will be either an ugly or good one, leading to polynomial delay.

A well known algorithm which uses this strategy without a certificate, as mentioned in Section 2.3.2, is the Bron-Kerbosch algorithm \([BK73]\), which is fast in practice but suffers from dead leaves.\(^8\) One that does use a certificate is the algorithm for listing cycles and st-paths by Birmelé et al. \([BFG+13]\), which manages to achieve Strongly Output-sensitive bounds. More examples are the algorithms proposed in Chapter 7 which are all based on binary partition.

When dealing with the enumeration of maximal solutions, however, the corresponding extension problem (is there a maximal solution \(H \in \mathcal{F}\) such that...?) is often \(\text{NP}\)-Complete, and thus algorithms (e.g., \([BK73]\)) often do not guarantee output-sensitive bounds.

### 2.3.4 Reverse search

Reverse search by Avis and Fukuda \([AF96]\) is a powerful technique which can be used for enumerating maximal as well as non maximal solutions, and can guarantee polynomial delay even in some problems for which the extension problem is \(\text{NP}\)-complete.

Its general schema uses problem-specific operations which allow jumping from a solution to another. These operations define a graph-like structure as that represented in Figure 2.4 (left),\(^8\) The algorithm divides the search space in more than two partitions each time, but the basic concept is the same.
where $S_1, S_2, \ldots, S_7$ represent (maximal) solutions of a set system $(\mathcal{U}, \mathcal{F})$, and each arrow a possible way to \textit{jump} from one solution to another with the given operations.

The key feature of reverse search is \textit{pruning} this structure to obtain a tree-like one, like that in Figure 2.4 (right): a tree-like structure is easy to traverse without incurring multiple times in the same vertices, whereas a more complex graph-like structure requires keeping in memory all the visited vertices.

In this tree-like structure, we call a solution a \textit{root} if it has in-degree 0, \textit{i.e.}, if it is not possible to reach this solution by using the jumping operation from any other one.

In order for the algorithm to find all solutions, the structure defined must have the following properties:

**Definition 2.1** (Reverse search solution tree). The tree induced by a traditional reverse search algorithm among the solutions (or maximal solutions) of a set system $(\mathcal{U}, \mathcal{F})$ has the following properties:

\textbf{parent} Every solution $S_i$ that is not a root has in-degree exactly 1 (we call parent of $S_i$ the solution on the other side of this unique in-edge).

\textbf{acyclic} The structure contains no cycles.

\textbf{roots} It is possible to find the unique root, which has in-degree 0, in polynomial time.

\textbf{children} Given $S_i$, it is possible to compute the set of children of $S_j$, \textit{i.e.}, $\{S_j : S_i \text{ is the parent of } S_j\}$.

Note how the first two condition define the structure in Figure 2.4 (right), while the other two allow us to navigate it. We can easily show that this is enough to find all solutions: let $\text{parent}(S_i)$ be the parent of $S_i$ (or $\text{null}$ if $S_i$ is the root). If we apply this parent function iteratively to an arbitrary solution $S_i$ to find its parent, then to its parent and so on, we must eventually find the root, since the number of solutions is finite, and there are no cycles. This means that every solution has a path from the root leading to itself. Thus if we can find the root, and we can find the \textit{children} of each given solution, we will eventually be able to find every solution.

The structure of the resulting algorithm is summarized in Algorithm 2.

| Algorithm 2: Structure of a reverse search algorithm |
|----------------------|----------------------|
| **Input** | Reverse searchable $(\mathcal{U}, \mathcal{F})$ |
| **Output** | All maximal $X \in \mathcal{F}$ |
| $R \leftarrow$ the root solution. SPAWN($R$) |
| **Function** SPAWN($X$) |
| output $X$ |
| \textbf{foreach} $S \in \text{children}(X)$ \textbf{do} |
| SPAWN($S$) |

Some example of reverse search algorithms are those proposed by Avis and Fukuda in their seminal paper [AF96], and the algorithm by Makino and Uno for enumerating maximal cliques and maximal bipartite cliques [MU04].

Note that the reverse searchable structure is not always explicitly defined, but is sometimes embedded in the computational structure of the algorithms. Furthermore, reverse search algorithms often define a single root, to obtain a simpler tree-like forest rather than a forest-like one.
The advantage of reverse search is that it guarantees polynomial delay even for the enumeration of maximal solutions, as long as the children of a solution can be found in polynomial time. The downside is that understanding if a set system is reverse searchable, i.e., defining a suitable structure, is often far from trivial. In Chapter 5 we attempt at giving some general techniques for applying reverse search to a range of problems.

2.3.5 General algorithms for connected hereditary properties

Cohen et al. [CKS08] proposed an algorithmic framework for enumerating maximal solution of hereditary and connected hereditary graph properties.

The framework uses techniques conceptually very similar to those of reverse search, in that it also performs “jumps” between solutions: these operations are defined using the restricted problem: given a graph $G = (V(G), E(G))$, a solution $X$ to the problem at hand, and a vertex $x \in V \setminus X$, the restricted problem consists in enumerating all solutions to the problem in the smaller graph $G[X \cup \{x\}]$.

Given a solution $X$, and a subroutine which can solve the restricted problem, the framework is able to generate new solutions which are maximal in $G$. This corresponds to jumping from $X$ to other solutions, which defines a graph-like structure as in Figure 2.4 (left), which is guaranteed to connect all solutions. How to solve the restricted problem is delegated to the user of the framework.

Differently from reverse search, however, Cohen et al. do not focus on building a tree-like structure, but simply visit this graph in a stateful manner, i.e., storing every solution already

---

9By simply adding elements from $G$ to the solution of the restricted problem in a greedy fashion until it is maximal in $G$. 
found in a “result set” so that the same solution is not visited twice.

The clear downside of this strategy is that storing such set requires exponential space, as the number of maximal solutions can indeed be exponential. Cohen et al. do provide a technique to improve the space usage to polynomial for hereditary properties (which essentially induces a tree-like structure like that of reverse search), but achieving polynomial space for connected hereditary properties is left as an open problem. Due perhaps to its simplicity, the framework appears to work well in practice, as it was subsequently used to produce a fast algorithm for enumerating $k$-plexes and connected $k$-plexes [BCK15].

Remarkably, the concept of restricted problem, although without a name, was also used by Lawler in [LLRK80], regarding the enumeration of maximal solutions of Independence Set Systems (which, on graphs, correspond to hereditary properties). This underlines how well the restricted problem encapsulates the hardness of the general problem.

In some cases the restricted problem will have just a constant number of solutions: it is easy to see how if $X$ is a maximal clique, $G[X \cup \{x\}]$ contains exactly two maximal cliques, $X$ and $\{x\} \cup (N_G(x) \cap X)$. On the other hand, there are cases in which the restricted problem will have an exponential amount of solutions: one example is the reduction given by Lawler [LLRK80] which models assignments that satisfy a SAT formula as maximal solutions of an Independence Set System.

Cohen et al. show that, for hereditary and connected hereditary properties, if the restricted can be solved in polynomial time (resp. incremental polynomial time, polynomial total time) then the general problem can be solved in polynomial delay (resp. incremental polynomial time, polynomial total time). On the other hand, Lawler shows that there are some problems even within Independence Set Systems (i.e., corresponding to hereditary properties), which do not allow an algorithm with polynomial delay unless $\text{P} = \text{NP}$ (more details are given in Section 2.6).

### 2.3.6 General algorithms for closed sets

Other works have focused on systems more general than the ones studied by [CKS08], like strongly accessible set systems, but they used different requirements for the objects to look for. This is the case of the framework by Boley et al. [BHPW10] and Arimura and Uno [AU09]. In these works, a set system $(\mathcal{U}, \mathcal{F})$ is given together with a closure operator $\sigma : \mathcal{F} \rightarrow \mathcal{F}$, which satisfies the following properties: for all $X, Y \in \mathcal{F}$, $X \subseteq \sigma(X)$ (extensivity), $X \subseteq Y \Rightarrow \sigma(X) \subseteq \sigma(Y)$ (monotonicity), and $\sigma(X) = \sigma(\sigma(X))$ (idempotence). In such a context, a set $F \in \mathcal{F}$ is called closed if $\sigma(F) = F$ and these works aim at listing all the $F \in \mathcal{F}$ which are closed. In principle, the concept of closure is similar to that of maximality, in that both allow the algorithm to ignore many solutions that are not significant in that their information is included in some other more important one, thus improving the significance of the output and the total running time. We remark, however, that this is a different kind of problem as we require maximal solutions instead, for which monotonicity does not apply.

### 2.3.7 Enumeration by reduction

Finally, we remark that an enumeration algorithm may sometimes be obtained from another one. Given an instance of problem $A$, it is sometimes possible to produce an instance of a problem $B$ such that there is a one-to-one correspondence between the solutions of $B$ and those of $A$, as well as a way to reconstruct the latter from the earlier ones. A classical example are cliques and independent sets, where a transformation is obtained by simply taking the complement graph: indeed each (maximal) clique in a graph is a (maximal) independent set in its complement, and vice versa. Some more examples are given in Section 7.8 as part of the result of Chapter 7, where algorithms for listing several types of graph orientations are obtained by applying a single
enumeration algorithm on suitably tweaked problem instances. While effective in theory, we
remark that it may not always be the a practical choice: in Chapter 4, for example, we show
how using a clique enumeration algorithm yields a correct but not satisfactory algorithm for
independent sets, and then proceed to design a new one by using ad-hoc techniques.

2.4 Maximality

In this thesis, we are interested in listing just the maximal solutions, whenever this definition
can be applied. This aims at reducing the redundancy by omitting solutions which are fully
contained in others, based on the principle that they do not provide new information. While
this may not be the case for systems of closed sets (see section 2.3.6), where features such as
the frequency of a solution may be relevant, the value of this principle should be obvious for,
e.g., hereditary or connected hereditary properties, since non maximal solutions can be trivially
obtained from maximal ones.

This is of interest for many applications in literature (e.g. [CN85, TIAS77, JYP88, KLW96])
where instances can have a number of maximal solutions that is much smaller than
\(|F| \leq 2^{|U|}\). However, generating only maximal solutions usually makes the listing problem harder, as it
generally requires more sophisticated techniques: for example, we are not aware of any Constant
Amortized Time algorithm for listing maximal patterns in graphs, while many have been designed
for non-maximal listing problems [Rus03, Uno15]. In some cases, it may even be difficult to decide
whether a solution \(X \in F\) is maximal or not. In this thesis we will often focus on problems that
fall within strongly accessible set systems: as previously defined, these are the systems for which
\(X, Y \in F\) and \(X \subset Y\) implies that there exists \(z \in Y \setminus X\) such that \(X \cup \{z\} \in F\). This class of
set systems is particularly relevant in that it is the most general one for which the maximality of
a solution can be verified in polynomial time.

Unfortunately, it has long been known that, unless \(P = NP\), achieving output-sensitive
algorithms for listing maximal solutions is not always possible, even for restricted classes of
problems such as hereditary properties, as discussed with more detail in Section 2.6. On the
other hand, we will be able to provide a non-trivial upper bound for the enumeration of maximal
solutions within strongly accessible set systems in Chapter 5.

2.5 Real-world networks

As we have seen in the previous sections, enumeration problems may be hard to solve, due both
to the large amount of solutions that can exist in a graph and the difficulties linked to finding
maximal solutions.

In this section we give an introduction to real-world networks, showing their structure and
common properties. We will then explain how these properties will drive the design of the
algorithms presented in this paper.

2.5.1 Common features

Extensive effort has been employed in studying and modelling networks to understand their
dynamics and driving principles. In the following, we describe some properties commonly found
in networks that arise from real-world phenomena, providing some contextual examples.

Locally clustered. Real-world networks tend to be clustered, meaning that they will contain
groups, often small, which are more densely connected than the average of the network. In social
networks this is supported by the intuitive argument that two people with a friend in common have
more chances of connecting than two random strangers, and by empirical data (e.g., [UKBM11]). This is true also on networks that represent interactions between proteins [YOB04] and other types of real-world networks [ASBS00].

**Small diameter.** The diameter of a graph is the maximum *distance* between any two of its vertices. The diameter of real-world networks tends to be at most logarithmic in the number of vertices ($O(\log n)$).

An early and well known observation of this phenomenon on social networks, comes from a famous experiment of psychologist Stanley Milgram, published in 1967 [Mil67]: the experiment involved candidates in the US attempting to send a letter to an unknown target person by forwarding it to personal friends with better chances of knowing the target. The so created “chains of correspondence” had size usually smaller than 10, and about 6 on average. Although fact that most chains were left incomplete generated criticism, this contributed to the popular idea that everyone in a social network is within “six degrees of separation” from everyone else, which in graph-related terminology can be translated to social networks having a diameter of 6.

Another classic example is the hubs-and-spokes structure of transportation networks such as plane routes [Ner99], where any two airports are connected by a short route due to the existence of few important airports, *i.e.*, with many connections, called hubs.

**Sparse.** In many situations, when a structure modeled as a network grows in size, the number of connections of an element in the network grows more slowly.

In social networks, the number of active connections a person can maintain is estimated to be in the order of hundreds at most, regardless of the size of the network [Dun11, UKBM11]. Transportation networks also attempt at minimizing the number of routes in order to reduce costs [Ner99]. In graph-related terms, we consider these *sparse* networks, meaning that $m = O(n)$, *i.e.*, the number of edges $m$ will be closer to the number of vertices ($n$), than to the maximum possible number of edges (approximately $n^2/2$). Another property that will be particularly useful in the proposed algorithms is that such networks tend to have small *degeneracy* [ELS13].

**( Mostly) connected.** Even though sparse, real-world networks tend to be connected, meaning that the vast majority of the vertices (often more than 99%, even in large networks [UKBM11, Bra17, AB02]) will be connected to each other. This is sometimes due to connectivity being a requirement, as is the case in road networks, plane routes or computer networks which must be connected to work properly, but is common even in networks without a man-made driving design. The largest set of connected vertices is often referred to as the giant connected component, or simply GCC.

**Scale-free.** We say that a graph is *scale-free* when its degree distribution follows a *power law*, meaning that the portion $P(k)$ of vertices having degree $k$ is approximately $k^{-\gamma}$. The name scale-free derives from the fact that such a distribution maintains a similar shape, when plotted, regardless of the scale of the axes. Less formally, in a scale-free network we have a majority of vertices with only few connections, and few important vertices, sometimes called *hubs* as a reference to the hubs-and-spokes structure model, which are connected to a large number of vertices instead. As real data is often imperfect, the more generic terms “heavy-tail” or “long-tail” have been coined to refer to this structure, where the vertices with high degree are considered the tail of the degree distribution. This phenomena is sometimes explained by the *preferential*
attachment principle, also informally known as “rich gets richer”, which states that the more connections an entity has, the easier it is to gain new ones.

A staggering example of how the degree distribution quickly declines in such networks is the social network Twitter: the most followed account, @katyperry, has over 104 million followers (as of October 2017); the 100th most followed account, @SnoopDogg, has about one order of magnitude fewer followers (17 million), and the 1000th most followed, @PLLTVSeries, has “only” about 4 million [Cou17], over a total base of over three hundred million active users. However, the network is extremely sparse, with the average number of connections per user being in the order of hundreds [KLPM10, Tel17]. Protein protein interaction networks [HDB+05], as well as many other types of real-world networks [ASBS00] have a scale-free (or heavy-tail) structure.

2.5.2 Common dynamics

A question that researchers have tried to answer in the last few decades is why do so many real-world networks, arising from such different domains, have these similar features. Some properties can be easily motivated: e.g., social networks are sparse because a person can only maintain so many friendships. Other, however, are harder to explain. Some of the effort devoted to answering this question has been by studying random graph generation models.

An accurate model can provide a deep understanding of the dynamics of the modeled structure, explain the reason behind some of its properties, and even act as an unlimited source of experimental data. While such a thing as a perfect random model for real-world networks may not exist, we can gain some insight by looking at some well-established ones.

Erdős and Rényi proposed in 1959 [ER59] a model which is arguably both the simplest and best known for generating graphs. The model itself is very straightforward and comes in two closely related versions:

- **ER(n,M):** Generate a graph with $n$ vertices and $M$ unique edges taken randomly from the set of all possible edges ($\{1, \ldots, n\} \times \{1, \ldots, n\}$).

- **ER(n,p):** Generate a graph with $n$ vertices and insert each possible edge with probability $p$.

The models are extremely similar when $M = p \cdot \frac{n(n-1)}{2}$, although the second one does not guarantee a specific number of edges.

The ER model represents a network in which connections are created randomly, without any driving principle. This, surprisingly, is enough to guarantee two properties commonly found in real-world networks: it has been proven that the graph $ER(n,p)$ will have a giant connected component, and its diameter will be $O(\log n)$, even when the graph is extremely sparse, i.e., when $np = \Theta(1) + 1$ [CL01].

However, this is not enough to explain other properties, as for example, ER graphs are not clustered: for example it is straightforward to see (in particular from the second model) that two neighbors of the same vertex have the same chance of being connected as two random vertices.

Watts and Strogatz proposed a simple model [WS98] to explain how real-world networks can have small diameter while being locally clustered, which is referred to as the small world phenomenon.

Generating a graph $WS(N,K,\beta)$ consists in:

1. Generate a ring lattice of size $N$.
2. Connect each vertex to its $K$ closest vertices.
3. Rewire every edge with probability $\beta$. 

30
Note that $0 \leq \beta \leq 1$ and, to model sparse networks, $K$ is assumed to be $O(\log N)$.

This model manages indeed to reproduce the small world phenomenon, by generating a graph in which vertices are mostly connected with other vertices that are locally close to them, and at the same time have a small diameter proper of real-world networks: the diameter will tend to $\frac{\log(N)}{\log(K)}$ as $\beta$ tends to 1, but the generated network will already approach the limit diameter even for small values of $\beta$.

Intuitively, this suggests that a network made by local communities without ties to each other, quickly becomes well connected with the addition of few random links, indeed becoming a small world.

Finally, Barabási and Albert, after studying the dynamics of real-world networks, proposed a model \cite{AB02} based on the principle of preferential attachment. The model simulates the growth of a network as follows: $BA(m_0, N, m)$ starts from an arbitrary sample of $m_0$ vertices, and adds new vertices until the total size $N$ is reached. Every time a new vertex is added, it is connected with $m$ other vertices already present in the network. The target for each connection is chosen with probability proportional to the target’s degree, making it more likely for vertices with high degree to gain even more connections (hence the rich-gets-richer nickname).

The authors proved that this preferential attachment is enough to generate graph with a scale-free degree distribution, and in particular that graph generated with this model will have a distribution that follows a power law ($P(k) \sim k^{-\gamma}$) with $\gamma = 3$.

Moreover, the clustering coefficient has been proven to be higher than in random (Erdős-Rényi) networks, and at the same time to scale negatively with the network size \cite{Bo03}.

Networks generated with this model seem to be the closest to some real-world networks, such as social networks, in terms of properties, as they do have similar diameter, high clustering coefficient and a scale-free degree distribution.

All these models are limited to reproduce only a subset of real-world networks, as each has limitations on the diversity of the network reproduces (e.g., $\gamma$ is always 3 in Barabási-Albert networks). However, the results obtained by these simple models suggest that the complex structure of real-world networks may be indeed driven, for the most part, by few and simple dynamics.

### 2.5.3 Exploiting the real-world network structure: some examples

This thesis is focused on the design of enumeration algorithm that are efficient on real-world networks. This will be done by, on one hand, employing classical enumeration techniques as efficiently as possible, and, on the other hand, by making the most out of the structure that we can expect to find in such network, which we discussed above.

We will now consider a few examples of how these properties can influence the design choices of algorithms, so that they can be made efficient for real-world networks. For reference, examples of real world graphs and their structural properties are given in Table 2.3.

Exploiting sparsity. In general graphs, the worst case bound for the number of edges $m$ is $\Theta(n^2)$. In sparse graphs, however, this is definitely not the case. Traditional graph algorithms may not differentiate between the two, and sometimes even give a complexity bound of $O(n^2)$, when this is really $O(m)$, this difference is extremely relevant in sparse networks, where $m$ is closer to $n$ than to $n^2$. When designing an algorithm meant for real-world networks, thus, we should try to avoid as much as possible loose bounds such as $O(n^2)$, in favor of tighter ones.

Other, more refined strategies can be applied. Take for example the algorithm by Tsukiyama et al. \cite{Tias77}, which enumerates maximal independent sets with delay $O(mn)$. As independent sets correspond to cliques in the complementary graph, this algorithm can be used to enumerate
maximal cliques with a delay of $O((n^2 - m) n^{11})$, i.e., with a worst case delay of $O(n^3)$. Makino and Uno, more recently [MU04] proposed an algorithm with a similar philosophy, which enumerates maximal cliques with delay $O(\Delta^4)$. The highest degree in a graph $\Delta$ can be at most $n - 1$, making the bound $O(n^4)$ in the worst case. In principle, this is worse than the $O(n^3)$ by using Tsukiyama et al.’s algorithm, however can become much faster on sparse networks, where $\Delta$ is significantly smaller than $n$.

**Data Structures.** Similar attention should be paid to the data structures involved: a matrix-based representation of the adjacency matrix of the graph usually takes $O(n^2)$ space. This is not suitable for large graphs: a graph with 10 million vertices, for example, will require about 11.6TB of main memory to be stored, assuming just one bit is used for each cell of the matrix. This means that such a graph could only be processed on a supercomputer, a rare and expensive commodity, at least at the time of writing. On the other hand, it should be noted that a graph with 10 million vertices is not even a very large graph: it is just 3% the size of Twitter [Bra17] (considering just active users) or 1% the size of Facebook [UKBM11].

It is thus clear that, for large graphs, adjacency matrices are simply not an option for larger graphs, since as we stated real-world networks may have hundreds of millions or even billions of vertices. On the other hand, a list-based representation of the adjacency matrix takes just $O(m)$ space, as for each vertex we will only store the neighbors (usually ordered by label, to perform binary searches). This will amount to using orders of magnitude less space on a sparse graph. Other efficient forms of representations such as sparse matrices also exists. Another advantage of adjacency lists is that, while checking adjacency of two vertices takes logarithmic time (by using binary searches) instead of constant time as on a matrix, scanning the whole neighborhood of a vertex will take time proportional to its degree, which is small on average, while on a matrix this takes $\Theta(n)$ time every time.

**Degeneracy.** It is worth remarking that the presence of hubs in real-world networks makes $\Delta$ sometimes very large, even in sparse graphs. Indeed, when few vertices have degree comparable to $n$, but the graph is sparse, we may have that $\Delta$, $n$ and $m$ are all within just a constant factor from each other.\footnote{11} This is when other sparsity measures may come in handy. In particular in this thesis we often exploit the degeneracy of networks, which is a well known sparsity measure [UKBM11, ELS13, WAU14, NOdM12], usually small on sparse networks even when $\Delta$ is large. We recall that this is the smallest value $d$ for which every subgraph has a vertex of degree at most $d$.

A more intuitive definition is as follows: imagine iteratively removing the vertex with smallest degree from the graph, until all vertices are removed. The sequence of vertices $v_1, v_2, \ldots, v_n$ which represent the order in which the vertices were removed is a degeneracy ordering, and the maximum among all degrees that vertices had the moment they were removed is $d$, the degeneracy of the graph. This also implies that each vertex has at most $d$ vertices occurring after itself in the ordering, i.e., the forward neighbors of a vertex are bounded by $d$.

As this process can be computed in just $O(m)$ time, the degeneracy ordering is a valid tool, used in several works which deal with sparse graphs. For example, it has been effectively exploited by Eppstein et al. [ELS13] in the context of input-sensitive algorithms for enumerating maximal cliques: while the worst case optimal time for listing maximal cliques has been proven to be $O(3^{n/3})$ by Tomita et al. [TTT06], [ELS13] improves this bound for graphs with small $\Delta$.\footnote{This is surprisingly possible even on large networks: take the Twitter example mentioned above, where the most followed user has as many followers as roughly one third of the active users, yet the average user only has tens to hundreds of followers.}
degeneracy: the algorithm essentially consists in decomposing a graph in smaller subgraphs where cliques must be found only among the forward neighbors of a given key vertex. This simple principle allows the algorithm to run in $O(d(n - d)3^{d/3})$. While this is higher than $O(3^{n/3})$ in the worst case (as $d$ can be up to $n - 1$), it can be much smaller in sparse graphs, where $d$ is much smaller than $n$.

Problem decompositions. When the input data is very large, it does not always fit in the main memory of the available hardware. Many different solutions to this problem have been proposed, many of which involve decomposing the problem in smaller sub-problems. In graphs, generic frameworks such as MapReduce [DG08] are difficult to apply due to the linked nature of the data, but ad-hoc approaches are not uncommon.

Xu et al. [XCFB14] proposed a distributed algorithm for finding all maximal cliques: similarly to [ELS13], the graph is decomposed in up to $n$ subgraphs with some key vertices, which include all neighbors of such vertices. This has clearly a lower bound of $\Delta$ vertices per subgraph in the worst case which, while effective on sparse graphs, may fail to reduce the problem’s size on scale-free graphs due to the presence of hubs (recall the 104 million Twitter followers of @katyperry [Cou17]). Some preliminary work by the author, object of his Master Thesis and published in [CVM+16b], shows a more suitable way of decomposing scale-free networks, which allows listing maximal cliques by processing subgraphs with at most $d$ vertices, i.e., not constrained by $\Delta$.

Space efficiency. Even when a graph fits in main memory, it is often important not to waste space. Indeed, any in-memory graph algorithm will use a certain amount of space to store the input graph, plus some additional space. As both need to fit in main memory, the smaller the additional space is, the larger are the graphs that can be processed.

One simple trick which can be applied to most recursive approaches is that of storing differences. For example, assume that a backtracking algorithm is gradually building a partial solution by adding one element at a time. The trivial way to store the solution is to build a new one in each nested recursive call, to then discard it and recover the previous one when backtracking. This recursive structure can be found, for example, in Chapter 7 as there are up to $n$ recursion levels in the algorithms considered, and solutions have size $O(m)$, such a trivial storage of the solution would require $O(mn)$ space. However, note that if we only add elements (in this case, edges) to the partial solution when generating a nested recursive call, it is sufficient to store just the set of new edges that were added to the partial solution, so that restoring the previous partial solution when backtracking can be done by removing these new edges from the current one. Since every edge may be added at most once to the solution, the total amount of space used by storing all partial solutions at any time is reduced to $O(m)$.

Stateless enumeration. By pushing the principle above a step further, we will introduce the concept of stateless enumeration, applied to reverse-search algorithms. Given a recursion tree, such as that of a recursive algorithm, rather than storing the difference between a recursive call and the “parent” one, a stateless algorithm does not store any information at all, but will restore the state of the parent call, in order to backtrack, just by using some well defined rules. This may seem a trivial improvement, but it is the key to achieve sublinear additional space: indeed many backtracking approaches on graphs have $n$ or $m$ recursion levels, thus storing even just one bit of information per level would result in using $\Omega(n)$ or $\Omega(m)$ additional space. By removing this requirement, and turning the recursion into an equivalent iterative structure, the additional space is given by just the amount of space used in a single call. More details, as well as application examples of this technique will be shown in Chapters 3, 4, and 5.
Table 2.3: Properties of some real-world graphs from the LASAGNE \[LoAoGN\] repository.

2.6 Limits of enumeration

2.6.1 Structural complexity of listing: $\mathcal{L}P$-completeness

Recall the efficiency classes for listing problems explained in Section 2.2.3 and graphically represented in Figure 2.2. The problems in these classes are all linked to $\text{NP}$ relations, \textit{i.e.}, such that the time to decide whether a given set is a solution, and the size of the solution itself are both polynomial in the size of the input. Let us take a step further in the hierarchy by defining the class $\mathcal{L}P$ of all listing problems associated with an $\text{NP}$ relation. This class was introduced in the PhD Thesis of Marco Rospocher \[Ros06\] and is analogous to the class $\#P$ for counting problems. The thesis also defines the class of $\mathcal{L}P$-complete problems: a problem in $\mathcal{L}P$ is $\mathcal{L}P$-complete if the existence of a $\text{PTT}$ algorithm for the problem implies the existence of a $\text{PTT}$ algorithm for all problems in $\mathcal{L}P$.

Consider the problem of listing \textit{prime implicants} of a monotone boolean formula: a prime implicant is an inclusion minimal set of variables such that, if all set to 1, satisfy the formula regardless of the values of the other variables. \[Ros06\] shows that, even though finding \textit{a single} prime implicant can be easily done in $\text{PTime}$, listing all prime implicants is $\mathcal{L}P$-complete. Furthermore, it is known from Leslie Ann Goldberg’s PhD Thesis \[Gol91\] that the problem of listing all prime implicants can \textit{not} be done in $\text{PTT}$ unless $\text{P} = \text{NP}$. Since a $\text{PTT}$ algorithm for any $\mathcal{L}P$-complete problem would imply a $\text{PTT}$ algorithm for listing prime implicants, this implies that, if $\text{P} \neq \text{NP}$, there is no $\text{PTT}$ time algorithm for any $\mathcal{L}P$-complete problem.

2.6.2 Hardness of listing independence set systems

A similar result was obtained in 1980 by Lawler et al., \[LLRK80\], who studied listing maximal sets within \textit{independence set systems} (see Section 2.1.2).

The work contains a reduction which, given a boolean formula with $N$ variables, generates an appropriate independence set system which has exactly $N$ dummy maximal solutions, plus one maximal solution for each satisfying assignment of the corresponding formula.

The reduction is striking in that, to check whether the boolean formula can be satisfied (a classical $\text{NP}$-complete problem), it is sufficient to run the algorithm until $N + 1$ solutions are found; any output-sensitive algorithm, even a $\text{PTT}$ one, would solve this problem in $\text{PTime}$. This implies that no general algorithm for listing maximal sets within independence set systems can be designed unless $\text{P} = \text{NP}$.

For this reason, both \[LLRK80\] and \[CS05\] try to link the complexity of a general enumeration problem on set systems to that of an easier \textit{restricted problem}, showing that an efficient algorithm...
can be obtained when the restricted problem can be solved efficiently.

Finally, the reduction by Lawler et al. [LLRK80] is also interesting in that it provides a clear parameterized lower bound for the complexity of listing problems if combined with the strong exponential time hypothesis [IP99] (seth):

The strong exponential time hypothesis essentially states that the satisfiability of a boolean formula with $N$ variables (SAT) can not be solved in less than $O(2^{\epsilon N})$ time for any $\epsilon < 1$.

In the SAT reduction by Lawler et al., in order to decide the satisfiability of the given formula, the listing algorithm only needs to list a polynomial number of solutions (up to $N + 1$). If $C$ is the cost per solution of the algorithm, seth implies that $C \cdot N > 2^N$. Recall from Section 2.1 that $q$ represents the maximum size of a solution of a listing problem; in this case $q$ is bounded by $2N$. Thus, an algorithm with cost per solution $O(2^{n/X} \cdot \text{poly}(N))$, with $X > 2$, could be used to solve a SAT problem in $O(2^{\epsilon N} \cdot \text{poly}(N))$ with $\epsilon = 2/X < 1$, which would immediately break seth.

We thus have that a general listing algorithm which can list maximal solutions within any independence set system must have a cost of $\Omega(\alpha \cdot 2^{n/2})$, unless seth is false.

Intuitively, this means that the hardness of an enumeration problem is linked to the size of its solutions. In Chapter 5 we further investigate this link, show that this lower bound is matched by an upper bound of $O(\alpha \cdot 2^{X} \cdot \text{poly}(\vert U \vert))$.

2.7 Beyond enumeration

While enumeration can be a powerful tool, as remarked in Section 1.1, the result of an enumeration algorithm usually needs to be further processed to obtain the needed information.

As noted for road networks 1.1.1, the user may use an enumeration algorithm in combination with a complex evaluation function in order to retain the best one(s), i.e., which corresponds to using an enumeration algorithm to solve an optimization problem.

In other cases, as for frequent itemset mining, the user is indeed interested in many itemsets, as each potentially contains unique and useful information.

Still, the amount of solutions can very well be exponential, but the user may not need all of them. The first and important principle which we use to cut redundancy is that of only finding maximal solutions, and discarding all solutions which are contained in larger ones. However, even maximal solutions can be redundant, in that two or more solutions may only differ by insignificant amounts and essentially represent the same information.

The second part of this thesis is devoted to addressing this issue. The part is called Beyond Enumeration, as the solutions proposed cannot be defined as just results of an enumeration problem, but use enumeration as a starting point.

The goal of this part is to improve the usability of the result. To do so we focus on two aspects:

Diversity

In some situations, we may be interested in a diversified set of solutions, e.g., such that each solution is significantly different from all others.

Such approaches are receiving attention in recent years, due to the ever increasing size of real-world datasets and thus the output size of enumeration algorithms. For example, some [YFL16, YQL+16] have investigated the top-k diversified subgraph enumeration problem, that is, finding $k$ suitable subgraphs that maximize the number of vertices involved in them. Finding the optimal set of solutions is harder than the enumeration itself, thus the problem calls for approximated or heuristic approaches. Furthermore, there does not seem to be a single widely accepted strategy
so far, thus the approaches presented in this part of the thesis are designed on a case-to-case basis which aims at highlighting the properties and desiderata of the problem at hand.

As a case study, consider the problem of finding common substructure between two biological networks, where each solution corresponds to “associating” a part of the first network to a similar part of the second one. It is straightforward to see why diversity is desired: two associations concerning the same areas of the networks are essentially carrying the same information. We will address this problem in Chapter 9. In Chapter 10 we will then consider the problem of retrieving a diverse, yet comprehensive set of cliques from a graph, by giving an efficient heuristic algorithm for finding an edge clique cover.

Significance

In some scenarios we may be interested in retrieving many solutions, but solutions may not all have the same importance. In community detection, for example, we often identify communities as dense subgraphs: however, a small dense subgraph may not be index of a real community, but may randomly occur due to some non significant links or some noise in the data. We thus may be interested in observing just larger communities. We address this problem in Chapter 12. Furthermore, in Chapter 11 we address the problem frequent itemset mining: we show that state-of-the-art approaches may either loose some significant solutions, or find many non-significant ones, which correspond to random associations between frequently occurring items. We thus address the problem of efficiently finding significant frequent itemsets without generating many non-significant ones.
Part I

Efficient Enumeration
This chapter deals with the widely studied problem of efficiently listing maximal cliques in a graph. When dealing with real-world networks, due to the potentially large size, reducing the space usage without losing efficiency has become a relevant issue for enumerating patterns and performing network analytics. This chapter presents techniques for efficiently listing cliques in undirected graphs, providing the first sublinear-space bounds with guaranteed delay per solution.

We propose an algorithm with sublinear space usage, whose delay also improves upon state of the art algorithm on sparse graphs. We then propose another algorithm which uses just minimal space, whose delay is still competitive with the state of the art. As a side note, we remark the relevance of theoretical delay bounds, by showing a class of bad graphs which cause the current fastest non-output-sensitive algorithm for the problem to incur in exponential delay.

3.1 Introduction

The design of efficient algorithms for enumerating all possible solutions of a given problem dates back to the 1950s \cite{Bie60, Gil56, Tre54}. Enumeration algorithms have the purpose of either counting the number of solutions or listing the solutions one by one. Their study originated in the area of complexity and optimization \cite{Fuk96, JYP88, Val79}, and then spread over several other application domains, including bioinformatics, machine learning, network analytics, social analysis and neuroscience \cite{ANRD15, MSOI02, SVP09, Ts07}. For instance, a number of papers described how to enumerate triangles \cite{BPWZ14, AYZ95, IR78, SW05} and their generalizations such as cliques or other dense subgraphs \cite{BK73, CYQ13, CN85, CR15, DWX09, ELS13, HR13, MU04, PS08, TTT06, Uno08}. Among the first problems attacked is the enumeration of maximal cliques \cite{Akk73, Bie60, BK73, GL79, Joh76, MC72}, where a maximal clique is a subset of pairwise connected vertices that is maximal under inclusion (e.g. the graph in Figure 3.1 contains three maximal cliques), and is still a central topic in network analytics nowadays.

Measures. This chapter focuses on two worst-case efficiency measures: delay and additional space. As discussed in Section 2.2, these are particularly relevant for enumeration algorithms.

We remark that delay and space are somehow related measures. An algorithm with good overall time can accumulate solutions in the shared memory or store them temporarily in a file for a subsequent phase of postprocessing. If space is limited (especially in fast memory), this approach cannot be taken into account. Furthermore, as most current output-sensitive approaches have a recursive nature, they can easily reach $\Theta(n)$ nesting levels in the worst case.
Figure 3.1: Running example of a graph with its maximal cliques $K_1 = \{1, 3, 6\}$, $K_2 = \{2, 3, 4\}$, $K_3 = \{3, 4, 5, 6\}$.

even for sparse graphs, thus using recursion should be avoided: indeed, just storing one memory word per recursion level kills sublinearity.

This scenario motivates the search for enumeration algorithms that provably use sublinear space and have good delay bounds. Some space-efficient algorithms [CZKC12, CVM+16b, XCFB14] work well in practice for real-world networks but cannot guarantee sublinear space.

In the following we will employ the degeneracy ordering for the vertices of the input graph $G$. As defined in Section 2.1, it means that there exists an integer $d$, small if $G$ is sparse, such that each vertex has $d$ or fewer forward neighbors, i.e., neighbors appearing later in the ordering.

The degeneracy ordering can be computed in $O(m)$ time and $O(n)$ additional space by repeatedly removing the vertex of minimum degree from $G$. As we aim at using sublinear space, we show in Section 3.7.1 how to reduce the space usage to $O(1)$. Furthermore, it can be proved that $d = O(\sqrt{m})$, and that the size $q$ of the maximum clique and the maximum degree $\Delta$ in the graph $G$ satisfy $q - 1 \leq d \leq \Delta \leq n - 1$. This means that an additional space usage of $O(d)$ is indeed sublinear.

Our results. We provide the first algorithms with sublinear space and bounded delay for enumerating the maximal cliques when the vertices of $G$ are provided in degeneracy order. The last two rows in Table 3.1 report our bounds, where the $\tilde{O}()$ notation ignores $\log^{O(1)}(n + m)$ factors. We observe that our bounds are parameterized in terms of $q, d, \Delta$ instead of $m, n$, whenever possible, as they are actually smaller. For example, network eu-2005 has $q = 387$, $d = 388$, $\Delta = 68,963$ and $n = 862,664$. Also, since both $q$ and $d$ are always $O(\sqrt{m})$, our space is provably sublinear (e.g. around 20 kilobytes for eu-2005). Moreover, our $O(q)$ space is close to optimal as we have to single out a subset of $q$ vertices from $G$. Our theoretical bounds have been validated with experiments in Section 3.9.

The other rows in Table 3.1 report the bounds for the main results in the state of the art (see Section 3.2 for a discussion). The setup time is the preprocessing cost before starting to list the solutions, and ours is comparable to that of previous results. As for the delay, the cost in the last row is asymptotically smaller in many cases, except for dense graphs, where matrix multiplication based algorithms [CR15, MU04] are preferable (but massive networks can hardly be processed.

\footnote{1Or an equivalent iterative approach that uses a stack.}

\footnote{2We remark that in this chapter we aim at emphasizing that small footprint enumeration algorithms have more chances to reduce memory contention, cache coherence, and memory bandwidth issues when run on modern processors. Explicitly addressing cache-efficient or cache-oblivious algorithms in the parallel or distributed setting might be the inspiration for future work in this direction.}

\footnote{3Note that $\Delta$ is not always sublinear in the graph size as real-world networks are sparse and could have $\Delta = \Theta(n)$, as shown in Table 4.2 (see also CVM+16b).}
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Add. Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Setup</td>
<td>Delay</td>
</tr>
<tr>
<td><strong>Bron-Kerbosch</strong> [BK73]</td>
<td>$O(m)$</td>
<td>unbounded</td>
</tr>
<tr>
<td><strong>Tomita et al.</strong> [TTT06]†</td>
<td>$O(m)$</td>
<td>$\Omega(n^2)$</td>
</tr>
<tr>
<td><strong>Eppstein et al.</strong> [ELS13]</td>
<td>$O(m)$</td>
<td>$\Omega(3^{n/6})$</td>
</tr>
<tr>
<td><strong>Tsukiyama et al.</strong> [TIAS77]</td>
<td>$O(n^2)$</td>
<td>$O((n^2 - m)n)$</td>
</tr>
<tr>
<td><strong>Chiba-Nishizeki</strong> [CN85]</td>
<td>$O(m)$</td>
<td>$O(md)$</td>
</tr>
<tr>
<td><strong>Johnson et al.</strong> [JYP88]</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
</tr>
<tr>
<td><strong>Makino-Uno</strong> [MU04]</td>
<td>$O(mn)$</td>
<td>$O(\Delta^4)$</td>
</tr>
<tr>
<td><strong>Chang et al.</strong> [CYQ13]†</td>
<td>$O(m)$</td>
<td>$O(\Delta h^3)$</td>
</tr>
<tr>
<td><strong>Makino-Uno</strong> [MU04]*</td>
<td>$O(n^2)$</td>
<td>$O(n^{2.37})$</td>
</tr>
<tr>
<td><strong>Comin-Rizzi</strong> [CRL15]*</td>
<td>$O(n^{5.37})$</td>
<td>$O(n^{2.09})$</td>
</tr>
<tr>
<td><strong>Proposed</strong></td>
<td>$\tilde{O}(m)$</td>
<td>$\tilde{O}(qd(\Delta + qd))$</td>
</tr>
<tr>
<td><strong>Proposed</strong></td>
<td>$\tilde{O}(m)$</td>
<td>$\tilde{O}(\min{md, qd\Delta})$</td>
</tr>
</tbody>
</table>

$n = \#\text{vertices}$  
$m = \#\text{edges}$  
$\Delta = \text{max degree}$  
$\alpha = \#\text{maximal cliques}$  
$q = \text{largest clique size}$  
$d = \text{degeneracy}$

Table 3.1: Bounds for maximal clique enumeration, where $q - 1 \leq d \leq \Delta \leq n - 1 \leq m$, $d = O(\sqrt{m})$. †: it does not list maximal cliques, but outputs a compressed representation. *: it uses matrix multiplication. A lower bound in the column for the delay means that there exists a family of graphs with that delay.

by quadratic space algorithms). As for the overall time, we have a similar improvement for output-sensitive bounds where the $\alpha$ term appears. Moreover, we observe that [ELS13] has great time performance in practice and is the state of the art for its performance on real-world graphs (see Section 3.9) but it is not output-sensitive as $\alpha$ does not appear in the complexity, and cannot guarantee sublinear space; also, its delay can be exponential as discussed in Section 3.8.

Summing up, our algorithms compete with the state of the art when suitably implemented, with the additional bonus of guaranteeing small space and bounded delay.

**Chapter structure.** After discussing the related literature in Section 3.2 and some preliminary notions in Section 3.3, we revisit the reverse search for listing maximal cliques in Section 3.4, adding further proprieties that will be the keys for its stateless implementation described in Section 3.5. The details of the implementations are further discussed in Section 3.6, along with the inplace preprocessing of the input graph shown in Section 3.7.2 to guarantee that our solution fully achieves its bounds. To complete the discussion, Section 3.8 introduces some families of graphs that cause the Bron-Kerbosch based algorithms to take very large delay, thus motivating our reverse search approach for a bounded delay. Section 3.9 reports the experimental study performed on the main algorithms discussed here. Final conclusions are drawn in Section 3.11.
3.2 Related work

The idea of using small space in enumeration algorithms is not new, as witnessed by the notion of "compactness" introduced by Fukuda [Fuk96]: however the goal is not sublinear space as this thesis, but polynomially bounded space in terms of the input size and the maximum output size of a solution. It is also worth mentioning that the class of CAT (constant amortized time) enumeration algorithms described in Ruskey’s book [Rus03] seems to be very promising but our algorithms cannot fall within this class as their amortized cost per solution is non-constant.

Our algorithms are based on the reverse search paradigm introduced by Avis and Fukuda [AF96] as it has been conceived to be space-efficient by its authors. Also, we reuse some of the machinery introduced by Tsukiyama et al. [TIAS77] and Makino and Uno [MU04]. In the reverse search the solutions are the nodes of a solution digraph, for which a “successor” function is defined to jump from one solution to the other. (Gély et al. [GNS09] study new combinatorial properties of this solution digraph.) A directed spanning tree of the solution digraph represents all the solutions, where the depth corresponds to the number of nested levels of the corresponding recursion. To achieve sublinear space, our algorithms avoid to materialize this spanning tree by employing a stateless reverse search without using the stack, plus other properties that exploit the structure of the maximal clique enumeration problem.

Turning to the state of the art for the maximal clique enumeration problem, Table 3.1 summarizes the main results. The papers by Bron and Kerbosch [BK73] and Tsukiyama et al. [TIAS77] have defined the main lines of research for algorithms using polynomial space, and they are currently at the heart of many other algorithms. The Bron-Kerbosch algorithm relies on a backtracking scheme that is adopted in several efficient algorithms [ELS13, TTT06, XCFB14]. The algorithm by Tsukiyama et al. has been originally conceived for the enumeration of maximal independent sets, which is a problem equivalent to that of maximal cliques, and inspired at least in part the algorithm by Johnson et al. [JYP88], which produces solutions in lexicographic order but requires exponential space (see also [GNS09]). The approach has been subsequently adapted to maximal clique enumeration by Chiba-Nishizeki [CN85] and Makino-Uno [MU04] has reinterpreted some ideas in [TIAS77] for the paradigm of reverse search.

**Bron-Kerbosch scheme.** The Bron-Kerbosch based algorithms are a popular choice for enumerating maximal cliques due to their simplicity and good performance in practice. The original version [BK73] does not provide any guarantee. The version in [TTT06] guarantees a total running time of $O(3^{n/3})$, which is optimal for Moon-Moser graphs as they have $3^{n/3}$ maximal cliques [CK08, MM65]. The version in [ELS13] further refines and improves the work for sparse graphs, which may have up to $(n - d)3^{d/3}$ maximal cliques, by producing an algorithm with $O(d(n - d)3^{d/3})$ time. All of these approaches use $O(n + q\Delta)$ space to store sets of candidates and visited vertices. It is possible to modify these algorithms to decrease their space usage (e.g. by modifying the data structure in [ELS13]) but, to the best of our knowledge, with the currently known techniques they would still require $\Omega(\Delta)$ time, which is not sublinear as $\Delta$ can be $\Theta(n)$ even in sparse graphs where $m = O(n)$. Algorithms that follow this scheme are characterized by their complexity being related to the worst-case number of maximal cliques in a graph (instead of $\alpha$), and give no guarantees on the cost per solution nor the delay: we show in Section 3.8 that both these costs are $\Omega(n^3)$ for [TTT06], while [ELS13] can have a delay of $\Omega(3^{n/6})$ for some families of graphs.

**Tsukiyama et al. scheme.** The motivation behind the algorithms in this class is to achieve an output-sensitive cost that is proportional to the number $\alpha$ of maximal cliques times a function that depends on the graph parameters. The original algorithm by Tsukiyama et al. [TIAS77]
is a backtracking procedure that enumerates maximal independent sets in \(O(mn)\) time per solution. As a maximal independent set is a maximal clique in the complementary graph, this gives an algorithm for enumerating maximal cliques in \(O((n^2 - m)n)\) time per solution. The adaptation of the algorithm to maximal cliques has been improved by Chiba and Nishizeki \cite{CN85}, who bring the delay down to \(O(md)\) \footnote{The work actually exploits the arboricity, but we use \(d\) for simplicity as the arboricity is \(\Theta(d)\).} this algorithm is still based on a stateful backtracking procedure in which the recursion tree has depth \(\Omega(n)\), making the required space \(\Omega(n)\). Makino and Uno \cite{MU04} take a step towards statelessness by adapting \cite{TIAS77} and \cite{JYP88} to the paradigm of reverse search. The algorithm is still a stateful recursive approach, which takes \(\Omega(n)\) space (the depth of the recursion tree). Differently from its predecessors, each node of the tree corresponds to a unique maximal clique whose children can be computed as a function of the graph and the maximal clique itself. The algorithm by Makino-Uno is provided in two versions, one combinatorial with delay \(O(\Delta^4)\) and one based on matrix multiplication with delay \(O(n^{2.37})\). The former version has been improved to \(O(\Delta^3)\) by Chang et al. \cite{CYQ13}, where \(\Delta\) is a sort of \(h\)-index, namely, the smallest integer \(h\) such that \(|\{v \in V : |N(v)| \geq h\}| \leq h\): since the reverse search is not stateless, here the space remains \(\Omega(n)\). The latter version of Makino-Uno has been improved to \(O(n^{2.09})\) by Comin and Rizzi \cite{CR15} using matrix multiplication but requiring higher space usage and setup time.

### 3.3 Preliminaries

Refer to Section 2.1 for terminology. The algorithms proposed in this chapter are intended for an undirected, connected, simple graph \(G = (V(G), E(G))\), with \(n\) vertices, \(m\) edges, maximum degree \(\Delta\), degeneracy \(d\) and largest clique size \(q\). We assume that the vertices of \(V\) are labelled as \(v_1 < v_2 < \cdots < v_n\) in degeneracy ordering, and \(G\) is represented using ordered adjacency lists. We denote by \(V_{\leq i}\) the set of vertices \(v_1 \ldots v_i\). Let \(N(v)\) be the set of neighbors of \(v\) and \(N_{\geq i}(v) = \{x \in N(v) : x > v\}\) those later in the degeneracy ordering. We define \(N_{\leq i}(v)\) analogously, and call \textit{heads} the vertices \(v\) such that \(N_{\leq i}(v) = \emptyset\). Given any set of vertices \(A \subseteq V\) and any \(v \in V\), we use the shorthand \(A_{\leq v} = A \cap V_{\leq v}\) and \(A_{> v} = A_{\leq v} \setminus \{v\}\). We also define \(N(A) = \bigcap_{v \in A} N(v)\) as the set of vertices adjacent to all those in \(A\).

A clique is a subset \(K \subseteq V\) of pairwise connected vertices: we will use \(K\) to denote both this subset of vertices and the subgraph \(G[K]\) induced by them. Given two cliques \(A\) and \(B\), we say that \(A < B\) if \(A\) is lexicographically smaller than \(B\) as vertex sets, thus inducing a lexicographic order on the cliques. Given a clique \(K \subseteq V\), we define \texttt{COMPLETE}(\(K\)) as the lexicographically smallest clique that contains \(K\) and is maximal. As we will see, \texttt{COMPLETE}(\(K\)) is one of the ingredients to avoid duplication when listing maximal cliques.

### 3.4 Reverse search revisited for maximal cliques

We begin by discussing a reverse search algorithm for the maximal clique enumeration that revises and simplifies the algorithm by Makino and Uno \cite{MU04}, which is itself a reinterpretation of the works of Tsukiyama et al. in \cite{TIAS77} and Johnson et al. in \cite{JYP88} for maximal independent sets.

Consider the graph shown in Figure 3.1 with its three maximal cliques \(K_1 < K_2 < K_3\). Suppose that we have found a (not necessarily maximal) clique \(K\): as \(K\) might be included in several maximal cliques, this ambiguity can be solved by singling out the one indicated by \texttt{COMPLETE}(\(K\)), namely the lexicographically smallest \textit{maximal} clique that contains \(K\). For example, \(K = \{3\}\) is included in \(K_1, K_2, K_3\), so we single out \texttt{COMPLETE}(\(K\)) = \(K_1\): one way of
thinking of \( \text{COMPLETE}(K) \) is that, as long as possible, it adds the smallest vertex that is adjacent to all those currently in \( K \); thus \( \text{COMPLETE}({3}) = \text{COMPLETE}({1, 3}) = \text{COMPLETE}({1, 3, 6}) = \{1, 3, 6\} \).

Next, following what done in [MU04], a deterministic way to go from one maximal clique to another needs to be specified. Let \( K \) be the starting maximal clique and choose a vertex \( v \notin K \). Consider the neighbors of \( v \) that belong to \( K \) and are smaller than \( v \) (i.e. those belonging to both \( N(v) \) and \( K_{<v} \)): observing that they form a clique when \( v \) is added to them, we apply \( \text{COMPLETE} \) to them and obtain the corresponding maximal clique \( D \). In formula,

\[
D = \text{COMPLETE}(K_{<v} \cap N(v) \cup \{v\})
\]

(3.1)

where \( v \notin K \) and \( K_{<v} \cap N(v) \) is nonempty.

For example, we can go from \( K = K_1 \) to \( D = K_2 \) using \( v = 2 \) or \( v = 4 \). Let us see how this holds for \( v = 4 \): we have \( K_{<v} = \{1, 3\} \) and \( N(v) = \{2, 3, 5, 6\} \), thus we obtain \( D \) by computing \( \text{COMPLETE}({3, 4}) = K_2 \).

The reverse search for maximal cliques refers to an implicit directed graph for the solutions, hereafter called solution digraph and defined as follows (e.g. see the example in Figure 3.2). There is a node per maximal clique, and there is an arc, labeled with \( v \), from maximal clique \( K \) to maximal clique \( D \) if and only if the condition in formula (3.1) is satisfied (in our example, an arc from \( K_1 \) to \( K_2 \) labeled with 4). This corresponds to the generic structure described in Section 2.3.4 (Figure 2.4), with the addition of labels on the edges. Note that the solution digraph has several potential issues: multiple arcs can connect two nodes (e.g. \( K_1 \) and \( K_2 \) are also connected with an arc labeled with 2), directed cycles can occur (e.g. that traversing \( K_1, K_3, K_2, K_1 \)), and there could be more than one source from which to start (\( K_1 \) and \( K_2 \) as we will see).

Traversing the solution digraph through an implicit directed spanning forest gives all the maximal cliques. To this end, we need to select at most one incoming arc for each maximal clique, using the notion of parent index introduced in [MU04], where the term parent clearly refers to the above forest. As the solution digraph is a multigraph, we further introduce the following notion of parent index to identify a single incoming edge from the parent.

**Definition 3.1 (parent index).** Given a maximal clique \( K \), its parent index \( \pi(K) \) is the smallest \( x \in K \) such that \( \text{COMPLETE}(K_{\leq x}) = K \).

Let us run our example in Figure 3.2 to see how to apply this notion. Consider first \( K_3 \), which has three entering arcs. We want to select one of them, thus the source of the arc becomes the parent of \( K_3 \). We first compute the parent index of \( K_3 \), observing that \( \pi(K_3) = 5 \) as \( \text{COMPLETE}({3, 4, 5}) = \text{COMPLETE}({3, 4, 5, 6}) = K_3 \) while \( \text{COMPLETE}({3, 4}) = K_2 \). Hence \( K_2 \) is the parent of \( K_3 \), and their selected arc is that labeled with \( v = \pi(K_3) = 5 \). In formula, we have the following setting.
**Algorithm 3:** Enumerate all maximal cliques (revisited Makino-Uno)

<table>
<thead>
<tr>
<th>Input</th>
<th>: Graph (G = (V, E)) where vertices are labeled in degeneracy ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>: Maximal cliques in (G)</td>
</tr>
</tbody>
</table>

Let \(v_1, v_2, \ldots, v_n \in V\) be the vertices labeled in degeneracy ordering, such that \(v_1, \ldots, v_j\) are heads, where \(j\) is the number of heads (\(1 \leq j \leq n\)).

<table>
<thead>
<tr>
<th>for (i \in {1, \ldots, j}) do</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R_i \leftarrow \text{COMPLETE}({v_i}))</td>
</tr>
<tr>
<td>(\text{spawn}(R_i))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function \text{COMPLETE}(K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>choose any (v \in K)</td>
</tr>
<tr>
<td>foreach increasing (w \in N(v)) do</td>
</tr>
<tr>
<td>if (K \subseteq N(w)) then</td>
</tr>
<tr>
<td>(K \leftarrow K \cup {w})</td>
</tr>
<tr>
<td>return (K)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function \text{spawn}(K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{CAND} \leftarrow \left{ w \in \bigcup_{u \in K} N_{&gt;}(u) \setminus K : w &gt; \text{pi}(K) \right} )</td>
</tr>
<tr>
<td>foreach (v \in \text{CAND}) do</td>
</tr>
<tr>
<td>(K'_v \leftarrow K&lt;v&gt; \cap N(v))</td>
</tr>
<tr>
<td>(D \leftarrow \text{COMPLETE}(K'_v \cup {v}))</td>
</tr>
<tr>
<td>if (\text{COMPLETE}(K'_v) = K) and (K'_v = D&lt;v&gt;) then</td>
</tr>
<tr>
<td>(\text{spawn}(D))</td>
</tr>
</tbody>
</table>

**Definition 3.2 (parent).** Given two maximal cliques \(K\) and \(D\), such that \(D<\text{pi}(D)\) is nonempty, we say that \(K\) is the parent of \(D\) if

\[
K = \text{COMPLETE}(D<\text{pi}(D)) \tag{3.2}
\]

When \(D<\text{pi}(D) = \emptyset\) (equivalently, \(\text{pi}(D) = \min(D)\)) we say that \(D\) is a root as it has no parent. For instance, \(K_2\) is a root, as \(\text{COMPLETE}(\{2\}) = \text{COMPLETE}(\{2, 3\}) = \text{COMPLETE}(\{2, 3, 4\}) = K_2\) and thus it has no parent. Also \(K_1\) is a root for the same reason.

Although the example in Figure 3.2 is small, it can be already observed that we obtain an implicit directed spanning forest with one or more roots, using the parent relation among the maximal cliques of the solution digraph. The roots are obtained by \text{COMPLETE} applied to each individual head as singleton (recalling that a head’s neighbors are all greater than it). As we will see, the parent relation cannot create directed cycles as the parent is always lexicographically smaller than its children.

The pseudocode shown in Algorithm 3 implements the above ideas. It is worth observing that, differently from \([MU04]\), the algorithm assumes that the vertices of \(G\) are given in degeneracy ordering and launches the recursive calls from the set of roots (instead of the lexicographically least maximal clique), where each root \(R_i\) corresponds to \text{COMPLETE} run on head \(v_i\). Moreover, we alter the given degeneracy ordering in such a way that the heads are at the beginning. This is always possible: as heads have no backward edges, when moved backwards in the ordering they will not change the number of forward edges of any vertex (see Section 3.7.2 for the details).

Looking at the pseudocode, we further observe that the function \text{spawn} makes use of the notion of parent index to restrict the set of candidate arcs to examine in the solution digraph. The \text{CAND} set contains the vertices that partially extend \(K\) and that are greater than \(\text{pi}(K)\). For each vertex in \text{CAND}, function \text{spawn} tries to generate the next maximal clique in the implicit directed spanning forest. Specifically, \(D\) is computed as in formula (3.1) and \(\text{spawn}(D)\) is executed if \(K\) is the parent of \(D\) using an equivalent form of formula (3.2), which is more efficient to compute and guarantees that each maximal clique is only traversed once as a node in the solution digraph\(^5\). We remark that each call of \text{spawn} returns at least a maximal clique.

Our proofs of correctness are shown in Section 3.4.1, while the space usage and delay are discussed in Section 3.4.2. They are the ground for our improved solution whose goals are discussed in Section 3.4.3.

\(^5\)The check in \text{spawn} corresponds to conditions (c) and (d) of Lemma 2 in \([MU04]\).
3.4.1 Correctness

We will make use of few technical properties to prove the correctness of Algorithm 3.

**Lemma 3.1.** Function \textsc{complete}(K) in Algorithm 3 returns the lexicographically smallest maximal clique containing the clique K.

**Proof.** \textsc{complete}(K) iteratively adds to K the smallest vertex adjacent to all vertices in K. If K is maximal the lemma is true. Otherwise, let \( Y = \text{complete}(K) \) and \( X \supset K \) a maximal clique s.t. \( X < Y \). Let \( x \) be the smallest vertex in \( X \setminus Y \). We have \( X_x \subseteq Y \) and clearly \( x \in N(K) \), thus \( x \) is a candidate to be added to \textsc{complete}(K) after the vertices in \( X_x \); as \( x \) is the smallest vertex in \( X \setminus Y \), this means that a larger vertex was chosen by \textsc{complete}(K) while \( x \) was a candidate, which is a contradiction.

\( \square \)

**Lemma 3.2.** For any two cliques \( K_1 \) and \( K_2 \), if \( K_1 \subseteq K_2 \) then \textsc{complete}(\( K_1 \)) \( \leq \) \textsc{complete}(\( K_2 \)).

**Proof.** Let \( C_1 \) and \( C_2 \) be the set of maximal cliques containing respectively \( K_1 \) and \( K_2 \), with \( K_1 \subseteq K_2 \). Clearly \( C_1 \supseteq C_2 \), and as \textsc{complete}(K) returns the lexicographically smallest maximal clique that contains \( K \) we have \textsc{complete}(\( K_1 \)) = \text{min}(C_1) \leq \text{min}(C_2) = \textsc{complete}(\( K_2 \)). \( \square \)

Although immediate to see, let us formally show that the notion of parent is robust.

**Fact 3.1.** If \( K \) is the parent of \( D \), then \( D_{\preceq \text{pi}(D)} = K_{\preceq \text{pi}(D)} \cap N(\text{pi}(D)) \) and formula (3.1) is satisfied with \( v = \text{pi}(D) \).

**Proof.** By Definitions 3.1 and 3.2, we know that \textsc{complete}(\( D_{\preceq \text{pi}(D)} \)) = K while \textsc{complete}(\( D_{\preceq \text{pi}(D)} \)) = D, with \( K \neq D \) (i.e., \( \text{pi}(D) \notin K \) and \( \text{pi}(D) \in D \)). Also, \( K_{\preceq \text{pi}(D)} \cap N(\text{pi}(D)) \) is nonempty as \( \text{pi}(D) \) is neighbor of the vertices in nonempty \( D_{\preceq \text{pi}(D)} \subseteq K_{\preceq \text{pi}(D)} \). Since \( \text{pi}(D) \) cannot be a neighbor of any vertex in \( K_{\preceq \text{pi}(D)} \setminus D_{\preceq \text{pi}(D)} \) (as otherwise \( D_{\preceq \text{pi}(D)} \) would have been larger), we have that \( D_{\preceq \text{pi}(D)} = K_{\preceq \text{pi}(D)} \cap N(\text{pi}(D)) \). Thus \( K_{\preceq \text{pi}(D)} \cap N(\text{pi}(D)) \cup \{\text{pi}(D)\} = D_{\preceq \text{pi}(D)} \), and \textsc{complete} applied to the latter gives \( D \). This way formula (3.1) is satisfied with \( v = \text{pi}(D) \).

\( \square \)

Note that the above fact implies that \( D_{\preceq \text{pi}(D)} \subseteq K \) as \( D_{\preceq \text{pi}(D)} \subseteq K_{\preceq \text{pi}(D)} \). The following lemma indicates the roots, which are the maximal cliques from which to start exploring the solution digraph.

**Lemma 3.3.** A maximal clique \( K \) is a root (i.e. it has no parent) iff \( K = \text{complete} \{v_i\} \) for a head \( v_i \) (i.e. satisfying \( N_<(v_i) = \emptyset \)).

**Proof.** Recall that \( K \) is a root if and only if \( \text{pi}(K) = \text{min}(K) \). Since \( \{\text{min}(K)\} = K_{\leq \text{min}(K)} \), we have that \( K = \text{complete}(\{\text{min}(K)\}) \) by definition of \( \text{pi} \). Moreover, the latter equality implies \( N_<(\text{min}(K)) = \emptyset \), hence \( v_i = \text{min}(K) \) is a head: indeed, if there were \( w \in N_<(\text{min}(K)) \), then \text{complete} would add \( w \) to \( \{\text{min}(K)\} \), making it different from \( K \). Vice versa, if \( N_<(v_i) = \emptyset \) holds, observe that \( K = \text{complete}(\{v_i\}) \) cannot contain a smaller neighbor of \( v_i \); then, its parent index is \( \text{pi}(K) = v_i = \text{min}(K) \) by definition, that is \( K \) is a root.

\( \square \)

Next results motivate the usage of formula (3.1) for the arcs among maximal cliques and formula (3.2) for the parent relation: specifically, a parent is always smaller than its children, which avoids loops when traversing the solution digraph from parents to children.
Lemma 3.4. Let $D$ be a non-root maximal clique, and $K$ be its parent. Then $K < D$ and $p(K) < p(D)$.

Proof. We know that $v = p(D)$ by Fact 3.1 and $K = \text{COMPLETE}(D_{<v}) \subset \text{COMPLETE}(D_{\leq v}) = D$ by Lemma 3.2. Using the fact that $K \neq D$ by construction (see the proof of Fact 3.1), we obtain that $K < D$.

We need to prove that $p(K) < v$. We have $D_{<v} \subset \text{COMPLETE}(D_{<v}) \subset \text{COMPLETE}(v)$ by definition of COMPLETE, and $K = \text{COMPLETE}(D_{<v})$ by formula (3.2) with $v = p(D)$. Hence, $D_{<v} \subset K_{<v}$. It follows that $K = \text{COMPLETE}(D_{<v}) \subset \text{COMPLETE}(K_{<v})$ by Lemma 3.2. Moreover, COMPLETE($K_{<v}$) $\subset$ COMPLETE($K$) $= K$, so COMPLETE($K_{<v}$) $= K$. By the definition of parent index, we then have that $p(K) \leq v$. The thesis follows from the fact that $v \notin K$.

Lemma below reflects what SPAWN is meant to generate in Algorithm 3.

Lemma 3.5. Let $D$ be a non-root maximal clique, and $K$ be a maximal clique. Then $K$ is parent of $D$ if and only if there exists $v \in \text{CAND}$ when invoking SPAWN($K$) such that the following facts hold, where $K'_{v}$ denotes $K_{<v} \cap N(v)$.

1. $v = p(D)$,
2. $D = \text{COMPLETE}(K'_{v} \cup \{v\})$,
3. $K'_{v} = D_{<v} \neq \emptyset$,
4. $\text{COMPLETE}(K'_{v}) = K$.

Proof. Suppose that $K$ is the parent of $D$. By Fact 3.1 formula (3.1) is satisfied with $v = p(D)$, which gives conditions 1 and 2 of the statement. Also, condition 3 is verified by Fact 3.1. Since formula (3.2) is satisfied by the hypothesis, condition 4 is equivalent to the formula as $K'_{v} = D_{<p(D)}$. Finally, $v \in \text{CAND}$ when invoking SPAWN($K$). Indeed we have $v \notin K$ by Fact 3.1 (as formula (3.1) holds) and $v > p(K)$ by Lemma 3.4. As $\emptyset \neq D_{<v} \subset K$, there exists $u \in D_{<v}$, which implies $v \in N_{>}(u)$. Hence, $v \in \text{CAND}$.

Suppose that there exists $v \in \text{CAND}$ when invoking SPAWN($K$) such that conditions 1-4 hold. Conditions 1-2 and the fact that $K'_{v} \neq \emptyset$ say that we can reach $D$ from $K$. Conditions 3-4 are equivalent to formula (3.2) as $v = p(D)$. Hence the parent of $D$ exists, and is $K$.

Theorem 3.1. Algorithm 3 lists all and only maximal cliques, and each of them is output once.

Proof. Since any clique listed by Algorithm 3 is the result of a COMPLETE call, which expands cliques until they are maximal, Algorithm 3 lists only maximal cliques. Suppose there exist one or more maximal cliques which are not listed, and let $D$ be the lexicographically smallest among them: we show that $D$ is found, contradicting the hypothesis. If $D$ is a root, it is surely found by Lemma 3.3. Thus suppose $D$ is not a root, and let $K$ be the parent of $D$ by formula (3.2). Note that $K < D$ by Lemma 3.4, which means that $K$ is found as $D$ is the smallest maximal clique not found. Thus SPAWN($K$) is executed but it finds $D$ by Lemma 3.5. This proves the first part of the statement.

Let us assume that there exists a maximal clique which is generated more than once. Let $D$ be the lexicographically smallest among such cliques. As roots have no parent, they are only generated once (from the corresponding head) by Lemma 3.3. This means that $D$ is not a root, and by Lemma 3.5 we have that $D$ is produced from its parent $K$. Since $K < D$ by Lemma 3.4, we know that $K$ is generated only once. Moreover, since $D$ is generated using a vertex $v \in \text{CAND}$,
and Algorithm 3 uses each element of \textsc{cand} exactly once, \( K \) generates \( D \) once. The second part of the statement follows.

### 3.4.2 Space usage and delay

We first discuss function \textsc{complete}. It only needs to store the evolving set \( K \), whose size is \( O(q) \), and performs \( O(\Delta) \) iterations, each taking \( \tilde{O}(q) \) time. The time cost can be also computed by observing that the cost of an iteration on vertex \( w \) is bounded by \( \tilde{O}(|N(w)|) \) time, and the cost of all the iterations over \( w \) is bounded by \( \tilde{O}(m) \) time. Overall this gives \( \tilde{O}(\min\{q\Delta,m\}) \) time. To compute the parent index \( \text{pi}(K) \) of a maximal clique \( K \), we observe that the vertices \( x \geq \text{pi}(K) \) have \( \text{complete}(K_{\leq x}) = K \) while those \( x < \text{pi}(K) \) have \( \text{complete}(K_{\leq x}) \neq K \): hence a binary search on \( K \)'s vertices suffices.

**Lemma 3.6.** Executing function \textsc{complete} in Algorithm 3 and computing the parent index \( \text{pi} \) take \( O(q) \) space and \( \tilde{O}(\min\{q\Delta,m\}) \) time.

We are now ready to discuss the complexity of Algorithm 3. For the sake of clarity, we will refer to the \textit{vertices} in \( G \) and to the \textit{nodes} in the directed spanning forest induced by \textsc{spawn} in Algorithm 3.

Consider the space occupancy for a node in \textsc{spawn}. The \( O(q) \) space taken by \textsc{complete} and the auxiliary variables is dominated by the space taken by \textsc{cand}. Indeed, its size is \( O(qd) \) and never exceeds \( O(n) \), since \( \textsc{cand} \subseteq V \) is the union of up to \( q \) adjacency sublists, each of size at most \( d \) by the degeneracy ordering. Hence we have \( |\textsc{cand}| \leq \min\{qd,n\} \) and, when \( qd < n \), this cost is parameterized by the maximum clique size \( q \) and graph degeneracy \( d \), rather than the graph size. The recursion depth of Algorithm 3 is \( O(n) \), for a total space requirement of \( O(\min\{qd,n^2\}) \). Unfortunately the space is not yet sublinear, so we need to refine the approach.

Consider now the delay of Algorithm 3. Selecting the next head to launch the recursion takes constant time as heads are contiguous. As we have no dead ends, the delay of the algorithm is bounded by the cost of \( O(1) \) recursive nodes by making use of alternative output [Uno03], where solutions are printed on alternated levels of the recursion tree (i.e. print when descending if the current distance from the root is odd, and print when ascending if it is even). In each recursive node, the cost of computing \textsc{cand} is \( \tilde{O}(\min\{qd,n\}) \) time. After that, \textsc{spawn} performs \( |\textsc{cand}| \) iterations, each having a cost dominated by \textsc{complete}. Note that the if condition tested in the loop of \textsc{spawn} takes less time than explicitly checking formula (3.2) as can be seen in Lemma 3.6. The total cost of the loop is thus \( \tilde{O}(\min\{qd,n\}\min\{q\Delta,m\}) \) time, which dominates the cost of computing \textsc{cand} and thus gives the total cost of a recursion node.

**Lemma 3.7.** Algorithm 3 uses \( O(\min\{nqd,n^2\}) \) space and takes \( \tilde{O}(\min\{qd,n\}\min\{q\Delta,m\}) \) delay.

Before getting into our improvements of the above results, we observe that our revisited description of the reverse search algorithm for the maximal cliques, presented in this section, has the merit of emphasizing which are the bottlenecks to attack, thus facilitating what discussed next. Indeed, the bounds in Lemma 3.7 already improve the state of the art.

### 3.4.3 Goals for an improved algorithm

Algorithm 3 does not meet yet our space requirements and claimed bounds. Nevertheless, it is a good starting point to build our space-efficient scheme.

The first issue is its recursive nature: in function \textsc{spawn}, each time a recursive call is performed the \textit{state} of the current call needs to be saved. As a result, the required space is that needed
by each recursive node multiplied by the height of the recursion tree. Note that the standard
stack-based transformation of recursive programs into iterative ones, as done in [CYQ13], does
not solve the issue, as the stack size could be $\Omega(n)$ anyway. Therefore we aim at navigating
implicitly the recursion tree induced by the reverse search (actually the implicit directed spanning
forest for the solution digraph discussed in Section 3.4), without using a stack.

The second issue to attack is the cand set, whose size is $|\text{cand}| \leq \min\{qd, n\}$. We need to
traverse cand without materializing it as the available space is either $O(q)$ or $O(d)$.

We will address these issues in Sections 3.5 and 3.6 and describe the initial setup in
Section 3.7.2.

3.5 Implicit navigation during the reverse search

The improved enumeration algorithm hinges on the representation of a node of the recursion
induced by spawn, so that we are able to reach its parent and its children in the spanning
forest for the solution digraph. Specifically, we represent the current node (i.e. the state of the
computation of spawn) with the following pair of bounded size.

- The current maximal clique $K$ that is the input for spawn, and
- the “bookmark” vertex $v$ in cand in the foreach loop of spawn($K$).

The bookmark vertex $v$ will allow us to resume the computation in the parent without the
need of storing information for each recursive level. Initially $v = \text{pi}(K)$ as all the vertices in
cand are larger than $\text{pi}(K)$.

Fact 3.2. The state $\langle K, v \rangle$ requires $O(q)$ space.

Along with the state, we need the navigation primitives to implicitly traverse the spanning
forest of the solution digraph.

- is-root ($K$) checks if $K$ is one of the roots (in particular, the root of the current recursion
tree).
- parent-state ($K$) returns the state for the parent node of $K$ in the recursion tree by
  formula (3.2).
- get-next-cand ($K, v$) finds the candidate vertex following $v$ in cand, for the current $K$.
- child-exists ($K, v$) checks whether the current state will lead to a child maximal clique
  by formulas (3.1)–(3.2).

Algorithm 4 shows how to implement and use the above primitives, and Lemmas 3.8, 3.9, 3.10, 3.13
prove their correctness. In particular, improved-spawn replaces spawn when invoked on a root.
Apart from the top-level while loop, improved-spawn has an internal while loop to search for
all the children of the current maximal clique $K$. To this end, it examines the next vertex in
cand and checks whether it can reach a child, setting $K$ to the latter in the positive case. When
there are no more children (childless is true), $K$ is set to its parent (or there is an exit if $K$
is the root). Hence, improved-spawn simulates a preorder traversal of the spanning forest of the
solution digraph, using iteration without a stack. As we will see, this is possible as we exploit
the properties of the current state. Moreover, we adopt the alternative output technique of [Uno03],
thus improved-spawn prints $K$ when descending if the current depth (distance from the root)
Algorithm 4: Improved enumeration of maximal cliques

Assume $\min(\emptyset) = \text{null}$, and adopt the following shorthands for maximal clique $K$:
sub-clique $K' \equiv K_{<v} \cap N(v)$ and vertex set $B_K \equiv \{u \in V : K_{<u} \subseteq N(u)\}$.

**Function** IMPROVED-SPAWN ($K$) 

\[
v \leftarrow \text{pi}(K) \\
\text{while} \ \text{true} \ \text{do} \\
\quad \text{childless} \leftarrow \text{true} \\
\quad \text{while} \ v \leftarrow \text{get-next-cand}(K,v) \neq \text{null} \ \text{do} \\
\quad \quad \text{if} \ \text{child-exists} \ (K,v) \ \text{then} \\
\quad \quad \quad K \leftarrow \text{complete}(K' \cup \{v\}) \\
\quad \quad \quad \text{/* print } K \text{ if depth is odd */} \\
\quad \quad \quad \text{childless} \leftarrow \text{false} \\
\quad \quad \text{break} \\
\quad \text{if} \ \text{childless} \ \text{then} \\
\quad \quad \text{/* print } K \text{ if depth is even */} \\
\quad \quad \text{if} \ \text{is-root} \ (K) \ \text{then return} \\
\quad \quad \text{else} \\
\quad \quad \quad (K,v) \leftarrow \text{parent-state}(K) \\
\text{end while} \\
\text{return} (K,v)
\]

**Function** IS-ROOT ($K$) 

\[
\text{return } \text{pi}(K) = \min(K)
\]

**Function** PARENT-STATE ($K$) 

\[
v \leftarrow \text{pi}(K) \\
\text{return } \langle \text{complete}(K_{<v}), v \rangle
\]

**Function** GET-NEXT-CAND ($K,v$) \n
\[
\text{return } \min\{w \in \bigcup_{u \in K} N_{>}(u) \setminus K : w > v\}
\]

**Function** CHILD-EXISTS ($K,v$) 

\[
\text{return } (N(K'_v) \cap (B_K \cup N_{<}(v))) = \emptyset
\]

is odd, and when ascending if the current depth is even\(^6\). In this way, every $O(1)$ nodes traversed by IMPROVED-SPAWN a new maximal clique is output.

**Lemma 3.8.** Let $K$ be any maximal clique examined in Algorithm.\(^3\) $K$ is a root iff $\text{pi}(K) = \min(K)$.

**Proof.** It immediately follows from Definition\(^3\). \hfill \Box

The next lemma states that given a maximal clique $D$, its parent index allows us to identify the maximal clique $K$ that generated $D$ and the vertex in $v \in \text{cand}$ used to produce $D$.

**Lemma 3.9.** Let $K$, $D$, and $v$ be defined as in \textsc{spawn} when the recursive call is launched on $D$ inside the for loop. Then \text{parent-state} ($D$) = $\langle K,v \rangle$ in IMPROVED-SPAWN, where $v = \text{pi}(D)$.

**Proof.** It follows from Lemma\(^3\). \hfill \Box

Since the vertices in \text{cand} for $K$ are examined in increasing order, when returning from the current child we are able to provide all the remaining children of $K$. Indeed let $v$ be $\text{pi}(D)$ for the current child $D$: as a consequence of Lemma\(^3\) we know that $D$ was generated from $K$ using candidate $v$ and that $K = \text{complete}(D_{<v})$. By using the next lemma, we can provide and test all the remaining candidates, i.e. the ones greater than $v$.

**Lemma 3.10.** For a given maximal clique $K$, let $\text{cand} = \{w \in \bigcup_{x \in K} N_>(x) \setminus K : w > \text{pi}(K)\}$ in \textsc{spawn} and let $z_1,...,z_r$ be the sequence of vertices generated by \textsc{get-next-cand} in IMPROVED-SPAWN. Then $\text{cand} = \{z_1,...,z_r\}$.

**Proof.** We give the proof by induction. The first time \textsc{get-next-cand} is invoked $v$ is $\text{pi} (K)$, meaning that $z_1$ is the minimum element of $\text{cand}$. Let $z_j$ be the last vertex generated by \textsc{get-next-cand} and let us assume that $z_1,...,z_{j-1}$ are the first $j$ vertices of $\text{cand}$. Then $\min\{w \in \bigcup_{u \in K} N_>(u) \setminus K : w > z_j\}$ is the $(j+1)$-th element of $\text{cand}$ if $|\text{cand}| > j$, \text{null} otherwise. \hfill \Box

\(^6\)Note that its implementation requires just a single bit for keeping the parity of the current depth, and this bit is flipped when going from parent to child or vice versa.
A little more work is required to see that the check done in \textsc{child-exists} \((K, v)\) is equivalent to the check done in the \textit{if} condition in \textsc{spawn}. For this, we introduce the set of “bad” vertices, defined as
\[
B_K = \{ u \in V \setminus K : K_{< u} \subseteq N(u) \}
\]  
and prove in Lemma \[3.13\] that the checks with the bad vertices are equivalent, using Lemmas \[3.11\] and \[3.12\]. The rationale for our choice is that the cost is lower and amortizes well among \(K\)’s children when using \(B_K\) (see Section \[3.6\]).

\textbf{Lemma 3.11.} \(N(K_v') \cap B_K = \emptyset\) is equivalent to \(\text{complete}(K_v') = K\), where \(B_K\) is defined as in formula \[3.3\].

\textit{Proof.} Let us first prove that if there exists a vertex \(w \in N(K_v') \cap B_K\) then \(\text{complete}(K_v') \neq K\). The fact that \(w \in N(K_v') \cap B_K\) implies that \(K_v' \subseteq N(z)\) for some \(z \in K_v'\), and the fact that \(w \in B_K\) implies that \(w \not\in N(z)\). Since \(w \in B_K\), it follows that \(w \not\in K\), and \(w\) is adjacent to all the vertices in \(K_{< w}\). Now, \(\text{complete}(K_v')\) will iteratively add to \(K_v'\) the smallest vertex \(z\) that is a neighbor of all the vertices in \(K_v'\), so clearly \(z \leq w\). If \(z \notin K\) then \(\text{complete}(K_v') \neq K\); if \(z \in K\) then \(z\) is still a candidate for \(\text{complete}\) after that \(z\) is added to \(K_v'\). As \(w\) remains a candidate when \(z \in K\), the process will eventually add either \(w\) or another \(z \notin K\). Hence, \(\text{complete}(K_v') \neq K\).

We now prove that if \(\text{complete}(K_v') \neq K\) then \(N(K_v') \cap B_K \neq \emptyset\). Let \(z\) be the first vertex not in \(K\) selected by \(\text{complete}(K_v')\). Since all vertices in \(K_{< z}\) were added to \(K_v'\) before \(z\), we have \(K_{< z} \subseteq N(z)\). Moreover, since \(z\) has been selected by the \text{complete} procedure, \(K_v' \subseteq N(z)\), which implies \(z \in N(K_v')\). It follows that \(N(K_v') \cap B \supseteq \{z\} \neq \emptyset\). \(\Box\)

\textbf{Lemma 3.12.} \(N(K_v') \cap N_{< v}(v) = \emptyset\) is equivalent to \(\text{complete}(K_v' \cup \{v\})_{< v} = K_v'\).

\textit{Proof.} We prove that if \(N(K_v') \cap N_{< v}(v) \neq \emptyset\) then \(\text{complete}(K_v' \cup \{v\})_{< v} \neq K_v'\). Let \(z\) be the smallest vertex in \(N(K_v' \cup \{v\}) \cap N_{< v}(v)\). Note that \(z \notin K_v'\), since \(z \in N(K_v')\). The first iteration of \(\text{complete}(K_v' \cup \{v\})\) selects the smallest vertex in \(N(K_v' \cup \{v\})\), which is \(z\), since \(N(K_v' \cup \{v\}) = N(K_v') \cap N(v)\). Since \(z < v\), we have \(z \in \text{complete}(K_v' \cup \{v\})_{< v}\), which implies \(\text{complete}(K_v' \cup \{v\})_{< v} \neq K_v'\).

We now prove that if \(\text{complete}(K_v' \cup \{v\})_{< v} \neq K_v'\) then \(N(K_v') \cap N_{< v}(v) \neq \emptyset\). Note that \(K_v' \subseteq \text{complete}(K_v' \cup \{v\})_{< v}\) because equality cannot hold by hypothesis and \(K_v' \subseteq V_{< v}\). Let \(z\) be the smallest vertex in \(\text{complete}(K_v' \cup \{v\})_{< v} \setminus K_v'\). Since \(z\) has been selected by function \text{complete}, we have \(z \in N(K_v' \cup \{v\})\), that is \(z \in N(K_v') \cap N(v)\). Since \(z < v\), we have \(N(K_v') \cap N_{< v}(v) \supseteq \{z\} \neq \emptyset\). \(\Box\)

Letting \(D = \text{complete}(K_v' \cup \{v\})\), it immediately follows that the checks are equivalent in \textsc{spawn} and \textsc{improved-spawn}.

\textbf{Lemma 3.13.} \textsc{child-exists} in \textsc{improved-spawn} satisfies the condition \(N(K_v') \cap (B_K \cup N_{< v}(v)) = \emptyset\) if and only if \(\text{complete}(K_v') = K\) and \(D_{< v} = \text{spawn}(K)\).\[3.13\]

Using Lemmas \[3.8\], \[3.9\], \[3.10\], and \[3.13\], we finally obtain the following result:

\textbf{Lemma 3.14.} \textsc{spawn} in Algorithm \[3\] and function \textsc{improved-spawn} in Algorithm \[4\] are equivalent.

\textit{Proof.} Setting \(D = \text{complete}(K_v' \cup \{v\})\) we observe that \textsc{improved-spawn} \((K)\) simulates the preorder traversal of the recursion tree induced by \textsc{spawn} \((K)\). When all the children of the current maximal clique \(K\) have been explored during the traversal, \textit{childless} is set to true. In this case, if \(K\) is the root of the current tree there are no more maximal cliques to be generated with the given head \(v_i\); otherwise, the state of the parent is restored. \(\Box\)
3.6 Implementation and analysis

Analogously to Section 3.4, the delay of Algorithm 4 is the sum of the running times of all the iterations corresponding to the same \( K \). Specifically, the number of iterations of the while loop is \(|\text{CAND}| \leq \min\{q\Delta, n\}\). The space usage of IMPROVED-SPAWN, due to its statelessness, corresponds to that of a single iteration. In order to give space and time bounds, let us analyze the costs of our navigation primitives.

Lemma 3.15. \( \text{IS-ROOT}(K) \) and \( \text{PARENT-STATE}(K) \) take \( \tilde{O}(\min\{q\Delta,m\}) \) time and \( O(q) \) space.

Proof. Since computing \( \text{pt}(K) \) dominates the space and time costs of functions \( \text{IS-ROOT} \) and \( \text{PARENT-STATE} \), we use the bounds in Lemma 3.6.

Lemma 3.16. \( \text{GET-NEXT-CAND} \) uses \( \tilde{O}(q) \) amortized time\(^7\) and constant space.

Proof. The next candidate can be obtained from \( K \) and the bookmark \( v \) as follows: for each element \( x \in K \), perform a binary search in \( N(x) \) to obtain the smallest vertex greater than \( v \); the minimum among the latter vertices is either the next candidate or a vertex of \( K \); in the latter case we repeat the operation. As there are \( O(q) \) vertices in \( K \), the additional cost of repeating this operation over an iteration of the candidates is \( \tilde{O}(q^2) \), which does not impact the amortized cost of a query.

We need a careful analysis of \( \text{CHILD-EXISTS} \), which provides the dominating cost of IMPROVED-SPAWN. Function \( \text{CHILD-EXISTS} \) makes use of the sets \( N(K'_v) \) and \( B_K \), where \( N(K'_v) \) can take \( \Omega(\Delta) \) space and \( B_K = \{v \in V \setminus K : K_{<v} \subseteq N(v)\} \) can be of size \( \Omega(n) \). Due to our memory constraints, we cannot store them explicitly, so we need to iterate without materializing them.

Iterating over \( N(K'_v) \). We first address \( N(K'_v) \), and show how to iterate over it using constant space. Let \( w \) be the vertex of \( K'_v \) with lowest degree. We scan the vertices \( z \in N(w) \) and, for each of them, we check whether \( K'_v \subseteq N(z) \): those \( z \) satisfying the latter condition form the set \( N(K'_v) \), so we just need to record which one we selected and find the next one by repeating this scan each time. The cost of finding the next one is \( \tilde{O}(|N(w)||K'_v|) = \tilde{O}(\sum_{y \in K'_v} |N(y)|) \), since \( w \) is the lowest degree element of \( K'_v \).

Lemma 3.17. The total cost of iterating over all of the sets \( N(K'_v) \) in constant space, for \( v \in \text{CAND} \), is \( \tilde{O}(d\min\{q\Delta,m\}) \) time.

Proof. Let \( C_K = \bigcup_{x \in K} N_>(x) \supseteq \text{CAND} \). Since the cost of iterating over a given \( N(K'_v) \) is bounded by \( \tilde{O}(\sum_{y \in K'_v} |N(y)|) \), where \( K'_v \equiv K_{<v} \cap N(v) \), the following steps provide an upper bound to the cost, proving the lemma. We drop the big-oh notation for clarity, and use binary indicator variables \( I_P \), such that \( I_P = 1 \) if and only if property \( P \) is true, observing that \( I_{\{y \in N_>(v)\}} = I_{\{v \in N_>(y)\}} \).

\(^7\)The amortized costs for all queries of the current node contribute to its worst-case time for the delay.
\[
\sum_{v \in C_K} \sum_{y \in K \cap N(v)} |N(y)| = \sum_{v \in C_K} \sum_{y \in K \cap N(v)} |N(y)|
\]

\[
= \sum_{v \in C_K} \sum_{y \in K} |N(y)| I(y \in N_K(v))
\]

\[
= \sum_{y \in K} \sum_{v \in C_K} |N(y)| I(v \in N(y))
\]

\[
= \sum_{y \in K} |N(y)|
\]

\[
= \sum_{y \in K} |N(y)| \cdot |N(y)|
\]

\[
\leq d \sum_{y \in K} |N(y)| \leq d \min\{q\Delta, m\}
\]

Iterating over \(B_K\). We characterize \(B_K\) in an alternative form that allows us to efficiently iterate over it.

**Lemma 3.18.** The set \(B_K\) is equal to \(V_{<\min(K)} \cup \{u \in N_{>(\min(K))} \setminus K : K_{<u} \subseteq N(u)\}\).

**Proof.** The vertices in \(V_{<\min(K)}\) are in \(B_K\) as \(K_{<\min(K)} = \emptyset\), thus they can be represented implicitly. All the other vertices must clearly be forward neighbors of \(\min(K)\). \(\square\)

We denote the non-trivial part of \(B_K\) in Lemma 3.18 as \(B'_K = \{u \in N_{>(\min(K))} \setminus K : K_{<u} \subseteq N(u)\}\).

**Lemma 3.19.** Scanning \(B'_K\) can be done in \(\tilde{O}(qd)\) time using \(O(q)\) space.

**Proof.** We can scan \(B'_K\) by testing \(|N_{>(\min(K))}| \leq d\) candidates in \(O(|K|) = O(q)\) time. For this, we need to keep \(K\) and a constant number of memory words. \(\square\)

Using Lemmas 3.18 and 3.19 we can address two possible ways of managing \(B_K\), namely, partially storing it or fully scanning over it, which lead to two alternative bounds (Lemmas 3.20 and 3.21).

**Partially storing \(B_K\).** By applying Lemma 3.18 since we can check in constant time whether a vertex is in \(V_{<\min(K)}\), we only need to store \(B'_K\), whose size is \(O(d)\) by definition. This dominates the space requirement. The computation of \(B'_K\), which costs \(\tilde{O}(qd)\) time by Lemma 3.19 is done once for \(K\) and once for each child. Hence its cost is payed at most twice for each maximal clique.

Observe that the cost of \(\text{CHILD-EXISTS}\) is dominated by that of the computation of \(N(K'_v)\), since testing \(N(K'_v) \cap B_K = \emptyset\) and \(N(K'_v) \cap N_{<}(v) = \emptyset\) can be done in \(\tilde{O}(1)\) time for each element of \(N(K'_v)\).

Applying Lemma 3.17 we obtain the following bound.

**Lemma 3.20.** Function \(\text{CHILD-EXISTS}\) can be implemented such that the cumulative cost of all the calls to \(\text{CHILD-EXISTS} (K, v)\) for a given \(K\) is \(O(d)\) space and \(\tilde{O}(d \min\{q\Delta, m\})\) time.
**Fully scanning** $B_K$. We can compute the intersection between $N(K'_v)$ and $B_K$ by iterating over both sets. The iterator for $B'_K$ works as described in Lemma 3.19. Since the elements of both $N(K'_v)$ and $B'_K$ are iterated in increasing order, the cost of computing their intersection for each call of CHILD-EXISTS is the sum of the costs of the two iterations. Consider the sum of these costs over all the calls of CHILD-EXISTS: applying Lemmas 3.17 and 3.19, we obtain a total cost of $\tilde{O}(d \min\{\Delta, m\} + qd|m\text{CAND}|) = \tilde{O}(d \min\{\Delta, m\} + qd \min\{qd, n\})$.

As a result, we have the following lemma.

**Lemma 3.21.** Function CHILD-EXISTS can be implemented such that the cumulative cost of all the calls to CHILD-EXISTS $(K, v)$ for a given $K$ is $O(q)$ space and $\tilde{O}(d \min\{\Delta, m\} + qd \min\{qd, n\})$ time.

The final algorithm corresponds to plugging Algorithm 4 into Algorithm 3, i.e., replacing SPAWN with IMPROVED-SPAWN. In order to show that the final algorithm takes $O(q)$ or $O(d)$ space as well, we should also perform the setup described in Section 3.7.2. Using Algorithm 5 as setup, and applying Fact 3.2 and Lemmas 3.13, 3.16, 3.20, and 3.21, we obtain our final result, recalling that $q - 1 \leq d \leq \Delta \leq n - 1 \leq m$ and $d = O(\sqrt{m})$.

**Theorem 3.2.** Let $G$ be an undirected connected graph with $n$ vertices and $m$ edges, whose adjacency lists are ordered and whose vertices are labeled in degeneracy ordering. Let $\Delta$ be the maximum degree of its vertices, $q$ the size of its largest clique, and $d$ its degeneracy. Then there exists an algorithm that lists all the maximal cliques in $G$ that has $O(m)$ setup time, $O(q)$ space usage, and $\tilde{O}(q d(\Delta + qd))$ delay. The latter bound improves to $\tilde{O}(qd\Delta)$ if space usage is increased to $\tilde{O}(d)$. Space is always sublinear in the size of $G$.

### 3.7 Setup

A degeneracy ordering can easily be computed in $O(m)$ time by iteratively removing the minimum-degree vertex. This process however takes $O(n)$ additional space. As in this chapter we aim at using sublinear space (i.e., $O(q)$, or at least $O(d)$), we describe here how, given the input graph in the form of ordered adjacency lists, we can compute the degeneracy ordering of the vertices with $O(1)$ additional space, tweaking the graph so that the incremental labelling of the vertices is a degeneracy ordering itself.

Note that this contradicts the assumption that the input graph is read-only: however, the input is only modified during this phase, which is completely separate from the clique enumeration phase.

Furthermore, we describe how to modify the degeneracy ordering of the vertices so as to place the heads at the beginning in $\tilde{O}(m)$ time and $O(1)$ space. As already noted, the resulting order is still a degeneracy ordering.

#### 3.7.1 Computing the degeneracy ordering in constant space

Let the $n$ nodes of the graph be labeled as $0, 1, \ldots, n - 1$ and represented as ordered adjacency lists. As a minimal memory representation we assume the adjacency lists to be a simple array of length $O(m)$ with an additional array of pointers which denote the beginning of each neighborhood (the end of $i$’s neighborhood can be found by looking at the beginning of $i + 1$’s neighborhood).

A degeneracy ordering can be computed as an elimination ordering in which the minimum degree vertex of the graph is recursively removed.

We show how to relabel and reorder the graph in place (with constant additional memory) so that the new labeling $0, 1, \ldots, n - 1$ is a degeneracy ordering, in $O(mn)$ time and $O(1)$ space.

Let $i$ represent initially the first node (i.e., $i = 0$).
1. Find \( j \), the node greater than \( i \), with the smallest amount of neighbors larger than \( i \).

2. Swap \( i \) and \( j \) in the ordering.

3. Increment \( i \) and repeat until \( i = n - 1 \).

The first task can trivially be done in \( O(m) \) time and constant space by simply reading the adjacency list of each node, and keeping a counter for the neighbors greater than \( i \).

The second task consists in swapping the adjacency lists of \( i \) and \( j \) and relabelling all their occurrences in the graph.

This can be done in \( O(m) \) as follows: imagine the adjacency lists of all the nodes between and including \( i \) and \( j \) as an array, and let \( \delta_i \) and \( \delta_j \) be the degrees of \( i \) and \( j \) respectively, and \( \delta_{\text{diff}} \) be \( \delta_j - \delta_i \). The swap operation can be done by first moving the last \( \delta_j \) elements to the beginning of the string, then the elements from the \((\delta_j + 1)\)th to \((\delta_j + \delta_i)\)th to the end of the string, which takes time linear in the size of the string, i.e., \( O(m) \); then we update the start of the pointers of all nodes between \( i \) and \( j \) (including the node labelled \( j \) after the swap), incrementing them by \( \delta_{\text{diff}} \), which takes \( O(n) \). Finally, in \( O(m) \) we can scan all adjacency lists and replace occurrences of \( i \) and \( j \). As up to 2 elements are changed in each list, so updating the ordering of the adjacency lists takes \( O(m) \) time, as we pay linear time for each list. The additional space usage is just the space required to keep track of \( i \), \( j \), and unitary swaps of elements, that is \( O(1) \).

Finally, the process is repeated \( O(n) \) times, for a total cost \( O(mn) \).

### 3.7.2 Computing the heads

The main idea is to conceptually maintain a sliding window over the array of adjacency lists: it contains the array’s entries from position \( s + 1 \) to \( e \), hosting the heads met so far. The window is initially empty (\( s = e = n \)); then, it incrementally collects the heads while moving from the last vertex to the first one in the ordering, thus with a backward scan. The window is shifted one position back by swapping its preceding vertex \( s \) with its last vertex \( e \). In case \( s \) is a head, the window is not shifted but simply enlarged to include \( s \). While shifting the window, the occurrences of the labels of the swapped vertices \( s \) and \( e \) must be updated accordingly. To this aim just the occurrences of \( s \) in \( N_x(v) \) are relabeled, for each neighbor \( v \) of \( s \). At the end, the backward neighborhoods can be updated by looking at the forward ones. The pseudocode for this in-place computation is shown in Algorithm 5.

The proof of correctness is based on the next invariant, where we say that an adjacency list \( L_x \) is correctly labelled if \( L_x \) is sorted and \( L_x = N(x) \) with the current labelling of the vertices.

**Lemma 3.22.** The following conditions are met at the end of each iteration of the for loop in line \( 2 \):

1. if \( x < s \) then \( L_x \) is correctly labelled.
2. if \( x \geq s \) then \( L_x \) is a correctly labelled adjacency list in which all entries smaller than \( x \) have been replaced with 0s.
3. if \( s \leq x \leq e \) then \( x \) is a head.
4. if \( x > e \) then \( x \) is not a head.

**Proof.** If \( s \) is a head, the properties still hold as \( G \) is unchanged, otherwise the following cases occur.

---

*The borderline case \( s = e \) is handled like the other cases, as it means that no heads are met so far and thus the empty sliding window shifts one position back.*
Algorithm 5: In-place algorithm for moving heads to the beginning of $V$

**Input:** $G = (V,E)$, with $V$ in degeneracy ordering and $E$ as ordered adjacency lists $L_1 \ldots L_n$.

**Output:** $G = (V,E)$ with $V$ relabeled in degeneracy ordering so that the $j$ heads are $v_1, \ldots, v_j$.

1. $e \leftarrow n$
2. for $s = n, n-1, \ldots, 1$ do
   // reorder, relabel $N_>(v)$
   3. if $s$ is not a head then
      foreach $v$ in $L_s$ such that $v < s$ do
         replace $s$ with $e$ in $L_v$
         replace $v$ with 0 in $L_s$
         swap $L_s$ and $L_e$
         $e \leftarrow e - 1$
   4. for $v = n, n-1, \ldots, 1$ do
      // relabel $N_<(v)$
      foreach $x$ in $L_v$ such that $x \neq 0$ do replace the rightmost 0 in $L_x$ with $v$

1. If $x < s$ is a neighbor of $s$, then line 5 correctly replaces $s$ with $e$ in $L_x$. The list is still sorted because none of the vertices between $s$ and $e$ are neighbors of $x$.

2. If $x \geq s$ then no vertex in $N_>(x)$ is relabeled during the loop. Vertices in $N_<(x)$ were already set to 0, unless for $x = s$: in the latter case they are set in line 6.

3. We know that before each iteration the vertices between $s + 1$ and $e$ are heads, and since $e$ is swapped with $s$, the vertices between $s$ and $e - 1$ are the heads met so far: after that, we set $e \leftarrow e - 1$ for this reason, so that when the for loop increases $s$ we preserve the invariant.

4. If $x > e$ then $x$ is unchanged, including the element soon after the window that is swapped in line 7 with $s$, which is not a head.

When the first loop terminates, it follows from the last two conditions that the heads are all at the beginning of $V$, thus the ordering is correct. Moreover, from the second condition we have that all the forward neighbor lists are correct and all the backwards neighbor lists are filled with 0s. Labeling the backward neighbors lists is now a trivial task, which is done by the second loop (line 9), as the 0s form a contiguous run in each sorted list.

As the relative order of all non-heads vertices is unchanged, and heads are only moved backwards (i.e. they are never swapped with a neighbor), this ordering preserves the needed properties of the original degeneracy ordering, that is $|N_>(v)|$ is unchanged for all $v$.

**Time and space cost.** The time needed by Algorithm 5 is $\tilde{O}(m)$. Indeed, in the loop in line 2 each adjacency list is iterated at most once, and each iteration just requires searching on a sorted list. The loop in line 9 has the same time bound, i.e. $\tilde{O}(m)$: it traverses the adjacency lists of all vertices performing replace operations, which can be done in logarithmic time on sorted lists. The constant space usage follows from the fact that just $O(1)$ variables are employed.
Algorithm 6: Algorithmic structure of Bron-Kerbosch

Input: A graph \(G = (V,E)\).
Output: All maximal cliques in \(G = (V,E)\).

1. \textsc{Bron-Kerbosch}(\(\emptyset, V, \emptyset\))
2. \textbf{Function} \textsc{Bron-Kerbosch}(\(C, P, X\))
3. \textbf{if} \(P \cup X = \emptyset\) \textbf{then} output \(C\)
4. \textbf{foreach} \(v \in P\) \textbf{do}
5. \quad \textsc{Bron-Kerbosch}(\(C \cup \{v\}, P \cap N(v), X \cap N(v)\))
6. \quad \(P \leftarrow P \setminus \{v\}\)
7. \quad \(X \leftarrow X \cup \{v\}\)

3.8 Exponential delay in Bron-Kerbosch based algorithms

In this section we provide some constructive lower bounds for the delay and the cost per solution for the algorithms falling within the Bron-Kerbosch scheme, namely the algorithm by Tomita, Tanaka and Takahashi [TTT06] and Eppstein, Löffler and Strash [ELS13], which are the state of the art (see Section 3.2). For brevity, we may refer to [TTT06] as \(\texttt{TTT}\) and to [ELS13] as \(\texttt{ELS}\).

First, we give a basic explanation of Bron-Kerbosch based algorithms. Then we show a lower bound on the delay of \(\texttt{ELS}\) and on the cost per solution of both \(\texttt{TTT}\) and \(\texttt{ELS}\). We will use the standard notation which is summarized in Section 3.3.

As shown in Algorithm 6, the structure of Bron-Kerbosch consists of a recursive, DFS-like, backtracking traversal of the input graph, based on three sets: \(C\) stores the temporary result; \(P\) stores the candidate vertices which can be added to \(C\); and finally \(X\), needed to avoid duplication, stores vertices which have been already visited, i.e., added to \(C\) in previous recursive calls. Invoking the algorithm on sets \((C, P, X)\) will output all cliques, maximal in \(G\), which contain all vertices of \(C\), one or more vertices of \(P\) (if \(P \neq \emptyset\)), and no vertices of \(X\). The algorithm is further optimized with the use of a pivot vertex \(u\), chosen arbitrarily in \(P \cup X\): line 4 will consist of an iteration on just vertices in \(P \setminus N(u)\), since any clique which contains \(u\) can be found by adding \(u\) to \(C\); all other cliques require adding to \(C\) a non-neighbor of \(u\).

\(\texttt{TTT}\) is an implementation of Algorithm 6 with a specific pivot choice: the selected pivot is the vertex \(u\) with highest rank, defined as \(|P \cap N(u)|\). This is a greedy choice for the pivot, since it minimizes the number of recursive calls done at this step. In few words, \(\texttt{ELS}\) consists in computing a degeneracy ordering \(\{v_1 \ldots v_n\}\) of the graph and running \(\texttt{TTT}\) on the \(n\) subgraphs induced by \(N(v_i) \cup \{v_i\}\), each time putting all vertices in \(N_C(v_i)\) directly in the \(X\) set.

3.8.1 Delay

Consider the graph class represented in Figure 3.3. It consists of a Moon-Moser graph \([\text{MM}65]\) with \(3k\) vertices\(^9\) (\(\text{k-MM}\) hereafter), a clique \(K_{3k}\) of size \(3k\), and the 8 numbered vertices connected as shown in the figure. The edges between numbered vertices and \(\text{k-MM}\) should be interpreted as that each numbered vertex is connected to all vertices of \(\text{k-MM}\), and the same applies to edges incident to the clique. It is worth noting that each of the numbered vertices is only adjacent to another numbered vertex, and that the total number of vertices is \(n = 6k + 8\).

We will now show that \(\texttt{ELS}\) has exponential delay \(\Omega(3^k) = \Omega(3^{n/6})\) when ran on this graph.

\(^9\)Also known as a Turán graph with parameters \((3k, k)\), where there are \(k\) groups of vertices, and each group is an independent set of 3 vertices. An edge connects two vertices if and only if they belong to different groups, so vertex degree is \(3k - 3\). This graph has \(3^k\) cliques of size \(k\) each.
Figure 3.3: A graph class in which the algorithm in [ELS13] has delay $\Omega(3^n/6)$

Let us consider the degree of the vertices in the graph: $3k + 1$ for vertices $1, \ldots, 4$, $3k + 3$ for vertices in $K_{3k}$, $3k + 5$ for vertices of K-MM and $6k + 1$ for vertices $5 \ldots 8$. Consider a degeneracy ordering of the graph. It is straightforward to see that the first 4 vertices of the ordering $v_1 \ldots v_4$ will be the numbered vertices $1 \ldots 4$ (in any order), and the fifth ($v_5$) a vertex of the K-MM; the rest is not relevant to our purpose.

Note that each clique of K-MM can be extended with one arbitrary vertex among $1 \ldots 4$ (and one among $5 \ldots 8$), thus when processing vertices $v_1 \ldots v_4$ ELS will find all such cliques. Consider now the processing of $v_5$: $X$ will be $\{v_1 \ldots v_4\}$ and $P$ will be K-MM’s vertices, plus $\{5, 6, 7, 8\}$, minus $v_5$. The pivot will be chosen in $P$ as those vertices have a higher rank $(3k + 1)$ than vertices in $X$ $(3k - 2)$. The difference in rank will remain constant as $P$ shrinks. Each recursive call, thus, will have 3 children (the pivot and the 2 members of the same independent set), and the depth of the tree will be $k - 1$, for a total number of $\Omega(3^k) = \Omega(3^n/6)$ recursive calls. As all cliques that contain $v_5$ also contain one vertex in $v_1 \ldots v_4$, no new cliques are found in this process, giving us the following lemma.

Lemma 3.23. There exists an infinite family of graphs for which the algorithm by Eppstein et al. [ELS13] has delay $\Omega(3^n/6)$, where $n$ is the number of vertices in the graph.

3.8.2 Cost per solution

We discuss the cost of TTT and ELS on a complete graph.

Lemma 3.24. The algorithm by Tomita et al. [TTT06] has cost $\Omega(n^3)$ on a complete graph $G = (V, E)$ with $|V| = n$.

Proof. Each call of TTT chooses a pivot in $P \cup X$. Observe that $X$ remains empty and size of $P$ decreases by exactly one at each step because $G$ is complete. At each step, the pivot is chosen as the vertex $u$ which maximizes the rank: this is done computing for each vertex $u$ in $P \cup X = P$ the rank $|P \cap N(u)|$, for a total cost of $\Omega(|P| \min\{|P|, |N(u)|\}) = \Omega(|P| \min\{|P|, n - 1\}) = \Omega(|P|^2)$.

Hence, the cost of the root-to-leaf path in the recursion path is $\Omega\left(\sum_{|P|=1}^{n-1} |P|^2\right) = \Omega(n^3)$. \qed

Considering that the first iteration of ELS on a complete graph will call TTT on a subgraph which contains a vertex and its neighbors (i.e. the whole graph), the following holds.

Lemma 3.25. There exists an infinite family of graphs with $n$ vertices, for which the algorithms by Eppstein et al. [ELS13] and Tomita et al. [TTT06] have cost per solution $\Omega(n^3)$.
<table>
<thead>
<tr>
<th>ALGO.</th>
<th>TIME</th>
<th>DELAY</th>
<th>MEM</th>
<th>TIME</th>
<th>DELAY</th>
<th>MEM</th>
<th>TIME</th>
<th>DELAY</th>
<th>MEM</th>
<th>TIME</th>
<th>DELAY</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sec</td>
<td>ms</td>
<td>MiB</td>
<td>sec</td>
<td>ms</td>
<td>MiB</td>
<td>sec</td>
<td>ms</td>
<td>MiB</td>
<td>sec</td>
<td>ms</td>
<td>MiB</td>
</tr>
<tr>
<td>CN</td>
<td>&gt;2h</td>
<td>475</td>
<td>97.56</td>
<td>&gt;2h</td>
<td>636</td>
<td>78.26</td>
<td>&gt;2h</td>
<td>5285</td>
<td>258.21</td>
<td>&gt;2h</td>
<td>4077</td>
<td>164.54</td>
</tr>
<tr>
<td>MU</td>
<td>39.5</td>
<td>18</td>
<td>3.01</td>
<td>16.9</td>
<td>22</td>
<td>3.01</td>
<td>6102.7</td>
<td>3691</td>
<td>52.62</td>
<td>&gt;2h</td>
<td>11395</td>
<td>30.52</td>
</tr>
<tr>
<td>CXQ</td>
<td>21.2</td>
<td>4</td>
<td>0.11</td>
<td>60.7</td>
<td>12</td>
<td>0.25</td>
<td>&gt;2h</td>
<td>6004</td>
<td>1.02</td>
<td>&gt;2h</td>
<td>621</td>
<td>3.15</td>
</tr>
<tr>
<td>RALG2</td>
<td>1.8</td>
<td>0.6</td>
<td>1.57</td>
<td>3.1</td>
<td>0.8</td>
<td>0.81</td>
<td>66.6</td>
<td>401</td>
<td>6.46</td>
<td>237.4</td>
<td>245</td>
<td>3.16</td>
</tr>
<tr>
<td>ALG2</td>
<td>2.3</td>
<td>15</td>
<td>0.01</td>
<td>4.4</td>
<td>0.9</td>
<td>0.01</td>
<td>100.7</td>
<td>410</td>
<td>0.03</td>
<td>363.8</td>
<td>277</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 3.2: Experimental results of our comparison with output-sensitive algorithms in the state of the art. For description of the parameters (upper part), we refer to Table 3.1. For the comparison (lower part) we have considered the total time (TIME), the delay (DELAY), and the additional space (MEM).
3.9 Experimental evaluation

We tested our algorithms on some large real-world networks that have been taken from LASAGNE [LoAoGN]. Our computing platform is a 24-core machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz, with 128GB of shared memory. The operating system is a Ubuntu 14.04.2 LTS, with Linux kernel version 3.16.0-30. The code has been written in C++ and forced to run on a single core.

Table 3.2 reports the results of the comparison between our algorithms and existing algorithms which use at most linear space and are output-sensitive, i.e. the ones providing a bounded delay or cost per solution, namely CN, MU, and CXQ (see Table 3.1). We limited the running time to two hours. Algorithm RALG2 refers to a recursive version of Algorithm 4. ALG2 refers to Algorithm 4 which takes $O(q)$ memory by using Lemma 3.21 (similar results can be found for the iterative version which uses Lemma 3.20). For each algorithm and each graph, we report the total time, the maximum delay found while running, and the memory usage (excluding the input size). Note that both RALG2 and ALG2 are significantly faster than CN, MU, and CXQ. It is worth observing that the results highlight how the delay of RALG2 and ALG2 depends on $d$ and $q$ as shown in Theorem 3.2.

The running times of ALG2 are in general higher than RALG2, even though its performance is competitive. On the other hand, ALG2 uses the smallest amount of memory, namely always less than 0.03 MiB even when these graphs have hundreds of thousands of vertices and millions of edges. The most striking result is for EU-2005, having more than 850 thousands of vertices and more than 16 millions of edges, where our algorithm uses just 0.02 MiB. Finally, observe that the memory seems to be proportional to $d$ and $q$, since IN-2004 and EU-2005 have relatively higher memory consumption.

For the sake of completeness, we also considered the state of the art for Bron-Kerbosch based algorithms (e.g. see Algorithm 6). In particular, we ran the algorithm in [ELS13], using the original source code kindly provided by the authors David Eppstein, Maarten Löffler, and Darren Strash. Even though its cost per solution can be higher than ours and its delay can be exponential, this algorithm is on the average 3.7 times faster than ALG2. On the other hand, as this algorithm uses linear memory, its memory consumption is on the average 878.9 times larger than the memory of ALG2. Interestingly, our experiments show that reverse search is not only a theoretical framework, but its careful implementation can compete with the algorithm engineering solutions offered by the state of the art.

3.10 Reducing the total space usage below the linear barrier

Some real-world networks have up to billions of vertices and edges, and in-memory approaches become unfeasible. In this scenario, several strategies have been developed to reduce the space usage below the size of the dataset.

Several consist in decomposing the network into parts that are independently processed in a distributed environment [CKF+10, CZKC12, DWX+06, GLM13, SK12, XGA13, XCFB14]. In some cases, such as XCFB14, the need for a distributed environment can be avoided by storing the graph in larger but slower external memory, and retrieving each part by repeatedly scanning the stored graph.

In order to be processed independently and in-memory, parts need to be small. However, in real-world networks, this may not be feasible: in particular, the mentioned approaches will produce blocks whose size is lower bounded by the degree of a vertex.

For example, the basic idea in XCFB14 is to select from the input graph $G(V,E)$ a vertex $v \in V$, extract the induced subgraph $G[\{v\} \cup N(V)]$, and find all maximal cliques that contain $v$ and do not contain any vertex smaller than $v$. It is straightforward to see that every maximal
clique will be found this way exactly once. While the approach can be very effective on sparse graphs, it is not suitable for real-world network which exhibit scale-free properties: as we remarked in Section 2.5 real-world networks may contain hubs, i.e., vertex with extremely high degree. The subgraph induced by the neighbors of a hub may have size comparable to that of the graph itself, which significantly diminishes the memory benefits of the decomposition.

A possible solution to this issue is proposed in [CVM+16b].

3.11 Final remarks

In this chapter we presented the first algorithm for enumerating maximal cliques using sublinear space, still guaranteeing a bounded delay. Our approach uses stateless reverse search to avoid storing the state of the recursion by rebuilding it on the fly. We employ a degeneracy ordering of the vertices to bound the candidate vertices to be analyzed and we use stricter conditions to limit the cost of analyzing unfruitful candidates. Our algorithm requires \( \tilde{O}(m) \) setup time and is given in two versions: the former has delay \( \tilde{O}(\min\{qd(\Delta + qd), md\}) \) using \( O(q) \) space, while the latter has delay \( \tilde{O}(\min\{qd\Delta, md\}) \) using \( O(d) \) space. The performance of both versions is competitive with the state of the art, with the additional benefit of providing worst-case guarantee on delay and space usage.

A natural question is whether these bounds can be improved further and what is the minimum delay, or cost per solution, that a maximal clique enumerator can achieve, and while this algorithm gives its best on very large, very sparse graph, it would be interesting to analyze and optimize its performance to denser graph or graphs containing dense areas. Moreover, it is also natural to wonder whether the techniques and the optimizations we used could be applied to other enumeration problems. We consider this final question in the following chapter, Chapter 4, by investigating the enumeration of maximal independent sets, and later in Chapter 5, where we attempt at designing general algorithms for a wide range of problems.
Listing maximal independent sets with minimal space

The independent set is a basic graph structure, and it is closely related to cliques: it is a set of vertices such that no pair of them is adjacent. It is maximal if there is no vertex outside the independent set that may join it. Listing maximal independent sets in graphs can be applied, for example, to sample vertices belonging to different communities or clusters in network analysis and document clustering. The problem has a rich history as it is related to maximal cliques, dominance sets, vertex covers and 3-colorings in graphs.

It is straightforward to see that independent sets in a graph correspond to cliques in its complement graph. As in Chapter 3 we describe new techniques for listing maximal cliques with bounded delay and sublinear space, a natural question is asking whether the same techniques can be applied to independent sets.

In this chapter we show that trivial adaptation of the algorithms in Chapter 3 is not satisfactory, as delay and space deteriorate significantly. However, we show how to leverage on the same design principles to obtain an efficient algorithm which uses $O(s)$ additional space, where $s$ is the size of the largest independent set, and whose delay is competitive with current approaches. Furthermore, we are able to provide a better bound on the delay by increasing the additional space usage to $O(n)$.

4.1 Introduction

Given an undirected graph $G = (V(G), E(G))$, a maximal independent set (MIS) $I \subseteq V(G)$ does not contain any two vertices connected by an edge, and is maximal under inclusion (no other $I' \supset I$ has this property). We pose the question whether listing MISs can be achieved efficiently in small space and bounded delay.

Although this problem originated in graph theory, as MISs are related to dominance sets, vertex covers and 3-colorings in graphs, we observe that data is networked in information systems nowadays. The classical problem of looking at patterns in texts or sequences, or trees, can be translated into graphs: the algorithmic techniques are different, and even the simple query asking if a path occurs in a graph is NP-hard. Nevertheless, discovering patterns in sequences and patterns in graphs are quite similar tasks, and can share techniques in some cases. MISs, in particular, can be seen as a way to build samples that are independent from each other, thus motivating the question.

One possible field of application is in networks analysis, such as social science, where a MIS identifies a group of persons from a tightly connected community that are isolated from each
other, or can be used as sample for communities, where each vertex of the MIS is a person from a different community.

MISs are a powerful tool for clustering: they can be used for clustering document collections (where two documents are linked if their content is similar), by using a MIS as a collection of starting points for the chosen clustering method or for clustering wireless networks to identify hierarchical structures [Bas01]. Moreover, they are often used to build efficient indexes for answering shortest path or distance queries (see for instance [FWCW13]). MISs are applied for clustering purposes also in image segmentation, that aims at grouping image pixels into visually “meaningful” segments. In this case, the goal is to select the segments of an image that are distinct, and together partition the image area. In a graph where segments are vertices and edges correspond to the overlap of the segments, all the maximal independent sets correspond to all the non-overlapping segment partitions. [BT10] studied the maximum weighted independent set (MWIS) to get the maximally distinctive partitions by encoding a distinctiveness scoring of the segments into the vertices weights. This approach was also extended to clustering aggregation in general [LL12, OCDV14] modeled co-occurrences of words and documents in the web as a graph, and used MWIS’s in this graph to find sets of important but distinct (i.e., rarely co-occurring) topics. However, the MWIS problem is NP-hard and hard to approximate. Listing all the MISs can also provide an exact solution for the latter problem, eventually testing different distinctiveness scoring systems.

**Our results.** In this paper we describe an algorithm that lists all the MISs with \(O(n\min\{n^d\Delta^2, mn\})\) delay and \(O(s)\) additional space, where \(s\) is the maximum size of a MIS. We assume that the input graph is read-only, and the space complexity is the additional working space.

As it can be seen, the additional space is asymptotically minimal, and the delay can be as low as \(O(n)\) (if \(d\) and \(\Delta\) are \(O(1)\)), but never larger than the baseline of \(O(nm)\) (ignoring logarithmic factors) given by Tsukiyama et al. [TIAS77]. We further reduce our time bound by providing a second algorithm with \(O(n\min\{d\Delta n, mn\})\) delay which increases the memory requirement to \(O(n)\): this simultaneously improves both best known bounds for delay and space as \(d\Delta\) is a pessimistic upper bound on the cost, which is smaller than \(m\) in practice.

**Related work.** Listing MISs is a classical problem in enumeration which dates back at least to the ’70s, with many results such as producing MISs in lexicographical order [LT81], experimentally or with guarantees [JYP88], achieving \(O(n^3)\) delay but using exponential space. Some results have been also proposed for particular classes of graphs: claw-free graphs [Min80], interval graphs, circular-arc graphs and chordal graphs [Leu84, OUU05, OUU08], trees [Leu84], permutation graphs [YC93].

In general, listing the MISs of a graph \(G\) is equivalent to listing the maximal cliques of its complement \(\overline{G} = (V, \overline{E})\). However improved algorithms for maximal cliques, such as the space efficient solution that we presented in Chapter 3 do not translate into improved bounds for listing MISs: the transformation from MISs to maximal cliques is not effective, especially in sparse graphs which have a dense complementary graph, but even in dense graphs, since their complementary graph can be dense too. These techniques mainly fall in the backtracking approach, as for [BK73], or in the reverse search paradigm introduced by [AF96].

In the former case, these approaches are not output sensitive for both cliques and MISs, in the sense that their guarantee on the running time is not related to the number of solutions. In the relatively recent work by Eppstein et al. [ELS13] for cliques, the overall time \(O(dn^{d/3})\) becomes \(O(n^2 \cdot 3^{n/3})\) to list all the MISs, as the degeneracy \(d\) can be \(\Theta(n)\) in the complementary graph, while the delay remains exponential, as in the case of the algorithm in [TTT06]. Moreover, the space usage, without storing the transformed graph, becomes \(O(ns)\) for [TTT06] and \(O(n^2)\) for [ELS13]. On the other hand, while adapting the reverse search for maximal cliques to MISs, the algorithms by Chiba and Nishizeki [CN85], by Makino and Uno [MU04] and Chang et
al. [CYQ13] require $O(n^2 - m)$ space: recalling that arboricity, maximum degree, and degeneracy of the complementary graph can be linear, the delay bound becomes $O(n(n^2 - m))$ for [CN83], and $O(n^4)$ for [MU04] and [CYQ13]. Moreover, as shown next in Section 4.2, the delay bound in Chapter 3 becomes $O(nm)$, which does not improve upon Tskuiyama et al. [TIAS77].

Also, since MISs can be considered a hereditary property or independence set system, they can be listed using the framework of Cohen et al. [CKS08] but the resulting bounds still do not improve over those of [TIAS77]. For these reasons ad hoc algorithms for cliques and MISs have been proposed separately in the literature, and the best output sensitive bounds for MISs are $O(nm)$ delay with $O(n^2)$ space [TIAS77], or $O(n^{2.37})$ delay with $O(n^2)$ space by using matrix multiplication [MU04], or $O(n^{2.09})$ delay with $O(n^{4.27})$ space [CR15].

Our approach. The algorithmic challenges addressed here are related to the reverse search, which is a powerful enumeration technique introduced by Avis and Fukuda [AF96]. Consider the graph-like structure shown in Figure 2.4 (left), called the solution graph: each “cloud” corresponds to a distinct MIS, and an arrow from MIS $I_i$ to MIS $I_j$ with label $v$ means that $I_j$ can be computed from $I_i$ through a vertex $v$, using a rule that is specific for the algorithm at hand. As in other techniques, such as divide and conquer, the algorithmic contribution is the efficient implementation of the generic step, for the problem at hand.

To list all the MISs, we use the rule to traverse the solution graph and output each solution once. An easy way to do so is to keep track of all visited vertices. Even though such methods have been used, e.g. in [JYP88], they are expensive as they require exponential memory. Reverse search avoids this issue by choosing a single parent $I_i$ for each MIS $I_j$, such that $I_i < I_j$ for some given order (such as the lexicographic one), among all MISs leading to $I_j$. This way it induces a directed spanning forest on the solution graph, as illustrated in Figure 3.2 (right). Some MISs have no parent and are the roots of the spanning forest. Note that the solution graph and directed spanning forest are for the purpose of explanation and never materialized. The roots can be easily identified and are at most $n$.

Traversing the solution graph can be implicitly done by performing a DFS: each time we explore the possible children solutions and recur in just the ones whose parent is the current solution, following [MU04]. This visit can be made iterative, by avoiding the stack of the recursion. We can restore the state of the parent when returning from the call to a child. This strategy is particularly useful if we want to achieve sublinear additional memory, since we avoid using memory proportional to the height of the recursion tree, where a single bit per recursion level is too much. Here the techniques in Chapter 3 for maximal cliques do not translate smoothly into improved bounds for MISs, as discussed in Section 4.2.

The complexity of the reverse search is dominated by the cost of applying the rule to the current MIS in the directed spanning forest. Computing the rule is expensive as the time spent checking not fruitful candidate children is completely charged on the delay of the algorithm. Thus we introduce a novel technique that allows us to apply the rule only to the children, rather than to all the out-neighbors in the solution graph. We check a necessary condition, which is lighter to compute than the rule, so that the rule is actually applied to selected out-neighbors. During this task, we use a small amount of space.

4.2 Using state of the art maximal clique algorithms for MISs

We discuss the application of the maximal clique techniques to MISs (see Chapter 3 related work on maximal cliques). The Bron-Kerbosch [BK73] algorithm is a backtracking algorithm which has been originally designed for maximal cliques and in principle does not give any guarantee. The bound for the overall running time given by the revised version in [TTT06], which is $O(3^{n/3})$, also applies to MISs. The algorithm by Eppstein [Epp03] is inspired to the Bron-Kerbosch
algorithm and gives all the MISs of size at most $k$ in $O(3^{4k-n^2}4^{n-3k})$ total time. These bounds are tight when $n/4 \leq k \leq n/3$. The approach, as given in [Epp03], uses $O(n^2)$ additional space; as far as we can see, this could be improved to $O(m)$, by using a compressed representation of the recursion stack. If we apply a more recent algorithm by Eppstein et al. [ELS13] for cliques, the overall time $O(dn3d/3)$ becomes $O(n^2 \cdot 3^{n/3})$ for MISs, as the degeneracy $d$ can be $O(n)$ in the complementary graph. Since the transformation does not improve the delay, for the algorithms above it remains exponential. We remark that the space usage, without storing the transformed graph, becomes $O(ns)$ for [TTT06] and $O(n^2)$ for [ELS13], as the maximum degree and degeneracy of the complementary graph can be $O(n)$.

Bron-Kerbosch based algorithms are not output-sensitive, in the sense that their guarantee on the running time is not related to the number of solutions. This is the case instead for the algorithm by Tsukiyama et al. [TIAS77], which is a backtracking procedure that enumerates MISs in $O(mn)$ time per solution using $O(n^2)$ space. This approach has been improved and adapted to maximal cliques by Chiba and Nishizeki [CN85], then to be revisited in the reverse search paradigm [AF96] by Makino and Uno [MU04] and Chang et al. [CYQ13]. By applying the transformation, all these approaches require $O(n^2 - m)$ space: recalling that arboricity, maximum degree, and degeneracy of the complementary graph can be linear, the delay bound becomes $O(n(n^2 - m))$ for [CN85], and $O(n^4)$ for [MU04] and [CYQ13]. In the same way, one can use the stateless reverse search algorithms in Chapter 3 for maximal cliques, with delay $O(\min\{qd\Delta, md\})$ and additional memory usage $O(d)$ or with $O(qd(qd + \Delta))$ delay and memory $O(q)$, where $q$ is the maximum clique size. Here, the delay of the algorithms becomes $O(\min\{n^2, n^2 - m\})$ or $\tilde{O}(n^2 s^2)$, while the additional memory becomes $\Omega(n^2 - m)$ as we are storing $\overline{G}$. This algorithm can be modified to obtain the similar bounds to [TIAS77], as shown next. Along these lines, all the results shown in Table 3.1 in Chapter 3 extend to MISs, by substituting the parameters of $G$ with the greater ones of $\overline{G}$.

Other approaches using reverse search employ matrix multiplication: in this case, the delay bounds for maximal cliques are preserved for MISs. Hence, one can use the algorithm in [MU04] and the one by Comin and Rizzi [CR15] to list MISs with delay respectively $O(n^{2.37})$ and $O(n^{2.09})$, at the cost of using respectively $O(n^2)$ and $O(n^{4.27})$ space.

Algorithm 7, which we call BASE, represent the direct adaption of the techniques in Chapter 3 to the MIS enumeration problem. All functions are syntactically equivalent to the ones in Algorithm 4 in Chapter 3 although whenever the neighborhood $N(x)$ of a vertex $x$ is considered, this is replaced by the complement neighborhood $\overline{N}(x)$.

We remark that some important optimizations in Chapter 3 for cliques are no more useful and do not bring any benefit for BASE. Indeed, while in Chapter 3 we have $|N_\geq(x)| \leq d$ thanks to the degeneracy ordering and $|N(x)| \leq \Delta$, there bounds do not hold for complementary neighborhoods. Hence, the improvements done to check the existence of a child (see Algorithm 4) does not improve upon the basic child conditions of Makino-Uno based approach [MU04], which is reported in Algorithm 8 (see also Algorithm 3 in Chapter 3). Indeed, one of the most important benefit of Algorithm 4 was the speed up and the memory improvements of this function thanks to the definition of $B_K$ (see Algorithm 4), which can be stored in a $O(d)$ and allows to discard candidates. In the case of MISs, dealing with $B_K$ can be costly, as its size can be $\Theta(n)$ for a generic $K$ in $\overline{G}$. For this reason we need to use a more basic check and the function CHILD-EXISTS is replaced with the equivalent, more basic check used in Algorithm 3.

Lemma 4.1 shows that BASE is correct, showing its equivalence with the algorithms in Chapter 3 with input $\overline{G}$.

**Lemma 4.1.** Algorithm BASE iterates over all maximal independent sets.\[1\]
Proof. We show that running Algorithm 7 (using CHILD-EXISTS in Algorithm 8) on \( G \) will produce the same output as running Algorithm 4 in Chapter 3 on \( \overline{G} \). As Algorithm 4 in Chapter 3 finds all maximal cliques in \( \overline{G} \), which correspond to all MISs in \( G \), this proves the lemma.

It is important to recall that \( \overline{N}(x) \) in \( G \) is equal to \( N(x) \) in \( \overline{G} \). With this in mind it is trivial to see how, for a given independent set \( I \) and vertex \( v \), the functions COMPLETE(I) and GET-NEXT-CAND(I, v) perform the same exact operations as COMPLETE(I) and GET-NEXT-CAND(I, v) in Chapter 3, provided that the input graph is \( \overline{G} \), and thus \( I \) is a clique rather than an independent set. It follows that functions \( \text{PI}, \text{IS-ROOT} \) and CHILD-EXISTS are also equivalent, since they are based on COMPLETE and GET-NEXT-CAND. Thus running Algorithm 7 on \( G \) will give the same output as running Algorithm 4 in Chapter 3 on \( \overline{G} \).

\[ \text{Lemma 4.2.} \] COMPLETE(I) can be computed in \( \tilde{O}(m) \) time.

Proof. Given an independent set \( I \), COMPLETE(I) consists in recursively adding \( \min\{\overline{N}(I)\} \) to \( I \) until \( \overline{N}(I) \) is empty (see [MU04] or Chapter 3 for proof). This can be done by iterating over \( V \) in order, adding each vertex \( v \) if \( N(v) \cap I = \emptyset \). Computing \( |I \cap N(v)| \) takes \( \tilde{O}(|N(v)|) \) using binary searches, so the total cost is the sum of all degrees (times a log), i.e., \( \tilde{O}(m) \) time.

4.4 Listing MISs

Given an independent set \( I \) and a vertex \( v \in V \setminus I \) such that \( I \cup v \neq \emptyset \), formula (4.1) generates a new maximal independent set.

\[ F(I, v) = \text{COMPLETE}(I_v \cup \{v\}) \quad (4.1) \]

As a reverse degeneracy ordering is required, it is sufficient to apply the same technique symmetrically, i.e., starting from the last vertex \( (i = n - 1) \) and considering \( j \) as the vertex smaller than \( i \) with the minimum amount of neighbors smaller than \( i \).
Algorithm 7: BASE, Enumeration of Maximal Independent Sets

Assume \( \min(\emptyset) = \text{null} \), and recall that \( I'_v \equiv I_{\leq v} \cap \mathcal{N}(v) \).
For function CHILD-EXISTS refer to Algorithm 8.

\[
\text{Function } \text{improved-spawn}(I) \\
v \leftarrow \pi(I) \\
\text{while } \text{true} \text{ do} \\
\text{childless } \leftarrow \text{true} \\
\text{while } v \leftarrow \text{get-next-cand}(I,v) \neq \text{null} \text{ do} \\
\text{if } \text{child-exists}(I,v) \text{ then} \\
I \leftarrow \text{complete}(I'_v \cup \{v\}) \\
\text{childless } \leftarrow \text{false} \\
\text{break} \\
\text{if } \text{childless} \text{ then } \text{if } \text{is-root}(I) \text{ then return} \\
\quad \langle I,v \rangle \leftarrow \text{parent-state}(I) \\
\text{else } \text{return} \langle I,v \rangle \\
\text{return } \langle I,v \rangle \\
\]

\[
\text{Function } \text{is-root}(I) \\
\text{return } \pi(I) = \min(I) \\
\]

\[
\text{Function } \text{parent-state}(I) \\
v \leftarrow \pi(I) \\
\text{return } \langle \text{complete}(I_{\leq v}), v \rangle \\
\]

\[
\text{Function } \text{get-next-cand}(I,v) \\
\text{return } \min\{ w \in \bigcup_{u \in I} \mathcal{N}_>(u) \setminus I : w > v \} \\
\]

\[
\text{Function } \text{complete}(I) \\
\text{while } \mathcal{N}(I) \neq \emptyset \text{ do} \\
I \leftarrow I \cup \min\{\mathcal{N}(I)\} \\
\text{return } I \\
\]

\[
\text{Function } \pi(I) \\
\text{return } \min\{ v \in I : \text{complete}(I_{\leq v}) = I \} \\
\]

In the solution graph illustrated in Figure 3.2, there is an arrow \( I_i \rightarrow I_j \) labeled with \( v \) if and only if \( I_j = F(I_i,v) \). The parent-child relationship, which defines the directed spanning forest, is as follows.

\[
\text{parent}(I) = \text{complete}(I_{<\pi(I)}) \\
\tag{4.2}
\]

where \( \pi(I) \), the parent index of \( I \), is the smallest element \( v \in I \) for which \( \text{complete}(I_{\leq v}) = I \). Note that if \( I_k = \text{parent}(I_j) \) and \( v = \pi(I_k) \), then \( I_i, I_j, v \) satisfy formula (4.1).

The roots, which have no parent, can be found as \( \text{complete}\{\{v\}\} \), for any \( v \in V \) such that \( \min\{\text{complete}\{\{v\}\}\} = v \); the number of roots is at most \( n \).

During the traversal of the solution graph, in the current MIS \( I_j \) we compute its parent \( I_i \) and the parent index \( v \), thus restoring the state of the traversal when returning from the call to \( I_j \). Keeping this in mind, the delay per listed MIS is bounded by the maximum amount of time spent in the current MIS, say \( I \), observing that this time is intermixed with the calls to its children.

The delay can be bounded by the cost of (i) computing \( I \) from its parent using formula (4.1); (ii) checking each candidate child to see if a call with formula (4.1) should be applied; finally (iii) if \( I \) is not a root, restoring the state to parent of \( I \) using formula (4.2). Indeed, once we get the costs (i)–(iii), we can employ the well-known alternative output technique in [Uno03], so that the delay is bounded by the costs (i)–(iii) times a constant. We recall that in the alternative output technique, when the level (i.e. the distance from the root) is odd the output is done before exploring the children, otherwise it is done soon after.

In the rest of the paper we give the details for our new two algorithms for listing MISs using the above notions. The first algorithm is presented in Section 4.5 and achieves \( O(s) \) additional memory and \( \tilde{O}(\min\{d\Delta^2 n, mn\}) \) delay. The second algorithm is presented in Section 4.6 and reduces the delay to \( \tilde{O}(\min\{d\Delta n, mn\}) \), albeit using \( O(n) \) space.

One of the core ideas for both algorithms is the efficient computation of the test in point (ii) above. For the sake of simplicity, this behavior is encapsulated by the function CHILD-EXISTS, which is executed for each possible candidate. Our algorithms minimize the space usage by implementing some efficient implicit iterators that avoid building sets explicitly. For instance, the set \( I'_v = I_{\leq v} \cap \mathcal{N}(v) \) in formula (4.1) is never materialized, as its explicit computation is expensive both in terms of time and space.
The cost follows.

In the first case, \( y \) is selected to be added to \( I \). Otherwise, if \( y < v \) for sets which are too costly to compute, and simulate the behavior of \( \text{COMPLETE} \) quickly and using only \( O(s) \) time.

Lemma 4.3. \( \text{PI}(I) \) and \( \text{PARENT}(I) \) can be computed in \( \tilde{O}(m) \) time.

Proof. Given a MIS \( I \), with \( x = \text{PI}(I) \), we have \( \text{PARENT}(I) = \text{COMPLETE}(I_{<x}) \). Furthermore, we know that \( \text{COMPLETE}(I_{<y}) \) is equal to \( I \) if \( y > x \). Thus by computing \( \text{COMPLETE}(I_{<y}) \) we know whether \( y \) is larger than \( x \) or not. We can thus look for \( x \) in a binary search-like fashion. We can thus find \( x \) by performing \( O(\log |I|) \) times a COMPLETE call, to then compute \( \text{COMPLETE}(I_{<x}) \)

The cost follows.

4.5 Using minimal space

In this section we present our first algorithm, whose focus is minimizing the additional memory: the algorithm will only store \( O(s) \) information on top of the input graph while keeping the performance competitive with state of the art approaches. This algorithm aims at improving the cost of \( \text{CHILD-EXISTS} \) of Algorithm 8 using \( O(s) \) space. The improvements are due to two factors.

On one hand we identify stricter theoretical conditions to determine whether \( \text{CHILD-EXISTS} \) will succeed or not, which are used in place of \( \text{CHILD-EXISTS} \); Lemmas 4.4 and 4.5 prove the correctness of these conditions, allowing us to prove the equivalence of Algorithms 8 and 9 in Lemma 4.6.

On the other hand, we provide non-trivial techniques which allow us to compute our theoretical conditions quickly and using only \( O(s) \) additional space: in detail, we provide fast implicit iterators for sets which are too costly to compute, and simulate the behavior of \( \text{COMPLETE} \), stopping it prematurely when suitable conditions are met.

Lemma 4.4. \( \text{COMPLETE}(I'_{<v}) = I'_v \) is equivalent to \( \min\{\overline{N}(I'_{<v})|v|\} > v \).

Proof. Recalling its definition (see Section 4.3), we know that \( \text{COMPLETE}(I) \) adds vertices to \( I \) in increasing order: indeed, at a given step we select the smallest vertex \( x \) in \( \overline{N}(I) \) and add it to \( I \); in the following step \( \overline{N}(I) \) will shrink because we added \( x \) to \( I \), and its minimum cannot be smaller than (or equal to) \( x \).

Let \( y \) be the first vertex selected by \( \text{COMPLETE}(I'_{<v}) \), that is, \( y = \min\{\overline{N}(I'_{<v})|v|\} \). If \( y < v \), then \( \text{COMPLETE}(I'_{<v}) \neq I'_{<v} \) as the earlier set contains \( y \) while the latter does not. Otherwise, if \( y > v \) all other vertices selected by the \( \text{COMPLETE} \) function will also be greater than \( v \); since \( I'_v = I_{<v} \cap \overline{N}(v) \) we thus have \( \text{COMPLETE}(I'_{<v}) \neq I'_v \).

Lemma 4.5. If \( v > \text{PI}(I) \), Algorithm 9 line 7 returns true iff \( \text{COMPLETE}(I'_v) = I \).

Proof. Lines 4-7 of Algorithm 9 correspond to simulating \( \text{COMPLETE}(I'_v) \) until the vertex \( x \) that is selected to be added to \( I'_v \) is not in \( I_{<v} \). Then two cases are possible: either \( x \notin I_{>v} \) or \( x \in I_{>v} \).

In the first case, \( \text{COMPLETE}(I'_v) \neq I \) since \( x \in \text{COMPLETE}(I'_v) \) and \( x \notin I \), so Algorithm 9 returns false. Otherwise, if \( x \in I_{>v} \) we show that all vertices in \( I_{<v} \) that were not in \( I'_v \) have been added
Algorithm 9: Improved CHILD-EXISTS-MS check with $O(s)$ additional memory ($I_v' \equiv I_{<v} \cap N(v)$ is actually not computed explicitly, see Lemma 4.10)

| Input: A maximal independent set $I$ and a vertex $v > p(I)$ |
| Output: true if complete($I_v' \cup \{v\}$) is a child of $I$, false otherwise |

```
1  Function CHILD-EXISTS-MS (I, v) 
2     if not min(\overline{N}(I_v' \cup \{v\})) > v then return false
3     A ← I_v'
4     while \overline{N}(A) ≠ ∅ do
5         x ← min(\overline{N}(A))
6         if x ∈ I_v then A ← A ∪ \{x\}
7         else return x ∈ I
```

We show that, as a result of Lemmas 4.4 and 4.5, we can conclude the following.

**Lemma 4.6.** CHILD-EXISTS-MS in Algorithm 9 can be used in place of CHILD-EXISTS in base.

**Proof.** By Lemma 4.4 we have that CHILD-EXISTS-MS will return false on line 2 iff complete($I_v' \cup \{v\}$)$_{<v} \neq I_v'$. Otherwise, by Lemma 4.5, CHILD-EXISTS-MS will return true if complete($I_v'$) = $I$, and false otherwise. Thus CHILD-EXISTS-MS will return the same result as CHILD-EXISTS.

**Space and time Cost of Algorithm 9.** In the following we provide space and time bounds for Algorithm 9 by firstly fixing some useful properties in Lemmas 4.7, 4.8, and 4.9.

Notice that $I_{<v} \setminus I_v' = N_{<v}(v) \cap I_{<v}$. Since $\overline{N}(I_v' \cup \{v\}) \subseteq \overline{N}(I_v')$, if there exists a vertex $x ∈ \overline{N}(I_v' \cup \{v\})$ smaller than $v$, then $x$ satisfies the properties shown in Lemma 4.7. We show that this lemma follows from the definition of parent index and it is useful to efficiently perform the computation in line 2 in Algorithm 9.

**Lemma 4.7.** Let $I$ be a MIS, and $v$ a vertex s.t. $v \notin I$ and $v > p(I)$. Then $\overline{N}(I_{<v})_{<v} = ∅$, and for each vertex $x$ in $\overline{N}(I_v')_{<v}$ we have that either $x$ is in $I \cap N_{<v}(v)$ or $x$ has a neighbor in $I \cap N_{<v}(v)$.

**Proof.** Since $v \notin I$ and $v > p(I)$, we have that complete($I_{<v}$) = $I$ by definition of $p(I)$. If $\overline{N}(I_{<v})$ does contain a vertex $x$ smaller than $v$, we could use one of such vertices to extend $I_{<v}$ and we would have complete($I_{<v}$) $\neq I$, a contradiction.

Consider now $\overline{N}(I_v')_{<v}$: As $I_v' \subseteq I_{<v}$, we have $\overline{N}(I_v')_{<v} \supseteq \overline{N}(I_{<v})_{<v}$. Since $\overline{N}(I_{<v})_{<v} = ∅$, however, we have that for each $x ∈ \overline{N}(I_v')_{<v}$, $x ∉ \overline{N}(I_{<v})$. Thus either $x$ is in $I_{<v}$ has a neighbor in it by definition of $\overline{N}(\cdot)$. As $x ∈ \overline{N}(I_v')_{<v}$, if $x$ is in $I_{<v}$, it is actually in $I_{<v} \setminus I_v' \subseteq N_{<v}(v)$. The statement follows.

Hence, we have to verify the conditions of Lemma 4.7 for the vertices $x ∈ \overline{N}(I_v' \cup \{v\})$. However, it is worth noting that the cost of storing $\overline{N}(I_v' \cup \{v\})$ is $O(n)$ which exceeds our memory requirements. To overcome this issue, we show in the following lemma how to build a heap-based iterator, that iterates over $\overline{N}(I_v' \cup \{v\})$ without computing it.
Lemma 4.8. Let $X \subseteq V$ be a set of vertices and $Y = \bigcup_{x \in X} N(x)$. We can iterate over every $y \in Y$ in increasing order (without explicitly storing $Y$) in $\tilde{O}(\min\{\abs{X}\Delta, m\})$ time using $O(|X|)$ additional space.

Proof. Allocate a heap and add to it, for each vertex $x$ in $X$, its smallest neighbor $y$ (saving the $x$ responsible for its addition). We can use this heap to iterate in order all vertices with a neighbor in $X$ as follows: iteratively remove the minimum element $y$ of the heap, recover the vertex $x$ responsible for the addition of $y$, and insert in the heap the smallest neighbor of $x$ larger than $y$. This way the smallest neighbor that we did not extract yet will always be on top of the heap. It is possible that the same vertex is extracted more than once; however we can trivially ignore duplicates as they appear contiguously, since vertices are extracted in increasing order. Adding/removing an element to/from the heap costs $\tilde{O}(1)$, so the total cost is bounded by $\tilde{O}(1)$ times the sum of all degrees of vertices in $X$, that is $\tilde{O}(\min\{|X|\Delta, m\})$. \hfill\qed

By Lemma 4.7 to answer the check at line 2 we can consider vertices $y$ belonging to $I \cap N_{<}(v)$ or $N(I \cap N_{<}(v))$. In particular, for each of them, we will have to check that $y \notin N(v)$ and $N(y) \cap I'_{v} \neq \emptyset$. To this aim, we use the following lemma.

Lemma 4.9. Let $I$ be a MIS, $v \notin I$ a vertex such that $v > \pi(I)$, and $y$ any vertex. We have $N(y) \cap I'_{v} \neq \emptyset$ iff there exists $z \in N(y)$ such that $z < v$ and $z \in I$ and $z \notin N_{<}(v)$.

Proof. Recall that $I'_{v} = I_{<v} \cap \overline{N}(v) = I_{<v} \setminus N_{<}(v)$, so all and only vertices in $I'_{v}$ are smaller than $v$, in $I$, and not in $N_{<}(v)$. As $N(y) \cap I'_{v} \neq \emptyset$ iff any vertex in $N(y)$ is in $I'_{v}$, and $z \in I'_{v}$ iff ($z < v$ and $z \in I$ and $z \notin N_{<}(v)$) the statement follows. \hfill\qed

We are now ready to prove the overall cost of Algorithm 9.

Lemma 4.10. child-exists-ms can be computed in $\tilde{O}(\min\{d\Delta^{2}, m\})$ time with $O(s)$ space.

Proof. Consider line 2 since $\overline{N}(I'_{v} \cup \{v\}) \subseteq \overline{N}(I'_{v})$, by Lemma 4.7 if there exists a vertex $x \in \overline{N}(I'_{v} \cup \{v\})$ smaller than $v$, then $x$ is a neighbor of a vertex in $I \cap N_{<}(v)$ (note that $x$ cannot be in $I \cap N_{<}(v)$ since it is in $\overline{N}(I'_{v} \cup \{v\})$). Thus, instead of computing $I'_{v}$ and its complementary-neighborhood, we iterate over vertex $y$ which has a neighbor in the set $X = I \cap N_{<}(v)$ using Lemma 4.8. For each $y$, we have to check that $y \notin N(v)$ and $N(y) \cap I'_{v} \neq \emptyset$; for this latter check we use Lemma 4.9. If any $y$ fails the check, then we return false, otherwise $\overline{N}(I'_{v} \cup \{v\})_{<v} = \emptyset$ and we can continue. We have $\abs{I \cap N_{<}(v)} \leq d$ (due to the reversed degeneracy ordering), thus the neighbors $y$ that we have to test can be at most $d\Delta$. It follows that the iteration will cost $\tilde{O}(\min\{d\Delta, m\})$ time by Lemma 4.8. Furthermore, testing each vertex $y$ as in Lemma 4.9 takes $\tilde{O}(\abs{N(y)})$ time as we can perform binary searches on $I$, thus the total cost of testing is bounded by $\tilde{O}(\min\{d\Delta^{2}, m\})$.

Consider now lines 3-4. Again, using Lemma 4.7 we know that all vertices $x \in \overline{N}(I'_{v})_{<v}$ are either in $I \cap N_{<}(v)$, or have a neighbor in it. We can rewrite this condition as: $x$ has a neighbor in $X' = (I \cap N_{<}(v)) \cup \{v\}$.

In order to compute $x$ in line 3 since $\overline{N}(A)_{<v} \subseteq \overline{N}(I'_{v})_{<v}$, we use Lemma 4.8 to iterate over all the neighbors of vertices in $X'$; this iteration will yield in increasing order all vertices at any point in $\overline{N}(A)_{<v}$. We actually do not store $A$, but only the vertices that are added to $A$ during the while, which we will here call $A'$. Thus to check that a vertex $x$ belongs to $\overline{N}(A)$, we check $N(x) \cap I'_{v} = \emptyset$ and $N(x) \cap A' = \emptyset$; the earlier part can be done with Lemma 4.9 while the latter in $\tilde{O}(A')$ time by using binary searches.

Once we found $x = \min\{\overline{N}(A)_{<v}\}$, if it passes the check on line 6 we add it to $A$ and repeat the loop, otherwise we return the result of the check.
Note that, as we are only iterating over \( \overline{N}(I'_v)_{<v} \), we will not find among them any vertex in \( I_{>v} \). However, this is easily fixed by saving the vertex \( \min \{ I_{>v} \} \): if at any point we have \( x > \min I_{>v} \), or we finish the iteration on \( X' \), we return true since in both cases the candidate to be added to \( A \) would have been \( \min \{ I_{>v} \} \).

As \( |X'| \leq \min \{ d + 1, s \} \), and the number of vertices with a neighbor in \( X \) are bounded by \( O(\min \{ d\Delta, n \}) \), similarly to above we can bound the cost of the iteration with \( \tilde{O}(\min \{ d\Delta, m \}) \), and the total cost of testing with \( \tilde{O}(\min \{ d\Delta^2, m \}) \). Furthermore, note that the condition in line 6 can only succeed up to \( \min \{ d, s \} \) times, as each time \( x \) is in \( I_{<v} \setminus I'_v \), and \( |I_{<v} \setminus I'_v| = |I \cap N_c(v)| \leq \min \{ d, s \} \); this means that \( |A'| \leq \min \{ d, s \} \).

Thus the total cost of \textsc{child-exists-ms} is \( \tilde{O}(\min \{ d\Delta^2, m \}) \), using additional space \( O(|X'| + |A'|) = O(\min \{ d, s \}) \) by Lemma \ref{lem:4.8}.

By using Lemma \ref{lem:4.10} we are now able to prove the following result.

**Theorem 4.1.** There exists an algorithm that enumerates all maximal independent sets with \( \tilde{O}(\min \{ nd\Delta^2, mn \}) \) delay and \( O(s) \) additional space.

**Proof.** By using the structure described in Section \ref{sec:4.3} we can create an algorithm that enumerates all MISs (that is Algorithm \ref{alg:7}) where \textsc{child-exists} is replaced by \textsc{child-exists-ms}, that is complete and correct by Lemma \ref{lem:4.6}. Its delay is bounded by the costs of (i) the generation function \( F(I, v) = \text{complete}((I_{<v} \cap N(v)) \cup \{ v \}) \) (ii) testing each candidate with \textsc{child-exists-ms}, and (iii) \textsc{parent-state} to return to the parent solution. These costs are respectively (i) \( \tilde{O}(m) \) (as shown in Lemma \ref{lem:4.2}), (ii) \( \tilde{O}(n \cdot \min \{ d\Delta^2, m \}) \) as we apply Lemma \ref{lem:4.10} to each of the \( O(n) \) candidates, and (iii) \( \tilde{O}(m) \) (as shown in Lemma \ref{lem:4.3}). Since \( m \leq n\Delta \), this gives a total cost of \( \tilde{O}(m + \min \{ nd\Delta^2, mn \}) = \tilde{O}(\min \{ nd\Delta^2, mn \}) \).

As we have no recursion stack, the additional space is simply storing \( I \) and \( v \), and the space required by \textsc{child-exists-ms}, that is \( O(s) \).

\[ \square \]

### 4.6 Faster version using \( O(n) \) additional memory

In this section, we propose a new algorithm which achieves a smaller time cost per solution by exploiting properties of the search space and an additional data structure of size \( O(n) \), which mainly stores the amount of neighbors in \( I_{<v} \) of each vertex in the graph. We use a function \textsc{child-exists-fast} which improves the time cost of Algorithm \ref{alg:7} by constructing and maintaining this data structure. This is fundamental to improve the running time since it allows us to reduce the search space of the vertices considered by \textsc{child-exists} (and the corresponding iterations), as just vertices with zero neighbors in \( I_{<v} \) need to be considered. Since \( v \) varies among all the possible candidates, even the ones not leading to a solution, this data structure cannot be rebuilt from scratch each time \( I_{<v} \) changes, but needs to be properly updated and restored wherever possible. We will prove that we can cover these costs.

For the sake of completeness, the final pseudo-code is shown in Algorithm \ref{alg:10}. The functions \textsc{is-root} and \textsc{parent-state} are the same as in \textsc{base} (see Algorithm \ref{alg:7}). We will now analyze the difference between \textsc{base} and Algorithm \ref{alg:10} to show that they are equivalent, and that, hence, Algorithm \ref{alg:10} is correct.

The first difference we can notice is that we use a new function \textsc{get-next-cand} with respect to the one in \textsc{base}. The new one is faster to compute, since the sum of the costs of all the calls done with the same \( I \) takes just \( O(n) \) time but returns a superset of the one of \textsc{base}. This fact increases the number of candidate vertices to test but, on the other hand, testing them will be faster here due to an improved version of \textsc{child-exists} (see Lemma \ref{lem:4.13}).

72
Algorithm 10: Fast enumeration of maximal independent sets

1. Assume \( \min(\emptyset) = \text{null} \).

2. Function \text{improved-spawn}(I)

3. \( v \leftarrow p(I); \ prev \leftarrow v \)

4. \text{build}(ws, I_{<v})

while true do

5. \( \text{childless} \leftarrow \text{true} \)

6. while \( v \leftarrow \text{get-next-cand}(I, v) \neq \text{null} \) do

7. \text{update}(ws, (I_{<v} \setminus I_{<prev})) \quad \text{prev} \leftarrow v

8. if \text{child-exists-fast}(I, v, ws) then

9. \( I \leftarrow \text{complete}(I' \cup \{v\}) \)

10. \( \text{childless} \leftarrow \text{false} \)

11. \text{build}(ws, I_{<v})

12. break

if \( \text{childless} \) then

13. if \( \text{is-root}(I) \) then return

14. else

15. \( \langle I, v \rangle \leftarrow \text{parent-state}(I) \)

16. \text{prev} \leftarrow v

17. \text{build}(ws, I_{<v})

\[ \]

Lemma 4.11. We can use function \text{get-next-cand} of Algorithm 11 in place of \text{get-next-cand} in base.

Proof. \text{get-next-cand} of Algorithm 11 iterates over \((V \setminus I)_{>v}\), while \text{get-next-cand} in base iterates over \(\{w \in \bigcup_{u \in I} \overline{N}(u) \setminus I : w > v\}\). As the latter is a subset of the former, Algorithm 11 will iterate over all the candidates that will lead to a child. Furthermore, vertices in \((V \setminus I)_{>v}\) are still greater than \(p(I)\), thus the conditions for \text{child-exists} are met, and the vertices that do not lead to a child will fail the check. \(\square\)

Notice that the candidate set size is \(\Theta(n)\), since a single complementary neighborhood has size \(\Theta(n)\). Another difference with respect to base, is that we use function \text{child-exists-fast} instead of \text{child-exists} function to improve its computational time. To this aim, we use an additional data structure \(ws\) (for \text{weights}), which we keep suitably updated in order to satisfy the following invariant.

Lemma 4.12. When \text{child-exists-fast}(I, v, ws) is called in Algorithm 10 for each vertex \( i \in V \), we have \(ws[i] = |N(i) \cap I_{<v}|\).

Proof. We first remark that for any \(A, B\) s.t. \(A \cap B = \emptyset\), if \(\forall i \in V ws[i] = |N(i) \cap A|\) and we call \text{update}(ws, B) we obtain \(\forall i \in V ws[i] = |N(i) \cap (A \cup B)|\). To prove the lemma, it is thus sufficient to show that just before line 8 is executed, \(\forall i \in V ws[i] = |N(i) \cap I_{<prev}|\) (for the value of \(prev\) at that point).

Let us consider when \(ws\) was last modified when line 8 is executed: If this is the first time that the while loop in line 7 is executed, then \(ws\) was last modified in either lines 4, 13 or 20 by calling \text{build}(ws, I_{<v}). Indeed, we have \(\forall i \in V ws[i] = |N(i) \cap I_{<prev}|\) by definition of \text{build}, as we set \(prev = v\) in lines 3, 9 and 19 respectively, and \(prev\) remained unchanged until line 8. Otherwise, note that \text{child-exists-fast} leaves \(ws\) unchanged (the changes at Line 2 are canceled out by Line 11), thus \(ws\) was last modified by the previous execution of line 8 in the while loop (line 7). We prove this case by induction: Let us refer to the values of \(v\) and \(prev\) at line 8 in the \(j\)-th
Algorithm 11: Fast check of the existence of a child and other auxiliary functions

1 Function child-exists-fast(I, v, ws)
2 for x ∈ I ∩ N_c(v) do
3     for y ∈ N(x) do ws[y]− = 1
4     C ← {x : ws[x] = 0}
5     if not min(|N(I_i) ∪ {v})| > v then return false
6     while C not empty do
7         c ← min{C}
8         if c ∈ (I \ I_i) then
9             C ← C \ (N(c) ∪ {c})
10        else
11            update(ws, I ∩ N_c(v))
12            return c ∈ I
13 Function build(ws, A)
14     ∀i ∈ V : ws[i] ← |N(i) ∩ A|
15 Function update(ws, A)
16     ∀i ∈ V : ws[i] + = |N(i) ∩ A|
17 Function get-next-cand (I, v)
18     return min((V \ I)_{>v})

Iteration of the loop as v_j and prev_j. Assume that ∀i ∈ V ws[i] = |N(i) ∩ I_{<prev_j}| was true at line 8 in the j-th iteration. As v_j = prev_j+1 (see line 9), after the line is executed, by the remark at the beginning of the proof, we have ∀i ∈ V ws[i] = |N(i) ∩ I_{<v_j}| = |N(i) ∩ I_{<prev_j}|. Since the condition is true for the first iteration, it is true for any iteration, thus the statement holds in each case.

By Lemma 4.12, the hypothesis on the ws data structure in the following lemmas are met, so that we can use the new CHILD-EXISTS-FAST instead of CHILD-EXISTS and CHILD-EXISTS-MS.

Lemma 4.13. Suppose that ws[x] = |I_{<v} ∩ N(x)| for each x ∈ V. Then CHILD-EXISTS-FAST (I, v, ws) in Algorithm 11 can be used instead of CHILD-EXISTS (I, v) and CHILD-EXISTS-MS (I, v).

Proof. Line 6 in Algorithm 11 is the same as line 2 in Algorithm 9. The loop at line 2 decrements ws[y] once for each neighbor of y in I ∩ N_c(v). After the loop we have that for each x ∈ V, ws[x] = |(I_{<v} \ N_c(v)) ∩ N(x)| = |I_{<v} ∩ N(x)|, thus x ∈ N(I_{<v}) iff ws[x] = 0. It follows that C is initialized to exactly N(I_{<v}). Thus, the loop in lines 6-12 will have the same outcome as the corresponding loop in lines 4-7 of Algorithm 9 when a vertex c is selected in line 8. C is updated as if c was added to I_{<v}, thus the following iterations will select the same vertices that would be selected in Algorithm 9 until finally line 12 is executed, which will give the same outcome as line 7 in Algorithm 9.

The function IMPROVED-SPAWN has been modified only with the addition of the function UPDATE, which has effect only on the variable ws which is used by GET-NEXT-CAND. Thus we can prove that the function SPAWN too is equivalent to the one of BASE, obtaining the following lemma.


Proof. The correctness of the algorithm follows from Lemmas 4.11, 4.13 and 4.12. Indeed, it is proven by those lemmas that functions GET-NEXT-CAND and CHILD-EXISTS-FAST used by Algorithm 10 are equivalent to those of Algorithm 7. The structure of the function IMPROVED-SPAWN is the same, except for the instructions BUILD and UPDATE, which have the only effect...
of updating the data structure ws in order to satisfy the invariant in Lemma 4.12. Thus the
algorithm will produce the same output as Algorithm 7, which is correct by Lemma 4.1. □

Space and time cost of Algorithm 10. In the following, we analyze the complexity of
Algorithm 10. We have already discussed the cost of the new GET-NEXT-CAND function. In
particular, we analyze the cost of maintaining the counters and the cost of CHILD-EXISTS-FAST.

Lemma 4.15. For any set \( A \subseteq V \), UPDATE(ws, A) takes \( O(\min\{|A|\Delta, m\}) \) time.

Proof. We can obtain the cost above by iterating over all vertices in \( A \) and for each vertex \( x \)
incrementing by 1 the counter of its neighbors. This is bounded by \( O(|A|\Delta) \), and by \( O(m) \) too
as it is a sum of the degrees of distinct vertices. □

The cost for BUILD is simply \( O(n + \min\{|A|\Delta, m\}) \) time, since we can set to zero each \( ws[x] \)
for each \( x \in V \) and then apply UPDATE(ws, A).

Lemma 4.16. CHILD-EXISTS-FAST takes \( O(d\Delta) \) time and \( O(n) \) space.

Proof. Recall from the proof of Lemma 4.10 that, as \( v \not\in I \) and \( v > \phi(I) \), we have \( \min\{N(I_{<v})\} > v \),
thus \( \forall x \in (V_{<v} \setminus I) \) we have \( ws[x] > 0 \).

Thus we compute \( C \) by simply adding to it any vertex \( y \) whose counter \( ws[y] \) is set to 0
during the loop at line 2. This is only guaranteed to add vertices which are smaller than \( v \), but
this will be enough. Recalling the proof of Lemma 4.13, we thus have that \( C = \overline{N}(I'_{<v}). \) This
costs us \( O(\min\{d\Delta, m\}) \) time, as it is the sum of the degrees of \( |I \cap N(<v)| \leq d \) distinct vertices.
Line 3 takes \( O(\Delta) \). Indeed, we can compute \( \overline{N}(I' \cup \{v\})_{<v} \) as \( C \setminus N(v) \). If this set is not empty,
then the check fails and we return false.

Let us now consider the cost of the loop at line 6. Keeping \( C \) in a dynamic dictionary, lines 7 and
8 take both \( \tilde{O}(1) \) time and \( \tilde{O}(|N(c)|) \) time. As the loop is executed a maximum of
\( |I_{<v} \setminus I'_{<v}| + 1 \leq |N_{<}(v)| + 1 \leq d + 1 \) times and \( c \) is different every time, this takes \( \tilde{O}(\min\{d\Delta, m\}) \) time.
As in Lemma 4.10 we should store \( \min\{I_{>v}\} \): since initially \( C \) only contains \( \overline{N}(I'_{<v}) \)
rather than \( \overline{N}(I') \), we return true if \( C \) actually becomes empty or if \( c > \min\{I_{>v}\} \), as in both
these cases \( \min\{I_{>v}\} \) would have been the candidate actually selected in line 9 that would result
in the algorithm returning true. Finally, before returning the result, we restore the data structure
ws by calling UPDATE(ws, \( I \cap N(<v) \)). This takes \( O(\min\{d\Delta, m\}) \) time by Lemma 4.15.

Since the additional space used is \( O(|C|) = O(\min\{d\Delta, n\}) \), CHILD-EXISTS-FAST can be computed in \( \tilde{O}(\min\{d\Delta, m\}) \) time, using \( O(n) \) space. □

By plugging the results of Lemmas 4.15 and 4.16 into the analysis of Theorem 4.1, we conclude
the following.

Theorem 4.2. Algorithm 10 lists all the maximal independent sets with a delay of \( \tilde{O}(\min\{nd\Delta, nm\}) \),
and \( O(n) \) additional space.

Proof. Using Theorem 4.1 with the costs of CHILD-EXISTS-FAST given by Lemma 4.16 we obtain
a total cost of \( \tilde{O}(\min\{nd\Delta, nm\}) \). We however need to add to steps (i) and (iii) the cost
of BUILD(ws, \( I_{<v} \)) which takes \( \tilde{O}(\min\{n\Delta, m\}) \). Furthermore, we add to step (ii) the cost of
UPDATE(ws, \( I_{<v} \setminus I_{<prev} \)) for each candidate \( v \). We argue that for any specific \( I \), any vertex
\( i \in I \) is in \( I_{<v} \setminus I_{<prev} \) only once, since \( v \) is never decreasing for \( I \) and prev keeps track of the
previous value of \( v \). The total cost is thus the same as UPDATE(ws, \( I \)), i.e., \( \tilde{O}(\min\{|I|\Delta, m|) \) by
Lemma 4.15. As \( |I| \leq n \), neither of these additions affects the total cost of \( \tilde{O}(\min\{nd\Delta, m\}) \).

Space usage is given by the size of ws and \( C \) stored in CHILD-EXISTS-FAST, i.e., \( O(n) \). □
4.7 Final remarks

In this chapter we studied the enumeration of maximal independent sets (MISs) in graphs, introducing new ideas to check efficiently which neighbors in the reverse search should be explored, as this task is time- and space-consuming. For a read-only input graph, the result is a P-DELAY algorithm with minimal additional space, proportional to the size of the largest MIS, and another P-DELAY algorithm which improves both delay and space usage of known approaches. We remark that a MIS can indeed have linear size: in this case, due to the modular nature of our algorithms, the execution of the minimal space version can switch on-the-fly to the faster version which uses $O(n)$ space without increasing the asymptotic space usage.
As discussed in the introduction of this thesis, the problem of listing subgraphs satisfying a given property can be modeled under the general framework of set systems. In scenarios where only maximal solutions are relevant, the ambitious goal is to cover a large class of set systems without compromising the efficiency of the corresponding enumeration algorithms. Among the latter ones, the best-known ones list the maximal subsets in time proportional to their number but may require exponential space. In this chapter we improve the state of the art in two directions by introducing an algorithmic framework that, under suitable conditions, simultaneously (i) extends the class that can be solved efficiently to strongly accessible set systems, and (ii) reduces the additional space usage from exponential in $|U|$ to stateless, thus accounting for just $O(q)$ space, where $q \leq |U|$ is the largest size of a maximal set. Furthermore, we provide an upper bound for the cost of listing maximal solutions parameterized in $q$, which approaches the lower bound obtained from Lawler’s work described in Section 2.6.2.

5.1 Introduction

In network analysis discovering special communities corresponds to finding all the subgraphs with a given property [ANRD15, DWP+07, LRJA10, MD08, Mok79, TBG+13]. In bioinformatics, listing all the solutions is desirable, as single or few solutions may not be meaningful due to noise of the data, noise of the models, or unclear objectives [CG08, KMS+12, LCTS08, MKC+14, TSS02]. In graph databases, graph structures are used for semantic queries, where vertices, edges, and properties are used to represent and store data; retrieving information corresponds to find all the suitable subgraphs in these databases [AG08, CS05, WHW07]. When dealing with incomplete information, it may be impossible to completely satisfy a query. Subgraph listing algorithms can find answers that maximally satisfy a partial query; for instance, there is a one-to-one correspondence between the results of a join or full disjunction query and certain subgraphs of the assignment graph, a special graph obtained by combining the relational database with the given query. Moreover, the kind of subgraphs to look for depend not only on the database, but also on the query [CS05].

In this scenario, graph enumeration has left the theoretical border [Was16] to meet more stringent requirements: not only a given listing problem must fit a given class of complexity, but its algorithms must be efficient also in real-world applications. To promote efficiency, many works try to eliminate redundancy by, for example, listing only solutions which respect properties of closure [BHPW10] or maximality [CKS08]. On the other hand, algorithm design has made a
big effort to generalize the graph properties to be enumerated and to unify the corresponding approaches [AF96, CNS5, CKS08, LLRK80, TIAS77]. These generalizations allow the same algorithm to solve many different problems.

The contribution of this chapter fits into this line of research: on one side, we want to obtain efficient listing algorithms able to list maximal solutions in large networks; on the other side, we aim at designing an algorithmic framework which solves simultaneously many problems and leave the designer in charge of few core tasks depending on the specific application. In particular, we focus on efficient enumeration algorithms for maximal subgraphs satisfying a given property (e.g. being a clique, a cut, a cycle, a matching, etc.), as they fall within the general framework of set systems, defined in Section 2.1.2.

Moreover, we assume in the rest of the chapter that $F$ is given algorithmically, i.e. we can check whether $X \in F$ in time $T_f = poly(|U|)$ and space $S_f = \Omega(q)$, where $q$ is the maximum size among the sets $X \in F$.

We are interested in the problem of listing just the maximal solutions of $F$, where a solution $X \in F$ is maximal if there exists no $Y \in F$ such that $X \subset Y$. This is of interest for many applications in literature (e.g. [CN85, TIAS77, JYP88, KLW96]) where instances can have a number of maximal solutions that is much smaller than $|F| \leq 2^{|U|}$. However, generating only maximal solutions usually makes the listing problem harder, as remarked in Section 2.4.

For this reason, we will focus on strongly accessible set systems. Recalling their definition (Section 2.1.2), they are the set systems $(U, F)$ for which $X, Y \in F$ and $X \subset Y$ implies that there exists $z \in Y \setminus X$ such that $X \cup \{z\} \in F$. This makes it the most general class of set systems which guarantees that the maximality of a solution can be verified in polynomial time.

We remark that there is a trade-off between generality and efficiency, as a truly general output sensitive framework for listing maximal solutions seems hard to achieve for all set systems: as remarked in Section 2.6 this would imply $P=NP$. For this reason, we follow the intuition of Lawler et al. [LLRK80] and later Cohen et al. [CKS08], who showed that the hardness of a listing problem can be linked to that of solving an easier core task, called in [CKS08] restricted problem. Keeping this in mind, we will proceed along two orthogonal directions simultaneously: (i) our framework can deal with a wide class of set systems; (ii) we heavily improve the memory usage from the currently known exponential space [CKS08] to polynomial. We actually go further when output has not to be stored, and show that our framework is stateless, meaning that it uses just minimal additional space, whenever the restricted problem allows for it.

**Case studies and open problems.** Many works in the literature focus their attention on listing maximal solutions of specific set systems, solving them with ad hoc techniques. For example, they focus on cliques [BK73, ELS13, TK07, MU04, Akk73], independent sets [JYP88, TIAS77], acyclic subgraphs [Yau67, SS02], matchings [GNS09, Uno01], $k$-plexes [BCK15, WP07] which all correspond to hereditary properties, or more independence set systems, defined in Section 2.1.2. These kinds of problems can be often efficiently solved in output sensitive time and polynomial
space by using general frameworks [CN85, CKS08, LLRK80].

However, some tricky properties do not fit in the above scenario. The maximal common connected induced subgraphs (MCCIS) between two graphs $A$ and $B$ is one notable example. This problem is at least as hard as the graph isomorphism problem [Bab16]. For this reason a weaker form is considered, where the individual isomorphisms are listed (noting that many isomorphisms can correspond to the same MCCIS). Surprisingly, no output-sensitive listing algorithm with polynomial space is known for even this version, while several papers [CCC+08, Koc01, KLW96] adapt existing techniques for maximal clique enumeration without any guarantee. We will use this problem as a running example through the chapter, showing that our framework yields an output-sensitive and polynomial space algorithm for the problem. What is known is the transformation by Koch [KLW96], so that it reduces to the enumeration of special cliques in a new graph, called product graph $G$, whose edges are either black or white (see Figure 5.1): let $G_B$ be the edge subgraph of $G$ containing only the black edges and define a $bc$-clique as a connected subgraph of $G_B$ whose vertices form a clique in $G$; then, for each (maximal) isomorphism of $A$ and $B$, there is a (maximal) $bc$-clique in $G$, and vice versa.

Listing $bc$-cliques share with other listing problems the so called connected hereditary property: all the solutions in $F$ are connected in some given graph ($G_B$ in the case of $bc$-cliques); moreover, given $X \subseteq Y \subseteq U$, if $X$ is connected and $Y \in F$ then $X \in F$, i.e., $F$ is closed w.r.t. connected induced subgraphs [CKS08]. In this chapter we propose a solution to the open problem of finding a memory-efficient framework for connected hereditary graph properties: we aim at using just polynomial space, and at the same time keeping the complexity competitive with the state of the art techniques, that require exponential space [CKS08, CS05] as they require keeping all the maximal solutions in memory.

We remark that hereditary and connected hereditary set systems are both also strongly accessible set systems, as shown in Figure 5.2. However, it is easy to show that the strongly accessible class contains more than just hereditary and connected hereditary properties: for example, if $R$ is a set of vertices, the property of “being a $bc$-clique containing a vertex of $R$” is not connected hereditary (as a connected subset will still be a $bc$-clique but may not contain any vertex from $R$), but it is straightforward to see that it is still strongly accessible. In general, we will refer to hereditary and connected hereditary properties with the additional constraint of containing a vertex from a required set as required hereditary and required connected hereditary respectively. Such properties are strongly accessible, but they are no more hereditary or connected hereditary. For the sake of clarity, Table 5.1 summarizes our definitions and gives some examples.

**Our results.** In this chapter, we provide a stateless output-sensitive framework for connected hereditary properties. This is the first polynomial space output-sensitive framework for this kind of systems, solving the open problem by [CKS08], as stateless implies polynomial space (actually it can be much less). As an application, our result implies the first output sensitive polynomial memory listing algorithm for maximal isomorphisms ($bc$-cliques) corresponding to maximal common connected induced subgraphs, using just $O(q)$ memory.

Our approach extends to the strongly accessible set systems independently from the core task, with a cost which is not far from the optimum, which we prove to be in the worst case $\Omega(\alpha^{2\epsilon/2})$, where $\alpha$ is the number of maximal solutions, unless the Strong Exponential Time Hypothesis [IP99] is false.

**Techniques.** When designing listing algorithms, binary partition is a simple yet powerful technique which consists in recursively dividing the set of solutions into two or more non-overlapping partitions. In order to get output-sensitive algorithms, it is necessary to decide whether each of these sets is empty. However, when dealing with maximal solutions, the latter
problem is often NP-complete, like in the case of cliques and many other problems.\footnote{Some additional discussion is provided in Section 8.3}

In this scenario, reverse search by Avis and Fukuda \cite{AF96} is a more powerful technique avoiding this kind of problems. Its general schema is based on operations which allow jumping from a solution to another. These operations define a graph-like structure as that represented in Figure 5.3 that can be traversed in order to find all of them. The key feature of reverse-search is then pruning this structure to obtain a tree-like (or forest-like) one: a tree-like structure is easy to traverse without incurring multiple times in the same vertices, whereas a more complex graph-like structure requires keeping in memory all the visited vertices. For the reader who is not familiar with reverse search, we refer to the original paper \cite{AF96} or to the overview in Section 2.3.4.

Whether reverse search could be efficiently adapted to be output sensitive and memory efficient on non-hereditary set systems was an open problem \cite{CKS08}. We show that state of the art strategies for many specific set systems cannot be applied, as the routines to transform a maximal solution into another, when using the lexicographical order of the elements, involve

| CLASS            | EXAMPLES                                                                 | PROPERTY                                                                 
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hereditary (ISS)</td>
<td>Clique, Independent Sets, Induced Forests</td>
<td>$X \subseteq Y \subseteq U$ and $Y \in F$ implies $X \in F$</td>
</tr>
<tr>
<td>Connected Hereditary</td>
<td>Connected $k$-plexes, Common Connected Subgraphs (bc-cliques), Induced Trees</td>
<td>Let $X \subseteq Y \subseteq U$. If $X$ is connected and $Y \in F$ then $X \in F$</td>
</tr>
<tr>
<td>Commutable</td>
<td>Hereditary, Connected Hereditary, Required Connected Hereditary</td>
<td>Are Strongly Accessible and respect the commutable property (Definition 5.3)</td>
</tr>
<tr>
<td>Strongly Accessible</td>
<td>Includes Commutable</td>
<td>$X, Y \in F$ and $X \subset Y$ imply that there exists $z \in Y \setminus X$ such that $X \cup {z} \in F$</td>
</tr>
<tr>
<td>(Weakly) Accessible</td>
<td>Includes Strongly Accessible, Dense subgraphs</td>
<td>$X \in F$ and $X \neq \emptyset$ imply that there exists $z \in X$ such that $X \setminus {z} \in F$</td>
</tr>
</tbody>
</table>

Table 5.1: Classes of Set Systems.

Figure 5.2: Hierarchy of Classes of Set Systems.
NP-hard problems. We solve the problem by representing a solution not as a bag of elements, but with a canonical representation which takes into account the topology of the solution. This new layer of complexity needs non trivial ideas and technicalities to avoid generating duplicate solutions, in order to ensure correctness without storing the output in memory.

5.2 Related work

Hardness and Restricted Problem for hereditary properties. Hereditary properties, also called Independence set systems, have been well-known and studied since the 80’s. One of the most striking result is the one by Lawler et al. [LLRK80], which shows that generating all the maximal solutions of an independent set system in an output-sensitive fashion implies P=NP. This is proved by reducing SAT formulas with $N$ variables to appropriate set systems which have $N$ dummy maximal solutions, plus one maximal solution for each satisfying assignment of the corresponding formula. Moreover, in the same work the notion of restricted problem has been introduced. As the restricted problem is a sub-problem of the general one, we have that if the general problem can be solved with some guarantee, the restricted one can be solved with the same guarantee. On the other hand, both [LLRK80] and [CS05] show that the running time and space requirement of the general problem also depends on those of the restricted one so that if the latter ones are easier to solve then the former can potentially achieve output sensitivity. For hereditary properties, [CKS08] proves that the general problem can be solved in polynomial space and polynomial delay if the restricted problem can be solved in polynomial space and polynomial total time.

The framework by Cohen et al. for connected hereditary properties. The work by [CKS08] proves significant insights for connected hereditary set systems, by providing an algorithm for the general problem which uses the restricted problem as a subroutine. They show that if the restricted problem can be solved in polynomial time (resp. incremental polynomial time, polynomial total time) then the general problem can be solved in polynomial delay (resp. incremental polynomial time, polynomial total time). This relationship extends that among hereditary properties and their restricted problems, at the cost of using exponential space. Indeed, for connected hereditary properties, their framework essentially corresponds to a visit on
Table 5.2: Comparison with the state of the art (\cite{LLRK80} and \cite{CKS08}), where \( G_T \) and \( G_S \) are respectively the time and space to recognize whether \( X \in F \), while \( R_T \) and \( R_S \) are the time and space required to get the next solution of the restricted problem, \( q \) is the maximum size of the solutions in \( F \).

<table>
<thead>
<tr>
<th>Class</th>
<th>Lawler et al.</th>
<th>Cohen et al.</th>
<th>This Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hereditary/ISS</td>
<td>( \text{poly}(\mid U \mid + R_S) )</td>
<td>( O(G_S + R_S) )</td>
<td>( \text{poly}(\mid U \mid + R_T) )</td>
</tr>
<tr>
<td>Connected Hereditary</td>
<td>( \exp(\mid U \mid) )</td>
<td>( O(G_S + R_S) )</td>
<td>( \text{poly}(\mid U \mid + R_T) )</td>
</tr>
<tr>
<td>Strongly Accessible</td>
<td>( O(G_S) )</td>
<td>( O(2^q) )</td>
<td>( O(2^q) )</td>
</tr>
</tbody>
</table>

A graph-like structure like that in Figure 5.3 storing in memory all maximal solutions found so far. This clearly requires exponential space, as the number of maximal solutions can be exponential. Reducing the space usage to polynomial, while keeping the delay bounded, is left as an open problem in \cite{CKS08}. This chapter will also fill this gap.

Calling \( R_T \) and \( R_S \) respectively the time and space required to solve the restricted problem for the problem at hand, we show in Table 5.2 a summary of the results achieved by the techniques in \cite{LLRK80} and \cite{CKS08}, as well as those of this work.

The frameworks for Closed Objects by Boley et al. and Uno et al. Other works have focused on set systems more general than the ones studied by \cite{CKS08}, like strongly accessible set systems, but they used different requirements for the objects to look for. This is the case of the framework by \cite{BHPW10} and \cite{AU09}. In these works, a set system \((U, F)\) is given together with a closure operator \( \sigma: F \to F \), which satisfies extensivity, monotonicity, and idempotence properties. In such a context, a set \( F \in F \) is called closed if \( \sigma(F) = F \) and these works aim at listing all the \( F \in F \) which are closed. In principle, the concept of closure is similar to that of maximality, in that both allow the algorithm to ignore many solutions that are not significant in that their information is included in some other more important one, thus improving the significance of the output and the total running time. We remark, however, that this is a different kind of problem as we require maximal solutions instead, for which monotonicity does not apply.

Low memory enumeration. Even when a listing algorithm is time-efficient, i.e. polynomial total time or polynomial delay, turning this algorithm in a polynomial space one can be challenging. Indeed, as the size of the output can be exponential, polynomial space means that the algorithm may only store up to a poly-logarithm of the size of the output. As observed by Fukuda, introducing the notion of “compactness”, “an enumeration algorithm may not have to store all output in the active memory”: indeed, “some algorithms can simply dump each output object without storing it” \cite{Fuk96}. On the other hand, it is sometimes hard to achieve both output sensitive time and polynomial space, for instance because all the previous solutions need to be stored to check whether a new solution has been already generated or not, like in the approach by Cohen et al. \cite{CKS08}. When the absence of duplication can be ensured by the algorithmic structure, however, each solution can be output and thrown away, so in these cases polynomial space is often sufficient. In some cases, it is even possible to design output sensitive algorithms whose additional space is just the maximum size of the objects to be listed (e.g., chapters 3 and 4). One of the main ingredients of these kind of results is a stateless structure, which is the
capability of a recursive algorithm to rebuild the recursion stack when returning from a nested call. As most current output-sensitive approaches have a recursive nature, they can reach \( \Theta(n) \) nesting levels even for sparse graphs. In these cases, using the stack should be avoided: indeed, just storing one memory word per recursion level takes the memory usage to \( \Omega(n) \). We will show that our framework can be implemented in a stateless fashion, using as little as \( O(q) \) additional space.

**Case Study: Maximal Common Connected Induced Subgraphs** For any two given input graphs \( A \) and \( B \), a subgraph \( S \) of \( A \) is in common with \( B \) if \( S \) is isomorphic to a subgraph of \( B \); it is maximal if there is no other common subgraph that strictly contains it. The maximal common subgraph (MCS) problem requires discovering all the MCS’s of \( G \) and \( H \). The MCS problem can be constrained to connected and induced subgraphs (MCCIS) \cite{CCC+08, Koc01, KLW96}, where the latter means that all the edges of \( G \) between vertices in the MCS are mapped to edges of \( H \), and vice versa. The connectivity constraint is important to remove redundant solutions corresponding to compatible combinations of disjoint connected subgraphs, while the induced constraint is used to reduce the search space while still preserving the significance of the result \cite{CCC+08}.

Finding an output sensitive algorithm with polynomial space for the MCCIS implies a polynomial algorithm for the well-known graph isomorphism problem, whose current best bound is quasi-polynomial \cite{Bab16}. A weaker version corresponds to finding maximal isomorphisms corresponding to MCCISS. Finding an output sensitive algorithm with polynomial space for listing these isomorphisms is open, as several works \cite{CCC+08, Koc01, KLW96} adapt existing techniques for maximal clique enumeration without any guarantee. Indeed, by applying the transformation by Levi \cite{Lev73}, the maximal isomorphisms corresponding to MCSs reduces to maximal clique enumeration in a new graph, called product graph. A variation of this product graph by \cite{KLW96} allows to reduce the maximal isomorphism corresponding to MCCISS to listing special cliques. In particular, given \( A \) and \( B \), the vertices of the product graph \( G \) obtained by \( A \) and \( B \), are the pairs in \( V(A) \times V(B) \) and there is a black (resp. white) edge between \((u, x) \) and \((v, y) \) whether \( \{u, v\} \in E(A) \) and \( \{x, y\} \in E(B) \) (resp. \( \{u, v\} \notin E(A) \) and \( \{x, y\} \notin E(B) \)). Intuitively, an edge of \( G \) is black whether the assignments are compatible and the corresponding vertices are both connected in \( A \) and \( B \). Let \( G_B \) be the edge subgraph of \( G \) containing only black edges. For each isomorphism corresponding to MCCISS of \( A \) and \( B \), there is a maximal clique in \( G \) connected by black edges in \( G_B \), and vice versa. We call these subgraphs of \( G \) as BC-clique. Hence, in general, given a graph \( G \) whose edges are either black or white, we will show how to list all the maximal BC-cliques in \( G \) in an output sensitive fashion without storing all the solutions.

### 5.3 Enumeration framework for strongly accessible set system

Reverse search is a powerful enumeration tool that can be made stateless, so a natural question is in which cases it can be efficiently applied to list the maximal solutions\(^2\) of set systems. We provide answers to this question by introducing our framework for strongly accessible set systems. As this task is relatively complex, the road map below may help the reader.

#### 5.3.1 Road Map

We begin in Section \[5.3.2\] by highlighting some properties (in Definition \[5.1\]) that allow a set system to have a reverse search algorithm listing its maximal solutions. Looking at the literature, these properties are adopted by many enumeration algorithms, which explicitly or implicitly\(^3\)

\(^2\)Non maximal solutions can also be listed using reverse search but this is outside the scope of the chapter.

\(^3\)In cases such as \[AP90\] this logic is encoded in the element selection order in the computational tree.
rely on a routine, which we denote COMPLETE, that is applied to a suitable portion of the current maximal solution to generate another maximal solution (Property 6.1).

Routine COMPLETE is at the heart of these enumeration algorithms because it gives a method to generate new maximal solutions from known ones. Its properties can be used to avoid generating duplicate solutions, thus giving a way to ensure correctness without storing the output in memory. Unfortunately, keeping the fundamental properties of COMPLETE valid for (commutable) strongly accessible set systems can be hard (Lemmas 5.2 and 5.3).

In Section 5.3.3 we avoid this issue by giving a different definition of COMPLETE, based on a black box CHOOSE to select the next element to be added to the current partial solution $S'$. This induces a canonical ordering of the elements of a maximal solution $S$, which is not necessarily the sorted order of its elements, but it is rather dictated by the order in which CHOOSE extracts the elements to be added. Differently from before, COMPLETE takes polynomial time (iff CHOOSE takes polynomial time) and the canonical ordering guarantees that the vertices in any prefix of $S$ induce a (partial) solution $S' \in \mathcal{F}$. We use this to define a total order among the maximal solutions, which can be listed by the reverse search (Theorem 5.1) when a suitable condition (Requirement 5.1) is satisfied.

In Section 5.4 we refine our approach when the strongly accessible set system is commutable. We define CHOOSE so that the elements of $S$ are partitioned into layers (Definition 5.4): intuitively, the elements in the same layer can be chosen in any given order; instead, those in the next layer should be taken only after all the elements in the current layer have been consumed by CHOOSE. Although this mechanism of layers is significantly more complex, it gives a more refined way to generate new maximal solutions from the current one (Lemma 6.10). Its strength lays in generalizing the previous approaches [CKS08, LLRK80] based on restricted problems to the wider class of commutable set systems. We obtain improved bounds wherever the restricted problem can be solved efficiently (Theorems 5.2, 5.3, and 5.4).

Finally, in Section 5.5 we observe that our algorithms can be made stateless, simulating the traversal of the directed forest implicitly induced by these algorithms. This approach uses just $O(q)$ more space than solving the restricted problem.

5.3.2 Using reverse search: initial obstacles

From now on, we will assume that $\mathcal{F}$ is nonempty, as otherwise the listing problem is trivial. For a set system $(U, \mathcal{F})$, we define PARENT as a partial function whose domain is the set of maximal solutions for $(U, \mathcal{F})$: given a maximal solution $S$, either PARENT returns another maximal solution $S' \neq S$ or is undefined. In the latter case, we say that $S$ is a root.

**Definition 5.1.** A set system $(U, \mathcal{F})$ is called reverse searchable if it admits a PARENT function that satisfies the following sufficient conditions for any maximal solution $S$.

(A) $\text{PARENT}(S)$ can be computed from $S$ given the knowledge of $(U, \mathcal{F})$; also, a compatible order $\prec$ (that is, $\text{PARENT}(S) \prec S$) on the maximal solutions of $(U, \mathcal{F})$ must exist.

(B) $\text{ROOTS} = \{X : X \text{ is a root}\}$ can be computed given the knowledge of $(U, \mathcal{F})$.

(C) $\text{CHILDREN}(S) = \{X : \text{PARENT}(X) = S\}$ can be computed from $S$ given $(U, \mathcal{F})$.

Given a set system $(U, \mathcal{F})$ that is reverse searchable for a given PARENT function, all the maximal solutions for $(U, \mathcal{F})$ can be enumerated with no duplicates. Indeed, conditions $\text{(A)(C)}$ in Definition 5.1 define a directed forest whose vertices, i.e., the maximal solutions of $(U, \mathcal{F})$, can be listed by simulating a traversal, as proved next. A visual example of this structure is shown in Figure 5.3 (right), where $S1$ and $S5$ are the roots. This differs from classical reverse search algorithms (see Section 2.3.4 and Figure 2.4) in that there can be more than one root.
Lemma 5.1. Given a set system $(U, F)$ that is reverse searchable for a given parent function, all the maximal solutions for $(U, F)$ can be enumerated with no duplicates.

Proof. The algorithm to find all the maximal solutions works as follows. For each $S \in \text{ROOTS}$, invoke the following recursive procedure $\text{rec}(S)$. Given a solution $S$, $\text{rec}(S)$ simply calls $\text{rec}(S')$ for each $S' \in \text{CHILDREN}(S)$. This suffices to find all the solutions. Indeed, by contradiction, let $S$ be the minimum solution under $\prec$ that is not found. If $S$ is a root, it is found by Definition 5.1B. Otherwise, there exists $P = \text{PARENT}(S)$, which is found since $P \prec S$ by Definition 5.1A. As $S \in \text{CHILDREN}(P)$, $S$ is then found by Definition 5.1C. Moreover, each solution is found exactly once. As each solution $S$ has only one parent and there are no cycles by Definition 5.1A, we have that there is only one $P$ such that $S \in \text{CHILDREN}(P)$.

To actually implement Definition 5.1, many algorithms rely (directly or indirectly) on the following complete function, whose domain is the whole $F$, not only the maximal sets in $F$ (Tsukiyama et al. [TIAS77] inspired many of these algorithms).

Property 5.1. A set system $(U, F)$ is reverse searchable if the following conditions are satisfied.

- The compatible order between maximal solutions, i.e. $\prec$, is the lexicographical order.
- Given a solution $X \in F$, not necessarily maximal, $\text{COMPLETE}(X)$ returns the lexicographic minimum among all the maximal solutions containing $X$.
- $\text{PARENT}(S)$, if it exists, is defined as $\text{COMPLETE}(X)$ for some $X \subset S$, with $\text{COMPLETE}(X) \neq S$.
- A child $S' \in \text{CHILDREN}(S)$, if it exists, is defined as $S' = \text{COMPLETE}(X \cup \{w\})$ for some $X \subset S$ and $w \in U$.

Proof. If we fail to compute $\text{PARENT}(S)$, then $S$ is a root. Otherwise, $\text{PARENT}(S) = \text{COMPLETE}(X) \neq S$ for some $X \subset S$, and the above conditions imply that $\text{COMPLETE}(X) \prec S$ as $S$ is one of the maximal solutions containing $X$ but it cannot be the lexicographic minimum. Finally, $\text{CHILDREN}(S)$ can be computed using $\text{COMPLETE}$ as stated in the last condition.

The conditions in Property 5.1 imply those in Definition 5.1 (the converse is not necessarily true). Unfortunately this approach cannot be applied to strongly accessible set systems, as the complete function in Property 5.1 can be NP-hard to compute, e.g. in the case of $bc$-cliques.

Lemma 5.2. Given a graph $G$ whose edges are either black or white and a non-maximal $bc$-clique $X$ of $G$, it is NP-hard to find the lexicographically minimum among the maximal $bc$-cliques containing $X$.

Proof. We prove that a $\text{COMPLETE}(X)$ function that returns the lexicographically minimum $bc$-clique containing $X$ can be used to solve a SAT problem in polynomial time, by building a graph with vertices linear in the amount of the clauses and variables in the formula.

Given a SAT formula with $n$ variables $x_1 \ldots x_n$ and $k$ clauses $d_1 \ldots d_k$, we build the graph in Figure 5.4 whose vertices are $C_1 \ldots C_k$, $T_1 \ldots T_n$, $F_1 \ldots F_n$ and $Y_1 \ldots Y_n$, labelled increasingly in this order (i.e., vertices $C_1 \ldots C_k$ have smaller label than all other vertices). Each $Y_i$ is connected with a black edge to $T_i$ and $F_i$ and, except for $Y_1$, also with $T_i - 1$ and $F_i - 1$. Each $C_i$, which corresponds to $d_i$, is connected with black edge to $T_j$ (resp. $F_j$) whether $d_i$ contains a positive (resp. negative) occurrence of $x_j$. Hence, vertices in $C_1 \ldots C_k$ are connected with black edge to an arbitrary amount of $T_i$ and $F_i$ vertices, but not to any $Y_i$ vertex. All other pairs of vertices are connected with a white edge, except for the pairs $T_i, F_i$ (symbolized...
by the red crossed edge in Figure 5.4. It is straightforward to see that any maximal \(bc\)-clique in this graph will contain exactly one between any pair of vertices \(Ti,Fi\), and that any maximal \(bc\)-clique containing all vertices in \(C1 \ldots Ck\) will be lexicographically smaller than any that does not contain all of them (as they have the smallest labels).

Consider \(\text{COMPLETE}({Y1})\), i.e., the lexicographically smallest maximal \(bc\)-clique containing \(Y1\). Any \(bc\)-clique containing \(Y1\) and all \(Ci\) vertices represents a satisfying assignment for the formula at hand. Indeed, in order for each \(Ci\) vertex to be reachable from \(Y1\) with black edges, at least one of the \(Tj\) or \(Fj\) vertices connected to \(Ci\) must be in the \(bc\)-clique; the set of \(Ti\) and \(Fi\) vertices in the \(bc\)-clique will thus give us the value (true or false) of the corresponding variable \(x_i\) (recall that we cannot have the pair \(Tj-Fj\) in the same \(bc\)-clique). Hence, in order to verify that the formula is satisfiable, we only need to compute \(\text{COMPLETE}({Y1})\) and check whether this contains all \(Ci\) vertices.

Lemma 5.3 represents an obstacle to implement reverse search algorithms using Property 5.1. Furthermore, a polynomial cost per solution cannot be guaranteed in the general case, as per the following lemma.

**Lemma 5.3.** An algorithm that generates all the \(\alpha\) maximal solutions of any strongly accessible set system has a worst case time complexity of \(\Omega(\alpha^{2^\alpha/2})\), unless \(\text{seth}\) is false.

**Proof.** Suppose by contradiction that an algorithm \(A\) exists to list all the maximal solutions in an independence set system, which is a particular case of strongly accessible set systems, in \(O(\alpha 2^\alpha)\) time for some \(k > 2\). This implies the existence of an algorithm \(B\) for SAT that runs in \(O(n2^{\frac{2n}{k}})\) time, where \(n\) is the number of variables. Indeed, using the reduction in [LLRK80], \(B\) first transforms a formula of SAT in a set system, having \(\alpha = n\) maximal solutions iff the formula is not satisfiable. Note that this reduction gives \(g = 2n\) for the size of the largest maximal set. At this point, \(B\) can execute \(A\) waiting \(O(n \cdot 2^{\frac{2n}{k}})\) time: the formula is not satisfiable iff \(A\) finds \(n\) solutions and terminates within this upper bound. This contradicts the hypothesis that \(k > 2\) as it must be \(k \leq 2\) by \(\text{seth}\).

### 5.3.3 Using reverse search: canonical ordering

We give a definition of \(\text{COMPLETE}\) that is different from that in Property 5.1, exploiting the definition of strongly accessible set systems: for any \(X, Y \in \mathcal{F}\) such that \(X \subset Y\), there exists
Algorithm 12: Basic framework for strongly accessible set systems

1. Input: Strongly accessible set systems $(\mathcal{U}, \mathcal{F})$
2. Output: All maximal $X \in \mathcal{F}$
3. for $R \in \text{ROOTS}$ do
   1. $\text{SPAWN}(R)$
4. Function $\text{COMPLETE}(X, A)$
   1. while $X_A^+ \neq \emptyset$ do
      1. $x \leftarrow \text{CHOOSE}(X, A)$
      2. $X \leftarrow X \cup \{x\}$
   3. return $X$
5. Function $\text{CHOOSE}(X, A)$
   1. return $\min X_A^+$

Proof. Fact 5.1.1 holds as $Y = \emptyset \subseteq X$ in $\mathcal{F}$ implies that there exists $z \in X \setminus Y$ such that $Y \cup \{z\} \in \mathcal{F}$ by definition of strongly accessible set systems. Hence, $z \in Z \cap X$ meaning that $Z \neq \emptyset$ if a nonempty solution exists. Fact 5.1.2 directly follows from the definition of strongly accessible set system.

Given $X \in \mathcal{F}$, we denote by $\text{SOURCE}(X)$ the smallest element in $X \cap Z$, which exists by Fact 5.1.1. Given a set $A \subseteq \mathcal{U}$, we define $\text{COMPLETE}(X, A)$ as a procedure that uses Fact 5.1.1 while $X_A^+$ is nonempty, in each iteration an element $x \in X_A^+$ is selected by a user-defined black box $\text{CHOOSE}(X, A)$, and $x$ is added to $X$ (see Algorithm 12 where $\text{CHOOSE}(X, A)$ returns the minimum). Different implementations of $\text{CHOOSE}$ lead to different results for our framework. We use $\text{COMPLETE}(X)$ as a shorthand for $\text{COMPLETE}(X, \mathcal{U})$.

Let us consider $\text{COMPLETE}((\text{SOURCE}(S), S))$: it naturally induces a permutation of the elements in $S$, called canonical order $s_1, \ldots, s_{|S|} \in S$, where $\text{SOURCE}(S)$ is the first element $s_1$, and each subsequent element $s_i$ is chosen during the iterations ($i = 2, 3, \ldots, |S|$) of $\text{COMPLETE}$ according to $\text{CHOOSE}$. This differs from Section 5.3.2 where the increasing order of $S$ is implicitly adopted.

---

In the case of $bc$-cliques, we have that every vertex is a $bc$-clique (i.e., $Z = \mathcal{U}$). However, in strongly accessible set systems, it can be $Z \subseteq \mathcal{U}$: consider the set of simple paths that contain a leaf in a tree, where $\mathcal{U}$ are the vertices of the tree; observe that $Z$ is the set of leaves.
**Example 5.1.** In the case of the bc-cliques in Figure 5.1, let $X = \{1,2\}$, $X^+$ is the set $\{5,6\}$ as both the vertices 5 and 6 can be added to $X$ while maintaining the bc-clique property (3 is not in $X^+$ as it is not black connected to $X$). $\text{COMPLETE}(X)$ adds to $X$ the vertex 5, since $5 < 6$; it then adds 3, as $X^+ = \{3,6\}$, and finally 6, as $X^+ = \{6\}$. When $X = \{1,2,3,5,6\}$, $X^+ = \emptyset$, as $X$ cannot be enlarged anymore maintaining the bc-clique property. A visual representation of the corresponding canonical order is shown top-down in Figure 5.5(b).

Note that $\text{COMPLETE}(X)$ may not return the lexicographically smallest bc-clique containing $X$, but we will show that its properties allow us to define a suitable PARENT function.

Given $S$ and $0 < j \leq |S|$, we define $S'[j] = s_1, \ldots, s_j$ as the $j$th PARENT of the canonical order of $S$. We observe that the following prefix-closure property holds for any CHOOSE.

**Lemma 5.4.** For any maximal solution $S \in \mathcal{F}$, we have $S'[j] \in \mathcal{F}$ for any $1 \leq j \leq |S|$.

**Proof.** We proceed by induction. $S[1] = \{s_1\} = \{\text{source}(S)\} \in \mathcal{F}$, by definition of $Z$ as $s_1 \in S \cap Z$. Assuming that $S'[j] \in \mathcal{F}$, we have that also $S'[j+1]$, which is obtained from $S'[j] \cup \{s_{j+1}\}$, is in $\mathcal{F}$ by definition of $S'[j]$ as $s_{j+1} \in S'[j]$.

Along with CHOOSE, our framework also requires a user-defined total ordering $\preceq$ between maximal solutions that satisfies the following constraint, needed to satisfy Definition 5.1A.

**Requirement 5.1.** For any maximal solution $S \in \mathcal{F}$, $\text{COMPLETE}(S'[j]) \preceq S$ for any $1 \leq j \leq |S|$.

**Lemma 5.5.** Algorithm 12 fulfills Requirement 5.1, where $\preceq$ is defined by lexicographically comparing the canonical orders of maximal solutions.\(^\dagger\)

**Proof.** If $\text{COMPLETE}(S'[j]) = S$, we have nothing to prove. Otherwise, let the canonical order of $\text{COMPLETE}(S'[j])$ and $S$ be respectively $t_1, \ldots, t_{|T|}$ and $s_1, \ldots, s_{|S|}$ and let $i > j$ be the smallest index such that $t_i \neq s_i$. We have $S[i-1] = T[i-1]$, so $t_i$ is the minimum of $S[i-1]^+$ and, since $s_i \in S[i-1]^+$, we have $t_i \leq s_i$.

**Example 5.2.** In the case of bc-cliques, refer to Figure 5.1, we can see that the bc-cliques $S_1 = \{1,2,3,5,6\}$ and $S_2 = \{3,4,5\}$ have respectively $(1,2,5,3,6)$ and $(3,5,4)$ as canonical orders. By comparing the two sequences we see that $(1,2,5,3,6) \prec (3,5,4)$ as $1 < 3$.\(\blacktriangleright\)

Continuing the description of our framework, we can now define the notions required for the reverse search (Definition 5.1).

**Definition 5.2.** Given a maximal solution $S$, define $\text{PI}(S)$ as the earliest element $s_j$ in the canonical order of $S$ such that $\text{COMPLETE}(S[j]) = S$: if $j > 1$, then $\text{core}(S) = S[j-1]$ and $\text{PARENT}(S) = \text{COMPLETE}(\text{core}(S))$; otherwise, $S$ has no parent and thus $S \in \text{ROOTS}$.

**Example 5.3.** Consider the bc-cliques $S_1 = \{1,2,3,5,6\}$ and $S_2 = \{3,4,5\}$ in Figure 5.1. Their canonical order is respectively $(1,2,5,3,6)$ and $(3,5,4)$. Since $\text{COMPLETE}(\{1\}) = S_1$, we have $S_1 \in \text{ROOTS}$. For $S_2$, since both $\text{COMPLETE}(\{3\})$ and $\text{COMPLETE}(\{3,5\})$ give $S_1$, $\text{PI}(S_2) = 4$ and $\text{Core}(S_2) = \{3,5\}$. As $\text{COMPLETE}(\text{Core}(S_2)) = S_1$ we have that the parent of $S_2$ is $S_1$. It is worth observing that $S_1 = \text{PARENT}(S_2) < S_2$ (see Example 5.2).\(\blacktriangleright\)

By Requirement 5.1, we have $\text{PARENT}(S) \prec S$ as $\text{PARENT}(S) = \text{COMPLETE}(\text{core}(S)) \neq S$, and thus it satisfies Definition 5.1A. From Definition 5.2 we have that $S$ is a root iff $\text{COMPLETE}(S[1]) = S$, thus satisfying Definition 5.1B. As for Definition 5.1C we observe that $\text{CHILDREN}(S)$ is defined as all the solutions $S'$ for which $\text{PARENT}(S') = S$.

\(\dagger\)We say that $S \preceq T$ for any two maximal solutions $S$ and $T$ when $S \preceq T$ and $S \neq T$.

\(\ddagger\)Given any two maximal solutions $S \neq T$, consider their canonical orders $s_1, \ldots, s_{|S|}$ and $t_1, \ldots, t_{|T|}$. Let $i$ be the smallest index for which $s_i \neq t_i$, which exists by their maximality. Then, $S \prec T$ iff $s_i < t_i$.\(\blacktriangleleft\)
Theorem 5.1. Given a total order $\leq$ among maximal solutions and a choose function such that Requirement\footnote{This is the case, for instance of the bc-cliques in Figure \ref{fig:example}.} \ref{req:choose} is fulfilled, and given \textsc{children}(\textit{S}), any strongly accessible set system is reverse searchable.

We now discuss how to design \textsc{children}(\textit{S}). Let \textsc{children}(\textit{S}, \textit{w}) be the function that returns all the maximal solutions $S' \in \textsc{children}(\textit{S})$ such that $w = \text{pi}(S')$. Clearly, \textsc{children}(\textit{S}) = $\bigcup_{w \in \mathcal{U}} \text{children}(\textit{S}, \textit{w})$, and the union is disjoint since a solution has a unique \text{pi} (Definition \ref{def:pi}).

With reference to Algorithm \ref{alg:impl}, we show how to implement \textsc{children}(\textit{P}, \textit{w}) given a maximal solution \textit{P} (the parent) and one element \textit{w} $\in$ $\mathcal{U}$.

Lemma 5.6. For any two maximal solutions $S$ and $P$, such that $P = \text{parent}(\textit{S})$, there exists $X \subset P$ and $w \notin X$ such that $S = \text{complete}(X \cup \{w\})$, where $X = \text{core}(\textit{S})$ and $w = \text{pi}(\textit{S})$.

Proof. According to Definition \ref{def:pi}, given a solution \textit{S}, whose canonical order is $s_1, \ldots, s_{|S|}$, the relationship between \textit{S} and $P = \text{parent}(\textit{S})$ is such that $P = \text{complete}(\text{core}(\textit{S}))$, where core($\textit{S}$) = $s_1, \ldots, s_j$ for some $j$. For each $i$, with $1 \leq i \leq j$, $s_i \in S \cap P$; $s_{j+1} = \text{pi}(\textit{S})$ is in $S \setminus \text{core}(\textit{S})$. Note that given \textit{S} and $P = \text{parent}(\textit{S})$, we have core($\textit{S}$) $\subset$ $P$, \text{pi}(\textit{S}) $\notin$ core($\textit{S}$), and $S = \text{complete}(\text{core}(\textit{S}) \cup \{\text{pi}(\textit{S})\})$.

Lemma \ref{lem:complete} essentially says that we can exploit algorithmically the inclusion \text{children}(\textit{P}, \textit{w}) $\subseteq$ \{\text{complete}(X $\cup$ \{\textit{w}\}) : $X \subset P$, $X \cup$ \{\textit{w}\} $\in$ $\mathcal{F}$\}. It is worth observing that while core($\textit{S}$) is a prefix of $\textit{S}$, it is not necessarily also a prefix of $\textit{P}$ as the canonical order of $\textit{P}$ could differ (including its source). This implies that we do not know a priori which subsets $X \subset P$ may be the core of some child of $\textit{P}$. The only hint we have is that $X \cup$ \{\textit{w}\} $\in$ $\mathcal{F}$ as its elements must be a prefix of $\textit{S}$ when put into canonical order\footnote{This is the case, for instance of the bc-cliques in Figure \ref{fig:example}.}

\textsc{children}(\textit{P}, \textit{w}) computes candidate children $S$ as \text{complete}(X $\cup$ \{\textit{w}\}), for each possible $X \in 2^P$ such that $X \cup$ \{\textit{w}\} $\in$ $\mathcal{F}$: it retains $S$ if $P = \text{parent}(\textit{S})$, $w = \text{pi}(\textit{S})$, and $X = \text{core}(\textit{S})$. While the first two conditions are sufficient to ensure that $S \in \text{children}(\textit{P}, \textit{w})$, the third one is required to output each child once: if we find an $S$ such that $P = \text{parent}(\textit{S})$, $w = \text{pi}(\textit{S})$ but $X \neq \text{core}(\textit{S})$, we discard it as surely we will find $S$ again when $X = \text{core}(\textit{S})$.

Example 5.4. Consider the case in Figure \ref{fig:example}. For the sake of simplicity we will report the elements of a solution in the canonical order. When $P = \{3, 5, 4\}$, $X = \{5\}$ and $w = 8$, the candidate child $S$ in Algorithm \ref{alg:impl} is \text{complete}(\{5, 8\}) = \{2, 5, 8, 7\}. However, parent($2, 5, 8, 7$) = \{1, 2, 5, 3, 6\}, thus we discard $S$. Now consider $P = \{1, 2, 5, 3, 6\}$, $X = \{2\}$ and $w = 8$: the corresponding candidate child $S$ is \{2, 5, 8, 7\}. Even though parent($2, 5, 8, 7$) = \{1, 2, 5, 3, 6\}, we have core($\{2, 5, 8, 7\}$) = \{2, 5\} $\neq$ $X$ and hence we discard $S$ again. When $P = \{1, 2, 5, 3, 6\}$, $X = \{2, 5\}$, and $w = 8$, we obtain the same $S$ a third time, but now $S$ is not discarded.

The running time of Algorithm \ref{alg:impl} is stated in the following lemma. It is worth noting that the delay is close to the conditional lower bound of Lemma \ref{lem:lower}. Recall that $\mathcal{T}$ is the time needed to check whether $X \in \mathcal{F}$.

Lemma 5.7. Algorithm \ref{alg:impl} has delay $O(q2^q \cdot |\mathcal{U}|^2 \cdot \mathcal{T})$.

Proof. \text{choose} can clearly be computed in $O(|\mathcal{U}| \cdot \mathcal{T})$ time. This means that we can compute \text{complete} in $O(q|\mathcal{U}| \cdot \mathcal{T})$. We can compute parent, $\text{pi}$ and core in $O(q|\mathcal{U}| \cdot \mathcal{T})$ too, since it is enough to check for every $j$ if \text{choose}($S[j]$) = $s_j$, as the greatest $j$ for which this does not happen identifies the core and the $\text{pi}$. To compute parent, we then only need to run \text{complete} once. Since we run these computations $2^q \cdot |\mathcal{U}|$ times for every solution found, the thesis follows.
5.4 Refined framework for connected hereditary properties

A natural question is whether a more refined method to generate children is possible. As we show in this section, this has a positive answer at least for connected hereditary graph properties, due to what we call the commutable property:

**Definition 5.3 (Commutable property)**. A strongly accessible set system \((U, F)\) verifies the commutable property if, given any \(X, Y \in F\) such that \(X \neq \emptyset\) and \(X \subset Y\), for any \(a, b \in Y \setminus X\), if \(X \cup \{a\} \in F\) and \(X \cup \{b\} \in F\) then \(X \cup \{a, b\} \in F\).

We will show that connected hereditary properties (that is, the set systems induced by connected hereditary properties) also verify this commutable property, and that this will allow us to design a smarter way to generate children solutions.

**Lemma 5.8.** The set systems induced by connected hereditary graph properties are strong accessible and verify the commutable property.

**Proof.** Let \((U, F)\) be the set system induced by a connected hereditary property, and let \(X, Y \in F\) be two solutions such that \(X \subset Y\). Firstly, there must be \(v \in Y \setminus X\) connected to some vertex in \(X\), otherwise \(Y\) would not be connected, as no vertex in \(X\) would be connected to any vertex in \(Y \setminus X\); since \(Y \in F\) and thus \(Y\) is connected, we have some \(v \in Y \setminus X\) connected to some vertex in \(X\), and \(X \cup \{v\} \in F\) by definition of connected hereditary, thus \((U, F)\) is strongly accessible. Furthermore, assume \(X \neq \emptyset\), and let \(a, b \in Y \setminus X\) be two elements such that \(X \cup \{a\} \in F\) and \(X \cup \{b\} \in F\). We thus have that \(X\) is connected, and so are \(X \cup \{a\}\) and \(X \cup \{b\}\). Since \(X\) is not empty it must be that \(X \cup \{a, b\}\) is connected. We have that \(X \cup \{a, b\}\) is a connected subgraph of \(Y\), which by definition of connected hereditary property implies \(X \cup \{a, b\} \in F\), thus \((U, F)\) verifies the commutable property. \(\square\)

![Figure 5.5](image-url)

Figure 5.5: (a) A maximal bc-clique, (b) its canonical order 1, 2, 5, 3, 6 from Section 5.3.3 and (c) its canonical order 1, 2, 5, 6, 3 from Section 5.4. (d) A non-maximal bc-clique \(X\) and its set \(X^+\).

Again, we define \textsc{choose} (and thus \(\preceq\)) so that \textsc{complete} creates a canonical order. To do so we first need to introduce the notion of layer.

**Definition 5.4.** Given \(X \in F\) and a starting element \(t \in X \cap Z\), define inductively \(B_i\).

- \(B_0 = \{t\}\)
- \(B_i = B_{i-1} \cup (B_i^+ \cap X)\), for \(i > 0\)

Then, for any \(y \in X \cup X^+\), its layer relatively to \(X\) from \(t\) is \(\text{lay}^X_t(y) = \min\{i : y \in B_i^+\}\).

\(^8\text{Note that }B_i\text{ is made only of elements from }X\text{ whereas }B_i^+\text{ is made of elements from }X \cup X^+\).
Where we recall that \( Z = \{ x \in U : \{ x \} \in F \} \). We simply write \( \text{LAY}^X \) when \( t = \text{source}(X) \). This is useful for defining \( \text{choose}(X,A) \), as given in Algorithm 13, for each element \( y \in X^+_A \), associate a pair \( (\text{LAY}^X(y), y) \) with \( y \); return the element \( y \in X^+_A \) whose associated pair is the lexicographically minimum. When \( \text{choose} \) is plugged into \( \text{complete} \), as we saw in Section 5.3.3, we obtain the canonical order of a maximal solution \( S \) and, consequently, the notions of \( \text{parent} \), \( \text{core} \), \( \text{pi} \).

**Algorithm 13:** Refined framework for connected hereditary properties: CHILDREN + \( \text{choose} \)

```plaintext
Function \( \text{CHILDREN}(P, w) \)
  if \( w \in P \) then return \( \emptyset \)
  foreach \( R \in \text{rest-}P(P, w) \setminus \{ P \} \) do
    foreach \( s \in (R \cap Z) \setminus \{ w \} \) do
      \( \text{core}S \leftarrow \text{complete}([s], R)_w \)
      \( S \leftarrow \text{complete}(\text{core}S \cup \{ w \}, U) \)
      if \( (\text{parent}(S), \text{pi}(S), \text{r}(S), \text{source}(S)) = (P, w, R, s) \) then
        yield return \( S \)

Function \( \text{choose}(X, A) \)
  return \( \arg \min_{y \in X^+_A} (\text{LAY}^X(y), y) \)
```

**Example 5.5.** Consider the bc-clique in Figure 5.5(a), where \( X = \{ 1, 2, 3, 5, 6 \} \) is the set of its vertices, and the starting element is \( t = \text{source}(X) = 1 \). The elements of \( X \) have layers \( 0, 1, 3, 2, 2 \), respectively, relatively to \( X \) from \( t \) (e.g. \( \text{LAY}^X(2) = 1 \) and \( \text{LAY}^X(5) = 2 \)). Running \( \text{complete}(\{ \text{source}(X) \}, X) \) with the choose function in Algorithm 13, we obtain the canonical order in Figure 5.5(c) as follows. We add 2 to \( \{ 1 \} \) as it is the only option for \( \text{choose} \). Next, we can choose between 5 and 6 for extending \( \{ 1, 2 \} \); we add 5, as \( \langle \text{LAY}^X(5), 5 \rangle \) is smaller than \( \langle \text{LAY}^X(6), 6 \rangle \). This gives \( \{ 1, 2, 5 \} \) and \( \text{choose} \) selects 6 between the next options 6 and 3. Finally, it selects 3, obtaining \( X \) and the above canonical order. Also, we observe that \( X \in \text{ROOTS} \) as \( \text{complete} \) applied to each prefix \( S[j] \) (i.e. \( 1; 1, 2; \ldots; 1, 2, 5, 6, 3 \)) always gives \( X \).

Consider now the other two bc-cliques, \( X_1 = \{ 3, 4, 5 \} \) and \( X_2 = \{ 2, 5, 7, 8 \} \), from Figure 5.1. Their canonical order is \( 3|5|4 \) (with layers in increasing order separated by "|") and \( 2|5, 8|7 \). Also, \( X \) is their parent: for example, \( X = \text{parent}(X_1) = \text{complete}(\{ 3, 5 \}) \) and thus \( \text{core}(X_1) = \{ 3, 5 \} \) and \( \text{pi}(X_1) = 4 \). We observe that during the execution of \( \text{complete} \), the source changes (hence the canonical order): \( \{ 3, 5 \} \rightarrow \{ 2, 5|3 \} \rightarrow \{ 1, 2|5, 3 \} \rightarrow \{ 1, 2, 5, 6|3 \} \). (Also, \( \text{core}(X_2) = \{ 2, 5 \} \) and \( \text{pi}(X_2) = 8 \).) In general, during the execution of \( \text{complete} \) the source of \( X \) may change and, consequently, the layers of the elements in \( X \cup X^+ \) change accordingly. This does not affect \( X^+ \), but only the relative order of its elements.

Recalling that we have to meet Requirement 5.1, we define the order \( \preceq \) between any two maximal solutions \( S \) and \( T \) as follows. Let their canonical order be \( s_1, \ldots, s_{|S|} \) and \( t_1, \ldots, t_{|T|} \). Then \( S \prec T \) iff \( \langle \text{LAY}^S_{s_j}(s_j), s_j \rangle \) is lexicographically smaller than \( \langle \text{LAY}^T_{t_j}(t_j), t_j \rangle \), for the smallest \( j \) giving different pairs (note that \( j \) exists as both solutions are maximal).

We are now ready to state the version of Lemma 5.5 for commutable set systems, which can be proven in the same way as the version for strongly accessible set systems.

**Lemma 5.9.** When combined with \( \text{choose} \) from Algorithm 13, \( \text{complete} \) in Algorithm 12 fulfills Requirement 7.1.

We finally need to implement \( \text{CHILDREN} \) to meet the conditions of Theorem 5.1 for reverse search. A key ingredient is the concept of restricted problem, originally introduced by Lawler.
et al. [LLRK00]: given a maximal solution $P \in \mathcal{F}$ and an element $w \in \mathcal{U} \setminus P$, the restricted problem REST-P($P, w$) asks to list all maximal solutions $R \neq P$ in the reduced set system $(P \cup \{w\}, \mathcal{F})$. Under suitable conditions, Cohen et al. [CKS08] remarkably prove that we can efficiently enumerate the maximal solutions in (connected) hereditary systems $(\mathcal{U}, \mathcal{F})$ if we can do that in its restricted version for $(P \cup \{w\}, \mathcal{F})$. We will assume that we can enumerate the solutions of any restricted problem REST-P($P, w$) in at most $\mathcal{R}_T$ time, and using at most $\mathcal{R}_S \geq q$ space. Moreover, we will use $\mathcal{R}_B$ to denote an upper bound on the number of solutions of any restricted problem. Using REST-P($P, w$) as a black box we give some necessary conditions to compute CHILDREN($P, w$), recalling that CHILDREN($P$) = $\bigcup w \in \mathcal{U}$ CHILDREN($P, w$).

The main conceptual step is showing that examining the solutions of REST-P($P, w$) is enough for our goal. Define the truncated function COMPLETE($X, A$)$_{|w}$ as follows: during the execution of COMPLETE, if CHOOSE($X, A$) returns $w$ then stop the execution and return the current $X$ (so that $w \notin X$). We need the above truncated function and this key lemma to implement CHILDREN($P, w$) in Algorithm [13]

**Lemma 5.10.** For any maximal solution $P$ and $w \in \mathcal{U}$, we have that CHILDREN($P, w$) is contained in the set of maximal solutions $S$ such that

1. there exists $R \in$ REST-P($P, w$) (with $R \neq P$ and $w \notin P$) and
2. $S = \text{COMPLETE} \text{(COMPLETE}([\{s\}, R]_{|w}) \cup \{w\}, \mathcal{U}) \text{ (with } s \in (R \cap Z) \text{ and } s \neq w)$.

**Proof.** We will prove that, for any $S$ such that PARENT($S$) = $P$, the two conditions of the lemma hold by choosing $w = \pi(S)$, $s = \text{source} (S)$ and $R = R(S) = \text{COMPLETE} \text{(COMPLETE}([\{s\}, R]_{|w}) \cup \{\pi(S)\}, \text{PARENT} (S) \cup \{\pi(S)\})^T$.

During the proof, we will use repeatedly the following key fact about our CHOOSE function: for any $Y \subseteq A$, if $b = \text{CHOOSE} (X, A)$ belongs to $(X \cup Y)^+_A$, then $b = \text{CHOOSE} (X \cup Y, A)$ (provided that the starting element stays the same). Indeed, anything in $Y \setminus X$ must have a layer at least as large as $b$, so anything in $(X \cup Y)^+_A$ that is not in $X^+_A$ must have a larger layer than $b$. Note that this property does not hold if we use the order of Section 5.3.3.

By construction of COMPLETE, $R(S)$ is a maximal solution in PARENT($S$) $\cup \{\pi(S)\}$ and it is different from PARENT($S$), as $\pi(S) \in R(S)$, provided that we can prove $\pi(S) \notin \text{PARENT} (S)$. Suppose by contradiction that $w = \pi(S) \in \text{PARENT} (S)$, and let $y = \text{CHOOSE} \text{(CORE} (S))$ be the first element that COMPLETE adds to CORE($S$) when generating PARENT($S$). Note that $y \neq w$ by definition of CORE. Clearly $y \notin S$, as otherwise CORE($S$) $\cup \{y\}$ would be a prefix of $S$ and so $y = \pi(S) = w$. By the commutable property, the fact that CORE($S$) $\cup \{w\} \cup \{y\} \subseteq \text{PARENT} (S)$ and both CORE($S$) $\cup \{w\}$, CORE($S$) $\cup \{y\} \in \mathcal{F}$ implies that $y \in (\text{CORE} (S) \cup \{w\})^+$. By the aforementioned key fact we then have that $y = \text{CHOOSE} \text{(CORE} (S) \cup \{w\})$, so COMPLETE($\text{CORE} (S) \cup \{w\}) \neq S$, which contradicts the definition of CORE($S$) and $\pi(S)$. This proves the first part of the statement.

Regarding the second part, it is enough to prove that COMPLETE([\{s\}, R]$_{|w}$ = CORE($S$). We will only consider the case in which CORE($S$) $\cup \{w\} \neq S$, as otherwise the proof is trivial (since $R(S) = S$). Let $s = s_1, \ldots, s_\text{cs}$ be the canonical order of CORE($S$), where cs = |CORE($S$)| and $w$ is denoted as $s_{\text{cs}+1}$. Suppose now that the first $cs$ elements added by COMPLETE($[\{s\}, R]$ are not $s_{2}, \ldots, s_{\text{cs}+1}$ and let $r_i \neq s_i$, $i > 1$, be the first “wrong” element that is added. Thus we have $r_i = \text{CHOOSE} (S[i-1], R)$, recalling that $S[i-1] = \{s_1, \ldots, s_{i-1}\}$. Note that $r_i \notin S$, as that would contradict the definition of canonical order. By the commutable property and the fact that CORE($S$) $\cup \{w\} \cup \{r_i\} \subseteq R$, we can easily prove (by induction) that $r_i \in S[j]_R$ for $j = i - 1, \ldots, cs + 1$. Since $S[cs + 1] = \text{CORE} (S) \cup \{w\}$, we have $r_i \in (\text{CORE} (S) \cup \{w\})^+_R$ and, by

\[\text{Note that } R(S) \text{ is well defined as } \text{CORE} (S) \cup \{\pi(S)\} \in \mathcal{F}, \text{ by Lemma 5.4, as it is a prefix of } S.\]
the aforementioned key fact, \( r_i = \text{choose}(	ext{core}(S) \cup \{w\}, R) \). This implies that \( r_i \) is a better candidate than \( s_{i+2} \) for \text{choose}: this continues to be true if we consider the whole \( U \) instead of just \( R \), so the first step of \text{complete(core}(S) \cup \{w\}) does not add \( s_{i+2} \). As a result, we have \text{complete(core}(S) \cup \{w\}) \neq S \), which is a contradiction. \( \square \)

We remark that Lemma \ref{lem:source-identifies} does not necessarily hold if we use the canonical order from Section \ref{sec:order} (as briefly noted in the proof), and avoids us to blindly target all of \( X \subset P \cup \{w\} \).

We observe that different solutions \( R \in \text{rest-}P(P, w) \) could lead to the same maximal solution \( S \). We avoid this by checking that \( R = r(S) \) when we generate a child, as \( r(S) \) is computed deterministically from \( S \).

The following theorem holds.

**Theorem 5.2.** Given a solution \( P \), an element \( w \in U \setminus P \), a solution \( R \neq P \) of the restricted problem \( \text{rest-}P(P, w) \), there exists an algorithm that enumerates all maximal solutions \( S \) such that \( \text{parent}(S) = P \), \( \text{pt}(S) = w \), \( r(S) = R \) in time \( O(q^3|U|\mathcal{F}_T) \) and space \( O(\mathcal{G}_S) \).

**Proof.** The proof follows from Lemma \ref{lem:source-identifies} Indeed, note that \( \text{source}(S) \in \text{core}(S) \subset R(S) \). As the tuple \( \langle \text{parent}(S), \text{pt}(S), \text{start}(S), r(S) \rangle \) uniquely identifies \( S \), and there are at most \(|R \setminus \{w\}| < q \) possible sources, there are at most \( q \) possible solutions to test. As each can be found by using Lemma \ref{lem:source-identifies} in time \( O(q^3|U|) \),\(^{10}\) the statement holds. At most \( O(q) \) space is used by the algorithm, giving a total space usage of \( O(q + \mathcal{G}_S) = O(\mathcal{G}_S) \). \( \square \)

As we scan all the possible solutions of the restricted problem, by Definition \ref{def:reverse-searchable} we obtain that Algorithm \ref{alg:12} using \text{children} and \text{choose} routines in Algorithm \ref{alg:13} makes any set system induced by a connected hereditary property reverse searchable.

**Theorem 5.3.** The delay of Algorithm \ref{alg:12} using routines in Algorithm \ref{alg:13} is \( O(\mathcal{F}_T \cdot |U| + q^3|U| \cdot \mathcal{G}_T \cdot \mathcal{G}_B \cdot |U|) \).

**Proof.** For every solution, we solve \(|U| \) times the restricted problem. Then, for every solution of the restricted problem, we run the computations described in Lemma \ref{lem:source-identifies}. The total running time follows easily. \( \square \)

### 5.5 Stateless enumeration and memory usage

The recursive version of our algorithms do not yet guarantee polynomial space: indeed, the number of nested recursive calls could be non-polynomial. Moreover, Algorithm \ref{alg:13} stores the solutions of the restricted problem, which may be non-polynomial in number.

We address the first of these problems by removing the explicit recursion. The state of the computation inside a certain recursive call is fully determined by the variables \( P, w, X \) in Algorithm \ref{alg:12} and \( P, w, R, s \) in Algorithm \ref{alg:13}. Moreover, when a recursive call is made in the algorithms the conditions written in the code imply that we can easily (and cheaply) compute the state variables using only information about the child. It is thus easy to modify these two algorithms to simulate the recursion avoiding an explicit stack. The detailed structure is shown in Algorithm \ref{alg:14}.

With regard to the restricted problem, note that we can iterate over the solutions of the restricted problem using \( \mathcal{F}_T \) time and \( \mathcal{G}_S \) space: we can restart the iteration whenever we backtrack in the (simulated) recursion tree, as this does not impact the delay. These observations allow us to state the following.

\(^{10}\) A naive implementation of \text{parent} takes \( q \) runs of \text{complete}, but it can actually be computed by a single run of \text{complete} in which at every step we check if the next vertex to be added would be the next vertex in \( S \) or not.
Algorithm 14: Stateless framework with minimal memory

```
Function improved-spawn (X)
    P ← X
    S ← null
    w ← next-node(null)
    R ← next-r(P, w, null)
    do
        do
            if S ← next-child(P, w, R, S) ≠ null then
                ⟨P, S, w, R⟩ ← ⟨S, null, null, null⟩ /* recur in child */
                break
        while R ← next-r(P, w, R) ≠ null
        while w ← next-node(w) ≠ null
        if is-root (P) then return
        else ⟨P, S, w, R⟩ ← ⟨parent(P), P, pi(P), r(P)⟩ /* backtrack */
    while true

Function is-root (X)
    return pi(X) = source(X)

Function next-node(w)
    return min {v ∈ U: v > w}

Function next-r(P, w, R)
    return the solution succeeding R in the restricted problem P ∪ {w} (or null if R is the last)

Function next-child(P, w, R, S)
    foreach x ∈ R : x > source(S) do
        C ← complete({x}, R)w
        D ← complete(C ∪ {w}, U)
        if ⟨parent(D), pi(D), r(D), source(D)⟩ = ⟨P, w, R, x⟩ then
            return D
    return null
```

Theorem 5.4. The stateless versions of Algorithm 12 and Algorithm 13 achieve the same delays while taking, respectively, \(O(G_S)\) and \(O(G_S + R_S)\) space.

Note that in the case of \(bc\)-cliques, \(G_T, G_S, R_T\) and \(R_S\) are \(O(q)\). Hence, as an application of Theorems 5.3 and 5.4, we obtain the following.

Theorem 5.5. There exists an algorithm to list all the maximal \(bc\)-cliques with delay \(O(q^5|U|^2)\) and \(O(q)\) space.

Corollary 5.1. For a commutable set system, the solutions of the general problem can be enumerated in polynomial total time and polynomial space if and only if the solutions of the restricted problem can be enumerated in polynomial total time and polynomial space. Moreover, the solutions of the general problem can be enumerated in polynomial delay and polynomial space if the solutions of the restricted problem can be enumerated in polynomial time and polynomial space.
5.6 Final remarks

In this chapter we described a space-efficient framework to design algorithms for enumerating the maximal solutions in strongly accessible set systems, and further improved its running time for connected hereditary properties. This result provides a new upper bound for the enumeration of maximal solution in strongly accessible set systems of $O(\alpha^{2^q})$ time which is, to the best of our knowledge, the first non-trivial bound for this class of problems, and almost matching the lower bound of $O(\alpha^{2^{q/2}})$ time given by Lawler [LLRK80]. Furthermore, we solve the open problem left by Cohen et al. [CKS08] of enumerating connected hereditary properties with bounded delay using polynomial space.

To give a more complete picture, we should remark that the improved version of our framework is not limited to just connected hereditary graph properties: we call the class of set systems to which it can be applied commutable set systems, which corresponds to all set systems that are strongly accessible and respect the commutable property.

It is easy to show that the class of commutable set systems includes more than connected hereditary properties: for example, required hereditary and required connected hereditary properties (defined in Section 5.1) clearly verify the commutable property, thus both version of our framework can be applied to them.

Future work is aimed at applying this framework to a variety of problems, with some already promising preliminary results. A problem that remains open is applying reverse search to a wider context of problems, e.g., without relying on the commutable property or strong accessibility.
Finding communities in the form of cohesive subgraphs is a fundamental problem in network analysis. In domains that model networks as undirected graphs, communities are generally associated with dense subgraphs, and many community models have been proposed. Maximal cliques are arguably the most widely studied among such models, with early works dating back to the ’60s, and a continuous stream of research up to the present. In domains that model networks as directed graphs, several approaches for community detection have been proposed, but there seems to be no clear model of cohesive subgraph, i.e., of what a community should look like. We extend the fundamental model of clique to directed graphs, adding the natural constraint of strong connectivity within the clique. We characterize the problem by giving a tight bound for the number of such cliques in a graph, and highlighting useful structural properties. We then exploit these properties to produce the first P-Delay algorithm for enumerating maximal strongly connected cliques.

Finally, we show how the techniques presented in Chapter 5 can be applied to reduce the space complexity of the algorithm.

6.1 Introduction

The problem of community detection in graphs has been extensively studied. In undirected graphs, dense subgraphs are often used to detect communities, with applications in areas such as social network analysis [Sco12, WF94], biology [JS11], and more [For10].

Several definitions of dense subgraph have been proposed to model communities [PYB12, WF94]. The earliest, and perhaps the most widely studied is that of the maximal clique: interest in the problem of finding maximal cliques started several decades ago [Akk73, BK73, MM65], and effort to produce efficient algorithms can still be seen in recent works [CR15, CGMV16, ELS13].

As for directed graphs, there seems to be consensus in literature [LLN10, LN08, RB08] on the fact that ignoring edge directions and applying community detection techniques for undirected graphs is not satisfactory. Several ad-hoc techniques for clustering and community discovery have been proposed, mirroring the goals of algorithms for undirected graphs.

Awerbuch et al. [AS01] proposed a bounded-error scheme for aggregating vertices in directed graphs as a hierarchical structure. Leicht et al. [LN08] adapted the concept of modularity to account for edge directions, with the aim of extracting more meaningful clusters. The LinkRank algorithm [LLN10] aimed at partitioning a directed graph into communities using random walks and the PageRank algorithm. More approaches can be found in [For10].
Subgraph-based community models in undirected networks are thoroughly studied in community detection (and network analysis in general), thus it would be natural to imagine that similar models were object of study in the directed area. Surprisingly, this seems to be a road less traveled\footnote{It should be mentioned that strongly connected components have been object of thorough study, however these may be very large, sparse, and thus may not be significant indicators of community structures.} Charikar et al. \cite{Charikar2000} considered communities in directed graphs as sets of vertices whose induced subgraphs have many edges, regardless of connectivity. The well-known work by Kleinberg et al. \cite{Kleinberg1999} defined a community in the web graph with respect to a topic as a special bipartite clique $K_{ij}$, in which each of the $i$ vertices has edges towards each of the other $j$ vertices, which represent authority pages on the topic. To the best of our knowledge, there are no other community models for directed graphs that are widely accepted and rigorous. This motivates our interest in combining the basic maximal clique model with connectivity in directed graphs, that is strong connectivity. We call this model a strongly connected clique (SCQ for short), and investigate both its properties and the problem of efficiently finding all maximal SCQs.

Generic enumeration techniques for maximal subgraphs have been proposed for strongly accessible properties \cite{Abdullah2009}, i.e., such that every non-maximal subgraph $A$ which verifies the property is included in a subgraph $B$ of size exactly $|A| + 1$ that also verifies the property. Cohen et al. \cite{Cohen2008} proposed an algorithmic framework for enumerating maximal subgraphs with respect to subsets of strongly accessible properties, namely hereditary and connected-hereditary graph properties. SCQs, however, fit in neither of these classes.

Finding maximal subgraphs satisfying a non accessible property is a challenging task, as their structure is unsystematic, and their enumeration requires new techniques and theoretical insight. In this chapter, we show that SCQs have a peculiar but rigorous structure, which fits under a relaxed, more general notion of accessibility. We then exploit this structure to design SCQ-ENUM, an efficient algorithm that enumerates SCQs with delay bounded by $O(\min(\omega(G)d^2\Delta^2, m^2))$, where $\omega(G)$, $d$, $\Delta$ and $m$ are respectively the largest size of an SCQ, degeneracy, maximum degree and number of edges of the input graph, and the delay is the maximum time elapsed between two consecutive outputs. The value of SCQ-ENUM is two-fold: on one hand, it constitutes a first step towards the characterization, and potentially towards general enumeration techniques, for a wider range of problems that are not accessible. On the other hand, SCQ-ENUM is also an efficient practical tool for discovering community structures in directed networks. Finally, we complete the analysis of the model by giving a tight bound for the number of maximal SCQs in an $n$-vertex graph.

### 6.2 Preliminaries

Refer to Section 2.1 for terminology. The graphs considered in this chapter are directed and simple (do not contain multiple edge and self loops but may contain twin edges of opposite direction). When $E(G)$ is symmetric, i.e., $(x,y) \in E(G)$ if and only if $(y,x) \in E(G)$, we call $G$ undirected, as its connectivity is equivalent to an undirected graph, and denote each edge $(x,y)$ of $G$ by $xy$ (equivalently $yx$). For a graph $G$, we denote by $u(G)$, called underlying (undirected) graph of $G$, the undirected graph with vertex set $V(G)$ and edge set $\{xy \mid (x,y) \in E(G)\}$. We recall that a directed graph is strongly connected if, for any pair of vertices $x$ and $y$ there is a directed path from $x$ to $y$ and from $y$ to $x$.

Let $G = (V, E)$ be a (directed) graph. A strongly connected clique (or SCQ for short) is a set $C \subseteq V(G)$ such that $G[C]$ is strongly connected, and $u(G)[C]$ is a clique (set of pairwise adjacent vertices) of $u(G)$. We note that a directed graph (or subgraph) is strongly connected if and only if for each bipartition $(V_1, V_2)$ of $V$ there is an edge from $V_2$ to $V_1$, and symmetrically from $V_1$ to $V_2$ (a formal proof is given in Section 7.10). We assume that a single vertex is an SCQ; we
We recall that a \( \text{scq} \) in \( G \) by \( \omega(G) \), and the maximum size of a clique in \( u(G) \) by \( \omega(u(G)) \). It is worth noticing that if \( G \) is undirected, \( \text{scqs} \) and cliques coincide. An \( \text{scq} \) \( C \) is maximal if there is no \( \text{scq} \) \( C' \) such that \( C \subset C' \). Given \( C \subseteq V \), the set \( X \subseteq V \setminus C \) is \emph{addible} to \( C \) if \( C \cup X \) is an \( \text{scq} \), and \( Y \subseteq C \) is \emph{removable} from \( C \) if \( C \setminus Y \) is an \( \text{scq} \). Furthermore, we say that a vertex \( x \) is a \emph{sink w.r.t.} \( C \) if there is no \( (x, y) \in E(G) \) with \( y \in C \), and a \emph{source w.r.t.} \( C \) if there is no \( (y, x) \in E(G) \) with \( y \in C \). A graph with its maximal \( \text{scqs} \) and cliques is shown in Figure 6.1.

We assume that any graph is given with a degeneracy ordering\(^2\).

### 6.3 Problem characterization

Maximal \( \text{scqs} \) are a challenging problem as they do not satisfy the \emph{strong accessibility} property. However, we provide some related properties that will be the key of our enumeration algorithm.

#### 6.3.1 Relaxed accessibility of strongly connected cliques

We recall that a \emph{set system} \((V, E \subseteq 2^V)\) is \emph{(weakly) accessible} if for each \( X \in E \), there is \( x \in X \) such that \( X \setminus \{x\} \in E \), and it is \emph{strongly accessible} if in addition for each \( X, Y \in E \) with \( Y \subset X \), there is \( x \in X \setminus Y \) such that \( Y \cup \{x\} \in E \). In both cases it is assumed that \( \emptyset \in E \). The following two lemmas prove a relaxed notion of weak and strong accessibility.

**Lemma 6.1.** Let \( C \) be a non-empty \( \text{scq} \) of \( G \). There exists \( Z \subseteq C \) removable from \( C \) and such that \(|Z| \leq 2 \).

**Proof.** As a single vertex and the empty set are \( \text{scqs} \), if \(|C| = 3\) any \( Z \subset C \) with \(|Z| = 2 \) is removable, and if \(|C| = 1\) or \(|C| = 2\) any vertex in \( C \) is removable. Suppose then that \(|C| \geq 4\) and let us prove that it has a removable vertex.

Let \( y \) be an arbitrary vertex of \( C \) and suppose that \( C' = C \setminus \{y\} \) is not strongly connected. Then, there exists a bipartition \((X, Y)\) of \( C' \) such that \( E(G[C]) \cap (Y \times X) = \emptyset \). As \( C = C' \cup \{y\} \) is strongly connected, we must have \( w \in X \) and \( w' \in Y \) such that \((y, w), (w', y) \in E(G[C])\), and \( y \) can reach every vertex in \( X \), and also every vertex from \( Y \) can reach \( y \). Since, \(|C| \geq 4\) and thus \(|C'| \geq 3\), either \(|X| \geq 2\) or \(|Y| \geq 2\). Assume \(|X| \geq 2\): let \( z \in X \) be a leaf of a traversal of \( X \cup \{y\} \) starting from \( y \) (recall that \( y \) can reach all vertices in \( X \)). As \( z \) is a leaf, if we remove it, \( y \) can still reach all vertices in \( X \setminus \{z\} \). Furthermore, each vertex in \( X \setminus \{z\} \) has an edge towards every vertex in \( Y \), as \( C \) is an \( \text{scq} \), and every vertex in \( Y \) can reach \( y \). Thus \( \{y\} \cup (X \setminus \{z\}) \cup Y = C \setminus \{z\} \) is an \( \text{scq} \), \( i.e., \), \( Z = \{z\} \) is a removable set in \( C \) with \(|Z| \leq 2\). If \(|X| = 1\), then \(|Y| \geq 2\), and the proof is symmetrical by choosing \( z \) as a leaf vertex in a traversal of \( G[Y \cup \{y\}] \) with the edges reversed, starting from \( y \).

**Lemma 6.2.** Given two \( \text{scqs} \) \( C \) and \( D \) such that \( D \subset C \), there exists \( X \subseteq C \setminus D \) addible to \( D \) with \(|X| \leq 2\).

\(^2\)See Section \ref{deg} for how to compute a degeneracy ordering in linear time.
Proof. If \(|C \setminus D| \leq 2\) the lemma is trivially true, so assume \(|C \setminus D| \geq 3\). Any vertex in \(C \setminus D\) with edges both towards and from vertices in \(D\) is addible to \(D\), so assume that no such vertex exists. Hence, all the vertices in \(C \setminus D\) are either sinks or sources w.r.t. \(D\), that we denote by \(K\) and \(R\), respectively. Any set \(\{k, r\}\) with \(k \in K\), \(r \in R\) and \((k, r) \in E(G[C])\) is an addible set to \(D\). Assume then that no such set does exist: all the vertices in \(K\) cannot reach \(R\) or \(D\), and all the vertices in \(R\) cannot be reached from \(K\) or \(D\). This is a contradiction as \(C = D \cup K \cup R\) is strongly connected, thus an addible set \(\{k, r\}\) exists.

And as any non-maximal SCQ is contained in a larger one, we obtain the following.

**Corollary 6.1.** An SCQ \(C\) is maximal in \(G\) if and only if there is no \(X \subseteq V(G) \setminus C\) addible to \(C\) with \(|X| \leq 2\).

Thanks to Lemmas 6.1 and 6.2, we can say that the property of being an SCQ belongs to a relaxed class of accessibility, since (i) for each \(X \in \mathcal{E}\), there is \(Z \subseteq X\) such that \(X \setminus Z \in \mathcal{E}\), and (ii) for each \(X, Y \in \mathcal{E}\) with \(Y \subseteq X\) there is \(Z \in X \setminus Y\) such that \(Y \cup Z \in \mathcal{E}\), where the size of \(Z\) is at most 2. This is a generalization of the definitions of strong and weakly accessible classes, which are obtained from the above by simply setting \(|Z| = 1\).

### 6.3.2 Maximal undirected and strongly connected cliques

On top of the accessibility of the problem, we are interested in studying the relationship between the SCQs in \(G\) and the cliques in the underlying undirected graph \(u(G)\). Lemma 6.3 highlights the first basic, but important, relationship.

**Lemma 6.3.** Given a directed (not necessarily strongly connected) clique \(D\), the strongly connected components of \(D\) are the maximal SCQS in \(D\).

**Proof.** Any SCQ \(C\) is contained in a strongly connected component of a directed clique by definition, as \(C\) is strongly connected and \(u(G[C])\) is a clique. Furthermore, an SCQ may not contain vertices from different strongly connected components, as it would not be strongly connected. Thus, a strongly connected component of \(D\) is a maximal SCQ in \(D\).

**Corollary 6.2.** A directed clique \(D\) contains at most \(|D|\) maximal SCQs, which are disjoint.

### 6.3.3 Bounding the number of maximal SCQs

For a graph \(G\), let us denote by \(gc(G)\) and \(gc(u(G))\) the number of maximal SCQs in \(G\) and maximal cliques in \(u(G)\) respectively, and for \(n\), let us denote by \(g(n)\) the maximum number of maximal SCQs in an \(n\)-vertex graph. From Corollary 6.2 any maximal SCQ in a directed graph \(G\) is contained in a maximal clique of \(u(G)\), i.e., \(gc(G) \leq \omega(u(G)) \cdot gc(u(G))\). But the number of SCQs in a graph can be much smaller than the number of cliques of its underlying undirected graph: For instance, an \(n\)-vertex DAG has exactly \(n\) maximal SCQs of size 1 while the number of maximal cliques of its underlying undirected graph can be arbitrarily large. As the maximum number of maximal cliques in an undirected \(n\)-vertex graph is \(3^n/2\) [MM65], we can immediately conclude that \(3^n/2 \leq g(n) \leq n \times 3^n/2\). We can adapt the proof from [MM65] to show that \(g(n)\) is indeed \(3^n/2\).

Let \(G\) be an \(n\)-vertex graph, and \(x, y\) two vertices of \(G\), and let \(G(x; y)\) be defined similarly to [MM65], as the graph obtained by removing all edges incident to \(x\), and replacing them so that the neighborhood of \(x\) is identical to that of \(y\), i.e., \((x, v) \in E(G(x; y))\) iff \((y, v) \in E(G)\) and \((v, x) \in E(G(x; y))\) iff \((v, y) \in E(G)\). Let \(\chi(x)\) be the number of maximal SCQs containing \(x\), let \(\alpha(x)\) be the number of new maximal SCQs created by removing \(x\) (i.e., subsets of SCQs
containing \( x \) which become now maximal), and \( \beta(x) \) the number of SCQs which are not maximal anymore after removing \( x \). It is straightforward to see that if \( x \) and \( y \) are not adjacent, the number of SCQs in \( G(x; y) \) is given by \( gc(G) + \chi(y) - \chi(x) + \alpha(x) \). Indeed all SCQs containing \( x \) have been removed, and replaced by \( \alpha(x) \) new maximal cliques; furthermore, for each of the \( \chi(y) \) maximal SCQs containing \( y \) in \( G \), we now have a new one containing \( x \) instead of \( y \). If \( x \) and \( y \) are adjacent, any maximal SCQ containing \( y \) in \( G(x; y) \) will be simply incremented with \( x \); as a result, the number of maximal SCQs in \( G(x; y) \) will be only \( gc(G) - \chi(x) + \alpha(x) \). We are now ready to characterize the graph with the highest number of maximal SCQs.

**Lemma 6.4.** Let \( G \) be a graph on \( n > 4 \) vertices with \( gc(G) = g(n) \). There exists \( G^* \), an \( n \)-vertex graph such that \( gc(G^*) = g(n) \) and \( u(G^*) \) is a complete multipartite graph.

**Proof.** If \( u(G) \) is a clique, it has at most \( n \) maximal SCQs by Lemma 6.3, so we can replace \( G \) with any DAG and still have \( n \) maximal SCQs. Thus we can assume that \( u(G) \) is not a clique, and has at least 2 non-universal vertices (i.e., not connected to every other vertex).

For two non-adjacent vertices \( x \) and \( y \), we know that \( G(x; y) \) and \( G(y; x) \) cannot have more SCQs than \( G \) by definition of \( G \). As \( \alpha(x) \geq 0 \), we have \( \chi(x) = \chi(y) \) for any pair of non-adjacent vertices. This implies \( \alpha(x) = 0 \) for every non-universal vertex \( x \). Thus, if \( x \) and \( y \) are non-adjacent, \( gc(G(x; y)) = gc(G) = g(n) \). From \( G \), we can obtain the graph \( G^* \) as follows: for each vertex \( x \), and for each vertex \( y \) not adjacent to \( x \), iteratively replace \( G \) with \( G(y; x) \).

Observe from the discussion above that \( gc(G^*) = gc(G) = g(n) \). Also, \( u(G^*) \) is a complete multipartite graph. Indeed, as each pair of non-adjacent vertices has the same neighbors, we can partition the graph into independent sets such that two vertices in two different independent sets are adjacent. Again, if \( u(G^*) \) is a clique (this may be the case for \( n = 4 \)), we replace \( G^* \) with any DAG without compromising the number of maximal SCQs and thus obtaining at least 2 non-universal vertices.

Thanks to Lemma 6.4, we can link the number of maximal SCQs in \( G^* \) to the number of maximal cliques in \( u(G^*) \). This relation will enable us to give a tight bound for \( g(n) \).

**Lemma 6.5.** Let \( G = (V, E) \) be a graph such that \( u(G) \) is a complete multipartite graph. Then \( gc(G) \leq gc(u(G)) \).

**Proof.** Let \( S = \{S_1, \ldots, S_k\} \) be the set of maximal independent sets of \( u(G) \), which forms a \( k \)-partition of \( V \). Let \( s(v) \) be the size of the unique maximal independent set \( S_i \) containing \( v \); as \( G \) and \( u(G) \) have no universal vertices, \( s(v) \geq 2 \). By definition each maximal SCQ in \( G \) is a subset of some maximal clique of \( u(G) \), and recall that each maximal clique of \( u(G) \) (a complete multipartite graph) is obtained by selecting exactly one vertex from each of its maximal independent sets.

Let the occurrence \( mc(C) \) of an SCQ \( C \) be the number of maximal cliques of \( u(G) \) that contain \( C \), and let the weight \( w(C) \) of \( C \) be \( \frac{1}{mc(C)} \), or 0 if \( C = \emptyset \). For a maximal clique \( Q \) of \( u(G) \), let the weight \( w(Q) \) of \( Q \) be instead \( \sum_X w(X) \) where \( X \) is a maximal SCQ of \( G(Q) \). The sum of the weights of all maximal cliques in \( u(G) \) will be at least equal to the number of maximal SCQs in \( G \): any maximal SCQ \( C \) will be considered \( mc(C) \) times, each time adding \( w(C) = 1/mc(C) \), for a total contribution of 1. This sum may be larger, as it can include subsets of maximal cliques which are not maximal SCQs in \( G \), but cannot be smaller.

Let \( C \) be a maximal SCQ and let \( T \subseteq S \) be the maximal independent sets that do not contain any vertex of \( C \). Then, the maximal cliques that contain \( C \) are all the ones obtained by adding a

\[ \text{Note that in the undirected case } \alpha(x) \text{ is bounded by } \chi(x), \text{ but in the directed case } \alpha(x) \text{ may be larger than } \chi(x) \text{ by up to an } n \text{ factor.} \]

101
single vertex from each independent set in $T$, thus $mc(C) = \prod_{S_i \in T} |S_i|$. This means that adding a vertex $v$ to $C$ reduces $mc(C)$ and increases $w(C)$ by a factor $s(v)$, while removing $v$ from $C$ has the opposite effect.

Let us now consider the highest possible weight of a maximal clique $Q$ in $u(G)$. Note that, by Corollary 6.2, the maximal SCQs within $Q$ are at most $|Q|$ and do not overlap. If $Q$ contains a single maximal SCQ $X$, we have $|X| = |Q|$, $mc(X) = 1$ and $w(X) = 1$, thus $w(Q) = 1$. Otherwise, let $X$ and $Y$ be two maximal SCQs in $Q$, $X$ being the one with highest weight. Note that $w(X) + w(Y) \leq 2w(X)$. Assume that we could remove a vertex $v$ from $Y$ and add it to $X$, obtaining $X'$ and $Y'$: if $s(v) = 1$, then $w(X') + w(Y') = w(X) + w(Y)$; otherwise, we have $w(X') = s(v) \cdot w(X)$, and as $s(v) \geq 2$, $w(X') + w(Y') = w(X) \cdot s(v) + w(Y') \geq 2w(X) \geq w(X) + w(Y)$. This hypothetical operation can increase the total weight of $Q$ but do not decrease it, i.e., for any distribution of maximal SCQs in $Q$, a different one in which the size of the largest SCQ is increased by 1, and the size of another one is reduced by 1, has greater or equal weight. We can repeat this hypothetical step, iteratively enlarging $X$ until we will finally consider a distribution with a single maximal SCQ $X$ of size $|X| = |Q|$. As $w(Q)$ in this case is at least as large as that obtained with any other distribution of maximal SCQs in $Q$, and as shown above $w(Q) = 1$ in this case, we have $w(Q)$ is always at most 1. Therefore, the number of maximal SCQs in $G$, i.e., the sum of all weights of the maximal cliques in $u(G)$ cannot be larger than $gc(u(G))$. □

It is known that the undirected graph with the highest number of maximal cliques is the Moon-Moser graph [MM65], which is a complete multipartite graph in which as many maximal independent sets as possible have size 3, while the remaining ones may only have size 2.\(^4\) Clearly, such a graph does not have universal vertices, and thus is compatible with the definition of $G^\ast$. We can thus say that the underlying graph $u(G)$ of the graph $G^\ast$ with the highest number of maximal SCQs will be a Moon-Moser graph. Furthermore, by Lemma 6.5 we get the upper bound $g(n) = gc(G^\ast) \leq gc(u(G^\ast))$.

It is now easy to prove that this is a tight bound: when $G$ is symmetric (i.e., $(x, y) \in E(G)$ iff $(y, x) \in E(G)$) then the connectivity in $G$ is the same as in $u(G)$ and each maximal clique in $u(G)$ will be a maximal SCQ in $G$. Thus we have $g(n) = gc(G^\ast) = gc(u(G^\ast))$. By combining this lower bound with Lemma 6.5 and the Moon-Moser bound [MM65], we can conclude the following (the case $2 \leq n \leq 4$, not covered by Lemma 6.4 is omitted for space reasons, but can be trivially verified).

**Theorem 6.1.** For every integer $n > 1$,

$$g(n) = \begin{cases} \frac{3n}{2} & \text{if } n \equiv 0 \pmod{3}, \\ \frac{4}{3} \cdot 3^{\lfloor \frac{n}{3} \rfloor} & \text{if } n \equiv 1 \pmod{3} \\ 2 \cdot 3^{\lfloor \frac{n}{2} \rfloor} & \text{if } n \equiv 2 \pmod{3}. \end{cases}$$ □

Finally, the same result can be proven for oriented graphs, that are directed graphs where each edge may only have one direction, as long as $n$ is not 5 or 6 (this is omitted for space reasons but also involves suitable orientations of Moon-Moser graphs).

### 6.4 Listing maximal SCQs

While the number of maximal SCQs in a graph $G$ is at most $n$ times the number of maximal cliques of its underlying (undirected) graph $u(G)$, and each maximal SCQ of $G$ is contained in

\(^4\)Equivalently, the remaining ones may also have size 4. However it is not necessary to consider this case.
some maximal clique of $u(G)$, one cannot efficiently use output-polynomial algorithms for listing maximal cliques in undirected graphs in order to list the maximal scqs of a graph. For example, any orientation of a Moon-Moser $n$-vertex graph into a DAG has exactly $n$ maximal scqs, while its underlying graph has $\Theta(3^{\frac{n}{3}})$ maximal cliques. This strategy would hence take exponential running time to find just a linear number of maximal scqs.

In this section we design an algorithm that enumerates all maximal scqs of a graph $G = (V, E)$ with polynomial delay.

Intuitively, given a maximal scq (called sometimes a solution) $S$, our algorithm uses the vertices in $V \setminus S$ to find other solutions similar to $S$; we refer to this process as visiting $S$. By visiting these newly found solutions the algorithm eventually finds all solutions in $G$.

6.4.1 Algorithm description

The algorithm, which we call scq-enum, is described in Algorithm 15. SCQ-ENUM uses a result set, which will store all solutions found so far. The primitive \texttt{contains}(S,\texttt{result}) is a subroutine that returns true if $S \in \texttt{result}$, i.e., $S$ has already been found and does not need to be visited again, and the primitive \texttt{add}(S,\texttt{result}) adds $S$ to the result set. Finally, SCQ-ENUM exploits the function \texttt{complete}(X, A), which will iteratively add the lexicographically minimum addible vertex or pair of vertices from $A$ to a scq $X$, until $X$ is maximal w.r.t. $A$, and return it. For brevity, \texttt{complete}(X) represents \texttt{complete}(X,V).

Thanks to the accessibility proven in Lemma 6.2 and Corollary 6.1, \texttt{complete}(X) will surely return a maximal scq. We recall that we assume the graph given with the degeneracy ordering, and we consider that ordering and its associated lexicographic one in the algorithm (see Section 6.2). The primitive \texttt{minlex}(T) finds the minimum in the collection $T \subseteq 2^V$.

**Algorithm 15: scq-enum**

- **Input**: A graph $G=(V,E)$
- **Output**: The set \texttt{result} containing all maximal scqs in $G$
- **Global**: \texttt{result} set, initially empty

```
1  for $v \in V$ do
2      \texttt{enum(complete}($\{v\}$))
3  
4  Function \texttt{enum}(S)
5      if \texttt{contains}(S,\texttt{result}) then return
6      \texttt{add}(S,\texttt{result})
7      foreach $x \in V \setminus S$ do
8          $I \leftarrow (S \cap N(x) \cup \{x\})$
9          $X \leftarrow \texttt{complete}($\{x\}$,I)$
10         if $X = \{x\}$ then continue
11         \texttt{enum}($\texttt{complete}(X)$)
12      endforeach
13      foreach $\{y,z\} \subseteq V \setminus S$ do
14          $I \leftarrow (S \cap N(y) \cap N(z) \cup \{y,z\})$
15          $X \leftarrow \texttt{complete}($\{y,z\}$,I)$
16         if $X = \{y,z\}$ then continue
17         \texttt{enum}($\texttt{complete}(X)$)
18  
```

SCQ-ENUM is in the same spirit as the one for listing the maximal cliques in an undirected graph [CGMV16]. It does a DFS traversal of the graph of solutions where $(S,S')$ is an edge if $S'$ can be obtained from $S$ by adding a new vertex (or a pair of vertices), removing its (their)
We will hereby prove the correctness of scq-enum. The principle of finding maximal solutions from other solutions is used by many enumeration algorithms, but this has so far been applied exclusively to properties with strong accessibility [AU09], such as hereditary [CGMV16, Fuk96, TIAS77], or connected-hereditary [ECK15, CKS08].

Thanks to the results obtained in Section 6.3, however, we will be able to prove the correctness of our technique, despite scqs not being strongly (or even weakly) accessible, similar to [CKS08]. In the following, given two scqs S and T, let S ∩seq T be the largest scq in S ∩ T; we recall that this may be a single vertex, which is indeed an scq.

Proving that no solution is found twice by scq-enum is trivial, as duplication is removed by the result set, and since every output is a maximal scq since it is the result of a complete call, we only need to prove that every maximal scq is found:

**Theorem 6.2.** scq-enum finds all and only maximal scqs exactly once.

**Proof.** Let T be any solution not yet found by the algorithm. Let S be the solution found by scq-enum which maximizes |S ∩seq T|. Note that |S ∩seq T| ≥ 1; for any v ∈ T, the algorithm will visit C = complete(\{v\}), a maximal scq containing v, so |C ∩seq T| ≥ 1. Now let Z = S ∩seq T. We have Z ≠ T, otherwise T would not be maximal, and by Lemma 6.2 there exists Y ⊆ T \ Z with 1 ≤ |Y| ≤ 2 s.t. Z ∪ Y is an scq. Note that Y is not contained in S, as otherwise Z ∪ Y would be a larger scq in S ∩seq T. Three cases are possible: (i) Y = \{x\}, then x ∈ V \ S and x is considered in the first for loop. (ii) Y = \{y, z\} ⊆ V \ S, then \{y, z\} is considered in the second

---

**Algorithm 16: complete(X, A)**

<table>
<thead>
<tr>
<th>Function complete(X, A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input ( \hat{X} ), an scq, and A ⊆ V, a set of vertices</td>
</tr>
<tr>
<td>Output X' ⊇ X, an scq maximal with respect to A</td>
</tr>
<tr>
<td>1 while EXT ← min-extension(X, A) ≠ null do</td>
</tr>
<tr>
<td>2 ( X \leftarrow X \cup \text{EXT}; )</td>
</tr>
<tr>
<td>3 return X;</td>
</tr>
<tr>
<td>5 Function min-extension(X, A)</td>
</tr>
<tr>
<td>6 ADD ← ( {Y \subseteq A \mid X : 1 \leq</td>
</tr>
<tr>
<td>7 return minlex(ADD)</td>
</tr>
</tbody>
</table>

---

non-neighbors and finally completing the obtained set into a maximal scq. Let us describe the algorithm.

SCQ-ENUM consists in calling the function enum(S), with S a maximal scq. In turn, enum(S) will find all solutions that have a non-empty overlap with S. The function will consider all vertices \( x \in V \setminus S \), and for each of them will try to generate a new maximal scq containing \( x \) and some vertices of S: by calling \( X \leftarrow \text{complete(\{x\}, I)} \) the algorithm will get the scq containing \( x \), maximal w.r.t. the induced subgraph \( G[S \cup \{x\}] \); note that there is only one such scq, as \( G[S \cup \{x\}] \) is a clique in \( G[S \cup \{x\}] \) (see Lemma 6.3). Then \( X \) is extended with complete(X) so that it is maximal w.r.t. \( G \), i.e., a solution. Then, in the second for loop, the same process is repeated for pairs of vertices rather than single vertices. Every time a solution \( S' \) is found, we recur in enum(\( S' \)), which will visit \( S' \), adding it to the result set and finding more solutions starting from \( S' \). If \( S' \) is already in the result set, however, it means it was already visited and all the relative solutions have been found, thus we can ignore it and backtrack.
for loop. (iii) \(|Y| = \{y, z\}\), with \(y \in V \setminus S\) and \(z \in S\), then \(y\) is considered in the first for loop and we will have \(z \in S \cap N(y) \cup \{y\}\).

In all these cases, the SCQ \(X\) (maximal in \(I\)) that is found, will contain \(Z \cup Y\) by Lemma 6.3. When we execute COMPLETE\((X)\), we will either find \(T\), or a maximal SCQ \(S'\) that contains \(Z \cup Y\). As \(|Z \cup Y| > |Z|\), we have \(|S' \cap \text{scq} \cap T| > |S \cap \text{scq} \cap T|\). By induction, when visiting \(S'\) we will either find \(T\), or \(S''\) such that \(|S'' \cap \text{scq} \cap T| > |S' \cap \text{scq} \cap T|\), until eventually, in at most \(|T|\) such steps, \(\text{SCQ-ENUM}\) will find \(T\). □

6.5 Complexity analysis

We now analyze the complexity of \(\text{SCQ-ENUM}\), i.e., Algorithm 15, showing that it lists maximal SCQs with delay \(O(\min(\omega(G)d^2\Delta^2,m^2))\). Recall that \(m, n, \Delta, d\) and \(\omega(G)\) are respectively the number of edges, number of vertices, maximum vertex degree, degeneracy ordering and maximum size of an SCQ in \(G\), and that the vertices \(v_1, \ldots, v_n\) are given in a degeneracy ordering. Firstly, we bound the complexity of the function COMPLETE:

**Lemma 6.6.** COMPLETE\((X, A)\) (Algorithm 16) can be executed in time \(O(\min(d\Delta, m))\).

**Proof.** Consider the vertices in \(A\) adjacent to all vertices of \(X\), i.e., \(\bigcap_{x \in X} (N(x)) \cap A\), and partition them into three sets, each stored in increasing lexicographical order: SINK contains all the sinks w.r.t. \(X\); SOURCE the sources w.r.t. \(X\), and BOTH all vertices that have at least one edge from and one towards some vertex in \(X\) (i.e., neither sinks nor sources w.r.t. to \(X\)). The computing time is the sum of the degrees of vertices in \(X\), i.e., \(\min(\omega(G)\Delta, m)\) and the total size \(|\text{SINK}| + |\text{SOURCE}| + |\text{BOTH}|\) of the sets is bounded by \(\Delta\).

As each step adds either one or two vertices to \(X\), and \(|X|\) is bounded by \(\omega(G)\), we will have at most \(\omega(G)\) steps. Whenever we add a vertex \(a\) to \(X\), we can update the SINK, SOURCE, BOTH sets by only looking at \(N(a)\): any non neighbor of \(a\) is excluded from these sets, any vertex in SINK with an edge towards \(a\), or vertex in SOURCE with an edge from \(a\) is moved in BOTH. This takes \(O(|N(a)|)\) time, thus \(O(\min(\omega(G)\Delta, m))\) time for all updates.

If the BOTH set is not empty, we can find the (lexicographically) smallest \(x\) in \(O(1)\) time. Then, we need to find the smallest pair \(a \in \text{SINK}, b \in \text{SOURCE}\) s.t. there is an edge from \(a\) to \(b\), if it exists. We do this by scanning vertices in \(\text{SINK} \cup \text{SOURCE}\) in order, and for each scanning its forward neighbors, still in order; we stop at the first pair that verifies the property. We never need to consider the same pair twice, as the condition (edge from \(a\) to \(b\)) will stay false (although the vertices might be later moved to BOTH and still enter \(X\), thus the total cost will be \(O(\min(d\Delta, m))\) for all steps. Finally, the smallest among \(x\) and \(\{a, b\}\) corresponds to MIN-EXTENSION\((X, A)\); we add it to \(X\) and update the SINK, SOURCE, BOTH sets. The total cost is given by \(O(\min(\omega(G)\Delta, m) + \min(d\Delta, m)) = O(\min(d\Delta, m))\)

Finally, we are ready to give the time complexity of \(\text{SCQ-ENUM}\):

**Lemma 6.7.** \(\text{SCQ-ENUM}\) (Algorithm 13) has \(O(\min(\omega(G)d^2\Delta^2,m^2))\) time delay.

**Proof.** Let us first focus on the amortized cost per solution, i.e., the cost of an execution of ENUM\((S)\) without considering children recursive calls (which lead to other solutions). To compute contains\((S, \text{RESULT})\) and add\((S, \text{RESULT})\) we store RESULT as a trie, whose depth will be \(O(\omega(G))\), and degree will be bounded by \(\Delta\) as SCQs are made of adjacent vertices.\(^5\) Checking the existence and adding a solution to this trie takes time \(O(\min(\omega(G)\log(\Delta), m))\).

In the for loops we only need to consider \(x\) and \(\{y, z\}\) s.t. \(I \neq \{x\}\) and \(I \neq \{y, z\}\), thus only vertices with a neighbor in \(S\): for \(x\) we have \(|S|\Delta \leq \min(\omega(G)\Delta, n)\) choices; as for \(\{y, z\}\)

\(^5\)The root of the trie has degree up to \(n\); we store this as a vector of size \(n\), accessible in \(O(1)\) time.
we have \min(\omega(G)\Delta, n) choices for \(y\), and for each \(y\) up to \(d\) choices for \(z\) (we only need to consider each pair once, e.g. when \(y < z\), so we only scan the forward neighbors of \(y\)). For a total of \(\min(\omega(G) d \Delta, m)\) choices. The cost of each loop is given by \(O(\Delta)\) for computing \(I\), and \(O(\min(d \Delta, m))\) to compute \(X\) and \(\text{COMPLETE}(X)\). If the recursive call \(\text{ENUM(COMPLETE}(X)\)) will generate a new solution, the cost will be attributed to the child solution; otherwise the recursive call will only perform the \(\text{contains}(S, \text{RESULT})\) call.

The total cost of an iteration of \(\text{ENUM}(S)\) is thus the cost of the \(\text{contains/add}\) procedures, plus the number of execution of the loops times the cost of a loop iteration, i.e., \(O(\min(\omega(G) \log(\Delta), m) + \min(\omega(G) d \Delta, m) \cdot (\min(\omega(G) \log(\Delta), m) + \min(d \Delta, m)))\). Thus, the cost per solution is bounded by both \(O(\omega(G)d^2\Delta^2)\) and \(O(m^2)\).

Finally, as each recursive call outputs a solution, we can exploit the alternative output method by Uno [Uno03], as done in [MU04, CGMV16]: we output a solution at the beginning of a recursive call when the depth of the recursion tree is even, and at the end when it is odd; this way the delay of the algorithm will be equal to the amortized cost per solution.

Calling \(\alpha\) the number of solutions, this gives us a total time of \(\alpha \cdot O(\min(\omega(G)d^2\Delta^2, m^2))\). The space complexity is dominated by the size of the \(\text{RESULT}\) set, that is \(O(\alpha \cdot \omega(G))\) as it will contain \(\alpha\) solutions of size bounded by \(\omega(G) \leq d + 1 \leq n\). While \(\alpha\) is potentially exponential, we remark that \(\text{SCQ-ENUM}\) can still be efficiently applied to analyze real-world networks: recalling Corollary 6.2 and the discussion in Section 6.3.3, we have that \(\alpha\) will only be up to a factor \(\omega(u(G)) \leq n\) larger than the number of maximal (undirected) cliques in \(u(G)\). It is generally agreed upon that real-world networks are sparse [ELS13, GG06], and as such contain an extremely small number of maximal cliques compared to the theoretical maximum [SSTP09]. Furthermore, the number of maximal cliques is actually polynomial when the degeneracy (or arboricity) is bounded [ELS13], which is the case in many sparse networks.

### 6.6 Achieving polynomial space

While the number of maximal SCQs is likely to be much smaller than the maximum possible in real-world networks, achieving a small memory footprint is still beneficial in many scenarios, as remarked in Section 2.2.2.

In the following we present a new algorithm which offers a trade-off with respect to the previous one: it significantly decreases the space usage, at the cost of a higher delay, which however remains polynomial.

To meet this goal, we use the paradigm of reverse search. The most crucial part of obtaining a reverse search algorithm is how to obtain a suitable parent-child relationship. Cohen et al. [CKS08] showed how to do this for hereditary properties, and in Chapter 5 we showed how to extend this to commutable and strongly accessible properties.

In the following, we extend the techniques presented in Chapter 5 (in particular, the version of Section 5.4) to design a suitable parent-child relationship among maximal SCQs and exploit it to enumerate them with polynomial space usage and delay.

For the sake of brevity, in the following we use the term \(\text{block}\) to refer to a set of vertices of size 1 or 2 connected in \(G\) (i.e., either a vertex or a pair of adjacent vertices).

#### 6.6.1 Revised Lexicographical order

We consider a vertex \(a\) (lexicographically) smaller than another vertex \(b\) if \(a\) has smaller \(\text{label}\), \(a\) smaller than a pair \(\{c, d\}\) if the label of \(a\) is smaller or the same as the smallest label in the pair, and a pair smaller than another pair if it contains the vertex not in common with smallest label.
Consider the order in which blocks (vertices or adjacent pairs) are selected during the execution of $\text{ad}$ (with block $Y$ that is the function defined as $B$ and is clear) is the smallest $\text{scq}$ is a

To establish a suitable-parent child relationship, we exploit the some useful properties that follow directly from the definitions of $\text{scq}$ and block sequence.

Figure 6.2: Two $\text{scqs} S$ and $T$, the adding distances of the blocks in them w.r.t. the vertex with label 1, and their block sequence.

To this we add the notion of adding distance ($\text{ad}$) of a block, with respect to a (not necessarily maximal) $\text{scq}$. This concept will play for blocks the same role as the concept of layer for vertices in Section 6.4.

**Definition 6.1** (adding distance). Let $C$ be an $\text{scq}$, $x$ a vertex in $C$, and $Y$ a block s.t. $C \cup Y$ is a $\text{scq}$, although not necessarily disjoint from $C$.

The adding distance of $Y$ w.r.t $C$ and $x$ (denoted as $\text{ad}(Y,C,x)$ or simply $\text{ad}(Y)$ if the rest is clear) is the smallest $i$ for which there exist $i$ blocks $B_1, \ldots, B_{i-1} \subseteq C \setminus \{x\}$ s.t. $B_1 = \{x\}$ and $B_1 \cup \ldots \cup B_{i-1} \cup \{v\}$ is a $\text{scq}$. The adding distance of $x$ w.r.t $C$ and $x$, i.e. $\text{ad}(x,C,x)$ is defined as 0, and the adding distance of a vertex $v$ is given by the smallest adding distance of a block $Y$ s.t. $C \cup Y$ containing $v$ (including the block $\{v\}$), if any exists.

The example in Figure 6.2 shows two $\text{scqs} S$ and $T$: the grey number next to vertices and edges represent the adding distance of the block containing the vertex and the extremes of the edge, w.r.t. $S$ (or $T$) and the vertex 1.

In this section, the lexicographical order among vertices and pairs, i.e. blocks, is given by comparing first their respective adding distance, considering smaller those with smaller adding distance, and then comparing labels if the adding distance is equal.

The $\text{COMPLETE}$ function is naturally updated by the new lexicographical order, which redefines the min function used in $\text{MIN-EXTENSION}(X,A)$, for clarity we call this version $\text{COMPLETE}_{\text{ad}}$.

To define the new lexicographical order among solutions, we define the block sequence of a $\text{scq}$:

**Definition 6.2** (block sequence). Let $C$ be a maximal $\text{scq}$ and $v$ its vertex of smallest label. Consider the order in which blocks (vertices or adjacent pairs) are selected during the execution of the function $\text{COMPLETE}_{\text{ad}}(\{v\}, C)$. This defines a sequence of blocks $C_1, \ldots, C_k$, where $C_1 = \{v\}$, that is the block sequence of $C$.

Given two maximal $\text{scqs} S$ and $T$, let $i$ be the smallest index for which $S_i$ and $T_i$ have either different labels or different adding distance. Then $S < T$ iff $\text{ad}(S_i, S, T_1) < \text{ad}(T_i, T, T_1)$, or if the adding distances are the same but the set $S_i$ is smaller than $T_i$ (as recalled in the beginning of this section).

Figure 6.2 shows the block sequences for $S$ and $T$, as well as the adding distances of the blocks in the sequence. We can see that the block $\{2, 5\}$ (with $\text{ad} = 1$) is smaller than $\{3, 4\}$ (with $\text{ad} = 1$) and thus in this case $T < S$.

**6.6.2** Parent-child relationship

To establish a suitable-parent child relationship, we exploit the some useful properties that follow directly from the definitions of $\text{COMPLETE}_{\text{ad}}$ and block sequence.
Lemma 6.8. 1. The first block is the vertex $x$ with smallest label in $C$.

2. Blocks appear in increasing lexicographical order; i.e., $C_i$ is smaller than $C_j$ w.r.t. $C$ and $x$ iff $i < j$.

3. Every prefix of this sequence $C' = C_1 \cup \ldots \cup C_j$, $j \leq k$ is a scq.

Thanks to these properties and the suitable COMPLETE$_{ad}$ function, we can now define the parent PARENT($C$) and parent index $pi(C)$ of $C$ similarly to Section 5.4.

Definition 6.3 (parent, parent index, root). PARENT($C$) is COMPLETE$_{ad}$(C$_1$ $\cup$ $\ldots$ $\cup$ C$_j$), where $j > 0$ is the highest $j$ for which COMPLETE$_{ad}$(C$_1$ $\cup$ $\ldots$ $\cup$ C$_j$) $\neq$ C. $pi(C)$ is C$_{j+1}$. Note that COMPLETE$_{ad}$(C$_1$ $\cup$ $\ldots$ $\cup$ C$_{j+1}$) = C. If COMPLETE$_{ad}$(C$_1$) = C then $C$ has no parent and is called a root.

Note that, as $C_1$ is always composed of a single vertex, all roots can be identified as COMPLETE$_{ad}$($\{v\}$) for some $v \in V$. We also denote as children of $C$ all the solutions $C'$ s.t. PARENT($C'$) = $C$.

When calling the function COMPLETE$_{ad}$() on a prefix of the block sequence of $C$ we can observe this property:

Lemma 6.9. Let $C$ be a maximal scq, $C_1$, $\ldots$, $C_k$ its canonical block sequence and $1 < j \leq k$ an integer. Then COMPLETE$_{ad}$(C$_1$ $\cup$ $\ldots$ $\cup$ C$_j$) $\leq$ C.

Proof. If COMPLETE$_{ad}$(C$_1$ $\cup$ $\ldots$ $\cup$ C$_j$) = C the statement holds.

Otherwise, let $C'$ be COMPLETE$_{ad}$(C$_1$ $\cup$ $\ldots$ $\cup$ C$_j$), and recalling that $C_1$ and $C'_1$ correspond to the single vertex with smallest label in $C$ and $C'$ respectively, let these vertices be $c$ and $c'$. As $c \in C'$, $c'$ cannot have bigger label than $c$. If $c'$ has smaller label than $c$ the statement holds. Otherwise, $c' = c$: in this case, let $C'_i$, with $i > j$, be the first block selected by COMPLETE$_{ad}$(C$_1$ $\cup$ $\ldots$ $\cup$ C$_j$) that is different from $C_i$. As $C_i$ was a possible candidate to be selected, and COMPLETE$_{ad}$ select the lexicographically smallest one, then $C'_i$ is smaller than $C_i$ and thus $C'$ is smaller than $C$.

As PARENT($C$) is obtained by calling COMPLETE$_{ad}$() on a prefix of $X$, we can thus conclude that

Lemma 6.10. Let $C$ be a maximal scq that is not a root, than PARENT($C$) $< C$.

The above is a key property in that it ensures the absence of cycles in this graph, which means that the parent-child edges in the hypothetical solution graph will form a forest-like structure which makes the problem (i.e., the corresponding set system) reverse searchable, as in Definition 5.1 as long as we are able to compute all children of a given solution.

6.6.3 Finding child solutions

In this section we complete the algorithm by describing how to find, given a solution $C$, all solutions $D$ s.t. PARENT($D$) = $C$, i.e., all children of $C$, using polynomial time and space. This is the last step required in order to complete our algorithm.

Recalling Definition 6.3 if $D_1, \ldots, D_k$ is the block sequence of $D$, and $D_j$ ($j \leq k$) the parent index of $D$, then $C$ = COMPLETE$_{ad}$(D$_1$ $\cup$ $\ldots$ $\cup$ D$_{j-1}$). As the COMPLETE$_{ad}$ function only adds vertices, we thus have that (D$_1$ $\cup$ $\ldots$ $\cup$ D$_{j-1}$) $\subset$ C. This means that $D$ may be found from $C$ and $D_j$ by selecting a subset $C'$ of $C$ corresponding to $D_1$ $\cup$ $\ldots$ $\cup$ D$_{j-1}$, and performing COMPLETE$_{ad}$(C$'$ $\cup$ D$_j$), which will yield $D$ by Definition 6.3.
While the number of possible $D_j$, i.e. sets of one or two (adjacent) vertices, is $O(n^2)$ and thus polynomial, the number of possible subsets of $C$ is not. We thus need a way to prune out subsets of $C$ that will not lead to a solution. To do so we adapt the notion of restricted problem from [CKS08]:

**Definition 6.4** (restricted problem). Let $S$ be a maximal scq in $G(V, E)$ and $Y$ a block, i.e. a set of one or two (adjacent) vertices. The restricted problem for $(S, Y)$ consists in enumerating all maximal scqs in the graph $G[S \cup Y]$ induced by vertices of $S \cup Y$.

This definition differs from the one in [CKS08] in that we allow up to two vertices, rather than one, to be added to $S$. While the number of solution to the problem is potentially higher in this version, it has the following property:

**Lemma 6.11.** Let $S$ be a maximal scq in $G(V, E)$ and $Y$ a block. The restricted problem for $(S, Y)$ has at most one solution containing $Y$, that is $\text{COMPLETE}_{\text{ad}}(Y; C \cup Y)$.

**Proof.** A maximal scq in $G[S \cup Y]$ is contained in $T = (S \cup Y) \cap N(Y)$; whether $Y$ is made of one or two (adjacent) vertices, $T$ is a clique in the underlying graph $u(G)$. By Lemma 6.3 we have that the maximal scqs in $T$ are the strongly connected components of $T$. If $Y$ is made of two vertices in different strongly connected components of $T$, there is no scq containing $Y$; otherwise, the only solution of the restricted problem containing $Y$ is the strongly connected component of $T$ containing $Y$, which by definition of $\text{COMPLETE}_{\text{ad}}$ can be found as $\text{COMPLETE}_{\text{ad}}(Y; C \cup Y)$.

Let $S$ be the solution of the restricted problem $(C, D_j)$ containing $D_j$. We show the property that will be the cornerstone to the whole algorithm, i.e. that we are able to find the set $D_1 \cup \ldots \cup D_{j-1}$ using $S$:

**Lemma 6.12.** Let $C, D$ be two maximal scqs s.t. $C = \text{PARENT}(D)$, $D_j$ the parent index of $D$, $X = (D_1 \cup \ldots \cup D_{j-1})$ and $S = \text{REST-P}(C, D_j)$ the solution of the restricted problem $(C, D_j)$ containing $D_j$. Then the set of vertices in $S$ lexicographically smaller than $D_j$ w.r.t. $S$ and $v$ is exactly $X$.

**Proof.** First, note that $(D_1 \cup \ldots \cup D_{j-1}) \subset C$, and $(D_1 \cup \ldots \cup D_{j-1} \cup D_j)$ is a scq, thus by Lemma 6.11 $X \cup D_j = (D_1 \cup \ldots \cup D_j) \subseteq S$.

Now, assume there exists a block $Y \in S \setminus X$ lexicographically smaller than $D_j$ w.r.t. $S$ and $D_1$. This means that the adding distance of $Y$ is smaller or equal to that of $D_j$, i.e. there exists a subset $X'$ of $X$ s.t. $X' \cup Y$ is a scq. In turn, this means that $X \cup D_j \cup Y$ is a scq, since it is a clique in the underlying graph, and all vertices in $G[X \cup D_j \cup Y]$ can reach and be reached by vertices of $X'$. When computing $\text{COMPLETE}_{\text{ad}}(X \cup D_j)$, thus, $Y$ will be a candidate to be selected. As $Y$ is lexicographically smaller than $D_j$, the function (which selects the lexicographically smallest block) will select some $Y' \leq Y$, i.e., either $Y$ or something smaller than $Y$. This will mean that, when computing the block sequence of $D$, $Y$ will appear before $D_j$. This however is an absurd, as by assumption vertices of $Y$ are not in $X = (D_1 \cup \ldots \cup D_{j-1})$. Thus such $Y$ does not exist.

Finally, assume that a block $D_i$ s.t. $i < j$ is lexicographically bigger than $D_j$ w.r.t. $S$ and $D_1$. Again, this is an absurd, as $D_i$ would appear after $D_j$ in the block sequence of $D$.

Thus, all and only vertices in $S$ that are lexicographically smaller than $D_j$ w.r.t. $S$ and $D_1$ are the vertices of $X$.

**Corollary 6.3.** Let $C$, $D_j$ and $v$ be respectively the parent, parent index, smallest element of a maximal scq $D$, and and $S = \text{REST-P}(C, D_j)$, then $\text{COMPLETE}_{\text{ad}}(\{x \in S : x < D_j\text{ w.r.t. } S \text{ and } v\} \cup D_j) = D$.

109
Algorithm 17: List all maximal scqs with polynomial delay and space usage.

Input: A graph \( G(V,E) \)
Output: All maximal scqs in \( G \)

1. for \( v \in V \) do
2.   if \( v = \min_{\text{lab}}(\text{COMPLETE}_{\text{ad}}(\{v\})) \) then \( \text{ITER-ENUM}(\text{COMPLETE}_{\text{ad}}(\{v\})) \)

Function \( \text{ITER-ENUM}(S) \)

3.   \( C \leftarrow S ; Y \leftarrow \text{NEXT-BLOCK}(\text{null}) ; v \leftarrow \min_{\text{lab}}(\text{REST-P}(C,Y)) \)
4.   output(\( C \)) // depth \( \leftarrow 0 \)

5.   while true do
6.     if \( Y = \text{null} \) then
7.       output(\( C \)) // if depth is odd
8.     if \( C \) is a root then return
9.     else
10.    \( Y \leftarrow \pi(C) \) // backtrack to parent solution
11.    \( v \leftarrow \min_{\text{lab}}(\{x \in \text{REST-P}(\text{PARENT}(C),Y) : x > \min_{\text{lab}}(C)\}) \)
12.    \( C \leftarrow \text{PARENT}(C) \) // depth--
13.   if \( Y = \text{null} \) then
14.     \( Y \leftarrow \text{NEXT-BLOCK}(Y) ; v \leftarrow \min_{\text{lab}}(\text{REST-P}(C,Y)) \)
15.     continue
16.   \( D \leftarrow \text{COMPLETE}_{\text{ad}}(\{x \in \text{REST-P}(C,Y) : x < Y \text{ w.r.t. } \text{REST-P}(C,Y) \text{ and } v\}) \)
17.   if \( \text{PARENT}(D) = C \) and \( \pi(D) = Y \) and \( \min_{\text{lab}}(D) = v \) then
18.     \( Y \leftarrow \text{NEXT-BLOCK}(\text{null}) \) // visit child solution
19.     \( v \leftarrow \min_{\text{lab}}(\text{REST-P}(D,Y)) \)
20.    \( C \leftarrow D \) // depth++
21.     output(\( C \)) // if depth is even
22.   else
23.     \( v \leftarrow \min_{\text{lab}}(\{x \in \text{REST-P}(C,Y) : x > v\}) \)
24.     continue

Using the above lemma, we are thus able to find \( D \) from its parent \( C = \text{PARENT}(D) \) as long as we identify the parent index of \( D \), i.e. \( D_j \), and smallest vertex in \( D \), i.e. \( D_1 \). We can clearly do so in polynomial time as there are at most \( n^2 \) choices for \( D_j \) and \( |C| \) for \( D_1 \), which we recall consists of a single vertex. Thus we are able to find all children of a given solution in polynomial time and space.

6.6.4 Final algorithm

The final algorithm is shown in Algorithm 17. In order to maintain the space usage polynomial for the whole algorithm, we traverse the solution space using a stack-less iterative approach, similar to Algorithm 14 in Chapter 5.

Where \( \text{REST-P}(C,Y) \) returns the only solution of the restricted problem (Definition 6.4) containing \( Y \), i.e. \( \text{COMPLETE}_{\text{ad}}(Y,C \cup Y) \), or \( \text{null} \) if none exists.

\( \text{NEXT-BLOCK}(Y) \) returns the block after \( Y \) in lexicographical order (if \( \text{null} \) is the argument it returns the lexicographically smallest block), or \( \text{null} \) if \( Y \) is the last block in the order.

\( \min_{\text{lab}}(X) \) returns the vertex with smallest label \( X \), or \( \text{null} \) if \( X = \emptyset \).

Algorithm 17 is in fact simulating the traversal of the solution trees defined by our reverse-search characterization, starting from each root. At any time, the algorithm will be considering a
solution $C$, and will try to find all its children by testing all possible combinations of parent index and smallest element using Corollary 6.3 (Line 17). Whenever a child is found, the algorithm will simulate a nested recursive call by replacing $C$ with the child just found (Lines 18-22) whenever it runs out of possible combinations to test, i.e. possible choices of parent index $Y$ and minimum vertex $v$, the algorithm will simulate a backtrack by setting $C$ as the parent of the current solution. Furthermore, by setting $Y$ and $v$ as the ones that were being tested when $C$ was found, the algorithm will proceed to test only the combinations that were not already tested, to avoid producing the same child more than once. It follows that, for any solution that is found, Algorithm 17 will eventually attempt to combine it with every possible parent index and minimum vertex of a possible child.

The algorithm outputs a solution in three different points: Lines 5, 8 and 22. This is simply an implementation of the alternative output technique [Uno03], which reduces the delay of the algorithm: As the algorithm corresponds to a tree traversal, in order to reduce the maximum time between two outputs, we output the solution in a pre-order traversal fashion, i.e. as soon as it is found, when its depth in the tree is even (Lines 5 and 22), and in a post-order traversal fashion, i.e. when its subtree has been explored and we backtrack to its parent, when its depth is odd (Line 8).

6.6.5 Correctness

In this section we prove that Algorithm 17 visits all maximal scqs exactly once.

As every output (Lines 5, 8 and 22) outputs $C$, which is at all times the result of a complete or parent call, the algorithm only outputs maximal scqs. Thus we need to prove that every maximal scq (i.e., solution) is found exactly once.


Proof. Assume that Algorithm 17 does not find all solutions, and let $S$ be the lexicographically smallest solution which is not found. If $S$ is a root, then $S = \text{complete}_{\text{ad}}(\{\text{min}_{\text{lab}}(S)\})$, and it is found in Line 2. Thus $S$ is not a root and has a parent $P = \text{parent}(S)$; by Lemma 6.10 $P$ is found. By iterating over all possible blocks $Y$, and for each $Y$ over all vertices $v \in \text{rest-p}(P,Y)$, eventually we will have that $Y = \pi(D)$ and $v = \text{min}_{\text{lab}}(D)$. It follows by Lemma 6.3 that $D$ is found in Line 17, an absurd. Finally, we need to show that no solution is found more than once: if $S$ is a root, then it is identified by its smallest vertex $v$; since $v$ is considered only once in Line 2 then so is $S$. Otherwise, $S$ has a unique parent, parent index and minimum element, and it is found when these three values are considered in Line 18. Consider Line 12 when backtracking from $C$ to its parent solution $\text{parent}(C)$, we know that $C$ was previously found (in Line 18 using $\text{parent}(C)$, $\pi(C)$ and $\text{min}_{\text{lab}}(C)$. We set $v$ as the smallest element larger than $\text{min}_{\text{lab}}(C)$ in $\text{rest-p}(\text{parent}(C), \pi(C))$. If no other $v$ exists, we get the next lexicographically larger $Y$ (Line 15). This way, we ensure that the same triplet parent/parent index/minimum element is not considered again, thus $S$ is found exactly once.

6.6.6 Complexity analysis

Recalling the algorithm description in Section 6.6.4, we can bound the cost per solution as the sum of all operations that are performed while considering a given solution $C$.

First, however, we need to bound the cost for the $\text{complete}_{\text{ad}}$ and $\text{parent}$ functions.

Lemma 6.14. $\text{complete}_{\text{ad}}(X, A)$, $\pi(S)$ and $\text{parent}(S)$ can be computed in $O(q \cdot \min(d\Delta, m))$ time using $O(\Delta)$ space.
Proof. The first necessary step is computing the adding distance of the blocks in \( A \) that may be added.

Let \( v = \min_{\text{ad}}(X) \), \( X' = \{ v \} \), and \( \text{sink}, \text{source}, \text{both} \) the tri-partition of \( \bigcap_{x \in X'}(N(x)) \cap A \) analogous to that in the proof of Lemma 6.6 which takes \( \mathcal{O}(|X'|^2) \) time.

All the blocks with adding distance 1 w.r.t \( X' \) and \( v \) are either vertices in \( \text{both} \), or pairs \( x \in \text{sink}, y \in \text{source} \) with an edge \( (x, y) \) (from \( x \) towards \( y \)). All such block can be found in \( \mathcal{O}(\Delta) \) for the vertices and \( \mathcal{O}(d\Delta) \) for the pairs thanks to the degeneracy ordering, by checking for each vertex in \( \text{sink} \cup \text{source} \) its forward neighbors. All such vertices are “tagged” with \( \text{ad} = 1 \), and the smallest is selected for addition to \( X' \). While \( X \setminus X' \neq \emptyset \) we select the smallest \( Y \) in \( X \setminus X' \) rather than in \( A \). When adding \( Y \), we update the sets \( \text{sink}, \text{source}, \text{both} \) as in proof of Lemma 6.6 for each vertex in \( Y \), but when adding a vertex \( x \) to \( \text{both} \), we tag \( x \) with \( \text{ad}(Y) + 1 \), unless \( x \) was already tagged with a smaller \( \text{ad} \): indeed the adding distance of \( Y \) is the smallest amount of blocks other than \( Y \) needed to build a SCQ containing \( v \) and \( Y \); \( x \) already had an edge towards (or from) \( v \), and as it is added to \( \text{both} \) it has an edge from (or towards) \( Y \), thus adding \( Y \) itself to such blocks allows us to add \( x \), making \( \text{ad}(x) \) at most \( \text{ad}(Y) + 1 \).

Once all elements from \( X \) have been selected, all elements in \( A \setminus X \) are tagged with their proper adding distance. We can thus continue this process, selecting vertices and updating \( \text{ad} \) for the remaining ones until \( X' \) is maximal in \( A \), paying a total cost of \( \mathcal{O}(\min(d\Delta, m)) \) time.

If, however, a vertex \( v' \) with label smaller than \( v \) is selected and added to \( X' \), all adding distances need to be computed again, paying an additional \( \mathcal{O}(\min(d\Delta, m)) \). As this may happen up to \( q \) times, the total cost is \( \mathcal{O}(q \cdot \min(d\Delta, m)) \). The space required corresponds to that used for storing \( \text{sink}, \text{source}, \text{both} \), that is \( \mathcal{O}(\Delta) \).

As for \( \text{pr}(S) \), let \( S_1 = \{ \min_{\text{ad}}(S) \} \), \( S_1, \ldots, S_j \) the block sequence of \( S \) (which can be computed by performing \( \text{COMPLETE}_{\text{ad}}(S_1, S) \)). Consider \( \text{COMPLETE}_{\text{ad}}(X = S_1) \): we repeat the above procedure to compute the adding distances, but rather then selecting the smallest block, we always select and add to \( X \) the next block \( S_i \) in the block sequence. It is straightforward to see that the parent index \( S_j \) of \( S \) is the last block in the sequence that was not the lexicographically smallest block that could be selected. \( \text{PARENT}(S) \) can then be computed as \( \text{COMPLETE}_{\text{ad}}(S_1 \cup \ldots \cup S_{j-1}) \), for a total cost of \( \text{pr} \) and \( \text{parent} \) equal to that of \( \text{COMPLETE}_{\text{ad}} \).

Lemma 6.15. Algorithm 17 has delay \( \mathcal{O}(\min(q^3d^2\Delta^2, m^2q^2)) \).

Proof. All the possible blocks \( Y \) correspond exactly to all the candidate sets of size one or two in Algorithm 15 that is \( \min(q\Delta, n) \) possibilities for blocks of size one, and \( \min(qd\Delta, m) \) possibilities for blocks of size two (see proof of Lemma 6.7), for a total of \( \mathcal{O}(\min(qd\Delta, m)) \). In Algorithm 17, however, we need to account for all the possible choices of minimum vertex \( v \), which are at most \( |C \cup Y| = \mathcal{O}(q) \).

Note that the cost of every line in Algorithm 17 is dominated by the cost of either a \( \text{REST-P()} \), \( \text{COMPLETE}_{\text{ad}}() \), or \( \text{PARENT()} \) call. As \( \text{REST-P()} \) consists essentially in a \( \text{COMPLETE}_{\text{ad}}() \) call, this cost is \( \mathcal{O}(q \cdot \min(d\Delta, m)) \) by Lemma 6.14.

The total cost per solution is thus \( \mathcal{O}(\min(qd\Delta, m) \cdot q \cdot q \cdot \min(d\Delta, m)) \), thus it is bounded by both \( \mathcal{O}(q^3d^2\Delta^2) \) and \( \mathcal{O}(m^2q^2) \).

As we are using alternative-output \cite{Uno03}, printing solutions in pre-order when the depth is even (Lines 5 and 22), and in a post-order when the depth is odd (Line 5), the delay is the same as the cost per solution times a constant, thus the proof is complete.

As for the space, it is in principle possible to obtain minimal space usage as in Chapters 3 and 4. However, efficiently implementing the \( \text{PARENT} \) operation requires \( \mathcal{O}(\Delta) \) space, thus the additional space used by Algorithm 17 will be \( \mathcal{O}(\Delta) \).
6.7 Final remarks

In this chapter we proposed a model for communities in directed graphs, that of maximal strongly connected cliques. We analyzed this model, giving tight bounds on the number of such cliques in an \( n \)-vertex graph and proving some accessibility properties. We exploited these properties to produce \texttt{scq-enum}, an algorithm that lists maximal strongly connected cliques with polynomial delay that can be a valid tool for analyzing the community structures of directed real-world networks. The delay can be \( O(\min(\omega(G)d^2\Delta^2, m^2)) \) if the whole result set is stored, or \( O(\min(q^3d^2\Delta^2, m^2q^2)) \) but using just \( O(\Delta) \) additional space.

A remarkable feature of this problem is the loose notion of accessibility found in it: even though it is not strongly accessible, or even weakly accessible, we are able to identify maximal \texttt{scq}s in polynomial time, and we are able to build upon state of the art technique for strongly accessible set systems in order to produce an enumeration algorithm with polynomial delay.

This result opens the way for interesting developments: the first is using the proposed algorithm to discover new community structures in directed networks, while the second is to further investigate the generalized definition of accessibility given by the existence of an addible (or removable) set of elements of bounded size to each non-maximal solution.
In this chapter we study efficient algorithms for enumerating the orientations of an undirected graph. An orientation of an undirected graph is obtained by assigning a direction to each of its edges. It is called cyclic when a directed cycle appears, otherwise it is called acyclic. If any two vertices are connected by a directed path in both ways, the orientation is called strong. To get the full picture, we consider both the cases of acyclic and cyclic orientations, under some rules specifying which vertices are the sources (i.e. their incident edges are all directed outwards). We also consider strong orientations, which are a special case of cyclic ones, and have long been studied in relation with road networks. The enumeration algorithms proposed use linear space and provide new bounds for the delay. We obtain a delay of $O(m)$ for acyclic orientations and $O(m)$ for cyclic ones. When just a single source is specified, these delays become $O(m \cdot n)$ and $O(m \cdot h + h^3)$, respectively, where $h$ is the girth of the graph without the given source. When multiple sources are specified, the delays are the same as in the single source case. As for strong orientations, we obtain amortized cost per solution of $O(m)$ using linear space, or $O(m)$ delay at the cost of $O(m^2)$ space usage and preprocessing. Furthermore, we show how the latter algorithm can be applied to a realistic model for road networks where, e.g., roads can be one-way or two-way. From a technical point of view, the algorithms presented in this chapter all use the binary partition technique. Recalling the recursion tree structures defined in Section 2.3.2, this chapter contains examples of both ugly and good trees. The results also show the significance of these classes of trees, as all algorithms characterized by an ugly tree achieve polynomial delay, and those characterized by a good tree further achieve strongly output sensitive bounds (ignoring poly-logarithmic factors in one case).

7.1 Introduction

An orientation of an undirected graph is any of the directed graphs that may be obtained by assigning directions to its edges. As such, the terminology of directed graphs applies to orientations: they are called cyclic when containing a directed cycle, and acyclic otherwise. Their sources are the vertices whose incident edges are all directed outwards. Strong orientations are those for which any vertex can reach and be reached by any other through a directed path. An example of graph orientation is given in Figure 7.1. Acyclic orientations have been studied in depth. Motivated by the fact that each acyclic orientation corresponds to a partial order for the underlying graph, Iriarte [Iri14] investigated which orientations maximize the number of linear extensions of the corresponding poset. Alon and Tarsi [AT92] looked for special orientations to
give bounds on the size of the maximum independent set or the chromatic number. Gallai, Roy, and Vitaver independently stated that every orientation of a graph with chromatic number \( k \) contains a simple directed path with \( k \) vertices [EKT68, Roy67, Vit62]. Johnson [Joh84] links acyclic orientations with exactly one source to problems of network reliability.

There are further problems that can be addressed by looking at acyclic orientations. For instance, Benson et al. [BCT10] showed that there exists a bijection between the set of the so-called superstable configurations of a graph and the set of its acyclic orientations with a unique source. Counting the number of acyclic orientations is a problem dating back to the 70s or earlier [Sta86], and Linial [Lin86] proved that this problem is \#P-complete. Stanley [Sta73] showed how to count them using the chromatic polynomial (a special case of Tutte’s polynomial).

A related problem is the acyclic orientation game. Alon and Tuza [AT95] inquired about the amount of oriented edges needed to define a unique orientation, and found this number to be almost surely \( \Theta(n \log n) \) in Erdős-Rényi random \( n \)-vertex graphs. Pikhurko [Pik10] showed that the number of these edges in the worst case is no greater than \( (\frac{1}{4} + o(1))n^2 \) in general.

Cyclic orientations have received less attention. Counting them is easily shown to be \#P-complete: a graph with \( m \) edges has exactly \( 2^m \) orientations, which are either cyclic or acyclic. Counting the latter latter ones is \#P-complete [Lin86], and we may obtain the number of acyclic orientations by subtracting that of cyclic one from \( 2^m \): it follows that counting cyclic orientations is \#P-complete too. Fisher et al. [FFLW97] studied the number of dependent edges, i.e. edges generating a cycle if reversed in an orientation. This number of edges implicitly gives a hint about the number of cyclic orientations in a graph.

We think that cyclic orientations have interesting properties to study, even though they seem more artificial and have much less applications than acyclic ones. However they can hopefully contribute to give a full picture, as acyclic and cyclic orientations correspond to an exact partition of all the \( 2^m \) possible orientations of a graph. As listing all orientations of a graph is straightforward, one might think that solving one problem yields the result for the other. This however is not at all satisfactory as we will see. Furthermore, techniques and invariants for one problem do not apply to the other: acyclic orientation can be obtained incrementally by keeping acyclicity as an invariant. This can be seen as a somewhat “global” property with no proper translation for the cyclic case: cyclic orientation are based on the “local” property of the existence of a directed cycle somewhere in the graph, and this cycle may appear at any time during an incremental approach. Ensuring the satisfiability of this “local” cycle-existence property, while seemingly easy, appears to yield slower algorithms than for the acyclic case. This hints at the possibility of the problem being harder, or simply that stronger techniques are required.

Strong orientations, that are a special case of acyclic orientations, have been object of many studies since the 30s [Rob39], due to their natural application to road networks and other traffic networks containing links that may be traversed in only one direction.

Roberts [Rob78] defines the one-way street problem, as that of finding whether a graph can be oriented so that every vertex can reach all others, a clear requirement for transportation networks. Remarkably, the necessary and sufficient condition was already given by Robbins [Rob39]: such an orientation exists if and only if the graph is 2-edge connected, i.e., any removal of a single edge does not disconnect the graph.

Several papers by Roberts and Xu deal with these feasible directions [RX89, RX88, RX92, RX94] in the one-way street problem. The results reduce the latter to the problem of finding a strong orientation of a mixed multigraph, which is a multigraph where both directed and undirected edges coexist. Robbins’ theorem has been extended by Boesch and Tindel [BT80] accordingly, and Chung et al. [CGT85] describe a linear time algorithm for finding a strong orientation in a mixed multigraph.

Some variations of the one-way street problem have been considered with the purpose of
minimizing the average \cite{DOW04} or the maximum \cite{FMR04, FMPR04, Gut94, LK12} distance among all pairs of vertices, both of which are NP-hard problems \cite{CT78, Ple84} (see \cite{KT02} for a survey). Moreover, the minimum diameter among all the strong orientations of a given graph has been shown to be related with its domination number \cite{FMPR04}. Other variations consider, for instance, the distance stretch for each pair of vertices \cite{HM89}, other connectivity constraints \cite{AH02}, cost-based constraints \cite{BFKW99}, degree-based constraints \cite{BAGN15}, and forced orientations \cite{CHSW94}. As remarked in Section 1.1.1 in domain such as this one where many useful goal functions arise, an efficient enumeration algorithm can be a handy tool as it prevents the need to create new ad-hoc algorithms for every considered goal.

Problems studied This chapter considers both acyclic and cyclic orientations, investigating how to enumerate them. We will use the terms enumerating and listing interchangeably. We are given an undirected connected graph $G(V, E)$ without self loops, where $n = |V|$ and $m = |E|$, and we will study the following cases.

Single Source Orientations (sso): Given a vertex $s \in V$, enumerate all the orientations of $G$, such that $s$ is the only source.

Single Source Acyclic Orientations (ssao): Given a vertex $s \in V$, enumerate all the acyclic orientations of $G$, such that $s$ is the only source.

Acyclic Orientations (ao): Enumerate all the acyclic orientations of $G$.

Single Source Cyclic Orientations (ssco): Given a vertex $s \in V$, enumerate all the cyclic orientations of $G$, such that $s$ is the only source.

Cyclic Orientations (co): Enumerate all the cyclic orientations of $G$.

Strong Orientations (so): Enumerate all the strong orientations of $G$.

As we will see the above variations of orientations provide an interesting playground for exploring enumerations algorithms on graphs. We propose new efficient algorithms and analyze their cost in terms of delay, which is a well-known measure of performance for enumeration algorithms corresponding to the worst-case time between any two consecutively enumerated solutions (e.g. \cite{JYP88}). We will focus on algorithms with guaranteed delay and space. Furthermore, we show in Section 7.8 how the proposed algorithms can be used to solve a wider range of orientation listing problems, in particular with respect to multiple sources.

Previous work We are not aware of any listing algorithm for sso. The result of \cite{SPO85} gives necessary and sufficient conditions for the existence of a single source orientation and a search algorithm to find few of them. However, this algorithm cannot be easily revised for enumeration purposes.
Problem AO has been investigated by Squire [Squ98]. His algorithm has a good amortized cost of $O(n)$ time per solution, but the delay can be $O(n^3)$ time. The algorithm by Barbosa and Szwarcfiter [BS99] solves AO with an amortized time complexity of $O(n + m)$ per solution, and the delay is $O(n \cdot m)$. An alternative way of solving AO is by applying any algorithm for maximal feedback arc set enumeration (after replacing each edge with a double arc). State of the art approaches for the latter problem incur in a delay of $\Omega(n^3)$ as shown by Schwikowski and Speckenmeyer [SS02].

It should be noted that AO has a correspondence with cell enumeration in an arrangement of hyperplanes, which has been solved in the paradigm of reverse search [AF99]. Even though such algorithm can be designed to be memory efficient, e.g. by employing the techniques in [CGMV16], its delay is $\Omega(n \cdot m)$.

SSAO seems to be harder to solve, as all the techniques above for AO (including the one in [Squ98]) do not extend smoothly. Johnson [Joh84] presented a backtracking algorithm for SSAO, aimed at solving problems on network reliability. However its complexity is hard to estimate, as it is based on a backtracking approach with dead ends. For this reason Squire [Squ98] writes that he has been unable to efficiently implement Johnson’s approach. As far as we know, there are no provably good bounds in the literature for problems SSAO and their variants, other than the special case in which $G$ is a planar biconnected graph, and a single target $t$ adjacent to $s$ is allowed [SSN11].

Regarding ssco and co, the best known method is to enumerate them by difference: go through all possible $2^m$ orientations and eliminate, say, the acyclic ones. This method does not guarantee polynomial delay as the number $\beta = 2^m - \alpha$ of cyclic orientations can be much larger or much smaller than $\alpha$: for example, a tree with $m$ edges has $\alpha = 2^m$ and $\beta = 0$. On the other hand, a clique with $n$ vertices and $m = \frac{n(n-1)}{2}$ edges has $\alpha = n!$, i.e. the possible transitive tournaments [Moo68], and $\beta = 2^n - n!$. As $2^m$ grows faster than $n!$, the ratio $\alpha/\beta = n!/ (2^m - n!)$ tends to 0 for increasing $n$.

Finally, the works mentioned above regarding SOS and the one-way street problem do not extend efficiently to our problem. By Robbins’ theorem [Rob39], the graphs that admit SOSs are exactly the 2-edge connected graphs: in these graphs, for every pair of vertices there are two edge-disjoint paths connecting them; hence, if $G$ is not 2-edge connected, the corresponding road network has no feasible direction. Its proof contains the following remarkable hint, which can be used to find all the SOSs, but has some issues: given an ear decomposition of $G$, it is possible to produce a SO by orienting each ear as a directed path, thus obtaining $2^k$ SOSs from an ear decomposition with $k$ ears. In general, listing ear decompositions and then obtaining SOSs seems a natural approach to our problem. However, two different ear decompositions can lead to the same SO. Figure 7.2 shows two possible ear decompositions of a graph yielding the same SO: first orient the cycle $\{1, 2, 3, 4, 5\}$ clock-wise in both orientations, then in the left one orient the ears as $(1, 6, 7)$ and $(3, 8, 7, 5)$, whereas in the right one as $(3, 8, 7)$ and $(1, 6, 7, 5)$. It is easy to generalize this example, so that the same SO is obtained by many distinct ear decompositions.

Figure 7.2: Two ear decompositions (left and right) and a SO obtained from both (center)
Problem | Previous work | Proposed algorithm
--- | --- | ---
SSO | unknown | $O(m)$
SSAO | unbounded | $O(m \cdot n)$
AO | $O(n^3)$ | $O(m)$
SSCO | unknown | $O(m \cdot h + h^3)$
CO | unknown | $\tilde{O}(m)$
SO | unknown | $O(m)$

Table 7.1: Summary of the delay of our listing algorithms and that of state of the art algorithms, where $m$ and $n$ are respectively the number of edges and vertices of the graph, and $h$ is the girth of the graph without the given source $s$.

A possible way to list once all the sos would be to consider one edge at a time and employ the algorithm in [CGT85] to check which orientations of that edge will lead to a solution. This approach would yield a recursive algorithm taking $O(m^2)$ time per solution because of the $O(m)$ recursion depth. It is natural to ask whether $O(m)$ time is possible, as each solution requires $O(m)$ to be output.

Results For SSO and SSAO we design the first enumeration algorithms with guaranteed delay of $O(m)$ and $O(m \cdot n)$, respectively, using $O(m)$ space. For AO, we show how to obtain $O(m)$ delay: even though the latter result does not improve the amortized cost of $O(n)$ in [Squ98, BS99], it improves their delay and that of SS02. For CO and SSCO, we provide the first enumeration algorithms with guaranteed delay, respectively, of $\tilde{O}(m)$ and $O(m \cdot h + h^3)$, where $h < n$ is the girth (length of the shortest cycle) of the graph without the given source. Results are summarized in Table 7.1. These problems have versions with multiple sources, instead of single, that can be easily solved by our algorithms (see Section 7.8) with the same delay. This chapter is divided into three parts. Part I describes the algorithms for SSO, SSAO and AO in Sections 7.3–7.5, while Part II describes those for SSCO and CO in Sections 7.6–7.7, and shows how to solve some variations of these enumeration problems, with respect constraints on the sources. Finally, Part III describes how the enumeration of feasible orientations of a realistic road network can be reduced to that of listing SO on a simple graph, and describes an algorithm for listing such orientations in $O(m)$ amortized cost per solution, using linear space, or $O(m)$ delay using $O(m^2)$ space and preprocessing time.

**7.2 Preliminaries**

We refer to Section 2.1 for terminology. In this section we will consider an undirected connected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges. An orientation of an undirected graph $G = (V(G), E(G))$ is the directed graph $\overrightarrow{G}(V(G), E(G))$ where for any pair of undirected edge $\{u, v\} \in E$ either the directed arc $(u, v)$ or $(v, u)$ is in $\overrightarrow{E}$. We call $\overrightarrow{E}$ an orientation of $E$. We say that the orientation $\overrightarrow{G}$ is acyclic when it does not contain cycles, or cyclic otherwise. If $\overrightarrow{G}$ is strongly connected, that is, each node has a directed path to all others, we say that the orientation is strong.

For the sake of clarity, in the following we will call edges the unordered pairs $\{x, y\}$ (undirected graph), while we will call arcs the two possible orientations $(x, y)$ and $(y, x)$ (directed graphs). We assume wlog that $G$ is connected and does not contain self-loops.
Recall that \( V_{\leq i} \) refers to the set \( \{v_1, \ldots, v_i\} \), \( N(v) = \{x: \{v, x\} \in E\} \) as the set of neighbors of the vertex \( v_i \), \( N_{\leq i}(v) \) as the restricted set \( N(v) \cap V_{\leq i} \), and that \( N_{< i}(v) \) means \( N_{\leq j}(v_j) \).

### Part I: Algorithms for Acyclic Orientations

The algorithms for sso, ssao and ao follow a common scheme. Starting from an empty directed graph \( \overrightarrow{G}_0 \), they add \( v_i \) to the previous directed graph \( \overrightarrow{G}_{i-1} \) for increasing values of \( i = 1, 2, \ldots, n \). For this, we define a direction assignment \( \overrightarrow{Z}_i \) as an orientation of the set of edges \( \{\{v_i, x\}: x \in N_{< i}(v_i)\} \). We denote two special direction assignments by

\[ X_i = \{(x, v_i): x \in N_{< i}(v_i)\} \]

\[ Y_i = \{(v_i, x): x \in N_{< i}(v_i)\} \]

The algorithms for sso, ssao and ao explore only the “valid” direction assignments to obtain the resulting directed graphs \( \overrightarrow{G}_i \): every time vertex \( v_n \) is reached, the corresponding \( \overrightarrow{G}_n \) is an orientation of \( G \) and thus is output as a new solution \( \overrightarrow{G} \). The algorithms employ different definitions of “valid”, as discussed in Sections 7.3–7.5.

#### 7.3 Single source orientations (sso)

We illustrate the basic scheme in some detail. We recall that in sso the orientations \( \overrightarrow{G} \) must have \( s \) as their only source. To avoid dead ends, we exploit a suitable ordering of the vertices.

**Definition 7.1** (full vertex). For \( 1 \leq j \leq i \) a vertex \( v_j \) is full in \( \overrightarrow{G}_i \) if \( N_{\leq i}(v_j) = N(v_j) \). We will also simply call a vertex full if \( \overrightarrow{G}_i \) is clear from the context.

**Definition 7.2** (valid direction assignment). Given \( \overrightarrow{G}_{i-1}(V_{\leq i-1}, \overrightarrow{E}) \), the direction assignment \( \overrightarrow{Z}_i \) is valid if every full vertex \( v_j \neq s \) in \( \overrightarrow{G}_{i}(V_{\leq i}, \overrightarrow{E} \cup \overrightarrow{Z}_i) \) is not a source, for \( 1 \leq j \leq i \).

The rationale is the following. When dealing with \( \overrightarrow{G}_i \), there are no unassigned incident edges for the full vertices: if any of them is a source, it will remain a source in the final orientation \( \overrightarrow{G} \). Hence, only \( s \) can be a full vertex that is a source. To guarantee this, we order the vertices suitably.

**Definition 7.3.** An ordering of the vertices \( v_1, \ldots, v_n \) is good if

- \( v_n = s \), and
- \( N_{<}(v_i) \neq N(v_i) \), for \( 1 \leq i < n \).

The first condition in Definition 7.3 says that \( s \) should be the last vertex as it is the only source. The second condition says that there is at least one unassigned incident edge for each added \( v_i \). We avoid dead ends when adding \( v_i \) to \( \overrightarrow{G}_{i-1} \), as at least one solution extending \( \overrightarrow{G}_{i-1} \) exists such that \( v_i \) is not a source. To see why, let us call \( \overrightarrow{G}_i \) a partial orientation (of \( \overrightarrow{G} \)) and give the following property.

**Property 7.1.** For any partial orientation \( \overrightarrow{G}_i \), there is always an orientation \( \overrightarrow{G} \) for \( G \) that has unique source \( s \) and extends \( \overrightarrow{G}_i \).

**Proof.** Consider the direction assignments \( Y_j \) \((j > i)\) and the resulting acyclic \( \overrightarrow{G} \). These direction assignments are valid as they cannot create new sources due to the good ordering, thus the only final source in \( \overrightarrow{G} \) is \( s \).
Algorithm 18: single-source-orientations

Input: \( G(V,E) \), partial orientation \( \overrightarrow{G}_{i-1}(V_{\leq i-1}, E) \), integer \( i \)
Output: Orientations of \( G \) containing \( \overrightarrow{G}_{i-1}(V_{\leq i}, E) \) with source \( s \)
1 if \( i > n \) then output \( \overrightarrow{G}_i \); return
2 for any valid direction assignment \( \overrightarrow{Z}_i \) for \( v_i \) starting from \( \overrightarrow{Z}_i = Y_i \) do
3 single-source-orientations(\( G, \overrightarrow{G}_i(V_{\leq i}, E \cup \overrightarrow{Z}_i), i+1 \))

Algorithm 19: Returning valid direction assignments for sso

Input: Graph \( G(V,E) \), partial orientation \( \overrightarrow{G}_{i-1}(V_{\leq i-1}, E) \), vertex \( v_i \)
Output: Valid direction assignments \( \overrightarrow{Z}_i \)
1 \( E_i \leftarrow \) set of full vertices in \( \overrightarrow{G}_i \), that are sources and not full in \( \overrightarrow{G}_{i-1} \)
2 \( \overrightarrow{Z}_i \leftarrow \{(v_i,y) : y \in F_i \} \)
3 Let \( x_1, \ldots, x_k \) be the vertices in \( N_<(v_i) \setminus F_i \)
4 Execute Generate \( (G, \overrightarrow{G}_i(v_i, \overrightarrow{Z}_i, 1)) \).
1 Procedure Generate \( (G(V,E), \overrightarrow{G}_{i-1}(V_{\leq i-1}, E), v_i, \overrightarrow{W}, j) \)
2 if \( j > k \) then add \( \overrightarrow{W} \) to the output list; return
3 Generate \( (G, \overrightarrow{G}_{i-1}, v_i, \overrightarrow{W} \cup \{(v_i,x_j)\}, j+1) \)
4 Generate \( (G, \overrightarrow{G}_{i-1}, v_i, \overrightarrow{W} \cup \{(x_j,v_i)\}, j+1) \)

A good ordering for \( G \) and \( s \) can be found in linear time by performing a DFS from \( s \) and considering its vertices in postorder. Observe that this is a good order according to our definition: \( s \) is the last vertex and, for each vertex, its parent in the DFS tree appears after it in the order.

Algorithm 18 details how to proceed according to the good ordering of the vertices. The initial call is single-source-orientations\( (G, \overrightarrow{G}_0, 1) \). The algorithm recursively explores all the possible ways of expanding the current partial solution \( \overrightarrow{G}_{i-1} \) by iterating over all the valid direction assignments, starting from \( Y_i \), which is surely valid by Property 7.1. The assignments are generated by a recursive computation. We have the primary recursion tree to generate all the wanted orientations (Algorithm 18), where each vertex has associated a secondary recursion tree to generate locally all the valid direction assignments as explained next.

Iterating over valid direction assignments

Algorithm 19 generates all and only the valid direction assignments \( \overrightarrow{W} \), for a given \( \overrightarrow{G}_{i-1} \) and a vertex \( v_i \), among the possible \( 2^{|N_<(v_i)|} \) ones. Let \( F_i \) be the set of vertices that are sources and not full in \( \overrightarrow{G}_{i-1} \) and that become full in \( \overrightarrow{G}_i \), where \( F_i \subseteq N_<(v_i) \). All the valid direction assignments should guarantee that vertices in \( F_i \) are not sources in \( \overrightarrow{G}_i \): this can be easily done by adding the arcs of \( \overrightarrow{Z}_i \) to \( \overrightarrow{W} \) (and this is mandatory as \( s \) is the only source). After that, we have to decide the orientation of the remaining edges, involving \( v_i \) and the vertices in \( N_<(v_i) \setminus F_i \). This part relies on procedure Generate. In particular, we have to assign a direction to the edges \( \{v_i, x_j\} \) for each vertex \( x_j \) in \( N_<(v_i) \setminus F_i \): both the directions \( (v_i, x_j) \) and \( (x_j, v_i) \) are explored.

Since both \( F_i \) and \( N_<(v_i) \setminus F_i \) are subsets of \( N(v_i) \), Lemma 7.1 holds.

Lemma 7.1. Algorithm 19 finds valid direction assignments with delay \( O(|N(v_i)|) \).

Lemma 7.2. Referring to Algorithms 18 and 19, the following holds. (1) All the orientations of \( G \) whose unique source is \( s \) are output; (2) only the orientations of \( G \) whose unique source is \( s \) are output; (3) there are no duplicates.

Proof. We prove the three statements separately using the good ordering of the vertices. (1) Consider the following process. For decreasing values of \( j \) remove \( v_j \) from \( \overrightarrow{G} \), and let \( \overrightarrow{W}_j \) be the
Lemma 7.3. Algorithm 18 has delay \( O(m) \) and uses space \( O(m) \).

Proof. We exploit the properties of the primary recursion tree induced by Algorithm 18: notice that each internal node (i.e., recursion node that is not a leaf) has at least one child because of Property 7.1. This means that all the leaves correspond to a solution. The depth of the recursion tree is \( O(n) \) as one new node is considered at each step. Thus, the delay of Algorithm 18 is bounded by the sum of the costs along a leaf-to-root and root-to-next-leaf path. This includes the cost of finding the first solution. Both paths consist in \( O(\sum_{v \in V} |\text{N}(v)|) = O(m) \). The space requirement is bounded by the information stored in a root-to-leaf path of the primary recursion. The graph takes \( O(m) \) space. The working space is dominated by the Generate iterators (calls to Algorithm 19), that is, all direction assignments \( \overrightarrow{W}_1, \ldots, \overrightarrow{W}_m \) on the root-to-leaf path, and is bounded by \( O(m) \). \qed

7.4 Single source acyclic orientations (SSAO)

For a given vertex \( s \) in the graph \( G \), we now want to enumerate the acyclic orientations \( \overrightarrow{G} \) such that \( s \) is the unique source. We apply the scheme of Section 7.3 with a different definition of valid direction assignment: it does not create cycles and each full vertex (except \( s \)) is not a source.

Definition 7.4 (valid direction assignment). Given \( \overrightarrow{G}_{i-1}(V_{\leq i-1}, \overrightarrow{E}) \), the direction assignment \( \overrightarrow{Z}_i \) is valid if

- \( \overrightarrow{G}_i(V_{\leq i}, \overrightarrow{E} \cup \overrightarrow{Z}_i) \) is acyclic, and
- any \( v_j \neq s \) that is full in \( \overrightarrow{G}_i(V_{\leq i}, \overrightarrow{E} \cup \overrightarrow{Z}_i) \) is not a source, for \( 1 \leq j \leq i \).

Given the good ordering in Definition 7.3, the following property, which is an adaptation of Property 7.1, holds.

Property 7.2. For any partial acyclic orientation \( \overrightarrow{G}_i \), there is always an acyclic orientation \( \overrightarrow{G} \) for \( G \) that has unique source \( s \) and includes \( \overrightarrow{G}_i \).

Proof. Consider the special assignments \( Y_j \) for \( j > i \). The only final source is \( s \) and no cycle is created as these \( Y_j \)'s induce a total ordering. \qed

To solve ssao we reuse Algorithm 18 and introduce Algorithm 20 that is obtained from Algorithm 19 by a different version of Generate described next.

7.4.1 Iterating over valid direction assignments

Consider the generic step when adding vertex \( v_i \). We have to include the arcs from \( \overrightarrow{Z}_i \) to form \( \overrightarrow{W} \) and, for the remaining edges, apply Generate. The latter is in charge of assigning a direction to the edges \( \{v_i, x_j\} \) for each of the vertices \( x_1, \ldots, x_k \in \text{N}_c(v_i) \setminus F_i \), and check if they do not create cycles and sources different from \( s \). Namely, for \( j = 1, 2, \ldots, k \), if the arc \( e \in \{v_i, x_j, (x_j, v_i)\} \) does not create a cycle in \( \overrightarrow{G}_i(V_{\leq i}, \overrightarrow{E} \cup \overrightarrow{W}) \), Generate proceeds recursively with \( \overrightarrow{W} = \overrightarrow{W} \cup \{e\} \).
We need to perform efficiently the reachability tests for \( x_j \) to update sets \( B \) and \( R \), where \( B \) corresponds to vertices that can lead to \( v_i \), while \( R \) corresponds to the ones reachable from \( v_i \). A naive approach requires \( O(k) \) DFS traversals in total \( O(k \cdot m) \) time. We show how to reduce the latter cost to \( O(m) \) time by truncating the DFSes whenever they touch a vertex in \( B \) or \( R \).

Since the partially built directed graph is acyclic, \( B \) and \( R \) are disjoint, and a vertex can belong to either one of them or none of them. Below we provide an analysis based on a simple coloring scheme.

**Lemma 7.4.** Algorithm 20 returns valid direction assignments with delay \( O(m) \).

**Proof.** The arcs in \( Z_k \) can be computed in \( O(m) \) time at the beginning. We discuss the rest of the arcs (i.e. \( \overrightarrow{W} \setminus \overrightarrow{Z}_j \)). We have to decide whether \((v_i, x_j)\) or \((x_j, v_i)\) creates a cycle or not, and we color incrementally the vertices for this purpose: all the vertices \( R \) reachable from \( v_i \) are red; all the vertices \( B \) that can lead to \( v_i \) are black; the remaining vertices are uncolored. Since \( G_i(V_{\leq i}, \overrightarrow{E} \cup \overrightarrow{W}) \) is acyclic, any vertex has just one color or is uncolored.

Initially, all the vertices are uncolored, and \( R \) and \( B \) are empty. We show that the sum of the costs to update \( R \) and \( B \) to produce a solution through the recursive calls of \( \text{Generate} \) is \( O(m) \). Since each leaf in the secondary recursion tree induced by \( \text{Generate} \) corresponds to a distinct solution, we should bound the sum of the costs along the \( k + 1 \) vertices from the root to that leaf. Specifically, the delay is upper bounded by the sum of the costs along two paths: (a) the leaf-to-root path of the current solution and (b) the root-to-next-leaf path for the next solution (actually only the latter for the first solution).

Observe that the cost of (a) is always \( O(|N_{\leq i}(v_i)|) \). As for (b), it is bounded by \( O(m) \) as follows. When \( j = 1 \), the red colors are assigned with a forward traversal and the black colors are assigned with a backward traversal in the graph \( G_i(V_{\leq i}, \overrightarrow{E} \cup \overrightarrow{W}) \). When \( j > 1 \), while adding the arc \((v_i, x_j)\) to \( \overrightarrow{W} \) we have only to make the traversed uncolored vertices red: since the forward traversal is rooted at \( v_i \), we continue the traversal avoiding to visit red vertices. (No black vertex can be reached, otherwise \( G_i \) would be cyclic). On the other hand, when adding the arc \((x_j, v_i)\) to \( \overrightarrow{W} \) we only have to make the traversed uncolored vertices black: once again, this corresponds to continuing the backward traversal rooted in \( v_i \) avoiding to visit black vertices (no red vertex can be reached). Since this process traverses each arc at most once for any \( 1 \leq j \leq k \), the sum of the costs of a root to leaf path in the secondary recursion tree induced by the \( \text{Generate} \) procedure is \( O(m) \). \qed

**Remark 7.1.** After the last valid direction assignment has been returned, Algorithm 20 recognizes that there are no more valid direction assignments, using extra \( O(m) \) time.
As it can be seen in Algorithm 18 we employ Generate as an iterator through all the valid direction assignments $\vec{Z}_i$. Its state consists in the last direction assignment returned and the information carried by $R$ and $B$.

**Lemma 7.5.** The state of the Generate iterator described in Algorithm 20 can be rebuilt in $O(m)$ time from the last direction assignment returned.

**Proof.** We need to mark for each vertex which edge in $\vec{Z}_i$ caused it to be added to $R$ or $B$. This can be done with a DFS rooted in $v_i$ in which edges leaving $v_i$ are visited in the same order as they were added to $\vec{Z}_i$; each time a new vertex is visited, the last edge of $\vec{Z}_i$ traversed was the one that caused it to be added to the set. For $B$ the visit is done by following the edges backwards. The state of the iterator is thus restored in the time required by a DFS, i.e. $O(m)$.

**Lemma 7.6.** Referring to Algorithm 18 and Algorithm 20 the following holds.

1. All the acyclic orientations of $G$ whose unique source is $s$ are output.
2. Only the acyclic orientations of $G$ whose unique source is $s$ are output.
3. There are no duplicates.

**Proof.** Similar to the proof of Lemma 7.2.

**Theorem 7.1.** Problem ssao can be solved with delay $O(n \cdot m)$ and space $O(m)$.

**Proof.** We solve the problem with Algorithm 18 but using Algorithm 20 instead of Algorithm 19 to generate the valid direction assignments. The proof is similar to that of Lemma 7.3. Again, the depth of the tree is $O(n)$, and each node leads to at least a solution (Property 7.2), meaning that each leaf is a solution. It follows that the delay is similarly bounded by the sum of the costs along a leaf-to-root and root-to-next-leaf path. The former is bounded by $O(n \cdot m)$: the height of the tree is $O(n)$ and each return we spend $O(m)$ to recognize that no more valid direction assignments are possible, as highlighted by Remark 7.1. The latter is still bounded by $O(n \cdot m)$, applying $n$ times Lemma 7.4 for a total delay of $O(n \cdot m)$. As for Algorithm 18 the space requirement is bounded by the information stored in a root-to-leaf path of the primary recursion. The input data and the sum of all direction assignments take $O(m)$ space. The state of Generate iterator takes $O(n)$ space for each node, but it does not need to be stored as it can be rebuilt from the last direction assignment generated when backtracking, as stated by Lemma 7.5. As rebuilding the state is done once for node and takes $O(m)$ time, this procedure takes $O(n \cdot m)$ which does not affect the total delay. Hence the algorithm has delay $O(n \cdot m)$ and space requirement $O(m)$.

**7.5 Acyclic orientations (AO)**

Differently from Sections 7.3–7.4, we have no restrictions about the possible sources when adding $v_i$. Hence we define the concept of valid direction assignment as follows.

**Definition 7.5** (valid direction assignment). Given $\vec{G}_{i-1}(V_{\leq i-1}, \vec{E})$, a direction assignment $\vec{Z}_i$ is valid if $\vec{G}_i(V_{\leq i}, \vec{E} \cup \vec{Z}_i)$ is acyclic.

Another difference from the previous section is that we do not need the good order (Definition 7.3). Namely, for any order of the vertices we can prove that the following property holds.
Proof. Trivial, as $v_i$ becomes either a source or a sink (i.e. its incident edges are all directed inwards) in $G_i$, thus cycles are not created. By induction $G$ can be oriented completely. \qed

Algorithm 21 provides a way of iterating over valid direction assignments. It explores all of these similarly to Algorithm 20. Notice that the first valid direction assignment produced is $X_i$ and the last valid direction assignment is $Y_i$. Moreover observe that the update of $R$ and $B$ is respectively not required when $W_i \cap Y_i = \emptyset$ and $W_i \cap X_i = \emptyset$, since these conditions mean respectively that the outdegree and the indegree of $v_i$ is zero in $G_i(V_{\leq i}, E \cup \overrightarrow{W})$. Summing up, the overall scheme remains the same as that of Algorithm 18. By Property 7.3 $Z_i = X_i$ and $Z_i = Y_i$ are always taken. The corresponding two recursive calls are done respectively at the beginning and at the end of the procedure. All the other valid direction assignments (if any) are explored in the other calls of the for cycle.

Lemma 7.7. Referring to Algorithm 18 and Algorithm 21 the following holds.

1. All the acyclic orientations of $G$ are output.
2. Just the acyclic orientations of $G$ are output.
3. There are no duplicates.

Proof. Similar to the proof of Lemma 7.2 \qed

Lemma 7.8. Algorithm 21 returns the first valid direction assignment in time $O(|N_<(v_i)|)$, and the remaining ones with delay $O(m)$.

Proof. Similar to the proof of Lemma 7.4 except that $X_i$ is returned in time $O(|N_<(v_i)|)$ since the update of $R$ is not needed and never performed for $X_i$ in Algorithm 21 \qed

By using Lemma 7.8 we obtain the following.

Theorem 7.2. Problem AO can be solved with delay $O(m)$ and space $O(m)$. 

125
Proof. We employ Algorithm 18 along with Algorithm 21 as iterator for valid direction assignments. The proof is similar to those for sso and ssao (see Theorem 7.1): while we bounded the delay with a leaf-to-root and root-to-next-leaf path, here we consider the lowest common ancestor node \( z \) between two consecutive leaves \( a \) and \( b \) (note that \( z \) might still be the root). As every node leads to a leaf/solution, we can assume that the path from \( a \) to \( z \) is made of last children \((Y_i)\), as otherwise there would be another leaf between \( a \) and \( b \). For the same reason, the path from \( z \) to \( b \) is made of first children \((X_i)\). As the backtracking cost after a last child is \( O(1) \), the cost of selecting the next valid direction assignment in \( z \) is \( O(m) \), and the cost of recurring in the first child is \( O(|N(v_i)|) \) (see Lemma 7.8), we have that the delay of the algorithm is \( O(n + m + \sum_{i \in V} |N(v_i)|) = O(m) \).

Part II: Algorithms for Cyclic Orientations

Problems co and scco address cyclic orientations. The algorithm for co (Section 7.6) preprocesses the graph in a way that ensures the existence of a cycle \( C \) of logarithmic length in the size of the graph. It then considers all the orientations of the residual graph after \( C \)'s removal, leading to an extended version of co that can assign to each edge a direction or a broken state. The algorithm finally reintegrates \( C \) in the solutions for the extended version, exploiting all suitable orientations of \( C \). For the sake of completeness, we also show in Section 7.7 how to solve scco. One crucial difference with the algorithm for co is the fact that we cannot perform the preprocessing phase so that we have no guarantee about the logarithmic length of \( C \), which negatively affects the running time.

7.6 Cyclic orientations (CO)

The intuition behind our algorithm is as follows. Suppose that \( G(V, E) \) is cyclic, otherwise there are no cyclic orientations. Consider a cycle \( C(V_C, E_C) \) in \( G \): we can orient its edges in two ways so that the resulting \( \overrightarrow{C} \) is a directed cycle\(^1\). At this point, any orientation of the remaining edges \( E_G \setminus E_C \), will give a cyclic orientation of \( G \). Thus we generate all possible orientations of the edges in \( E_G \setminus E_C \), and then assign some suitable orientations to the edges in \( E_C \). This guarantees that we have at least two solutions for each orientation of \( E_G \setminus E_C \), namely, setting the orientation of \( E_C \) so that \( \overrightarrow{C} \) is one of the two possible directed cycles. Yet this is not enough as we may have a cyclic orientation even if \( \overrightarrow{C} \) is acyclic.

Therefore we have to handle some cases in general. One easy case is that the orientation of \( E_G \setminus E_C \) already induces a directed cycle: any orientation of \( E_C \) will give a cyclic orientation of \( G \). Another easy case, as seen above, is when \( E_C \) is oriented such that \( \overrightarrow{C} \) is a directed cycle: any orientation \( \overrightarrow{G} \) including \( \overrightarrow{C} \) will be cyclic. The remaining case is not trivial: when the orientations of \( E_G \setminus E_C \) and \( E_C \) are individually acyclic, putting them together might or not induce a cyclic orientation of \( G \). To efficiently deal with the latter case, we need to “massage” \( G \) and transform it into a multigraph as summarized in Algorithm 22 and discussed below.

7.6.1 Algorithm setup

We reduce the problem of enumerating cyclic orientations to an extended version that allows us to neglect bridges and chains.

Bridge and degree-0 vertex removal. Given an undirected graph \( G(V, E) \), a bridge is an edge whose removal increases the number of connected components. As no cycle in \( G \) can contain

\(^1\)We will actually use a chordless cycle of logarithmic size (called log-hole).
**Algorithm 22**: Returning all the cyclic orientations of $G$

**Input**: An undirected connected graph $G(V, E)$

**Output**: All the cyclic orientations $\mathcal{O}(V, E)$

1. **Algorithm setup (Section 7.6.1)**:
   2. $G(V, E) \leftarrow$ apply bridge and degree-0 vertex removal to $G$
   3. $M(V, M) \leftarrow$ chain compression of $G$
   4. $C(V, M) \leftarrow$ any log-hole of $M$
   5. $M'(V, M') \leftarrow$ delete $C$’s edges from $M$ (i.e. $E'_M = E_M \setminus E_C$)

6. **Enumerate cyclic orientations (Section 7.6.2)**:
   7. for each extended orientation $M''$ of multigraph $M$ do
   8. for each legal orientation $\hat{C}$ of log-hole $C$ (see Algorithm 23) do
   9. $M''(V, M'') \leftarrow$ combine $M'$ and $\hat{C}$, where $\hat{E}'' = \hat{E}'_M \cup \hat{E}_C$
   10. Output each of the cyclic orientations $\mathcal{O}$ of $G$ corresponding to $\hat{M}''$

---

a bridge, we can remove all bridges from $G$. After a cyclic orientation of the remaining edges is produced, all bridges can be reintegrated in the graph with all possible direction assignments.

For the sake of simplicity, we also remove isolated vertices (i.e. vertices of degree zero), so that all remaining vertices have degree 2 or greater. The bridges of a graph can be found in linear time [Tar74]. Finding and removing bridges and removing isolated vertices can be done in $O(m)$ time and space.

**Chain compression.** Consider a maximal path $v_1, \ldots, v_k$ where $v_i$ has degree 2 (with $2 \leq i \leq k - 1$). As the internal vertices of the path have degree 2, this path can only be part of a directed cycle if it is a directed path from $v_1$ to $v_k$, or from $v_k$ to $v_1$. We exploit this property by replacing each of these paths with a single edge, called *chain edge*. A chain edge can be given three direction assignments: $(v_1, v_k)$ and $(v_k, v_1)$, which correspond to the directed paths mentioned above, and *broken*, that is any other direction assignment (which does not induce a directed path). Identifying and compressing all chains can be accomplished in $O(m)$ time by traversing the graph $G$ in a DFS fashion from a vertex of degree $\geq 3$.

The output is an undirected multigraph $M(V, M)$, since it might contain parallel edges or loops. Vertices in $V_M \subseteq V$ are the vertices of $V$ whose degree is $\geq 3$, and edges in $E_M$ are the chain edges plus all the edges in $E$ which are not part of a chain. We call the latter ones simple edges to distinguish them from the chain edges. In the rest of the chapter, $M$ will be considered a multigraph where $|V_M| \geq 4$ and each of the edges has a label in \{simple, chain\}. For this, we define the concept of extended orientation as follows.

**Definition 7.6 (Extended Orientation).** For a multigraph $M(V, M)$ having self loops and edge labels in \{simple, chain\}, an extended orientation $\hat{M}(V, M)$ is a directed multigraph whose arc set $\hat{E}_M$ assigns a direction or broken to each edge in $E_M$: in particular, for any simple edge $(u, v) \in E_M$, exactly one direction between $(u, v)$ and $(v, u)$ is assigned; for any chain edge $(u, v) \in E_M$, either the edge is broken, or exactly one direction between $(u, v)$ and $(v, u)$ is assigned. A directed cycle in $\hat{M}$ cannot contain a broken edge.

We can explore extended orientations to list cyclic orientations with the following lemma.

**Lemma 7.9.** If we have an algorithm that lists all the extended cyclic orientations of $M(V, M)$ with delay $f(|E_M|)$, for some $f : \mathbb{N} \to \mathbb{N}$, then we can produce an algorithm that lists all the cyclic orientations of the graph $G(V, E)$ with delay $O(f(|E_M|) + |E|)$.

\(^2\)if no such vertex exists $G$ is a cycle or a path and the listing problem is trivial.
We recall that our goal is to list all the cyclic orientations of $G$ with degree at least one, like simple edges. We will use the following well-known result.

**7.6.2 Enumerating cyclic orientations**

- **Removing the log-hole.** Remove the edges in $E_C$ from $M$, obtaining $M' = M - E_C$.

We will use the following well-known result.

**Lemma 7.10** (Logarithmic girth). Let $G(V,E)$ be a graph in which every vertex has degree at least 3. The girth of $G$ is at most $2[\log |V|]$.

As every vertex in $M$ has degree $\geq 3$, this means that the log-hole $C$ of $M$ has length at most $2[\log |M|] + 1$, thus motivating our terminology.

The log-hole $C$ can be found by applying the algorithm in [IR78], which easily extends to multigraphs, in time $O(|V_M|^2)$: indeed, the algorithm finds a cycle that is either of minimum size, or larger by one. If chords are present in $C$, we can check whether $C$ includes a smaller cycle and redefine $C$ accordingly in time $O(|C|^2) = O(\log^2 |V_M|)$.

**7.6.2 Enumerating cyclic orientations**

We recall that our goal is to list all the cyclic orientations of $G$. By Lemma 7.9 this is equivalent to listing the extended cyclic orientations of $M$. We now show that the latter task can be done by suitably combining some orientations from the labeled multigraph $M'$ and the log-hole $C$ using the following steps.

1. **Finding extended orientations.** Enumerate all extended orientations (not necessarily cyclic) $M'$ of the multigraph $M'$ (Section 7.6.3).

---

3Minimum cycle means any cycle having minimum number of edges (e.g. a self loop). Chain edges count just one, like simple edges.
2. **Putting back the log-hole.** For each listed \( \overrightarrow{M'(V_M, E'_M)} \), consider all the extended orientations \( \overrightarrow{C'(V_C, \overrightarrow{E_C})} \) of the log-hole \( C \) such that \( \overrightarrow{E'_M \cup \overrightarrow{E_C}} \) contains a directed cycle, and obtain the extended cyclic orientations for the multigraph \( M \) (Section 7.6.4).

### 7.6.3 Finding extended orientations

This is an easy task. For each edge \( \{u, v\} \) in \( E'_M \) that is labeled as simple, both the directions \((u, v)\) and \((v, u)\) can be assigned; if \( \{u, v\} \) is labeled as chain, the directions \((u, v)\) and \((v, u)\), and broken can be assigned. Each combination of these decisions produces an extended orientation of \( M'(V_M, E_M) \). If there are \( s \) simple edges and \( c \) chain edges in \( M' \), where \( s + c = |E'_M| \), this generates all possible \( 2^s3^c \) extended orientations. Each of them can be easily listed in \( O(|E'_M|) \) delay (actually less, but this is not the dominant cost).

### 7.6.4 Putting back the log-hole

For each listed \( \overrightarrow{M'} \) we have to decide how to put back the edges of the cycle \( C \), namely, how to find the orientations of \( C \) that create directed cycles.

**Definition 7.7.** Given the cycle \( C(V_C, E_C) \) and \( \overrightarrow{M'(V_M, E'_M)} \), we call legal orientation \( \overrightarrow{C'(V_C, \overrightarrow{E_C})} \) any extended orientation of \( C \) such that the resulting multigraph \( \overrightarrow{M''(V_M, \overrightarrow{E''})} \) is cyclic, where \( \overrightarrow{E''} = \overrightarrow{E'_M \cup \overrightarrow{E_C}} \).

The following two cases are possible.

1. \( \overrightarrow{M'} \) is cyclic. In this case each edge in \( E_C \) can receive any direction, including broken if the edge is a chain edge: each combination of these assignments will produce a legal orientation that will be output.

2. \( \overrightarrow{M'} \) is acyclic. Since \( C \) is a cycle, there are at least two legal orientations obtained by orienting \( C \) as a directed cycle clockwise and counter-clockwise. Moreover, adding just an oriented subset of edges \( D \subseteq C \) to \( \overrightarrow{M'} \) may create a cycle in \( \overrightarrow{M'} \); in this case, any orientation of the remaining edges of \( C \setminus D \) (including broken for chain edges) will clearly produce a legal orientation.

While the first case is immediate, the second case has to efficiently deal with the following problem.

**Problem 7.1.** Given \( \overrightarrow{M'} \) acyclic and cycle \( C \), enumerate all the legal orientations \( \overrightarrow{C'(V_C, \overrightarrow{E_C})} \) of \( C \).

In order to solve Problem 7.1, we exploit the properties of \( C \). In particular, we compute the reachability matrix \( R \) among all the vertices in \( V_C \), that is, for each pair \( u, v \) of vertices in \( V_C \), \( R(u, v) = 1 \) if \( u \) can reach \( v \) in \( \overrightarrow{M'} \), 0 otherwise. We say that \( R \) is cyclic whether there exists a pair \( i,j \) such that \( R(i,j) = R(j,i) = 1 \). This step can be done by performing a BFS in \( \overrightarrow{M'} \) from each vertex in \( V_C \); by Lemma 7.10 we have \( |V_C| \leq 2|\log |V_M|| + 1 \), and so the cost is \( O(|E'_M| \cdot \log |V_M|) \) time. Deciding the orientation of the edges and the chain edges in \( E_C \) is done with a ternary partition of the search space described below.

**Scheme for legal orientations** The steps are shown in Algorithm 23. At the beginning the reachability matrix \( R \) is computed as described above, and passed to the recursive routine **Legal Orientations**. At each step, \( \overrightarrow{C'} \) is the partial legal orientation to be completed and \( I \) is the set of broken edges declared so far. Also, \( j \) is the index of the next edge \( \{c_j, c_{j+1}\} \) of the cycle \( C \), with \( 1 \leq j \leq h \) (we assume \( c_{h+1} = c_1 \) to close the cycle): if \( j = h + 1 \) then all the edges
Algorithm 23: Returning all legal orientations of $C$

\begin{algorithm}
\begin{algorithmic}
\Input{$\vec{M}'(V_M, E_M')$ acyclic, a cycle $C(V_C, E_C)$ with $V_C \subseteq V_M$}
\Output{All the legal orientations $\vec{C}(V_C, E_C)$}
\State Build the reachability matrix $R$ for the vertices of $V_C$ in $\vec{M}'$
\State Let $V_C = \{c_1, \ldots, c_h\}$, where $c_{h+1} = c_1$ by definition
\State Execute \texttt{LegalOrientations} $(\vec{C}'(0,0), 1, R, \emptyset)$
\Procedure{LegalOrientations}{$\vec{C}'(V_C', E_C'), j, R, I$}
\If{$j = h + 1$} \texttt{output $\vec{C}'$ and its set $I$ of broken edges}
\Else
\State $u \leftarrow c_j$, $v \leftarrow c_{j+1}$
\State $R_1 \leftarrow R$ updated by adding the arc $(u, v)$
\If{$R_1$ is cyclic or has positive reachability test on $\{c_{j+1}, \ldots, c_{h+1}\}$}
\State \texttt{LegalOrientations} $(\vec{C}'(V_C', E_C' \cup \{(u, v)\}), j + 1, R_1, I)$
\Else
\State $R_2 \leftarrow R$ updated adding the arc $(v, u)$
\If{$R_2$ is cyclic or has positive reachability test on $\{c_{j+1}, \ldots, c_{h+1}\}$}
\State \texttt{LegalOrientations} $(\vec{C}'(V_C', E_C' \cup \{(v, u)\}), j + 1, R_2, I)$
\Else
\If{$\{u, v\}$ is a chain edge}
\State \texttt{if $R$ is cyclic or has positive reachability test on $\{c_{j+1}, \ldots, c_{h+1}\}$}
\State \texttt{LegalOrientations} $(\vec{C}', j + 1, R, I \cup \{(u, v)\})$
\EndIf
\EndIf
\EndIf
\EndProcedure
\end{algorithmic}
\end{algorithm}

of $C$ have been considered and we output the solution $\vec{C}'$ together with the list $I$ of broken edges in $\vec{C}'$. Each time the procedure is called we guarantee that the reachability matrix $R$ is updated.

Let $\{u, v\}$ be the next edge of $C$ to be considered, where $u = c_j$ and $v = c_{j+1}$: for each possible direction assignment $(u, v)$ or $(v, u)$ of this edge, we have to decide whether we will be able to complete the solution considering this assignment. This is done by trying to add the arc to the current solution. If there is already a cycle, clearly we can complete the solution. Otherwise, we perform a \textit{reachability check} on $\{c_{j+1}, \ldots, c_{h+1}\}$: it is still possible to create a directed cycle if and only if any two of the vertices in $\{c_{j+1}, \ldots, c_{h+1}\}$, say $c_f$ and $c_g$ satisfy $R(c_f, c_g) = 1$ or $R(c_g, c_f) = 1$. This condition guarantees that a cycle will be created in the next calls, since we know there are edges in $C$ between $c_f$ and $c_g$ that can be oriented suitably. Finally, when $\{u, v\}$ is a chain, the broken assignment is also considered: $R$ does not need to be updated as the broken edge does not change the reachability of $\vec{M}'$.

We discuss now how to perform the reachability and cyclicity checks. Updating $R$ when adding an arc $(u, v)$ corresponds to making $v$, and all vertices reachable from $v$, reachable from $u$ and vertices that can reach $u$. This can be done by simply performing an or between the corresponding rows in time $O(\log^2 |V_M|)$, since $R$ is $|C| \times |C|$. The reachability check can be done in $O(\log^2 |V_M|)$ time. The cyclicity (checking whether a cycle has been already created) takes the same of time by looking for a pair of vertices $x', y'$ in $\{c_1, \ldots, c_j\}$ such that $R(x', y') = R(y', x') = 1$. We also have to restore $R$ but the cost is dominated by the rest.

Lemma 7.11. Algorithm \texttt{23} outputs in time $O(|E_M'| \log |V_M|)$ the first legal orientation of $C$, and each of the remaining ones with $O(\log^3 |V_M|)$ delay.

Proof. Before calling \texttt{LegalOrientations} we have to compute the reachability matrix from scratch and this costs $O(|E_M'| \log |V_M|)$ time. In the following we bound the delay between two outputs returned by \texttt{LegalOrientations}. Firstly, note that each call produces at least one solution. This is true when $j = 1$ since we have two possible legal orientations of $C$. Before
performing any call at depth $j$, the caller function checks whether this will produce at least one solution. Only calls that will produce at least one solution are then performed. This gives a recursion tree similar to the ones seen for ssao, where every internal node has at least one child and each leaf corresponds to a solution. Hence the delay between any two consecutive solutions is bounded by the cost of a leaf-to-root path and the cost of a root-to-the-next-leaf path in the recursion tree induced by \texttt{LegalOrientations}. Since the height of the recursion tree is $O(\log |V_M|)$, i.e. the edges of $C$, and the cost of each recursion node is $O(\log^2 |V_M|)$, delay between any two consecutive solutions is bounded by $O(\log^3 |V_M|)$. As it can be seen, it is crucial that the size of $C$ is (poly)logarithmic. 

\begin{lemma}
Regarding Algorithm\texttt{22}, the following properties hold:
\begin{enumerate}

\item All the extended cyclic orientations of $\mathcal{M}$ are output.
\item Only extended cyclic orientations of $\mathcal{M}$ are output.
\item There are no duplicates.
\end{enumerate}
\end{lemma}

As a result, we obtain an algorithm with delay $\tilde{\mathcal{O}}(|E_M|)$.

\begin{lemma}
The extended cyclic orientations of $\mathcal{M}(V_M, E_M)$ can be enumerated with delay $\tilde{\mathcal{O}}(|E_M|)$ and space $O(|E_M|)$.
\end{lemma}

\begin{proof}
Finding extended orientations $\mathcal{M}'$ of $\mathcal{M}'$ can be done with $O(|E_M'|)$ delay. Every time a new $\mathcal{M}'$ has been generated, we apply Algorithm\texttt{23} By Lemma\texttt{7.11} we output the first cyclic orientation $\mathcal{M}$ of $\mathcal{M}$ with delay $O(|E_M|\log |V_M|)$ and the remaining ones with delay $O(|E_M|\log |V_M|)$. Hence the maximum delay between any two consecutive solutions is $O(|E_M'| + |E_M|\log |V_M|) = O(|E_M'| + |E_M|\log |V_M|) = \tilde{\mathcal{O}}(|E_M'|)$. The space usage is linear: in particular in Algorithm\texttt{23} the space is $O(\log^2 |V_M|)$, because of the reachability matrix $R$, which is smaller than $O(|E_M|)$.
\end{proof}

By Lemma\texttt{7.13} and Lemma\texttt{7.9}, and considering the setup cost in Section\texttt{7.6.1} ($|V_M| \leq |V|$ and $|E_M| \leq |E|$), we can conclude as follows.

\begin{theorem}
Algorithm\texttt{23} lists all cyclic orientations of $G(V, E)$ with setup cost $O(|V|^2)$ and delay $\tilde{\mathcal{O}}(|E|)$. The space usage is $O(|E|)$.
\end{theorem}

It is possible to modify our approach to get a setup time equal to the delay, requiring space $\Theta(|V| \cdot |E|)$.

\begin{theorem}
All cyclic orientations of $G(V, E)$ can be listed with setup cost $\tilde{\mathcal{O}}(|E|)$, delay $\tilde{\mathcal{O}}(|E|)$, and space usage of $\Theta(|V| \cdot |E|)$.
\end{theorem}

\begin{proof}
We use $n = |V|$ and $m = |E|$ for brevity. Let $A_1$ be the following algorithm that takes $T_1 = O(mn)$ time to generate $n$ solutions, each with $\tilde{\mathcal{O}}(m)$ delay, starting from any given cycle of size $\geq \log n$. This cycle is found by performing a BFS on an arbitrary node $u$, and identifying the shortest cycle $C_u$ containing $u$. Now, if $|C_u| < \log n$, since $C_u$ is a log-hole as required, we stop the setup and run the algorithms in the previous sections setting $C = C_u$. The case of interest in this section is when $|C_u| \geq \log n$. We take a cyclic orientation $C_u^2$ of $C_u$, and then $n$ arbitrary orientations of the edges in $G \setminus C_u$. The setup cost is $O(m)$ time and we can easily output each solution in $\tilde{\mathcal{O}}(m)$ delay. We denote this set of $n$ solutions by $Z_1$.

Also, let $A_2$ be the algorithm behind Theorem\texttt{7.3} with a setup cost of $O(mn)$ and $\tilde{\mathcal{O}}(m)$ delay (i.e. Algorithm\texttt{22}). We denote the time taken by $A_2$ to list the first $n$ solutions, including the $O(mn)$ setup cost, by $T_2 = \tilde{\mathcal{O}}(mn)$, and this set of $n$ solutions by $Z_2$. Since $Z_1$ and $Z_2$ can have nonempty intersection, we want to avoid duplicates.
We show how to obtain an algorithm $A$ that lists all the cyclic orientations without duplicates with $\hat{O}(m)$ setup cost and delay, using $O(mn)$ space. Even though the delay cost of $A$ is larger than that of $A_1$ and $A_2$ by a constant factor, the asymptotic complexity is not affected by this constant, and remains $\hat{O}(m)$.

Algorithm $A$ executes simultaneously and independently the two algorithms $A_1$ and $A_2$. Recall that these two algorithms take $T_1 + T_2$ time in total to generate $Z_1$ and $Z_2$ with $\hat{O}(m)$ delay. However those in $Z_2$ are produced after a setup cost of $\hat{O}(mn)$. Hence $A$ slows down on purpose by a constant factor $c$, thus requiring $c(T_1 + T_2)$ time: it has time to find the distinct solutions in $Z_1 \cup Z_2$ and build a dictionary $D_1$ on the solutions in $Z_1$. (Since an orientation can be represented as a binary string of length $m$, a binary trie can be employed as dictionary $D_1$, supporting each dictionary operation in $\hat{O}(m)$ time.) During this time, $A$ outputs the $n$ solutions from $Z_1$ with a delay of $c(T_1 + T_2)/n = \hat{O}(m)$ time each, while storing the rest of solutions of $Z_2 \setminus Z_1$ in a buffer $Q$.

After $c(T_1 + T_2)$ time, the situation is the following: Algorithm $A$ has output the $n$ solutions in $Z_1$ with $\hat{O}(m)$ setup cost and delay. These solutions are stored in $D_1$, so we can check for duplicates. We have buffered at most $n$ solutions of $Z_2 \setminus Z_1$ in $Q$. Now the purpose of $A$ is to continue with algorithm $A_2$ alone, with $\hat{O}(m)$ delay per solution, avoiding duplicates. Thus for each solution given by $A_2$, algorithm $A$ suspends $A_2$ and waits so that each solution is output in $c(T_1 + T_2)/n$ time: if the solution is not in $D_1$, $A$ outputs it; otherwise $A$ extracts one solution from the buffer $Q$ and outputs the latter instead. Note that if there are still $d$ duplicates to handle in the future, then $Q$ contains exactly $d$ solutions from $Z_2 \setminus Z_1$ (and $Q$ is empty when $A - 2$ completes its execution). Thus, $A$ never has to wait for a non-duplicated solution. The delay is the maximum between $c(T_1 + T_2)/n$ and the delay of $A_2$, hence $\hat{O}(m)$. The additional space is dominated by that of $Q$, namely, $O(mn)$ space to store up to $n$ solutions.  

We also have a bound of $\hat{O}(|E_M|)$ on the amortized cost using the lemma below with $f(x) = \hat{O}(x)$ and $s = |V|$.

**Lemma 7.14.** Listing all the extended cyclic orientations of $M(V_M, E_M)$ with delay $O(f(|E_M|))$ and setup cost $O(s \cdot |V_M|)$ implies that the average cost per solution is $O(f(|E_M| + |E_M|)).$

**Proof.** We perform a BFS on an arbitrary node $u$, and identify the shortest cycle $C_u(V_u, E_u)$ that contains $u$. This costs $O(m)$ time. Note that $C_u(V_u, E_u)$ is a hole (i.e. it has no chords). Note that a minimum cycle in $M$ either is $C_u$ or contains a node in $V_M \setminus V_u$: hence we perform all the BFSs from each node in $V_M \setminus V_u$, as explained in [IR78] with an overall cost of $O(|V_M| \cdot |V_M \setminus V_u|)$. The number of extended orientations of $M$ is at least $2^{|E_M - E_u|} \geq 2^{|V_M \setminus V_u|}$. Our setup cost is $O(s \cdot |V_M|)$, with $s \leq |V_M|$, and the number of solutions is at least $2^s$. The overall average cost per solution is at most $O(2^s \cdot f(|E_M| + s \cdot |V_M|)/2^s$, which is $O(f(|E_M|) + |E_M| \cdot \frac{2^s}{2^s})$.  

### 7.7 Single source cyclic orientations (SSCO)

For the sake of completeness, we show how the techniques presented can be used to enumerate all the cyclic orientations of an undirected graph $G(V, E)$, whose only source is a given node $s$. Our algorithm is similar to that in Section 7.6 except for the usage of extended orientations, and works as follows.

1. Find a chordless cycle of small size $C(V_C, E_C)$ in $G(V, E) \setminus \{s\}$, called hole, and remove $E_C$ from $E$, obtaining $G'(V, E')$, where $E' = E \setminus E_C$.

2. Find all the orientations $\hat{G}'$ of $G'$, where $s$ is a source and the vertices in $V_C$ can be sources.

We call these orientations special orientations, which can be listed with linear delay.
Algorithm 24: Find the cyclic orientations of $G$ with single source $s$

| Input: An undirected connected graph $G(V, E)$ |
| Output: All the cyclic orientations $\overrightarrow{G}(V, E)$ |

1. **Find and remove a cycle not involving $s$ (Section 7.7.1):**
   - $C(V_C, E_C) \leftarrow$ any hole of $G \setminus \{s\}$ (i.e. a chordless cycle not involving $s$)
   - $G'(V, E') \leftarrow$ delete $C$’s edges from $G$ (i.e. $E' = E \setminus E_C$)

2. **Enumerate special orientations (Section 7.7.2) and put back the hole (Section 7.7.3):**
   - for each special orientation $\overrightarrow{G}'$ of $G'$ do
     - for each legal orientation $\overrightarrow{C}$ of $C$ (see Algorithm 25) do
       - $\overrightarrow{G}(V, E'') \leftarrow$ combine $\overrightarrow{G}'$ and $\overrightarrow{C}$, where $E'' = E' \cup E_C$
     - Output $\overrightarrow{G}$

3. For each special orientation $\overrightarrow{G}'$, exploit all the possible ways of orienting the edges in $E_C$, putting them in $\overrightarrow{G}'$ in a way that the only source is $s$ and there is at least a cycle.

The three steps above are summarized in Algorithm 24 and explained more in detail respectively in Sections 7.7.1, 7.7.2 and 7.7.3.

### 7.7.1 Finding and removing a cycle not involving $s$

Given the graph $G$ in input, we find a hole $C(V_C, E_C)$ not involving $s$ in $G$ (in linear time with a simple DFS after removing $s$) and we remove the edges in $E_C$ from $G$, obtaining $G'(V, E')$, where $E' = E \setminus E_C$. We will denote the length of $C$ as $h$, that is $|V_C| = h$. Note that either $C$ exists, or the graph does not allow any single source cyclic orientation.

### 7.7.2 Enumerating special orientations

Given $G'$, the node $s$ and the vertices $V_C$, we want all the orientations such that $s$ is a source and vertices in $V_C$ can be sources. This can be trivially done modifying the algorithm presented in Section 7.3. In particular, by redefining the notion of **valid direction assignment** as follows.

**Definition 7.8 (valid direction assignment).** Given $\overrightarrow{G}_{i-1}(V_{\leq i-1}, \overrightarrow{E})$, the direction assignment $\overrightarrow{Z}_i$ is **valid if**

- any $v_j \neq s$ and not in $V_C$ that is full in $\overrightarrow{G}_i(V_{\leq i}, \overrightarrow{E} \cup \overrightarrow{Z}_i)$ is not a source, for $1 \leq j \leq i$.

The definition of full node, i.e. **Definition 7.1**, as well the algorithm remains the one of Section 7.3. The fact that $s$ is a source in the final orientation is still guaranteed by the fact that we are using the good order in **Definition 7.3**.

The idea behind the new definition of valid direction assignment is the following. When dealing with $\overrightarrow{G}_i$, since full vertices sources in $\overrightarrow{G}_i$ will be a sources also in any extension of $\overrightarrow{G}_i$, i.e. in the final orientation $\overrightarrow{G}$, we impose that the only vertices that can be full sources are the ones in $V_C$.

Similarly to Section 7.3, the following result can be easily proved.

**Lemma 7.15.** Special orientations can be listed with linear delay.

It is worth observing that the notion of special orientation here replaces the notion of extended orientation in Section 7.6.3.
Algorithm 25: Returning all legal orientations of $C$

| Input: $G' = (V, E')$ cyclic, a cycle $C(V_C, E_C)$ with $V_C \subseteq V$ |
| Output: All the legal orientations $C(V_C, E_C)$ |

1. Build the reachability matrix $R$ for the vertices of $V_C$ in $G'$
2. Let $V_C = \{c_1, \ldots, c_h\}$, where $c_{i+1} = c_i$ by definition and $c_1$ has indegree $> 0$ in $G'$.
3. Execute $\text{LegalOrientations}(C'(0, 0), 1, R)$

```
Procedure $\text{LegalOrientations}(C'(V_C, E_C), j, R)$

if $j = h + 1$ then
  output $C'$
else
  if $R_1$ is cyclic or has positive reachability test on $\{c_{j+1}, \ldots, c_{h+1}\}$ then
    $\text{LegalOrientations}(C'(V_C, E_C), j, R_1)$
  else
    $R_2$ ← $R$ updated adding the arc $(v, u)$
    if $R_2$ is cyclic or has positive reachability test on $\{c_{j+1}, \ldots, c_{h+1}\}$ then
      $\text{LegalOrientations}(C'(V_C, E_C), j, R_2)$
```

7.7.3 Putting back the hole.

As for putting back the hole in Step 3 for each listed $G'$ we have to decide how to put back the edges of the cycle $C$, namely, how to find the orientations of $C$ that create directed cycles with $s$ being the only source. This phase is similar to the one in Section 7.6.4, but it deals with simple orientations of $C$ rather than extended orientations.

Considering this difference, the corresponding definition of legal orientation is the following.

**Definition 7.9.** Given the cycle $C(V_C, E_C)$ and $\tilde{G}'(V, E')$, we call legal orientation $\tilde{C}(V_C, E_C)$ any orientation of $C$ such that the resulting graph $\tilde{G}''(V, \tilde{E})$, where $\tilde{E} = \tilde{E}' \cup \tilde{E}_C$, is cyclic and $s$ is the only source.

In particular, we have to efficiently deal with the following problem: given a cycle $C(V_C, E_C)$ and a special orientation $\tilde{G}'$, enumerate all the legal orientations $\tilde{C}(V_C, E_C)$ of $C$. We remark that for each $G'$ we have at least two legal orientations, namely the clockwise and the counter-clockwise orientations of $C$. In order to also enumerate the other legal orientations of $C$, as in Section 7.6.4, we exploit a reachability matrix $R$ for the vertices in $V_C$ in $G'$. This matrix can be built with a time cost $O(|E'| \cdot h)$. All the steps are shown by Algorithm 25.

Note that deciding the orientation of the edges in $E_C$ is done with a binary partition instead of a ternary partition as in Algorithm 23, since we are dealing with simple orientations of $C$ rather than extended orientations. Hence, for each edge $\{u, v\}$ in $E_C$, we try the two possible directions and update the reachability matrix $R$ to check whether the current partial direction assignment will produce at least one solution. It is worth observing that both the update of $R$ and the dead-end check can be done in $O(h^2)$ (that is, the size of $R$).

Analogously to Lemma 7.11, the following result holds.

**Lemma 7.16.** Algorithm 25 outputs in $O(|E'| \cdot h + h^3)$ time the first legal orientation of $C$, and each of the remaining ones with $O(h^3)$ delay.

As a result, we obtain the following Lemma, whose proof is similar to the one of Lemma 7.13.
Lemma 7.17. Given a cycle of length $h$ in $G$ without $s$, the cyclic orientations of $G(V, E)$ whose only source is $s$ can be enumerated with delay $O(|E| \cdot h + h^3)$ and space $O(|E| + h^2)$.

Let $h$ be the girth of $G$ without $s$; considering that we can apply the algorithm in [IR78] to find a cycle of length $\leq h + 1$ in $G$ without $s$ with cost $O(|V|^2)$, by using Algorithm 24 and applying Lemma 7.17 we can prove Theorem 7.5.

Theorem 7.5. All cyclic orientations of $G(V, E)$ whose only source is $s$ can be enumerated with setup cost $O(|V|^2 + |E| \cdot h + h^3)$, delay $O(|E| \cdot h + h^3)$, where $h$ is the girth of $G \setminus \{s\}$, and space $O(|E| + h^2)$.

We observe that the strategy described in Theorem 7.4 could be also applied to reduce the setup time, at the cost of increasing space usage, in Theorem 7.5.

7.8 Variations of the enumeration problems

We consider here some variations of ssao that could be of independent interest for a given undirected connected graph $G(V, E)$ with $n$ vertices and $m$ edges.

**single source acyclic orientations (weak ssao):** Given a set of vertices $S \subseteq V$, enumerate all the acyclic orientations $G'$ of $G$ such that there is exactly one source $x$ and $x \in S$.

**multiple source acyclic orientations (strong mao):** Given a set of vertices $S \subseteq V$, enumerate all the acyclic orientations $G'$ of $G$ such that all the vertices in $S$ are the only sources.

**multiple source acyclic orientations (weak mao):** Given a set of vertices $S \subseteq V$, enumerate all the acyclic orientations $G'$ of $G$ such that if $x$ is a source then $x \in S$.

We show that these problems can be reduced to ssao. It is easy to see that weak ssao can be solved simply by enumerating all the ssaos in $G$ with source $s$ for each $s \in S$. It is worth observing that for each $s$ there is at least a solution, meaning that the size of $S$ does not influence the delay of weak ssao.

Let us consider weak mao. To solve it, we create a dummy node $s$, and connect it to every node in $S$. More formally, we build $G'(V \cup \{s\}, E \cup E_s)$, where $E_s = \{\{s, x\} : x \in S\}$. Any weak mao of $G$ can be transformed into a ssao of $G'$ if we add $s$ and all edges in $E_s$ (oriented away from $s$): $s$ is a source and all vertices in $S$ are no longer sources since they can be reached from $s$, hence $s$ is the single source. Note that the orientation is still acyclic as $s$ is a source and cannot be part of a cycle. The opposite is true as well: any ssao of $G'$ can be transformed into a weak mao of $G$ by removing $s$ and the edges in $E_s$. This process only removes edges incident to vertices in $S$, hence only vertices in $S$ possibly become sources. Clearly the orientation is still acyclic as removing vertices and edges cannot create cycles.

Finally, consider strong mao. To solve it, we simply collapse all vertices of $S$ into one node $s$. More formally, we generate $G''(V \cup \{s\} \setminus S, E \cup E_s)$, where $E_s = \{\{s, x\} : \exists y \in S \text{ with } \{y, x\} \in E\}$. As $s$ and all vertices in $S$ must be sources, all of their incident edges must be oriented away from them in all acyclic orientations, while the rest of the graph is exactly the same for both cases. Clearly, any ssao for $G''$ induces a strong mao of $G$ that can be obtained by removing $s$ and $E_s$ and re-integrating $S$ and the edges between $S$ and $V \setminus S$ (oriented away from $S$). Similarly, removing $S$ (and the edges between $S$ and $V \setminus S$) and integrating $s$ and $E_s$ (with edges oriented

---

4These orientations are possible if and only if $S$ is an independent set.

5Not all vertices in $S$ must be sources, but there cannot be sources in $V \setminus S$.
away from \( s \), creates a ssao for \( G'' \): there is an edge from \( s \) to any node in \( V \setminus S \) that was previously connected with \( S \), hence these vertices cannot be sources; all other vertices in \( V \setminus S \) were not connected to \( S \) and hence their in-degrees and out-degrees are unchanged.

By Theorem 7.1, observing that the above transformations requires \( O(m) \) time, we can conclude the following result.

**Corollary 7.1.** Problems weak ssao, strong msa, and weak msa can be solved with delay \( O(m \cdot n) \) and space \( O(m) \).

Along the same lines, we can easily solve these variations.

**single source cyclic (weak ssco):** Given a set of vertices \( S \subseteq V \), enumerate all the cyclic orientations \( \overrightarrow{G} \) of \( G \) such that there is exactly one source \( x \) and \( x \in S \).

**multiple source cyclic (strong mscO):** Given a set of vertices \( S \subseteq V \), enumerate all the cyclic orientations \( \overrightarrow{G} \) of \( G \) such that all the vertices in \( S \) are the only sources.

**multiple source cyclic (weak mscO):** Given a set of vertices \( S \subseteq V \), enumerate all the cyclic orientations \( \overrightarrow{G} \) of \( G \) such that if \( x \) is a source then \( x \in S \).

**Part III: Road network reduction and algorithm for strong orientations**

Problem so concerns a special type of cyclic orientations. We first consider the problem of giving feasible orientation to a real road networks, and show how we reduce this problem to that of enumerating sos on a suitable simple graph.

Consider a road network as a connected network \( R \) with \( N \) vertices that correspond to road intersections, and \( L \) edges that correspond to road traits. Of the latter, \( K \leq L \) are tagged as one-way roads whose direction must be decided, whereas the rest are two-way roads taken in both directions. Due to the topology of road network, there may be self-loops and multi-edges. The network has a feasible direction if there is an assignment of direction to each one-way road, so that from every node it is possible to reach all the other ones in the network. We recall that the problem has been named the one-way street problem by [Rob78].

**Problem definition.** This part of the chapter addresses the problem of discovering all the feasible directions in the one-way street problem: we reduce the problem of finding feasible directions in a road network \( R \) to the problem of finding strong orientations of a simple, undirected graph \( G \) with \( n \leq 2K \) vertices and \( m \leq 2K \) edges, where each strong orientation (so afterwards) of \( G \) produces a distinct directed graph that is strongly connected, that is, every node can reach any other node.

**Contribution.** We present the first algorithm for efficiently listing once all the sos in a graph \( G \) with \( m \) edges, with a cost of \( O(m) \) time per solution and using \( O(m) \) preprocessing time and total space. The cost can be deamortized to obtain \( O(m) \) delay with \( O(m^2) \) preprocessing time and space, where the delay is the maximum time elapsed between any two consecutive outputs. Using this result, we are able to find all the feasible directions of the road network in \( O(K) \) amortized time per solution, using \( O(L) \) preprocessing time and total space; also, the cost can be deamortized to obtain \( O(K) \) delay using \( O(K^2 + L) \) preprocessing time and total space.

---

6 These acyclic orientations are possible if and only if \( S \) is an independent set.

7 Not all vertices in \( S \) must be sources, but there cannot be sources in \( V \setminus S \).
Furthermore, our approach easily extends to the enumeration of totally cyclic orientations, which are orientations in which every edge is part of a cycle. On a connected graph, these orientations are exactly the sos [Bol98], otherwise they are combinations of the sos of each component. Note that sos are not related to acyclic and cyclic orientations [CGMR13 [CGMR16], orientations with respectively no cycles or at least one, which require different algorithmic techniques.

7.9 From one-way streets to strong orientations

We show how to list solutions of the one-way street problem for a road network by a reduction to the problem of listing strong orientations on a simple graph. As in [Rob78], we use the notion of mixed graph \( G = (V, E, A) \), i.e. a graph with vertices (in \( V \)) linked by the (undirected) edges in \( E \) and by the (directed) arcs in \( A \). Clearly, both directed and undirected graphs are special cases of mixed graphs, in which \( E = \emptyset \) or \( A = \emptyset \) respectively. Given the mixed graph \( G = (V, E, A) \), we say that node \( x \) reaches node \( y \) if there is a path from \( x \) to \( y \) that uses directed edges in their correct orientation and/or undirected edges. \( G \) is strongly connected if \( u \) reaches \( v \) for every pairs of vertices \( u, v \in V \), and is 2-edge connected if there are two edge-disjoint paths connecting \( u \) and \( v \) for every pair of distinct vertices \( u, v \in V \). We refer to \( G \) as a mixed multigraph when \( E \) or \( A \) are multisets.

Consider a road network \( R \) with \( N \) intersections, \( K \) one-way roads and \( L - K \) two-way roads. We thus model \( R \) as a mixed multigraph \( M = (V_M, E_M, A_M) \) in which every node in \( V_M \) represents a road intersection, \( E_M \) is the multiset of edges corresponding to the one-way roads, and \( A_M \) is the multiset of directed arcs, that contains \( (x, y) \) and \( (y, x) \) for each two-way road linking the intersections modeled by \( x \) and \( y \) (hence, \(|V_M| = N\), \(|E_M| = K\), and \(|A_M| = 2(L - K)\)).

A strong orientation of \( M \) is a direction assignment for the edges in \( E_M \) such that the resulting directed multigraph is strongly connected. Any edge \( \{u, v\} \in E_M \) has two possible orientations \( (u, v) \) and \( (v, u) \), representing how the corresponding road is directed. We consider this to hold for self-loops as well.

It is straightforward to see how a strong orientation of \( M \) corresponds to a feasible way of directing \( R \). We will map strong orientations of the mixed multigraph \( M \) to strong orientations of a suitable graph \( G \).

To this aim, we introduce the following operation on mixed multigraphs:

**Definition 7.10** (contraction of a directed cycle). Given a mixed multigraph \( M = (V_M, E_M, A_M) \) and a set of vertices \( C \subseteq V_M \) which form a directed cycle, the contraction of \( C \) as a node \( c \) modifies \( M \) as follows: \( V_M' = (V_M \setminus C) \cup \{c\} \); for each edge \( e \in E_M \) and each arc \( a \in A_M \), any endpoint of \( e \) and \( a \) in \( C \) is replaced by \( c \); finally, any oriented self-loop on \( c \) created this way is removed.

Note that a contraction can create unoriented selfloops that we preserve along with their endpoints before the contraction.

Lemma 7.18 shows a useful property of the contraction of a directed 2-cycle, while Lemma 7.19 shows how to neglect undirected self-loops as well.

**Lemma 7.18.** Let \( M \) be a mixed multigraph, and \( x, y \) a pair of vertices such that both arcs \( (x, y) \) and \( (y, x) \) exist in \( M \). Let \( M' \) be the mixed multigraph obtained by contracting the directed 2-cycle \( C = \{x, y\} \) as a node \( c \). There is a one-to-one correspondence between the strong orientations of \( M \) and the ones of \( M' \).

**Proof.** Let us show that any so of \( M \) induces a unique so of \( M' \) and vice versa. We remark that all undirected edges of \( M \) are preserved in \( M' \), although some might have become undirected selfloops, thus we have a mapping from each undirected edge of \( M \) to a distinct one of \( M' \). Note
that this gives us a bijective mapping of the orientations of $M$ and $M'$, as each orientation is defined by the direction assignment of the undirected edges. Consider now a strong orientation of $M$: each node can reach/be reached by both $x$ and $y$, thus in the correspondent orientation of $M'$ each node will reach/be reached by $c$ by construction, making $M'$ strongly connected. Similarly a strong orientation of $M'$ induces a strong orientation $M$. Thus we have a one-to-one correspondence between strong orientations of $M$ and $M'$.

**Lemma 7.19.** Let $M'$ be the multigraph obtained by removing all the $k$ unoriented self-loops in the mixed multigraph $M$. Each strong orientation of $M'$ corresponds to $2^k$ unique strong orientations of $M$, and all strong orientations of $M$ can be found this way.

**Proof.** Strong connectivity is not influenced by the removal of self-loops. Thus, removing all self-loop from a strong orientation of $M$ gives us a strong orientation of $M'$. Moreover, given a strong orientation of $M'$, we can obtain $2^k$ unique strong orientations of $M$ by assigning arbitrary orientations to any self-loop (recall that each edge, including self-loops, has two possible orientations). Since two orientations obtained in this way from different orientations of $M'$ are clearly distinct, the statement follows.

Using Lemma 7.18 and Lemma 7.19 we transform $M = (V_M, E_M, A_M)$ in a graph $G = (V, E)$, exploiting the fact that all the arcs in $A_M$ form a set of directed cycles of size 2 by construction. Our transformation proceeds as described next.

1. We contract every directed cycle in $M$ to obtain an undirected multigraph $M'$ according to Lemma 7.18. Note that $M'$ contains only unoriented self-loops.

2. We remove all the self loops in $M'$ according to Lemma 7.19.

3. From the resulting multigraph $M''$ we obtain $G = (V, E)$ as follows: for each edge $\{x, y\}$ in $M''$, we have edges $\{x, z\}$ and $\{z, y\}$ in $E$, where $z$ is a new dummy node, and $V$ is made of the vertices of $M''$ plus the new dummy vertices.

Note that $|V| = n \leq |V_M| + |E_M| = 2K$ and, similarly, $|E| = m \leq 2K$ by construction. We now show that this transformation is correct.

**Lemma 7.20.** Let $G$ be the graph obtained by applying the above transformation to a mixed multigraph $M$ modelling a road network. Each strong orientation of $G$ corresponds to $2^k$ unique strong orientations of $M$, where $k$ is the number of self-loops removed in the transformation. Each strong orientation of $M$ can be obtained this way.

**Proof.** By Lemmas 7.18 and 7.19 we only need to prove that there is a one-to-one correspondence between the strong orientations of $G$ and the ones of $M''$. Let $d_i$ denote the dummy node of $G$ introduced by the transformation when “splitting” $i$-th edge $\{x, y\}$. Given an arbitrary orientation of $M''$, we define an orientation of $G$ in the following way: if $\{x, y\}$ is the orientation of $\{x, y\}$, then the orientations of $\{x, d_i\}$ and $\{y, d_i\}$ are $(x, d_i)$ and $(d_i, y)$. This mapping is clearly injective.

It is now sufficient to prove that any strong orientation of $G$ is induced by a strong orientation of $M''$ and vice versa. Let $u, w$ be two vertices of $G$. We can assume wlog that neither of them is a dummy node: if, say, $u = d_i$ for edge $(x, y)$ of $M''$, then we have edges $(x, u), (u, y)$ in $G$ and we can replace $u$ with $y$. Since $G$ is strongly connected, and only has edges between dummy and non-dummy vertices, there exists a directed path $u = v_1, d_1, \ldots, d_{k-1}, v_k = w$ in $G$, which alternates non-dummy and dummy vertices. By construction of $G$ and the mapping, it follows that $v_1, v_2, \ldots, v_{k-1}, v_k$ is a directed path from $u$ to $v$ in $M''$. For the converse, let $u, w$ be two vertices of $M''$. Since $M''$ is strongly connected, there is a path $u = v_1, \ldots, v_k = w$ in $M''$. By construction, $G$ has the path $u = v_1, d_1, \ldots, d_{k-1}, v_k$.

\[\square\]
7.10 Finding strong orientations (SO)

In this section we show how to efficiently find all the strong orientations (SOs) of an undirected graph \( G = (V, E) \). We assume w.l.o.g. that \( G = (V, E) \) is 2-edge connected: this is a consequence of the following well-known result [Rob39], as otherwise there are no SOs.

**Theorem 7.6** (Robbins’ theorem). A graph \( G \) admits a strong orientation iff it is 2-edge connected.

We introduce the key definitions and properties that will be used to build our algorithm. Using the standard definitions, we call a cut of \( G \) any bipartition \( V_1, V_2 \) of its vertices and we say that an edge \( \{x, y\} \) or an arc \((x, y)\) crosses the cut if \( x \in V_1 \) and \( y \in V_2 \), or vice versa. We define two kinds of cuts which will help us model the problem, namely ONE-WAY CUT and FORCING CUT.

**Definition 7.11** (One-way cut). Given a mixed graph \( G = (V, E, A) \), we call a cut \( V_1, V_2 \) of \( V \) a one-way cut if the cut is crossed only by arcs, which are all oriented towards \( V_1 \) (alternatively, the arcs are all oriented towards \( V_2 \)).

We will also exploit another kind of cut that lets us foresee which orientations of which edges will produce a one-way cut:

**Definition 7.12** (Forcing cut). Given a mixed graph \( G = (V, E, A) \), we call a cut \( V_1, V_2 \) of \( V \) a forcing cut if the cut is crossed by exactly one undirected edge, called bound edge, and by one or more arcs that are all oriented towards \( V_1 \). We call bound direction the one obtained by orienting the bound edge towards \( V_2 \). (The roles of \( V_1 \) and \( V_2 \) can be interchanged.)

Note that we cannot have zero arcs in a forcing cut of \( G \) as otherwise \( G \) would not be 2-edge connected.

**Lemma 7.21.** Let \( G = (V, E, A) \) be a 2-edge connected mixed graph that has no one-way cut. Then any node \( x \) reaches any other node \( y \).

**Proof.** Let us suppose by contradiction that there exist two vertices \( x, y \) such that \( x \) does not reach \( y \). Let \( V_x \) be the set of vertices that are reachable from \( x \). Since \( y \notin V_x \), we have that \( V_x, V \setminus V_x \) is a cut of the graph. Moreover, by its definition there can be no edge going from a node of \( V_x \) to a node of \( V \setminus V_x \), so \( V_x, V \setminus V_x \) is a one-way cut as the graph is connected.

The above lemma together with Theorem 7.8 are crucial to understand the idea behind our approach. For this, we need the following known theorem in [BT80], that extends Robbins’ theorem.
Theorem 7.7 (Boesch and Tindell). A mixed graph $G$ has a SO if and only if $G$ is strongly connected and 2-edge connected.

We say that a mixed graph can be completed or extended to a SO if there exists a direction assignment for its edges such that the resulting digraph is a SO. By ensuring that our partial orientation never admits a ONE-WAY CUT, we can ensure the existence of a strongly connected extension using Boesch and Tindell’s theorem.

**Theorem 7.8.** A 2-edge connected mixed graph $G = (V,E,A)$ can be completed to form a SO iff $G$ does not admit a one-way cut.

**Proof.** If $G$ has a one-way cut $V_1, V_2$, clearly it cannot be extended to a SO. Indeed, as all edges between $V_1, V_2$ are already oriented, the cut will still be a one-way cut in any extension, thus vertices in $V_2$ will not be reachable by vertices in $V_1$.

To prove the other implication, note that by Lemma 7.21 we have that in $G$ any node can reach any other node. Moreover we know by hypothesis that $G$ is 2-edge connected. Boesch and Tindell’s theorem implies that such a graph has a SO, proving our result.

Finally, we show how the concept of FORCING CUT is important for the completion of an orientation as a SO. In particular, Theorem 7.9 extends Lemma 2 in [CGT85].

**Lemma 7.22.** Let $G = (V,E,A)$ be a 2-edge connected mixed graph, and $V_1, V_2$ a cut of $V$. Then $V_1, V_2$ can be turned into a one-way cut by orienting exactly one undirected edge iff $V_1, V_2$ is a forcing cut.

**Proof.** The proof follows immediately from the definitions of one-way cut and forcing cut.

**Theorem 7.9.** Let $G = (V,E,A)$ be a 2-edge connected mixed graph that has no one-way cut, and $\{x,y\}$ an undirected edge in $E$. Then neither of the orientations $(x,y)$ and $(y,x)$ of the edge will create a one-way cut iff $\{x,y\}$ is not a bound edge.

**Proof.** If $\{x,y\}$ is not a bound edge, both orientations lead to a solution. Indeed, any cut crossed by $\{x,y\}$ is not a forcing cut, thus by Lemma 7.22 any orientation of $\{x,y\}$ will not produce a one-way cut. If $\{x,y\}$ is a bound edge, then there is a cut $V_1, V_2$ of $V$ in which all edges are oriented towards $V_1$ except for $\{x,y\}$. Orienting $\{x,y\}$ towards $V_1$ will create a one-way cut.

**7.10.1 Algorithm description**

The above properties are the guidelines for a simple and efficient algorithm to enumerate the SOS of $G$. The core idea hinges on bound edges to guarantee that each recursive call either outputs a new SO or yields two calls that will produce at least one new SO each.

The ideas are detailed in Algorithm 26: it is a recursive approach that consists in incrementally exploring all the possible ways of orienting edges of $G$ that will lead to a solution. In the beginning $G$ is completely undirected, so it will not contain a one-way cut. By Theorem 7.9 we know that the edges that can create a one-way cut are exactly all the bound edges; let $B$ be the set of such edges. Each edge in $B$ must be oriented according to its bound direction, as it would otherwise create a one-way cut. Note that as a consequence of Boesch and Tindell’s theorem [BT80], if there is at least one SO, then the bound direction does not create a one-way cut. For all other edges, we are free to choose any orientation. Thus we orient the edges in $B$ according to their suitable direction, pick an arbitrary edge $\{x,y\}$ (if any), and recur on both
Algorithm 26: Finding all strong orientations (sos)

Input: Graph \( G = (V, E) \).

Output: All sos of \( G \).

Function Strong-Orientations\((V, E, \emptyset)\)

\[
\begin{aligned}
B &\leftarrow \text{bound edges in mixed graph } G = (V, E, A) \\
E &\leftarrow E \setminus B \\
A &\leftarrow A \cup \{(b, c) : (b, c) \text{ is the bound direction of } \{b, c\} \in B\} \\
\text{if } E = \emptyset &\text{ then output } so \leftarrow \overrightarrow{G} = (V, A) \\
\text{else } &\{x, y\} \leftarrow \text{an arbitrary edge in } E \\
E &\leftarrow E \setminus \{\{x, y\}\} \\
\text{Strong-Orientations}(V, E, A \cup \{(x, y)\}) &
\text{Strong-Orientations}(V, E, A \cup \{(y, x)\})
\end{aligned}
\]

possible ways \((x, y)\) and \((y, x)\) of orienting \(\{x, y\}\). When there are no more edges that can be oriented we output the current orientation.

It remains to describe how to find the bound edges in \(B\). In any recursive step, our algorithm starts with a mixed graph \( G = (V, E, A) \), where \(A\) are the edges that have been already directed, and \(E\) the ones that have not. We need to find in this graph all the bound edges in \(E\), that is, all the forcing cuts of \(M\). As we will show in Lemma \ref{lem:strong-bridges}, these are actually all the undirected edges which are strong bridges.

Definition 7.13 (strong bridge). Given a mixed graph \( G \), a strong bridge is an edge that, if removed, increases the number of strongly connected components of \( G \).

Using the algorithm by Italiano et al. \cite{ILS12} we can find all strong bridges in \( G \) in \(O(|E| + |A|)\) time. The algorithm is intended for directed graphs, but it can also be applied to mixed graphs by considering each undirected edge \(\{x, y\}\) as a pair of arcs \((x, y)\), \((y, x)\) with opposite directions, so as to traverse \(\{x, y\}\) in both directions: whichever is chosen between \((x, y)\) and \((y, x)\) as a strong bridge, gives the bound direction to \(\{x, y\}\). (Note that \((x, y)\) and \((y, x)\) cannot be both chosen as strong bridges.)

7.10.2 Correctness

As any edge that is not bound can be oriented in both ways and lead to a solution by Theorem \ref{thm:all-sos}, we observe the following fact.

Lemma 7.23. Let \( e \) be an edge that is not bound in \( G \). Then, orienting any bound edge of \( G \) in its forced direction does not make \( e \) a bound edge.

Proof. It follows from the observation that any cut involving \( e \) has at least two undirected edges, thus orienting the bound edges cannot affect \( e \).

Lemma 7.24. Let \( \{x, y\} \) be an undirected edge in a strongly connected mixed graph \( G \). Then \( \{x, y\} \) is bound iff it is a strong bridge.
Proof. We will first prove that if \( \{ x, y \} \) is a bound edge then it is a strong bridge. Indeed, if \( V_x, V_y \) is the forcing cut of \( \{ x, y \} \), where all other edges go from \( V_x \) to \( V_y \), then removing \( \{ x, y \} \) makes vertices in \( V_y \) unable to reach vertices in \( V_x \), increasing the number of strongly connected components of \( G \), thus \( \{ x, y \} \) is a strong bridge.

Suppose now that \( \{ x, y \} \) is a strong bridge. Let \( V_x \) and \( V_y \) be the set of vertices reachable from respectively \( x \) and \( y \) without using the edge \( \{ x, y \} \). Since \( \{ x, y \} \) is a strong bridge, either \( V_x \neq V \) or \( V_y \neq V \). Let \( V_1 \) be the set, chosen between \( V_x \) and \( V_y \), satisfying the latter disequality. Let \( V_2 = V \setminus V_1 \) be the complement set, which is nonempty, and consider the cut \( V_1, V_2 \): all the arcs in this cut (except \( \{ x, y \} \}) must be oriented towards \( V_1 \), as otherwise \( V_1 \) would be larger.

Hence, \( V_1, V_2 \) is a forcing cut for \( \{ x, y \} \) because \( V_1 \) has no outgoing edges to \( V_2 \) other than \( \{ x, y \} \) itself.

**Theorem 7.10.** Given a 2-edge connected graph \( G = (V, E) \), our algorithm correctly outputs all the strong orientations of \( G \) exactly once.

Proof. A 2-edge connected mixed graph can be completed to form a so iff it does not admit a one-way cut by Theorem 7.8. Hence, we prove by induction on \( |E| \) that, if \( G' = (V, E, A) \) is a mixed graph with no one-way cut, our algorithm outputs all the sos of \( G' \) once.

Base case for \( |E| = 0 \). Then \( G' \) is completely oriented and with no one-way cut, so by Lemma 7.21 it is strongly connected. Moreover it has exactly one so \( G'' = (V, A) \), which we output.

Inductive step for \( |E| > 0 \). We can identify all the bound edges in \( G' \) and their bound directions by Lemma 7.24 using the algorithm in [ILS12]. Orienting bound edges in their bound direction does not alter the set of sos of \( G' \), since there is no so that has a bound edge in the other direction: as each bound edge belongs to a forcing cut, orienting that edge otherwise would create a one-way cut by Lemma 7.22. Also, orienting a bound edge in its bound direction cannot create a new bound edge by Lemma 7.23. We can thus consider \( G' \) as having no bound edges, without loss of generality. If \( G' \) has no more undirected edges, we fall back to the base case. Otherwise, given an undirected edge \( e \) of \( G' \), we know that orienting it either way does not produce any one-way cut by Theorem 7.9. Any so must have \( e \) in either one direction or the other. Let \( G'_1 \) and \( G'_2 \) be the graphs obtained by orienting \( e \) in each way, respectively. Since both \( G'_1 \) and \( G'_2 \) have a smaller number of undirected edges than \( G' \), we know by inductive hypothesis that our algorithm terminates, outputting all the sos of \( G'_1 \) and \( G'_2 \) once. Any so of \( G' \) is a so of either \( G'_1 \) and \( G'_2 \), and the latter have no intersection as they differ on the orientation of \( e \). Hence, the algorithm produces all the sos of \( G' \) once.

7.10.3 Analysis

We now analyze the time and space cost of our algorithm on the graph \( G = (V, E) \), with \( |V| = n \) and \( |E| = m \) assuming wlog that it is connected. We remark that each recursion node which is not a leaf has at least two children, and that every leaf of the computation tree outputs a distinct solution. This gives us a computation tree with no unary vertices and \( \alpha \) leaves, where \( \alpha \) is the number of solutions. It follows that the total number of recursion nodes is bounded by \( 2 \cdot \alpha \) and thus the amortized cost per solution of the algorithm is bounded by the cost of a single recursion node.

Consider the structure of Algorithm 26. We show how every step takes \( O(m) \) time. Computing bound edges is done in \( O(m) \) time by finding the strong bridges and selecting the undirected ones; moreover, the algorithm by Italiano et al. [ILS12] is applied to a directed graph where each

\(^*\)This is crucial, as the presence of unary vertices is the reason behind the \( O(m^2) \) cost of the approach based on [CGTS93], mentioned in the introduction.
undirected edge is represented by two directed arcs, thus finding a strong bridge will immediately
give us the bound direction of the corresponding bound edge, making the assignment of bound
directions clearly $O(m)$ time. All other steps involve updating or scanning sets of size $O(m)$,
which trivially take $O(m)$ time each. The total cost is $O(m \cdot \alpha)$, or equivalently $O(m)$ amortized
cost per solution. We remark that this cost is optimal for merely printing each so.

Finally we show that the space cost is bounded by $O(m)$ as well: indeed, the working space
of a single recursion node is $O(m)$, but the information that needs to be passed on to child
recursive calls, other than the input, is simply the partial orientation of the graph. If stored
as the difference with the partial orientation in the parent node, the space requirement of a
root-to-leaf path (and thus of the whole algorithm) is always $O(m)$. Thus the following holds:

**Theorem 7.11.** Given a 2-edge connected graph $G = (V, E)$, Algorithm 26 outputs all the strong
orientations of $G$ exactly once, in $O(m)$ amortized time, using $O(m)$ total space.

We observe that the delay of Algorithm 26 is bounded by the sum of the costs along a
leaf-to-root path and a root-to-leaf path. Since the cost of each recursion node is $O(m)$, and the
depth of the computation tree is at most $m$, we obtain $O(m^2)$ delay. We will now show how the
delay can be reduced to $O(m)$ using the Output Queue Method by Uno [Uno03], which suitably
accumulates solutions that arrives at an irregular pace to output them in a regular fashion, using
a queue of bounded size. The method depends on two parameters: $T^*$, the maximum cumulative
cost in a root-to-leaf path of the recursion tree, and $\bar{T}$, an upper bound on the amortized cost
per solution in any subtree of the computation. In our case, the former is $O(m^2)$ as discussed
above, and the second is $\Theta(m)$, as each $k$-size subtree of our binary recursion tree has $\Theta(k)$ leaves
(i.e. solutions since there are no unary nodes), and a node takes $O(m)$ time. As a result, using
a queue of $O(T^*/\bar{T}) = O(m)$ solutions, we can output each solution with delay $O(\bar{T}) = O(m)$.
This takes $O(T^* + \bar{T}) = O(m^2)$ preprocessing time and $O(m \cdot T^*/\bar{T}) = O(m^2)$ space.

**Theorem 7.12.** Given a 2-edge connected graph $G = (V, E)$, there exists an algorithm that
outputs all the strong orientations of $G$ exactly once, with $O(m)$ delay, using $O(m^2)$ preprocessing
time and total space.

### 7.11 Final remarks

In this chapter we have shown a collection of algorithms for enumerating graph orientations
with constraints regarding sources, cycles, and reachability. For the problem of AO we improve
previous work by reducing the delay to $O(m)$; for all the other problems considered (namely
SSO, SSSAO, SSSCO, CO and SO) we show the first algorithms with guaranteed delay, characterized
with recursion trees (see Section 2.3.2) which are ugly (for SSO, SSSAO, SSSCO) and good (for AO, CO
and SO). We also show, for SO, how the algorithm can be used to list feasible orientation of a
realistic road network, which is best represented as a multigraph rather than a simple graph, and
may have roads which are naturally two-way. The algorithms also solve some generalizations,
as discussed in Section 7.8 but the problem posed by Squire [Squ98], of efficiently enumerating
acyclic orientations with sources and targets lying within two arbitrary sets $S$ and $T$, remains
open.
The enumeration of dense subgraphs, such as cliques and quasi-cliques, has been the object of a large number of studies due, for example, to their connections to community structures. The opposite problem, enumerating sparse subgraphs, has also been the object of study: some example are independent sets and trees. However, this problem has not received as much attention as its dense counterpart. In this chapter, we consider the problem of listing the maximal $k$-degenerate induced subgraphs of a chordal graph. The degeneracy is a well known measure of sparsity, thus $k$-degenerate subgraphs are a plausible model of sparse subgraphs. We propose a P-Delay algorithm with delay $O(m \cdot q)$ for an $n$-vertex chordal graph with $m$ edges, where $q \leq n$ is the maximum size of a clique in $G$. The problem generalizes that of enumerating maximal independent sets and maximal induced forests, which correspond to respectively 0-degenerate and 1-degenerate subgraphs. We also prove that the corresponding Extension problem is NP-complete: the Extension problem is a common building block of enumeration algorithms which, given subsets $X$ and $Y$ of $V(G)$, asks whether there is a maximal induced $k$-degenerate subgraph $H$ of $G$ that contains all vertices of $X$ but none of $Y$.

8.1 Introduction

Dense subgraphs are object of extensive research, especially due to their close relationship to community detection; however, one may be interested in finding sparse graphs as many networks are sparse even if locally dense. For instance, the paper [WAU14] addresses the enumeration of induced trees in $k$-degenerate graphs, and several address that of independent sets (e.g., [TIAS77]).

The degeneracy of a graph is the smallest integer $k$ for which every subgraph of the graph has a vertex of degree at most $k$. A graph is said to be $k$-degenerate if its degeneracy is $k$ or less. Degeneracy is also referred to as the coloring number or $k$-core number, as a $k$-degenerate graph may contain a $k$-core but not a $k+1$-core, and is a widely used sparsity measure [UKB11, ELS13, CGMV16, WAU14, NOdM12]. Several studies tend to take into account the degeneracy of graphs, as it tends to be very small in real-world networks [UKB11]. Many important graph classes in structural graph theory are degenerate [NOdM12]. Furthermore, it is straightforward to see that $k$-degenerate subgraphs generalize well known structures, as 0-degenerate subgraphs correspond to independent sets, while 1-degenerate subgraphs correspond to induced forests. We thus take degeneracy as our sparsity measure of choice, and consider the enumeration of $k$-degenerate induced subgraphs.

Alon et al. [AKSS7] investigated the size of the largest $k$-degenerate induced subgraph in a graph, giving tight lower bounds in relation to the degree sequence of the graph. Whilst Pilipczuk et al. [PP12] showed that a maximum $k$-degenerate induced subgraph can be found in
randomized time $O((2 - \epsilon_k)^n n^{O(1)})$, for some $\epsilon_k > 0$ depending only on $k$, and moreover showed that there are at most $(2 - \epsilon_k)^n$ such subgraphs. See [BKW10, LMZ15] for other recent studies on degeneracy.

In this chapter we provide an efficient polynomial delay algorithm for listing $k$-degenerate induced subgraphs in input chordal graphs.

Chordal graphs (also known as triangulated graphs) have been a topic of intensive study in computer science due to the applications in phylogenetic networks and also many NP-complete problems become tractable when the inputs are chordal graphs [BP93, Gav74, EK11, RTL76, TY84]. A graph is chordal if and only if every cycle of length 4 or more has a chord, i.e. an edge joining two non-consecutive vertices. Chordal graphs have been equivalently characterized in different ways: they are the graphs that allow a perfect elimination ordering, that is an elimination ordering in which every eliminated vertex is simplicial (its neighbors form a clique) [RTL76, TY84]; the graphs that allow a clique tree [BP93] (see Section 8.2.1); the intersection graphs of subtree families in trees [Gav74]. In our case, we will consider the characterization by clique-trees. It is well-known that $n$-vertex chordal graphs have at most $n$ maximal cliques. A clique-tree of a chordal graph $G$ is a tree $T$ whose nodes are in bijection with the set of maximal cliques, and such that for each vertex $x$ the set of maximal cliques containing $x$ form a subtree of $T$.

Our algorithm is based on the binary partition technique, which involves solving an Extension Problem, and exploits the structure given by the clique-tree of the input graph. The enumeration can be reduced to the following question: Given two subsets of vertices $S$ and $X$, decide whether there is a maximal $k$-degenerate induced subgraph which contains $S$ and does not intersect $X$. As remarked in Section 2.3.3, if we can answer this question in polynomial time, the algorithm can be summarized as follows: start from the empty set, and in each iteration with given sets $(S, X)$ pick a vertex $v$ and partition the problem into those containing $v$ (a call to the iteration $(S \cup \{v\}, X)$) or those not containing $v$ (a call to the iteration $(S, X \cup \{v\}$), both calls depending on the answer given by the Extension problem. The delay of such algorithms is usually $O(n \cdot \text{poly}(n))$ with poly(n) being the time to decide the Extension problem. As we will show in Section 8.3, however, this problem is NP-complete for generic graphs, and even for split graphs, which are a subset of chordal graphs. We thus need some additional constraints on the inputs of the Extension problem. For our algorithm we do not consider all possible sets $(S, X)$ for the Extension problem, but only some special cases driven by the clique-tree. Our special case of the Extension problem is the following (we consider the clique-tree $T$ to be rooted):

**Input.** A node $C$ of $T$, a partition $(S, X)$ of the set of vertices in all the cliques preceding $C$ in a pre-order traversal of $T$ and a partition $(S', X')$ of $C \setminus (S \cup X)$.

**Output.** Decide whether there is a maximal solution containing $S \cup S'$ and avoiding $X \cup X'$.

We propose a notion of greedy solution and show that this special case of the Extension problem is a Yes-instance if and only if a greedy solution exists; we also propose an $O(m)$-time algorithm to compute the greedy solution. The final result is a P-Delay binary partition algorithm whose recursion tree has the form of a good tree (as defined in Section 2.3).

### 8.2 Preliminaries

Refer to Section 2.1 for terminology. In this chapter, we assume that graphs are simple, undirected, and given with a linear ordering of its vertices. We can further assume graphs to be connected as the solutions of a non-connected graph are obtained by combining those of its connected components.

Recalling that $N_G(x)$ is the neighborhood of vertex $x$, we defined the closed neighborhood of $x$ as $N_G[x] = N_G(x) \cup \{x\}$, and let $d_G(x) = |N(x)|$ denote the degree of $x$. 

146
We denote by \( \mathcal{Q}(G) \) the set of maximal cliques of \( G \), and by \( q \) the maximum number of vertices in a clique in \( \mathcal{Q}(G) \). For a vertex \( x \), we denote by \( \mathcal{Q}(G,x) \) the set of maximal cliques containing \( x \).

For a rooted tree \( T \) and two nodes \( u \) and \( v \) of \( T \), we call \( v \) an ancestor of \( u \), and \( u \) a descendant of \( v \), if \( v \) is on the unique path from the root to \( u \); \( u \) and \( v \) are incomparable if \( v \) is neither an ancestor nor a descendant of \( u \).

### 8.2.1 Chordal graphs and clique trees

A graph \( G \) is a chordal graph if it does not contain an induced cycle of length more than three. It is well-known that a chordal graph \( G \) has at most \( n \) maximal cliques, and they can be enumerated in linear time \cite{Chalermsook01}. With every chordal graph \( G \), one can associate a tree that we denote by \( \mathcal{QT}(G) \), called a clique tree, whose nodes are the maximal cliques of \( G \) and such that for every vertex \( x \in V(G) \) the set \( \mathcal{Q}(G,x) \) is a subtree of \( \mathcal{QT}(G) \) \cite{Gavril74}. Moreover, for every chordal graph \( G \), one can compute a clique tree in linear time (see for instance \cite{Gavril95}). In the rest of the chapter all clique trees are considered rooted.

In what follows, for a maximal clique \( C \) of a chordal graph \( G \), \( C \) depending on the context, may refer to its set of vertices, the subgraph induced by \( C \), or the corresponding node in the clique tree.

### 8.2.2 \( k \)-degenerate graphs

Recalling the definition of degeneracy from Section 2.5.3, every \( k \)-degenerate induced subgraph of \( G \) corresponds to a set of vertices \( X \) for which \( G[X] \) is a \( k \)-degenerate graph.

**Problem 8.1.** Given a chordal graph \( G \) and a positive integer \( k \), enumerate all maximal \( k \)-degenerate induced subgraphs in \( G \), with polynomial delay.

Note that a complete graph \( K_n \) is an \((n-1)\)-degenerate graph, as all its vertices have degree \( n-1 \). Therefore, for any clique \( C \) of a graph \( G \), any \( k \)-degenerate induced subgraph of \( G \) may have no more than \( k+1 \) vertices belonging to \( C \). Chordal graphs have the following property.

**Theorem 8.1.** The degeneracy of a chordal graph is exactly \( q-1 \).

**Proof.** Since the degeneracy is a hereditary property (i.e., any subgraph of a \( k \)-degenerate graph is \( k \)-degenerate), and the complete graph \( K_n \) has degeneracy \( n-1 \), \( q-1 \) is a lower bound for the degeneracy of any graph. The fact that \( q-1 \) is an upper bound on chordal graphs relies on the fact that every chordal graph has at least a vertex whose neighbor is a clique \cite{Rosenfeld76}. Therefore, in any chordal graph we can find a vertex of degree at most \( q-1 \). \( \square \)

### 8.3 Hardness of the extension problem

In this section we consider the Extension problem for maximal \( k \)-degenerate induced subgraphs, which corresponds to answering the following question:

**Problem 8.2** (Extension problem for maximal \( k \)-degenerate induced subgraphs). Given a graph \( G \), and two sets of vertices \( S \subseteq V(G) \) and \( X \subseteq V(G) \) such that \( S \cap X = \emptyset \) and \( G[S] \) is \( k \)-degenerate, is there a maximal \( k \)-degenerate induced subgraph \( M \) such that \( S \subseteq M \) and \( X \cap M = \emptyset \)?
Assuming that \( S \) is \( k \)-degenerate is without loss of generality: indeed if \( S \) is not \( k \)-degenerate then the answer to the problem is always negative, as any subgraph containing \( S \) may not have smaller degeneracy than \( S \).

For brevity we reuse the notation in Problem 8.2 in the remainder of the section, so as not to similarly redefine \( G, S, X \) and \( M \) multiple times. Furthermore, we use simply “Extension problem” as a shorthand for “Extension problem for maximal \( k \)-degenerate induced subgraphs”.

As we mentioned in Section 8.1, if we could answer this question in polynomial time, say, \( p(|V(G)|) \), then we could use this as a black box to create a binary partition algorithm with polynomial delay, as it is straightforward to see that its delay would be bounded by \( O(|V(G)| \cdot p(|V(G)|)) \).

In the following, however, we show that this problem is \( \text{NP} \)-complete, even when \( G \) is a split graph. A graph \( G \) is a split graph if its vertices can be partitioned into a clique and an independent set. It has long been known that this is equivalent to saying that both \( G \) and its complement \( \bar{G} \) (i.e., the graph on the same vertices as \( G \) that has an edge iff the edge is not in \( G \)) are chordal \([FH77]\).

As split graphs are chordal, this hardness result also applies to chordal graphs, and, of course, to generic graphs.

### 8.3.1 Polynomiality of restricted cases

First, we consider two restricted versions of the Extension problem which actually can be solved in polynomial time.

**Lemma 8.1.** For any graph \( G \), if \( X = \emptyset \) then the answer to the Extension problem is always yes.

**Proof.** As \( X \) is empty, \( M \) may not contain vertices of \( X \), thus the question is simply whether there exists a maximal \( k \)-degenerate induced subgraph \( M \) which contains \( S \). As mentioned in the proof of Theorem 8.1 any subgraph of a \( k \)-degenerate graph is itself \( k \)-degenerate. This means that \( M \) may be found in a greedy fashion by initially setting \( M = S \), and adding arbitrary vertices to it for which \( M \) is still \( k \)-degenerate. When no such vertex can be found, then the \( M \) is a maximal \( k \)-degenerate induced subgraph which contains all vertices in \( S \) but none of those in \( X \) (which is empty). \( \square \)

More in general, the above lemma can be applied to the Extension problem of not just maximal \( k \)-degenerate subgraphs, but to that of maximal subgraphs for all hereditary properties.

In order to prove the polynomiality for the following case, however, we need to restrict our attention to split graphs.

**Proposition 1.** Let \( G \) be a split graph, with vertices partitioned in a clique \( C \) and an independent set \( I \). If \( I \cap S = \emptyset \) and \( I \cap X \neq \emptyset \), then the answer to the Extension problem can be found in polynomial time. Furthermore, if \( S \cup X \subset C \), and \( |C \setminus X| \geq k + 1 \) then the answer to the Extension problem is yes.

**Proof.** Firstly, recall that \( k \)-degenerate subgraphs may contain cliques with up to \( k + 1 \) vertices, that induced subgraphs of chordal graphs are chordal, and by Theorem 8.1 that chordal graphs have degeneracy \( q - 1 \). Thus a subgraph of \( G \) is \( k \)-degenerate if and only if its largest clique is of size at most \( k + 1 \). Note that every vertex in \( I \) is simplicial i.e., its neighbors form a clique) as its neighbors are a subset of the clique \( C \).

**Case 1:** \( I \cap S = \emptyset \) and \( I \cap X \neq \emptyset \). First notice that, if a \( k \)-degenerate induced subgraph \( M \) not intersecting \( X \) is maximal, then \( |N(x) \cap V(M)| = k + 1 \) for each \( x \in (I \cap X) \), otherwise \( G[V(M) \cup \{x\}] \) is still \( k \)-degenerate as the largest clique in \( G[V(M) \cup \{x\}] \) would be still \( k + 1 \),
contradicting the maximality of $M$. We can thus conclude that $|M \cap N(x)| = k + 1$ for each $x \in I \cap X$ is a necessary condition if $I \cap X$ is not empty, independently from $S$. We now show that this is also sufficient and a suitable $M$ can be found in polynomial time, if any exists.

Let $C' = \bigcap_{x \in I \cap X} N(x)$, and note that $C' \subseteq C$. Recall also that a maximal $k$-degenerate induced subgraph of $G$ can contain at most $k + 1$ vertices from $C'$ since $C$ is a clique. Moreover, if $M$ is a maximal $k$-degenerate induced subgraph of $G$, then $V(M) \cap C \subseteq C'$, otherwise there is a vertex $x \in I \cap X$ such that $|N(x) \cap (V(M) \cap C)| \leq k$, i.e., $G[V(M) \cup \{x\}]$ is also $k$-degenerate, contradicting the maximality of $M$. Therefore, if $|C' \setminus X| < k + 1$ or $(C \setminus C') \cap S \neq \emptyset$, then the answer to the Extension problem is no. Assuming $|C' \setminus X| \geq k + 1$ and $(C \setminus C') \cap S = \emptyset$, let $C_M$ be an arbitrary subset of $C' \setminus X$ of size $k + 1$, and $I_M$ be the set of all vertices $v$ in $I \setminus X$ such that $|N(v) \cap C_M| < k + 1$, and let $M = G[C_M \cup I_M]$. Clearly, all the vertices $w$ in $I \setminus I_M$ may not be added to $M$ as it would form a clique of size $k + 2$ with $C_M$. Hence, $M$ is a maximal $k$-degenerate induced subgraph of $G$, which means the answer to the Extension problem is yes. Since constructing $C'$, $C_M$ and $I_M$ can all be done in polynomial time, we are done.

**Case 2:** $S \cup X \subseteq C$, and $|C \setminus X| \geq k + 1$. Let $C_M \subseteq C \setminus X$ be a set of exactly $k + 1$ containing $S$ (we know that $C_M$ can be chosen as such from the assumption that $|C \setminus X| \geq k + 1$). Furthermore, let $I_M$ be the set of all vertices $v \in I$ such that $|N(v) \cap C_M| < k + 1$. We claim that $M := G[C_M \cup I_M]$ is a maximal $k$-degenerate induced subgraph of $G$. Indeed, as $|C_M| = k + 1$, no vertex from $C$ can be added to $M$, otherwise $M$ would have a clique of size $k + 2$. Also, any vertex $w$ in $I \setminus I_M$ is such that $C_M \subseteq N(w)$, i.e., $G[C_M \cup \{w\}]$ is a clique of size $k + 2$. Thus $M$ is a maximal $k$-degenerate induced subgraph of $G$ and the answer to the Extension problem is yes.

**Corollary 8.1.** For any split graph $G$ whose vertices are partitioned into a clique $C$ and an independent set $I$, a maximal $k$-degenerate induced subgraph $M$ of $G$ satisfies $|N(x) \cap V(M)| = k + 1$ for all $x \in I \setminus V(M)$.

### 8.3.2 Hardness proof

In the following we show that, for arbitrary subsets $S$ and $X$ of vertices, the Extension problem is NP-complete even on split graphs.

In order to do so, we will use a polynomial-time reduction from an instance of the Exact Hitting Set problem, whose definition we recall below, to that of solving the Extension problem on a split graph $G$.

**Problem 8.3** (Exact Hitting Set). Given a set of elements $U$, called universe, and a family $W$ of subsets of $U$ whose union equals $U$, the problem asks whether there exists a subset $T \subseteq U$ of size $h$ such that for each $W \in W$, $|W \cap T| \geq 1$.

The Exact Hitting Set problem is a well known NP-complete problem, and is equivalent to set cover, one of Karp’s 21 NP-complete problems. The problem is also reported in [GJ90], which shows that it is still NP-complete even if $W$ is constrained so that every $W \in W$ has at most two elements.

**Theorem 8.2.** The answer to the Extension problem for maximal $k$-degenerate induced subgraphs (Problem 8.3) cannot be found in polynomial time on arbitrary input split graphs unless $P = NP$.

**Proof.** We prove this theorem by reducing an arbitrary instance of the Exact Hitting Set problem to the Extension problem on a split graph. In the following we consider an instance $(U, W, h)$ of the Exact Hitting Set problem, referring to the notation in Problem 8.3. For $W \in W$, let $\overline{W}$ be $U \setminus W$.
Building the instance. Let us build a suitable $G$ for our instance of the Extension problem. Let $V$ be a set disjoint from $U$ and in bijection with $W$, and let $v_x$ be a vertex disjoint from $U \cup V$. Let $G$ be a graph with vertex set $V \cup U \cup \{v_x\}$ such that $U \cup \{v_x\}$ is a clique in $G$, $V$ is an independent set in $G$, and all the other edges are $uv$, for $u \in U$ and $v \in V$, whenever $u \notin W_v$, with $W_v$ the set in $W$ in correspondence with $v$. $G$ is clearly a split graph with its vertex set partitioned into a clique $U$ and an independent set $I = V \cup \{v_x\}$, and its size is polynomial in $|U| + |W|$, that is the size of the Exact Hitting Set problem. Let $S = V$, $X = \{v_x\}$ and $k = h - 1$. We claim that the Extension problem with instance $(G, S, X)$ is yes if and only if the Exact Hitting Set problem is yes. For brevity, let us denote $W_v$ the set in $W$ in correspondence with $v \in V$.

Equivalence. Let $M$ be a maximal $(h - 1)$-degenerate induced subgraph of $G$. Recalling Corollary [8,1] since $v_x \in I \cap X$, then $|V(M) \cap U| = h$, i.e., $M$ must contain exactly $h$ vertices from $U$. Moreover, $|N(v) \cap (V(M) \cap U)| < h$ for each $v \in V = I \cap S$. Since, for $u \in U$ and $v \in V$, $uv \in E(G)$ only when $u \notin W_v$, we can conclude that $W \cap (V(M) \cap U) \neq \emptyset$ for all $W \in W$, and thus the answer to the Exact Hitting Set problem is yes.

Conversely, let $T$ be a solution to the Exact Hitting Set problem. We claim that $M = G[T \cup V]$ is a maximal $(h - 1)$-degenerate induced subgraph of $G$. Indeed, since $U$ is a clique, $N(u) \cap T = h$ for each $u \in (U \setminus T) \cup \{v_x\}$, i.e., $G[V(M) \cup \{u\}]$ is not $(h - 1)$-degenerate. Since, each set $W_v$ intersects $T$, by the definition of the edge set of $G$, there is a vertex in $T$ that is not adjacent to $v$ in $G$, i.e., the maximal clique size of $M$ is $h$, which concludes the proof.

As mentioned above, this hardness result extends to chordal graphs, as split graphs are chordal, and of course to general graphs. Furthermore, this is a somewhat surprising result if compared to our proposed algorithm $k\text{MIG}$, as the algorithm itself uses a restricted form of the Extension problem, which can be solved in polynomial time.

8.4 Enumeration algorithm

This section describes our algorithm for enumerating all maximal $k$-degenerate induced subgraphs of a given chordal graph $G = (V, E)$. In the following, we sometimes refer to maximal $k$-degenerate induced subgraphs as solutions, and we denote them by their vertex set as for cliques, to ease the reading.

Our proposed algorithm is based on the binary partition method. The outline of our algorithm is as follows. We start with an empty induced subgraph $S$. Then we pick a vertex $v$ from $G$ and add $v$ to $S$. If $S + v$ is a maximal $k$-degenerate induced subgraph, then we output $S + v$, otherwise we choose another vertex and add it to $S + v$. After that we backtrack and add $v$ to an excluded set $X$, to generate all solutions that contain $S$ and not $v$. By recursively applying the above operation to $G$ we can enumerate all solutions. However, certain pairs $(S, X)$ may not generate a solution, as there may be no maximal $k$-degenerate induced subgraph containing $S$ but no vertex in $X$ (e.g., if $S = \emptyset, X = V$). If we test all the possibilities that will not lead to a solution, the cost of this process is not output sensitive, i.e., not bounded by a polynomial in the number of solutions. To develop an efficient enumeration algorithm, we have to limit such redundant testing as much as possible by answering instances of the Extension problem. But, this later problem is $\text{NP}$-complete. To overcome this difficulty, we focus on the rooted clique tree $QT(G)$ to restrict the instances of the Extension problem to tractable cases. For doing so, we introduce the concepts of greedy filling and partial solution. In what follows we let $G$ be a fixed chordal graph.
8.4.1 Greedy filling strategy

Let \( R \) be a fixed maximal clique, called the root of \( QT(G) \), and let us root \( QT(G) \) at \( R \). For a maximal clique \( C \) of \( G \), whose parent in \( QT(G) \) is the clique \( P \), we call private vertices of \( C \), denoted by \( P v(C) \), the set of vertices in \( C \setminus P \). Because all cliques in \( QT(G) \) are different and inclusion-maximal, and by the properties of the clique tree, one can deduce the following.

Lemma 8.2. Given a clique tree \( QT(G) \), every clique in \( QT(G) \) contains at least one private vertex, and every vertex \( v \) is private in exactly one clique in \( QT(G) \).

Let \( C \) be a maximal clique of \( G \). For \( X \subseteq V(G) \), let \( A(C, X) = 1 \) if \( |C \setminus X| \geq k + 1 \), and \( A(C, X) = 0 \) otherwise. For any vertex \( v \in X \), let \( A(v, X) = \sum_{C \in QT(G)} A(C, X) \), i.e., the number of maximal cliques containing \( v \) for which \( |C \setminus X| \geq k + 1 \). As adding more than \( k + 1 \) vertices from the same clique to any solution \( M \) would cause \( M \) to not be \( k \)-degenerate anymore, we say that \( C \) is saturated in \( M \) if \( |C \cap M| = k + 1 \).

The function \( A \) allows us to check the maximality of a \( k \)-degenerate induced subgraph, thanks to the following lemma.

Lemma 8.3. Let \( G = (V, E) \) be a chordal graph and \( M \subseteq V \) be a \( k \)-degenerate induced subgraph of \( G \), with \( X = V \setminus M \). Then, \( M \) is maximal if and only if \( A(x, X) \geq 1 \) for each \( x \in X \).

Proof. Assume \( A(x, X) \geq 1 \) for each \( x \in X \) and there exists a \( k \)-degenerate induced subgraph \( M' \supset M \), with \( v \in M' \setminus M \). As \( v \in X \) we have \( A(v, X) \geq 1 \), thus there exists a clique \( C \) containing \( v \) s.t. \( |C \setminus X| \geq k + 1 \). As \( M = V \setminus X \), we have \( |C \setminus X| = |C \cap M| \geq k + 1 \). As \( M \cup \{v\} \subseteq M' \) we have \( |C \cap M'| \geq k + 2 \), thus \( M' \) contains a complete subgraph with \( k + 2 \) vertices and is not \( k \)-degenerate, which contradicts the hypothesis.

On the other hand, if for a vertex \( x \in X \) we have \( A(x, X) = 0 \), then for any clique \( C \) containing \( x \) we have \(|(M \cup \{x\}) \cap C| \leq k + 1 \), since \( |C \setminus X| = |C \cap M| < k + 1 \). Thus the largest clique in \( M \cup \{x\} \) has size at most \( k + 1 \), and as \( M \cup \{x\} \) is a chordal graph (it is an induced subgraph of \( G \)) it is \( k \)-degenerate by Theorem 8.1. Thus \( M \) is not maximal, which contradicts the hypothesis.

We now define the notion of partial solution as a pair of disjoint vertex subsets \((S, X)\), where \( S \) contains vertices (to include) in the \( k \)-degenerate induced subgraph, and \( X \) is a set of vertices that must be excluded from the solution, with some additional properties:

Definition 8.1 (partial solution). A pair \((S, X)\) of subsets of \( V(G) \) with \( S \cap X = \emptyset \) is a partial solution if

1. \(|S \cap C| \leq k + 1 \) for any maximal clique \( C \),
2. \( A(x, X) \geq 1 \) for each \( x \in X \),
3. for each maximal clique \( C \), if \( P v(C) \cap (S \cup X) \neq \emptyset \), then \( C' \subseteq S \cup X \) for all ancestors \( C' \) of \( C \).

Given a pair \((S, X)\) of disjoint subsets of \( V(G) \), it is not trivial to decide whether there exists a solution \( M \supseteq S \) with \( M \cap X = \emptyset \) (see Section 8.3.2). However, as we will later demonstrate, this is always true if \((S, X)\) is a partial solution. Next, we introduce the strategy that will be used by our algorithm to guarantee the existence of solutions. Let \( \pi : \{1, \ldots, |Q(G)|\} \to Q(G) \) be a fixed linear ordering of \( Q(G) \) obtained from a pre-order traversal of \( QT(G) \), and let us call \( \pi^{-1}(C) \) the rank of \( C \in Q(G) \). We use the rank of the cliques to define the order in which they are considered by the following procedure.
Definition 8.2 (Greedy filling). The greedy filling of a partial solution \((S, X)\) consists in the following. Let \(C\) be the maximal clique with the smallest rank for which \(C \setminus (S \cup X) \neq \emptyset\). Add vertices one by one from \(C\) to \(S\) until \(C\) is saturated for \(S\) or \(C \setminus (S \cup X) = \emptyset\). Then add the remaining vertices in \(C \setminus (S \cup X)\) to \(X\), if any, and repeat the process until no such clique \(C\) exists.

Finally, we can now show that a partial solution can always be extended into a maximal one by means of a greedy filling.

Lemma 8.4. For any partial solution \((S, X)\), the greedy filling yields a maximal \(k\)-degenerate induced subgraph \(M\) of \(G\) such that \(S \subseteq M\) and \(M \cap X = \emptyset\).

Proof. Let \(M\) be the greedy filling of \((S, X)\). By definition, \(S \subseteq M\) and \(X \cap M = \emptyset\).

We prove the statement by showing that at all times during the greedy filling \((S, X)\) maintains the property of being a partial solution (see Definition 8.1), so in the end we have \(A(x, V \setminus M) \geq 1\) for each \(x \in V \setminus M\), making \(M\) a maximal \(k\)-degenerate induced subgraph by Lemma 8.3. Let \(Q\) be the maximal clique of the smallest rank for which \(Q \setminus (S \cup X) \neq \emptyset\). Let \((S', X')\) be the new pair constructed from \(Q\) by the greedy filling, and let \((S_Q, X_Q)\) be the partition of \(Q \setminus (S \cup X)\) such that \(S' = S \cup S_Q\) and \(X' = X \cup X_Q\). First notice that for all the ancestors \(Q'\) of \(Q\) we have \(Q' \setminus (S \cup X) = \emptyset\) as their rank is smaller than the one of \(Q\).

By definition of greedy filling, \(|Q \cap S'| = |(Q \cap S) \cup S_Q| \leq k + 1\). If \(X_Q = \emptyset\), then \(X' = X\) and \(A(x, X') = A(x, X) \geq 1\) for each \(x \in X'\). Otherwise, by definition of greedy filling, \(Q\) is saturated in \(S'\) (\(|Q \cap S'| = k + 1\)). Hence, \(A(Q, X') = 1\), and for each \(x \in Q\) \(A(x, X') \geq 1\), while for each \(x \in X' \setminus Q = X \setminus Q\) \(A(x, X') = A(x, X) \geq 1\). Thus, \((S', X')\) is a partial solution, which completes the proof. \(\square\)

8.4.2 Binary partition method

We are now ready to describe our algorithm \(k \text{MIG}(G, k)\), whose pseudo-code is given in Algorithm 27.

The principle is to start from the partial solution \(S = \emptyset, X = \emptyset\), where \(S\) represent the vertices that will be in the solution, and \(X\) the vertices that are excluded from the solution, and proceed with binary partition: in each recursive call we consider a vertex \(v \in Q\), initially from the clique \(Q\) with the smallest rank, i.e., the root of \(QT(G)\); we will first add \(v\) to \(S\) and find all the solutions containing \(S \cup \{v\}\) and nothing in \(X\); then add \(v\) to \(X\) and find all the solutions containing \(S\) and nothing in \(X \cup \{v\}\), if any exists. At any step, we keep the invariant that \((S, X)\) is a partial solution: If we add \(v\) to \(S\) (Line 12), this is equivalent to performing a step of the greedy filling, thus we know that \((S \cup \{v\}, X)\) is still a partial solution (see proof of Lemma 8.4). When, on the other hand, we try to add \(v\) to \(X\) (Line 14), we only explore this road if there exists a solution that contains all the vertices in \(S\) and no vertex in \(X \cup \{v\}\). Thanks to \((S, X)\) being a partial solution we will be able to discover this efficiently, and we will demonstrate (Lemma 8.5 in Section 8.4.3) that this is true if and only if \((S, X \cup \{v\})\) is still a partial solution. Only once \(Q \setminus (S \cup X)\) is empty, we then proceed to the clique \(Q'\) next in the ranking (Lines 16-17). This guarantees that \(Q\) is always the clique of smallest rank such that \(Q \setminus (S \cup X) \neq \emptyset\), thus Condition 3 of Definition 8.1 still holds, and so \((S, X)\) is still a partial solution. It is important to remark that, as all ancestors of \(Q\) are fully contained in \(S \cup X\), and \(v \notin S \cup X\), then \(v\) is always a private vertex of \(Q\), not contained in the ancestors of \(Q\).

Finally, if \(S \cup X = V\) we can output \(S\) as a solution: by keeping the invariant that \((S, X)\) is a partial solution, we know by Lemma 8.3 that \(S\) is a maximal \(k\)-degenerate induced subgraph of \(G\).
Algorithm 27: \( k \text{MIG} \): Enumerating all maximal \( k \)-degenerate induced subgraphs in a chordal graph \( G = (V,E) \)

\[
\begin{align*}
1 & \textbf{Procedure } k \text{MIG}(G, k) \\
2 & \quad \text{Compute } QT(G) \text{ of } G; \\
3 & \quad R \leftarrow \text{the root clique of } QT(G); \\
4 & \quad \pi : \{1, \ldots, |Q(G)|\} \rightarrow Q(G) \text{ be a pre-order traversal of } QT(G); \\
5 & \quad \text{Call } Subk \text{MIG}(G, R, \emptyset, \emptyset, k); \\
6 & \textbf{Procedure } Subk \text{MIG}(G, Q, S, X, k) \\
7 & \quad \text{if } V = S \cup X \text{ then} \\
8 & \quad \quad \text{Output } S; \\
9 & \quad \text{if } Q \setminus (S \cup X) \neq \emptyset \text{ then} \\
10 & \quad \quad v \leftarrow \text{the smallest vertex in } Q \setminus (S \cup X); \\
11 & \quad \quad \text{if } |Q \cap S| < k + 1 \text{ then} \\
12 & \quad \quad \quad \text{Subk \text{MIG}(G, Q, S \cup \{v\}, X, k);} \\
13 & \quad \quad \text{if there exists a solution } S^* \text{ s.t. } S \subseteq S^* \land S^* \cap (X \cup \{v\}) = \emptyset \text{ then} \\
14 & \quad \quad \quad \text{Subk \text{MIG}(G, Q, S, X \cup \{v\}, k);} \\
15 & \quad \text{else} \\
16 & \quad \quad Q' \leftarrow \pi^{-1}(Q) + 1; \\
17 & \quad \quad \text{Subk \text{MIG}(G, Q', S, X, k);} \\
\end{align*}
\]

8.4.3 Correctness

In this section we show the following theorem, that is the correctness of our algorithm.

**Theorem 8.3.** Let \( G \) be a chordal graph and \( k \) be a non-negative integer. Then \( k \text{MIG}(G, k) \) outputs all and only maximal \( k \)-degenerate induced subgraphs of \( G \) without duplicates.

As mentioned in the description, \( k \text{MIG}(G, k) \) uses binary partition, thus every recursive call has either a single child (Line [17]) which will simply extend the current solution, or will produce two recursive calls (Lines [12] and [14]) that will lead to different solutions, as the first one considers only solutions for which \( v \in S \), and the second only solutions for which \( v \notin S \) (if any). Thus the same solution cannot be found more than once.

Furthermore, as we keep the invariant that \((S, X)\) is a partial solution, by Lemma 8.3 we know that when \( V = S \cup X \) then \( S \) is a maximal \( k \)-degenerate induced subgraph, thus \( k \text{MIG}(G, k) \) outputs only solutions.

Finally, any solution, \( i.e., \) maximal \( k \)-degenerate induced subgraph \( M \) is found by the algorithm, and we can prove this by induction: consider the set of cliques \( Q_1, Q_2, \ldots \) in \( QT(G) \), ordered by ranking. As a base condition, assume that \((S, X)\) is a partial solution such that \( S \subseteq M, X \cap M = \emptyset \); this is always true in the beginning, when \((S = \emptyset, X = \emptyset)\). Let \( Q_1 \) be the clique that we are considering, \( i.e., \) the one of smallest rank such that \( Q_1 \setminus (S \cup X) \neq \emptyset \), and \( v \) be the smallest vertex in \( Q_1 \setminus (S \cup X) \). If \( v \in M \), then the recursive call in Line [12] will consider a partial solution which has one more vertex in common with \( M \), \( i.e., \) \((S \cup \{v\}, X)\). Otherwise, \( v \notin M \), that is, there exists a solution \( S^* \) such that \( S \subseteq S^* \land S^* \cap (X \cup \{v\}) = \emptyset \), thus the recursive call in Line [14] is executed; this recursive call will consider a partial solution that has one more vertex in common with \( V \setminus M \), \( i.e., \) \((S, X \cup \{v\})\). In both cases the base condition is still true, thus by induction \( k \text{MIG}(G, k) \) will find \( M \). In order to prove Theorem 8.3 it only remains to show how to decide whether, given \((S, X)\), there is a solution containing \( S \) but
nothing in \(X \cup \{v\}\), i.e., how to compute Line 13. This is shown in the following lemma.

**Lemma 8.5.** Let \((S, X)\) be any partial solution of \(G\), \(Q\) be a clique such that its ancestor cliques are fully contained in \(S \cup X\), and \(v \notin S \cup X\) be a private vertex of \(Q\). Then, there exists a solution \(S^*\) such that \(S \subseteq S^*\) and \(S^* \cap (X \cup \{v\}) = \emptyset\), if and only if \(A(x, X \cup \{v\}) \geq 1\) for each vertex \(x \in N[v] \cap (X \cup \{v\})\).

**Proof.** Let \(X' = X \cup \{v\}\). If for each vertex \(x \in N[v] \cap X', A(x, X') \geq 1\), then \((S, X')\) still satisfies all the properties in Definition 8.1 as \(A(w, X)\) is unchanged for any vertex \(w \in X \setminus N(v)\). Thus \((S, X')\) is a partial solution, and a solution \(S^*\) is given by Lemma 8.4.

Suppose that there is a vertex \(x \in X'\) such that \(A(x, X') = 0\), i.e., there is no clique \(Q\) containing \(x\) such that \(|Q \setminus X'| \geq k + 1\). As \(X' \subseteq V \setminus S^*\) for any solution \(S^*\), there is no clique \(Q\) containing \(x\) such that \(|Q \setminus (V \setminus S^*)| \geq k + 1\), thus \(A(x, V \setminus S^*) = 0\), and there is no maximal solution \(S^*\) by Lemma 8.3.

Thus Theorem 8.3 is true, and \(k\text{-MIG}(G, k)\) finds all and only maximal \(k\)-degenerate induced subgraphs of the chordal graph \(G\) exactly once.

### 8.5 Complexity analysis

In this section we analyze the cost of our algorithm, and prove that it can enumerate all maximal \(k\)-degenerate induced subgraphs of \(G\) in \(O(m \cdot q)\) time per solution. First, we recall some important properties of cliques in chordal graphs.

**Remark 8.1** (From [BP93] and [GHP95]). Let \(G\) be a connected chordal graph with \(n > 1\) vertices and \(m\) edges. Then the number of maximal cliques in \(G\) is at most \(n - 1\), and the sum of their sizes is \(\sum \{|C| : C \in Q(G)\} = O(m)\).

And regarding the cliques in \(G\) containing a specific node, we can state the following.

**Lemma 8.6.** In a chordal graph \(G\), the number of maximal cliques containing a vertex \(v\) is at most \(|N(v)|\).

**Proof.** Consider \(G[N[v]]\), the subgraph of \(G\) induced by vertices of \(N[v]\). \(G[N[v]]\) is chordal as it is an induced subgraph of a chordal graph, it has \(|N[v]|\) vertices, and at most \(|N[v]| - 1 = |N(v)|\) maximal cliques, which exactly correspond to the maximal cliques in \(G\) containing \(v\).

Now, consider the cost of executing Line 13, which dominates the cost of each iteration of the algorithm. We show in the next lemma that it can be done efficiently by exploiting Lemma 8.5.

We recall that \(q\) denotes the maximum size of a clique in \(G\).

**Lemma 8.7.** Line 13 can be executed in time \(O(q \cdot |N(v)|)\).

**Proof.** By Lemma 8.5 it is sufficient to check, for every vertex in \(v \in N[v]\), whether there is a clique \(Q'\) containing \(x\) such that \(|Q' \setminus (X \cup \{v\})| \geq k + 1\). As \((S, X)\) is a partial solution, if a vertex \(x\) is not contained in any clique such that \(|Q' \setminus (X \cup \{v\})| \geq k + 1\), then there exists a clique \(Q'\) such that \(|Q' \setminus X| \geq k + 1 > |Q' \setminus (X \cup \{v\})|\) = \(k\), and thus \(x\) is contained in one of the cliques containing \(v\).

Assume we have a table that keeps track of the value \(B(Q) = |Q \setminus (X \cup \{v\})|\) for every clique \(Q\), and one that keeps the value \(A(x) = \left|[\{Q \mid x \in Q \land B(Q) \geq k + 1\}]\right|\). When adding \(v\) to \(X\), we can update the \(B\) table by decrementing \(B(Q)\) by 1 for every clique containing \(v\). The number of such cliques in a chordal graph is at most \(|N(v)|\) by Lemma 8.6. Every time the value of \(B(Q)\) is decremented to less than \(k + 1\), we can update the \(A\) table by decrementing \(A(x)\) by 1 for each vertex \(x\) in \(Q\). During this process, the check fails if and only if \(A(x)\) is decremented to 0 for any \(x\). The time required is \(|Q| \leq q\) for each considered clique, for a total cost of \(O(q \cdot |N(v)|)\).
Finally, we are ready to prove the complexity bound for \( k\text{MIG}(G,k) \).

**Theorem 8.4.** \( k\text{MIG}(G,k) \) runs with delay \( O(m \cdot q) \).

**Proof.** First, we need to compute \( QT(G) \), which takes \( O(n + m) \) time \([GHP95]\). Note that \( O(m + n) = O(m) \) as \( G \) is connected. Computing a pre-order traversal of \( QT(G) \) takes \( O(n) \) time as \( QT(G) \) has at most \( n \) nodes.

In each recursive call we add a vertex either in \( S \) or in \( X \) or consider a next maximal clique. Hence, the depth of the tree of recursive calls is bounded by \( 2n \). To bound the delay between two solutions \( M \) and \( M' \), it is enough to bound the sum of the cost of all recursive calls in the path from the recursive call outputting \( M \) to the one that outputs \( M' \). For clarity, let us use the term **recursive node** to refer a node in the tree of the recursive calls. Note that the recursive nodes that output a solution are exactly the leaves of this tree, thus the path between \( M \) and \( M' \) is bounded by the sum of the cost of a root-to-leaf and a leaf-to-root path.

As to execute Line 13 we use tables \( A \) and \( B \) (see Lemma 8.7), let us explain how to initialise them (we already explain in Lemma 8.7 how to update them). For each vertex \( x \), we set \( A(x) = |\{Q \in Q(G,x) \mid |Q| \geq k + 1\}| \), and set \( B(Q) = |Q| \) for each \( Q \in Q(G) \). In order to set these values we can simply iterate over all maximal cliques in \( QT(G) \): initialising \( B(Q) \) takes \( O(1) \) time, and if \( |Q| \geq k + 1 \) we increment \( A(x) \) by 1 for each \( x \in Q \), which takes \( O(|Q|) \) time. The total running time for initialising the tables \( A \) and \( B \) take thus \( O(n + m) = O(m) \) time (see Remark 8.1).

Let \( v_1, \ldots v_t \) be the recursive nodes in the path from the root to the node that outputs \( M' \). First, \( t \leq 2n \) as in each step either we add \( v \) to \( S \) or to \( X \) or we take another \( Q \). The delay now is the sum of the cost of each \( v_i \). Lines 9-14 can be done in time \( O(|N(x)| \cdot q) \) by Lemma 8.7. The cost for Lines 16-17 is \( O(1) \). By summing, we have the upper bound \( \sum_{Q \in Q(G)}O(1) + \sum_{x \in V(G)}O(|N(x)| \cdot q) = O(m \cdot q) \). The \( O(m) \) preprocessing cost is negligible as there always exists at least one solution.

Note that this holds for any value of \( k \): indeed, by Theorem 8.1 we know that chordal graphs are \( q - 1 \)-degenerate, thus for any \( k \geq q \), the problem is trivial as the only maximal solution is \( G \) itself.

### 8.6 Final remarks

We presented the first output-polynomial algorithm for enumerating maximal \( k \)-degenerate induced subgraphs in a chordal graph. The algorithm runs in \( O(m \cdot q) \) time per solution for any given \( k \). It would be interesting for future work to investigate the feasibility of an output-polynomial algorithm for general graphs. It is worth noticing that the enumeration of maximal independent sets in graphs is a special case as \( X \) is an independent set in \( G \) if and only if \( G[X] \) is 0-degenerate.
Part II

Beyond Enumeration
Finding diversified common substructures in protein networks

In this chapter we present a fast algorithm for the classical problem of finding common subgraphs between two graphs, which are useful for detecting structural relationships between biological macromolecules. We apply this algorithm to the problem of finding common substructures between two proteins, which are modeled as attributed graphs whose links represent chemical bonds.

As many solutions to the problem can correspond to essentially the same common substructure, with only tiny variations, we propose a method for diminishing the redundancy of the result to provide a set of diversified solutions, i.e., such that any returned solution is significantly different from all others.

Although the problem is computationally expensive, we improve performance by several orders of magnitude compared to known algorithms.

Furthermore, we show how our proposed algorithm is more suitable for finding diverse solutions under time constraints: while state of the art solutions are technically correct, they will tend to find similar solutions close to each other, thus spending a large amount of time to process a local area of the input networks; the proposed approach instead quickly finds solutions covering different areas of the networks, without getting “stuck” inside a local area.

These findings are validated by experiments on proteins with thousands of atoms.

9.1 Introduction

For any two given input graphs \( G \) and \( H \), a subgraph \( S \) of \( G \) is in common with \( H \) if \( S \) is isomorphic to a subgraph of \( H \); it is maximal if there is no other common subgraph that strictly contains it, and maximum if it is the largest. The maximum common subgraph problem asks for the maximum ones, or simply for their size: this problem is classical and useful for modeling structural similarity. The maximal common subgraph (MCS) problem further requires discovering all the MCS’s of \( G \) and \( H \).

The MCS problem can be constrained to connected and induced subgraphs (MCCIS) \[\text{CCC}^+ 08, \text{Koc01, KLW96}\], where the latter means that all the edges of \( G \) between nodes in the MCS are mapped to edges of \( H \), and vice versa: considering induced subgraphs reduces the search space \[\text{CCC}^+ 08\], and requiring connected subgraphs has been employed to further alleviate the explosion of the number of solutions \[\text{Koc01, KLW96}\].

Problem of interest. In this chapter, we consider the following modified version of the MCCIS problem, which we call \( T \)-MCCIS problem.

159
Problem 9.1 (T-MCCIS Problem). Given two graphs $G$ and $H$ and a spanning tree $T$ of $G$, list all the maximal common connected induced subgraphs between $G$ and $H$ for which the subgraph in $G$ is connected using edges of $T$.

We call these subgraphs $T$-MCCIS’s, and some examples are shown in Fig. 9.1. As we will see, their interest arises for their biological application. While a $T$-MCCIS is not necessarily a MCCIS, it is still a common connected subgraph. The adoption of the $T$-MCCIS model allows to find in practice common structures which are of larger size and more numerous with respect to the ones found by state of the art looking for MCCIS (see Section 9.4.2).

Note that listing MCCIS and $T$-MCCIS is computationally more demanding than listing maximal common induced subgraphs (relaxing the connectivity constraint). While the latter problem can be solved by finding maximal cliques, for which output sensitive enumeration algorithms are well known [CGMV16], the former problems require finding cliques with additional properties, to which known clique enumeration algorithms do not apply with guarantees, see for instance [KLW96]. We remark that spanning trees have been previously employed to prune the search for frequent subgraphs [HWPY04], although such techniques do not extend to this problem.

Furthermore, it should be remarked that in all state of the art approaches, as well as in this thesis, by MCCIS (or $T$-MCCIS) we actually mean isomorphisms corresponding to a MCCIS (or $T$-MCCIS), i.e. the subgraphs $G$ and $H$ and a compatible mapping of the nodes of $G$ into those of $H$. There are currently no known techniques for efficiently finding actual common subgraphs without considering isomorphisms. Listing isomorphisms may result in finding the same pair $G$ and $H$ more than once with different mappings; however, we will also address the issue of removing this redundancy a posteriori (see Section 9.3).

When nodes of $G$ and $H$ are labeled, the notion of isomorphism naturally extends by requiring, for instance, nodes of the common subgraph to have matching labels. We can also consider a generalized compatibility function between the nodes of the two graphs that determines whether a given node $i$ of $G$ can be mapped in a node $j$ of $H$. An analogous definition can be given for edges.

Contributions. In this chapter we solve the $T$-MCCIS problem by providing an output sensitive algorithm for (isomorphisms corresponding to) $T$-MCCIS’s. In particular, given the two graphs $G$ and $H$, and a spanning tree $T$ of $G$, let $\Delta_G$ and $\Delta_H$ be their maximum degree, respectively. For each listed $T$-MCCIS, we pay a cost of $O(q^3\Delta_{black}^2(\Delta_G + \Delta_H))$ time, where $q$ is the number of nodes of the listed $T$-MCCIS, and $\Delta_{black}$ is a parameter bounded by $O(\Delta_G \Delta_H)$. A strength of this bound is that it is independent of the sizes of $G$ and $H$. Moreover, no output sensitive algorithm is currently known for MCCIS’s, thus we give the first such algorithm for $T$-MCCIS.

To achieve our goal, we explore a variation of the product graph $P$ [Lev73] obtained from $G$ and $H$, so that $T$-MCCIS’s are found as special cliques in $P$. Recent advances on clique enumeration [CGMV16] might help to find these special cliques but, due to the new constraints, we have to solve several non-trivial issues to avoid the explosion of combinations of partial solution to get the new ones. Moreover, our approach does not materialize $P$, but navigates it implicitly to improve memory usage and running time. We refer the reader to Section 9.2.

Another contribution is that we use our solution to design a new method to find LACCIS’s,
which are large common connected induced subgraphs (not necessarily maximal nor maximum). They are relevant when comparing macromolecules in computational biology. For a set of spanning trees $T_1, \ldots, T_k$, we consider the set of LACCIS’s such that each LACCIS $L$ contains a $T$-mccis $S$ for some $T \in T_1, \ldots, T_k$. In general, $L$ satisfies $S \subseteq L \subseteq M$ for a maccis $M$, where $\subseteq$ denotes the containment relation among induced subgraphs. Hence, the larger $L$, the closer it is to a maccis.

Our algorithm, called FLASH (Fast Laccis Searching Heuristic), takes two connected labeled graphs $G$ and $H$ as input, along with some random spanning trees $T_1, \ldots, T_k$ of $G$. It applies the above approach, and returns a set of LACCIS’s, where each LACCIS is represented as a pair of subsets of nodes, one from $G$ and the other from $H$.

FLASH uses our solution for the $T$-mccis problem for each tree, accumulating the found $T$-mccis’s for $T = T_1, \ldots, T_k$. Then, it greatly reduces their number by a filtering criterion to make sense of the massive output: for a user-defined percentage $\sigma$ (e.g. 70%), it selects a “covering” set of small size, such that each of the discarded $T$-mccis’s has more than $\sigma$ overlap with a retained one (priority is given to large ones). Interestingly this filter shows that FLASH quickly finds solutions spanning different parts of $G$ and $H$, whereas other approaches such as [Klaw96] tend to spend lot of time on the same nodes: small local additions and deletions of nodes produce a plethora of different subgraphs that significantly overlap.

We show that using $T$-mccis’s instead of mccis’s is more efficient for finding LACCIS’s since the running time of FLASH for a given spanning tree $T$ is provably proportional to the number of reported $T$-mccis’s. This is in contrast with the known algorithms that use maximal common subgraphs, which have the drawback of running into a computational blackhole, going through an exponential number of substructures even if there are few $T$-mccis’s: for a $T$-mccis of $k$ nodes, these algorithms have to potentially find and discard $2^k$ included subgraphs of it. When dealing with graphs of non-trivial size (e.g., thousands of nodes) we argue that the state-of-the-art approaches that use maximal common subgraphs do not terminate within a conceivable time, thus making a practical comparison hard to perform. In our experiments, the size $k$ of common subgraphs can easily be in the order of the hundreds and FLASH performs well in practice even though its theoretical worst-case complexity is exponential. We refer the reader to Section 9.4.2.

**Maximum vs maximal common subgraphs.** Graph-based methods provide a natural complement to sequence-based methods in bioinformatics and protein modeling. Graph algorithms can identify compound similarity between small molecules, and structural relationships between biological macromolecules that are not spotted by sequence analysis [ASW05]. These algorithms find motivation in the increasing amount of structured data arising from X-ray crystallography and nuclear magnetic resonance. Many examples of graphs fall under this scenario, such as chemical structure diagrams [Bon91, GAW97], 3D patterns for proteins [GAW97, Klaw96], amino acid side-chains [APG+94], and compound similarity for the prediction of gene transcript levels [VBWDG+13], to name a few. It is believed that finding similar structures leads to highlighting similar biochemical properties and functionalities [OYH+07, SK02]. For these reasons, the bioinformatics community has repeatedly expressed its interest in common subgraphs detection from a computational point of view [AK14, BGF12, CFV07, Kan92, Klaw96, VBWDG+13, RGW02, ASW05, BB76, BW87, HLJ06, ER11, CVM77].

Some people might confuse the problem of finding the maximum common subgraph with that of finding all the maximal common subgraphs: they are indeed different problems, solved with different techniques and results. It is in practice much faster to find the maximum common subgraph than all MCS’s (e.g. see [CFV07]). However, a maximum common subgraph is not always meaningful as a structural motif, as it does not necessarily contain all the relevant or large common structures. Consider the two molecules represented by the graphs in Figure 9.2 as an example. If we examine these molecules, a natural question is determining which parts of
the two are common. As it can be seen, the two molecules share a tetracenic moiety (blue) and an N-methylbenzopiperidinic moiety (red). It is worth observing that the maximum common structure in terms of size is represented by the blue group, which is however a fairly common structure in organic molecules; the red one which is maximal (but not maximum), however, is more likely to be interesting as it is more peculiar. In general, there may be arbitrarily large common substructures that give few information because of their frequent appearance in special type of macromolecules or polymers. In this context, modeling the problem from a mathematical perspective can be useful to efficiently retrieve these common structures.

Furthermore, when spotting structural motifs, it is not always possible to fix a priori the scoring system, and the maximum common subgraphs are not necessarily the ones getting the best score: a postprocessing can apply several scoring systems with a fast filtering and ranking of the MCCIs. This is more efficient than repeating a branch-and-bound search for each score.

Related works. Both maximum and maximal common subgraphs problems have been studied for decades [BW87, CJG08, GAW97]. In the case of the maximum common subgraph, the corresponding decision version is NP-complete as it solves the subgraph isomorphism problem. The problem remains NP-hard, even on restricted graph classes such as outerplanar graphs, and becomes polynomial only if the degree is bounded or for trees [AT13]. The problem is difficult to approximate (MAX-SNP hard) even within a polynomial factor [Kan92], and is \( W[1]\)-hard when parameterized by the treewidth of the input graphs [AK14]. Due to these difficulties, enumerating all the maximum common subgraphs cannot be done in an output sensitive way, e.g. with polynomial delay, unless \( P=NP \). Listing all maximal common connected induced subgraphs is a different problem, since it can produce exponentially more solutions than finding maximum common subgraphs. Moreover, finding just the maximum common subgraphs, as opposed to listing all maximal ones, allows for very effective cuts to the search space (e.g. branch-and-bound) which makes the computation much faster in practice, allowing researchers to process larger graphs with the available resources.

Due to the strong connection between graph isomorphism and common substructures, the bioinformatics community has repeatedly expressed its interest in both these problems from a computational point of view, looking at them as two possible ways of finding LACIS’s. However, we observe that the great majority of the works deal with maximum common subgraphs, rather than maximal ones.

For both the problems, the previous work can be roughly classified into three categories: clique-based methods [GAW97, KWT96, SAKZ05], non-clique-based backtracking methods [KH04, Mcg82, WZ97], and other techniques based on special classes of graphs [AT13, GN98].

Clique-based methods are widely employed and rely on the product graph \( P \), transforming the common subgraphs of \( G \) and \( H \) into maximal cliques in \( P \). This reduction dates back to the 70s [Lev73] and has been shown to be effective on biological networks [CFV07, GAW97, KWT96, SAKZ05]. For finding the maximal cliques, the algorithms by Bron and Kerbosch [BK73] or Carraghan and Pardalos [CP90] have been employed. Cao et al. [CJG08] observe that
materializing $P$ can be memory-wise expensive, and we show how to avoid this in our approach.

As previously observed, finding MCCIS’s of $G$ and $H$ corresponds to finding “special” cliques [Koc01]. To this aim, previous works on explicit product graphs for listing all the MCCIS’s, such as the ones by Koch [Koc01], employ a modified version of the Bron-Kerbosch algorithm which does not perform pivoting, a pruning technique. The resulting algorithm is not output sensitive, since it iterates on every possible subset of each common subgraph, and its complexity and cost per solution are not clearly bounded. Many works focused on using and improving Koch’s algorithm to get only one maximum common connected induced subgraph: [BDDS15] compares favorably Koch’s algorithms with other existing algorithms to compute this maximum. This problem often finds its application in the comparison of the global structure of proteins [FMKH13], lists all the common connected induced subgraphs, including the non maximal, to extract the maximum without using a product graph. [VV08] relaxed the constraint of induced and compared to the maximum found by using Koch. When instead considering all the maximal common connected induced subgraphs, Koch’s algorithm is still the state of the art (e.g. [Was16] or [Wel11]) greatly used in practice, even in recent years (e.g. [YWCW15]).

Backtracking algorithms mostly build up on Ullman’s strategy [Ull76] for subgraph isomorphism (e.g. [Mcg82]). They often use branch-and-bound heuristics based on the specific requirements of the application at hand. The comparison in [CFV07] shows how direct implicit methods for the maximum common subgraph, such as the one in [Mcg82], can outperform methods that exploit the product graph if the input graphs are small or contain many different labels. However, they do not apply efficiently to listing all the MCCIS’s. Other techniques, such as dynamic programming can be employed for special classes of graphs [AT13].

Other variations of the MCCIS problem have been considered, when $G$, $H$, and their isomorphisms are restricted to trees [DHKM14].

9.2 Output sensitive T-MCCIS’s enumeration

This section shows how to efficiently solve the T-MCCIS enumeration problem. In Section 9.2.1 we show how to turn the problem of finding T-MCCIS’s into that of finding suitable cliques in a new graph. In Section 9.2.2 we perform an output sensitive listing of such cliques.

9.2.1 Problem transformation: implicit product graph

We employ a variant of the transformation adopted by Koch [KLW96] and borrowed from Levi [Lev73], where we modify the color rule to take into account the edges of the spanning tree $T$. Define a colored product graph $P = GH$. Letting $P = (V_P, E_P)$ be the resulting undirected graph, the nodes in $V_P$ corresponds to ordered pairs of compatible nodes from $G$ and $H$, the first from $G$ and the second from $H$, and the edges in $E_P$ are as follows. Given two nodes in $V_P$ corresponding to $(x, i)$ and $(y, j)$, where $x, y \in G$, $i, j \in H$ and $x \neq y$, $i \neq j$, there is an edge in $E_P$ between $(x, i)$ and $(y, j)$ iff: (i) $\{x, y\} \in T$ (tree edge) and $\{i, j\} \in H$ (in this case, the edge is black), (ii) $\{x, y\} \in G \setminus T$ (non-tree edge) and $\{i, j\} \in H$ (the edge is white), (iii) both $\{x, y\} \notin G$ and $\{i, j\} \notin H$ (the edge is white).

The difference with the transformation by Koch [KLW96] is in condition (ii): we obtain exactly Koch’s transformation if those edges are marked as black instead of white. Nonetheless, we will show that this difference is quite crucial to obtain our output sensitive algorithm. An example of the above transformation is shown in Fig. 9.3. As in [KLW96], there is a one-to-one correspondence between maximal cliques in $P$ and maximal isomorphisms between subgraphs of $G$ and $H$. The main difference is that in our case a maximal clique connected by black edges corresponds to a maximal subgraph connected by edges of $T$, instead of generic edges of $G$. We
Figure 9.3: (a) Two graphs $G$ and $H$, where $T$ is shown in red; (b) a portion of the induced subgraph of $P$ involving $(A,1),(B,2),(C,3),(D,1)$, and $(E,5)$ (note that the edge $(A,1),(D,1)$ is not present); (c) a $T$-MCCIS (d) not a $T$-MCCIS since it is common but not spanned by $T$.

call this kind of black-connected maximal clique a BC-clique, and reduce the problem of finding the $T$-MCCIS’s to that of finding the isomorphisms/BC-cliques in the implicit $P$. We observe that the same $T$-MCCIS can give raise to several maximal isomorphisms/BC-cliques (matching the two sets of nodes in that $T$-MCCIS) that should be successively distilled to list it exactly once.

Building and navigating $P$ is costly: $P$ is a dense, massive graph with large maximum degree even when $G$ and $H$ are relatively small, sparse and with bounded degree. We avoid storing $P$ explicitly, and only store $G$ and $H$: we check compatibility between assignments in constant time and iterate on neighbors in constant time per element by applying “on the fly” the rules used for the generating $P$. This saves both memory and time, as $G$ and $H$ are much smaller and faster to access than $P$. In Section 9.2.2 we will show how to find the BC-cliques in $P$.

9.2.2 Finding BC-cliques

We describe in this section our algorithm for finding BC-cliques in the implicit product graph $P$. The algorithm works incrementally by finding all BC-cliques of increasingly larger subgraphs $P_{\leq i}$ of $P$, to finally find all BC-cliques of $P$. We first give a simplified version of the algorithm in Section 9.2.2 then the final one in Section 9.2.2.

First algorithm for BC-cliques

Our algorithm works by extending cliques: given a BC-clique $K$, a node $v$ fully extends $K$ if it is connected to all the nodes in $K$ and has at least one black edge to a node in $K$; $v$ partially extends $K$ if it has at least one black edge to a node in $K$, but is not connected to all the nodes in $K$; $v$ extends $K$ if it fully or partially extends $K$. A crucial ingredient is finding a good ordering of the nodes in $V_P$, which is defined as follows.

Definition 9.1. Given a graph $P$ whose edges are either black or white, we say that an ordering of the nodes in $V_P$ is a good ordering if: for any node $v$ that has neighbors connected to it with black edges that come before $v$ in the ordering, any pair of these neighbors is not connected with a white edge.

Note that the neighbors of $v$ smaller than $v$ may be connected with a black edge, or disconnected. In other words, if a node has two black edges towards neighbors that come earlier in the good ordering, either those neighbors are also connected by a black edge or they are not neighbors. We show that it is always possible to find a good order for the product graph $P$ defined in Section 9.2.1.

Lemma 9.1. Given two graphs $G$ and $H$, and a spanning tree $T$ of $G$, it is always possible to find a good order for the corresponding product graph $P$.

Proof. We can number the nodes of $G$ using a preorder traversal of $T$, so that given any node $u$ of $G$, there is only one edge of $T$ between $u$ and its neighbors $u'$ with $u' < u$. Consider $H$, we
just number its nodes in arbitrary way. As for $V_G$, we consider the lexicographical order $\prec$ on the pairs $(x, i)$ that corresponds to the nodes in $V_P$, where $x$ follows the order for $G$ mentioned above and $i$ that for $H$. The numbering of the nodes in $V_P$ obtained by numbering them consecutively in increasing lexicographical order $\prec$ of the corresponding pairs is a good ordering.

The following lemma holds.

**Lemma 9.2.** Let $P_{<v}$ denote the subgraph of $P$ induced by the nodes $v' < v$, and $P_{<v} \cup \{v\}$ the one induced by $v' \leq v$: if $C$ is a bc-clique in $P_{<v} \cup \{v\}$, then $C \setminus \{v\}$ is connected with black edges in $P_{<v}$.

**Proof.** If $v$ has only one black edge towards the rest of $C$, then $C \setminus \{v\}$ would still be connected with black edges as no path made of black edges between nodes of $C \setminus \{v\}$ could involve $v$. Thus let us assume $v$ had at least two black edges towards $C \setminus \{v\}$. Let $v = (x, i)$ and the other endpoints of these edges be $(y, j)$ and $(y', j')$. Since $(x, i), (y, j)$ and $(y', j')$ are connected, by construction we have $x \neq y \neq y'$, and since $(C \setminus \{v\}) \in P_{<v}$ we have $y < x$ and $y' < x$. Still by construction, there are only black edges between $(x, i)$ and nodes of $P$ which contain neighbors of $x$ in $T$. By the good ordering, we know that $x$ has only 1 neighbor in $T$ smaller than itself. Thus either $y$ or $y'$ is not a neighbor of $x$ in $T$, and one of the edges is not black, a contradiction.

Observe that a bc-clique in $P_{<v} \cup \{v\}$ is either a bc-clique of $P_{<v}$ or contains $v$. Moreover, a bc-clique $C$ containing $v$ is such that $C \setminus \{v\}$ is connected by black edges, because of Lemma 9.2 and so it is contained in a bc-clique of $P_{<v}$. This property ensures that every bc-clique can be found incrementally: when adding a new node $v$ to the set of bc-cliques found up to that point for the nodes of $P_{<v}$, two or more of the latter bc-cliques cannot be united because of the black edges incident to $v$, since the removal of $v$ cannot disconnect the bc-clique. Hence we can consider just one of them to be extended by $v$, rather than any combination of them.

We are now ready to describe our algorithm which assumes that $P$ is explicitly stored. We call this algorithm **INC-GENERATOR** (incremental-generator).

Let $R_v$ be the set of bc-cliques in $P_{<v} \cup \{v\}$. Considering the nodes $v_1, \ldots, v_p \in P$ in the good order, for each $v_i$ we build the set $R_{v_i}$ from $R_{v_{i-1}}$ ($R_{v_1}$ contains only $\{v_1\}$). Clearly, $R_{v_p}$ is the set of all bc-cliques of $P$. In order to obtain $R_{v_i}$,

1. we iterate over the cliques $K$ in $R_{v_{i-1}}$ and we do the following.
   a. If $v_i$ fully extends $K$, add $K \cup \{v_i\}$ to $R_{v_i}$, otherwise add $K$ to $R_{v_i}$.
   b. If $v_i$ partially extends $K$, define $C$ as the component connected by black edges and containing $v_i$ in the graph induced by $K \cap N(v_i) \cup \{v_i\}$

   * If $C$ cannot be fully extended with a node smaller than $v_i$, add the clique $C$ to $R_{v_i}$.

2. If no clique in $R_{v_{i-1}}$ is extended by $v_i$, add $\{v_i\}$ to $R_{v_i}$.

Note that the same clique might be generated more than once, thus it is important to keep $R_{v_i}$ as a set, not retaining duplicates. **INC-GENERATOR** tries to extend each $K$ in $R_{v_{i-1}}$ with $v_i$. If $v_i$ does not fully extend $K$, $K$ is maximal also in $R_{v_i}$. Moreover, each time an extension (full or partial) of $K$ is possible, this is done, except in the step 1.b: indeed since $C$ is not maximal, it is not added to $R_{v_i}$. Note that the completeness of the approach is not affected by discarding $C$: by Lemma 9.3, if there exists a node $v_j$ smaller than $v_i$ extending $C$, there exists a clique $C'$ in $R_{v_{i-1}}$ such that $C \cup \{v_j\} \subseteq C'$.

**Lemma 9.3.** Any bc-clique in $R_{v_i}$ is either in $R_{v_{i-1}}$ or can be obtained by extending a clique in $R_{v_{i-1}}$ with $v_i$.

---

1Note that $C$ is a bc-clique and contains some, but not all, nodes of $K$. 165
Proof. Assume there exist a bc-clique \( C \in R_{v_i} \) that is not obtained by the algorithm when processing \( v_i \). All cliques in \( R_{v_i} \) which do not contain \( v_i \) are in \( R_{v_{i-1}} \) by definition, thus \( C \) contains \( v_i \). By Lemma 9.2 \( C \setminus \{ v_i \} \) is a clique connected with black edges. If \( C \setminus \{ v_i \} \) is maximal in \( P_{<v} \), then it is fully extended by \( v_i \) and \( C \) is found in step 1.a. Otherwise, \( C \setminus \{ v_i \} \) is contained in a clique \( C' \) maximal in \( P_{<v} \). We thus have \( C \subseteq (C' \cup \{ v_i \}) \). If \( C \neq C' \cup \{ v_i \} \) then \( C \) is not maximal in \( R_{v_i} \), a contradiction.

\[ \square \]

**Lemma 9.4.** INC-GENERATOR outputs just BC-cliques.

Proof. We prove that at any step, \( R_{v_i} \) contains only BC-cliques, which are maximal in \( P_{<v_i} \cup \{ v_i \} \). By induction, all the cliques that contain \( v_i \) are discarded if not maximal (step 1.b) and the other ones, since they cannot be fully extended by \( v_i \) and were maximal in \( R_{v_{i-1}} \), are still maximal. Thus in the final step \( R_{v_p} \) will contain only BC-cliques maximal in \( P_{<v_p} \cup \{ v_p \} = P \).

By Lemma 9.3 and since INC-GENERATOR outputs just BC-cliques by Lemma 9.4 we obtain the following.

**Corollary 9.1.** Algorithm INC-GENERATOR outputs all and only BC-cliques.

For the sake of completeness we remark that Algorithm INC-GENERATOR has polynomial cost per solution, as shown by the following lemma.

**Lemma 9.5.** Algorithm INC-GENERATOR has polynomial cost per solution.

Proof. Note that, since \( |R_{v_{i-1}}| \leq |R_{v_i}| \), we have that \( \sum |R_{v_i}| \leq p|R_{v_p}| \); in the execution of our algorithm, the number of times in which any clique is processed is bounded by \( p \) times the number of BC-cliques of \( P \). Since processing each clique takes polynomial time, our total running time is given the number of BC-cliques in the graph multiplied by a polynomial.

**Refined version**

In this section we present a refined version of the algorithm shown in Section 9.2.2 showing that our new version is equivalent to INC-GENERATOR. Our new algorithm is called GENERATOR and is a variant of the reverse search that takes into account the distinction between black and white edges and recursively examines all the BC-cliques as explained before. In this version, we avoid using the sets \( R_{v_i} \) as they can potentially include an exponential number of BC-cliques. Moreover, we deal with the fact that the product graph is not explicitly materialized, meaning that when retrieving desired nodes of \( P \), which are pairs of nodes in \( G \) and \( H \), this task should be addressed just by looking at \( G \) and \( H \). Note that deciding whether a node in \( P \) is less than another, can be done just by looking at the order induced on \( G \) and \( H \) as shown in Lemma 9.1.

The pseudocode is shown in Algorithm 28. The roots of the recursion trees are given by all the nodes that have no black backwards edges. Recall that, in algorithm INC-GENERATOR, each BC-clique \( K' \in R_{v_i} \), for any \( i \), is either a single node with no backward black edges or is generated from another BC-clique \( K \) in \( R_{v_j} \), with \( j < i \). Given a BC-clique \( K \) in \( R_{v_j} \), we analyze the properties of all \( K' \) generated from \( K \) in algorithm INC-GENERATOR. In particular, we identify for which \( i > j \), \( K \) can generate a clique in \( R_{v_i} \). The Lemma below trivially follows from the definition of BC-clique.

**Lemma 9.6.** In algorithm INC-GENERATOR, if a BC-clique \( K \) generates a new BC-clique when processing \( v_i \), then there is a black edge from \( v_i \) to a node in \( K \).

Thus, it is easy to find all nodes \( v_i \) that can be used to extend \( K \) using the following.
Algorithm 28: GENERATOR: Finding the BC-cliques in the product graph \( P \)

**Input:** Two graphs \( G \) and \( H \), and a spanning tree \( T \) of \( G \).

**Output:** The BC-cliques in the implicit product graph \( P = GH \) for \( T \).

```plaintext
1. Let \( v_1, \ldots, v_p \) be the nodes of \( V_T \) in lexicographic order.
2. \textbf{foreach} \( v_i \) having no black edges going to \( v_j \) with \( j < i \) \textbf{do}
   1. \( \text{Generate}([v_i]) \);
   2. \textbf{if} \( K \) is maximal \textbf{then} output \( K \);
   3. Let \( X \) be the nodes larger than \( \text{max}(K) \) connected to \( K \) by a black edge.
   4. \textbf{for} \( v_i \in X \) \textbf{do}
      5. Let \( C \) be the nodes in \( (K \cap N(v_i)) \cup \{v_i\} \) that \( v_i \) can reach using black edges in \( (K \cap N(v_i)) \cup \{v_i\} \).
     6. \textbf{if} \( C \) is a child of \( K \) \textbf{then} \( \text{Generate}(C) \).
```

**Property 9.1.** A node \( v_i \) can extend \( K \) if it is connected to \( K \) with black edges and it is greater than the largest node in \( K \) in the good ordering.

The set of these nodes \( v_i \) corresponds to the set \( X \) in Algorithm 28. Property 9.1 helps to find the \( K' \) which are generated from \( K \) in Algorithm \textsc{inc-generator}: they are found among the ones obtained recursively extending \( C \cup \{v_i\} \), where \( C \) is the set of nodes in \( K \cap N(v_i) \) that \( v_i \) can reach using black edges. This allows us to generate the BC-cliques without using the sets \( R_{v_i} \) explicitly.

Given a BC-clique \( C \), and letting \( v \) be the largest node of \( C \) in the good ordering, we define the parent of \( C \) as the BC-clique in \( P_{<v} \) obtained by maximalizing \( C \setminus \{v\} \) in \( P_{<v} \), that is, recursively adding the smallest node in \( P_{<v} \) that can fully extend the current (non-maximal) BC-clique until it is maximal in \( P_{<v} \) (no more nodes can be added).

**Lemma 9.7.** The parent \( K \) of a BC-clique \( C \in R_{v_i} \) with \( v_i = \text{max}(C) \), is unique and is a BC-clique in \( R_{v_{i-1}} \).

**Proof.** By Lemma 9.2, \( C \setminus \{v_i\} \) is a clique connected by black edges. By recursively adding nodes in \( P_{<v_i} \) that fully extend it, we obtain a clique \( K \) that is connected by black edges and maximal in \( P_{<v_i} \), that is, a BC-clique in \( P_{<v_i} \). As \( P_{<v_i} = P_{<v_{i-1}} \cup v_{i-1} \), \( K \) is a BC-clique in \( R_{v_{i-1}} \). Furthermore, recall that \( R_{v_{i-1}} \) is a set; as there are no duplicates the parent \( K \) of \( C \) is unique.

Considering the nodes added to \( C \setminus \{v_i\} \) to compute \( K \) are not neighbors of \( v_i \) reachable with black edges (otherwise \( C \) would have not been maximal), we have that \( (K \cap N(v_i)) \cup \{v_i\} \supseteq C \), thus by Lemma 9.7 it follows that:

**Corollary 9.2.** A clique \( C \), with \( v = \text{max}(C) \), can be obtained by extending its parent with \( v \).

**Definition 9.2.** Given two cliques \( K \) and \( C \), and \( v = \text{max}(C) \), we say that \( C \) is a child of \( K \) iff \( C \) is maximal in \( P_{<v} \), and the parent of \( C \) is \( K \).

Note that in Algorithm 28 when trying to generate \( C \) from \( K \), we accept \( C \) only if its parent is \( K \); otherwise, \( C \) is discarded. This avoids generating duplicates: as every BC-clique has exactly one parent, and can only be generated in one way from any BC-clique (when adding \( v_i \)), clearly it is impossible to generate any clique more than once, obtaining:

**Lemma 9.8.** Any BC-clique produced by \textsc{generator} is generated exactly once.

Finally, we characterize for which \( v_i \) the BC-clique \( \{v_i\} \) in the algorithm in Section 9.2.2 is added to \( R_{v_i} \). This corresponds to the \( v_i \) having no black edges going to \( v_j \), with \( j < i \). For each of these sets, in Algorithm 28 we start our recursive procedure. From this observation, applying Property 9.1 and Lemma 9.7, we can conclude the following.
Lemma 9.9. **Algorithm generator is equivalent to algorithm inc-generator.**

By Lemma 9.1 and Lemma 9.9 we can conclude that Algorithm 28 is correct. Let us now discuss its time complexity, explaining also how not to store explicitly $P$.

In the following, we refer to $q$ as the maximum number of nodes in a bc-clique, $\Delta_{\text{black}}$ as the maximum number of black neighbors of a node in $P$, and $\Delta_G$ and $\Delta_H$ as the maximum degrees respectively in $G$ and $H$. Note that checking the existence of an edge in $P$ takes constant time, assuming that takes constant time in $G$, $H$ and $T$, e.g. using Cuckoo Hashing. Indeed, given two nodes $(x, i)$ and $(y, j)$ it is sufficient to apply the rules described in Section 9.2.1, i.e., checking the existence of an edge in respectively $G$, $H$ and $T$. The following two results hold.

**Lemma 9.10.** **Maximalizing a clique $K$ in $P_{<v}$ can be done in $O(q\Delta_{\text{black}}(\Delta_G + \Delta_H))$ time.**

**Proof.** Maximalization of $K$ in $P_{<v}$ can be done by computing the set $B$ of all the black neighbors of nodes in $K \setminus \{v\}$; then consider each $b$ in $B$ in increasing lexicographical order, and add $b$ to the clique if it is adjacent to all nodes of $K$. Each time a node is added, its black neighbors are added to $B$. The total cost for building $B$ is $O(q\Delta_{\text{black}})$ and its size $|B| \leq q\Delta_{\text{black}}$. Note that, if we assume that our graph is given by the product of two graphs $G$ and $H$, checking if a node $v = (a, b)$ is adjacent to all nodes of $K$ can be done in $O(\Delta_G + \Delta_H)$ time. Indeed, we only need to check if the sets of neighbours of $a$ in the $G$-side of $C$ is mapped exactly to the sets of neighbours of $b$ in the $H$-side of $C$. Since these two sets can be computed in $O(\Delta_G)$ and $O(\Delta_H)$ respectively, the total time taken is $O(\Delta_G + \Delta_H)$. This gives us a total cost of $O(|B|(\Delta_G + \Delta_H)) = O(q\Delta_{\text{black}}(\Delta_G + \Delta_H))$.

**Lemma 9.11.** **In Algorithm 28 for each $v_i \in X$, $C$ can be computed in $O(q\Delta_{\text{black}})$ time.**

**Proof.** Let $C' = (K \cap N(v_i)) \cup \{v_i\}$; computing $C'$ is done in $O(|K|)$ by iterating over $K$ and checking the neighborhood with $v_i$ in $P$ (which takes constant time). To compute $C'$, it is sufficient to retain in $C'$ only nodes reachable from $v_i$ with black edges in $C'$. This can be done with a traversal of $C'$ from $v_i$ that only uses black edges in $C'$. As the number of black neighbors of a node is bounded by $\Delta_{\text{black}}$, this takes $O(|C'|\Delta_{\text{black}})$. As $|C'| \leq |K| + 1$ the cost $O(q\Delta_{\text{black}})$ follows.

**Lemma 9.12.** **A recursive call of Algorithm 28 takes $O(q^2\Delta_{\text{black}}^2(\Delta_G + \Delta_H))$ time.**

**Proof.** Checking whether $K$ is maximal takes $O(q\Delta_{\text{black}}(\Delta_G + \Delta_H))$ by Lemma 9.10. Computing $X$ (line 3) can be done in $O(q\Delta_{\text{black}})$ by iterating over the black neighbors of all nodes in $K$, and it gives us $|X| \leq q\Delta_{\text{black}}$. The loop is executed $|X|$ times, and each iteration costs the sum of lines 5 and 6. Line 5 takes $O(q\Delta_{\text{black}})$ by Lemma 9.11 and line 6 takes $O(q\Delta_{\text{black}}(\Delta_G + \Delta_H))$ by Lemma 9.10 as it can be done by checking that maximizing $C \setminus \{\text{max}(C)\}$ in $P_{<\text{max}(C)}$ yields $K$, and maximizing $C$ in $P_{\leq\text{max}(C)} \cup \{\text{max}(C)\}$ yields $C$. The total cost is thus dominated by the cost of the loop, that is $O(q\Delta_{\text{black}}(q\Delta_{\text{black}} + q\Delta_{\text{black}}(\Delta_G + \Delta_H))) = O(q^2\Delta_{\text{black}}^2(\Delta_G + \Delta_H))$.

As for each recursive call that outputs a bc-clique there are at most $q - 1$ calls which do not, the total cost per solution is given by the following.

**Theorem 9.1.** **Algorithm 28 has cost per bc-clique equal to $O(q^3\Delta_{\text{black}}^2(\Delta_G + \Delta_H))$ time.**

**Proof.** Either a solution is output in line 2 or $K$ is not maximal, meaning that it can be fully extended by one or more nodes in $X$. Let $v$ be the smallest of such nodes; $K' = K \cup \{v\}$ is certainly a good child of $K$, as there are no smaller candidates, nor nodes smaller than $\text{max}(K)$, that can fully extend $K'$, and $K' \setminus \text{max}(K') = K' \setminus \{v\} = K$ thus $K$ is the parent of $K'$. 168
The same is true for $K'$, which will either be maximal or produce a child which includes $K'$. Finally a descendant recursive call will output a maximal BC-clique $K'' \supseteq K' \supseteq K$. As in every recursive call a node is added to the temporary result, the number of such calls is bounded by $|K''|$. Thus, for each recursive call that outputs a BC-clique there are at most $q - 1$ calls which do not output a BC-clique.

It follows that the cost per solution of Algorithm 28 is bounded by $q$ times the cost of a recursive call, i.e. $O(q^3 \Delta_{black}^2 (\Delta_G + \Delta_H))$.

9.3 Improving the usability of the result

This section shows how to use our solution for the $T$-mccis problem to obtain a new heuristic, called FLASH, to improve the practical usability of the result, that is, finding larger and less redundant common subgraphs between two input labeled graphs $G$ and $H$. We then validate the effectiveness of FLASH in the following section.

FLASH aims to find LACCIS’s for two labeled undirected graphs $G$ and $H$. First, it solves the $T$-mccis problem for a set of given (random) spanning trees of $G \{T_1, \ldots, T_k\}$, obtaining the set of $T$-mccis’s for each $T \in \{T_1, \ldots, T_k\}$. Then, it post-processes the results as described below.

9.3.1 Filtering

It is important to distill all the $T$-mccis’s found for each $T$. Those leading to the same LACCIS’s are clearly redundant, and those that are small or mostly overlapping prevent us from making sense of a massive output. We base our filtering approach on two principles: firstly, larger common subgraphs are in general more significant than smaller ones; secondly, a common subgraph $X$ is significant by itself, but if some other, larger, common subgraph $X'$ covers a large enough amount of its vertices, then $X$ becomes redundant.

In the following, we define as $\tau$ the smallest size for which a common subgraph is a significant LACCIS, and $\sigma$ the containment threshold for which a subgraph can be considered redundant. The FILTER procedure is then implemented as follows:

1. Let $R$ be an empty set.
2. We scan the found solutions in decreasing size order.
3. A solution is added to $R$ iff (i) it is not smaller than $\tau$ and (ii) it does not overlap with any solution in $R$ for more than $\sigma$.
4. Finally, $R$ is returned to the user.

Formally speaking, as a common subgraph $X$ corresponds to both a subgraph $X_G$ of $G$ and a subgraph $X_H$ of $G$ which are isomorphic, we say that $X$ overlaps with $X'$ more than $\sigma$ if either the portion of vertices in $X_G$ contained in $X'_G$, or that of vertices of $X_H$ contained in $X'_H$ is greater than $\sigma$.

The final $R$ has the property that every solution, on top of being at least as large as $\tau$, will be significantly different from any other solution, thus the amount of redundancy will be reduced. At the same time, however, all of the information in the solutions discarded will be somewhat preserved in $R$: every discarded solution will be contained by a significant amount ($\sigma$) in one that is in $R$.

\[\text{Other non-random selection strategies for the spanning trees can be considered, but we found no significant difference in the result.}\]
It should be remarked that while the size of $R$ can be kept under control (i.e., by increasing $\tau$ or decreasing $\sigma$), the set of all $T$-mccis is likely to be too large to be stored in main memory. An effective way to perform this step is to store the result in secondary memory as it is produced, then sort it by size with external memory algorithms, and finally read it in a streaming fashion for the filter procedure.

### 9.3.2 Recombining

As observed in the introduction, the $T$-mccis found may be fragments of larger (maximal) common subgraphs. This is true for both $T$-mccis with respect to the same spanning tree, or to different ones.

After running the filter procedure on the merged output of all trees, FLaSH runs a procedure which iteratively combines compatible $T$-mccis together in order to produce a larger LACCIS.

Two $T$-mccis’s (or LACCIS) $X$ and $Y$ are compatible if they can be (partially) merged and their induced subgraphs in $G$ and $H$ are connected by one or more edges: RECOMBINE takes a maximal part of $Y$ that can be added to $X$, and creates a larger LACCIS by merging them.

This is easily implemented by looking at the implicit product graph $P$: consider the subgraph $P[X \cup Y]$, that is, the union of two bc-cliques. RECOMBINE takes the bc-clique corresponding to $X$, and greedily adds to it vertices of $Y$ until there are no more for which the result is still a bc-clique. Note that this only gives a common subgraph “maximal” with respect to $X$ and $Y$, however computing the largest possible means finding a maximum clique, that is NP-hard.

After obtaining $X'$ from $X$ and $Y$, $X$ is discarded as it is now included in $X'$, and the process is repeated as long as new LACCIS’s can be created.

After the RECOMBINE phase cannot generate new LACCIS’s, FILTER is applied again to remove redundant and partially overlapping isomorphisms, if any. We refer to the sequence of operations FILTER, RECOMBINE, FILTER as process. The resulting isomorphisms identify LACCIS’s composed of parts of the $T$-mccis’s for $T = T_1, \ldots, T_k$. This is the final output of FLaSH.

### 9.4 Experimental evaluation

#### 9.4.1 From proteins to graphs

FLASH can bring benefits when modeling proteins as graphs since higher resolution can be exploited. Current approaches benefit from a reduced computational load as they use coarse-grained models. For example, the 3D patterns of secondary structure elements in proteins have been modeled as graphs by using secondary structure elements, such as the $\alpha$-helices and the $\beta$-strands, as nodes. They are approximately linear structures and they are represented as vectors in space, sometimes annotated with the length of their residues and hydrophobicity. As for the edges, they represent relationships between nodes expressed in terms of the angles and the distance between midpoints of the corresponding vectors [VBWDG+13]. In another representation nodes are represented similarly, but edges are calculated on the basis of contacts between the atoms belonging to the respective structures/nodes, and indicate the spatial arrangements of the structures. In this way structural patterns can be also found in proteins with weaker similarities [KLLW96]. We refer the reader to Table 1 in [VBWDG+13] for a list of applications.

We think that exploring fine-grained models with FLaSH, which was precluded with previous algorithms, can give finer details once data noise is filtered. However the design and validation of a finer-grained model is outside the scope of this chapter, and deserves further independent study.
Consider Fig. 9.4 which shows an example of the coarse-grained model adopted in [KLV96]. Indeed, note how the two proteins, which would have respectively 2763 (9488) and 3841 (12923) nodes (edges) in the all-atom representation, are reduced to graphs which are orders of magnitude smaller. While patterns in such graphs can indeed be valuable, the loss of information may be significant.

We created a stress test with an all-atom fine-grained model for generating graphs from proteins from the PDB (www.rcsb.org). We thus exploited PDB data of 1ald, 1fcb (chain A), and 1gox proteins (which belong to TIM barrel families) to generate graphs where labeled nodes represent atoms within known secondary structures (as reported in PDB) while edges represent covalent bonds (both backbone and non-backbone) as well as non-covalent interactions.

We generated input graphs by means of pdb2graph [Hol15]. First, PDB data is processed to generate edges from covalent bonds. Non-covalent interactions are estimated by extending the interaction distance up to 3.2 Å. Nodes are labeled with the element symbol and a secondary structure identifier according to the PDB data. We thus generated 3 graphs with 2763 (9488), 3841 (12923), and 2696 (9059) nodes (edges) for 1ald, 1fcb, and 1gox respectively. Furthermore, we also considered two variants of a structure extracted from 1ald to test consistency and robustness of FLASH, discussed later.

9.4.2 Experimental results

We describe our experimental results for FLASH. In order to better understand its performance, we analyzed FLASH by considering the aggregated raw result after its output sensitive search, and the post-PROCESS form after filtering and recombining the latter results by the method process. Specifically, we fixed $\tau = 10$ for the threshold on minimal size and $\sigma = 70\%$ for the overlapping threshold of filter (recall that process indicates the sequence filter, recombine, filter).

We chose to run FLASH with $k$ random spanning trees for several values of $k$ and with a set

---

The graphs can be found at [http://ptgl.uni-frankfurt.de/](http://ptgl.uni-frankfurt.de/)
of spanning trees which forms a cover of the graph $G$ (in our test case the number of spanning tree covering $G$ was 5). We refer to the former variant as $k$-FLASH, with $k = 1, 3, 6, 12$, and to the latter as $c$-FLASH. As for the raw result, FLASH runs $k$ threads, one for each spanning tree, which (are forced to) terminate within a fixed time $t$, and then aggregates their results. We let each thread run for at most $t/k$ hours, so that the bound for the overall CPU time is the same for all runs, with $t = 12$ hours.

Following the discussion in the introduction, the baseline for the comparison of FLASH is Koch’s algorithm [Koc01], which produces LACCIS’s using MCCIS and is the best algorithm known so far for MCCIS [Wel11]. For a fair comparison, we optimized its implementation, denoted KOCH, so that it can use the implicit product graph $P$ as well, noting that this optimization greatly improves its performance [Ver15]; moreover, the computation is terminated after fixed time $t = 12$ hours. We remark that the RECOMBINE step has no effect on KOCH, whose output is made of MCCIS’s that cannot be enlarged, and thus its additional time is negligible.

The above framework has been implemented in C++. Our computing platform is a 24-core machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz, with 128GB of shared memory. The operating system is Ubuntu 14.04.2 LTS, with Linux kernel version 3.16.0-30.

For each pair of graphs in Table 9.1, we report the real execution time, that is the time (bounded by $t/k$ hours for RAW) of the threaded execution PAR, and the total CPU time WORK (bounded by $t = 12$ hours for RAW). Note that WORK of FLASH for RAW is less than $t$ in all the cases as almost all the threads terminate earlier than the time limit $t/k$. We also report some analysis of the results, before and after applying PROCESS, in columns RAW and post PROCESS.

---

### Table 9.1: Experimental results

<table>
<thead>
<tr>
<th>METHOD</th>
<th>TIME (h.m)</th>
<th>PAR</th>
<th>WORK</th>
<th>MAX</th>
<th>H-IND</th>
<th>COUNT</th>
<th>TIME (h.m)</th>
<th>PAR</th>
<th>WORK</th>
<th>MAX</th>
<th>H-IND</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-FLASH</td>
<td>0:12</td>
<td>0:13</td>
<td>63</td>
<td>59</td>
<td>921518</td>
<td>2</td>
<td>0:01</td>
<td>0:06</td>
<td>70</td>
<td>39</td>
<td>17668</td>
<td></td>
</tr>
<tr>
<td>3-FLASH</td>
<td>0:12</td>
<td>0:25</td>
<td>62</td>
<td>62</td>
<td>2216549</td>
<td>0:31</td>
<td>0:45</td>
<td>179</td>
<td>43</td>
<td>29082</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-FLASH</td>
<td>0:29</td>
<td>1:01</td>
<td>59</td>
<td>58</td>
<td>47329927</td>
<td>0:24</td>
<td>1:05</td>
<td>155</td>
<td>48</td>
<td>41080</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12-FLASH</td>
<td>0:30</td>
<td>2:28</td>
<td>70</td>
<td>69</td>
<td>106735973</td>
<td>15:41</td>
<td>17:27</td>
<td>229</td>
<td>53</td>
<td>55231</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c-FLASH</td>
<td>0:13</td>
<td>0:40</td>
<td>55</td>
<td>54</td>
<td>38315376</td>
<td>0:09</td>
<td>0:45</td>
<td>118</td>
<td>42</td>
<td>41410</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KOCH</td>
<td>12</td>
<td>12</td>
<td>63</td>
<td>63</td>
<td>1297231</td>
<td>&lt;0:01</td>
<td>&lt;0:01</td>
<td>63</td>
<td>24</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-FLASH</td>
<td>0:32</td>
<td>1:53</td>
<td>68</td>
<td>68</td>
<td>60120966</td>
<td>4:30</td>
<td>5:10</td>
<td>49</td>
<td>6</td>
<td>26954</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KOCH</td>
<td>12:00</td>
<td>12:00</td>
<td>64</td>
<td>64</td>
<td>4775963</td>
<td>&lt;0:01</td>
<td>&lt;0:01</td>
<td>64</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-FLASH</td>
<td>2:08</td>
<td>8:18</td>
<td>151</td>
<td>151</td>
<td>144658776</td>
<td>0:13</td>
<td>1:08</td>
<td>151</td>
<td>47</td>
<td>26657</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KOCH</td>
<td>12:00</td>
<td>12:00</td>
<td>82</td>
<td>82</td>
<td>4142149</td>
<td>&lt;0:01</td>
<td>&lt;0:01</td>
<td>82</td>
<td>25</td>
<td>158</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-FLASH</td>
<td>2:01</td>
<td>9:00</td>
<td>171</td>
<td>170</td>
<td>58925057</td>
<td>0:07</td>
<td>0:25</td>
<td>171</td>
<td>38</td>
<td>6561</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KOCH</td>
<td>12:00</td>
<td>12:00</td>
<td>60</td>
<td>60</td>
<td>197236</td>
<td>&lt;0:01</td>
<td>&lt;0:01</td>
<td>60</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-FLASH</td>
<td>0:10</td>
<td>0:18</td>
<td>162</td>
<td>160</td>
<td>6876538</td>
<td>0:02</td>
<td>0:03</td>
<td>162</td>
<td>35</td>
<td>7592</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KOCH</td>
<td>12:00</td>
<td>12:00</td>
<td>60</td>
<td>60</td>
<td>81884</td>
<td>&lt;0:01</td>
<td>&lt;0:01</td>
<td>65</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

4As already discussed, the backtracking algorithms for the maximum common subgraph, which are faster than MCS and can operate on large graphs [CFV07], employ a cutting rule unsuitable for LACCIS’s.
respectively. For each result set, we show the size of the greatest LACCIS in this set (i.e., MAX), the maximum $h$ such that there are at least $h$ LACCIS’s of size $h$, (i.e., H-IND), and the number of LACCIS’s found (i.e., COUNT).

**On the choice of the spanning trees.** Referring to the upper part of Table 9.1 given the pair of graphs 1ald and 1fcb, we compare the results of our $k$-FLASH for different values of $k$ and of $c$-FLASH. It is worth observing (RAW column) that the number of $T$-MCCIS’s found increases with the number of spanning trees used, recalling that $c$-FLASH uses 5 spanning trees, 6-FLASH and 12-FLASH produce a higher number of $T$-MCCIS’s. After the post-processing, $c$-FLASH and 6-FLASH produce a similar number of LACCIS’s, while 12-FLASH produces a larger number of LACCIS’s but at the price of an higher post-processing time. For these reasons, we decided to focus on 6-FLASH in the remaining experiments in this section.

**Running the Experiments.** For the following pairs of graphs, 1ald vs 1fcb, 1ald vs 1goy, and 1fcb vs 1goy, we report the results for both KOCH and 6-FLASH. We remark how our algorithm, though heuristic, finds in this given time slot more LACCIS’s than KOCH, whose result set is in theory complete. Moreover, it seems that FLASH is able to find larger LACCIS’s than KOCH, as shown in the post PROCESS columns, where LACCIS’s found by FLASH are greatly enlarged. Furthermore, it is clear that KOCH focuses the search on a limited portion of the graph, while FLASH is able to produce many more LACCIS’s that do not overlap with each other (see COUNT in post PROCESS). For instance, consider the 1ald vs 1goy comparison. Even though KOCH finds 4775963 LACCIS’s, after PROCESS we are left with just 6, of size at most 64: this means that all the remaining LACCIS’s found by KOCH overlap with these 6 by at least $\sigma = 70\%$. This is not the case with FLASH, which obtains 26954 LACCIS’s after PROCESS. Even though our post PROCESS time is greater, this is compensated by the quantity and quality of our results, as well as the smaller running time of the algorithm.

**Consistency and robustness.** To test the consistency and quality of the results, we extracted a portion of the protein 1ald (from Pro158 to Asn180), including an $\alpha$-helix, and searched for it in the original protein. The corresponding subgraph has 171 nodes and 584 edges. Clearly, an effective algorithm must find at least a LACCIS which involves a large portion of the helix. Table 9.1 (HelixD-1ald vs 1ald) shows that our algorithm finds a LACCIS involving the whole helix (171 nodes), while this is not the case for KOCH in the time slot.

For the robustness, we introduced errors in the helix: we changed the labels of the alpha carbon atoms of Arg172 and Asn166 to a dummy label ‘X’. We refer to this modified graph as mod-HelixD-1ald. A robust algorithm should not be significantly influence by the introduced noise, and should find results similar to the ones obtained with HelixD-1ald. The lower part of Table 9.1 i.e. mod-HelixD-1ald vs 1ald, shows that once again our algorithm finds almost the whole helix, while KOCH, although consistent with the previous result, still finds just a small portion of the helix.

### 9.5 Final remarks

Finding all maximal common connected induced subgraphs between two graphs is computationally challenging. The proposed approach improves significantly upon simple adaptations of the Bron-Kerbosch algorithm. The approach aims at making the search for common subgraphs feasible for larger graphs. Even though some subgraphs can be missed, omitting them does not affect in practice the quality of the result. As supported by the experimental results, the algorithm is able to find a relatively small set of result which “dominates” all solutions found, and thus is able to present a much more concise yet comprehensive result to the user.

We also show that while Bron-Kerbosh-based approaches tend to process a local area fully
before proceeding to the next, our approach quickly covers different areas of the input networks, which may be a more suitable strategy under time constraints.

This approach may allow higher quality analysis of large molecules, by enabling researchers to find all significant common structures between two molecules rather than just the largest one.
Finding maximal cliques is a widely studied problem in network analysis as discussed, e.g., in Chapter 3. While the number of cliques in real-world graphs is usually much smaller than the theoretical maximum, this may still be very large. To remove redundancy, one may want a smaller set of cliques, such that each clique is not redundant, i.e., contains some information not present in any other, but at the same time no information from the input graph is ignored. We consider, as a possible solution, the edge clique covering (ECC) problem: this problem deals with discovering a set of (possibly overlapping) cliques in a given network, such that each edge is part of at least one of these cliques. We address the ECC problem from an alternative perspective reconsidering the quality of the cliques found, and proposing more structured criteria with respect to the traditional measures such as minimum number of cliques. In the case of real-world networks, having millions of vertices, such as social networks, the possibility of getting a result is constrained to the running time, which should be linear or almost linear in the size of the network. Our algorithm for finding ECCs of large networks has linear-time performance in practice, as our experiments show on real-world networks whose number of vertices ranges from thousands to several millions.

10.1 Introduction

The usage of cliques and similar dense subgraphs such as $n$-cliques, $n$-clans, $k$-plexes, $k$-cores, $f$-groups, in social network analysis is now consolidated and many methods exist to discover them under several contexts [WF94]. A clique is a subset of the vertices in a network such that every two vertices are connected. Among the variations of clique discovering, the edge clique covering (ECC) problem deals with finding a set of (possibly overlapping) cliques in a given network, such that each edge of the network is part of at least one of these cliques.

In this chapter, we propose new algorithmic tools for ECC that are potentially useful for social network analysis and, at the same time, are very easy to implement. Our tools can quickly process graphs of massive size to find their ECC in practically linear time and use heuristics for attacking the ECC problem, which was born in a different context [Kel73] and is hard to solve exactly.

Previous work In the literature, problems related to ECC have been independently addressed in the mid 70’s in social network analysis [Ber77, Rob76]. From a computational point of
view, its vertex clique covering version, where the cliques are disjoint and the vertices are to be covered, is one of Karp’s original problems shown to be NP-complete in his famous 1972 paper. The ecc appeared with different terminology such as keyword conflict [KG73], covering by cliques, intersection graph basis, or clique partition [GJ79], as discussed in [CPP13]. The latter reports that the problem of finding the ecc having the minimum number of cliques is NP-hard [Ori77], even for planar graphs [CM01]. The parameterized exact algorithms at the state of the art [GGHN09] are probably optimal [CPP13]. Although they can be applied to social network analysis, they do not scale to large networks.

Some heuristic algorithms for cliques work surprisingly well for large real-world networks (e.g. [APR99, AKK73, CKEF11, ELSS13, KGGP11]), as even approximation algorithms with some guarantee are too expensive. Indeed, due to the massive size of available real-world large networks, it is convenient to consider algorithms or heuristics taking (almost) linear time in the size of the network. Some papers try to achieve this goal for ecc.

The state of the art in this direction is Gramm et al. [GGHN06], which gives an efficient implementation of the original Kellerman heuristic [Kel73], with the post-processing step of Kou et al. [KSW78]. However, their bounds are not linear. For a network with $n$ vertices and $m$ edges, the original results by Kellerman finds a clique covering in $O(nm^2)$ time. The post-processing of Kou et al. requires at least $\Omega(nm)$ time in the worst case. Gramm et al. show how to obtain the same heuristic solution in total $O(nm)$ time, using $O(n^2)$ cells of memory as space.

**Our Results** In this chapter, we present some new and simple algorithmic tools for heuristics on ecc that are tailored for large real-world networks. The classical well-studied measure is minimizing the number of cliques. Recently also the minimum assignment objective has been proposed, which is minimizing the sum of the sizes of the cliques [EFE12]. In addition to them, we consider an alternative view to the ecc problem, where more options are available.

Our view is well represented by the clique cover graph shown in Figure 10.2 for the example network in Figure 10.1. This bipartite graph has vertices representing the cliques of the ecc on the left, and vertices representing the vertices of the input network on the right. There is an edge to indicate that a given vertex is part of the given clique. On the right, we may represent the edges of the network rather than the vertices. A formal definition is given in Section 10.2. This graph is reminiscent of the clique representation in intersection graph theory [EGP66].

Looking at the classical measure of minimizing the number of cliques in the ecc, corresponds to having the number of vertices on the left side of the clique cover graphs of the candidate eccs. On the other hand having a minimum assignment, as in [EFE12], corresponds to minimizing the number of edges in this graph. However, more information on the quality of the solution can be obtained from the cover graph, which suggests using also other measures when evaluating heuristics for the ecc problem. For example, the weight of an ecc, the clique size distribution, or the cover index distribution described in Section 10.2.

Our contributions in this chapter are summarized below.

First, we introduce an experimental analysis of heuristics for the ecc problem on real-word networks that is based on a more informative evaluation tool, based on the clique cover graph, rather than the mere number of cliques in ecc. In other words, we do not look just at the minimization problem but we consider also several statistics on all the cliques emerging from an ecc, based on these networks.

Second, we propose a new efficient and simple framework for fast algorithms that scale well and perform well according to a variety of measures based on the clique cover graph, when compared to the state of the art. They require $O(m + n)$ space and their time cost is given at the end of Section 10.3 and can be upper bounded as $O(m \log \Delta + k \Delta d)$, where $k \leq m$ is the

---

*Finding an arbitrary ecc is easy, for example the trivial one made up of just the individual edges.*
number of cliques in the ECC, $\Delta < n$ is the maximum degree of the graph, and $d \leq \Delta$ is the degeneracy of the graph. This cost becomes $O(m\Delta)$ in some cases, thus improving performance, i.e. time and space needed to get the solution. Furthermore, we show experimentally that the running time is in practice linear in $m$ for the graphs considered.

Finally, we perform an experimental study for our framework on a set of real-world and synthetic networks (see Table 10.3). The comparison among 20 variants of our algorithms and the state of the art involved a data set of 44 networks to select the best variants. We adopted some measures based on the clique cover graph, and compared with the state of the art on an extensive data set.

We adopted some measures based on the clique cover graph, and compared with state of the art on an extensive data set. Our algorithms improve in practice the quality of the solution, also according to the traditional measures such as the minimum number of cliques in ECC.

In particular, we show that ecc-rc, one of our most effective algorithms, is able to outperform the state of the art with respect to both performance and quality of the result, also according to the traditional measures such as the minimum number of cliques in ECC. We test ecc-rc on 160 networks, showing that we can efficiently deal with large real-world data.

Some insight to our algorithms for the ecc problem has been borrowed from enumeration algorithms. In particular the Bron-Kerbosch algorithm [BK73] (for short, bk algorithm) is a popular algorithm for maximal clique enumeration, whose variants have been shown to achieve the best results in time and space for sparse graphs [ELS13, TTT06, SSTP09].

We hope that these findings can inspire further work on network analysis that uses clique cover graphs, as the latter ones could be customized to deal with new quality measures that depend on the domain applications.

### 10.2 Clique covering

We consider an undirected connected graph $G = (V(G), E(G))$. Refer to Section 2.1 for terminology. A clique $C$ is maximal if there is no other clique $C'$ such that $C \subset C'$, and trivial if it is maximal and a single edge (i.e. $|C| = 2$).

An edge clique cover for $G$ (for short, ecc) is a set of cliques $C_1, C_2, \ldots, C_k$ such that (1) no clique $C_i$ is contained in another clique $C_j$, where $i \neq j$, and (2) for each edge $\{u, v\} \in E$ there is at least one clique $C_i$ such that $\{u, v\} \in C_i$. [KSW78, GGHN09] Note that an ecc always

---

Other definitions exist in the literature. In vertex clique cover, condition (2) is replaced by asking that for each vertex $v$ there is a clique $C_i$ such that $v \in C_i$. In clique partition, the cliques $C_1, C_2, \ldots, C_k$ form a partition of $V$. Note that these definitions lose information regarding the edges of $G$, while the ecc, on which we focus, contains all the information necessary to reconstruct $G$. 

---

Figure 10.1: Graph (i) taken from [KSW78], and two examples of ECCs in (ii) and (iii).
exist (e.g. choose \( k = m \) and the individual edges as cliques). Maximality on each \( C_i \) is not required in the definition of ecc, but it is not difficult to see how to transform each \( C_i \) into a maximal clique, if this is needed, by greedily expanding each non-maximal clique. An ecc is minimal if there is no clique contained in the union of the others, namely, it is \( C_i \not\subseteq \bigcup_{j \neq i} C_j \) for \( 1 \leq i \leq k \). The clique number of \( G \) is the smallest \( k \) such that an ecc of \( k \) cliques exists for \( G \). Figure [10.1] illustrates our definitions.

Beside the possible definitions of cliques, the quality of a solution, i.e. a clique set, has been often measured as its cardinality, that is, the quality of the ecc \( C_1, \ldots, C_k \) is simply \( k \). Hence, the so called minimum ecc aims to minimize the size \( k \). Although finding a minimum ecc has been already proved to be hard and not arbitrarily approximable, heuristics are often satisfying since in practice the quality of a solution could also depend on the application.

### 10.2.1 Clique cover graph

It is convenient to define a clique cover graph for the given ecc \( C_1, \ldots, C_k \) to better highlight its properties and assess its qualities. It is a bipartite undirected graph \( G' = (V'_1 \cup V'_2, E') \), such that \( V'_1 \) contains a vertex for each clique, hence \( |V'_1| = k \), and \( V'_2 \) contains a vertex for each vertex in \( V \), hence \( |V'_2| = n \). There is an edge \( u_1, u_2 \in E' \), where \( u_1 \in V'_1 \) represents a clique \( C_i \) in the ecc and \( u_2 \in V'_2 \) represents a vertex \( v \in V \), if and only if \( v \in C_i \). An example is shown in Figure [10.2].

Depending on the application, we may define a clique cover graph with \( V'_2 \) representing the edges in \( E \), where \( |V'_2| = m \). In general, the choice of \( V'_2 \) representing either the vertices in \( V \) or the edges in \( E \) will be clear from the context.

While traditional work aims at minimizing the cardinality of \( V_1 \), namely, finding the ecc of minimum cardinality \( k \) (or giving an approximation of it), in this chapter other possibilities are offered. Not only we can obtain the measures previously known in the literature by looking at the clique cover graphs for the possible eccs of \( G \), but we can also define some additional ones in a natural and smooth way.

1. Finding the ecc of minimum weight \( \sum_{i=1}^{k} |C_i| \) is equivalent to finding the ecc whose clique cover graph has the minimum number of edges (see also [EFE12]).
2. Finding the clique size distribution in the given ecc is equivalent to the degree distribution of the vertices in \( V'_1 \) for the corresponding clique cover graph.
3. Finding the covering index distribution in the given ecc, namely, computing the number \( y \) of elements from \( V'_2 \) that are contained in \( x \) cliques of the ecc, for all feasible values of \( x \) and \( y \), is equivalent to the degree distribution of the vertices in \( V'_2 \) for the corresponding clique cover graph.

We devote a significant part of the chapter to an experimental study of the properties of the clique cover graph in real-world networks.

### 10.3 Framework and variants

Our general approach relies on a simple algorithmic framework, summarized in Algorithm 29, which takes a graph \( G = (V, E) \) as input. It begins with an empty ecc \( C \), and during the steps, cliques are added to \( C \). At any step the edges in \( E \) are partitioned as covered and uncovered: edge \( \{u, v\} \) is covered if there exists a clique \( C_i \in C \) such that \( \{u, v\} \in C_i \); it is uncovered otherwise. Initially, all edges in \( E \) are uncovered. As long as there are uncovered edges, one of them is chosen by \texttt{SELECT\_UNCOVERED\_EDGE}, described in Section [10.3.1]. Let us call \( \{u, v\} \) this
edge. We then find a new clique $R$ that contains $\{u,v\}$ using FIND\_CLIQUE\_OF, described in Section 10.3.2. Note that the clique $R$ must be new, as at least $\{u,v\}$ is uncovered. We then add $R$ to $C$, and mark all of the edges $\{x,y\} \in R$ as covered.

The framework resembles that of algorithms for maximal clique enumeration, such as the BK algorithm. To enumerate all maximal cliques, the FIND\_CLIQUE\_OF step needs to look for all the maximal cliques containing the selected edge and no previously selected edges, while the last step marks just the selected edge as covered. In the case of the BK algorithm this is implemented by using a backtracking approach that recursively tries to enlarge a clique picking vertices from a suitable candidate set. In our case, however, the purpose of FIND\_CLIQUE\_OF is to find only one clique; this is implemented by enlarging the clique in the same way as BK, but no backtracking is performed as the algorithm terminates when the first clique is found.

We study several variants of this framework that differ from the way they implement the SELECT\_UNCOVERED\_EDGE and FIND\_CLIQUE\_OF operations.

### 10.3.1 Operation SELECT\_UNCOVERED\_EDGE

Given a vertex $u \in V$, we denote its set of neighbors by $N(u) = \{v \mid \{u,v\} \in E\}$ and its degree by $d(u) = |N(u)|$. We also denote the set of uncovered edges by $U \subseteq E$, and define the uncovered neighbors of $u \in V$ as $N_U(u) = \{v \mid \{u,v\} \in U\}$. Consequently, the uncovered degree of $u$ is $d_U(u) = |N_U(u)|$. When $d_U(u) > 0$, we call $u$ eligible.

We consider three variants of SELECT\_UNCOVERED\_EDGE, denoted by $r$, $m$, $M$, $U$, assuming that the edge set $U$ is nonempty, as shown in Table 10.1.

```
Algorithm 29: The general framework

Input: A graph $G(V,E)$

Output: A covering $C$ of $G$

All edges in $E$ are marked as uncovered

$C \leftarrow \emptyset$

while there are uncovered edges do

$\{u,v\} \leftarrow$ SELECT\_UNCOVERED\_EDGE()

$R \leftarrow$ FIND\_CLIQUE\_OF($\{u,v\}$)

$C \leftarrow C \cup R$

Mark all edges of $R$ as covered
```

179
**random**: return an uncovered edge \{u, v\} from \(U\) uniformly at random.

**min-degree**: choose an eligible vertex \(u\) of minimum degree \(d(u)\), and return an arbitrary edge \{u, v\} \(\in U\) (i.e. \(v \in N(u)\)).

**max-degree**: as above, except that \(u\) has maximum degree \(d(u)\).

**max-uncovered-degree**: choose an eligible vertex \(u\) of maximum uncovered degree \(d_U(u)\), and return an arbitrary edge \{u, v\} \(\in U\).

<table>
<thead>
<tr>
<th>Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>random</strong></td>
</tr>
<tr>
<td><strong>min-degree</strong></td>
</tr>
<tr>
<td><strong>max-degree</strong></td>
</tr>
<tr>
<td><strong>max-uncovered-degree</strong></td>
</tr>
</tbody>
</table>

Table 10.1: Variants of `SELECT_UNCOVERED_EDGE`

### 10.3.2 Operation `FIND_CLIQUE_OF`

Starting out from the given uncovered edge \{u, v\} in the input graph \(G\), we can find a clique \(R\) containing \{u, v\} by following the idea of the Bron-Kerbosh technique as shown in Algorithm 30.

After initializing \(R\) with \{u, v\}, the candidate set \(P\) is built by intersecting the neighbors of \(u\) and \(v\). A vertex \(z\) is suitably extracted from \(P\), if it exists (i.e. \(z \neq \text{null}\)), and added to \(R\) as \(z\) is surely adjacent to all the vertices in \(R\). Since \(z\) is now part of \(R\), the vertices in the candidate set \(P\) must be also neighbors of \(z\), so \(P\) is updated with the intersection with \(N(z)\). In general, each of the vertices in \(P\) is a neighbor of all the vertices in \(R\). The expansion of \(R\) terminates when either \(P\) is empty (and thus \(R\) is maximal) or \(z\) cannot be found in \(P\): in both cases it is \(z = \text{null}\).

The descriptions of the variants of `EXTRACT_NODE(P)` shown in Table 10.2 explain why \(z\) can sometimes be \(\text{null}\) even in a nonempty \(P\). If \(P\) is empty, all variants return \(z = \text{null}\), so the descriptions assume without loss of generality that \(P\) is nonempty.

The pivoting variant \(p\) is inspired by the result of Tomita et al. [TTT06], originally aimed at minimizing the number of calls to the BK algorithm. In our scenario we use it as a greedy heuristic to find a locally large clique: adding \(v\) to the current result set will maximize the size of \(P\) at the next step, as \(P\) will only retain the neighbors of \(v\) in \(P\). The clean variant \(c\) aims at maximizing the number of uncovered edges that will become covered after adding \(z\) to the current clique \(R\). The semi-clean variant \(s\), in case that the clean variant

<table>
<thead>
<tr>
<th>Algorithm 30: FIND_CLIQUE_OF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A graph (G(V, E)), an uncovered edge {u, v} (\in E)</td>
</tr>
<tr>
<td><strong>Output:</strong> A clique (R) containing {u, v}</td>
</tr>
<tr>
<td>(R \leftarrow {u, v})</td>
</tr>
<tr>
<td>(P \leftarrow N(u) \cap N(v))</td>
</tr>
<tr>
<td>(z \leftarrow \text{EXTRACT_NODE}(P))</td>
</tr>
<tr>
<td><strong>while</strong> (z \neq \text{null}** <strong>do</strong></td>
</tr>
<tr>
<td>(R \leftarrow R \cup {z})</td>
</tr>
<tr>
<td>(P \leftarrow P \cap N(z))</td>
</tr>
<tr>
<td>(z \leftarrow \text{EXTRACT_NODE}(P))</td>
</tr>
</tbody>
</table>
random: return a vertex \( z \) from \( P \) uniformly at random.

pivoting: return any vertex \( z \) with maximum \(|N(z) \cap P|\) (as in [TTT06]).

clean: return any vertex \( z \) (if any) with maximum \(|N_U(z) \cap R| > 0\); otherwise, return \( z = \text{null} \).

semi-clean: return any vertex \( z \) (if any) with maximum \(|N_U(z) \cap R| > 0\); else, return any vertex \( z \) (if any) with maximum \(|N_U(z) \cap P| > 0\); otherwise, return \( z = \text{null} \).

dirty: return any vertex \( z \) (if any) with maximum \(|N_U(z) \cap R| > 0\); else, return any vertex \( z \) (if any) with maximum \(|N_U(z) \cap P| > 0\); otherwise, return any vertex \( z \) in \( P \).

Table 10.2: Variants of EXTRACT\_NODE

fails, attempts to make the clean variant successful at the next call, as the candidate set \( P \) will meet the condition. Indeed, adding \( z \) to \( R \) will cause a vertex in the resulting \( P \) to have uncovered edges to the resulting \( R \) (as there is at least one that has has an uncovered edge to \( z \)). Finally, the d variant has the same aim as s, but never returns \( z = \text{null} \) (unless \( P \) is empty).

Remark 10.1. Recall that we are proposing a heuristic for a hard problem. For example, the traditional task of minimizing the number of cliques could benefit by a variant of EXTRACT\_NODE (\( P \)) that finds a vertex in the largest maximal clique containing \( P \), but unfortunately the latter problem is NP-hard. We therefore propose variants that are fast to compute.

In the following, we use the notation \( \text{ecc-xy} \) to denote the variant of our framework given in Algorithm 29, where \( x \in \{r,m,M,U\} \) encodes one of the edge selection variants described in Table 10.1, and \( y \in \{r,p,c,s,d\} \) encodes one of the vertex extraction variants described in Table 10.2. For example, \( \text{ecc-rc} \) selects an uncovered edge \( \{u,v\} \) in a uniformly random fashion, and then it finds the clique containing \( \{u,v\} \) by extracting each time the vertex \( z \) (if it exists) with the maximum number of uncovered edges to \( R \). Hence, we have a total of 20 variants of our framework, and we expect just a subset of them to be effective for \( \text{ecc} \) (see Table 10.3).

10.4 Implementation and complexity

We give some details on the \( \text{eccs} \) found by the variants of our algorithmic framework, and their implementation and analysis of costs. We first show that these variants correctly compute an \( \text{ecc} \) as defined in Section 10.2.

Lemma 10.1. All the variants of Algorithm 29 correctly compute an \( \text{ecc} \) of the input graph \( G \).
Proof. Let \( \text{ecc-xy} \) be any variant of Algorithm 29 where \( x \in \{r, m, M, U\} \) and \( y \in \{r, p, c, s, d\} \). Let \( C_1, C_2, \ldots, C_k \) be the cliques of the ecc in their order of discovery by \( \text{ecc-xy} \), namely, \( C_i \) is discovered before \( C_j \) iff \( i < j \). We have to prove that (1) no clique \( C_i \) is contained in another clique \( C_j \), where \( i \neq j \), and (2) for each edge \( \{u, v\} \in E \) there is at least one clique \( C_i \) such that \( \{u, v\} \in C_i \). The latter point is easily met by \( \text{ecc-xy} \) as the while loop in Algorithm 29 terminates when all edges are covered.

We thus focus on point (1). Suppose by contradiction that \( C_i \subseteq C_j \) for \( i \neq j \). First, we observe that it cannot be \( C_i = C_j \) as they both must be found from one uncovered edge, and the first discovered of the two would cover all the edges of the other. Thus, it must be \( C_i \subset C_j \). Second, it must be \( i < j \) for the same reason: if it were \( j < i \), then \( C_j \) would contain only covered edges.

Now we get a contradiction for different reasons.

When \( y \in \{r, p, d\} \), the cliques in their eccs are all maximal. Indeed, a vertex \( z \in P \) is returned as long as \( P \) is nonempty, and the clique found is maximal when \( P \) is empty since if it could be incremented with a vertex, that vertex would be in \( P \). As by definition a maximal clique cannot be part of a larger clique, we have \( C_i \not\subseteq C_j \) which contradicts what previously found.

As for \( y = s \), recall that \( C_j \) must contain at least one uncovered edge in \( C_j \setminus C_i \). Since \( C_j \setminus C_i \subseteq P_i \), one endpoint of that edge would have been chosen as \( z \) in the while loop for \( C_i \), thus further extending \( C_i \) by at least one vertex. This is a contradiction as \( C_i \) was returned as it could not be further extended.

Finally, when \( y = c \), returning \( C_i \) means that all the edges having one endpoint in \( C_i \) and the other endpoint in \( C_j \setminus C_i \) are covered, otherwise \( C_i \) could be expanded. Furthermore, all edges in \( C_j \) are then covered. Thus \( C_i \) must be discovered from an edge with both ends in \( C_j \setminus C_i \). A moment of reflection can pinpoint the fact that it is impossible to reach \( C_i \)'s vertices from the endpoints \( u \) and \( v \) (of that uncovered edge) using other uncovered edges during the execution of \( \text{FIND_CLIQUE_OF}(\{u, v\}) \) in \( \text{ecc-xy} \), as there is no clean edge between \( C_j \setminus C_i \) and \( C_i \). Thus it is a contradiction that \( C_i \) is contained in \( C_j \).

\[ \square \]

Remark 10.2. Looking at the above proof, we can see that \( k \), the number of cliques in the covering, is always at most \( m \). Moreover, \( \text{ecc-xy} \) discovers maximal cliques when \( y \in \{r, p, d\} \), while this is not necessarily true when \( y \in \{c, s\} \). Note that the resulting ecc is not necessarily minimal, but we can apply the method in \cite{KSW78} to make it so.

We now consider how to implement \( \text{SELECT_UNCOVERED_EDGE} \) efficiently in \( \text{ecc-xy} \) (see Section 10.3.1). We store the set \( U \) of uncovered edges as an unordered array, which supports random access and deletion. Both operations can be performed in constant time with some bookkeeping: each edge in \( U \) knows its position in the array and, whenever it is selected to be marked as covered and deleted from \( U \), its position is filled with the last entry in the array, and the array size conceptually decreases by 1.

The above is enough to implement variant \( x = r \) in constant time. As for the other variants \( x \in \{m, M, U\} \), we store the set of eligible vertices in at most \( \Delta \) unordered lists \( L_1, L_2, \ldots, L_\Delta \), where \( \Delta = \max_{u \in V} d(u) \) is the graph’s maximum degree. Specifically, eligible vertex \( u \) with \( d(u) = i \) is stored in \( L_i \). A similar organization is maintained also for the uncovered degree \( d_U(u) \) of each eligible vertex \( u \). With some bookkeeping, it takes constant time to be removed from a list or change list for \( u \).

For variants \( x \in \{M, U\} \), we can keep the largest \( i \) such that \( L_i \) is nonempty. Since eligible vertices can be only deleted or moved to lists with lower indices, the amortized cost is constant for each change. For variant \( x = m \), we maintain a priority queue on the values \( i \) such that \( L_i \) is nonempty, and each operation takes \( O(\log \Delta) \) time in the worst case (actually time can be
lowered with more sophisticated data structures but in practice this is enough. We thus have the following cost.

Lemma 10.2. Operation SELECT_UNCOVERED_EDGE in ECC-xy can be implemented in linear space, taking (amortized) constant time for variants \(x \in \{r, m, u\}\) and \(O(\log \Delta)\) time for variant \(x = m\), where \(\Delta\) is the graph’s maximum degree.

We now discuss how to implement FIND_CLIQUES_OF efficiently in ECC-xy (see Section 10.3.2). Operation EXTRACT_NODE(P) takes constant time by storing \(P\) as an unordered array with random access and deletion with variant \(y = x\) (see above for \(x = r\)). For the other variants, we observe that the sizes of \(P\) and \(U\) decrease while that of \(R\) increases, thus the values of \(|N(z) \cap P|\) and \(|N_U(z) \cap P|\) decrease and that of \(|N_U(z) \cap R|\) can change. For these values, we thus use unordered lists \(L_1, L_2, \ldots, L_4\) as discussed above for the variants \(x \in \{m, M, U\}\), where \(\delta = \min\{d(u), d(v)\} \leq \Delta\) and \(\{u, v\}\) is the uncovered edge on which FIND_CLIQUES_OF is launched. We maintain priority queues at the cost of \(O(\log \delta)\) time per query or deletion. Consequently, EXTRACT_NODE(P) takes constant time for variant \(y = p\), and \(O(\log \delta)\) time for the remaining variants \(y \in \{c, s, d\}\). As a result, we have the following cost, since both \(|P|, |R| \leq \delta\).

Lemma 10.3. Operation FIND_CLIQUES_OF(\(\{u, v\}\)) in ECC-xy can be implemented to find the clique \(R\) in linear space, taking \(O(|R| \min\{d(u), d(v)\})\) time.

Proof. Looking at Algorithm 30, let \(\delta = \min\{d(u), d(v)\}\). The initialization of \(R\) takes constant time and that of \(P\) takes \(O(\delta \log \delta)\) time. As we saw, the extraction of \(z\) and its addition to \(R\) take \(O(\log \delta)\) time, repeated for \(|R| - 2\) times, thus giving a cost of \(O(|R| \log \delta)\) time. The cost of updating \(P\) with \(N(z)\) take \(O(\delta)\) time, and is repeated for \(|R| - 2\) times, thus giving a cost of \(O(|R| \delta)\) time, which is the dominant cost of FIND_CLIQUES_OF.

Finally we can give an upper bound to the cost of ECC-xy in our algorithmic framework.

Theorem 10.1. For an undirected graph with \(n\) vertices and \(m\) edges, Algorithm 29 finds an ECC \(C_1, C_2, \ldots, C_k\) in \(O(m + n)\) space and \(O(m \log \Delta + \sum_{i=1}^{k} |C_i| \min\{d(u_i), d(v_i)\})\) time \((k \leq m)\), where \(u_i, v_i\) is the uncovered edge leading to the discovery of clique \(C_i\), and \(\Delta\) is \(G\)’s maximum degree.

Proof. Since each \(C_i\) corresponds to at least one distinct uncovered edge, it is \(k \leq m\). We pay \(O(m \log \Delta)\) time to maintain the set of uncovered edges and choose one of them, by Lemma 10.2. We also pay \(O(|C_i| \min\{d(u_i), d(v_i)\})\) time by Lemma 10.3 for \(1 \leq i \leq k\).

The cost in Theorem 10.1 can be upper bounded using the degeneracy \(d_G\) of the graph \(G\), defined as the smallest \(g\) such that every subgraph of \(G\) has a vertex of degree at most \(g\) (equivalently, the maximum among the smallest degrees in the induced subgraphs of \(G\)). Note that \(d_G \leq \Delta\), and that \(|C_i| \leq d_G\) and that \(\min\{d(u_i), d(v_i)\} \leq \Delta\), thus giving the following corollary.

Corollary 10.1. A crude upper bound to the time cost of ECC-xy is \(O(m \log \Delta + k \Delta d_G)\), where \(d_G \leq \Delta\) is the degeneracy of the graph \(G\). This cost becomes \(O(m \Delta)\) for variant \(y = c\).

Proof. We can bound the sum \(\sum_{i=1}^{k} |C_i| \min\{d(u_i), d(v_i)\}\) by \(O(kd_G \Delta)\), as \(d_G\) is an upper bound for the size of any clique in \(G\). As for variant \(y = c\), each \(C_i\) must contain \(\Omega(|C_i|)\) uncovered edges. Thus \(\sum_{i=1}^{k} |C_i|\) cannot exceed \(O(m)\), and thus we can bound \(\sum_{i=1}^{k} |C_i| \min\{d(u_i), d(v_i)\}\) as \(O(m \Delta)\).
10.5 Experimental evaluation

Interestingly, the heuristic algorithm by Kellerman [Kel73] is still the state of the art, as far as we know, in terms of the traditional quality measure that aims at minimizing the number of cliques found. Gramm et al. [GGHN09] considered also an exact parameterized algorithm to find an ECC with the minimal number of cliques but it does not seem to scale for large graphs. Furthermore, Gramm et al. [GGHN06] show how to improve the performance of Kellerman’s algorithm from $O(m^2 \cdot n)$ to $O(m \cdot n)$, while preserving the same exact result.

We use as a baseline the original Kellerman algorithm [Kel73], with the postprocessing step of Kou et al. [KSW78], and complexity improvements by Gramm et al. [GGHN06], which we call $g$-alg.

Moreover, recent works pointed out the speed and the quality of the solution provided by $g$-alg also with respect to the minimum assignment objective [EFE12]: hence, we think that $g$-alg is a natural choice to evaluate the pros and the cons of the proposed algorithms.

We implemented our variants ecc-$xy$ (Sections 10.3.1–10.3.2) and k-alg in Java 1.8.0_25.

3 The experiments were run on a 24 core machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz, with 128GB of shared memory. The operating system is Ubuntu 14.04.2 LTS, with a Linux kernel version 3.16.0-30.

10.6 Experiments with real-world and synthetic networks

We performed a large-scale preliminary selection of these 20 variants and K-ALG: as shown in Table 10.3, the following variants emerged from the rest and the outcome was mostly independent from how the vertices where permuted in the input graph.

- **ecc-xp**: the pivoting extraction of vertices where the uncovered edge selection uses the variant $x \in \{M, m, U\}$ of max degree, min degree, or max uncovered degree.
- **ecc-xrs**: the semi-clean extraction of vertices where the uncovered edge selection uses the variant $x \in \{M, m, r\}$ of max degree, min degree, or random.
- **ecc-xc**: the clean extraction of vertices where the uncovered edge selection uses the random variant.

10.6.1 Data sets and experimental guidelines

All the networks in our experiments have been collected from the datasets SNAP [LK15] and LASAGNE [LoAoGN]. Real-world networks include autonomous systems, biological, citation, collaboration, communication, word-adjacency, peer-to-peer, social, and web networks. Synthetic networks include networks generated using Erdös-Rényi [ER61], Forest Fire [LFK05], and Kronecker [LCK+10]. For more details about the dataset files we used, we refer the reader to Table 10.5.

Our guidelines for the experiments are based on the clique cover graphs $G’ = (V’ \cup V’_2, E’)$ for the graphs $G = (V, E)$ in the datasets, as described in Section 10.2.1. In particular we considered the following ones to establish the results in Table 10.3.

---

3We also considered the OCaml implementation of g-ALG provided by the authors: while for small networks we obtained results consistent with our implementation, the performance of the original implementation did not allow us to process large networks in our dataset.
Table 10.3: Summary of the best-performing algorithms for each measure

<table>
<thead>
<tr>
<th>measure</th>
<th>goal</th>
<th>best algs</th>
<th>vs g-ALG</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of cliques</td>
<td>MIN</td>
<td>ECC-rc, ECC-ms</td>
<td>12%, 12%</td>
</tr>
<tr>
<td>maximum clique size</td>
<td>MAX</td>
<td>ECC-Mp, ECC-Up</td>
<td>6%, 6%</td>
</tr>
<tr>
<td>average clique size</td>
<td>MAX</td>
<td>ECC-Mp, ECC-mpl</td>
<td>26%, 27%</td>
</tr>
<tr>
<td>coefficient of variation</td>
<td></td>
<td></td>
<td>range: 0.013–0.016</td>
</tr>
</tbody>
</table>

(a) clique size distribution

<table>
<thead>
<tr>
<th>measure</th>
<th>goal</th>
<th>best alg.</th>
<th>vs g-ALG</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum vertex covering</td>
<td>MIN</td>
<td>ECC-Ms</td>
<td>37%</td>
</tr>
<tr>
<td>average vertex covering</td>
<td>MIN</td>
<td>ECC-rc</td>
<td>11%</td>
</tr>
<tr>
<td>coefficient of variation</td>
<td></td>
<td></td>
<td>range: 0.026–0.035</td>
</tr>
</tbody>
</table>

(b) vertex covering index distribution

<table>
<thead>
<tr>
<th>measure</th>
<th>goal</th>
<th>best alg.</th>
<th>vs g-ALG</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum edge covering</td>
<td>MIN</td>
<td>ECC-Ms</td>
<td>87%</td>
</tr>
<tr>
<td>average edge covering</td>
<td>MIN</td>
<td>ECC-rc</td>
<td>23%</td>
</tr>
<tr>
<td>coefficient of variation</td>
<td></td>
<td></td>
<td>range: 0.021–0.026</td>
</tr>
</tbody>
</table>

(c) edge covering index distribution

For the sake of completeness, we report the coefficient of variation, for short CV. The CV is a measure of dispersion of a distribution and it corresponds to the standard deviation divided by the mean. This measure is used as a normalized way to evaluate the variability of a covering (the closer to 0, the better is).

Table 10.3 can be read as follows. The first column indicates the measures described above and the range of values for the CV. The second column indicate whether it is good to minimize or maximize the measure in the corresponding row. The third column reports which algorithms ECC-xy resulted to be the best performers according to the measure in the row. The fourth column reports how they compared to k-alg: a value of $p\%$ for the minimization indicates that...
the measure found by ECC-xy is p\% smaller than that provided by K-ALG; for the maximization, it indicates that the measure is p\% larger than that by K-ALG. As it can be seen, all the algorithms ECC-xy in Table 10.3 provide a solution whose measure outperforms that of K-ALG.

Figure 10.3 reports the plots for distributions (a)–(c) for the graphs as-itdk0304_rlinks (autonomous system network), cit-HepPh (citation network), and email-Enron (communication network). It displays the distribution achieved by K-ALG and the best algorithms reported in Table 10.3. For the sake of clarity, in the case of ties we have decided to show in the plots just one of the options, so that the reported methods are: ECC-ms, ECC-Mp, ECC-Ms, and ECC-rc.

Consistently with Table 10.3, ECC-Mp (green line) achieves more often a greater maximum clique, that is the maximum value on the x-axis such that the line is draw in the plots in the first column of Figure 10.3 (namely, (a-1), (a-2), and (a-3)). On the other hand, the area under curve of clique size distributions in these latter plots corresponds to the number of cliques returned by each algorithm: the plots confirm that ECC-ms is the more effective, since the y-axis is in log scale and ECC-ms seems to achieve much less small cliques with respect to the other methods.

Looking at the plots in the second column of Figure 10.3, the situation appears more complicated: however it can be seen that ECC-Ms (blue line) and ECC-rc (violet line) correspond to the curve more likely to be close to the y-axis, meaning that the maximum and average covering index induced by these distribution is more likely to be smaller. The same can be said looking at the third column of Figure 10.3, where the edge covering distribution is shown: in this case, the higher improvement permits to appreciate better the lower maximum and average covering index of the edges in the solutions of ECC-Ms and ECC-rc.

In the rest of this section, we give more details on the experimental study of these measures for distributions (a)–(c).

10.6.2 Analyzing clique size distribution

Number of Cliques In the great majority of cases we found that ECC-rs and ECC-ms are the most effective. On the average, they provide about 9\% less cliques than K-ALG. In general, the latter finds more cliques than all ECC-xy in more than 60\% of the cases. Moreover, many edges of real-world networks do not belong to any triangle and hence they must be covered by trivial cliques (i.e. of size 2). The ratio between the number of trivial cliques and the number of edges seems to be positively correlated with the inverse of the average degree in the graph, i.e. \( n/m \): the correlation is quite high indeed, 0.7. We have thus considered the difference between the number of cliques found by each algorithm and the number of trivial cliques. Also in this case, the best methods to minimize the number of cliques are ECC-rs and ECC-ms and they find 12\% less cliques than K-ALG.

Maximum Clique Size dimension In all the experiments the most effective algorithm in finding a (local) maximum size clique are ECC-Mp and ECC-Up. In particular they are the most effective, respectively, for the 95\% and 90\% of the cases. On the average both of them find a clique that is 6\% larger than that of K-ALG. In all the cases, K-ALG never finds a (local) maximum clique larger than the one found by the two algorithms above. When considering all ECC-xy, the algorithm finding the smallest among all is K-ALG for 60\% of the cases.

Average clique size In all the cases ECC-mp and ECC-Mp achieve the maximum average size among all the algorithms. In the 85\% of the cases, the average size of the cliques in the solution provided by K-ALG is smaller than the average size of the cliques found by ECC-mp or ECC-Mp. The average clique sizes returned by ECC-mp and ECC-Mp are, respectively, 27\% and 26\% larger than the one by K-ALG.
CV of clique size distribution  We observe that the CV ranges between 0.013 (ecc-rs) and 0.016 (ecc-Mp). The CV of K-ALG is 0.015. This means that the overall variability of the distributions is quite similar when comparing the algorithms by averaging each of them on all the graphs.

10.6.3 Vertex covering index

Mean vertex covering  In order to avoid redundancy it could be preferable to minimize this measure, that is minimizing the space occupied by the solution. Evaluating $\sum_{u \in V} c(u)$ corresponds to evaluate the sum of the sizes of the cliques in terms of vertices, i.e. $\sum_{i=1}^{k} |C_i|$, so that the average vertex covering index is $\frac{\sum_{i=1}^{k} |C_i|}{n}$. The method ecc-rc achieves more often (in the 45% of the cases) a solution whose mean is less than all the others. On the average this value is 11% smaller than that of K-ALG. In the 22% of the cases the minimum mean is achieved by ecc-mc. On the average its value is 6% smaller than that of K-ALG. Note that the latter never finds the minimum mean among all the algorithms.

Max vertex covering  The smallest maximum vertex covering index is achieved by ecc-Ms in almost 62% of the networks, never by K-ALG. On the average, the former achieves a max vertex covering index that is 37% smaller than the one provided by the latter.

CV  The coefficient of variation ranges in between 0.026 (ecc-rc) and 0.035 (ecc-Ud). An higher variability is achieved by ecc-xr and ecc-xd, while a lower variability is achieved for the variants of ecc-xc.

10.6.4 Edge covering index

Mean edge covering  The algorithms ecc-rc and ecc-mc achieve the lowest mean edge covering for the majority of the networks: they are the best algorithms, respectively, in the 38% and in the 36% of the cases. In all the remaining instances, K-ALG is never the most effective. On the average the mean edge covering index of the solution returned by ecc-rc and ecc-mc is, respectively, 23% and 13% smaller than that by K-ALG. Note that the average edge covering index is equal to $\sum_{i=1}^{k} |C_i| \cdot \frac{|C_i - 1|}{2m}$, where $C_1, C_2, \ldots, C_k$ are the cliques in ecc.

Max edge covering  For half of the networks, the algorithm achieving the minimum among the maximum edge covering indexes is ecc-Ms. Other effective algorithms for are ecc-rc and ecc-mc (respectively, in the 18% and 12% of the cases). Also for this measure, K-ALG is never the most effective. On the average the maximum edge covering indexing of a solution found by ecc-Ms is 87% smaller than that by K-ALG.

CV  The coefficient of variation of the edge covering distributions for all the algorithms range between 0.021 (ecc-Ms) and 0.035 (ecc-Mc, ecc-Uc, and Kellerman algorithm).

10.6.5 Time performance and scalability

In this section we consider the performance of ecc-rc, and compare it to the state of the art g-ALG.

As shown in Section 10.4, ecc-rc uses $O(m + n)$ space and takes $O(m \Delta)$ time, which is, in the worst case, comparable to the $O(m \cdot n)$ time complexity of g-ALG [GGHN06].
In practice, we show that ECC-rc runs in time linear in \( m \), and outperforms g-ALG especially on large graphs, as shown with a dataset of 160 real-world graphs taken from LASAGNE LoAoGN.

Figure 10.4 reports the time used by ECC-rc to compute clique covers as a function of the number of edges in the graph. In other words, for each graph in our dataset, we have run ECC-rc placing a red cross in position \((x, y)\) whether the graph has \( x \) edges and ECC-rc spent \( y \) milliseconds to finish the computation. Looking at the plot, the red crosses seem to be disposed over a line which is parallel to the line \( y = x \) suggesting a linear running time of our algorithm. This hypothesis is experimentally confirmed by the results of a linear regression over the original data (not in log scale): the time \( y \) can be related to the number of edges \( x \) by using \( y = a \cdot x + b \), where \( a = 0.041 \) and \( b = -71005.486 \); these estimates have respectively \( p\)-value \( 7.563 \cdot 10^{-136} \) and \( 0.0174 \) (the correlation is 98.961%). Due to the statistical significance of our tests, we argue that ECC-rc is linear when dealing with large real-world networks.

This feature clearly emerges with the largest networks we considered (see Table 10.4): for the largest one, a snapshot of the web, our algorithm terminates after 32733 seconds, which is quite fast considering the size of the network, which is processed on a single core.

<table>
<thead>
<tr>
<th>Category</th>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>col.</td>
<td>imdb</td>
<td>913201</td>
<td>37588613</td>
<td>202065</td>
</tr>
<tr>
<td>web</td>
<td>enwiki</td>
<td>13834640</td>
<td>42336692</td>
<td>526448</td>
</tr>
<tr>
<td>social</td>
<td>LiveJournal</td>
<td>4847571</td>
<td>42851366</td>
<td>225213</td>
</tr>
<tr>
<td>P2P</td>
<td>p2p</td>
<td>5792297</td>
<td>142038401</td>
<td>1845340</td>
</tr>
<tr>
<td>web</td>
<td>web</td>
<td>39454463</td>
<td>783027125</td>
<td>32733749</td>
</tr>
</tbody>
</table>

Table 10.4: Sample of the largest networks we considered with the running time of ECC-rc.

The average time used by ECC-rc to obtain the covering is 62% smaller than the one used by g-ALG. This means that our algorithm employs on average almost one third of the time employed by g-ALG to complete the task (the standard error for the time values by ECC-rc is 52% smaller with respect to the one by g-ALG). Fig. 10.5 clearly shows that in the case of larger graphs, the situation is much worse for g-ALG: for each graph in our dataset we draw a red cross which is the ratio between the time needed by g-ALG and the one needed by ECC-rc, as a function of the number of the edges. Indeed the plot shows that very often the performance of g-ALG is several orders of magnitude worse than that by ECC-rc. The variability seems to dramatically affect the performance of g-ALG as the number of edges increases. Furthermore, we remark that the comparison does not take into account the (large) networks that only ECC-rc was able to process.
10.7 Conclusions

In this chapter we have considered new and simple algorithms to deal with ECC. Inspired by the *clique cover graph*, we have introduced several measures to assess the quality of a solution rather just the classical one, i.e. minimizing the number of cliques. Our algorithms are part of a simple framework and scale well to deal with large real-world networks. Moreover, we showed that our algorithms improve the state of the art according to all the several measures we considered based on the clique cover graph, experimenting our 20 variants of the framework on a data set of 44 networks. Our algorithms improve in practice the quality of the solution, also according to the traditional measures.
Figure 10.3: Distributions (a)–(c) of the best performing algorithms (mentioned in Table 10.3) together with $k$-alg for a sample of three graphs, namely, as-itdk0304_rlinks (autonomous system network), cit-HepPh (citation network), and email-Enron (communication network). The axes are in logarithmic scale, except for the x-axis of (a-1), (a-2), and (a-3). The plots can be zoomed in the electronic version of this chapter.
Figure 10.5: Ratio between the time used by g-ALG and ECC-rc for each graph, as a function of the number of the edges (y-axis is in log-scale).
<table>
<thead>
<tr>
<th>Category</th>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>Trivial Cliques</th>
</tr>
</thead>
<tbody>
<tr>
<td>autonomous</td>
<td>itdk0304_rlinks_undirected</td>
<td>192244</td>
<td>609066</td>
<td>248035</td>
</tr>
<tr>
<td>biological</td>
<td>Drosophila_melanogaster</td>
<td>10625</td>
<td>40781</td>
<td>33300</td>
</tr>
<tr>
<td>biological</td>
<td>Homo</td>
<td>1027</td>
<td>1166</td>
<td>980</td>
</tr>
<tr>
<td>biological</td>
<td>hprd_pp</td>
<td>9465</td>
<td>37039</td>
<td>17657</td>
</tr>
<tr>
<td>biological</td>
<td>ppi_gcc</td>
<td>37333</td>
<td>135618</td>
<td>71734</td>
</tr>
<tr>
<td>citation</td>
<td>cit-HepPh</td>
<td>34546</td>
<td>420876</td>
<td>24986</td>
</tr>
<tr>
<td>citation</td>
<td>cit-HepTh</td>
<td>27770</td>
<td>352284</td>
<td>15467</td>
</tr>
<tr>
<td>citation</td>
<td>citeseer</td>
<td>259217</td>
<td>532040</td>
<td>343466</td>
</tr>
<tr>
<td>citation</td>
<td>hep-th-citations_MAX</td>
<td>27400</td>
<td>352021</td>
<td>15289</td>
</tr>
<tr>
<td>collaboration</td>
<td>Newman-Cond_mat_95-99</td>
<td>22015</td>
<td>58578</td>
<td>58350</td>
</tr>
<tr>
<td>collaboration</td>
<td>advogato</td>
<td>7418</td>
<td>42892</td>
<td>5100</td>
</tr>
<tr>
<td>collaboration</td>
<td>ca-AstroPh</td>
<td>18771</td>
<td>198050</td>
<td>2236</td>
</tr>
<tr>
<td>collaboration</td>
<td>ca-CondMat</td>
<td>23133</td>
<td>93439</td>
<td>3888</td>
</tr>
<tr>
<td>collaboration</td>
<td>ca-GrQc</td>
<td>5241</td>
<td>14484</td>
<td>1606</td>
</tr>
<tr>
<td>collaboration</td>
<td>ca-HepPh</td>
<td>12006</td>
<td>118489</td>
<td>2588</td>
</tr>
<tr>
<td>collaboration</td>
<td>ca-HepTh</td>
<td>9875</td>
<td>25973</td>
<td>3558</td>
</tr>
<tr>
<td>communication</td>
<td>email-Enron</td>
<td>36691</td>
<td>183830</td>
<td>14069</td>
</tr>
<tr>
<td>communication</td>
<td>email-EnAll</td>
<td>265214</td>
<td>364480</td>
<td>253357</td>
</tr>
<tr>
<td>word-adj</td>
<td>darwinBookInter_st</td>
<td>7381</td>
<td>44207</td>
<td>4714</td>
</tr>
<tr>
<td>word-adj</td>
<td>frenchBookInter_st</td>
<td>8325</td>
<td>23841</td>
<td>9683</td>
</tr>
<tr>
<td>word-adj</td>
<td>japaneseBookInter_st</td>
<td>2704</td>
<td>7998</td>
<td>2916</td>
</tr>
<tr>
<td>word-adj</td>
<td>spanishBookInter_st</td>
<td>11586</td>
<td>43065</td>
<td>8177</td>
</tr>
<tr>
<td>p2p</td>
<td>p2p-Gnutella31</td>
<td>62586</td>
<td>147891</td>
<td>142503</td>
</tr>
<tr>
<td>social</td>
<td>soc-Slashdot0811</td>
<td>77360</td>
<td>469140</td>
<td>231064</td>
</tr>
<tr>
<td>social</td>
<td>soc-Slashdot0902</td>
<td>82168</td>
<td>504230</td>
<td>247909</td>
</tr>
<tr>
<td>social</td>
<td>trust</td>
<td>49288</td>
<td>381036</td>
<td>64870</td>
</tr>
<tr>
<td>social</td>
<td>wiki-Vote</td>
<td>7115</td>
<td>100761</td>
<td>8655</td>
</tr>
<tr>
<td>synthetic</td>
<td>Gnp_1e4</td>
<td>10000</td>
<td>59849</td>
<td>59061</td>
</tr>
<tr>
<td>synthetic</td>
<td>Gnp_2e3</td>
<td>2000</td>
<td>8994</td>
<td>8698</td>
</tr>
<tr>
<td>synthetic</td>
<td>Gnp_5e3</td>
<td>5000</td>
<td>24809</td>
<td>24307</td>
</tr>
<tr>
<td>synthetic</td>
<td>forest1e4</td>
<td>10000</td>
<td>49354</td>
<td>2698</td>
</tr>
<tr>
<td>synthetic</td>
<td>forest1e4_2</td>
<td>10000</td>
<td>153925</td>
<td>2967</td>
</tr>
<tr>
<td>synthetic</td>
<td>forest5e4</td>
<td>5000</td>
<td>243441</td>
<td>13740</td>
</tr>
<tr>
<td>synthetic</td>
<td>kron14</td>
<td>8156</td>
<td>24482</td>
<td>21016</td>
</tr>
<tr>
<td>synthetic</td>
<td>kron16</td>
<td>30429</td>
<td>65494</td>
<td>63790</td>
</tr>
<tr>
<td>synthetic</td>
<td>ud_1e3</td>
<td>1000</td>
<td>16727</td>
<td>9</td>
</tr>
<tr>
<td>synthetic</td>
<td>ud_1e4</td>
<td>10000</td>
<td>313726</td>
<td>16</td>
</tr>
<tr>
<td>synthetic</td>
<td>ud_2e3</td>
<td>1999</td>
<td>35697</td>
<td>30</td>
</tr>
<tr>
<td>synthetic</td>
<td>ud_5e3</td>
<td>4998</td>
<td>97027</td>
<td>41</td>
</tr>
<tr>
<td>synthetic</td>
<td>us_1e3</td>
<td>1000</td>
<td>14334</td>
<td>0</td>
</tr>
<tr>
<td>synthetic</td>
<td>us_1e4</td>
<td>10000</td>
<td>242533</td>
<td>0</td>
</tr>
<tr>
<td>synthetic</td>
<td>us_2e3</td>
<td>2000</td>
<td>37928</td>
<td>0</td>
</tr>
<tr>
<td>synthetic</td>
<td>us_5e3</td>
<td>5000</td>
<td>135833</td>
<td>0</td>
</tr>
<tr>
<td>web</td>
<td>GoogleNw</td>
<td>15763</td>
<td>148585</td>
<td>6756</td>
</tr>
</tbody>
</table>

Table 10.5: A summary of our dataset
In this chapter we consider the problem of Frequent Itemset Mining. The problem has a strong link to graph algorithms as frequent itemsets can be modeled as bipartite cliques in a bipartite graph.

Given a set of items, and a set of transactions, or baskets, which correspond to sets (or multisets) of items, the problem asks which sets of items appear frequently in the same basket.

A fundamental part of the problem is choosing an appropriate threshold for deciding how many times must an itemset occur to be considered frequent.

When dealing with real-world data, issues may arise from the presence of a power-law behavior. Indeed, some items may occur more frequently than others by orders of magnitude. Using a single threshold to process the whole dataset may not always be a suitable choice. Indeed, a high threshold may ignore the less frequently occurring items, while a low one may report as frequent itemsets large amounts of non-significant item sets made of frequently occurring items.

We propose a heuristic approach which decomposes of the problem into smaller instances, aiming at processing different parts of the input data using different granularity. The approach assumes a power-law structure in the input data, and we show through experimental evaluation how it is able to find significant itemsets involving all items rather than just a certain type.

11.1 Introduction

Frequent Itemset Mining (FIM) is a fundamental data mining technique, originally conceived for the task of market analysis, but nowadays applied to many different problems related to data mining and information retrieval [Bor12]. The aim of FIM is finding sets of items that are frequently chosen together. The instance of the problem is usually represented as a collection of baskets, which are sets or multisets of items. The classical setting that gives this naming is that of observing customer behavior in stores, as the knowledge that product X is frequently bought by people that buy product Y may enable sellers to make intelligent marketing strategy to promote one product or the other.

Given an itemset (set of items) $A$, the set of baskets that contain all items in $A$ is called the support of $A$, hereafter denoted as $\text{support}(A)$.

The principle that drives FIM is to find itemset with large support, which correspond to groups of items frequently chosen together.

$A$ is a frequent itemset for a given threshold $\tau$ if $|\text{support}(A)| \geq \tau$. An itemset is called closed if no item can be added to it without decreasing the size of its support, and maximal if no item
can be added to it without decreasing its support below \( \tau \) (thus a maximal itemset is also closed).

Since, by definition, any non-closed itemset is contained in some closed one with the same support, state of the art approaches usually focus on closed frequent itemsets rather than all frequent itemsets to greatly cut the result size without losing significant information.

We finally remark this basic yet useful property of itemsets:

**Remark 11.1.** Given an item set \( A \) and a nonempty subset \( B \subseteq A, \) \(|\text{support}(B)| \geq |\text{support}(A)|\).

In other words, the property of being a frequent itemset is hereditary.

**State of the art.**

Many algorithms for frequent itemset mining have been proposed so far. Among the best known ones we have the *a priori* method [AMS+96], which essentially consist in exploring the search space by recursively adding all possible items to an itemset while the support is still satisfactory. Due perhaps to its simplicity, the *a priori* method can achieve good performance even on large datasets. Another well known approach which shows great practical performance is LCM [UKA04]. Many more approaches exist (see [Bor12] for a survey); for the scope of this chapter, however, we are particularly interested in LCM. This interest does not arise just from the performance[1], but also from the fact that the algorithm models FIM as a graph enumeration problem. As explained in Section 11.2, closed frequent itemsets correspond to maximal bipartite cliques on a suitably defined graph.

It should also be noted that not just the support, but a whole plethora of different metrics have been proposed for the problem: some approaches consider *high utility* itemsets [LLC05], which essentially correspond to a weighted version of the problem where to each item is associated a profit; others consider sets of items such that the baskets containing them respect some notion of similarity [SB11], and several more evaluation metrics have been proposed in [TKS02].

### 11.2 FIM as a problem on bipartite graphs

We represent the data as a simple bipartite graph \( G(V_B, V_I, E) \) which we call the *dataset*. Vertices in \( V_B \) represent baskets, vertices in \( V_I \) represent items, and an edge \( \{b, i\} \in E \) means that the basket \( b \) contains at least one occurrence of the item \( i \). We call \( \Delta_B \) and \( \Delta_I \) respectively the maximum degree of a vertex in \( V_B \) and \( V_I \). For the rest, the terminology is consistent with Section 2.1.

Note that any itemset \( A \subset V_I \) with nonempty support induces a bipartite clique (*biclique*) in \( G \), that is \( (B, A) \) where \( B = \text{support}(A) \). This follows from the way the support is defined.

**Lemma 11.1.** Closed itemsets in \( G \) are equivalent to maximal bicliques in the dataset \( G_I \) of \( G \).

*Proof.* Given the biclique \( (B, A) \) with \( B = \text{support}(A) \), we have only two ways to violate the maximality of \( (B, A) \) and make the lemma false. (1) There is a vertex \( b \in V_B - B \) such that \( (B \cup \{b\}, A) \) is still a clique. This implies that \( b \) is adjacent to all vertices in \( A \), and hence it is already in \( \text{support}(A) \), which is \( B \); a contradiction. (2) There is a vertex \( a \in V_I \) such that \( (B, A \cup \{a\}) \) is still a clique. If this was possible, the itemset \( A \cup \{a\} \) would have the same support as \( A \) while containing \( A \); this implies that \( A \) is not closed, a contradiction.

Observe that maximal itemsets are also closed, thus by Lemma 11.1 they too are maximal bicliques in the dataset.

---

[1] LCM (ver. 2) was awarded the FIMI’04 “diapers and beer” best implementation award [02].

194
11.3 Our goal

State of the art approaches exhibit an important limitation on large, real-world datasets: they are efficient at extracting itemsets with high support \cite{Bor12}, or high-profit itemsets \cite{LLC05}, which usually include the most popular items, i.e., those most frequently bought, but they use a fixed-threshold paradigm which is unable to provide a meaningful output for items that are less popular (niche items): indeed, the more frequently two items are bought, the higher the chance that they will appear in the same basket. This means that a group of popular items which are bought together by coincidence, may be higher support than a group of niche items which are strongly correlated, simply because they appear in a much smaller number of baskets. While this behaviour is unlikely if the items have all comparable popularity, we can expect it to occur when data exhibits a power-law structure, with the popular items occurring more frequently than others by orders of magnitude.

For analyzing niche items we need to use a low threshold, but this would imply a huge amount of non-significant results given by many combinations of popular items.

We aim at providing an adaptive filter which allows to efficiently separate groups of items which are correlated and at the same time in the same importance bracket, and allows us to analyze the data with different levels of granularity.

To do so, we take advantage of the very property that makes state of the art approaches unsuitable for fine-grained analysis of real-world data, that is their scale-free nature: given a threshold \(\tau\), our approach will identify which items can be properly analyzed, and which items possess a large number of non-significant association, i.e., correspond to hubs, and should be analyzed with a higher threshold.

In particular, we will produce an approach that is efficient and will give significant output under the following assumptions:

**Definition 11.1** (input structure assumptions). 1. The degree distribution of vertices in \(V_I\) follows a power-law.

2. \(|V_I|\) is relatively small.

3. Even though \(|V_B|\) is large, vertices in \(V_B\) have small degree.

Which we will motivate by analysis of dataset from \cite{Rep}.

11.4 I-GRAPH

In this section we will define the I-GRAPH, and give a fast technique to build it on real-world datasets.

**Definition 11.2** (I-GRAPH). Given the input data, i.e., the bipartite graph \(G(V_B, V_I, E)\) and a threshold \(\tau\), we define the I-GRAPH as \(G_I(V_I, E_I)\): the vertices of \(G_I\) correspond to the items of \(G\), and for each distinct pair of items \(x, y \in V_I\) there is an edge \(\{x, y\} \in E_I\) iff \(|N(x) \cap N(y)|\) (in \(G\)) is greater than or equal to \(\tau\).

The I-GRAPH possesses useful properties that will allow us to decompose the problem, such as the one shown in the following lemma:

**Lemma 11.2.** Given a bipartite graph \(G(V_B, V_I, E)\) and a threshold \(\tau\), if \(A \subseteq V_I\) is a frequent itemset in \(G\) then \(A\) is a clique in the I-GRAPH \(G_I(V_I, E_I)\).

Proof. If \(A\) is a frequent itemset, we have \(|\text{support}(A)| \geq \tau\). For any pair \(x, y \in A\) we have \(|N(x) \cap N(y)| = |\text{support}(x, y)| \geq \tau\) by Remark 11.1 and thus \(x, y \in E_I\).
It should be noted that the implication is not true in the other direction: while all frequent itemsets are cliques in $G_I$, a clique in $G_I$ might not correspond to a frequent itemset, thus the problem of finding frequent itemsets in $G$ is not equivalent to that of finding cliques in $G_I$.

While the $i$-graph $G_I$ does not contain all the information in $G$, it helps us to “separate the wheat from the chaff”. In other words, using $G_I$ we can avoid trying to put together some items that will surely not lead to fruitful solutions, thus reducing the search space. Observe that the cuts in the search space correspond to vertices that are not neighbors in the $i$-graph, so the effectiveness of the approach will depend on the sparsity of the $i$-graph.

### 11.4.1 Building the $i$-graph

We present a way to efficiently build the $i$-graph, which takes advantage of the structure of the input described above.

The approach, shown in Algorithm 31, aims at building a weighted $i$-graph, where each edge $\{x,y\}$ has a weight corresponding to the support of the itemset containing just $x$ and $y$. Once this is computed, the edges with weight smaller than $\tau$ are removed.

**Algorithm 31: Building $i$-graph**

| Input: | The dataset $G(V_B, V_I, E)$, a threshold $\tau$ |
| Output: | The $i$-graph $G_I(V_I, E_I)$ |
| 1 | $M \leftarrow$ a map from pairs $= \langle V_I, V_I \rangle$ to an integer |
| 2 for $b$ in $B_I$ do |
| 3 | for all distinct $i, j$ in $N(b)$ do |
| 4 | if $M[i, j]$ is not initialized then $M[i, j] \leftarrow 0$ |
| 5 | $M[i, j] \leftarrow M[i, j] + 1$ |
| 6 | for all entries $[i, j] \in M$ do |
| 7 | if $M[i, j] \geq \tau$ then |
| 8 | $E_I \leftarrow E_I \cup \{i, j\}$ |
| 9 | return $G_I(V_I, E_I)$ |

### 11.4.2 Complexity

Recalling that $m$ is the number of edges in $G$, note that $m \leq |V_B| \cdot |V_I|$ and $\Delta_B \leq |V_I|$. We only need to keep the map $M$ and the $i$-graph in memory, while we can read the baskets in a streaming fashion. As the size of one basket is $O(\Delta_B)$, and the size of both $M$ and the $i$-graph is at worst $O(|V_I|^2)$, the space cost is $O(|V_I|^2)$. The time complexity is dominated by the operations on the neighborhood of $b$. As there are $O(|N(b)|^2)$ pairs to analyze, the time complexity is $O\left(\sum_{b \in V_B} |N(b)|^2\right)$; since $|N(b)| \leq \Delta_B$ and $O\left(\sum_{b \in V_B} |N(b)|\right) = m$, this is bounded by to $O(m \cdot \Delta_B)$. However, we remark again that the average degree in $V_B$, and so the cost, will be small in practice.

As some edges are created and then removed from the $i$-graph as their threshold is too small, the space cost is bounded by $O(|V_I|^2)$ rather than by the number of edges in the $i$-graph.
11.5 Problem decomposition

In this section we describe our decomposition approach, which aims at partitioning the graph into blocks which contain significant itemsets. The strategy exploits the i-graph to identify groups of items that strongly interact with each other, among which significant itemsets may be found.

In order to find these groups of items we adapt some useful techniques from sparse matrix compression [KF11] and clique enumeration [CVM+16b].

**Definition 11.3** (item cluster). Given a dataset \( G(V_B, V_I, E) \) and a threshold \( \tau \), an item cluster \( C = \langle K, B \rangle \) consists in two disjoint sets \( K, B \subseteq V_I \), called kernel and boundary respectively, where \( B = \{ x : \text{sup}(\{x,y\}) \geq \tau \text{ for some } y \in K \} \).

**Definition 11.4** (item clustering). Given a dataset \( G(V_B, V_I, E) \) and a threshold \( \tau \), an item clustering of \( G \) is a set of item clusters \( C_1 = \langle K_1, B_1 \rangle \ldots C_j = \langle K_j, B_j \rangle \) s.t. each item \( x \in V_I \) appears in the kernel of exactly one item cluster.

These definition are related to the one of block in [CVM+16b], and will allow us to decompose the problem in smaller subproblems, without compromising the correctness of the result.

**Lemma 11.3**. Given a dataset \( G(V_B, V_I, E) \), a threshold \( \tau \), a frequent itemset \( A \) (respectively closed, maximal) and an item cluster \( C = \langle K, B \rangle \) s.t. \( A \cap K \neq \emptyset \), then \( A \) is a frequent itemset (respectively closed, maximal) in the bipartite graph induced by \( C \) and the baskets connected to it.

**Proof.** Consider the graph \( G'(V_B', V_I', E') \) induced by \( C = \langle K, B \rangle \) and the baskets connected to it, so \( V_I' = K \cup B \) and \( V_B' \) consists in all the baskets that contain an item in \( V_I' \). \( A \subseteq V_B' = K \cup B : A \cap K \neq \emptyset \), every other element of \( A \) is in \( B \) by definition of item cluster and Remark [11.1]. Thus \( A \) is a frequent itemset in \( G' \), as \( \text{sup}(A) \) in \( G' \) is equal to \( \text{sup}(A) \) in \( G \). If \( A \) is closed (respectively maximal) in \( G \), then \( \exists x \in V_I \) s.t. \( \text{sup}(A \cup \{x\}) = \text{sup}(A) \) (respectively \( \text{sup}(A \cup \{x\}) \geq \tau \)). As the neighborhood of vertices in \( V_I' \) in \( G' \) is equal to the one in \( G \) by construction of \( G' \), the same holds for \( G' \), thus \( A \) is closed (respectively maximal) in \( G' \).

We call **processing** a cluster the operation of retrieving all such itemsets form a given cluster:

**Definition 11.5** (Processing a cluster). Given an item cluster \( C = \langle K, B \rangle \) of a graph \( G \) for a threshold \( \tau \), processing \( C \) corresponds to listing all frequent itemset \( A \) (respectively closed, maximal) in \( G \) such that \( A \cap K \neq \emptyset \).

The following Lemma follows directly from Lemma [11.3] and will guarantee the completeness and correctness of our decomposition approach.

**Lemma 11.4**. Given a dataset \( G(V_B, V_I, E) \), a threshold \( \tau \), and an item clustering \( C_1 \ldots C_j \) of \( G \), processing all clusters will yield all frequent itemsets (resp. closed, maximal) of \( G \).

**Proof.** Follows from Lemma [11.3], as each item belongs to the kernel of an item cluster, thus each itemset \( A \) will necessarily have nonempty intersection with the kernel of at least one item cluster.

We thus know that, processing each item cluster of a clustering with a state of the art FIM algorithm will yield all and only the frequent itemsets of the original dataset.

Observe, that a given itemset may have nonempty intersection with the kernel of multiple item clusters, and thus said itemset may be found more than once. We will show however that the item clustering built by our approach avoids generating duplicates.

It is straightforward to see that the i-graph facilitates the computation of item clusters, as given any set of items \( K \), i-graph can be used to compute \( B \) as

\[
N^u(K) = \bigcup_{x \in K} N(x) \setminus K.
\]
11.5.1 Finding an item clustering

In this section we describe FIND-I-CLUSTERING(), the basic operation at the core of our approach which will find a suitable item clustering from a given I-GRAPH.

FIND-I-CLUSTERING() takes inspiration from the SlashBurn algorithm [KF11], meant for sparse matrix compression, and adapts it to the task of finding an item clustering, while at the same time identifying the hubs of the I-GRAPH, i.e., the vertices with a large number of associations which should be analyzed with a higher threshold.

The SlashBurn algorithm has a simple structure: let $H$ (for hubs) and $S$ (for spokes) be two empty lists. Given a graph $G(V,E)$, the algorithm consists in iteratively removing the highest vertex degree from $G$, and appending it to the tail of $H$. Whenever the graph becomes disconnected, the algorithm restricts the graph to the largest connected component, and appends vertices of the other connected components to the head of $S$. When all vertices have been removed from the graph, the permutation of the vertices obtained by concatenating $H$ and $S$ yields an adjacency matrix with large blank areas, which is suitable for compression.

In our approach we process the I-GRAPH similarly, and generate a cluster for each connected component which is added to $S$, plus a final one for the vertices in $H$. The process is detailed in Algorithm 32, where max-degree($G_I$) corresponds to the vertex of highest degree in $G$. As the algorithm performs at most $n = |V_I|$ removals (Line 7), and checking the connectivity of $G_I$ takes $O(|E_I|)$, the complexity of Algorithm 32 is bounded by $O(|V_I| \cdot |E_I|)$.

Algorithm 32: FIND-I-CLUSTERING: generating an item clustering

| Input: | The I-GRAPH $G_I(V_I,E_I)$ |
| Output: | An item clustering $C = \langle K_1,B_1 \rangle \ldots \langle K_j,B_j \rangle$ |

1. $H \leftarrow \emptyset$
2. $C \leftarrow \emptyset$
3. while $V_I \neq \emptyset$ do
   4. while $G_I$ is connected do
      5. $h \leftarrow $ max-degree($G_I$)
      6. $H \leftarrow H \cup \{h\}$
      7. $G_I \leftarrow G_I \setminus \{h\}$ // remove $h$ and all edges incident to it
      8. $M \leftarrow$ largest connected component of $G_I$
   9. foreach Connected component $K_i$ of $G_I$ except $M$ do
      10. $C \leftarrow C \cup \langle K_i,N^u(K_i) \rangle$ // create a new item cluster
   11. $G_I \leftarrow M$
12. $C \leftarrow C \cup \langle H,N^u(H) \rangle$ // hubs item cluster
13. return $C$

An interesting property of the algorithm is that the number of vertices in $H$, as well as the size of the single components which get added to $S$, tend to be small when the graph exhibits scale-free properties (and, in particular, when the graph has small wing-width) [KF11]. Furthermore, the key point which will be beneficial to our approach, is that whenever a component $S_i$ gets disconnected during the procedure and added to $S$, the vertices in that component may be only adjacent to each other, or to some vertex in $H$. This implies that every cluster has a small kernel, and a boundary which will also be small as it is a subset of $H$. The cluster whose kernel is $H$ itself makes an exception, as its boundary can be large, however but this is not relevant for the final algorithm, as $H$ correspond to the hub vertices, which have a large number of non-significant connections for the current threshold and should be analyzed with a higher one.

198
Finally, we should note that two different clusters which are not $H$ can not lead to the discovery of the same frequent itemset: indeed, as the boundary of each cluster is completely contained in $H$, a cluster may not have in its boundary a vertex belonging to the kernel of another. This, combined with Lemma 11.3, gives us the following result:

**Lemma 11.5.** Given a dataset $G = (V_B, V_I, E)$, its i-graph $G_I$ and the item clustering $\mathcal{C}$ output of FIND-I-CLUSTERING($G_I$), processing all item clusters except that induced by $H$, will yield all and only the frequent itemset (resp. closed, maximal) with at least a vertex in $S = V \setminus H$ without duplication. □

While the decomposition approach in [CVM+16b] requires a careful analysis and removal of duplicate elements in each block, the absence of duplicates is a strong point of this decomposition. For the sake of completeness, it should be noted that the remaining itemsets (those completely contained in $H$) may be found by processing the cluster $\langle H, N(H) \rangle$, and discarding all itemsets which contain a vertex of $N(H)$.

### 11.6 Dynamic threshold decomposition

Here we present our final approach, which uses the building steps described above in order to filter the data and efficiently retrieve a set of significant itemsets which cover both popular and niche items. The approach takes as input the dataset $G(V_I, V_B, E)$, and a set of thresholds $\tau_1, \ldots, \tau_k$, which correspond to the levels of granularity with which the data should be analyzed. For each $\tau_i$, the approach will select which items should be analyzed with respect to it, and which should be analyzed with a different threshold.

Consider the possible types of frequent itemsets found by processing the result of FIND-I-CLUSTERING for some threshold $\tau$:

**Itemsets completely contained in $S = V \setminus H$.** We consider these itemsets the most significant. These itemsets corresponds to groups of items in the same “bracket” of importance which frequently occur together, which are those that our approach aims to find.

**Itemsets with at least a vertex in $S$ and one in $H$.** As we discussed above, we consider the vertices in $H$, the hubs, those whose associations (edges in the i-graph) are too many to be significant. In other words, we consider non significant the correlation between vertices in $S$ and those in $H$. On the other hand, the subset of the itemset made by only the vertices of $S$ is indeed still a frequent itemset (by Remark 11.1) of the first type described above, and should thus be considered if it has not been already found independently.

**Itemsets completely contained in $H$.** These itemsets may or may not be significant, however the current threshold $\tau$ is too low to discriminate which itemsets in this class correspond to a real correlation and which to just noise, thus they should be analyzed with a higher $\tau$.

For a given $\tau_i$, we thus retrieve itemsets of the first and second type (removing the hubs from the latter), and proceeds to process the vertices in $H$ with a higher threshold, performing a new decomposition of the i-graph induced by them.

### 11.7 Experimental results

**Experimental datasets.** We considered three real-world datasets taken from the FIMI 2004 [04] repository. Table 11.1 shows the properties of the i-graph build from them. The properties of
Algorithm 33: Listing significant frequent itemsets

Input: The dataset $G(V_B, V_I, E)$, a set of thresholds $\tau_1 \ldots \tau_k$

Output: A set of significant frequent itemsets

1 foreach $\tau_i$ in increasing order do
2 $G_I \leftarrow$ the i-graph of $G$ for $\tau_i$
3 $\mathcal{C} \leftarrow$ FIND-I-CLUSTERING($G_I$)
4 foreach $C_i \in \mathcal{C}$ except $H$ do
5 List frequent itemsets in $C_i$ w.r.t. threshold $\tau_i$
6 $G \leftarrow G[H]$

the datasets seem to support our assumptions: there are much fewer items than baskets, and the average and maximum basket size are relatively small.

<table>
<thead>
<tr>
<th>Name</th>
<th>Baskets</th>
<th>Items</th>
<th>$\Delta_B$</th>
<th>Edges</th>
<th>Avg Basket Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCIDENTS</td>
<td>340183</td>
<td>468</td>
<td>51</td>
<td>11,500,870</td>
<td>33.8</td>
</tr>
<tr>
<td>KOSARAK</td>
<td>990002</td>
<td>41270</td>
<td>2497</td>
<td>8,018,988</td>
<td>8.1</td>
</tr>
<tr>
<td>RETAIL</td>
<td>88162</td>
<td>16470</td>
<td>76</td>
<td>908,576</td>
<td>10.3</td>
</tr>
</tbody>
</table>

Table 11.1: Properties of the dataset of the considered real-world datasets.

Results. Here we show the result of our algorithm, showing that the decomposition is an effective way of partitioning the graph in small, coherent, components, and that the itemsets found are significant.

As a quality measure, we consider a generalized form of Jaccard index [KMP⁺12], which is, for itemset $A$,

$$\text{QUALITY}(A) = \frac{\text{support}(A)}{|\bigcup_{i \in A} N(i)|}$$

where $N(i)$ is the set of baskets which contain the item $i$. The value of QUALITY is clearly within 0 and 1, and a value closer to 1 means a higher significance, and a value close to 0 symbolizes no deep correlation among the items in the cluster. Other than just the support, this metric takes into account the correlation between items: sets items which are often bought together, but are more frequently bought separately, will achieve a high support but not a high QUALITY.

The FIMI 2004 repository includes experiments from the algorithms that participated in the challenge. State of the art algorithm find all closed frequent itemsets. As our approach aims at finding only a subset of frequent itemsets, made of items in the same category of popularity, it is not clear how to compare them with our algorithm. However, we can select similar thresholds to those considered in the challenge, and show our results.

We selected 4 threshold for each dataset, taken evenly within the range used for each of these dataset in the FIMI 2004 challenge. For each set of thresholds we ran Algorithm 33 on the corresponding graph. Table 11.2 shows the thresholds considered, as well as the average quality...
obtained from the processed datasets. Furthermore, we show in Figure 11.1 the size distribution of the clusters found.

Since each cluster is only connected within itself, and with the single hub cluster, the small size of the clusters obtained is remarkable. The results seem thus to suggest that Algorithm 33 is indeed a suitable way of decomposing the graph in many smaller instances. Table 11.2 further shows that these instances are significant, since the quality of the closed frequent itemsets reported is definitely not trivial: the ACCIDENTS and KOSARAK datasets show an average cluster correlation of respectively 29% and 18%. This value is somewhat smaller for the RETAIL dataset, however we argue that a 7% correlation is still insignificant, as, e.g., the number of baskets that may contain an item is potentially large, and the QUALITY measure would be negatively impacted by such a case.

<table>
<thead>
<tr>
<th>NAME</th>
<th>$\tau_1$</th>
<th>$\tau_2$</th>
<th>$\tau_3$</th>
<th>$\tau_4$</th>
<th>QUALITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCIDENTS</td>
<td>2000</td>
<td>5000</td>
<td>15000</td>
<td>30000</td>
<td>0.2934</td>
</tr>
<tr>
<td>KOSARAK</td>
<td>500</td>
<td>1000</td>
<td>5000</td>
<td>10000</td>
<td>0.1830</td>
</tr>
<tr>
<td>RETAIL</td>
<td>10</td>
<td>20</td>
<td>100</td>
<td>200</td>
<td>0.0704</td>
</tr>
</tbody>
</table>

Table 11.2: Properties of the dataset of the considered real-world datasets.
11.8 Final remarks

In this chapter we considered the problem of Frequent Itemset Mining, modeling it as a graph enumeration problem, and making assumptions on the structure of the graphs generated by real-world datasets.

We considered a new perspective for the problem, that of finding significant itemsets, made of items which have similar popularity. This aims at acting as a noise filter, which allows to focus on itemsets made by non-popular items, without having to deal with the large amount of noise generated by processing popular items with a low threshold.

We proposed a heuristic approach which decomposes of the problem into smaller instances, and tested it on real-world data, showing that it can effectively decompose the graph and find itemsets which are significant and contain items with similar popularity.

Future work is aimed, on one hand, at obtaining theoretical guarantees for the quality of the result, and on the other hand, at finding suitable threshold values directly from analyzing the input.
k-plexes are a formal yet flexible way of defining communities in networks. They generalize the notion of cliques and are more appropriate in most real cases: while a vertex of a clique \( C \) is connected to all other vertices of \( C \), a vertex of a \( k \)-plex may miss up to \( k \) connections. Unfortunately, computing all maximal k-plexes is a gruesome task and state-of-the-art algorithms can only process small-size networks. In this chapter we propose a new approach for enumerating large k-plexes in networks that speeds up the search by several orders of magnitude, leveraging on (i) methods for strongly reducing the search space and (ii) efficient techniques for the computation of maximal cliques. Several experiments show that our strategy is effective and is able to increase the size of the networks for which the computation of large k-plexes is feasible from a few hundred to several hundred thousand vertices.

12.1 Introduction

In the vast majority of networks representing real-world scenarios the distribution of edges is not uniform and it is often possible to clearly distinguish groups of vertices that are highly connected. The automatic detection of these groups, often called communities, helps to discover fundamental properties of large networks in a variety of different domains. For this reason this problem has been largely investigated [For10].

A clique is a set of vertices in a network with all possible edges among them, and is a formal and strict way of defining a community. So strict, in fact, that cliques are generally thought to be too rigid to be used in practice [PYB12]. A more appropriate notion in many practical cases is the \( k \)-plex: a set of vertices such that each of them has edges with all the others, with the possible exception of up to \( k \) missing neighbors (including itself). So, for example, for \( k = 1 \), \( k \)-plexes are cliques, for \( k = 2 \), each vertex may miss one edge, etc. Hence, \( k \)-plexes are a simple and intuitive generalization of cliques.

The problem of finding \( k \)-plexes arises in social network analysis [BBH11], but it has wider applicability in several important areas employing graph-based data mining [PYB12, BCK15, ZHOT16]. Unfortunately, the detection of all maximal \( k \)-plexes in a network is unpractical being hindered by two main problems: (i) maximal \( k \)-plexes are even more numerous than maximal cliques, even if most \( k \)-plexes are small and not significant; (ii) the most efficient algorithms in the literature for computing maximal \( k \)-plexes can only be used on small-size graphs: we show in our experiments that the largest networks we were able to analyze with the algorithm in [BCK15] have a few hundred vertices.

In this chapter we propose a solution to the first issue that is also a solution to the second one. Namely, if we restrict the search to large \( k \)-plexes, which are the most meaningful in practice, we
can devise efficient algorithms to detect them.

Indeed, computing all maximal \( k \)-plexes does not make sense when the purpose is that of detecting communities. In this respect, it is useful to focus on the relationship between \( s \), the size of a \( k \)-plex, and \( k \) itself. Starting from \( k = 1 \), which corresponds to cliques, if we increase the value of \( k \), we obtain progressively sparser communities that are clearly less interesting in practice. In addition, there is a dramatic effect on small \( k \)-plexes: it is trivial that if \( s \leq k \) a \( k \)-plex can be composed of isolated vertices, but it is possible to show that even if \( s < 2k \), the \( k \)-plex can be disconnected (see Section 12.5). Hence, small \( k \)-plexes do not correspond to communities. In particular, in order to avoid finding the degenerate \( k \)-plexes mentioned above, it is natural to impose at least that \( s \geq 2k \).

In this framework, our strategy for finding large \( k \)-plexes relies on two main observations. First, the complexity of the problem can be reduced in the vast majority of cases on the basis of certain properties of large \( k \)-plexes that can be efficiently checked and that allows us to filter out a large portion of the network before starting their search. The second consideration is that, differently to what happens for \( k \)-plexes, the state-of-the-art techniques to compute all maximal cliques are able to scale up to millions of vertices by decomposing the network into small blocks \cite{CZKC12, CVM+16b}. Unfortunately, the decomposition approach cannot be easily adapted to the detection of \( k \)-plexes. However, we demonstrate that we can find all \( k \)-plexes non-smaller than \( m \) by looking in the neighborhood of cliques of a size that depends on \( k \) and \( m \). Hence, it turns out that the knowledge of maximal cliques in a network provides a hint for finding all the significant \( k \)-plexes.

In sum, our contributions are the following.

- We present techniques to efficiently compute all maximum \( k \)-plexes of a network in the hypothesis that they are greater than a fixed threshold, which is adequate to obtain significant results. Our approach is based on the intuition that the efficient computation of all maximal cliques can be exploited to guide the search for \( k \)-plexes towards specific portions of the network, filtering out uninteresting parts.

- We propose an algorithm to detect all maximal \( k \)-plexes whose size is at least a given threshold. The algorithm is based on a decomposition of the network in smaller blocks, which is, to the best of our knowledge, the first decomposition that is proposed for this purpose and, again, is based on the maximal cliques that are detected in the network.

- We illustrate an experimentation showing that these techniques are able to speed up the computation of several orders of magnitude with respect to traditional algorithms, increasing the size of the networks for which computing maximal \( k \)-plexes is a feasible task from a few hundred vertices to several hundred thousand vertices.

The rest of this chapter is organized as follows. Section 12.2 contains an overview of our approach and results. Sections 12.3 and 12.4 describe in detail our approach to find all largest \( k \)-plexes in the network and all the most significant \( k \)-plexes, respectively. Section 12.5 contains the theoretical basis of our algorithms. The efficiency of the algorithms is experimentally measured in Section 12.6. Finally, Sections 12.7 and 12.8 contain related work and our concluding remarks.

### 12.2 Overview

As mentioned in the introduction, our approach is based on two main ideas: (i) before starting the search of \( k \)-plexes, we can filter out a relevant portion of the network in which necessary conditions for the presence of large \( k \)-plexes do not hold, and (ii) in large networks, cliques
can drive the search of $k$-plexes. While the first point provides an effective way to simplify the problem at hand, the second can lead to an efficient strategy for finding $k$-plexes.

Let us elaborate on these ideas starting with the problem of finding all $k$-plexes of maximum size. Assume that we have computed all the maximal cliques of a network and let $\omega$ be the size of the maximum clique. Then, a maximum $k$-plex has size at least $\omega$, since cliques are also $k$-plexes. For example, suppose we are searching for $k$-plexes in the network in Figure 12.1a, which we will use as a running example in this section: we have that $\omega = 5$, since the maximum clique (the blue subgraph on the left hand side) involves five vertices.

At this point, it turns out that two filtering criteria can be applied.

1. **Coreness** Our first intuition follows from the very definition of $k$-plex: all the vertices of a $k$-plex of size $m$ must have degree non-smaller than $m - k$. If we know that the size of a maximum $k$-plex is at least $\omega$, this means that we can iteratively filter out any vertex that has degree lower than $\omega - k$. This corresponds to computing the coreness of all the vertices of the network, that is, the largest integer $h$ for which the vertices belongs to an $h$-core\footnote{An $h$-core is a connected subgraph of minimum degree at least $h$.}. This is formally proven in Lemma 12.2 and can be executed in linear time \cite{BZ03}. For example, suppose we are searching for 2-plexes in the running example of Figure 12.1a for which $\omega = 5$: we can filter out the three black vertices on the top of the picture since they have coreness 2, which is less than $\omega - k = 3$. In larger networks we show that this criterion allows us to cut up to 99% of the vertices.

2. **Cliqueness** The second intuition is that any vertex of a $k$-plex of size $m$ must be included in a clique of a size that depends on $m$. This is confirmed by Corollary 12.2 stating that

![Figure 12.1: An example network.](image-url)
any vertex of a $k$-plex larger than or equal to $m$ is included in a clique of size at least $\lceil m/k \rceil$. Then, if the size of the maximum $k$-plex is at least $\omega$, we can cut out all vertices with cliqueness less than $\lceil \omega/k \rceil$, where we call cliqueness of a vertex the size of the largest clique that includes it. For example, if we are searching for 2-plexes in the network depicted in Figure 12.1a we can filter out all vertices that do not belong to cliques of size at least $\lceil 5/2 \rceil = 3$, that is, the pair of black vertices in the bottom of the network. We will show in Section 12.6 that in larger instances this criterion can be tested efficiently and is able to cut up to 98% of the vertices.

Even if some vertices can be filtered out both because their low cliqueness and low coreness, the network in Figure 12.1a shows that the two filtering criteria are indeed independent. When both criteria are applied, the size of the network is reduced of magnitude and standard techniques for finding $k$-plexes may become applicable even to very large networks.

We push ahead this approach to devise a technique that allows us to further increase the tractable cases. In fact, a consequence of Corollary 12.2 is that if the largest $k$-plex has size $p$, then the network contains a clique of size at least $p/k$, which in turn must be less than the size of the maximum clique $\omega$. This implies that the size $p$ of a maximum $k$-plex in the network cannot exceed $k \cdot \omega$. Hence, the size $p$ of the searched $k$-plexes is in the interval $[\omega, k \cdot \omega]$. In our running example, this would mean to search in the interval $[5, 10]$. Now we iteratively apply the following algorithm:

1. We make an educated guess of a value of $p$ in the interval $[\omega, k \cdot \omega]$ that allows us to filter out many vertices based on the cliqueness criterion.

2. We launch a traditional method to find all $k$-plexes on such reduced network.

3. If we find some $k$-plex of size greater than or equal to $p$ we are done (our guess was correct and we found all maximum $k$-plexes).

4. If we find only $k$-plexes of size $s \in [\omega, p-1]$, then we know that our guess was too optimistic and we iterate, using $p-1$ as new upper bound and using $s$ as new lower bound (and filtering out vertices based on it).

This technique is effective at least whenever the size $p$ of the largest $k$-plex is much larger than the size $\omega$ of the largest clique, which is a reasonable property for some networks.

Once we have found the largest $k$-plexes, we may be interested in finding smaller ones. We have noted in the introduction that an exhaustive search does not make much sense, since very small $k$-plexes are not significant, to the point that they may be even disconnected or composed by a set of isolated vertices. Hence, the second problem we tackle is to find all maximal $k$-plexes in the network of size bigger than a threshold $m$.

As mentioned above, our idea is to start from cliques, which are $k$-plexes but not necessarily maximal, and possibly enlarge them to find maximal $k$-plexes. Building on the cliqueness criterion, which ensures that each vertex of a $k$-plex $C$ of size $s$ is included into a clique of size at least $\lceil s/k \rceil$, we start from each of such cliques $K$. If we set $m \geq k^2$, we have that $|K| \geq \lceil s/k \rceil > k$, which implies that any other vertex of $C$ must be adjacent to at least one vertex of $K$ (in other words, $K$ is a dominating set of $C$). Hence, we can search for $C$ restricting to a block including $K$ and all its adjacent vertices. For example, suppose you are searching for all maximal 2-plexes of size at least 5 in the network in the top of Figure 12.1b. Consider any clique of size at least $\lceil s/k \rceil = \lceil 5/2 \rceil = 3$, for example the clique $K = \{a, b, e\}$ (yellow triangle in Figure 12.1b). The vertices of any $k$-plex of size at least 5 containing $K$ are adjacent to $K$ (surrounded vertices of Figure 12.1b top).
We further reduce the size of the block by proving that $C$ can be obtained by considering only vertices belonging to $K$ and to other cliques of size at least $\lceil s/k \rceil$ intersecting with $K$ (Lemma 12.5). For example, the 2-plex of size 6 on the right of Figure 12.1b is all contained into the clique $\{a, b, e\}$ and three other cliques of size 3 intersecting with $K$ (surrounded vertices of Figure 12.1b bottom). This gives rise to an efficient searching algorithm that decomposes the network into blocks each composed of one clique as the core, and all intersecting cliques as the boundary. Each block can be separately processed, possibly in a distributed environment.

12.3 Finding maximum k-plexes

In this section, we show our algorithms for enumerating the maximum $k$-plexes of the input graph $G$. Intuitively, our approach consists of enumerating all the $k$-plexes of a targeted sub-graph of $G$, that we refer to as $H$, and then selecting the largest ones. Clearly, the smaller is $H$ with respect to $G$, the faster is solving the problem, as long as $H$ contains all the maximum $k$-plexes of $G$. Precisely, we aim at extracting a sub-graph $H$ out of $G$, that is:

1. small enough to make the enumeration fast;
2. large enough to capture all maximum $k$-plexes of $G$, and allow the computation of a correct solution.

We design two different sub-graph extraction criteria, dubbed coreness and cliqueness. Our criteria are of the form “all the $k$-plexes larger than or equal than $m$ consist of vertices with property XYZ”, and have the above desiderata for a suitable choice of $m$. Practically speaking, we make the following observations.

- If $m$ is too small, then the criteria are trivially met by most (or even all) vertices of $G$ (violating the first desiderata);
- If $m$ is too large, then the criteria are met only by few (or even none) vertices of $G$ (violating the second desiderata);

To this end, we show what are the best settings of the threshold $m$ in different scenarios. The theoretical analysis in Section 12.5 proves that the solution provided by our settings is correct, and the experiments in Section 12.6 show that the enumeration on the resulting sub-graph is greatly faster than in the input graph.

In the rest of this section, we first describe our criteria and related concepts, then we describe the corresponding algorithms for enumerating maximum $k$-plexes, based on different choices for the threshold $m$.

12.3.1 Criteria

We now describe our coreness and cliqueness criteria, and how to use them for extracting the sub-graph $H$ out of $G$.

Coreness. Our first criterion is the simplest, and it is based on the intuition that all the vertices of a $k$-plex $C$, with $|C| \geq m$ have degree non-smaller than $m - k$. Clearly, we do not know a priori what are the vertices of $C$. However, we can iteratively filter out any vertex that has degree lower than $m - k$. Formally, this is equivalent to search for the $(m - k)$-cores of $G$. We define this concept in the following.

---

Ideally, $H$ consists exclusively of the vertices of the $k$-plexes we are looking for.
Definition 12.1. An $h$-core of $G$ is a connected subgraph of $G$ in which all vertices have degree at least $h$. A vertex $u$ has coreness $h$ if it belongs to an $h$-core, but not to any $(h+1)$-core.

An $h$-core is one of the connected components of the sub-graph of $G$ formed by repeatedly deleting all vertices of degree less than $h$. We are now ready to state our coreness criterion, as follows.

Criterion 12.1 (Coreness). All the $k$-plexes of $G$ larger than or equal to $m$ consist of vertices having coreness larger than or equal to $m - k$.

Cliqueness. Our second criterion is based on the intuition that all the vertices of a $k$-plex $C$, with $|C| \geq m$, form smaller cliques with other vertices of $C$. Informally speaking, if we try to “draw” a $k$-plex by adding one edge at a time, we soon realize that there are no ways of placing edges without forming progressively larger cliques here and there. Lemma 12.3 in Section 12.3 proves that every vertex of $C$ participates in a clique non-smaller than $m \cdot \omega$. Therefore, we can filter out any vertex that only participates in smaller cliques. We define this concept in the following.

Definition 12.2. A vertex $u$ has cliqueness $h$ if it belongs to a clique of $G$ of size $h$, but not to any clique of $G$ of size $h + 1$.

We are now ready to state our cliqueness criterion, as follows.

Criterion 12.2 (Cliqueness). All the $k$-plexes of $G$ larger than or equal to $m$ consist of vertices having cliqueness larger than or equal to $\frac{m \cdot \omega}{k}$.

Algorithm 34: prune($G, k, m$) algorithm that computes a sub-graph of $G$ according to Criterion 12.1 and 12.2.

1. $G' \leftarrow \{v \in G : G.coreness(v) \geq m - k\}$
2. $H \leftarrow \{v \in G' : G'.cliqueness(v) \geq \frac{m \cdot \omega}{k}\}$
3. return $H$

Computing the sub-graph. Let $m$ be given as input. The procedure prune($G, k, m$), shown in Algorithm 34, returns the sub-graph $H$ resulting from a combination of CORENESS and CLIQUENESS. We first compute the $(m - k)$-cores (line 1), and then we filter out all the vertices with low cliqueness in the cores (line 2). Note that the two criteria can be applied sequentially, in any order. Since computing coreness is easy [BZ03], we chose to apply CORENESS first and compute cliqueness on the smaller graph $G'$ (line 2).

12.3.2 Algorithms for maximum $k$-plexes

Let $\omega$ be the size of any maximum clique in $G$. Criterion 12.2 shows that the size of any maximum $k$-plex lies in the range $[\omega, k \cdot \omega]$. In principle, for the problem at hand, $m$ can be set anywhere in the interval $[\omega, k \cdot \omega]$. With our desiderata in mind (complete solution and fast enumeration), we identify two alternative strategies that make sense with no prior knowledge on the maximum $k$-plexes, that we refer to as the cautious choice and the greedy choice.

• Cautious choice. We set $m = \omega$. The pros are that we cannot miss the maximum $k$-plexes. The cons are that if the size of the maximum $k$-plex is closer to $k \cdot \omega$ than to $\omega$, $H$ may be much larger than needed.
Algorithm 35: max_plexes($G, k$) algorithm.

1. $K \leftarrow \text{all_cliques}(G)$
2. $m \leftarrow \max_{K \in K} |K|
3. H \leftarrow \text{prune}(G, k, m)$
4. $K \leftarrow \text{all_plexes}(H, k)$
5. $P \leftarrow \arg\max_{K \in K} |K|
6. \text{return } P$

Algorithm 36: max_plexes_binary($G, k$) algorithm.

1. $K \leftarrow \text{all_cliques}(G)$
2. $m \leftarrow \max_{K \in K} |K|
3. \text{LB} \leftarrow m
4. \text{UB} \leftarrow k \cdot m$
5. \text{while } \text{LB} \neq \text{UB} \text{ do}
   6. \hspace{1em} $x \leftarrow \frac{\text{LB} + \text{UB}}{2}$ \hspace{1em} // Pivot
   7. $H \leftarrow \text{prune}(G, k, x)$
   8. $K \leftarrow \text{all_plexes}(H, k)$
   9. $x' \leftarrow \max_{K \in K} |K|
10. \text{LB} \leftarrow \max\{\text{LB}, x'\}
11. \text{UB} \leftarrow \max\{x, x'\}
12. \text{return } P$

- **Greedy choice.** We set $m$ in the middle. The pros are that $H$ is very small. The cons are that we may miss the maximum $k$-plex. To this end, we can iterate in a binary search fashion.

Let us introduce the following useful methods.

- **all_cliques($G$):** The method returns all the maximal cliques of $G$.
- **all_plexes($H, k$):** The method returns all the maximal $k$-plexes of the sub-graph $H$, for a given $k$, with any appropriate state-of-art method.

**Cautious choice** Our first strategy is illustrated in Algorithm 35 that we refer to as max_plexes(). The procedure is very simple. We first enumerate all the maximal cliques of $G$ (line 1), and then extract the sub-graph $H$ using our criteria and the maximum clique size as threshold (line 2–3). Finally, we enumerate all the maximal $k$-plexes of the sub-graph $H$, and return the maximum ones (line 4–5). Note that some cliques can be included in the solution. We observed that some instances in our experiments only have one maximum clique $C_{\text{max}}$, that also corresponds to the maximum $k$-plex. In such extreme cases, the sub-graph $H$ only consists in $C_{\text{max}}$, and we can even do without all_plexes().

**Greedy choice.** Our second strategy is illustrated in Algorithm 36 that we refer to as max_plexes_binary(). We first enumerate all the maximal cliques of $G$ (line 1), as in max_plexes(). After that, rather than using the maximum clique size as threshold, the algorithm attempts to extract a smaller sub-graph by setting $m$ to the middle point of the range $[\omega, k \cdot \omega]$ (line 6–7). Then, we enumerate all the maximal $k$-plexes of the sub-graph $H$, and compute the maximum size $x'$ (line 9). If $H = \emptyset$ we set $x' = 0$. The following scenarios are possible.
• If $x' \in [0, x)$, then there are no $k$-plexes larger than the pivot $x$ and the search continues. The lower-bound does not change (line 10), and the pivot becomes the new upper-bound (line 11).

• If $x' \geq x$, then we found $k$-plexes non-smaller than the pivot $x$. The search comes to an end, since the value $x'$ becomes both the new lower-bound (line 10) and the new upper-bound (line 11).

In particular, if $x' \geq x$, then $x'$ is also the maximum $k$-plex size of $G$ by our criteria. Equivalently, all the maximum $k$-plexes of the input graph $G$ must be included in $K$ (line 8). Therefore, at the end of the search, we return the maximum elements of $K$ (line 13–14).

Note that if we replace the selection of $x = \frac{LB + UB}{2}$ with $x = \omega$ we obtain the same behavior than \texttt{max_plexes()} algorithm.

12.4 Finding large $k$-plexes

Let $\omega_k$ be the size of any maximum $k$-plex of $G$, as computed for instance with the algorithm \texttt{max_plexes()} in the earlier section. A natural next step is to find all the $k$-plexes within a range $(1 - \epsilon)\omega_k$. One may be tempted to re-use the approach of \texttt{max_plexes()} as a template, and change line 2 of Algorithm 35 by setting $m = (1 - \epsilon)\omega_k$. While this is valid in principle, we observed in practice that large $k$-plexes are more numerous than maximum $k$-plexes (that in most case only consist of a single max $k$-plex) even if $\epsilon$ is small. On the one hand, this confirms our initial intuition that enumerating all the $k$-plexes of $G$ is impractical. On the other, this calls for the design of an efficient algorithm specific for the large $k$-plexes problem at hand. To this end, we introduce a third criterion, that we refer to as \texttt{OVERLAPPINGCLIQUES}, which enables the enumeration of large $k$-plexes in networks that are orders of magnitude larger than previously considered.

**Overlapping cliques criterion.** This is an advanced application of the cliqueness criterion. Let $C$ be a $k$-plex non-smaller than $m$. Consider any maximal clique $K \in C$. We know from the cliqueness criterion that $|K| \geq m k$.

- If $m k \geq k$, every vertex of $C \setminus K$ must be adjacent to at least one vertex $v \in K$, since every vertex of a $k$-plex can miss up to $k - 1$ neighbors.

- Every vertex of $C \setminus K$ must itself participate in a clique $K'$, at least as large as $\frac{m k}{k}$.

Intuitively, for large enough $k$-plexes with $m \geq k^2$, all the vertices of $C$ not in $K$ participate in cliques overlapping with $K$, that is, sharing at least one vertex with $K$. (We provide a proof of this statement in Lemma 12.3) This allows us to target our search for large $k$-plexes to the neighborhood of each clique $K$. More formally, to all the cliques overlapping with $K$, that we refer to as its boundary. We state the overlapping cliques criterion as follows.

**Criterion 12.3 (OverlappingCliquess).** All the $k$-plexes of $G$ with size $m \geq k^2$ consist of vertices either belonging to a clique $K$ s.t. $|K| \geq \frac{m}{k}$, or to overlapping cliques $K'$, s.t. $|K'| \geq \frac{m}{k}$ and $K \cap K' \neq \emptyset$.

**Algorithm for large $k$-plexes.** Our algorithm for large $k$-plexes is illustrated in Algorithm 37 that we refer to as \texttt{large_plexes()}. The procedure uses the same auxiliary methods than \texttt{max_plexes()} (see Section 12.3), and the \texttt{boundary}(K) primitive in addition.

- \texttt{boundary}(K): The method returns all the vertices included in a given clique $K$, or in overlapping cliques.
Algorithm 37: large_plexes\((G, k, m)\) algorithm.

\begin{verbatim}
1 \(H \leftarrow \text{prune}(G, k, m)\)
2 \(K \leftarrow \text{all_cliques}(H)\)
3 for \(K \in K\) do
4 \(B \leftarrow \text{boundary}(K)\)
5 \(P \leftarrow \text{all_plexes}(H[B], k) // H[B] is the subgraph of \(H\) induced by \(B \subseteq V\)
6 for \(P \in P, |P| \geq m\) do
7 \(\text{yield } P // \text{duplicates admitted}\)
\end{verbatim}

Algorithm 38: boundary\((K)\) algorithm.

\begin{verbatim}
1 \(B \leftarrow \emptyset\)
2 for \(K' \in K\) do
3 if \(K' \cap K \neq \emptyset\) then
4 \(B \leftarrow B \cup K'\)
5 \return B
\end{verbatim}

The large_plexes() algorithm first extracts the sub-graph \(H\) using our criteria and the input threshold \(m = (1 - \epsilon)\omega_k\) (line 1). Note that prune() selects all the vertices of \(G\), except those that do not participate in any \(k\)-plex larger than or equal to \(m\). Then, it enumerates all the maximal cliques of the sub-graph \(H\) (line 2). The resulting set \(K\) (line 2) only consists of cliques that are larger than \(m_k\) (because of the constructive process of \(H\)) and thus can be used as “seeds” for growing large \(k\)-plexes, according to the OVERLAPPING_CLIQUE criterion. Finally, the large_plexes() algorithm iterates over \(K\) (lines 3–9) and for each clique considers its boundary (line 4). Let \(B\) the current set returned by boundary(). We first enumerate all the maximal \(k\)-plexes of the sub-graph of \(H[B]\) of \(H\) induced by \(B\) (line 5). Then, we return only \(k\)-plexes non-smaller than \(m\) (lines 6–8), and proceed with the next clique.

In Algorithm 38 we show a simple implementation of the boundary() algorithm. At line 2 the data structure \(K\) is shared with the caller method (i.e., large_plexes()).

Duplicates. Note that the above method can return the same \(k\)-plex multiple times (line 7). To this end, we design a clever test, that guarantees that every \(k\)-plex is returned at most once. Let \(C\) be any \(k\)-plex computed by all_plexes\((H[B], k)\). We define the concept of parent clique \(P(C)\), and return \(C\) only when the current clique is equal to \(P(C)\). Specifically, let

- \(\text{min}(C)\) be the vertex \(u\) in \(C\) with smallest id;
- \(\text{complete}(X, Y)\) be a method that iteratively adds to the clique \(X\) the next minimum vertex in the set \(Y\) s.t. \(X\) is still a clique;

We define \(P(C)\) by construction as in the following equation.

\[
P(C) = \text{complete}(\text{complete}(\{\text{min}(C)\}, C), H)
\]  

Practically speaking, we start from the vertex \(u\) in \(C\) with smallest id. Then, the process of construction has two phases. In the first phase, we extend \(u\) within \(C\) in increasing order of id. Then we keep extending by selecting vertices from the whole \(H\).

We can thus rewrite line 6 as “\textbf{if } \(P(P) = K \text{ then yield } P\):” this way each \(k\)-plex is returned exactly once.

211
Parallelism. We note that the main operations introduced so far, can be efficiently executed in a distributed setting. Cliques of the input graph and cliqueness values of vertices can be computed in a cluster environment, as shown in [CVM+16b]. In addition, for large_plexes(), different machines can process different clique boundaries in parallel. We are not aware of parallel techniques for implementing the all_plexes() sub-routine, and we leave this for future work.

12.5 Theoretical basis

A k-plex $C = (V, E)$ with $s$ vertices, $s \geq 2k$, is a graph such that every vertex is adjacent at least to all other vertices in $C$ except $k$. For the sake of simplicity $C$ may refer to both the set of vertices it contains and to the induced graph. We refer to vertices adjacent to a given vertex $u$, as neighbors of $u$. A clique can be thought of as 1-plex. Any subset of a k-plex is also a k-plex, and a k-plex is also a $k + 1$-plex. We need to prove the three criteria described in the previous sections, dubbed coreness, cliqueness and Overlapping Cliques.

Coreness criterion. Let $\Delta(C)$ denote the diameter of $C$, that is, the largest number of vertices which must be traversed in order to travel from one vertex to another. While a clique, or a 1-plex, has diameter equals 1, k-plexes with $k > 1$ come in a variety of forms and can have arbitrarily high diameter (which is not a desirable property for a community). However, for $k \leq \frac{s}{2}$ – which means that every vertex is adjacent more than half vertices in $C$ – the diameter is only at most 2. This is proven in the following.

**Lemma 12.1.** A k-plex $C$ with $|C| \geq 2k$ has diameter $\Delta(C) \leq 2$.

**Proof.** If $C$ has diameter larger than 2, there are at least two vertices $u$ and $v$ at distance more than 2. Since $u$ is missing at most $k$ edges, it has at least $|C| - k$ neighbors. However, $v$ is not connected to neither itself, $u$ nor its neighbors, and therefore it is missing at least $|C| - k + 2$ edges, which is larger than $k$ if $k \leq \frac{|C|}{2}$. A contradiction. □

**Corollary 12.1.** A k-plex $C$ with $|C| \geq 2k$ then $C$ is connected.

We are now ready for proving the coreness criterion.

**Lemma 12.2 (Coreness).** Every vertex in a k-plex $C$ has coreness at least $|C| - k$.

**Proof.** It is easy to verify that for every vertex $u \in C$, the degree of $u$ is at least $|C| - k$. Since $C$ is connected by Corollary [12.1] recalling Definition [12.1] $C$ is a $|C| - k$-core. □

Other criteria. We give the technical lemma below, that we use for deriving the cliqueness and adjacency criteria, and the advanced search principle.

**Lemma 12.3.** Given a k-plex $C$, with $|C| = s$, every clique $X \subseteq C$ s.t. $|X| < \frac{s}{k}$ is included in a bigger clique $X_{big}$, s.t. $|X_{big}| \geq \frac{s}{k}$.

**Proof.** Let $X \subseteq C$ be any clique of $C$, s.t. $|X| < \frac{s}{k}$. Let $N \subseteq C$ be the set of vertices which are not adjacent to all vertices of $X$, that is, that are adjacent from 0 to $|X| - 1$ vertices of $X$. By picking any vertex $u' \in C \setminus (X \cup N)$, we have that $X' = X \cup \{u'\}$ is a clique of size $|X| + 1$. Since every $u \in X$ can miss at most $k$ neighbors including itself, $|N \cup X| \leq |X|(k - 1) + |X| = k|X|$. This means that at most $k|X|$ vertices are excluded for the selection of $u'$. Let $N'$ be the vertices not adjacent to all vertices of $X'$. We can repeat the process and grow $X'$, by picking any vertex $u'' \in C \setminus (X' \cup N')$, until we run out of vertices. Note that the newly-excluded vertices for selecting $u''$ are $u'$ and its missing neighbors. Such a clique-growing process can be thought of as an iterative process starting from a vertex and growing a clique – as if $X$ itself were grown after $|X|$ steps of the process – and excluding at most $k$ vertices at a time. Therefore, the process will run at least $\frac{s}{k}$ steps, after which $X$ has been grown to $\frac{s}{k}$ vertices. □

212
Table 12.1: Real-world networks from [LoAoGN] used in our experiments, sorted by number of vertices.

Note that, in case $\frac{s}{k}$ is not integer, the proof yields $|X_{\text{big}}| \geq \lceil \frac{s}{k} \rceil$.

The cliqueness and the adjacency criteria directly follow.

**Corollary 12.2** (cliqueness). Every vertex in $C$ has cliqueness at least $\lceil \frac{s}{k} \rceil$.

**Lemma 12.4.** Consider a clique $X \subseteq C$, s.t. $|X| \geq \frac{s}{k}$. If $s \geq k^2$, every vertex in $C$ either belongs to $X$ or has a neighbors in $X$.

**Proof.** We know that such a clique always exists from Lemma 12.3. Since its size is at least $k$ by assumption, then every $u \in C \setminus X$ has to be adjacent to at least one vertex in $X$. \hfill $\square$

Our last criterion is a stricter form of the above lemma, which we formalize as follows.

**Lemma 12.5** (OverlappingCliques). Consider a clique $X \subseteq C$, s.t. $|X| \geq \frac{s}{k}$. If $s \geq k^2$, every vertex in $C$ either belongs to $X$ or to an overlapping clique $X'$, s.t. $|X'| \geq \frac{s}{k}$ and $X \cap X' \neq \emptyset$.

**Proof.** Let $u$ be any vertex of $C \setminus X$. We know from Lemma 12.4 that exists a vertex $v \in X$ adjacent to $u$. Since $\{u, v\}$ is a clique of size 2, we can apply Lemma 12.3 and conclude that both vertices belong to a clique $X'$ larger than or equal to $\frac{s}{k}$. Finally, $v \in X \cup X'$. \hfill $\square$

### 12.6 Experiments

In this section, we compare our and previous algorithms over different real-world networks, and show the advantages and limitations of our approach.

**Datasets.** We considered a mix of real-world networks with various sizes and characteristics. All our networks are publicly available on the LASAGNE meta-repository [LoAoGN], and come from different human activities. Specifically, we consider:

- small to large collaboration networks, where vertices represent authors of published papers or books, and edges represent co-authorship;
- a large citation network, where vertices represent published papers or books, and edges represent citations;
Implementation details. We implement the auxiliary methods used by our algorithms, with the most recent methods in literature for the corresponding tasks, to the best of our knowledge.

- In all\_cliques(G), we use the algorithm in [CVM+16b] for enumerating all the maximal cliques of the input graph G.

- In all\_plexes(H, k), we use the algorithm in [BCK15] for enumerating all the maximal k-plexes of the sub-graph H.

We implement CORENESS and CLIQUENESS criteria, respectively, with the method in [BZ03] and the already mentioned algorithm in [CVM+16b]. The implementation of [BCK15] was kindly provided by the authors, which we also thank for useful discussions on the subject of this chapter. The source code of our approach is available at [CFM+].

Methodology. We compare the execution of our methods for enumerating targeted k-plexes, with the most recent method for enumerating all k-plexes of the input graph [BCK15]. In other words, we compare the algorithms max\_plexes(G, k), max\_plexes\_binary(G, k), large\_plexes(G, k, m) with the algorithm all\_plexes(G, k) over the same input graph, for different values of k and threshold m. To the best of our knowledge, indeed, there are no faster approaches than [BCK15] that are specific for maximum k-plexes and k-plexes non-smaller than a threshold. Targeted enumeration is indeed our contribution.

Test environment. Our experiments were performed on a machine with two CPU Intel Xeon E5-520 units with 4 cores each, running at 2.26GHz, with 8MB of cache and 32GB RAM. The operating system was Linux CentOS 6.7, with kernel version 2.6.32, Java Virtual Machine version 1.8.0_111 (64-Bit) and Python version 2.6.6 (64-Bit). All our executions have a reasonable 6 hours timeout, after which they are interrupted. In the experiments, we show that even the smallest networks with few thousands vertices timeout with traditional methods. Our algorithms, instead, can process networks up to hundreds of thousands of vertices.

12.6.1 Results

In Figure 12.2, we show the number of vertices in the sub-graph H computed by the algorithm prune(G, k, m), with different values of k and m = $\omega$, i.e, the maximum clique size. As frame of comparison, we show the number of vertices in G (i.e., n). The figure shows that, except
Figure 12.3: Number of vertices (log scale) surviving the CORENESS criterion.

Figure 12.4: Number of vertices (log scale) surviving the CLIQUENESS criterion.

Figure 12.5: Fraction of vertices surviving CORENESS and CLIQUENESS.
for newm where $\omega = 3$, the sub-graph $H$ is order of magnitudes smaller than $G$. Notably, for different networks (jazz, geom, hepPh, newm, and dblp), $H$ is left with only the vertices of the maximum $k$-plex. In such lucky cases we can even skip the execution of the enumeration step all_plexes($H, k$). Since the sub-graph produced by a given $k$ is included in the sub-graph produces by $k + 1$, it is not surprising that higher values of $k$ yield more vertices in $H$. (Remember that for $k = n$ we have $G = H$.) However, for most instances, the sub-graph is small with respect to $G$ (thus allowing for faster enumeration of $k$-plexes) for different values of $k$. Finally, our criteria have little impact on the newm network, because its maximum clique size is only 3.

Figures 12.3 and 12.4 report the number of vertices of $G$ residual after the coreness and cliqueness criteria, applied separately. Figure 12.5 shows the fraction of vertices of $G$ residual after both criteria, applied together as in our prune() algorithm. For the considered networks, most vertices are filtered out by coreness. Then, the structures that are too connected to be filtered by coreness but too small to play a role in the search for $k$-plexes non-smaller than $\omega$, are filtered out by cliqueness. Such an additional cliqueness step has bigger impact in advogato and patents. In principle, our coreness and cliqueness criteria can be applied repeatedly, in alternation, until there are no more vertices to filter out. In practice, we observed that after three iterations (that is, coreness–cliqueness–coreness–cliqueness–coreness–cliqueness) there were no additional vertices to filter out of networks in our dataset, and that only few vertices (i.e., less than 10) were removed from second iteration on.

Maximum $k$-plexes. In Table 12.2 we show running times for different steps of max_plexes($G, 2$), compared to the time required for enumerating all 2-plexes (column “FULL ENUM”), over the same input graph.

<table>
<thead>
<tr>
<th>GRAPH</th>
<th>FULL ENUM</th>
<th>OUR APPROACH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CORE</td>
</tr>
<tr>
<td>jazz</td>
<td>2 h</td>
<td>0.008 s</td>
</tr>
<tr>
<td>grQc</td>
<td>&gt; 6 h</td>
<td>0.001 s</td>
</tr>
<tr>
<td>geom</td>
<td>&gt; 6 h</td>
<td>0.001 s</td>
</tr>
<tr>
<td>advogato</td>
<td>&gt; 6 h</td>
<td>0.003 s</td>
</tr>
<tr>
<td>hepPh</td>
<td>&gt; 6 h</td>
<td>0.001 s</td>
</tr>
<tr>
<td>astroPh</td>
<td>&gt; 6 h</td>
<td>0.046 s</td>
</tr>
<tr>
<td>newm</td>
<td>&gt; 6 h</td>
<td>0.22 s</td>
</tr>
<tr>
<td>mathSci</td>
<td>&gt; 6 h</td>
<td>0.009 s</td>
</tr>
<tr>
<td>dblp</td>
<td>&gt; 6 h</td>
<td>0.005 s</td>
</tr>
<tr>
<td>patents</td>
<td>&gt; 6 h</td>
<td>1.148 s</td>
</tr>
</tbody>
</table>

Table 12.2: Running time for finding all the largest 2-plexes.
The time for computing our criteria has little impact on the overall running time, which is dominated by the enumeration of 2-plexes of the residual sub-graph when necessary. As a consequence, in all the networks where \( H \) is left with only the vertices of the maximum clique, which is in turn also the maximum 2-plex, the computation of \( \text{max}_\text{plexes}() \) ends successfully after fractions of seconds. For such networks, we write “no need” in the “enum” column. As frame of comparison, full enumeration of 2-plexes (i.e., as in \[\text{BCK15}\]) requires hours even on our smallest network (jazz), and times out on the other networks.

When \( H \) is not as minimal as in the networks with no need for \( \text{all}_\text{plexes}(H, k) \), the running time of \( \text{max}_\text{plexes}() \) still ranges from fractions of seconds (most networks) to 2 h, proportionally to the size of \( H \) (see Figure 12.2 for comparison). In general, we can conclude that the running time of our algorithm mostly depends on the size of the sub-graph computed by \( \text{prune}() \). The only two instances that requires more than 6 h are indeed the nemw and patents networks, that correspond to the top two largest filtered sub-graphs (see Figure 12.2).

We observed that the results for higher values of \( k \), namely \( k = 3 \) and \( k = 4 \), are similar. This is because the sub-graph \( H \) produced with higher valued of \( k \) contains only few more vertices than the sub-graph produced with \( k = 2 \), as shown in Figure 12.2.

**Binary search.** Our “cautious choice” strategy for the maximum \( k \)-plexes is able to process quickly all the networks in Table 12.2 except nemw and patents. For the nemw network, we observe that \( \omega = 3 \) (see Table 12.1) and the size of the maximum 2-plex is \( \omega_2 = 5 \), that is, \( \omega_2 \) is closer to \( 2 \cdot \omega \) (i.e., the theoretical maximum size for a 2-plex) than to \( \omega \). This makes the nemw network a good candidate for a “greedy choice” as in the \( \text{max}_\text{plexes}_\text{binary}() \) algorithm described in Section 12.3.2. In practice, the value of \( \omega_2 \) is not known a priori. However, we can decide for the greedy choice also by observing that \( \omega \) is small. We observed experimentally for the nemw networks that the computation of \( \text{max}_\text{plexes}_\text{binary}(G, 2) \) terminates after only one iteration in less than our time out. Instead, the network patents is hard to process even with the greedy choice. To improve on this result is a challenging future task.

**Large \( k \)-plexes.** In Table 12.3, we show the overall running time of \( \text{large}_\text{plexes}(G, k, m) \) on different networks in our dataset, for different values of \( k \). For this experiment, we set \( m = 0.8\omega_k \), where \( \omega_k \) is the maximum \( k \)-plex size, as computed by \( \text{max}_\text{plexes}(G, k) \). The time required for enumerating all \( k \)-plexes (column “FULL ENUM”) of such networks is always larger than our timeout (6 hours). The table also show the number of \( k \)-plexes returned (column “FOUND”). All the networks considered contain less than a dozen \( k \)-plex non-smaller than \( 0.8\omega_k \), which are quickly found by our algorithm in most cases. Note that in this experiment we call \( \text{prune}(G, k, 0.8\omega_k) \), that is, we compute the sub-graph \( H \) using a different threshold than in

<table>
<thead>
<tr>
<th>GRAPH</th>
<th>( k )</th>
<th>FULL ENUM</th>
<th>OUR APPROACH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>grQc</td>
<td>2</td>
<td>&gt; 6h</td>
<td>4.65 s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>&gt; 6h</td>
<td>2.7 s</td>
</tr>
<tr>
<td>astroPh</td>
<td>2</td>
<td>&gt; 6h</td>
<td>5 h 44 m</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>&gt; 6h</td>
<td>&gt; 6h</td>
</tr>
<tr>
<td>mathSci</td>
<td>2</td>
<td>&gt; 6h</td>
<td>2.75 s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>&gt; 6h</td>
<td>&gt; 6h</td>
</tr>
</tbody>
</table>

Table 12.3: Running time for finding all \( k \)-plexes larger than 80% of the maximum clique.
The \textit{astroPh} network requires much more time than other networks. To this end, for this specific network, we plot in Figure \ref{fig:running_time} the overall running time of \texttt{large\_plexes}(G, 2, m) (i.e., for 2-plexes), for \( m \) ranging in \([0, \omega_2]\). We computed using \texttt{max\_plexes}(G, 2) that \( \omega_2 = 57 \). On the one side of the spectrum, for \( m = 0 \), all the 2-plexes ought to be returned (i.e., the same result than \texttt{all\_plexes}()). On the other side, for \( m = \omega_2 \), only a single 2-plex – the maximum 2-plex – is returned. The plot shows that the running time quickly decreases from hours to seconds as \( m \) approaches to \( \omega_2 \). The data-point close to 5 hours corresponds to \( m = 0.8\omega_2 \), as in Table \ref{tab:max_2_plexes}. We compare these results with the cumulative distribution function (CDF) of 2-plexes of \textit{astroPh}, as reported in Figure \ref{fig:2_plexes_CDF}. As frame of comparison, we also show CDF of cliques (that are 1-plexes). As expected, most 2-plexes have size smaller than 10.\footnote{The figure may be counter-intuitive to read, as while the portion of 2-plexes larger than 10 is smaller than the one of cliques, the absolute number is still larger.} For \( m > 0.8\omega_2 \), which represents the 99th percentile for 2-plexes, the running time of \texttt{large\_plexes}() is even smaller than half an hour, confirming the effectiveness of our targeted enumeration system.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{running_time.png}
\caption{(a) Running time.
\includegraphics[width=0.5\textwidth]{2_plexes_CDF.png}
\caption{(b) CDF of 2-plexes.
}
\end{figure}

Figure \ref{fig:running_time} (possibly \( 0.8\omega_k \geq \omega \)).
12.7 Related works

In the field of network analysis, dense substructures in graphs (aka dense subgraphs) are associated with communities, or more in general sets of closely related elements [For10, PYB12]. The problem of finding these substructures has been extensively studied for decades, and continues to be the object of cutting edge research. The simplest and most rigorous definition of dense subgraph is the clique, i.e., a subgraph in which all vertices are pairwise connected. Many algorithms for finding all maximal cliques have been developed, most of them being inspired to the Bron-Kerbosh algorithm [BK73], such as [ELS13, TTT06] or to the more recent paradigm of reverse search [AF96], such as [MU04, CR15, CGMV16]. The definition of clique may be too strict in some instances, such as in real datasets where data can be noisy and incomplete, so several definitions of pseudo-clique have been produced [PYB12], such as the $k$-plex [SF78].

McClosky [MH12] performs a thorough study to devise exact algorithms for finding the largest $k$-plex, and heuristics for finding lower upper bounds on its size, exploiting co-$k$-plexes (i.e., $k$-plexes on the complement graph) and graph coloring techniques. The usability of the algorithms for finding the largest $k$-plex is however limited to small networks, as the running time exceeds the hour for graphs with hundreds of vertices.

Wu et al. [WP07] propose Pemp, a parallel algorithm for enumerating $k$-plexes, which successfully improves its performance with the usage of multiple cores.

Cohen et al. [CKS08] give a generic framework for enumerating all maximal subgraphs with respect to hereditary and connected hereditary graph properties, i.e., properties that are closed with respect to induced subgraphs and connected induced subgraphs, respectively. Berlowitz et al. [BCK15] apply the framework in [CKS08], together with insights on the $k$-plex problem, to produce efficient algorithms for the enumeration of maximal $k$-plexes and maximal connected $k$-plexes, which are respectively hereditary and connected hereditary. The algorithm for connected $k$-plexes in [BCK15] outperforms the other state of the art algorithms for enumerating or finding the largest $k$-plex, and constitutes our baseline for the experimental evaluation. More quasi clique models have been considered, many of which can be found in this survey by Pattillo et al. [PYB12].

While effort is often devoted to finding the largest communities, the work by Zhai et al. [ZHOT16] stands out in the opposite direction: they remark that small communities are also important in some cases, as long as they are recognized properly. They define a variant of the $k$-plex, adding connectivity constraints (called CLB), and obtain a new quasi clique model for which even small communities are meaningful and densely connected. The additional constraints also allow them to produce a well performing enumeration algorithm for the problem, which makes for a powerful network analysis tool, especially when small and medium sized communities are desired.

Real-world networks can often be large, with millions of vertices and billions of edges. However, algorithms for finding dense subgraphs tend to have high computational complexity, and the number of solutions can be exponential in the worst case [TTT06]. Thus, a great amount effort was dedicated to find ways to process these difficult networks.

Some have proposed decomposition approaches to limit the memory usage: Cheng et al. [CZKCI2] propose a decomposition approach for enumerating maximal cliques with memory proportional to the largest degree in a graph. As this can still be large even in sparse networks, Conte et al. [CVM+16b] proposed a new decomposition approach whose memory is proportional to the degeneracy of a graph, exploiting the fact that this is small in real-world networks. Others, such as Zhai et al. [ZHOT10], exploit properties specific to the considered quasi-clique model to prune the search space.
12.8 Final remarks

This chapter shows a novel approach to the enumeration of large $k$-plexes, which are a formal and meaningful way to define interesting communities in real-world networks that generalizes the notion of clique. Two main clues have driven our solution:

- a large portion of the network can be filtered out before starting the detection
- cliques, which can be computed more efficiently, can be used as starting points for the search of $k$-plexes in the network. The efficiency of the approach over state-of-the-art algorithms has been confirmed by our experiments.

Interesting directions for future work are to extend the applicability of our approach by tackling the combinatorial explosion derived from increasing $k$, as well as the problem of computing large $k$-plexes on real-world networks with millions of vertices, and experimenting with a variety of networks coming from different domains, such as (but not limited to) biological networks, web graphs, and product co-purchasing networks.
Bibliography


221


Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2005.


234


235


236
[TBG+13] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo,
and Maria Tsiarli. Denser than the densest subgraph: extracting optimal quasi-

[Tel17] The Telegraph. Average twitter user is an an american woman with an iphone
html, 2012 [Online; accessed October 2017].

[TIAS77] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new
algorithm for generating all the maximal independent sets. SIAM J. Comput.,

for finding a maximum clique with computational experiments. Journal of Global

[TKS02] Pang-Ning Tan, Vipin Kumar, and Jaideep Srivastava. Selecting the right
interestingness measure for association patterns. In Proceedings of the eighth
ACM SIGKDD international conference on Knowledge discovery and data mining,

[Tre54] Horace M. Trent. A note on the enumeration and listing of all possible trees
in a connected linear graph. Proceedings of the National Academy of Sciences,


[TSS02] Amos Tanay, Roded Sharan, and Ron Shamir. Discovering statistically significant

[TTT06] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time
complexity for generating all maximal cliques and computational experiments.

[TY84] Robert E Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test
chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic

[UAUA03] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. Lcm: An efficient

[UKA04] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2: Efficient mining

[UKBM11] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The

1976.

In Algorithms and Computation, 12th International Symposium, ISAAC 2001,
Christchurch, New Zealand, December 19-21, 2001, Proceedings, pages 367–379,


