Graphs and Trees: cycles detection and stream segmentation

Lorenzo Cioni
Dipartimento di Informatica
Largo Pontecorvo 3
Pisa
lcioni@di.unipi.it
Main topics of the talk

- Two algorithms:
  - “segmentation” of a stream of data
    - application: syllabification of written Italian
  - cycles detection in directed (even complete) graphs
    - application: graphic editor for graphs editing with cycles detection, designed to be used for the definition of directed graphs used in causal loop graphs (system dynamics)
Trees and “segmentation” 1

- “RB-tree” (paper of mine, 1996): syllabification of written Italian;
  - binary tree;
  - recursive calls;
  - vowels/consonants.
- “m-ary tree”, under development
  - $m$ direct descendants of each node;
  - recursive calls;
  - alphabet of $m$ distinct symbols.
Trees and “segmentation” 2

Lexicon

- A alphabet, $|A|=m$
- $S$ strings on $A$, length $n$, potentially unbound
- $S=\{s^+ \mid s \in A\}$
- $M$, set of markers
- $M \cap A = \emptyset$
- $R$, rules, fixed and finite set, $R=\{r_1, \ldots, r_k\}$
- $W$, weights, fixed and finite set, $W=\{w_1, \ldots, w_k\}$
- $r_i \leftrightarrow w_i$ one-to-one correspondence

Lexicon/Syntax

- $A=\{\alpha_0, \ldots, \alpha_{m-1}\}$
- $\beta=m$ base of the “numbering” scheme
- $\hat{W}=\{w_j \mid w_j=\sum_{i=0,k} \beta_i\}$
- $k=[0, \ldots, K]$, $K$ max number of elements of a rule
- $W \subset \hat{W}$ identifies the set of weights to which corresponds a rule
- $m$-ary tree dynamically built, no permanent data structures
- rules as weights

AIRO 2006 Cesena 12-15 Settembre 2006
Trees and “segmentation” 3

- input stream “in in in ... in ... in ...” with $i_n \in A$

- $\hat{w} = \sum_{i} i_n \beta^i$ where $i_n$ is a coding of each input symbol (cf. the examples)

- if $\hat{w} \in W$ then it identifies a rule $r_j$

- formally we have: we define a running sum $\hat{w} = \sum_{i} i_n \beta^i$ and, at each step, we check if $\hat{w} \in W$ or not. In the former case we apply the corresponding rule $r_j$ otherwise we step to the following input symbol and update $\hat{w}$. 
- evaluation of $\hat{w}$

- check if $\hat{w} \in W$, cost $O(1)$, two cases:
  - no: we increment $k$ (the pointer on the input stream) by one ($k=k+1$) and evaluate a new value of $\hat{w}$ as $\hat{w}=\hat{w}+\text{in}_k \beta^k$
  - yes: we apply the corresponding rule $r_i$ and reset to zero both the pointer $k$ and the running sum $\hat{w}$

- the application of a rule is equivalent to the insertion of:
  - one marker:
    - after, segmentation
    - inside, “syllabification”
  - two markers, one before and one after, extraction
Properties of the rules

- **Uniformity:**
  - we have uniformity if every rule involves the same number $\sigma$ of input symbols otherwise we have a value $\sigma_{\text{max}}$ that define the longest rule[s]

- **Completeness:**
  - a set of rules is complete if there is a rule for every value of the running sum from 0 to $\beta^\sigma - 1$

- There is no relation between the two properties

- Uniformity and completeness translate in properties of the structure of the m-ary tree, dynamically built
Execution of the rules

- if $\exists w_i \leftrightarrow r_i$ then
  - apply $r_i$
  - reset pointer $k$ and running sum
- else
  - $k = k+1$
  - update running sum with the current input symbol
- apply = insert, in the proper position[s] one or two markers
Example 1

- $A=\{a, b\}$, $b \leftrightarrow 0$, $a \leftrightarrow 1$, $M=\{\ast\}$, $S=\{s^+ \mid s \in A\}$

- completeness and uniformity

- $r_0 = bb \ \leftrightarrow 0$, $r_1 = ab \ \leftrightarrow 1$, $r_2 = ba \ \leftrightarrow 2$, $r_3 = aa \ \leftrightarrow 3$

- $R=\{r_0, r_1, r_2, r_3\}$

- $W=\{w_0, w_1, w_2, w_3\}$

- $\forall w_i \exists$ a rule that defines where to insert the marker[s]

- $r_0, r_3$ marker inside $aa \rightarrow a* a$, $bb \rightarrow b*b$

- $r_1, r_2$ marker after $ab \rightarrow ab*$, $ba \rightarrow bab*$

- abbbbaaab... $\rightarrow$ ab*b*ba*aab*...
Example 1 continued

- $A = \{a, b\}$, $b \leftrightarrow 0$, $a \leftrightarrow 1$, $M = \{\ast\}$, $S = \{s^+ \mid s \in A\}$

  - no completeness but uniformity

  - $r_0^{bb} w = 0$; $r_3^{aa} w = 3$

  - $r_0^{b}, r_3^{a}$ marker before and after

  - $abbbaaab... \rightarrow ab^{*bb}{**a}^{a}ab...$

- no completeness and no uniformity

  - conflicting rules?

  - used in “syllabification”

AIRO 2006 Cesena 12-15 Settembre 2006
Example 1 continued

- \( r_0 = ab \ w_0 = 1; \ r_0 = aba \ w_0 = 5 \) conflict: the former “covers” the latter

- no completeness no uniformity (cf. figure)

- \( r_1 = ab \ w_1 = 1; \)

- \( r_2 = bab \ w_2 = 2 \)

- \( r_2 = bba \ w_2 = 4 \)
Example 2: syllabification

- **Main features:**
  - marker “inside”
  - no uniformity
  - completeness
  - lookahead & stepback vs. stepforward

- **Two versions:**
  - simple version: “small” alphabet, complex rules;
  - complex version: “big” alphabet, simpler rules
Example 2 continued

- **Simple version:**
  - alphabet A: vowels V (with/without stress), consonants C, separators S (spaces, tabs) and punctuation marks P
  - \( A = V \cup C \cup S \cup P \)
  - markers \( M = \{-\} \)
  - weights \( W = \{w_i\} \)
  - rules \( R = \{r_i\} \)
    - \( r_i \leftrightarrow w_i \)

- **Rules**
  - rules are applied with a lookahead and a stepback with a recursive call
  - rules are complex since we have to discriminate many cases within a rule
Example 2 continued

• Some rules of simple version:

- $v \in V$, $v \leftrightarrow 1$, $c \in C$, $c \leftrightarrow 0$

- $w = 10 \leftrightarrow r_{10} = cvcv \text{ cv-cv}$

- $w = 9 \leftrightarrow r_{9} = vc \text{ c v}$

- some cases:
  - if $c_1 = c_2$ then $vc \text{ c } v$
  - if $c_1 = n$ then $vc \text{ c } v$
  - if $c_1 = s$ then $v-c \text{ c } v$

• Examples (in blue substrings analysed with recursive calls)

- $pallone_{9} \rightarrow \text{ pal-} \text{lone}_{10} \rightarrow \text{ pal-lo-ne}_{10}$

- $asmatico_{9} \rightarrow \text{ a-} \text{smatico}_{10} \rightarrow \text{ a-sma-tico}_{10} \rightarrow \text{ a-sma-ti-co}_{10}$
Example 2 continued

- Complex version (under development):
  - alphabet $A' = A \cup V^n \cup C^2 \cup C^3 \ldots$
  - alphabet $A'$ contains “old” alphabet plus groups of vowels $(V^n)$ and of two $(C^2)$, three $(C^3)$ consonants and all the relevant subgroups
  - markers $M = \{-\}$
  - weights $W = \{w_i\}$, rules $R = \{r_i\}, r_i \leftrightarrow w_i$
  - $\beta = |A'|$

- Rules
  - rules are applied with a lookahead and a stepback with a recursive call
  - we have more rules but each rule is simpler since it contains a small set of sub-cases
“Segmentation”: computational complexity

- **Constant number of rules**

- **Input stream of n symbols. Two cases:**
  - $n$ fixed (not really relevant for complexity);
  - $n$ not known a priori.

- **Only stepforward**
  - $\star$ complexity $O(n)$

- **With lookahead and stepback**
  - $\star$ complexity $> O(n)$
  - $\star$ complexity $< O(n^2)$
  - $\star$ complexity $O(n \log n)$??
Cycles detection in directed graphs

Data structures
- G=(N,E) directed graph, |N|=n, |E|=m
- A alphabet A={a_i}_{i=1,...,n}
- S=Â={a_1....a_k | a_i ∈ A k ≥2}
- a_i ↔ n_i ∈ N, (a_i,a_k) ↔ ∈ (n_i,n_k) ∈ E

Three main steps
- mapping: from graphs to strings
- searching substrings with given properties (every substring identifies a cycle)
- pruning of “equivalent” substrings (duplicate cycles)
Cycles detection in directed graphs 2

Data structures

- $G=(N,E)$ directed graph, $|N|=n$, $|E|=m$
- $m=n(n-1)$ at the most
- Number of cycles (complete graph): $\binom{n}{2} + 2 \sum_{i=3}^{n} \binom{n}{i}$

First example

- Five cycles: ABBA, BCCB, ACCA, ABBCCA (anti clock wise), ACCBBA (clock wise)
- CBBAAC and ACCBBA equivalent through shift left
- BCCAAB and ABBCCA equivalent through shift left
Cycles detection in directed graphs 3

- Second example

- three cycles:
  - ABBA
  - BCCB,
  - ABBCCA

- casual mapping: ABCBCABCBA

- lexicographic mapping (by nodes order and by forward star): ABBABCCACB
Second example continued

- Cycles detection in ABBABCCACB (head node in blue, tail node in black, subcycles in italics)

- ABBABCCACB
  - ABBA 1 cycle, 1 scan
  - ABBCCB subcycle (premature closure), discard
  - ABBCCA 1 cycle, 2 scans

- ABBABCCACB
  - BAAB 1 duplicate cycle, 2 scans

- ABBABCCACB
  - BCCAAAB 1 duplicate cycle, 2 scans
  - BCCB 1 cycle, 1 scan

- ABBABCCACB
  - CAABBA subcycle, discard
  - CAABBC 1 duplicate cycle, 2 scans

- and so on....
Operations

**Mapping**
- $g \in G=(N,E) \rightarrow s \in S = \{s = aa^+ | a \in A\}$
- uniform vs. non uniform number of symbols for each node identifier
- modes of mapping: casual, lexicographic

**Searching**
- number of scans: maximum $n-1$ with $n=|N|$ 
- search operations 
- produces a list of substrings, one for each (duplicate) cycle

**Pruning (or removal of duplicate cycles)**
- ex-ante
- ex-post
Search Operations

* contiguity_check (cc)
  - finds couple of contiguous arcs (pairwise underlined), returns a boolean

* premature_closure (pc)
  - finds subcycles (i.e. cyclical substrings) to be discarded (velvet)
  - condition: head ≠ tail

* closure (cl)
  - finds a cycle as represented by a cyclic substring without pc
  - condition: head (blue) = tail (italic blue)

* Examples
  - ABB CCACB
  - ABB CCBCA
  - ABB CCA
Pruning 1

- removes duplicate cycles i.e. substrings of equal length and equivalent under left/right shift

- ex-ante
  - during the search phase
  - does not insert equivalent substrings in the list, usually one single growing vector C[]

- ex-post
  - after the search phase
  - two main cases:
    - one single unordered vector C[] of k elements
    - a set of vectors C_i[] i=2, ..., n, each of k_i elements, some even empty
  - all substrings contained in the list created at the end of the search phase are examined and equivalent substrings are removed
Pruning 2

**ex-ante**

- To each arc (of two distinct nodes) corresponds a string $s_1 \in S$ so that to a chain of arcs corresponds a chain of strings $s_1 s_2 s_3 \ldots s_k$

- $C_1[]$ is either empty (at the very beginning) or contains the substrings-cycles defined up to a certain point

- We have an iterated algorithm: at step $i$ ($i=1, \ldots, k$) we check if $s_i \in C_i[]$ and define

  $$C_{i+1}[] = \{ c_j \in C_i[] | c_j \cap s_i \neq \emptyset \}$$

- If the cycle contains $k$ arcs and the corresponding string is of $2k$ “symbols” and $C_k[] \neq \emptyset$ then the current substring corresponds to a duplicate cycle and must be discarded

- If, at any step, we have $C_i[] = \emptyset$ then the current string (if satisfies the aforesaid properties) corresponds to a new cycle and must be added to $C_1[]$
Pruning 3

’ex-post'

first case: one single unordered vector C[] of k elements

we have a cycle for i=1 to k-1, at each step i we have:

- confront C[i] with all the shifted versions of C[j] for j=i+1 to k
- every matching string must be removed from C[] whose number of elements reduces
- at the end C[] contains the residual substrings to each of which corresponds a cycle of the given graph

second case: a set of vectors C_i[] i=2, ..., n, each of k elements, some even empty

we repeat the algorithm we have designed for the single vector case for all the non empty vectors C_i[]
Applications of the algorithm

- cycles detection
- connectivity ?
- more?
Complexity of the algorithm

★ mapping (complete graph of n nodes): $O(n^2)\$
★ searching (complete graph of n nodes): $O(n^{n-1})$ ??
★ pruning:
  ★ ex-ante, “costly”
  ★ ex-post, “very costly”
Concluding remarks

* so many things to do and so a short time...
Game Over......

Thank you for your attention

Lorenzo Cioni
Dipartimento di Informatica
Largo Pontecorvo 3
Pisa
lcioni@di.unipi.it