Unlike linear programming problems, integer programming problems are very difficult to solve. In fact, no efficient general algorithm is known for their solution.

Given our inability to solve integer programming problems efficiently, it is natural to ask whether such problems are inherently “hard”. Complexity theory, offers some insight on this question. It provides us with a class of problems with the following property: if a polynomial time algorithm exists for any problem in this class, then all integer programming problems can be solved by a polynomial algorithm, but this is considered unlikely.

Algorithms for integer programming problems rely on two basic concepts:

- Relaxations
- Bounds

There are three main categories of algorithms for integer programming problems:

- **Exact** algorithms that guarantee to find an optimal solution, but may take an exponential number of iterations. They include cutting-planes, branch-and-bound, and dynamic programming.

- **Heuristic** algorithms that provide a suboptimal solution, but without a guarantee on its quality. Although the running time is not guaranteed to be polynomial, empirical evidence suggests that some of these algorithms find a good solution fast.

- **Approximation** algorithms that provide in polynomial time a suboptimal solution together with a bound on the degree of sub-optimality.
1 Optimality, Relaxation, and Bounds

- Given an IP \( \max \{ c^T x \mid x \in X \subset \mathbb{Z}_+^n \} \), how is it possible to prove that a given point \( x^* \) is optimal?

- Put differently, we are looking for some optimality conditions that will provide stopping criteria in an algorithm for IP.

- The “naive” but nonetheless important reply is that we need to find a lower bound \( \bar{z} \leq z \) and an upper bound \( \bar{z} \geq z \) such that \( \bar{z} = z = \bar{z} \).

- Practically, this means that any algorithm will find a decreasing sequence of upper bounds, and an increasing sequence of lower bounds, and stop when \( \bar{z} - \bar{z} \leq \epsilon \).

- **Primal bounds**: every feasible solution \( x^* \) provides a lower bound \( \bar{z} = x^* \leq z \).

- **Dual bounds**: The most important approach is by relaxation, the idea being to replace a “difficult” (max) IP by a simpler optimization problem whose optimal value is at least as large as \( z \).

- For the relaxed problem to have this property, there are two obvious possibilities:
  - **Enlarge the set** of feasible solutions so that one optimizes over a larger set, or
  - **Replace the (max) objective** function by a function that has the same or a larger value everywhere.

- **Def.:** A problem (RP) \( z^R = \max \{ f(x) \mid x \in T \subset \mathbb{R}^n \} \) is a relaxation of (IP) \( z = \max \{ c(x) \mid x \in X \subset \mathbb{R}^n \} \) if:
  1. \( X \subseteq T \), and
  2. \( f(x) \geq c(x) \forall x \in X \).

- If \( RP \) is a relaxation of \( IP \) \( \implies z^R \geq z \).

- \( z^R \) is an upper bound for \( z \).

- The question then arises of how to construct interesting relaxations:
  - **Linear Programming** relaxation: note that the definition of “better” formulation is intimately related to that of linear programming relaxations. In particular, better formulations give tighter (dual) bounds.
  - **Combinatorial** relaxation: the relaxation is an “easy” combinatorial problem that can be solved “rapidly”.
  - Relaxation by elimination: suppose we are given an integer program \( IP \) in the form \( z = \max \{ c(x) \mid Ax \leq b, \ x \in X \subset \mathbb{R}^n \} \). If the problem is too difficult to solve directly, one possibility is just to drop the constraints \( Ax \leq b \). Clearly, the resulting problem \( z = \max \{ c(x) \mid x \in X \subset \mathbb{R}^n \} \) is a relaxation of \( IP \).
- **Lagrangian relaxation**: An important extension of this idea is not just to drop complicating constraints, but then to add them into the objective function with Lagrange multipliers ("penalizing" the violation). Let \( z(u) = \max\{c(x) + u(b - Ax) \mid x \in X \} \). Then, \( z(u) \geq z \forall u \geq 0 \).

- **Surrogate relaxation**: A subset of constraints is substituted by a unique constraint corresponding to their linear combination using a multiplier vector \( u \geq 0 \): \( z(u) = \max\{c(x) \mid u^T(Ax) \leq u^Tb, x \in X \} \).

## 2 Exact methods

### 2.1 Cutting-Planes

Given an integer programming problem \( ILP = \max\{c^T x \mid Ax \leq b, x \in \mathbb{Z}^n \} \), and its continuous relaxation \( LP = \max\{c^T x \mid Ax \leq b, x \in \mathbb{R}^n \} \) a generic cutting plane algorithm looks as follows:

1. Solve the linear programming relaxation \( LP \). Let \( x^* \) be an optimal solution.
2. If \( x^* \) is integer stop; \( x^* \) is an optimal solution to \( ILP \).
3. If not, add a linear inequality constraint to \( LP \) that all integer solutions in \( ILP \) satisfy, but \( x^* \) does not; go to Step#1.

### 2.2 LPs with “too many” rows: constraint separation

Note that the above cutting-planes scheme can be generalized to solve LP problems with a “large” number of constraints (e.g., exponential). We call this generalization dynamic constraint generation or row generation:

1. Initialize reduced master problem \( \tilde{A}x \leq \tilde{b} \) with a “small” subset of constraints \( Ax \leq b \).
2. Solve LP associated to the reduced master problem \( RMP = \max\{c^T x \mid \tilde{A}x \leq \tilde{b}, x \in \mathbb{R}^n \} \) and obtain \( x^* \).
3. If \( x^* \) satisfies all constraints in \( Ax \leq b \) stop; otherwise add a (some) violated constraints to \( \tilde{A}x \leq \tilde{b} \) and goto #2.

The scheme is very simple: constraints are added only if needed. The crucial part is step #3, aka separation problem, where we want to find a violated constraint (or prove that none exists). If the number of constraints is large, we cannot enumerate all the constraints explicitly and check them one by one. So we need to formulate the separation problem as an LP, more likely as an ILP. It may sound strange to solve an ILP in an algorithm to solve an LP, yet this scheme is quite efficient in practice! Note that according to the scheme
above, the optimal value $c^T x^*$ of any reduced master is always greater than or equal to the optimal value of the original LP problem, so even if we stop the procedure before the stopping condition is met, we still have a valid upper bound available.

### 2.3 LPs with “too many” columns: column generation

What about models where the number of variables is too “large” to deal with explicitly (e.g., exponential)? Again, the idea is simple: we consider only a subset of the variables. The process is called column generation or variable pricing:

1. Initialize reduced master problem $\tilde{A}x \leq b$ with a “small” subset of variables.

2. Solve LP associated to the reduced master problem $\text{RMP} = \max \{ \tilde{c}^T x \mid \tilde{A}x \leq b, x \in \mathbb{R}^n \}$ and obtain primal solution $x^*$ (padding with 0 the values of the variables that do not appear in the restricted master). Let $y^*$ be the corresponding dual solution.

3. If $x^*$ is optimal for the original LP, i.e. $y^*$ is dual feasible, stop; otherwise add a (some) columns with positive reduced cost, i.e., columns corresponding to violated dual constraints, and goto #2.

Note that in the dynamic row generation algorithm we had to test feasibility of the solution $x^*$ of the RMP. Here, in the dynamic column generation algorithm, we need to test optimality of the solution $x^*$ of the RMP, i.e., we need to test if there is any column with positive reduced cost. Testing optimality is equivalent to testing dual feasibility, so given a dual solution $y^*$ of RMP, the quantity $c_j - \sum_{i=1}^{m} a_{ij}y^*_i$ is the reduced cost of variable $x_j$.

Again, the crucial part is step #3, aka pricing problem, where we want to find a violated dual constraint (or prove that none exists). If the number of dual constraints is large, we cannot enumerate all the dual constraints explicitly and check them one by one. So we need to formulate the pricing problem as an LP, more likely as an ILP. And again, it may sound strange to solve an ILP in an algorithm to solve an LP, yet this scheme is quite efficient in practice!

Note that during the column generation process, the value $\tilde{c}x^*$ is always smaller than the optimal cost of the original LP, so we do not know whether such value is larger or smaller than the optimal cost of the original ILP. Hence, if we stop the procedure before the stopping condition is met, we do not have a valid upper bound.

### 2.4 Branch-and-Bound

Branch-and-bound uses a “divide and conquer” approach to explore the set of feasible solutions. However, instead of exploring the entire feasible set, it uses bounds on the optimal cost to avoid exploring certain parts of the set of feasible integer solutions. Let $X$ be the set of feasible integer solutions to the problem $\max \{ c^T x \mid x \in X \}$. We partition the set $X$ into a finite collection of subsets $X_1, \ldots, X_k$ and solve separately each one of the subproblems: $\max \{ c^T x \mid x \in X_i \} \ i = 1 \ldots k$. We then compare the optimal solutions to the
subproblems, and choose the best one. Each subproblem may be almost as difficult as the
original problem and this suggests trying to solve each subproblem by means of the same
method; that is by splitting it into further subproblems, etc. This is the branching part
of the method and leads to a tree of subproblems. We also assume that there is a fairly
efficient algorithm, which for every $X_i$ of interest, computes an upper bound $\bar{z}(X_i)$ to the
optimal cost of the corresponding subproblem. In the course of the algorithm, we will also
occasionally solve certain subproblems to optimality, or simply evaluate the cost of certain
feasible solutions. This allows us to maintain a lower bound $L$ on the optimal cost, which
could be the cost of the best solution found thus far. The essence of the algorithm lies in the
following observation. If the upper bound $\bar{z}(X_i)$ corresponding to a particular subproblem $X_i$
satisfies $\bar{z}(X_i) \leq L$, then this subproblem need not be considered further, since the optimal
solution to the subproblem is no better than that the best feasible solution encountered thus
far. A generic branch-and-bound algorithm looks as follows:

1. Select an active subproblem $X_i$.
2. If the subproblem is infeasible, delete it; otherwise, compute
   upper bound $\bar{z}(X_i)$ for the corresponding subproblem.
3. If $\bar{z}(X_i) \leq L$ delete the subproblem.
4. If $\bar{z}(X_i) \geq L$, either obtain an optimal solution to the subproblem,
   or break the corresponding subproblem into further subproblems,
   which are added to the list of active subproblems.

Note that there are several “free parameters” in this algorithm:

1. There are several ways to choose the subproblem, i.e., a rule for which subproblem to
   work on next, e.g., breadth-first search, depth-first search, best-first search.
2. There are several ways to obtain an upper bound, e.g., continuous relaxation.
3. There are several ways of breaking a problem into subproblems, i.e., a rule for deciding
   how to branch (aka branching rule).

In particular, an LP-based Branch-and-Bound scheme for a (max) ILP problem, looks as
follows:

1. Init subproblems list with ILP and set $L = -\infty$.
2. If list is empty: stop; otherwise select subproblem $P$ from the list.
3. Solve LP relaxation of $P$ and find optimal solution $x^*$.
4. If infeasible goto #2.
5. If $x^*$ integer, set $L = \max\{L, c^T x^*\}$ and goto #2.
6. If $c^T x^* \leq L$ goto #2.

7. Select variable $x_j$ such that $x_j^*$ is fractional, generate two subproblems one with constraint $x_j \leq \lfloor x_j \rfloor$ and the other with constraint $x_j \geq \lceil x_j \rceil$, add the two subproblems to the list, and goto step #2.

2.4.1 Cut-and-Branch

In the 1970s and 1980s, people experimented with the following approach:

1. Solve an initial LP relaxation (by primal simplex).

2. Let $x^*$ be the solution. If $x^*$ is integer and feasible, output the optimal solution and stop.

3. Search for strong (possibly facet-defining) inequalities violated by $x^*$. If none are found go to Step#6.

4. Add the inequalities to the LP as cutting planes.

5. Resolve the LP (by dual simplex) and go to Step#2.

6. Run branch-and-bound (keeping the cutting planes in the ILP formulation).

Cut-and-branch has one disadvantage: if our ILP formulation has an exponential number of constraints (as in the TSP) then cut-and-branch might lead to an infeasible solution! This is because the final solution may be integer, but may violate a constraint that was not generated in the cutting-plane phase.

In the 1980s, Grötschel, Padberg and coauthors used a modified version of cut-and-branch to solve quite large TSPs (up to 300 cities or so):

1. Run the cutting-plane algorithm.

2. If the solution represents a tour, stop.


4. If the integer solution represents a tour, stop.

5. Find one or more sub-tour elimination constraints that are violated by the integer solution. Add them to the ILP formulation and return to Step#3.
2.4.2 Branch-and-Cut

The idea was: why not search for cutting planes at every node of the branch-and-bound tree? The term *branch-and-cut* was coined by Padberg & Rinaldi (1987, 1991), they used it to solve large TSP instances (up to 2000 cities or so). The main ingredients of branch-and-cut are:

- Linear programming solver (capable of both primal and dual simplex)
- Branch-and-bound shell which enables cutting planes to be *added at any node* of the tree.
- Subroutines that search for strong cutting planes.

In other words, branch-and-cut utilizes cuts when solving the subproblems. In particular, we augment the formulation of the subproblems with additional cuts, in order to improve the bounds obtained from the linear programming relaxations.

2.4.3 Branch-and-Price

Imagine that we have established that the LP relaxation of our (max) ILP can be solved using column generation:

- If we then want to solve the original ILP to proven optimality, we have to start branching!
- So, we *price at each* node of the branch-and-bound tree, just in case more columns of positive reduced cost can be found.
- The name *branch-and-price* comes from Savelsbergh (1997).

2.5 Dynamic Programming

In the previous (sub)section, we introduced branch and bound, which is an exact, intelligent enumerative technique that attempts to avoid enumerating a large portion of the feasible integer solutions. In this section, we introduce another exact technique, called *dynamic programming* that solves integer programming problems sequentially.

We start recalling a well-known algorithm for the shortest path problem in a di-graph with no negative cycles: *Bellman-Ford algorithm*.

This is a *label correcting* algorithm.

- At iteration $k$, label $\pi_j$ represents the cost of a shortest path from $s$ to $j$ with *at most* $k$ arcs.
- The correctness is based on the the so called Bellman-Ford *optimality conditions.*
Algorithm 1: Bellman-Ford algorithm, time complexity $\mathcal{O}(|V||A|)$

**Input:** A di-graph $G = (V, A)$ with arc costs $c$, no negative cycles, and a source vertex $s \in V$.

**Output:** Shortest path from $s$ to $v \in V$ or reports negative-weight cycle.

1. $\pi(s) := 0$; $p(s) := /$
2. foreach $v \in V \setminus \{s\}$ do
   3. $\pi(v) := \infty$; $p(v) := /$
3. end
4. for $k \leftarrow 1$ to $|V| - 1$ do
5.   $\pi' = \pi$
6.   foreach $(i, j) \in A$ do
7.     if $(\pi'(j) > \pi'(i) + c_{ij})$ then
8.       $\pi(j) = \pi(i) + c_{ij}$
9.       $p(j) = i$
10. end
11. end
12. end
13. end
14. foreach $(i, j) \in A$ do
15.   if $(\pi(j) > \pi(i) + c_{ij})$ then
16.     report that a negative-weight cycle exists
17. end
18. end

- Optimality Conditions: given a graph $G = (V, A)$ with costs $c_{ij} \forall (ij) \in A$ and source $s$, the labels $\pi_v \forall v \in V$ represent the costs of shortest paths from $s$ to $v$ if and only if $\pi_v$ are the lengths of feasible paths from $s$ to $v$ and $\pi_j - \pi_i \leq c_{ij} \forall (ij) \in A$.

- The algorithm was designed as an application of dynamic programming, in fact the optimality conditions can also be proved in terms of dynamic programming optimality principle for shortest paths.

Dynamic programming is based on Bellman’s **Optimality Principle**: “Regardless of the decisions taken to enter a particular state in a particular stage, the remaining decisions made for leaving that stage must constitute an optimal policy.” **Consequence:** If we have entered the final state of an optimal policy, we can trace it back.

Guidelines for constructing dynamic programming algorithms:
1. View the choice of a feasible solution as a sequence of decisions occurring in stages, and so that the total cost is the sum of the costs of individual decisions.

2. Define the state as a summary of all relevant past decisions.

3. Determine which state transitions are possible. Let the cost of each state transition be the cost of the corresponding decision.

4. Define the objective function and write a recursion on the optimal cost from the origin state to a destination state.

3 Heuristics

A heuristic algorithm is any algorithm that finds a feasible solution. Typically, heuristics are fast (polynomial) algorithms. For some problems, even feasibility is \(NP\)-complete, in such cases a heuristic algorithm generally cannot guarantee to find a feasible solution. Heuristic algorithms can be classified in the following way:

- **Constructive**
  - start from an “empty” solution
  - iteratively add new elements to the current “partial” solution, until a “complete” solution is found
  - *greedy, optimization based, implicit enumeration based*

- **Local Search**
  - start from an initial feasible solution
  - iteratively try to improve it by “slightly” modifying it
  - stop when no improving adjustments are possible (*local optimum*)

- **Meta-heuristics**
  - an “evolution” of local search algorithms
  - avoid local optima using “special” techniques
  - try to avoid “cycling” in the execution of the algorithm

Typically, the quality of the solutions found improves as we move from constructive, to local search, and finally to meta-heuristics.
3.1 Constructive algorithms

Constructive algorithms construct a feasible solution starting only from the input data. For some problems where feasibility is \( \mathcal{NP} \)-complete, they try to find a solution “as feasible as possible”.

3.1.1 Greedy

The idea is to start from an empty solution and iteratively construct a solution using an “expansion criterion” that allows one to take a “convenient” choice at each iteration subject to the problem constraints. Often times, the expansion criterion is based on sorting the elements according to a score that represents the “quality” of the choice. This way, we try to reward at each iteration the “most promising choice. The performance of the algorithm, generally improves if the scores are updated dynamically as the algorithm proceeds. Note that any choice taken, cannot be changed!

Here are some examples.

**Greedy for KP-01:**

1. \( S := \emptyset, \bar{c} := c \)
2. If \( w_j > \bar{c} \forall j \notin S \to \text{STOP} \)
3. Among \( j \notin S \) with \( w_j \leq \bar{c} \), determine \( j^* \) having max score \( k(j) := \frac{p_j}{w_j} \), set \( S := S \cup \{ j^* \}, \bar{c} := \bar{c} - w_j^* \to \text{GOTO 2.} \)

**Greedy for SCP:**

1. \( S := \emptyset, \bar{M} := \{1 \ldots m\} \)
2. If \( \bar{M} = \emptyset \to \text{STOP} \)
3. Among \( j \notin S \) determine \( j^* \) such having min score \( k(j) := \frac{c_j}{\sum_{i=1}^{m} a_{ij}} \), set \( S := S \cup \{ j^* \}, \bar{M} := \bar{M} \setminus \{ i : a_{ij^*} = 1 \} \to \text{GOTO 2.} \)

3.1.2 Optimization based

These are iterative heuristics where at each iteration the choice is based on the solution of an optimization problem that is easier than the original one. The most common ones are heuristics based on relaxations, where the scores correspond to the information obtained from the solution of the relaxation (e.g., optimal solution, reduced costs, Lagrangian costs etc.)

The following scheme has proved to be successful in practice for many classes of ILP problems:

1. Let \( P \) be the ILP problem to be solved.
2. Solve the LP relaxation of \( P \) and obtain solution \( \bar{x} \).

3. If \( \bar{x} \) is integer stop.

4. **Fix some variables** \( x_j \) to an integer value \( a_j \) **suggested** by \( \bar{x} \) and add the corresponding constraint \( x_j = a_j \) to \( P \); goto step #2.

Clearly, depending on the **fixing rule** at step #4 we obtain different algorithms and solutions.

Here are some examples.

*ContHeur for KP-01:*

1. \( S := \emptyset, \bar{c} := c, R := \{1...n\} \)

2. If \( R = \emptyset \) \( \rightarrow \) STOP

3. Solve \( C(KP01) \) for \( j \in R \) with capacity \( \bar{c} \), let \( \bar{x} \) denote the corresponding optimal solution.

4. Among \( j \in R \) determine \( j^* \) having max score \( \bar{x}_j \),
   set \( S := S \cup \{j^*\}, \bar{c} := \bar{c} - w_{j^*}, R := R \setminus (\{j^*\} \cup \{j : w_j > \bar{c}\}) \rightarrow \) GOTO 2.

*LagrCostHeur for KP-01:*

1. \( S := \emptyset, \bar{c} := c \)

2. If \( w_j > \bar{c} \forall j \notin S \) \( \rightarrow \) STOP

3. Among \( j \notin S \) with \( w_j \leq \bar{c} \), determine \( j^* \) having max **lagrangian profit** \( \tilde{p}_j := p_j - \lambda w_j \),
   set \( S := S \cup \{j^*\}, \bar{c} := \bar{c} - w_{j^*} \rightarrow \) GOTO 2.

*ContHeur for SCP:*

1. \( S := \emptyset, \bar{M} := \{1...m\} \)

2. If \( \bar{M} = \emptyset \) \( \rightarrow \) STOP

3. Solve \( C(SCP) \) with additional constraint \( x_j = 1 \forall j \in S \), let \( \bar{x} \) denote the corresponding optimal solution.

4. Among \( j \notin S \) determine \( j^* \) having max score \( \text{score} \ \bar{x}_j \),
   set \( S := S \cup \{j^*\}, \bar{M} := \bar{M} \setminus \{i : a_{ij^*} = 1\} \rightarrow \) GOTO 2.

*LagrCostHeur for SCP:*

11
1. $S := \emptyset, \bar{M} := \{1 \ldots m\}$

2. If $\bar{M} = \emptyset$ → STOP

3. Among $j \notin S$ determine $j^*$ having min lagrangian cost $\tilde{c}_j = c_j - \sum_{i=1}^{m} a_{ij} \lambda_i$,
   set $S := S \cup \{j^*\}, \bar{M} := \bar{M} \setminus \{i : a_{ij^*} = 1\}$ → GOTO 2.

**LagrSolHeur for SCP:**

1. $S := \emptyset, \bar{M} := \{1 \ldots m\}$

2. For $j = 1 \ldots n$ do
   
   • $\tilde{c}_j = c_j - \sum_{i=1}^{m} a_{ij} \lambda_i$
   • if $\tilde{c}_j \leq 0$ set $S := S \cup \{j\}, \bar{M} := \bar{M} \setminus \{i : a_{ij} = 1\}$

3. If $\bar{M} = \emptyset$ → STOP

4. Foreach $i \in \bar{M}$ do:
   
   • $\bar{c} = \min \{\tilde{c}_j : a_{ij} = 1\}$
   • $u_i = u_i + \bar{c}$
   • Foreach $j \notin S$ such that $a_{ij} = 1$
     
     - $\tilde{c}_j := \tilde{c}_j - \bar{c}$
     - if $\tilde{c}_j = 0$ set $S := S \cup \{j\}, \bar{M} := \bar{M} \setminus \{l : a_{lj} = 1\}$

3.1.3 Implicit enumeration based

These heuristics are based on the partial execution of an implicit enumeration algorithm (branch-and-bound). The most common version consists in stopping the branch-and-bound algorithm after a certain time-limit. Another possibility is to limit the number of subproblems generated at each node, keeping only the most promising ones.

3.2 Local Search

These algorithms start from an initial solution ("current" solution) and try to improve it with "small adjustments". The algorithm terminates when no adjustment can improve the current solution (local optimum). They define a neighborhood of a feasible solution $x$ as a function $N : x \rightarrow N(x)$ where $N(x)$ is a subset of the feasible set. In other words, the neighborhood of $N(x)$ is defined by the set of available modifications that can be applied to $x$ maintaining feasibility.

The basic scheme for a local search algorithm (maximization problem) looks as follows:

1. if $(\exists x' \in N(x) : f(x') > f(x))$ then
2. \( x := x' \)

3. if \( (f(x) = UB) \) stop \((x \text{ is optimal})\)

4. goto #2

5. else

6. stop \((\text{local optimum})\)

7. endif

Here are some examples.

\underline{Local Search for KP-01:}

1. Init solution \( S \)

2. Consider the following \textit{moves}:
   \begin{itemize}
   
   \item \( S \cup \{j\}, j \notin S \) (insertion of one item)
   
   \item \( S \cup \{j\} \setminus \{i\}, j \notin S, i \in S \) (exchange between two items)
   
   \end{itemize}

3. Let \( R \) be the “best” among these moves

4. if \( \sum_{j \in R} p_j \leq \sum_{j \in S} p_j \) \to STOP \((\text{local optimum})\)

5. else \( S := R \) \to GOTO2.

\underline{Local Search for SCP:}

1. Init solution \( S \)

2. Consider the following \textit{moves}:
   \begin{itemize}
   
   \item \( S \setminus \{j\}, j \in S \) (removal of one column)
   
   \item \( S \setminus \{j\} \cup \{i\}, j \in S, i \notin S \) (exchange of two columns)
   
   \end{itemize}

3. Let \( R \) be the “best” among these moves

4. if \( \sum_{j \in R} c_j \geq \sum_{j \in S} c_j \) \to STOP \((\text{local optimum})\)

5. else \( S := R \) \to GOTO2.