Methods and Models for Combinatorial Optimization

Heuristis for Combinatorial Optimization

L. De Giovanni

1 Introduction

Solution methods for Combinatorial Optimization Problems (COPs) fall into two classes: exact methods and heuristic methods. Exact methods are able, at least in theory, to provide an optimal solution, that is, a feasible solution that optimize (minimize or maximize) the value of the objective function, whereas heuristic methods provide a feasible solution with no optimality guarantee.

In some cases, we may be able to find "efficient" exact algorithms to solve a COP: for example, the problem of finding the shortest paths on a graph, under some reasonable assumptions often met in practise, can be solved by the Dijkstra or Bellman-Ford algorithms, able to provide optimal solutions in running times according to a polynomial function (of small degree). For more complex problems, when no "efficient" algorithms are available, a possible approach may be formulating the COP as a Mixed Linear Programming (MILP) model and solving it by a MILP solver (e.g. Cplex, Gurobi, Xpress etc.), which makes use of general purpose exact algorithms that guarantee, at least in theory, to find the optimal solution. These methods have an exponential computational complexity, so that the time to solve the problem may grow exponentially with its size.

It is not always possible or appropriate to apply exact solution methods, due to basically two concurrent issues: the inner complexity of a COP (e.g. an NP-Hard problem), and the time available to provide a solution, which may be limited. To this respect it is important to clarify that the use of a heuristic method instead of an exact one must be well motivated: the inner complexity of a problem does not justify in itself the use of heuristics, since literature may provide viable exact algorithms. The use of heuristic is thus motivated by the inner complexity of the COP together with consideration on the opportunity of implementing exact methods (which may require considerable implementation resources), the available computational time, the size of the instances to be solved etc. For example, it is always advisable to make an attempt to formulate a model of the COP (e.g. a MILP model): this effort is useful in the analysis phase, but also as an operational tool, since the growing efficiency of solvers may make the implementation of the model a viable approach to obtain an exact solution in reasonable running times. Nevertheless, the

inner complexity of the problem may make an hard task to obtain a accurate enough formulation. Think about the problem of configuring a transportation network in order to minimize the congestion level, which depends also on the behaviour of the network users: even if several mathematical behavioural models are available in literature, often the only way for obtaining realistic results is simulation, which can be hardly embedded in a mathematical programming model (in general, an in a MILP model in particular). In other contexts, even if we have an accurate formulation, it may be the available time that inhibits the use of exact methods, since their computational complexity does not give any guarantee on the required running time: solving a problem in order of hours may be appropriate in some cases, non acceptable in others.

The problem features and/or the solution context may make inappropriate the application of exact methods, while it is necessary to provides "good" feasible solutions in "reasonable" amount of time. It is worth nothing that, while in some cases the availability of a provably optimal solution is necessary, in many other cases, including perhaps the majority of real cases, a good approximate solution is enough, in particular for large size instances of a COP. For example:

- for many parameters (data) determining a COP coming from a real application, just estimates are available, which may be also subject to *error*, and it may be not worth waiting a long time for a solution whose value (or even feasibility) cannot be ensured;
- a COP aims at providing one possible solution to a real problem aiming at a *quick* scenario evaluation (e.g., operational contexts, integration of optimization algorithms into interactive Decision Support Systems);
- a COP may be stated in a *real time* system, so that it is required that a "good" feasible solution is provided within a limited amount of time (e.g. fractions of seconds).

These examples attest for the extended use, in practise and in real applications, of methods that provide "good" solutions and guarantee acceptable computing times, even if they cannot guarantee optimality: they are called *heuristic methods* (from greek *eurìskein = to find*).

For many COPs it is possible to devise some *specific heuristic* that exploits some special feature of the COP itself and the human experience of who solves the problem in practice. In fact, very often, an "optimization" algorithm is just coding, when available, the rules applied to "manually" solve the problem. In this case, the quality of the obtained solution, that is, the effectiveness of the algorithm, depends on the rules themselves and, hence, on the amount of "good practice" that the algorithm embeds: if the amount is high, we will obtain 'fairly "good" solutions (hardly better than the current ones, anyway); if the amount is little (or lacking, as it may happens if the developer has computer skills but no knowledge about the problem to be solved), the method is likely to be "the first reasonable algorithm come to our mind".

In the last decades, academic and practitioners' interest has been devoted to *general heuristic approaches*, able to outperform specific heuristics on the field. Related literature is vaste and has been fed by the ingenuity (and often the fantasy – see the work by S oresen cited below) of researchers. So many techniques have been proposed that it is a very hard task to attempt a systematic and broadly accepted classification. A possible classification is the following:

- **constructive heuristics**: they can be applied when the solution is given by selecting the "best" subset of a given set of elements. One starts from the empty set and iteratively add one element to the solution, by applying some specific selection criterion. For example, if the selection criterion is some "local optimality" (e.g., the element that best improves the objective function), we obtain the so called *greedy* heuristics. The basic feature of such construction approaches is the fact that, in principle, the selection made at a certain step influences only the following steps, that is, no backtracking is applied;

- **meta-heuristics**: they are general, multi-purpose methods or, better, algorithmic schemes which are devised independently from a specific COP. They defines some components and their interactions, allowing them to provide an hopefully good solutions. In order to devise a real algorithm for a specific COP, we need to devise and specialize each component. Among the well known met-heuristics we cite: *Local Search*, *Simulated Annealing*, *Tabu Search*, *Variable Neighborhood Search*, *Greedy Randomized Adaptive Search Techniques*, *Stochastic Local Search*, *Genetic Algorithms*, *Scatter Search*, *Ant Colony Optimization*, *Swarm Optimization*, *Neural Networks* etc.

- **approximation algorithms**: they are a special class of heuristis methods abel to provide a *performance guarantee*, that is, it is possible to formally prove that, for any instance of the COP, the obtained solution will never be worst than the optimal solution (which may be unknown) over a specified threshold (which is, often, rather large): for example, we will obtain a solution which is at most 30% far from the optimum.

- **iper-heuristics**: we are here at the boundary between Operations Research and Artificial Intelligence, and the research is at its early stages. The aim is defining algorithm general algorithms able to automatically build good algorithms for a specific problem (for example, by trial and errors, they try to put together the right instructions, evaluating each attempt by applying each trial algorithm).

- **etc. etc. etc.**

what we are going to say in the following is necessarily very schematic and mostly based on examples. For an insight into this topic, a possible (among many others) reading is

> *C. Blum and A. Roli, "Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison", ACM Computer Surveys 35:3, 2003 (p. 268-308)*

The article provides also useful links to specific techniques. We also suggest reading the following paper, which criticizes the trend, registered in the last-years literature, to propose an increasing number of "new" meta-heuristic methodologies, based on natural metaphors where just the metaphor changes, whereas the relying mechanisms are essentially the same:

> *K. Sorensen, "Metaheuristics – the metaphor exposed", International Transactions in Operational Research (22), 2015 (p. 3-18)*

2 Constructive heuristics

Constructive heuristics provide a solution by building it based only on input data and using a scheme that does not consider, or strongly limits, backtracking: they start from an empty solution and, iteratively, at each step, new elements are added to the solution according to a predefined *expansion criterion*, until a complete solution is defined.

Among the many possible constructive heuristics, we will here consider *greedy* (myopic) algorithms, algorithms embedding exact techniques for simplified problems, and algorithms that simplify potentially exact methods. Notice that all of these techniques are devised such that the overall final computational complexity is polynomial.

2.1 Greedy algorithms

These algorithms adopt a local expansion criterion, that is, the choice is the one which seems to be best choice in that moment, also taking into account the constraints of the problem: at each iteration, the element to add to the current solution is the one that provides the best improvement to the objective function. The concept scheme of a greedy algorithm is the following:

1. Initialize solution S;
2. For each choice to be made:
3. make the most convenient choice (taking feasibility constraints into account, e.g., by excluding elements that make the solution unfeasible).

The scheme is very basic, nevertheless, greedy algorithms are widespread used. In fact:

- the algorithm often "simulates" the most intuitive rules applied in practice to build solutions (it is actually among the "first methods coming into mind");
- the implementation is very simple;
- running times are normally very small (if steps 2. and 3. can be efficiently implemented to identify the current choice and to evalute the choise itself), since just one evaluation per step is required (linear number of evaluations);

- greedy algorithms are often embedded in more sophisticated algorithms using them to solve sub-problems (e.g., a greedy algorithm may provide the initial solution for local search algorithms, or feasible solutions to speed-up exact techniques).

Greedy algorithms may consider elements sorted according to a *Dispatching Rule*, that is, elements are considered according to this sorting and inserted into the solution. Normally, sorting criteria are based on associating a numeric "score" to each element, which identifies the most promising elements. The score can be evaluated once before starting the algorithm, based only on input data (pre-sorting); or, which often eds up with better performance, the score can be dynamically updated, taking into account previous choices. The disadvantage of dynamic scoring is an increased computational effort (and related running time), since the evaluation is required before each choice, instead of once.

Of course, the final solution depends on the selection criterion and, with the aim of obtaining different and possibly better solutions, the basic scheme can be slightly modified, by iterating the same greedy algorithm with a different sorting (or selection criterion). A straightforward and general way to obtain different sorting, is to *randomize* the dispatching rule. For example, at each repetition, the score may be biased by a random component, so that, at each step, the selection will me made on the element which is not the best, but close to the best: the algorithm would thus be less myopic, since some locally optimal element has the chance to be preserved for future steps, when choices may becomes more critical. A similar randomization can be obtained by making, at each step, a random choice among the best n elements still available.

Normally, greedy algorithms are used to build a feasible solution from the empty set, in which case we say that they are a *primal heuristic*. They may be also used as *dual heuristic*, meaning that they start from an unfeasible solutions and the score is related to the reduction of the unfeasibility degree: at each step, the choice that decreases as much as possible the unfeasibility degree (trying to take a good value of the objective function) is made.

2.2 Algorithms embedding exact solution methods

The expansion criteria, that is choosing *the best* element to add, can be interpreted as an optimization (sub)problem, which is easier than the original COP. For example, the subproblem (or one approximation) may be modelled as a MILP and, if the MILP is still difficult to sole, the linear relaxation may be solved by standard solvers. In this case, the information provided by the solution may be used to define the scores. As already seen, the sub-problem may be solved once before starting the heuristic procedure, or before each iteration, taking into account already fixed choices.

Normally, the time needed to run these algorithms is longer than the one required by greedy procedures, but the provided solutions are generally better. In fact, using an optimization model involves all (still-to-be-fixed) decision variables , so that the choices made at each step take to some extent into account a global optimality criterion.

2.3 Simplifying exact procedures

Some heuristic approaches simplifies exact procedures in order to make their complexity polynomial in time, at the cost of losing optimality guarantee. For example, an exact algorithm may be based on implicit enumeration (e.g., branch-and-bound) that gives rise to an exponential number of alternatives to be evaluated: a heuristic approach is to use some greedy criteria to select only a subset of alternatives. A simpler alternative is to stop an enumeration scheme, visited in depth, after a given number of alternatives have been explored, or after a fixed time-limit, and take the best solution generated so far. In practise, we may implement a MILP model using a standard solver (e.g. Cplex) and run it for a fixed amount of time, getting the best incumbent solution (if any).

A more sophisticated variant to an enumeration scheme is the *beam search*. It considers a *partial* breath-first visit of an enumeration tree: at each node, all the b alternatives (child nodes) are generated and, at each level of the tree, only k nodes are branched, k being a parameter to be calibrated according to the available computational time. The k nodes are chosen by associating to each node an evaluation that should be related to the likelihood a node is on the way to an optimal solution: a rapid evaluation (e,g, by greedy completing the partial solution at the node) of a possible solution in the subtree rooted at the node itself, a bound on the value of the optimal solution in the subtree, a weighted sum of these two values or any other evaluation. Then the k most promising nodes are chosen at each level, whereas the others are discarded: in such a way, we do not have the combinatorial explosion of the size of the enumeration tree, since at each level (at most) k nodes are taken and the enumeration trees reduces to a *beam* of $n \cdot k$ nodes (n being the number of levels of the enumeration tree). Hence, the overall complexity is polynomial, if polynomial is the node evaluation procedure. More in details, denoting by n the size of the problem in terms of variables to be fixed (number of tree's levels), by b the number of alternatives for each decision variable, that is, the number of children of each node, and by k the beam size, we will evaluate $O(n \cdot k \cdot b)$ nodes. The bottom level is made of k leaves, corresponding to k alternative solutions among which the best is chosen. Notice that, starting from the time needed to evaluate a single node, one advantage of this technique is the possibility of a good estimation of the time needed to run it, once k is fixed; or, it is possible to estimate the parameter k based on the maximum available time.

In its basic implementation, *beam search* does not backtrack (it is not possible to go back to previous levels once the k nodes have been selected): this is the reason why we have included beam search among the constructive heuristics. In fact, it may be classified as a meta-heuristic, since it defines a framework with some components (tree-search structure, node evaluation) to be specified in order to get an algorithm for a specific COP. This shows that the boundary between classes of heuristics is floating (and, in fact, not relevant in practice). Moreover, a variant to beam search is the *recovery beam search*, where backtracking is allowed: if at a given level one of the k nodes seems to be not as promising as supposed, it is possible to prune it and go back to a previous level and follow a new path (according to how backtracking is implemented, polynomial complexity may

be lost).

2.4 Example: the 0-1 Knapsack problem (KP/0-1)

We are given n objects and a knapsack of capacity W, and let w_j and p_j be the weight and the profit of object j. We want to select a subset of the objects such that the profit is maximized and the capacity of the knapsack is not exceeded. In this problem, a "good" object is one having high profit and small weight. A possible greedy selection order is thus the one given by ascending ratio between $p - J$ and w_j, yielding the following greedy heuristic.

- **Greedy algorithm KP/0-1**

  1. Sort object according to ascending $\frac{p_j}{w_j}$.
  2. Initialize: $S := \emptyset$ (selected objects),
     $\bar{W} := W$ (remaining knapsack capacity),
     $z := 0$ (current solution value).
  3. **for** $j = 1, \ldots, n$ **do**
  4. **if** $(w_j \leq \bar{W})$ **then**
  5. $S := S \cup \{j\}, \;\; \bar{W} := \bar{W} - w_j, \;\; z := z + p_j.$
  6. **endif**
  7. **endfor**

  Notice that the expansion criterion is here static and evaluated once.

- **Beam search for KP/0-1**

  We illustrate by the following example.

$$\begin{aligned} \max \quad & 30x_1 + 36x_2 + 15x_3 + 11x_4 + 5x_5 + 3x_6 \\ & 9x_1 + 12x_2 + 6x_3 + 5x_4 + 3x_5 + 2x_6 \leq 17 \\ & x_i \in \{0, 1\} \quad \forall\, i = 1 \ldots 6 \end{aligned}$$

  The result is shown in Figure 1 and considers the following implementation:

  - binary branching ($b = 2$): at tree level i, we fix either to 0 or 1 variable x_i, according to decreasing $\frac{p_j}{w_j}$ (in the examples it corresponds to $1 \ldots 6$). The number of levels is equal to the number of variables (6 in this case).
  - $k = 2$;

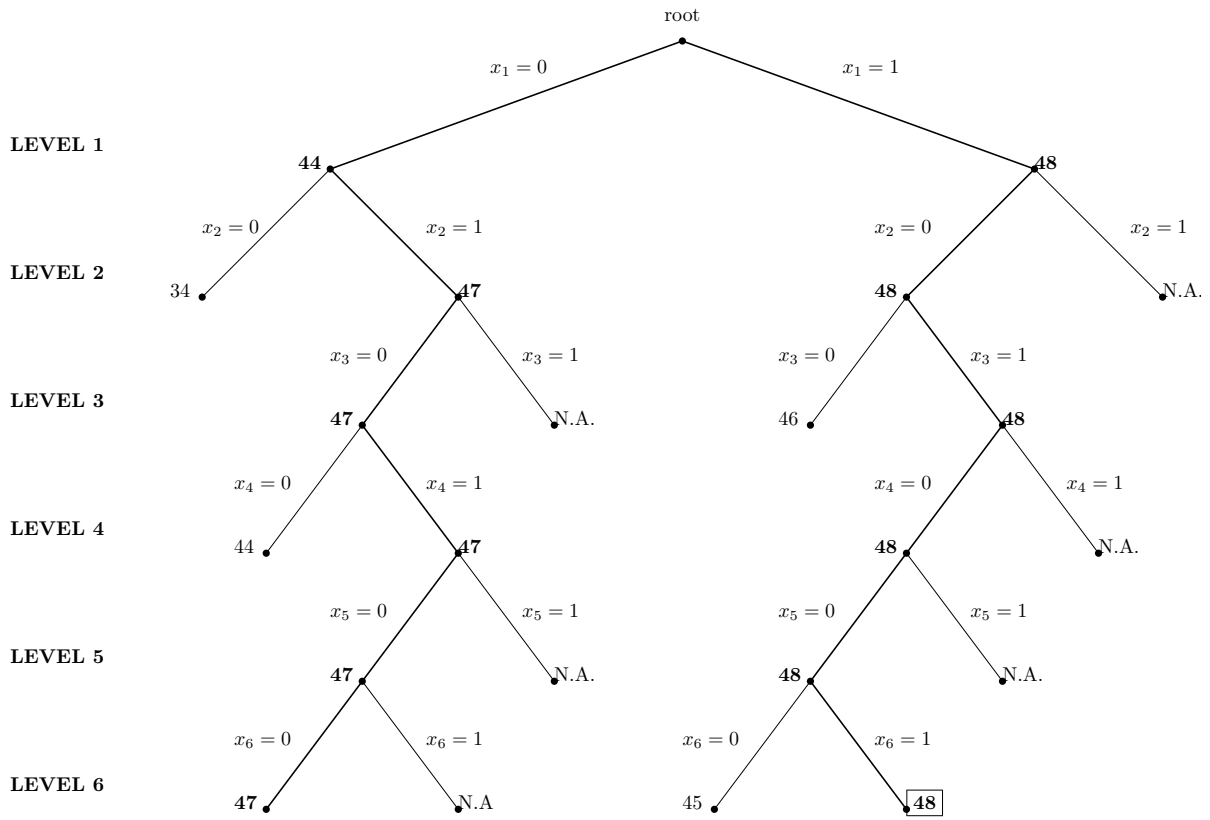

Figure 1: An example of search tree associated to a beam-search method for KP/0-1.

– the heuristic algorithm described above is used to provide the evaluation of each node ($O(n)$ complexity, once the variable are initially fixed, at the root node, once for all), taking into account the value fixed at previous levels. We thus evaluate at each node a lower bound (feasible solution) and choose, at each level, the $k = 2$ nodes with higher lower bound.

2.5 Example: the Set Covering problem

The *set covering* problem (SCP) takes a set M and a set $\mathcal{M} \subseteq 2^M$ of subsets of M as input. For each $j \in \mathcal{M}$ a cost c_j is given and we want to select a combination of subsets in $\mathcal{M}$ whose union is M (the subsets *cover* M) and whose total cost in minimized. For each $i \in M$, $j \in \mathcal{M}$, let a_{ij} be the input parameter equal to 1 if $i \in j$, 0 otherwise. A subset is "good" if its cost is small and it if covers many elements (among the ones to be still covered). The following greedy algorithm associates to each subset a score computed as a function of the subset cost and the number of new elements covered by adding the set itself to the solution.

- **Greedy algorithm for SCP**

  1. Initialize: $S := \emptyset$ (selected subsets),
     $\bar{M} := \emptyset$ (elements already covered by the currently selected subsets),
     $z := 0$ (objective function value).
  2. if $\bar{M} = M$ ($\Leftrightarrow$ all elements are covered), STOP;
  3. compute the set $j \notin S$ minimizing the ratio $\dfrac{c_j}{\sum_{i \in M \setminus \bar{M}} a_{ij}}$;
  4. set $S := S \cup \{j\}$, $\bar{M} := \bar{M} \cup \{i : a_{ij} = 1\}$, $z := z + c_j$ and go to 2.

  Notice that, in this case, scores are dynamically evaluated, since it is related not only to the subset at hand, but also to previous choices determining the current partial solution, which determines the number of *additional* covered elements.

- **Algorithm for SCP based on linear programming relaxation**

  A possible model for SCP is the following:

$$\begin{aligned} \min & \sum_{j \in \mathcal{M}} c_j x_j & & \\ s.t. & \sum_{j \in \mathcal{M}} a_{ij} x_j & \geq 1 & \quad \forall\, i \in M \\ & x_j & \in \{0,1\} & \quad \forall\, j \in \mathcal{M} \end{aligned}$$

  The model is hard to be solved to integrality by a MILP solver, whereas its linear relaxation, that is, the problem with constraints $x_j \in [0,1]$ instead of $x_j \in \{0,1\}$ can be efficiently solved. We can embed the exact solution of the linear relaxation in a constructive heuristic for SCP as follows.

  1. Initialize: $S := \emptyset$ (selected subsets),
     $\bar{M} := \emptyset$ (elements already covered by the currently selected subsets),
     $z := 0$ (objective function valule).
  2. se $\bar{M} = M$ ($\Leftrightarrow$ tall elements are covered), STOP;
  3. solve the linear programming relaxation of SCP (with additional constraints $x_j = 1$ ($j \in S$), and let x^* be the corresponding optimal solution;
  4. let $j = \arg\max_{j \notin S} x^*_j$;
  5. set $S := S \cup \{j\}$, $\bar{M} := \bar{M} \cup \{i : a_{ij} = 1\}$, $z := z + c_j$ and go to 2.

3 Neighbourhood Search: local search

Let P be an optimization problem defined by an objective function f and a feasible region X. A *neighborhood* is an application

$$N : s \to N(s)$$

that associates, to each point $s \in X$ a subset $N(s) \subseteq X$.

The basic idea of the meta-heuristic known as *neighborhood search* is the following: start from an initial solution (*current solution*) x and try to improve it by exploring a suitable neighbourhood of x. If the neighbourhood contains a solution y better than x, then iterate the process considering y as the new current solution.

The simplest version of the neighbourhood search is the *local search* (LS): the algorithm stops when no the neighbourhood of the current solution contains no improving solutions, so that the current solution is a *local optimum*.

The basic LS scheme is the following (for a minimization problem) :

1. Determine an initial solution x;
2. **while** $(\exists\ x' \in N(x) : f(x') < f(x))$ **do {**
3. $\quad x := x'$
4. **}**
5. **return**(x) (x is a local optimum)

We remark that, in general, LS guarantees that the final solution is a *local optimum*, that is, a solution with no improving neighbour.

The scheme is extremely simple and general. To obtain an algorithm for a specific problem the following components should be specialized:

- a method to find an initial solution;
- a solution representation, which is the base for the following elements;
- the application that, starting from a solution, generates the neighbourhood (moves);
- the function that evaluates solutions;
- a neighbourhood exploration strategy.

As we will see, these elements are the basis for any neighbourhood search, and not only for LS.

3.1 Initial solution

A straightforward way to obtain the initial solution is to generate it at random, even if it is advisable to start from a good solution, for example the one obtained by a heuristic algorithm, or the one implementing the current pratice, or the one currently adopted on the field (if available). In fact, no theoretical results nor experience attest that better solution are obtained starting from better initial points: normally, LS is applied starting from different initial solutions (*multistart*), obtained by heuristics or randomly generated, or by randomized heuristics. In such a way, different portions of the feasible region are explored and different local optima are identified (and the best is chosen).

3.2 Solution representation

The solution representation encodes the features of the solutions as to provide the "concrete" support for the operations that allow us exploring the solution space. As we will see, different solution representations may be adopted for the same problem, which influences the design of the remaining LS elements.

> Example. The solutions of the KP/0-1 problem (which, we recall, is to select, among n items with weights w_i and profit p_i, the ones to be loaded in a knapsack with capacity W) can be represented as follows:
>
> 1. list of the loaded items;
> 2. characteristic vector of the set of the loaded items, i.e., a binary vector of dimension n;
> 3. ordered sequence of items.

It is not always possible to derive directly the solution from its representation: for example, the last option for KP/0-1 does not identify directly the subset of loaded items. In these cases, a *decoding* procedure is necessary to transform the representation into a solution.

> In the KP/0-1 example, decoding the first two representations is immediate. The third representation can be decoded by loading items in the given order and loading them if the residual space is enough, until all items are considered (this is in fact the same heuristic procedure described before, with an initial sorting different from the greedy one).

3.3 Neighbourhood definition (moves)

Given a solution x, its *neighbours solutions* are obtained by applying some *moves* that perturb x (called the *centre* of the neighbourhood).

> For KP-0/1, possible moves are: (i) insert an item in the knapsack, (ii) swap an item in the knapsack with one outside, etc.

3.3.1 Neighbourhood size

Normally, moves implies small changes to the centre solution, to "take control" of the path in the solution space. Hence, the neighbourhood size, that is, the number of neighbour solutions, is relatively small and a rapid evaluation is possible. The LS designer should determine a good trade-off between the neighbourhood size (hence time needed to explore it) and its effectiveness, that is the ability to contain improving solutions. Notice that, in theory, we may define more and more complex moves, up to make all the feasible solutions be included in a neighbourhood: in this case we are sure that an improving move exists (unless we are already at the optimum), but exploring the neighbourhood may be impractical, since an exponential-time enumeration is needed. This simple example shows us that, each time we design a neighbourhood, we should consider the following first question:

1) *what is the size of the neighbourhood (number of neighbour solutions generated by the moves)?*

   following the example, (i) the item-insertion neighbourhood and (ii) the item pairwise swap neighbourhood have, respectively, size $O(n)$ e $O(n^2)$.

Some authors use the term *strength* of the neighbourhood: a strong neighbourhood is able to lead a LS towards good solutions, independently from the starting point. This is related to *landascape analysis*, which is outside our scope: we let here simply notice the LS finds local optima, and the definition of a local optimum depends also on the neighbourhood definition. In other words, the "combinatorial convexity" of the solution space depends, at a first glance, from the objective function and from the neighbourhood definition, that is, a solution x is not a local optimum in itself, but because the defined neighbourhood of x does not contain any solution with better objective function value.

> In KP/0-1, a stronger neighbourhood may be obtained by including moves implying swapping two items outside with two items inside, at the cost of a larger size $O(n^4)$.

The availability of a strong neighbourhood reduces, at least in principle, the need for a good initial solution, so that it should be not worth spending many time in evaluating it, but it may be advisable to randomly generate different initial solutions, as to obtain a good sampling of the feasible region leading to better local optima (if not the global optimum).

Another desirable feature for a neighbourhood is *connection*: given two feasible solutions x and y, it is always possible to build a sequence of moves from x to y. Non connected neighbourhoods limit a-priori the possibility of exploring the solution space.

> The insertion neighbourhood (i) is not connected (it is only possible to reach only solutions containing all the initial items). The swapping neighbourhood is not connected too (the number of items in the solution cannot change). A connected neighbourhood may be the one including two types of moves: 1. inserting an item, and 2. removing an item. Notice that, a straightforward implementation of LS would never select removing moves, which do not improve the objective function.

3.3.2 Solution representation is important!

The way the neighbourhood is devised and designed strongly depends on the way solutions are represented.

> All the moves we have previously described for KP-0/1 comes from the first representation (insert or remove are list operations). The same moves can be easily adapted to the second representation (characteristic vector): flipping a 0 to 1 (insertion), flipping a 1 to 0 (removing), swapping a 0 and a 1 (pairwise swap).
>
> The third representation (ordered item list) yield different move definitions, since a neighbour solution is given by a different order. A move may be swapping the position of two items in the sequence: for example, if $n = 7$ and the centre solution is $1-2-3-4-5-6-7$, neighbour solutions are $1-6-3-4-5-2-7$ (swap 2 and 6) or $5-2-3-4-1-6-7$ (swap 1 and 5). The size of this neighbourhood is $O(n^2)$, and it is connected (with respect to maximal solutions, that is, the ones where no further items can be included preserving feasibility).

3.3.3 Neighbourhood evaluation and overall complexity

It is important to take the *efficiency* of a neighbourhood into account, that is, the overall time taken to explore it, that is, to evaluate all of its solutions. One of the reasons why LS is a successful heuristic is its capability of quickly evaluating many solutions. The time to explore a neighbourhood depends on its size and on the computational complexity of the procedure that evaluates one single neighbour. To this end, it is crucial devising an *incremental evaluation*, that is, a procedure that provides the evaluation of a neighbour exploiting the information od the centre solution. In fact, a second question arises:

2) *What is the computational complexity of the (possibly incremental) neighbour solution evaluation?*

Notice that moved yielding small perturbations make more likely an incremental evaluation (the neighbour is "similar" to the centre): this is the reason why simple moves (yielding small weaker neighbourhoods) are preferred to more complex moves (yielding to stronger neighbourhood that are larger and normally less efficient).

Overall, the *neighbourhood complexity* is the product of its size and the computational complexity of evaluating a single neighbour solution.

> In the KP/0-1 example, evaluating each neighbourhood coming from insertion or removing moves takes constant time. In fact, an incremental evaluation is possible: starting from the information (value) of the centre, we just need to add or subtract the profit of the selected item. The overall complexity of these neighbourhoods is thus $O(1 \cdot n) = O(n)$.
>
> The swapping moves, as defined fro the first and the second representations, the incremental evaluation time remains constant (add/subtract the profits of two items), determining an $O(1 \cdot n^2) = O(n^2)$ overall complexity.

Taking the third representation into account, the related position-swapping move can be hardly evaluated in a (fully) incremental way, and the evaluation is $O(n)$, hence the overall neighbourhood complexity is $O(n^3)$.

3.4 Solution evaluation function

The solution *evaluation function* is the value associated to each solution and used to compare solutions to each others: the best neighbour (chosen as the next move) is the one having the best value of the evaluation function. Normally, a solution is evaluated by computing the value of the objective function. However, different evaluations are possible. For example, when problems are strongly constrained and a feasible solution is not available, a first step is to find a feasible solution, which can be done by starting from an unfeasible solution and applying a local search having the objective of minimizing a measure of the infeasibility. In fact, a neighbourhood may often contain unfeasible solutions. If they are excluded, the connection features may be lost, so that one may chose to allow choosing them. Of course we need to evaluate and compare both feasible and unfeasible solutions, we thus use an evaluation function different from the objective function. For example, we combine the objective function with a penalty term related to unsatisfied constraints.

In the KP-0/1 problem, we have seen that the neighbourhood including both insertion and removing moves is connected but only in theory, since removing moves are never improving according to the objective function. We thus may use an evaluation combining the objective function and a penalty measuring the unfeasibility. Representing a solution with X, the subset of items in the knapsack, the evaluating function becomes:

$$\tilde{f}(X) = \alpha \sum_{i \in X} p_i - \beta \max \left\{ 0, \sum_{i \in X} w_i - W \right\} \qquad (\alpha, \beta > 0).$$

When the knapsack capacity is saturated with inserting moves, the LS will be able to continue if, by properly calibrating the parameters α and β, we are able to temporarily overload the knapsack and then, after the capacity violation is sufficiently large, to reach a better feasible solution by removing moves.

3.5 Exploration strategies

The basic LS scheme depicted above make the search go one if the neighbourhood of the current solution contains an *improving* solution. The choice of which improving neighbour solution to select is not unique and depends on th e*exploration strategy*. The common alternatives are:

- *first improvement*: as soon as the first improving sbeighbour is generated, it is selected as the next current solution. Notice that the final results (e.g. the local minimum found, or the running times for a single move) depend on the order in

which neighbour solutions are explored. In order to reduce running times, we may adopt a heuristic order, to give priorities to the moves that are more likely to yield an improvement. A random order may be used instead, so that different repetition of the local search (starting from the same initial solution) may lead to different local optima.

- *steepest descent* or *best improvement*: all the neighbourhood is explored and evaluated, and the next move is determined by the best one.

Several variants are possible, e.g., choosing ***at random*** among the best k neighbours (to introduce randomness into a quasi-best improvement scheme), or ***storing*** a number of the "second-best" neighbours (discarded by a steepest descent strategy), and using them as further starting points after the first LS has been run.

TO BE CONTINUED ... I