Chapter 4
Constructive Methods

Having ascertained that the problem to be solved is intractable and that the design of a heuristic is justified, the next step is to imagine how to construct a solution. This step is directly related to the problem modeling.

4.1 Systematic Enumeration

When we have to discover the best possible solution for a combinatorial optimization problem, the first idea that comes is to try to build all the solutions to the problem, evaluate their feasibility and quality, and return the best that satisfies all constraints. Clearly, this approach can solely be applied to problems of moderate size. Let us examine the example of a small knapsack instance in 0-1 variables with two constraints:

$$\begin{aligned} \max r &= 9x_1 + 5x_2 + 7x_3 + 3x_4 + x_5 \\ \text{Subject } & 4x_1 + 3x_2 + 5x_3 + 2x_4 + x_5 \leqslant 10 \\ \text{to : } & 4x_1 + 2x_2 + 3x_3 + 2x_4 + x_5 \leqslant 7 \\ & x_i \in \{0, 1\} (i = 1, \ldots, 5) \end{aligned} \tag{4.1}$$

To list all the solutions of this instance, an enumeration tree is constructed. The first node separates the solutions for which $x_1 = 0$ of those where $x_1 = 1$. The second level consists of the nodes separating $x_2 = 0$ and $x_2 = 1$, etc. Potentially, this problem has $2^5 = 32$ solutions, many of which are unfeasible, because of constraint violations. Formally, the first node generates two sub-problems that will be solved recursively. The first sub-problem is obtained by setting $x_1 = 0$ in (4.1):

© The Author(s) 2023

É. D. Taillard, *Design of Heuristic Algorithms for Hard Optimization*, Graduate Texts in Operations Research, https://doi.org/10.1007/978-3-031-13714-3_4

$$\begin{aligned} \max r &= 0 + 5x_2 + 7x_3 + 3x_4 + x_5 \\ \text{Subject} \quad & 3x_2 + 5x_3 + 2x_4 + x_5 \leqslant 10 \\ \text{to :} \quad & 2x_2 + 3x_3 + 2x_4 + x_5 \leqslant 7 \\ & x_i \in \{0, 1\} (i = 2, \dots, 5) \end{aligned}$$

The second sub-problem is obtained by setting $x_1 = 1$ in (4.1):

$$\begin{aligned} \max r &= 9 + 5x_2 + 7x_3 + 3x_4 + x_5 \\ \text{Subject} \quad & 3x_2 + 5x_3 + 2x_4 + x_5 \leqslant 6 \\ \text{to :} \quad & 2x_2 + 3x_3 + 2x_4 + x_5 \leqslant 3 \\ & x_i \in \{0, 1\} (i = 2, \dots, 5) \end{aligned}$$

To avoid enumerating too many solutions, the tree can be pruned by noting that all branches arising from a node with a constraint violation will lead to unfeasible solutions. Indeed, for this problem instance, all constraint coefficients are non-negative. For instance, if the x_1, x_2, x_3 variables are already fixed to 1, both constraints are violated, and all the sub-problems that could be created from there will produce unfeasible solutions. Therefore, it is useless to develop this branch by trying to set values of the x_4 and x_5 variables.

Another way to prune the non-promising branches is to estimate by a short computation whether a sub-problem could lead to a better solution than the best found so far. This is the *branch and bound* method.

4.1.1 Branch and Bound

To quickly estimate whether a sub-problem may have a solution, and if the latter is promising, a technique is to *relax* one or more constraints. The optimal solution of the relaxed problem is not necessarily feasible for the initial one. However, few interesting properties can be deduced by solving the relaxed problem: If the latter has no solutions or its optimal solution is worse than the best feasible solution already found, then there is no need to develop the branch from this sub-problem. If the relaxed sub-problem contains an optimal solution feasible for the initial problem, then developing the branch is also unnecessary. In addition to the Lagrangian relaxation seen above (Sect. 3.1.1), several relaxation techniques are commonly used to simplify a sub-problem.

Variable integrality Imposing integer variables makes Problem (4.1) difficult. We can therefore remove this constraint and solve the problem:

$$\begin{aligned} \max S &= 9x_1 + 5x_2 + 7x_3 + 3x_4 + x_5 \\ \text{Subject } & 4x_1 + 3x_2 + 5x_3 + 2x_4 + x_5 \leqslant 10 \\ \text{to: } & 4x_1 + 2x_2 + 3x_3 + 2x_4 + x_5 \leqslant 7 \\ & 0 \leqslant x_i \leqslant 1 (i = 1, \ldots, 5) \end{aligned} \tag{4.2}$$

This linear problem can be solved efficiently in polynomial time. Its optimal solution is (0.5; 1; 1; 0; 0) with objective value of 16.5. Since it comprises a fractional value, this solution is not feasible for the initial problem. However, it informs us that there is no solution to Problem (4.1) whose value exceeds 16.5 (or even 16 since all the coefficients are integers). Therefore, if an oracle gives us the feasible solution (1; 0; 1; 0; 0) of value 16, we can deduce this solution to be optimal for the initial problem.

Constraint aggregation (surrogate constraint) A number of constraints are linearly combined to get another one. In our simple example, we get:

$$\begin{aligned} \max S &= 9x_1 + 5x_2 + 7x_3 + 3x_4 + x_5 \\ \text{Subject } & 8x_1 + 5x_2 + 8x_3 + 4x_4 + 2x_5 \leqslant 17 \\ \text{to: } & x_i \in \{0, 1\} (i = 1, \ldots, 5) \end{aligned} \tag{4.3}$$

This problem is a standard knapsack. It is easier to solve than the initial problem. The solution (1; 1; 0; 1; 0) is optimal for the relaxed Problem (4.3) but is not feasible for the initial problem because the second constraint is violated. As the relaxed problem is NP-hard, this approach may be problematic.

Combined relaxation Clearly, several types of relaxation can be combined, for instance, the aggregation of constraints and the integrality variables. For our small example, we get:

$$\begin{aligned} \max S &= 9x_1 + 5x_2 + 7x_3 + 3x_4 + x_5 \\ \text{Subject } & 8x_1 + 5x_2 + 8x_3 + 4x_4 + 2x_5 \leqslant 17 \\ \text{to: } & 0 \leqslant x_i \leqslant 1 (i = 1, \ldots, 5) \end{aligned} \tag{4.4}$$

This problem can be solved in $O(n \log n)$ as follows: the variables are sorted in the order of decreasing r_i/v_i values, where r_i represents the revenue of the object i and v_i its aggregated volume. In our example, the indices are already sorted. The objects are selected one after the other in this order until a new object would overcharge the knapsack. This leads to $x_1 = x_2 = 1$. The next object is split to completely fill the knapsack ($\Longrightarrow \; x_3 = 4/8$ for a total value of the knapsack $S = 9 + 5 + 7 \cdot 4/8 = 17,5$). Since all the coefficients are integers in our example, $S = \lfloor 17,5 \rfloor = 17$ is also a valid bound for the optimal value of the initial problem.

Algorithm 4.1 provides the general framework of the branch and bound method. Figure 4.1 shows a partial enumeration tree that can be obtained by solving the small problem instance (4.1). Three components should be specified by the user for implementing a complete algorithm.

Algorithm 4.1: Branch and bound framework for an objective to maximize. It is necessary to provide three methods: α for the management of the sub-problems to be solved (generally, a priority queue (based on a heuristic criterion) or a stack), a method β for evaluating the relaxation of the sub-problems, and a heuristic γ for choosing the next variable to separate for generating new sub-problems

```
Input: A problem with n variables x_1,...,x_n, policy α for managing sub-problems,
       relaxation method β, branching method γ
Result: An optimal solution x* of value f*
1  f* ← −∞                                               // Value of best solution found
2  F ← ∅                                                 // Set of fixed variables
3  L ← {x_1,...,x_n}                                     // Set of free variables
4  Q ← {(F,L)}                                           // Set of sub-problems to solve
5  while Q ≠ ∅ do
6      Remove a problem P = (F,L) from Q according to policy α
7      if P can potentially have feasible solutions with values already fixed in F then
8          Compute a relaxation x of P with method β, modifying only variables x_k ∈ L
9          if x is feasible for the initial problem and f* < f(x) then Store the improved
            solution
10             x* ← x
11             f* ← f(x)
12         else if f(x) > f* then Expand the branch
13             Choose x_k ∈ L according to policy γ
14             forall possible value v of x_k do
16                 Q ← Q ∪ {(F ∪ {x_k = v}, L \ {x_k})}
17         else No solution better than x* can be obtain
18             Prune the branch
```

First, the management policy of the set Q of sub-problems awaiting treatment must be specified. If Q is managed as a queue, we have a breadth-first search. If Q is carried as a stack, we have a depth-first search. The last promotes a rapid discovery of a feasible solution to the initial problem.

A frequent choice is to manage Q as a priority queue. This implies computing an evaluation for each sub-problem. Ideally, this evaluation should be strongly correlated with the best feasible solution that can be obtained by developing the branch. A typical example is to use the value S of the relaxed problem. The choice of a management method for Q is frequently based on very empirical considerations.

The second component to be defined by the user is the relaxation technique. This is undoubtedly one of the most delicate points for designing an efficient branch and bound. This point strongly depends on both the problem to be solved and the numerical data.

The third choice left is how to separate the problem into sub-problems. A simple policy is to choose the smallest index variable or a non-integer variable in the

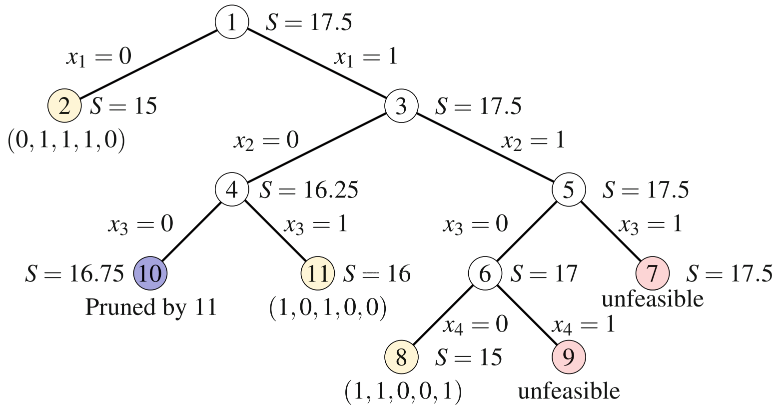

Fig. 4.1 Solving Problem (4.1) with a branch and bound. Sub-problem set Q managed as a stack. The nodes are numbered by creation order. Branching is done by increased variable index. Nodes 9 and 7 are pruned because they cannot lead to feasible solutions. Node 10 is pruned because it cannot lead to a solution better than node 11

solution of the relaxed problem. Frequently, the policy adopted for branching is empirical.

A simple implementation of this framework is the $A*$ *search* algorithm. The last manages Q as a priority queue and evaluates a heuristic value before inserting a sub-problem in Q.

In some cases, the number of possible values for the next x_k variable to set is significant, especially when x_k can take any integer value. A branching technique is to consider the fractional value y taken by a variable x_k and to develop two branches, one with the additional constraint $x_k \leqslant \lfloor y \rfloor$ and the other with $x_k \geqslant \lfloor y \rfloor + 1$. In this case, the sets of fixed and independent variables are unchanged on Line 16. This technique was proposed by Dakin [2].

Another technique, known as *branch and cut* is to add constraints to the relaxed sub-problem. The goal of the new constraints is to remove the unfeasible solution obtained by solving the sub-problem. For instance, such a constraint may prevent a variable to take a given fractional value.

4.1.1.1 Example of Implementation of a Branch and Bound

A naive branch and bound implementation manages the sub-problem set as a stack (policy α). This is performed automatically with a recursive procedure.

For the TSP, a solution is a permutation p of the n cities. The element p_i provides the ith city visited. Assume that the order of cities has been fixed up to and including i and L is the set of cities remaining to be ordered. A lower bound on the optimal tour length can be obtained by considering that:

1. The ith city is connected to the closest of L.

2. Each city of L is connected to another of L that is the closest.
3. The first city is connected to the closest of L.

Doing so, a valid tour is eventually obtained for the complete problem. When only one "free" city remains ($|L| = 1$), we have to go to this one and then to return to the departure city. In this situation, a valid tour is obtained. The procedure given by Code 4.1 returns a flag indicating whether a feasible tour is found.

Code 4.1 tsp_lower_bound.py Code for computing a naive lower bound to the optimal tour. The procedure returns the bound and can alter the order of the last cities of the tour. In the event the length of the modified tour is equal to the value of the lower bound, the procedure indicates that the tour is optimal

```
######### Computation of a naive lower bound for the TSP
def tsp_lower_bound(d,                                        # Distance matrix
                    depth,                        # tour[0] to tour[depth] fixed
                    tour):                                             # TSP tour

    n = len(tour)
    lb = 0 #Compute the length of the path for the cities already fixed in tour
    for j in range(depth):
      lb += d[tour[j]][tour[j+1]]

    valid = 1  # valid is set to 1 if every closest successor of j build a tour
    for j in range(depth, n-1):     # Add the length to the closest free city j
        minimum = d[tour[j]][tour[j+1]]
        for k in range(n-1, depth, -1):
            if k != j and minimum > d[tour[j]][tour[k]]:
                minimum = d[tour[j]][tour[k]]
                if (k > j):
                    tour[k], tour[j+1] = tour[j+1], tour[k]
                else:
                    valid = 0
        lb += minimum

    minimum = d[tour[n-1]][tour[0]]        # Come back to first city of the tour
    for j in range (depth+1, n-1):
        if (minimum > d[tour[j]][tour[0]]):
            valid = 0
            minimum = d[tour[j]][tour[0]]
    lb += minimum
    return lb, tour, valid      # Lower bound, tour modified, lb == tour length
```

To implicitly list all possible tours on n cities, an array as well as a `depth` index can be used. From the `depth` index, all the possible permutations of the last elements of the array are enumerated. This procedure is called recursively with `depth + 1` after trying all the remaining possibilities for the `depth` array entry. To prune the enumeration, no recursive call is performed either if the lower bound computation provides an optimal tour or if the lower bound of the tour length is larger than that of a feasible tour already found. Code 4.2 implements an implicit enumeration for the TSP.

Code 4.2 tsp_branch_and_bound.py Code for implicitly enumerating all the permutations of n elements

```
from tsp_lower_bound import tsp_lower_bound                          # Listing 4.1

######### Basic Branch & Bound for the TSP
def tsp_branch_and_bound(d,                                        # Distance matrix
                         depth,  # current_tour[0] to current_tour[depth] fixed
                         current_tour,                    # Solution partially fixed
                         upper_bound):                        # Optimum tour length

    n = len(current_tour)
    best_tour = current_tour[:]
    for i in range(depth, n):
        tour = current_tour[:]
        tour[depth], tour[i] = tour[i], tour[depth]
        lb, tour, valid = tsp_lower_bound(d, depth, tour)
        if (upper_bound > lb):
            if (valid):
                upper_bound = lb
                best_tour = tour[:]
                print("Improved: ", upper_bound, best_tour)
            else:
              best_tour, upper_bound = tsp_branch_and_bound(d, depth+1, tour, \
                                                             upper_bound)
    return best_tour, upper_bound
```

It should be noted here that this naive approach requires a few seconds to a few minutes to solve problems up to 20 cities. However, this represents a significant improvement over an exhaustive search, which would require a computing time of several millennia. The relaxation based on the notion of 1-tree presented in Sect. 3.1.1.2 could advantageously replace that provided by Code 4.1.

In recent years, so-called exact methods for solving integer linear programs have made substantial progresses. The key improvements are due to more and more sophisticated heuristics for computing relaxations and branching policies. Software like *CPLEX* or *Gurobi* include methods based on metaheuristics for computing bounds or obtaining good solutions. This allows a faster pruning of the enumeration tree. Despite this, the computational time grows exponentially with the problem size.

4.2 Random Construction

A rapid and straightforward method to obtain a solution is to generate it randomly among the set of all feasible solutions. We clearly cannot hope to reliably find an excellent solution like this. However, this method is widely implemented in iterative local searches repeating a constructive phase followed by an improvement phase. It should be noted here that the modeling of the problem plays a significant role, as

noted in Chap. 3. In case finding a feasible solution is difficult, one must wonder whether the problem modeling is adequate.

Note that it is not necessarily easy to write a procedure generating each solution of a feasible set with the same probability. Exercise 4.1 deals with the generation of a random permutation of n items. Naive approaches such as those given by Algorithms 4.5 and 4.6 can lead to non-uniform solutions and/or inefficient codes.

4.3 Greedy Construction

In Chap. 2, the first classical algorithms of graphs passed in review—Prim and Kruskal for building the minimum spanning tree and Dijkstra for finding the shortest path—were greedy algorithms. They are building a solution by including an element at every step. The element is permanently added on the base of a function evaluating its relevance for the partial solution under construction.

Assuming a solution is composed of elements $e \in E$ that can be added to a partial solution s, the greedy algorithm decides which element to add by computing an incremental cost function $c(e, s)$. Algorithm 4.2 provides the framework of a greedy constructive method.

Algorithm 4.2: Framework of a greedy constructive method. Strictly speaking, this is not an algorithm since different implementation options are possible, according to the definition of the set E of the elements constituting the solutions and the incremental cost function

Input: A trivial partial solution s (generally $\varnothing$); set E of elements constituting a solution; incremental cost function $c(s,e)$

Result: Complete solution s

```
1 R ← E                                        // Elements that can be added to s
2 while R ≠ ∅ do
3 |   ∀e ∈ R, compute c(s,e)
4 |   Choose e′ optimizing c(s,e′)
5 |   s ← s ∪ e′                               // Include e′ in the partial solution s
6 |_  Remove from R the elements that cannot be added any more to s
```

Algorithms with significantly different performances can be obtained according to the definition of E and $c(s, e)$. Considering the example of the Steiner tree, one could consider E as the set of edges of the problem and the incremental cost function as the weight of each edge. In this case, a partial solution is a forest.

Another modeling could consider E as the Steiner points. The incremental cost function would be to calculate a minimum spanning tree containing all terminal nodes plus e and those already introduced in s.

We now provide some examples of greedy heuristics that have been proposed for a few combinatorial optimization problems.

4.3.1 *Greedy Heuristics for the TSP*

Countless greedy constructive methods have been proposed for the TSP. Here is a choice illustrating the variety of definitions that can be made for the incremental cost function.

4.3.1.1 Greedy on the Edges

The most elementary way to design a greedy algorithm for the TSP is to consider the elements e to add to a partial solution s are the edges. The incremental cost function is merely the edge weight. Initially, we start from a partial solution $s = \varnothing$. The set R consists of the edges that can be added to the solution, without creating a vertex of degree > 2 or a cycle not including all the cities. Figure 4.2 illustrates how this heuristic works on a small instance.

4.3.1.2 Nearest Neighbor

One of the easiest greedy methods to program for the TSP is the nearest neighbor. The elements to insert are the cities rather than the edges. A partial solution s is, therefore, a path in which the cities are visited in the order of their insertion. The incremental cost is the weight of the edge that connects the next city. Figure 4.3 illustrates the execution of this heuristic on the same instance as above. It is a coincidence to get a solution identical to the previous method.

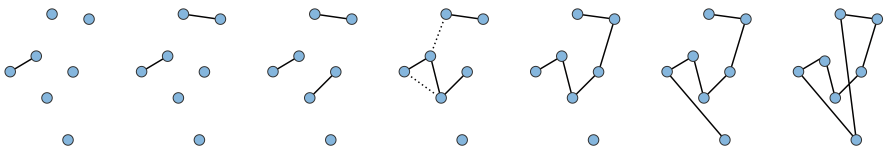

Fig. 4.2 Steps of a greedy constructive method based on the edge weight for the TSP

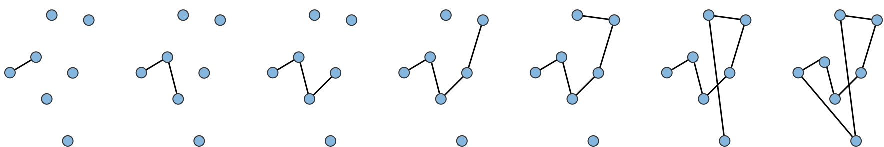

Fig. 4.3 Running the nearest neighbor for a tiny TSP instance

The nearest neighbor greedy heuristic can be programmed very concisely, in $\Theta(n^2)$, where n is the number of cities (see Code 4.3).

Code 4.3 tsp_nearest_neighbor.py Nearest neighbor for the TSP. Note the similarities with the implementation of the Dijkstra algorithm given by Code 2.1

```
######### Nearest Neighbor greedy heuristic for the TSP
def tsp_nearest_neighbor(d,                                        # Distance matrix
                         tour):                            # List of cities to order

    n = len(tour)
    length = 0                                                          # Tour length
    for i in range(1, n):          # Cities from tour[0] to tour[i-1] are fixed
        nearest = i                                  # Next nearest city to insert
        cost_ins = d[tour[i-1]][tour[i]]                     # City insertion cost
        for j in range(i+1, n):                         # Find next city to insert
            if d[tour[i-1]][tour[j]] < cost_ins:
                cost_ins = d[tour[i-1]][tour[j]]
                nearest = j
        length += cost_ins
        tour[i], tour[nearest] = tour[nearest], tour[i]  # Definitive insertion

    length += d[tour[n - 1]][tour[0]]                            # Come back to start

    return tour, length
```

4.3.1.3 Largest Regret

A defect of the nearest neighbor is to temporarily forget a few cities, which subsequently causes significant detours. This is exemplified in Fig. 4.3. To try to prevent this kind of situation, we can evaluate the increased cost for not visiting city e just after the last city i of the partial path s. In any case, the city e must appear in the final tour. This will cost at least $\min_{j,k \in R} d_{je} + d_{ek}$. Conversely, if e is visited just after i, the cost is at least $\min_{r \in R} d_{ie} + d_{er}$. The largest regret greedy constructive method chooses the city e maximizing $c(s, e) = \min_{j,k \in R} (d_{je} + d_{ek}) - \min_{r \in R} (d_{ie} + d_{er})$.

4.3.1.4 Cheapest Insertion

The cheapest insertion heuristic involves inserting a city in a partial tour. The set E consists of cities, and the trivial initial tour is a cycle on both cities which are the nearest. The incremental cost $c(s, e)$ of a city is the minimum detour that must be consented to insert the city e in the partial tour s between two successive cities of s. Figure 4.4 illustrates this greedy method.

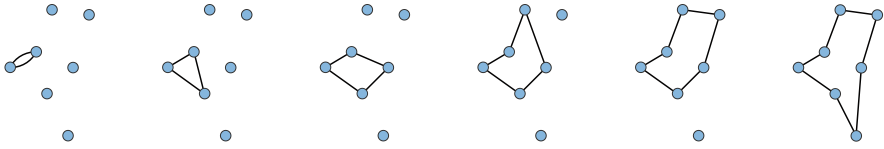

Fig. 4.4 Running the cheapest insertion for a tiny TSP instance

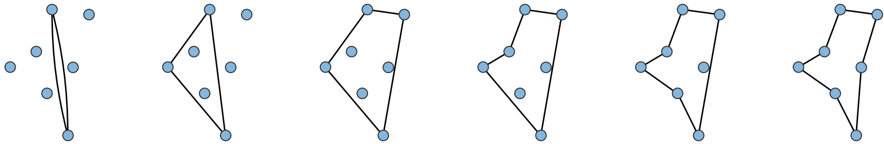

Fig. 4.5 Running the farthest insertion for a tiny TSP instance

4.3.1.5 Farthest Insertion

The farthest insertion heuristic is similar to the previous one, but it selects the city whose insertion causes the most significant detour. However, each city is inserted at the best possible place in the tour. Figure 4.5 illustrates this greedy method. It seems counter-intuitive to choose the most problematic city at each step. However, this type of construction reveals less myopic and frequently produces better final solutions than the previous heuristics.

Here, we have provided only a limited range of greedy constructive methods that have been proposed for the TSP. The quality of the solutions they produce varies. It is usually not challenging to find problem instances for which a greedy heuristic is misguided and makes choices increasingly bad. On points uniformly distributed on the Euclidean plane, they typically provide solutions a few tens of percent above the optimum.

4.3.2 Greedy Heuristic for Graph Coloring

After reviewing several methods for the TSP, it is necessary to present a not too naive example for another problem.

A relatively elaborate greedy method for coloring the vertices of a graph tries to determine the node for which assigning a color may be the most problematic. The DSatur [1] method assumes it corresponds to the node with already colored neighbors using the broadest color palette. For this purpose, the *saturation* degree of a vertex v is defined, noted DS (v), corresponding to the number of different colors used by the vertices adjacent to v. At equal degree of saturation—particularly at the start, when no vertex is colored—the node with the highest degree is selected. At equivalent degree and saturation degree, the nodes are arbitrarily selected. Algorithm 4.3 formalizes this greedy method.

Algorithm 4.3: *DSatur* algorithm for graph coloring. The greedy criterion used by this algorithm is the saturation degree of the vertices, corresponding to the number of different colors used by adjacent nodes

```
   Input: Undirected graph G = (V, E);
   Result: Vertex coloring
1  Color with 1 the vertex v with the highest degree
2  R ← V \ v
3  colors ← 1
4  while R ≠ ∅ do
5  |   ∀v ∈ R, compute DS(v)
6  |   Choose v′ maximizing DS(v′), with the highest possible degree
7  |   Find the smallest k (1 ⩽ k ⩽ colors + 1) such that color k is feasible for v′
8  |   Assign color k to v′
9  |   if k > colors then
10 |   |_ colors = k
11 |_  R ← R \ v′
```

4.4 Improvement of Greedy Procedures

The chief drawback of a greedy construction is that it never changes a choice performed in a myopic way. Conversely, the shortcoming of a complete enumerative method is the exponential growth of the computational effort with the problem size. To limit this growth, it is therefore necessary to limit the branching. This is typically achieved on the basis of greedy criteria. This section reviews two partial enumeration techniques that have been proposed to improve a greedy algorithm.

First, the beam search was proposed within the framework of an application in speech recognition [5]. Second is the more recent pilot method, proposed by Duin and Voß [3]. It was presented as a new metaheuristic. Other frames have been derived from it [4].

4.4.1 Beam Search

Beam search is a partial breadth-first search. Instead of keeping all the branches, at most, p are kept at each level, on the basis of the incremental cost function $c(s, e)$. Arriving at level k, the partial solution at the first level is completed with the element e which leads to the best solution at the last enumerated level. Figure 4.6 illustrates the principle of a beam search.

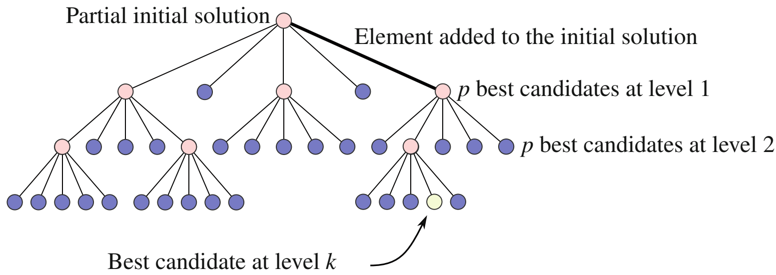

Fig. 4.6 Beam search with $p = 3$ and $k = 3$. Before definitively choosing the element to insert in the partial solution, a breadth-first search is carried out up to a depth of k, only retaining the p best candidates at each depth

A beam search variant proceeds by making a complete enumeration up to a level containing more than p nodes. The p best of them are retained to generate the candidates for the next level.

4.4.2 Pilot Method

The framework of the pilot method requires a so-called pilot heuristic to fully complete a partial solution. This pilot heuristic can be a simple greedy method, for example, the nearest neighbor heuristic for the TSP, but it can equally be a much more sophisticated method, such as one of those presented in the following chapters.

The pilot method enumerates all the partial solutions that can be obtained by including an element to the starting solution. The pilot heuristic is then applied to all these partial solutions to end up with as many complete solutions. The partial solution at the origin of the best complete solution is used as the new starting solution, until there is nothing more to add. Figure 4.7 illustrates two steps of the method.

Algorithm 4.4 specifies how the pilot metaheuristic works. In this framework, the ultimate "partial" solutions represent a feasible complete solution which is not necessarily the solution returned by the algorithm. Indeed, the pilot heuristic can generate a complete solution that does not necessarily include the elements of the initial partial solution, especially if it includes an improvement technique more elaborated than a simple greedy constructive method.

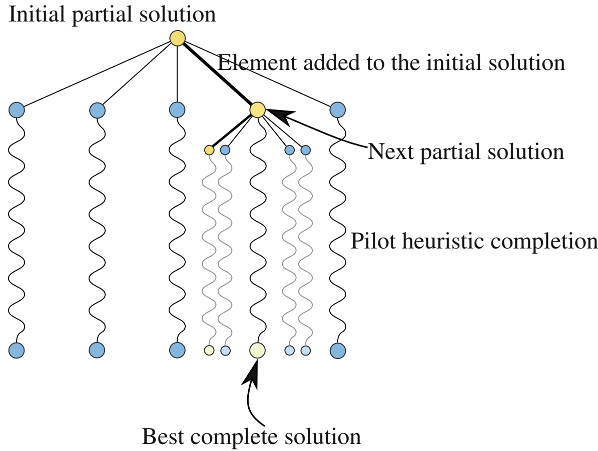

Fig. 4.7 Pilot method. An element is included in the partial solution; then a pilot constructive heuristic is applied to fully complete it. The process is repeated with another element added to the partial solution. The element finally inserted is the one that led to the best complete solution

---

Algorithm 4.4: Frame of a pilot method

Input: s_p trivial partial solution; set E of elements constituting a solution; pilot heuristic $h(s_e)$ for completing a partial solution s_e; fitness function $f(s)$

Result: Complete solution $s*$

```
1  R ← E                                        // Elements that can be added to s
2  while R ≠ ∅ do
3  |   v ← ∞
4  |   forall e ∈ R do
5  |   |   Complete s_p with e to get s_e
6  |   |   Apply h(s_e) to get a complete solution s
7  |   |   if f(s) ⩽ v then
8  |   |   |   v ← f(s)
9  |   |   |   s_r ← s_e
10 |   |   |   if s is better than s* then Store the improved solution
11 |   |   |   |   s* ← s
12 |   s_p ← s_r                          // Add an element to the partial solution s_p
13 |   Remove from R the elements that cannot properly be added to s_p
```

Code 4.4 provides an implementation of the pilot method for the TSP. The pilot heuristic is the nearest neighbor.

Code 4.4 tsp_pilot_nearest_neighbor.py Implementation of a pilot method with the nearest neighbor (Code 4.3) as pilot heuristic

```
from tsp_utilities import tsp_length                              # Listing 12.2
from tsp_nearest_neighbor import *                                # Listing  4.3

######### Constructive algorithm with Nearest Neighbor as Pilot method
def tsp_pilot_nearest_neighbor(n,                               # Number of cities
                               d):                              # Distance matrix
    tour = [i for i in range(n)]                      # All cities must be in tour

    for q in range(n - 1):               # Cities up to q at their final position
        length_r = tsp_length(d, tour)
        to_insert = q
        for r in range(q, n):          # Choose next city to insert at position q
            sol = [tour[i] for i in range(n)]                   # Copy of tour in sol
            sol[q], sol[r] = sol[r], sol[q]         # Tentative city at postion q
            sol[q:n], _ = tsp_nearest_neighbor(d, sol[q:n])
            tentative_length = tsp_length(d, sol)
            if length_r > tentative_length:
                length_r = tentative_length
                to_insert = r

        # Put definitively to_insert at position q
        tour[q], tour[to_insert] = tour[to_insert], tour[q]

    return tour, tsp_length(d, tour)
```

Problems

4.1 Random Permutation

Write a procedure to generate a random permutation of n elements contained in an array p. It is desired a probability of $1/n$ to find any element in any position in p. Describe the inadequacy of Algorithms 4.5 and 4.6.

Algorithm 4.5: Bad algorithm to generate a random permutation of n elements

Input: A set of n elements $e_1, \dots, e_n$
Result: A permutation p of the elements

1. $i \leftarrow 0$ // Number of element already chosen
2. **while** $i \neq n$ **do**
3.   Draw a random number u uniformly between 1 and n
4.   **if** e_u *is not already chosen* **then**
5. $i \leftarrow i + 1$
6. $p_i \leftarrow e_u$

Algorithm 4.6: Another bad algorithm to generate a random permutation of n elements

```
   Input: A set of n elements e_1, ..., e_n
   Result: A permutation p of the elements
1  i ← 0                                         // Number of element already chosen
2  while i ≠ n do
3      Draw a random number u uniformly between 1 and n
4      i ← i + 1
5      if e_u is already chosen then
6          Find the next u' such that e_u' is not chosen
7          p_i ← e_u'
8      else
9          p_i ← e_u
```

4.2 Greedy Algorithms for the Knapsack

Propose three different greedy algorithms for the knapsack problem.

4.3 Greedy Algorithm for the TSP on the Delaunay

We want to build the tour of a traveling salesman (on the Euclidean plane) using only edges belonging to the Delaunay triangulation. Is this always possible? If this is not the case, provide a counter-example; otherwise, propose a greedy method and analyze its complexity.

4.4 TSP with Edge Subset

To speed up a greedy method for the TSP, only the 40 shortest edges adjacent to each vertex are considered. Is this likely to reduce the algorithmic complexity of the method? Can this cause some issues?

4.5 Constructive Method Complexity

What is the complexity of the nearest neighbor heuristic for TSP? Same question if we use this heuristic in a beam search by retaining p nodes at each depth and that we go to k levels down. Similar question for the pilot method where we equally employ the nearest neighbor as the pilot heuristic.

4.6 Beam Search and Pilot Method Applications

We consider a TSP instance on five cities. Table 4.1 gives its distance matrix.

Apply a beam search to this instance, starting from the city 1. At each level, only $p = 2$ nodes are retained, and the tree is developed up to $k = 3$ levels down.

Apply a pilot method to this instance, considering the nearest neighbor as the pilot heuristic.

Table 4.1 Distance matrix for Problem 4.6

	1	2	3	4	5
1	—	5	3	19	7
2	13	—	1	18	6
3	12	4	—	14	6
4	11	9	8	—	10
5	23	11	7	21	—

4.7 Greedy Algorithm Implementation for Scheduling

Propose two greedy heuristics for the permutation flowshop problem. Compare their quality on problem instances from the literature.

4.8 Greedy Methods for the VRP

Propose two greedy heuristic methods for the vehicle routing problem.

References

1. Brélaz, D.: New methods to color the vertices of a graph. Commun. ACM. **22**(4), 251–256 (1979). https://doi.org/10.1145/359094.359101
2. Dakin, R.J.: A tree search algorithm for mixed integer programming problems. Comput. J. **8**(3), 250–255 (1965). https://doi.org/10.1093/comjnl/8.3.250
3. Duin, C., Voß S.: The pilot method: a strategy for heuristic repetition with application to the steiner problem in graphs. Networks. **34**, 181–191 (1999). https://doi.org/10.1002/(SICI)1097-0037(199910)34:3%3C181::AID-NET2%3E3.0.CO;2-Y
4. Furcy, D., Koenig, S.: Limited discrepancy beam search. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 125–131 (2005)
5. Lowerre, B.: The Harpy Speech Recognition System. Ph. D. Thesis, Carnegie Mellon University (1976)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.