CHAPTER 2

Single-Solution Based Metaheuristics

While solving optimization problems, single-solution based metaheuristics (S-metaheuristics) improve a single solution. They could be viewed as "walks" through neighborhoods or search trajectories through the search space of the problem at hand [163]. The walks (or trajectories) are performed by iterative procedures that move from the current solution to another one in the search space. S-metaheuristics show their efficiency in tackling various optimization problems in different domains.

In this chapter, a unified view of the common search concepts for this class of metaheuristics is presented. Then, the main existing algorithms, both from the design and the implementation point of view, are detailed in an incremental way.

This chapter is organized as follows. After the high-level template of S-metaheuristics is presented, Section 2.1 details the common search components for S-metaheuristics: the definition of the neighborhood structure, the incremental evaluation function, and the determination of the initial solution. The concept of very large neighborhoods is also presented. Section 2.2 discusses the landscape analysis of optimization problems. Sections 2.3, 2.4, and 2.5 introduce, in an incremental manner, the well-known S-metaheuristics: local search, simulated annealing, and tabu search. Then, Sections 2.6, 2.7, and 2.8 present, respectively, the iterative local search, the variable neighborhood search, and the guided local search algorithms. In Section 2.9, other S-metaheuristics are dealt with, such as GRASP, the noisy and the smoothing methods. Finally, Section 2.10 presents the ParadisEO–MO (moving objects) module of the ParadisEO framework that is dedicated to the implementation of S-metaheuristics. Some design and implementations of S-metaheuristics such as local search, simulated annealing, tabu search, and iterated local search are illustrated.

2.1 COMMON CONCEPTS FOR SINGLE-SOLUTION BASED METAHEURISTICS

S-metaheuristics iteratively apply the generation and replacement procedures from the current single solution (Fig. 2.1). In the generation phase, a set of candidate solutions are generated from the current solution s. This set $C(s)$ is generally obtained by local

Metaheuristics: From Design to Implementation, by El-Ghazali Talbi

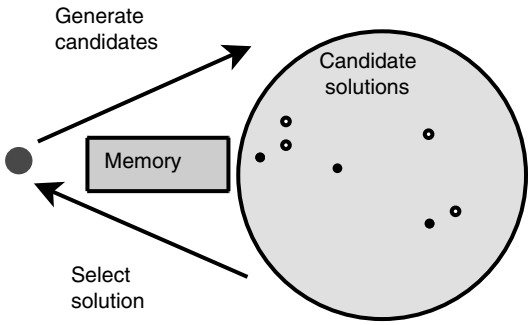

FIGURE 2.1 Main principles of single-based metaheuristics.

transformations of the solution. In the replacement phase,[1] a selection is performed from the candidate solution set $C(s)$ to replace the current solution; that is, a solution $s' \in C(s)$ is selected to be the new solution. This process iterates until a given stopping criteria. The generation and the replacement phases may be *memoryless*. In this case, the two procedures are based only on the current solution. Otherwise, some history of the search stored in a memory can be used in the generation of the candidate list of solutions and the selection of the new solution. Popular examples of such S-metaheuristics are local search, simulated annealing, and tabu search. Algorithm 2.1 illustrates the high-level template of S-metaheuristics.

Algorithm 2.1 High-level template of S-metaheuristics.

Input: Initial solution s_0.
$t = 0$;
Repeat
  /* Generate candidate solutions (partial or complete neighborhood) from s_t */
  Generate($C(s_t)$) ;
  /* Select a solution from $C(s)$ to replace the current solution s_t */
  s_{t+1} = Select($C(s_t)$) ;
  $t = t + 1$;
Until Stopping criteria satisfied
Output: Best solution found.

The common search concepts for *all* S-metaheuristics are the definition of the *neighborhood* structure and the determination of the *initial solution*.

2.1.1 Neighborhood

The definition of the neighborhood is a required common step for the design of any S-metaheuristic. The neighborhood structure plays a crucial role in the performance

[1] Also named transition rule, pivoting rule, and selection strategy.

of an S-metaheuristic. If the neighborhood structure is not adequate to the problem, any S-metaheuristic will fail to solve the problem.

Definition 2.1 Neighborhood. *A neighborhood function N is a mapping $N : S \rightarrow 2^S$ that assigns to each solution s of S a set of solutions $N(s) \subset S$.*

A solution s' in the neighborhood of s ($s' \in N(S)$) is called a *neighbor* of s. A neighbor is generated by the application of a *move* operator m that performs a small perturbation to the solution s. The main property that must characterize a neighborhood is *locality*. Locality is the effect on the solution when performing the move (perturbation) in the representation. When small changes are made in the representation, the solution must reveal small changes. In this case, the neighborhood is said to have a strong locality. Hence, a S-metaheuristic will perform a meaningful search in the landscape of the problem. Weak locality is characterized by a large effect on the solution when a small change is made in the representation. In the extreme case of weak locality, the search will converge toward a random search in the search space.

The structure of the neighborhood depends on the target optimization problem. It has been first defined in continuous optimization.

Definition 2.2 *The neighborhood $N(s)$ of a solution s in a continuous space is the ball with center s and radius equal to ϵ with $\epsilon > 0$.*

Hence, one have $N(s) = \{s' \in R^n / \|s' - s\| < \epsilon\}$. By using the Euclidean norm, we obtain $\|s' - s\| = \sqrt{(s'_1 - s_1)^2 + (s'_2 - s_2)^2 + \cdots + (s'_n - s_n)^2}$, which is the Euclidean distance between s' and s (Fig. 2.2).

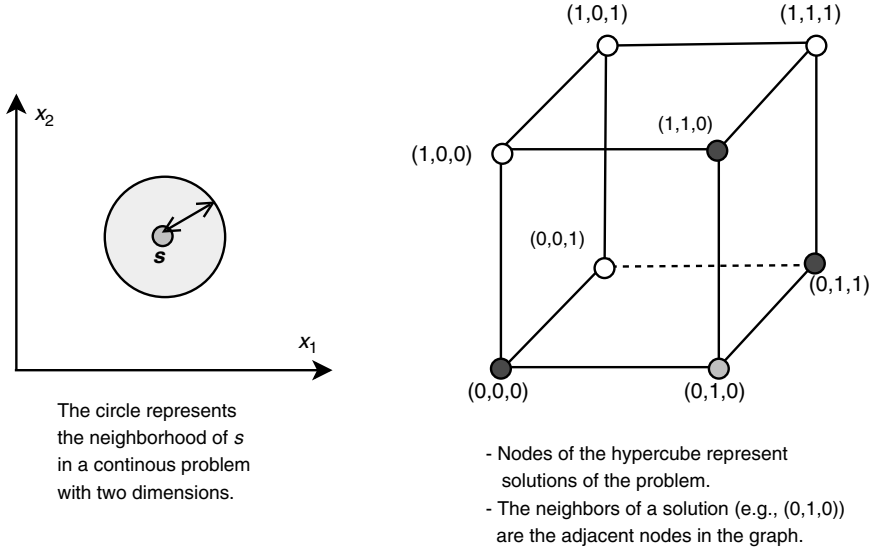

FIGURE 2.2 Neighborhoods for a continuous problem and a discrete binary problem.

If the gradient (derivates of the objective function) can be calculated or approximated, a steepest descent (or ascent) indicating the direction to take can be applied. The main parameter in this first strategy is the length of the move in the selected direction. Otherwise, a random or deterministic generation of a subset of neighbors is carried out. In random generation of a neighbor, the normal variable $(0, \sigma_i)$ is added to the current value, where σ_i represents the main parameter in this second strategy.

The concept of neighborhood can be extended to discrete optimization.

Definition 2.3 *In a discrete optimization problem, the neighborhood $N(s)$ of a solution s is represented by the set $\{s'/d(s', s) \leq \epsilon\}$, where d represents a given distance that is related to the move operator.*

The neighborhood definition depends strongly on the representation associated with the problem at hand. Some usual neighborhoods are associated with traditional encodings.

The natural neighborhood for binary representations is based on the Hamming distance. In general, a distance equal to 1 is used. Then, the neighborhood of a solution s consists in flipping one bit of the solution. For a binary vector of size n, the size of the neighborhood will be n. Figure 2.2 shows the neighborhood for a continuous problem and a discrete binary problem. An Euclidean distance less than ϵ and a Hamming distance equal to 1 are used to define the neighborhood.

The Hamming neighborhood for binary encodings may be extended to any discrete vector representation using a given alphabet $\sum$. Indeed, the substitution can be generalized by replacing the discrete value of a vector element by any other character of the alphabet. If the cardinality of the alphabet $\sum$ is k, the size of the neighborhood will be $(k - 1)$. n for a discrete vector of size n.

For permutation-based representations, a usual neighborhood is based on the swap operator that consists in exchanging (or swapping) the location of two elements s_i and s_j of the permutation. For a permutation of size n, the size of this neighborhood is $n(n - 1)/2$. This operator may also be applied to any linear representation. Figure 2.3 shows the neighborhood associated with a combinatorial optimization problem using a permutation encoding. The distance is based on the swap move operator.

Once the concept of neighborhood has been defined, the local optimality property of a solution may be given.

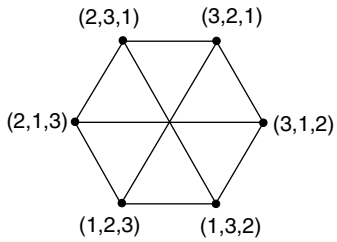

FIGURE 2.3 An example of neighborhood for a permutation problem of size 3. For instance, the neighbors of the solution (2, 3, 1) are (3, 2, 1), (2, 1, 3), and (1, 3, 2).

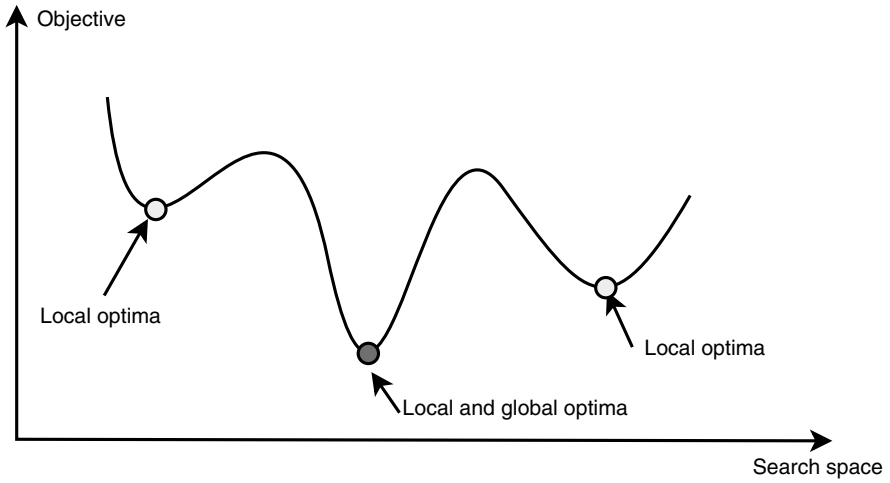

FIGURE 2.4 Local optimum and global optimum in a search space. A problem may have many global optimal solutions.

Definition 2.4 Local optimum. *Relatively to a given neighboring function N, a solution $s \in S$ is a local optimum if it has a better quality than all its neighbors; that is, $f(s) \leq f(s')$[2] for all $s' \in N(s)$ (Fig. 2.4).*

For the same optimization problem, a local optimum for a neighborhood N_1 may not be a local optimum for a different neighborhood N_2.

Example 2.1 *k*-distance neighborhood versus *k*-exchange neighborhood. For permutation problems, such as the TSP, the exchange operator (swap operator) may be used (Fig. 2.5). The size of this neighborhood is $n(n-1)/2$ where n represents the number of cities. Another widely used operator is the k-opt operator,[3] where k edges are removed from the solution and replaced with other k edges. Figure 2.5 (resp. Fig. 2.6) illustrates the application of the *2-opt operator* (resp. *3-opt operator*) on a tour. The neighborhood for the 2-opt operator is represented by all the permutations obtained by removing two edges. It replaces two directed edges (π_i, π_{i+1}) and (π_j, π_{j+1}) with (π_i, π_j) and (π_{i+1}, π_{j+1}), where $i \neq j-1, j, j+1 \; \forall i, j$. Formally, the neighbor solution is defined as

$$\pi'(k) = \pi(k) \quad \text{for } k \leq i \text{ or } k > j$$
$$\pi'(k) = \pi(i+j+1-k) \quad \text{for } i < k \leq j$$

The size of the neighborhood for the 2-opt operator is $[(n(n-1)/2) - n]$; all pairs of edges are concerned except the adjacent pairs.

[2]For a minimization problem.
[3]Also called k-exchange operator.

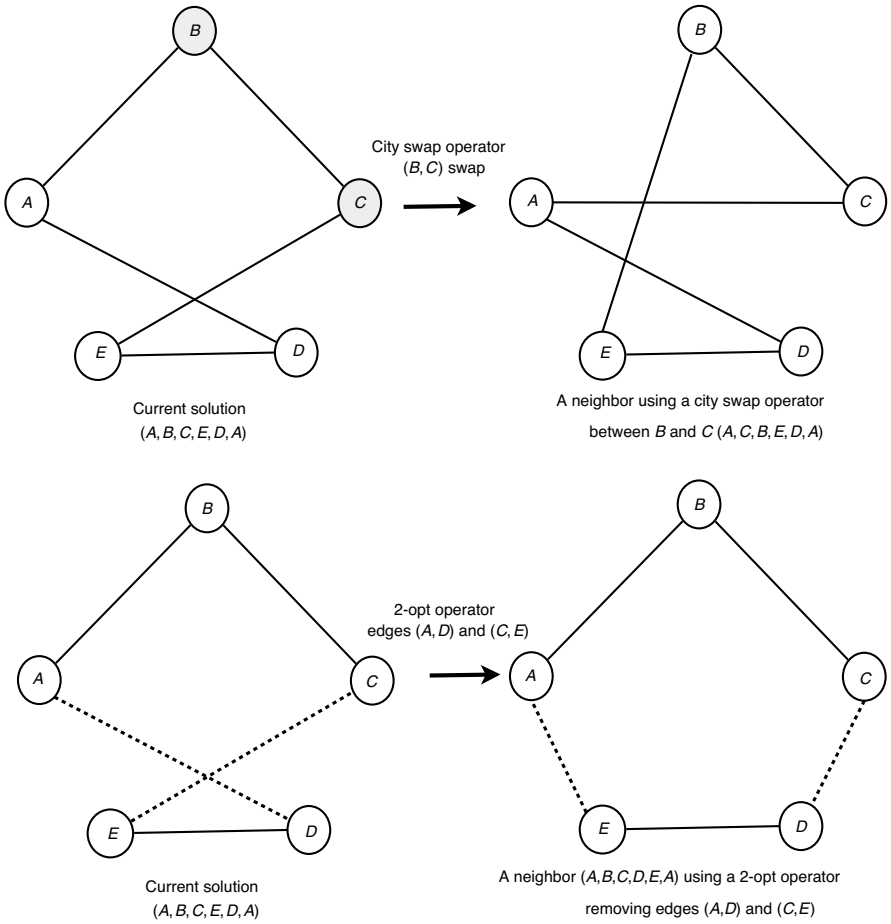

FIGURE 2.5 City swap operator and 2-opt operator for the TSP.

As mentioned, the efficiency of a neighborhood is related not only to the representation but also to the type of problems to solve. For instance, in scheduling problems, permutations represent a priority queue. Then, the relative order in the sequence is very important, whereas in the TSP it is the adjacency of the elements that is important. For scheduling problems, the 2-opt operator will generate a very large variation (weak locality), whereas for routing problems such as the TSP, it is a very efficient operator because the variation is much smaller (strong locality).

Example 2.2 Neighborhoods for permutation scheduling problems. For permutations representing sequencing and scheduling problems, the k-opt family of operators is not well suited. The following operators may be used:

- **Position-based neighborhood:** Figure 2.7 illustrates an example of a position-based operator, the *insertion operator*. In the insertion operator, an element at one

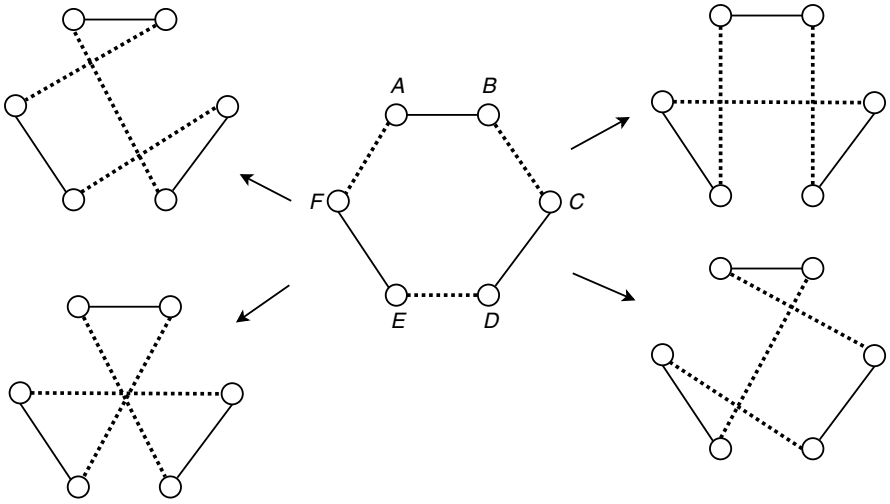

FIGURE 2.6 3-opt operator for the TSP. The neighbors of the solution (*A,B,C,D,E,F*) are (*A,B,F,E,C,D*), (*A,B,D,C,F,E*), (*A,B,E,F,C,D*), and (*A,B,E,F,D,C*).

position is removed and put at another position. The two positions are randomly selected.

- **Order-based neighborhood:** Many order-based operators can be used such as the exchange operator where arbitrarily selected two elements are swapped as shown in Fig. 2.8, and the inversion operator where two elements are randomly selected and the sequence of elements between these two elements are inverted as shown in Fig. 2.9.

Those operators are largely used in scheduling problems and seldom for routing problems such as the TSP for efficiency reasons.

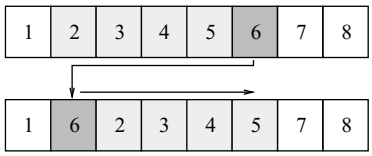

FIGURE 2.7 Insertion operator.

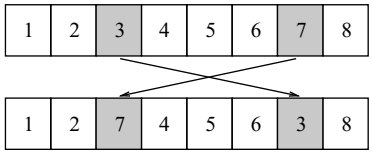

FIGURE 2.8 Exchange operator.

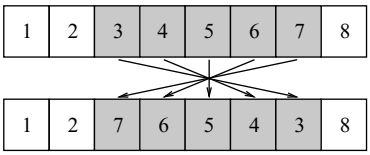

FIGURE 2.9 Inversion operator.

2.1.2 Very Large Neighborhoods

In designing a S-metaheuristic, there is often a compromise between the size (or diameter) and the quality of the neighborhood to use and the computational complexity to explore it. Designing large neighborhoods may improve the quality of the obtained solutions since more neighbors are considered at each iteration (Fig. 2.10). However, this requires an additional computational time to generate and evaluate a large neighborhood.

The size of the neighborhood for a solution s is the number of neighboring solutions of s. Most of the metaheuristics use small neighborhoods that is, in general, a polynomial function of the input instance size (e.g., linear or quadratic function). Some large neighborhoods may be high-order polynomial or exponential function of the size of the input instance. The neighborhood is *exponential* if its size grows exponentially with the size of the problem. Then, the complexity of the search will be much higher. So, the main issue here is to design efficient procedures to explore large neighborhoods. These efficient procedures identify improving neighbors or the best neighbor without the enumeration of the whole neighborhood (Fig. 2.11).

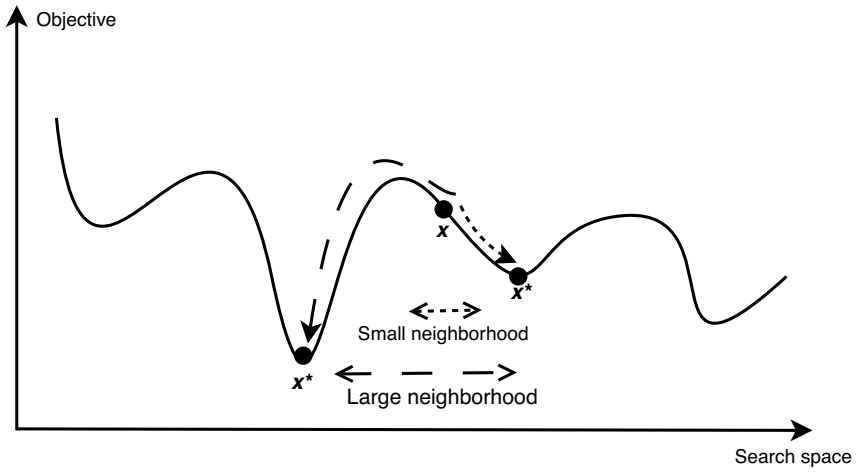

FIGURE 2.10 Impact of the size of the neighborhood in local search. Large neighborhoods improving the quality of the search with an expense of a higher computational time.

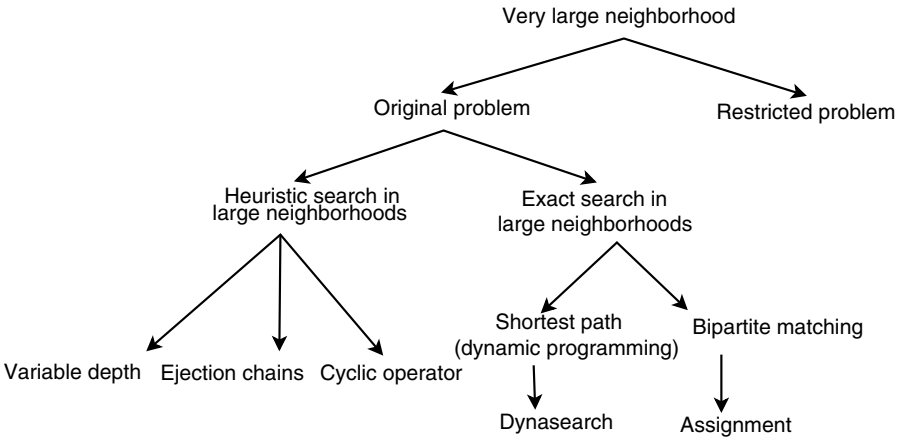

FIGURE 2.11 Very large neighborhood strategies.

A comprehensive survey of very large search neighborhoods may be found in Ref. [14].

2.1.2.1 Heuristic Search in Large Neighborhoods This approach consists in searching heuristically very large neighborhoods. A partial set of the large neighborhood is generated. Hence, finding the best neighbor (i.e., local optimum) is not guaranteed.

Let us consider a given neighborhood N defined by neighbors of distance $k = 1$ from a solution $s(N_1(s) = \{s' \in S/d(s, s') = 1\})$. In a similar way, a larger neighborhood $N_k(s)$ of distance k $(d = 2, 3, \ldots, n)$[4] is defined as the set

$$N_k(s) = N_{k-1}(s) \cup \{s''/\exists s' \in N_k(s) : s'' \in N_1(s')\}$$

Larger is the distance k, larger the neighborhood is. Since $N_n(s)$ is the whole search space, finding the best solution in this neighborhood is NP-hard if the original problem is NP-hard. Variable-depth search methods represent those strategies that explore partially the $N_k(s)$ neighborhoods. In general, a variable distance move (k-distance or k-exchange) with a distance of 2 or 3 may be appropriate.

This idea of variable-depth search methods has been first used in the classical Lin–Kernighan heuristic for the TSP [508] and the ejection chains. An *ejection chain* is a sequence of coordinated moves. Ejection chains were proposed first by Glover to solve the TSP problem [324]. They are based on alternating path methods, that is, alternating sequence of addition and deletion moves.

[4] n is related to the size of the problem (i.e., maximum distance between any two solutions).

Example 2.3 Ejection chains for vehicle routing problems. This example illustrates the application of ejection chains to the capacitated vehicle routing problem [645]. An ejection chain is defined as a sequence of coordinated moves of customers from one route to a successive one. Each ejection chain involves k levels (routes) starting at route 1 and ending at route k (Fig. 2.12). It ejects one node from route 1 to the successive route 2, one node from the route 2 to the route 3, and so on and one node from level $k-1$ to k (Fig. 2.12). The last node is bumped from route k to the route 1. The successive moves represent an ejection chain if and only if no vertex appears more than once in the solution. Figure 2.12 illustrates the ejection chain moves.

The heuristically searched large neighborhoods (e.g., ejection chain strategy, variable depth) have been applied successfully to various combinatorial optimization problems: clustering [220], vehicle routing [645], generalized assignment [832], graph partitioning [230], scheduling [712], and so on.

Many neighborhoods are defined by cycles. Cycle operators may be viewed as the generalization of the well-known 2-opt operator for the TSP problem, where more than two elements of the solution are involved to generate a neighbor.

Example 2.4 Cyclic exchange. A very large neighborhood based on cyclic exchange may be introduced into partitioning problems [15]. Let $E = \{e_1, e_2, \ldots, e_n\}$ be a set of elements to cluster into q subsets. A q-partition may be represented by the clustering $S = \{S_1, S_2, \ldots, S_q\}$. A 2-neighbor of the solution $S = \{S_1, S_2, \ldots, S_q\}$ is obtained by swapping two elements that belong to two different subsets. In its general form, the cyclic operator is based on swapping more than two elements and then involves more than two subsets (Fig. 2.13). A move involving k subsets S_i is represented by a cyclic permutation π of size k ($k \leq q$), where $\pi(i) = j$ represents the fact that the element e_i of the subset S_i is moved to the subset S_j. Hence, a solution $Q = \{Q_1, Q_2, \ldots, Q_q\}$ is a cyclic neighbor of the solution $S = \{S_1, S_2, \ldots, S_q\}$ if there exists a sequence $L = (o_1, \ldots, o_m)$ of size $m \leq k$ of moving single elements that generates Q from S. Let us notice that the first and the last move must concern the same partition (Fig. 2.13). The size of this neighborhood is $O(n^k)$. Using the cyclic operator, finding the best neighbor in the very large neighborhood has been reduced to the subset disjoint minimum cost cycle problem [769].

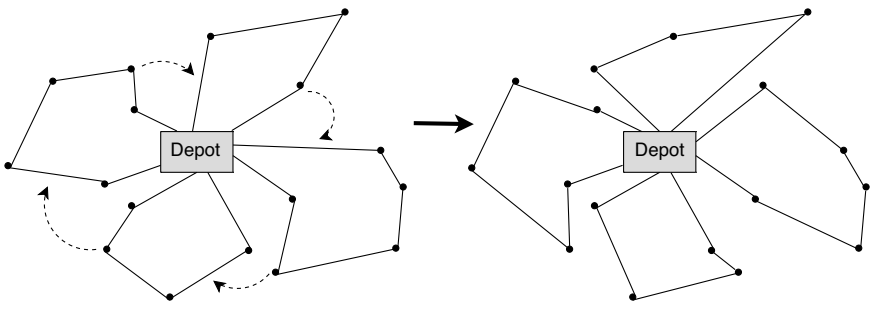

FIGURE 2.12 A four-level ejection chain for vehicle routing problems. Here, the ejection chain is based on a multinode insertion process.

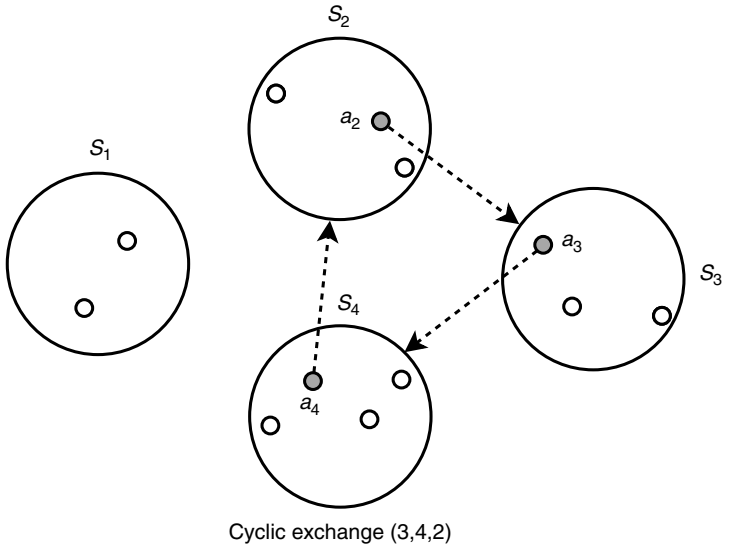

FIGURE 2.13 Very large neighborhood for partitioning problems: the cyclic exchange operator. Node a_2 is moved from subset S_2 to subset S_3, node a_3 is moved from subset S_3 to subset S_4, and node a_4 is moved from subset S_4 to subset S_2.

Indeed, finding an improving neighbor is based on the construction of the *improvement graph* [769]. Let $E = \{e_1, e_2, \ldots, e_n\}$ be the set of elements associated with the partitioning problem and let $x[i]$ be the subset containing the element e_i. The improvement graph $G = (V, E)$ is constructed as follows. The set $V = \{1, 2, \ldots, n\}$ of nodes is defined by the set of elements $(i = 1, \ldots, n)$ corresponding to the indices of the elements belonging to E. An edge $(i, j) \in E$ is associated with each move operation that transfers the element i from $x[i]$ to $x[j]$ and removes the element j from $x[j]$. Each edge (i, j) is weighted by a constant c_{ij}, which represents the cost increase of $x[j]$ when the element i is added to the set $x[i]$ and j is removed:

$$c_{ij} = d[\{i\} \cup x[j] - \{j\}] - d[x[j]]$$

Definition 2.5 Subset-disjoint cycle. *A cycle Z in a graph $G = (V, E)$ is subset disjoint if*

$$\forall i, j \in Z, x[i] \neq x[j]$$

that is, the nodes of Z are all in different subsets.

It is straightforward to see that there is a correspondence between cyclic exchange for the partitioning problem and subset-disjoint cycles in the improvement graph, that is, for every negative cost cyclic exchange, there is a negative cost subset-disjoint cycle in the improvement graph. The decision problem of whether there is a subset-disjoint cycle in the improvement graph is an NP-complete problem. Hence, the problem of finding a

negative cost subset-disjoint cycle is NP-hard [769]. So, heuristics are generally used to explore this very large neighborhood [769].

The cyclic operator has been mainly used in grouping problems such as graph partitioning problems, vehicle routing problems [248,307], scheduling problems [283], constrained spanning tree problems [16], or assignment problems [759].

2.1.2.2 Exact Search in Large Neighborhoods This approach consists in searching exactly very large neighborhoods. The main goal is to find an improving neighbor. They are based on efficient procedures that search specific large (even exponential) neighborhoods in a polynomial time.

Searching the best or an improving neighbor for some large neighborhoods may be modeled as an optimization problem. This class of method is mainly based on network flow-based improvement algorithms. Two types of efficient search algorithms solving this optimization problem may be found in the literature [14]:

- **Path finding:** Where shortest path and dynamic programming algorithms are used to search the large neighborhood and identify improving neighbors.
- **Matching:** Where well-known polynomial-time matching algorithms are used to explore large neighborhoods for specific problems.

For instance, swap-based neighborhoods in permutations can be generalized by the use of multiple compounded swaps (e.g., dynasearch).

Definition 2.6 Independent swaps. *Given a permutation* $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$*, a swap move* (i, j) *consists in exchanging the two elements* π_i *and* π_j *of the permutation* π*. Two swap moves* (i, j) *and* (k, l) *are independent if* $(max\{i, j\} < min\{k, l\})$ *or* $(min\{i, j\} > max\{k, l\})$*.*

A swap-based large neighborhood of a permutation can be defined by the union of an arbitrary number of independent swap moves.

Example 2.5 Dynasearch. Dynasearch is another classical large neighborhood approach that has been introduced for optimization problems where solutions are encoded by permutations (e.g., sequencing problems) [150]. The dynasearch two-exchange move is based on improving a Hamiltonian path between two elements $\pi(1)$ and $\pi(n)$ (see Fig. 2.14). The operator deletes the edges $(\pi(i), \pi(i + 1))$, $(\pi(j), \pi(j + 1))$, $(\pi(k), \pi(k + 1))$, and $(\pi(l), \pi(l + 1))$. The following conditions must

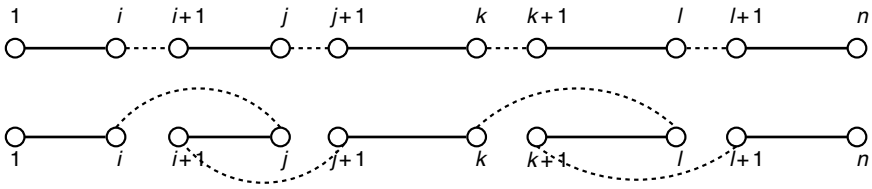

FIGURE 2.14 Dynasearch using two independent two-exchange moves: polynomial exploration of exponentially large neighborhoods.

hold: $1 < i + 1 < j \leq n$, $\pi(j + 1) \neq \pi(i)$, and $1 < k + 1 < l \leq n$, $\pi(l + 1) \neq \pi(k)$. The segments are independent if $j \leq k$ or $l < i$. Indeed, when this condition holds, the segments that are exchanged by the two moves do not share any edge.

The size of this exponential neighborhood is $O(2^{n-1})$. The problem of finding the best neighbor is reduced to the shortest path problem in the improvement graph. Then, it is explored in an efficient polynomial way by a dynamic programming algorithm in $O(n^2)$ for the TSP and in $O(n^3)$ for scheduling problems [94,396]. If the compounded swaps are limited to adjacent pairs, the time complexity of the algorithm is reduced to $O(n^2)$ in solving scheduling problems.

Another exponential neighborhood is defined by the *assignment neighborhood* [345]. It is based on the multiple application of the insertion operator. Let us consider the TSP problem. Giving a tour $S = (1, 2, \ldots, n, 1)$, where $d[i, j]$ represents the distance matrix. The assignment neighborhood consists in constructing the following bipartite improvement graph: select and eject k nodes $V = \{v_1, v_2, \ldots, v_k\}$ from the tour S, where $k = \lfloor n/2 \rfloor$. The set $U = \{u_1, u_2, \ldots, u_{n-k}\}$ represents the nonejected nodes of the tour S; that is, $U \cup V = S$. Then, a subtour $S' = (u_1, u_2, \ldots, u_{n-k}, u_1)$ is created. Finally, a complete bipartite graph $G = (N, V, E)$ is constructed, where $N = \{q_i : i = 1, \ldots, n - k\}$, q_i represents the edge (u_i, u_{i+1}) for $i = 1$ to $n - k - 1$,

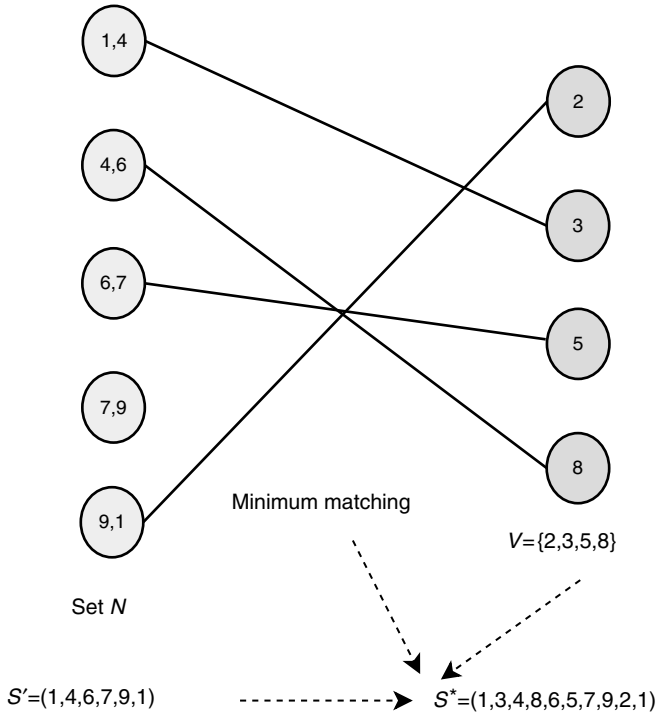

FIGURE 2.15 Assignment neighborhood: constructing the bipartite improvement graph and finding the optimal matching.

and $q_{n-k} = (u_{n-k}, u_1)$ (Fig. 2.15). The set of edges $E = (q_i, v_j)$ is weighted by $c[q_i, v_j] = d[u_i, v_j] + d[v_j, u_{i+1}] - d[u_i, u_{i+1}]$.

A neighbor S^* for the tour S is defined by inserting the nodes of V in the subtour S'. No more than one node is inserted between two adjacent nodes of the subtour S'. The problem of finding the best neighbor is then reduced to the minimum cost matching on a bipartite improvement graph. Figure 2.15 illustrates the construction of the improvement graph on the tour $S = (1, 2, \ldots, 9, 1)$ composed of nine nodes, where $V = \{2, 3, 5, 8\}$ and $U = \{1, 4, 6, 7, 9\}$. Hence, the subtour S' is defined as $(1, 4, 6, 7, 9, 1)$. For simplicity reasons, only the edges of the minimum cost matching are shown for the bipartite graph G. According to the optimal matching, the obtained tour is $(1, 3, 4, 8, 6, 5, 7, 9, 2, 1)$.

The generalization of the assignment operator for arbitrary k and n, and different inserting procedures (e.g., edges instead of nodes) in which the problem is reduced to a minimum weight matching problem is proposed [344].

Matching-based neighborhoods have been applied to many combinatorial optimization problems such as the TSP [347], quadratic assignment problem [203], and vehicle routing problems [225].

2.1.2.3 Polynomial-Specific Neighborhoods Some NP-hard optimization problems may be solved in an efficient manner by restricting the input class instances or by adding/deleting constraints to the target optimization problem.

For instance, many graph problems (e.g., Steiner tree, TSP) are polynomial for *specific* instances of the original NP-hard problem (e.g., series–parallel, outerplanar, Halin). This fact may be exploited in defining large neighborhoods based on those special cases solvable in polynomial time. Indeed, this strategy consists in transforming the input instance into a specific instance that can be solved in a polynomial manner. This class of strategies are not yet well explored in the literature [14].

Example 2.6 Halin graphs—a polynomial solvable instance class. Halin graphs deal with properties of minimal connectivity in graphs. They are generalizations of

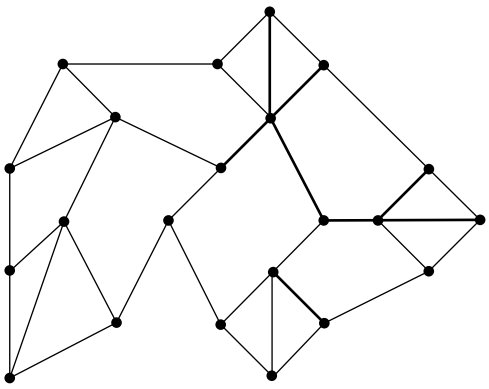

FIGURE 2.16 An example of a Halin graph [734].

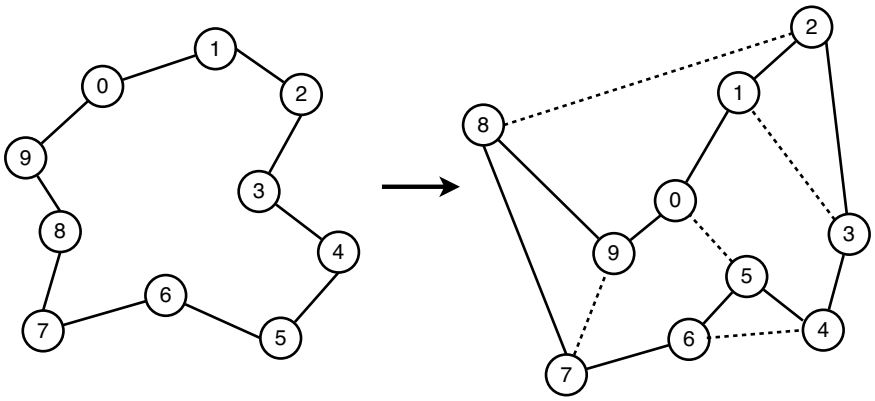

FIGURE 2.17 Extending a given input instance to a Halin graph.

tree and ring graphs. A graph is a Halin graph if it is formed by embedding a tree having no degree-2 vertices in the plane and connecting its leaves by a cycle that crosses none of its edges (Fig. 2.16). For this class of instances, a polynomial-time algorithm is known to solve the traveling salesman problem in $O(n)$ [157] and the Steiner tree problem [822]. Hence, Halin graphs may be used to construct large neighborhoods for the TSP as follows [14]: let π be the current tour. The solution π_H is a Halin extension of π if π_H is a Halin graph and π is a subgraph of π_H (Fig. 2.17). Suppose $HalinConstruct(\pi)$ is an efficient function that returns a Halin extension of the solution π. Then, the neighborhood $N(\pi)$ is constructed as $\{\pi' : \pi'$ is a tour in HalinConstruct$(\pi)\}$. Hence, finding the best tour in $N(\pi)$ is reduced to find the best tour in the solution $HalinConstruct(\pi)$.

2.1.3 Initial Solution

Two main strategies are used to generate the initial solution: a *random* and a *greedy* approach. There is always a trade-off between the use of random and greedy initial solutions in terms of the quality of solutions and the computational time. The best answer to this trade-off will depend mainly on the efficiency and effectiveness of the random and greedy algorithms at hand, and the S-metaheuristic properties. For instance, the larger is the neighborhood, the less is the sensitivity of the initial solution to the performance of the S-metaheuristics.

Generating a random initial solution is a quick operation, but the metaheuristic may take much larger number of iterations to converge. To speed up the search, a greedy heuristic may be used. Indeed, in most of the cases, greedy algorithms have a reduced polynomial-time complexity. Using greedy heuristics often leads to better quality local optima. Hence, the S-metaheuristic will require, in general, less iterations to converge toward a local optimum. Some approximation greedy algorithms may also be used to obtain a bound guarantee for the final solution. However, it does not mean that using better solutions as initial solutions will always lead to better local optima.

Example 2.7 Greedy initial solutions are not always better. The Clark–Wright greedy heuristic is a well-known efficient heuristic for the TSP and other routing problems, such as the vehicle routing problem. The solutions obtained by this greedy heuristic are much better in quality than random solutions. However, using a local search algorithm based on the 3-opt move operator, experiments show that using the initial solutions generated by this greedy heuristic leads to worse solutions than using random initial solutions in solving the TSP problem [421]. Moreover, the Clark–Wright heuristic may be also time consuming in solving large instances of the TSP problem. For this example, the random strategy dominates the greedy one in both quality and search time!

The random strategy may generate a high deviation in terms of the obtained solutions. To improve the robustness, a hybrid strategy may be used. It consists in combining both approaches: random and greedy. For instance, a pool of initial solutions can be composed of greedy solutions and randomly generated solutions. Moreover, different greedy strategies may be used in the initialization process. In fact, for some optimization problems, many greedy constructive algorithms exist and are easy to design and implement.

Example 2.8 A hybrid approach for the TSP. Given an instance with n cities, the following three algorithms can be used to generate a pool of initial solutions:

- An algorithm that generates a random permutation with uniform probability.
- A nearest-neighbor greedy algorithm that chooses a starting city i randomly. Then, among unsequenced cities, it chooses a city j that minimizes the length between i and j and iterates the same process on the city j until a complete tour is constructed. Using different starting cities allows to generate different final solutions. Hence, n different solutions may be obtained with this greedy procedure.
- Another greedy procedure, the cheapest insertion algorithm, first finds the minimum length path (i, j). Let C be the tour (i, j, i). Then, it chooses an unsequenced city that minimizes the length of the tour C and inserts it into C in the best possible way. This process is iterated until the tour C is composed of all the cities.

In some constrained optimization problems, it may be difficult to generate random solutions that are feasible. In this case, greedy algorithms are an alternative to generate feasible initial solutions.

For some specific real-life operational problems, the initial solution may be initialized (partially or completely) according to expertise or defined as the already implemented solution.

2.1.4 Incremental Evaluation of the Neighborhood

Often, the evaluation of the objective function is the most expensive part of a local search algorithm and more generally for any metaheuristic. A naive exploration of

the neighborhood of a solution s is a *complete* evaluation of the objective function for every candidate neighbor s' of $N(s)$.

A more efficient way to evaluate the set of candidates is the *evaluation* $\Delta(s, m)$ of the objective function when it is possible to compute, where s is the current solution and m is the applied move. This is an important issue in terms of efficiency and must be taken into account in the design of an S-metaheuristic. It consists in evaluating only the transformation $\Delta(s, m)$ applied to a solution s rather than the complete evaluation of the neighbor solution $f(s') = f(s \oplus m)$. The definition of such an incremental evaluation and its complexity depends on the neighborhood used over the target optimization problem. It is a straightforward task for some problems and neighborhoods but may be very difficult for other problems and/or neighborhood structures.

Example 2.9 Incremental evaluation of the objective function. First, let us present an incremental evaluation for the 2-opt operator applied to the TSP. The incremental evaluation can be stated as follows:

$$\Delta f = c(\pi_i, \pi_j) + c(\pi_{i+1}, \pi_{j+1}) - c(\pi_i, \pi_{i+1}) - c(\pi_j, \pi_{j+1})$$

Let us consider the clique partitioning problem defined in Example 2.22. For any class C of the current solution s, $W(i, \emptyset) = 0$ and for $C \neq \emptyset$, $W(i, C) = \sum_{j \in C, j \neq i} w(i, j)$. The incremental evaluation of the objective function f in moving a node i from its class C_i to another one C will be $W(i, C) - W(i, C_i)$. Then, for an improving neighbor, we have $W(i, C) < W(i, C_i)$. The complexity of finding the best class C^* for a node i (i.e., $W(i, C^*)$ is minimum over all classes $C \neq C_i$) is $O(n)$.

The incremental evaluation function may also be approximated instead of having an exact value (e.g., *surrogate functions* [777]). This will reduce the computational complexity of the evaluation, but a less accuracy is achieved. This approach may be beneficial for very expensive incremental functions.