1

Introduction

In many different fields, including management science, computer science, and electrical and industrial engineering, we are confronted with combinatorial optimization problems. In such problems we are given a finite or countably infinite set of solutions from which we have to find one that minimizes or maximizes a given cost function. One of the best studied combinatorial optimization problems is the TRAVELING SALESMAN PROBLEM (TSP), which can be formulated as follows. Consider a salesman who wants to visit each city from a set of n cities exactly once and who wants to end up in the same city in which he started, where the distance between any two cities is given. In what order should the salesman visit the cities, such that the traveled distance is minimal? To see that this problem is indeed a combinatorial optimization problem, observe that it corresponds to picking a tour with minimum length from the finite set of all possible tours visiting the n cities.

As a representation of a tour we can use a permutation of the cities that is given by $\tau = (\tau(1), \tau(2), \ldots, \tau(n))$, where $\tau(i)$ denotes the ith city visited in the tour. As the number of tours, which equals the number of permutations, is factorial in the number of cities, it grows exponentially with the number of cities. More specifically, for one hundred cities the number of tours exceeds 10^{150}, which is larger than the estimated number of particles in the universe. So the solution space is extremely large, and the search for the optimum tour by evaluating them all is impracticable.

The formulation of TSP is characteristic of combinatorial optimization problems. Such problems, more specifically their solution space, can typically be formulated in terms of discrete structures, such as sequences, permutations, graphs, and partitions.

This use of discrete structures and the use of sophisticated algorithms that work on these structures, positions combinatorial optimization, which is the discipline that deals with combinatorial optimization problems, at the intersection of two well-developed scientific fields: discrete mathematics and computer science.

A major achievement in combinatorial optimization is the development of computational complexity theory. This theory formalizes the difference between easy and hard problems. A problem is called easy if it can be solved by a polynomial-time algorithm, where we say that an algorithm solves a problem if it always returns an optimal solution. A problem is called hard, formally referred to as NP-hard, if it is commonly believed that a polynomial-time algorithm that solves it does not exist. Many interesting combinatorial optimization problems, including TSP, have this property. We assume the reader to be acquainted with the basics of complexity theory. If not, the reader is referred to the standard work of Garey & Johnson [1979]. An overview of the discipline is given in Appendix B.

When confronted with an NP-hard combinatorial optimization problem, we have two options for tackling it. The first option is to aim for an optimal solution, despite the NP-hardness of the problem. One reason why this can still be feasible is that the instances at hand may have some special structure that makes the special case easy to solve. Another reason is that having an exponential running time does not necessarily mean that an algorithm based on enumeration, such as branch-and-bound, cannot be useful. It can, for instance, be adequate if one is interested in solving relatively small problem instances. Furthermore, the NP-hardness of a problem only indicates that (most probably) no algorithm exists with a *worst-case* polynomial running time. The average-case running time, however, can be much better than the worst-case running time.

A second option for tackling an NP-hard combinatorial optimization problem is to use a heuristic algorithm. Solutions found by such an algorithm are not necessarily optimal, but they are found within an acceptable amount of time. Hence, heuristic algorithms trade off optimality against computing time.

Heuristic algorithms can be classified into two categories: constructive algorithms and local search algorithms. A constructive algorithm generates a solution through a number of steps, where in each step the partial solution obtained so far is extended until in the last step a complete solution is obtained. The order in which the steps are carried out and the actions performed in each step are often strongly problem dependent. Local search algorithms, on the other hand, try to find high-quality solutions by searching through the solution space. More precisely, these algorithms start with an initial solution and then iteratively generate a new solution that is 'near' to the current solution. A neighborhood function defines for a given solution the solutions that are near to it. As mentioned above, solutions of many combinatorial optimization problems can be represented by discrete structures such as sequences, permutations, graphs, and partitions. Local search algorithms typically use these representations by defining neighborhood functions in terms of local rearrangements, such as swapping, moving, and/or replacing items, that may be applied to a representation to obtain a neighboring solution.

In this book, we focus on local search algorithms. In practice, excellent results have been obtained by using local search algorithms for a wide variety of problems. This has led to a growing interest in theoretical results concerning the approach in the past two decades. However, many problems are still open as a challenge for the interested reader. The material presented in this book can be subdivided into two main categories: on the one hand, theory and results that are problem independent, and on the other hand more dedicated results for several classical combinatorial optimization problems. Besides having their own merits, we hope the latter results help the reader to derive similar results for other problems.

We remark that this book should not be considered as a handbook on how to apply local search in practice. For instance, we do not give any specific details on how to implement various local search metaheuristics, such as simulated annealing and tabu search. The aim of this book is to give a better understanding of the fundamental behavior of local search by proving theoretical results.

1.1 Basics of Local Search

As mentioned, this book deals with combinatorial optimization problems. These problems can be formalized as follows.

Definition 1.1. An *instance* of a combinatorial optimization problem is a pair (S, f), where the solution space S is a finite or countably infinite set of solutions and the cost function f is a mapping $f : S \to \mathbb{R}$ that assigns a real value to each solution in S called the cost of the solution. □

Definition 1.2. A *combinatorial optimization problem* Π is specified by a set of problem instances and it is either a minimization or a maximization problem. The problem is to find for a given instance (S, f) a solution $s^* \in S$ that is *globally optimal*. For a minimization problem, this means that $f(s^*) \leq f(s)$ has to hold for all $s \in S$, and for a maximization problem it means that $f(s^*) \geq f(s)$ has to hold for all $s \in S$. If no confusion can arise, a globally optimal solution is simply called optimal. □

In this definition the adjective 'combinatorial' refers to the constraint that S has to be finite or countably infinite. If this constraint does not hold, then the problem is called an optimization problem. In a problem instance (S, f) of TSP, set S consists of all possible tours over a given number of cities and f defines for each tour in S its length. We note that a problem instance (S, f) is generally defined implicitly by using a compact data representation and not by giving the complete solution set S and the cost of each solution. A problem instance of TSP can, for instance, be defined by giving for each pair i, j of cities the distance from city i to j. The size of the representation, expressed as the number of bits required for storing it, is called the size of the problem instance. An algorithm is called a polynomial-time algorithm if its running time is polynomial in the size of a problem instance. Otherwise, it is called exponential.

We consider local search algorithms for tackling combinatorial optimization problems. The key feature of these algorithms is their neighborhood function. This

function specifies, for each solution, which solutions are in some sense near to it. The neighborhood function is generally defined in terms of small changes that may be applied to a solution to obtain a neighboring solution.

Definition 1.3. For an instance (S, f) of a combinatorial optimization problem, a *neighborhood function* is a mapping $N : S \to 2^S$, where 2^S denotes the powerset $\{V|V \subseteq S\}$. The neighborhood function specifies for each solution $s \in S$ a set $N(s) \subseteq S$, which is called the *neighborhood* of s. The cardinality of $N(s)$ is called the *neighborhood size* of s. We say that solution s' is a *neighbor* of s if $s' \in N(s)$. The neighborhood function N is said to be *symmetric* in the case that we have $s' \in N(s)$ if and only if $s \in N(s')$. □

Throughout this section, we let S and f define a problem instance of an arbitrary combinatorial optimization problem and we let N be an arbitrary neighborhood function for S. Furthermore, without loss of generality we only consider minimization problems. A maximization problem can be transformed into a minimization problem by reversing the sign of cost function f.

A local search algorithm starts with an initial solution that is constructed by some heuristic algorithm. Next, the local search algorithm searches through the solution space by continually moving from a solution to one of its neighbors. This process can be modeled as a walk through the neighborhood graph.

Definition 1.4. The *neighborhood graph* of an instance (S, f) of a combinatorial optimization problem and an accompanying neighborhood function N is a directed node-weighted graph $G = (V, E)$. The node set V is given by the set S of solutions, and the arc set E is defined such that $(i, j) \in E$ if and only if $j \in N(i)$. Furthermore, we define the weight of a node as the cost of the corresponding solution. If the neighborhood function is symmetric, then the directed graph can be simplified to an undirected graph by replacing arcs (i, j) and (j, i) by a single edge $\{i, j\}$. □

We note that the neighborhood functions used in practice are almost always symmetric. We assume the reader to be familiar with basic graph theory. An overview of the basic graph theory concepts used in this book is presented in Appendix A.

Definition 1.5. A solution j is *reachable* from solution i if the neighborhood graph G contains a path from i to j. This means that a sequence of solutions $s_1, s_2, \ldots, s_k$ exists with $k \geq 1$ such that $s_1 = i$, $s_k = j$, and $s_{l+1} \in N(s_l)$ with $1 \leq l < k$. □

Definition 1.6. A neighborhood graph is *strongly connected* if, for each pair i, j of solutions, j is reachable from i. A neighborhood graph is *weakly optimally connected* if, for each solution i, it contains a path from i to an optimal solution. □

Different strategies have been proposed for walking through a neighborhood graph. The most obvious strategy is used by the *iterative improvement algorithm*, also known as the *hill climbing algorithm*. This is the basic local search algorithm. In each iteration, the algorithm searches in the neighborhood of the current solution for a solution with better cost. If such a solution is found, it replaces the current

solution. Otherwise, the algorithm stops and returns the current locally optimal solution.

Definition 1.7. A solution $\hat{s} \in S$ is called *locally optimal* with respect to N or N-*optimal* if $f(\hat{s}) \leq f(s)$ for all $s \in N(\hat{s})$. □

In this book, a star as superscript indicates global optimality and a hat indicates local optimality. Hence, s^* is a globally optimal solution, whereas $\hat{s}$ is locally, but not necessarily globally optimal.

We note that if we have two different neighborhood functions N_1 and N_2 for the same problem instance and if N_1 is dominated by N_2, as defined below, then each local (global) optimum for N_2 is also a local (global) optimum for N_1.

Definition 1.8. Let N_1 and N_2 be two different neighborhood functions for the same instance (S, f) of a combinatorial optimization problem. If for all solutions $s \in S$ we have $N_1(s) \subseteq N_2(s)$, then we say that N_2 *dominates* N_1. □

Instead of being modeled as a walk through a neighborhood graph, an execution of iterative improvement can be modeled more precisely as a walk through a transition graph.

Definition 1.9. The *transition graph* of an instance of a combinatorial optimization problem and an accompanying neighborhood function is a directed, acyclic subgraph of their neighborhood graph G. It is obtained from G by deleting all arcs (i, j) for which it holds that the cost of solution j is worse than or equal to the cost of solution i. □

Note that a solution is locally optimal if and only if it has outdegree zero in the transition graph. The rule used by iterative improvement to select a neighboring solution in the case that a solution has multiple neighbors with a better cost is called the *pivoting rule*. Well-known pivoting rules are *first improvement* and *best improvement*. In first improvement we generate neighbors and we accept the first solution encountered with better cost, where the neighbors can be generated randomly or in some specified order. When using best improvement, we replace s by a best solution in its neighborhood, provided that it has better cost. Figure 1.1 gives the iterative improvement algorithm in pseudo-code. If the neighborhood function is exact, then the algorithm is guaranteed to give a globally optimal solution.

Definition 1.10. A neighborhood function is called *exact* if each local optimum is also a global optimum. □

Example 1.1. To illustrate the definitions given above, consider the problem instance (S, f) and the (symmetric) neighborhood function N defined in Table 1.1. Figure 1.2 depicts the corresponding neighborhood graph. The figure also shows the corresponding transition graph, which can be derived from the neighborhood graph. The problem instance has two global optima, namely solutions 3 and 6. Furthermore, solution 1 is a local optimum. This implies that the neighborhood function

algorithm Iterative Improvement

begin
  $s :=$ some initial solution;
  repeat
    generate an $s' \in N(s)$;
    if $f(s') < f(s)$ **then** $s := s'$;
  until $f(s') \geq f(s)$ for all $s' \in N(s)$;
end;

Figure 1.1. Iterative improvement algorithm for a minimization problem. The first-improvement pivoting rule generates neighbors from $N(s)$ at random or in some specified order. The best-improvement pivoting rule generates a neighbor from $N(s)$ with lowest cost.

we consider is not exact. To make it exact, it suffices to change the neighborhood function such that solutions 1 and 3 are neighbors of each other.

The neighborhood graph of Figure 1.2 contains two disconnected components: the subgraph G_1 induced by solutions 1, 2, 3, and 4 and the subgraph G_2 induced by solutions 5, 6, and 7. Solutions from G_1 are not reachable from solutions from G_2 and vice versa. Hence, the neighborhood graph of Figure 1.2 is not strongly connected. However, as G_1 and G_2 both contain a global optimum it is weakly optimally connected. □

One often uses the metaphor of walking in a mountainous region to get an intuitive idea of local search. Consider an instance of a minimization problem. If we define the height of a solution as its cost, then walking through a neighborhood graph to find a global optimum can be seen as walking on a three-dimensional surface to find the lowest point. This looks like walking in a dense fog over the surface of the earth with its mountains and valleys, where we are looking for the lowest point and we cannot see farther than one step ahead. Furthermore, a local optimum corresponds to the lowest point in a valley and applying iterative improvement means that we only allow downhill moves.

A major drawback of applying iterative improvement is that it may get trapped in poor local optima. One way to tackle this problem is to find a better, possibly more complex, neighborhood function. Alternatively, one can allow non-improving moves or perform multiple runs of iterative improvement. The latter two approaches are frequently used in practice and often with success. The most popular local search algorithms based on them are discussed in Chapter 7.

The performance of iterative improvement is determined not only by the quality of the local optima, but also by the time it takes to reach a local optimum. The time required to reach a local optimum is determined by two aspects: the time needed for moving from one solution to the next and the number of solutions that are visited before one arrives at a local optimum. To keep the first part small, it is important

Table 1.1. Problem instance (S, f) with $S = \{1, 2, \ldots, 7\}$ and neighborhood function N discussed in Example 1.1.

solution	$f(s)$	$N(s)$
1	2	$\{2\}$
2	3	$\{1,3\}$
3	1	$\{2,4\}$
4	2	$\{3\}$

solution	$f(s)$	$N(s)$
5	2	$\{6\}$
6	1	$\{5,7\}$
7	2	$\{6\}$

that the outdegree of the nodes in the neighborhood graph is not too large, i.e., the cardinality of the neighborhoods of solutions may not be too large. Consider, for instance, the extreme case in which the neighborhood graph is a complete graph. This means that for each $i, j \in S$, we have $i \in N(j)$. We then have high-quality local optima because the neighborhood function is exact. However, testing for local optimality and finding a better neighbor in the case that a solution is not locally optimal comes down to solving the original problem.

To keep the number of iterations to reach a local optimum small, it is important that the transition graph has a small 'potential'.

Definition 1.11. Let T be a transition graph, and let $\hat{V}$ be the nodes in T with outdegree zero, i.e., the nodes that correspond to local optima. The *potential* of a node v is defined as the minimum distance of v to a node in $\hat{V}$, where the distance from node i to j is defined as the minimum number of arcs in a connecting path from i to j. The potential of transition graph T is the maximum potential over all its nodes. □

The potential of a transition graph gives a lower bound on the number of iterations that are maximally required by iterative improvement if it can select an arbitrary solution as the starting solution. The lower bound is tight whenever an optimal pivoting rule is used, where optimal means that a shortest path to a locally optimal solution is chosen.

As mentioned, local search algorithms that admit non-improving moves are often used in practice to overcome the problem of ending up in poor local optima. In the remainder of this section, we consider such algorithms and discuss some related issues. First, we observe that they can still result in low-quality solutions if a neighborhood graph consists of several disconnected components, i.e., if the solution space can be partitioned into subsets $V_1, V_2, \ldots, V_m$, such that from solution $i \in V_j$ only other solutions from V_j can be reached. If in that case we start with a solution from a subset V_j with only low-quality solutions, then the local search algorithm still has a poor performance. Obviously, this problem does not occur if we require the neighborhood graph to be strongly connected. However, when we start with an initial solution i, we are actually not interested in whether all solu-

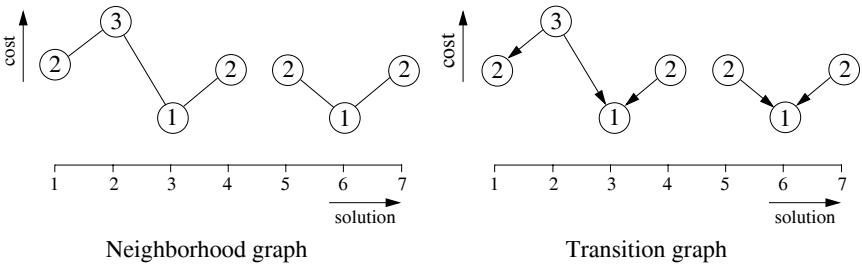

Figure 1.2. Neighborhood graph and transition graph of the problem instance and neighborhood function defined in Table 1.1. The number written inside a node defines its weight, i.e., the cost of the solution it represents.

tions are reachable. We only want to know whether we can reach a global optimum. Hence, the condition that the neighborhood graph must be strongly connected can be weakened to that it must be weakly optimally connected.

We do not only want that a global optimum is reachable. We also want it to be reachable within a reasonable number of steps. This is realized in a strongly connected neighborhood graph if its diameter is not too large.

Definition 1.12. The *diameter* of a neighborhood graph is the maximum distance between any pair of nodes, where the distance is as defined in Definition 1.11. If the neighborhood graph is not strongly connected, then its diameter is defined to be infinitely large. □

Although the performance of a local search algorithm generally improves by allowing deteriorating moves, it should not be too eager to carry out these moves in order to prevent it from carrying out a random search. As a result, the difficulty in reaching a solution j from a solution i via a path p strongly depends on the height of the path.

Definition 1.13. Let $p = (s_1, s_2, \ldots, s_k)$ be a path in a neighborhood graph, where s_i is the ith node in the path. Then the *height* of path p is given by the maximum label of any of its nodes, i.e., by $\max_{1 \leq i \leq k} f(s_i)$. □

Using this definition, we can define the depth of a local optimum to express how hard it is to escape it.

Definition 1.14. Let $\hat{s} \in S$ be a local optimum. The *depth* of $\hat{s}$ is defined as the minimum height of a path p from $\hat{s}$ to a solution s with $f(s) < f(\hat{s})$. If such a solution s does not exist, then the depth of $\hat{s}$ is ∞. □

To obtain an effective local search algorithm, it is important that the solution space does not contain large *plateaus*, where a plateau refers to a part of the solution space in which all solutions have (approximately) the same value. On a plateau it is difficult for a local search algorithm to guide itself to better solutions as all

directions look the same. Another problem arising on plateaus is *cycling*. Cycling means that the algorithm revisits solutions over and over again. If, for instance, we have a plateau containing two local optima, s and s', that are neighbors of each other, have the same cost, and are surrounded by solutions with very high cost, then a local search algorithm will tend to alternate between s and s'. To avoid cycling a local search algorithm can remember the last visited solutions or fingerprints of these solutions.

1.2 Outline of the Book

The remainder of this book is organized as follows. In the next chapter we present some classical combinatorial optimization problems with accompanying neighborhood functions. Besides illustrating how the use of different solution representations may give rise to different neighborhood functions, these are the problems and neighborhood functions for which we prove problem-dependent results in this book.

Neighborhood function can be based on representations that directly define a solution and on representations that indirectly define a solution. In Chapter 3 we show the use of indirect representations. In Chapter 4 we derive properties of neighborhood functions, such as the diameter and connectivity of their corresponding neighborhood graph. Performance guarantees for neighborhood functions, i.e., for their local optima, are proved in Chapter 5. In this chapter we also prove the negative result that if it is NP-hard to find an R-approximate solution for some problem, then the problem does not admit an efficient neighborhood function with a performance bound of R. While Chapter 5 relates to the quality of local optima, Chapter 6 is concerned with the time complexity of finding a local optimum. This includes a formal theory that is similar to the theory of NP-completeness.

Chapter 7 elaborates on several different techniques to overcome the problem that iterative improvement stops at the first local optimum it encounters. One of the techniques presented is simulated annealing, for which we prove in Chapter 8 that, under some mild conditions, it converges asymptotically to the set of globally optimal solutions.

1.3 Bibliographical Notes

Classical textbooks on combinatorial optimization are Cook, Cunningham, Pulleyblank & Schrijver [1998], Korte & Vygen [2002], Lawler [1976], Nemhauser & Wolsey [1988], and Papadimitriou & Steiglitz [1982]. All these textbook have been written for a more advanced audience. Foulds [1984] presents a more introductory textbook written at the undergraduate level. Beale [1988] presents a more general introduction to optimization with a chapter on combinatorial optimization. Recently, Schrijver [2003] published a three-volume textbook that provides a quite elaborate and in-depth presentation of the field with a wealth of references to the existing literature. Annotated bibliographies on combinatorial optimization are given by O'Heigeartaigh, Lenstra & Rinnooy Kan [1985] and Dell'Amico, Maffioli & Martello [1997].

Aarts & Lenstra [1997] present an overview of both theoretical and practical aspects of local search. Hoos & Stützle [2004] present a book that serves as a practical guide to the application of local search algorithms and its many variants. A gentle introduction to local search is given by Michalewicz & Fogel [2000]. Overviews on local search algorithms are given by Glover & Kochenberger [2003], Reeves [1993], and Ribeiro & Hansen [2002]. An annotated bibliography on local search is given by Aarts & Verhoeven [1997].

1.4 Exercises

1. The set $A = \{1,3,6,9\}$ of integers is given. We define the cost of a sequence τ of the numbers in A as $f(\tau) = \sum_{i=1}^{4} i\tau(i)$, where each number of A occurs exactly once in τ and where $\tau(i)$ denotes the ith integer in τ.
   We consider the problem of finding a sequence τ over A that minimizes $f(\tau)$. This problem is equivalent to finding a descending ordering of the numbers in A.

   a) Give the solution space of the described problem instance.

   b) Give the cost of each solution/sequence.

   Consider the swap neighborhood function in which sequence τ' is a neighbor of sequence τ if and only if τ' can be obtained from τ by changing the order of two adjacent numbers.

   c) Give the neighborhood graph for the swap neighborhood function.

   d) Give the transition graph for the swap neighborhood function.

   e) Verify that the swap neighborhood function is exact.

   f) Determine the depth of the local optima.

2. Consider the iterative improvement algorithm described in Figure 1.1. We apply this algorithm to the problem instance of Exercise 1, where an initial solution is selected uniformly at random. Show that applying the best-improvement pivoting rule results in a smaller expected number of iterations than applying the first-improvement pivoting rule, where the first-improvement pivoting rule generates neighbors uniformly at random.

3. We defined iterative improvement such that a neighbor is only accepted if it has strictly better cost. Give a reason to not accept a neighbor that has equal cost.

http://www.springer.com/978-3-540-35853-4

Theoretical Aspects of Local Search
Michiels, W.; Aarts, E.; Korst, J.
2007, VIII, 238 p. 70 illus., Hardcover
ISBN: 978-3-540-35853-4