Ministry of Education and Science of the Russian Federation
Novosibirsk State University
Department of Mechanics and Mathematics
Chair of Theoretical Cybernetics

Adil I. Erzin, Yury A. Kochetov

# ROUTING  PROBLEMS

A textbook

Novosibirsk
2015

UDK 519.8
BBK V 173
E 709

The textbook was prepared within the frame of the Novosibirsk State University Development Program (2009–2018).

**E-709 Erzin, A. I.**
Routing problems: a textbook / A. I. Erzin, Yu. A. Kochetov; Novosibirsk State University. – Novosibirsk: Editorial and Publishing Center of NSU, 2015. – 90p.

ISBN 978-5-4437-0409-8

This textbook introduces a reader to some chapters of the course "Operations Research" which is read by the authors on the different streams of Mechanics and Mathematics Department of Novosibirsk State University. The textbook contains the necessary definitions, statements, algorithms, examples and exercises. The manual includes sections on the synthesis of spanning trees and Steiner trees, on the construction of optimal paths and circuits, as well as the methods for solving the traveling salesman problem.

It is designed for the students of Mechanics and Mathematics Faculty of NSU, and for anyone who wants to master the course on their own as well.

UDK 519.8
BBK V 173

ISBN 978-5-4437-0409-8

# CONTENTS

# Preface

In 2000, professor Toshiyuki Nakagaki, a biologist and a physicist at the University of Hokkaido, Japan, took a sample of yellow fungus mold Physarum polycephalum and put it at the entrance to the labyrinth, which is used to test the intelligence and memory of mice. At the other end of the labyrinth, he placed a sugar cube.

Physarum polycephalum smelled sugar and began to send their shoots on its quest. Cobweb fungus bifurcate at each intersection of the labyrinth, and those who fell into a dead end turns around and starts to look in other directions. Within a few hours of mushroom cobweb filled labyrinth of passages and by the end of the day one of them found its way to sugar.

After that, Toshiyuki and his team of researchers took a piece of gossamer fungus, which participated in the first experiment, and laid it at the door of copies of the same maze, also with a cube of sugar on the other end. What happened surprised everyone. Gossamer was divided into two: one process has paved its way to the sugar, without any extra turn, the other – climbed the wall of the maze and crossed it on the ceiling directly to the target. Mushroom gossamer not only memorized the road, but also changed the rules of the game.

Further studies Toshiyuki found that mushrooms can plan transportation routes are not worse and faster than engineering professionals. Toshiyuki took the map of Japan and put the pieces of food in locations corresponding to the major cities of the country. Mushrooms he put "on Tokyo". After 23 hours, they are building a network of spider webs all pieces of food. The result is an almost exact copy of the railway network in the Tokyo area.

"Not too difficult to connect several dozen points but put them together effectively and economically the most – it's not easy. I believe that our research will not only help to understand how to improve the infrastructure, but also how to build more effective information network." – Toshiyuki Nakagaki.

This textbook is devoted to the study of algorithms that allow to builds effectively the best routes – the sub-graphs of a given graph that have given extreme properties. The spanning trees, the shortest paths in graphs, the communication networks, and Hamiltonian cycles will be in the tutorial in the spotlight. Along with the classical algorithms of discrete optimization, we'll become acquainted with new approaches to the construction of algorithms borrowed from nature: a genetic algorithm, a simulated annealing algorithm, etc.

The book consists of five chapters. The first one is devoted to the problems of construction of the spanning trees in a given weighted graph. The second chapter is a collection of problems associated with the construction of the shortest paths. The third chapter is devoted to the Steiner's tree problem, the analysis of its complexity and the description of the approximation algorithms to solve it. In the 4th chapter the traveling salesman problem is considered.

Further we will use the following notation.

$Z_+$ – set of non-negative integers;

$Z_+^n$ – set of $n$-dimensional non-negative integer vectors;

$N = \{1, 2, \ldots, n\} \subset Z_+$;

$R$ – set of real numbers;

$R_+ = \{x \in R, x \geq 0\}$;

$R^n$ – set of $n$-dimensional real vectors $x = (x_1, \ldots, x_n)$, $x_j \in R$;

$R_+^n = \{x \in R^n, x_j \geq 0, j = 1, \ldots, n\}$;

$B^n$ – set of $n$-dimensional Boolean vectors $x = (x_1, \ldots, x_n)$, $x_j \in \{0, 1\}$;

$|M|$ – cardinality of the set $M$;

$\lfloor a \rfloor$ – integral part of $a$;

$\lceil a \rceil$ – the smallest integer that is not less than $a$;

$a^+ = \max\{0, a\}$;

$x^* = (x_1^*, \ldots, x_n^*)$ – optimal solution;

□ – end of the proof.

For the reader's convenience, we recall the definitions of some basic concepts that will be used further.

**Definition 1.** A *graph* $G = (V, E)$ is a pair of sets $V$ and $E$, where $V = \{1, \ldots, n\}$ is the set of vertices and $E = \{(i, j) \mid i, j \in V\}$ is the set of edges of the graph. To display the node, usually the point as an image edge – the line segment between two points will be used. Ordered pair of vertices defines the *arc* and is depicted by an arrow, which ends at the end node of the arc.

**Definition 2.** Two vertices which are connected by an edge are *adjacent*. If the node is one of the ends of the edge, then the vertex and the edge are *incident* to each other. A vertex's *degree* is a number of edges incident to it. The *degree of the graph* coincides with the maximum node's degree.

**Definition 3.** A *simple chain* is an acyclic connected sub-graph of degree 2. Directed chain in which all edges are oriented from the beginning to the end of the chain, called the *path*.

**Definition 4.** Connected sub-graph with all vertices of degree 2 is called a cycle. The cycle that includes all vertices of the graph is a *Hamiltonian* cycle.

**Definition 5.** Graph, each edge of which $(i, j) \in E$ is attributed to the number – the "weight", is called *weighted*. Sum of weights of the edges belonging to the sub-graph is called the *weight* of the sub-graph.

**Definition 6.** The number of characters in the standard (binary) data coding of instance problem $X \in P$ is called the *length* of the input and denoted as $L = L(X)$.

**Definition 7.** Let algorithm $A$ solves a problem $P$ and $t_A(X)$ is the number of elementary operations (arithmetic and comparison operations) performed by an algorithm $A$ in solving the instance $X \in P$. Then the function

$$T_A(n) = \sup_{X \in P}\{t_A(X) : L(X) = n\}$$

is a *complexity* of algorithm $A$.

A *polynomial* algorithm is one with complexity equals $T_A(n) = O(n^d)$, where $d$ is a positive integer.

Algorithm which complexity is not limited by a polynomial of the length of the input is called *exponential*.

In the computational complexity theory a decision problem, that is problems in which the possible answer "Yes" or "No" are considered. For example, is it true that a given graph is a tree? Among the decision problems it is customary to distinguish the classes P and NP. Recall that the class P consists of decision problems that are solvable in polynomial time. Class NP is more inclusive. It includes all of the decision problems, to which the answer "Yes" can be tested in polynomial time. The problem belongs to this class if, without even knowing how to solve it, one can "easily" check the answer. It is sufficient to be able to check only the answer "Yes". Sometimes checking the answer "Yes" may be easier or more difficult than checking the answer "No". Consider the Hamiltonian graph: given a simple undirected graph, and we wants to find out whether it contains a Hamiltonian cycle? This problem belongs to the class NP. Really, suppose that the graph is Hamiltonian, and someone suggested the answer by clicking one of these cycles. Can we try this tip in polynomial time? To do this, check that the specified set of edges forms a simple cycle, and it covers all the vertices. Obviously, this is "easy" to do, and therefore, the problem belongs to the class NP. Note that the answer "No" is much more difficult to check in for this problem.

It is said that the decision problem belongs to co-NP, if the answer "No" can be checked in polynomial time. It is easy to show that, the following problem belongs to co-NP. Asked a simple undirected graph, is it true that it is not a Hamiltonian? "No" means an existence of Hamiltonian cycle, and it can be easily checked.

It's evident that P $\subseteq$ NP. To prove or disprove the reverse inclusion no one can so far. To date, this is one of the central problems of mathematics – "the problem of the Millennium". For its decision the American Mathematical Society promises a prize – a million dollars.

Many years of intensive studies suggest that P $\neq$ NP. An indirect proof of this hypothesis is the fact that in the class of NP found the so-called NP-complete problems.

**Definition 8.** The problem in the class NP is called *NP-complete* if the existence of a polynomial algorithm for its solution implies the existence of polynomial algorithms for all the problems in the class NP.

By now we know a lot of NP-complete problems [4, 14]. However, for none of them an exact polynomial-time algorithm has been developed.

**Definition 9.** *Approximation* algorithm for the problem $P$ is an algorithm for constructing a *feasible* solution. Estimates of the accuracy of the algorithm $A$ is said to be a number $\varepsilon > 0$, such that the ratio $W_A(I)/W^*(I)$ does not exceed $\varepsilon$ for any instance $I \in P$ for minimization problem, and at least $\varepsilon$ for any instance $I \in P$ for the maximization problem. Here

$W^*(I)$ is the value of the objective of the optimal solution to the instance *I*, and $W_A(I)$ is the value of the functional of the solution yielded by algorithm *A*. In this case, the algorithm *A* is called *ε-approximate*.

# Chapter 1. Spanning trees

Given a simple undirected connected graph $G = (V, E)$ with vertex set $V = \{1, \dots, n\}$ and edge set $E$, $|E| = m$.

**Definition 1.1.** *Spanning sub-graph* of $G$ is a connected sub-graph $O = (V, E_O)$, $E_O \subseteq E$, i.e. connected sub-graph containing *all* the vertices of the graph $G$.

**Definition 1.2.** *Spanning tree* in a graph $G$ is acyclic spanning sub-graph.

Obviously, in the spanning tree, every pair of vertices is connected by a simple chain, and a spanning tree has the minimum number of edges among all spanning sub-graphs.

The number of different spanning trees of a graph is exponentially dependent on the number of vertices $n$. For example, in the complete graph, the number of different spanning trees equals $n^{n-2}$. This fact is proved by dozens of different ways [1]. If the selection criterion is difficult for formalization, or is subjective, then in order to find the "best" tree one need to iterate over all of them. The correct trees enumeration algorithm is proposed, for example, in [1].

## 1.1. Minimum spanning tree

**Definition 1.3.** A spanning tree, in which the sum of the edge's weights is minimal, is called a *minimal spanning tree* (MST).

Many practical problems can be reduced to the construction of the MST. Suppose one wants to link a given set of settlements with the road network in such a way as to minimize the associated costs. If one knows the cost of building the road between each pair of points – the weight of the corresponding edge, then by finding the MST in a complete graph whose vertices correspond to the settlements, one will solve the problem. There are several efficient (polynomial time) algorithms for finding the MST. Here are the most popular.

## Prim's algorithm

The idea of the algorithm belongs to Prim, but its effective implementation is proposed by Dijkstra [1]. The algorithm consists of $n-1$ iterations. At each iteration, to the partially constructed tree one vertex and one edge are added. First, the tree under construction $T=(V_T, E_T)$ comprises one arbitrary vertex and none of the edges. The edge to be added links the vertex $i$ in $V_T$, with the vertex $j$ not in $V_T$:

$$(i,j)=\arg\min_{(p,q):p\in V_T, q\notin V_T} c_{pq}.$$

Set $V_T=V_T\cup\{j\}$, $E_T=E_T\cup\{(i,j)\}$. If $|V_T|=n$, then stop, MST is constructed.

The effective implementation of the algorithm [1] is that each not in $V_T$ node $j$ is assigned a label $(\alpha_j, \beta_j)$, where $\alpha_j$ is the closest to $j$ node in $V_T$, and $\beta_j$ is the weight of the edge $(\alpha_j, j)$. Then, after the accession of the next edge, the label of each vertex is updated (with time complexity $O(1)$). The number of iterations is $n-1$, each iteration's complexity is $O(n)$. Therefore, the overall complexity of effective implementation of Prim's algorithm is $O(n^2)$.

## Kruskal's algorithm

The algorithm [1] starts with the trivial graph $T=(V, \varnothing)$. Order the edges in decreasing order of their weights and add the edges to $T$ in the order. Next edge is added to $T$ and is removed from the list if it does not lead to the formation of a cycle. Otherwise, it is simply removed from the list, and we consider the next edges in the list. This is repeated until the number of edges in $T$ becomes $n - 1$. The constructed tree is MST.

The complexity of sorting the edges is $O(m \log m)$. Obviously, when constructing the tree, in the worst case all $m$ edges are considered. Until MST is built, partially constructed graph is disconnected, and the added edge links vertices in the different connected components. In reviewing the edges in the ordered list, one need to avoid any cycles, i.e. do not span the nodes of one connected component. The corresponding procedure described in [1] (Sections 2.2.1 and 2.2.2), and it can be done with constant complexity. The connected components conveniently stored in the form of a system of disjoint sets. All operations performed in this case with complexity $O(m\, \alpha(m, n))$, where $\alpha$ is the inverse of Ackermann's function [2]. Since in any practical problems $\alpha(m, n) < 5$, it is possible to take it as a constant, so the overall complexity of Kruskal's algorithm is $O(m \log m)$. Therefore, this algorithm is suitable for the construction of MST in the graphs with a small amount of edges.

## Borůvka's algorithm

The algorithm was first published in 1926 by O. Borůvka as a method of finding the optimal power supply system. The algorithm consists of iterations, each of which is sequentially add one edge forming a spanning forest until one tree is built. The algorithm assumes that the weights of the edges are different or edges somehow arranged to select only one edge with minimal weight (in the case of a several edges having a minimum weight, chosen, for example, an edge with a minimal number).

First, $T = (V_T, E_T)$, $V_T = V$, $E_T = \varnothing$, is a spanning forest in which each vertex is a tree. While $|E_T| < n - 1$ do:

- For each connected component (tree) find the minimum weight edge that connects this component with some other connected component.

- Add all the found edges in the set $E_T$.

The resulting tree $T$ is a MST.

At each iteration, the number of trees in a forest reduced at least twice, so the algorithm performs $O(\log n)$ iterations. The complexity of one iteration is $O(m)$, so the overall complexity of the algorithm is $O(m \log n)$.

## 1.2. Spanning trees and applications

In practical problems it is often required to construct a spanning tree that satisfies the various additional properties. This section describes some of these problems.

Limited degree MST is a minimum spanning tree in which each node is adjacent to at most $d$ other vertices, where $d$ is a given integer. If $d = 2$, then this is a problem of constructing a minimal Hamiltonian chain, which implies that the problem of building a limited degree MST is NP-hard in the general case [4]. If the vertices of the graph are the points in the plane, and edge weights are equal to the Euclidean distance between them, and we need to build a MST with degree at most $k$, then for $k = 5$, the problem is polynomially solvable [3].

In some applications, it is necessary to construct a spanning tree in which the maximum weight of the edge is minimal. Obviously, in this case, MST is one of the desired trees.

When describing the algorithms Prima and Kruskal, we do not impose restrictions on the signs of the edge weights, so that the problem of constructing a spanning tree of maximum weight can be solved by algorithms Prim or Kruskal after multiplication the edge weights by $-1$ and the construction of a MST in the graph with the new weights.

Let us consider (as an example) the problem of building a single-source communication network of minimum cost with the restriction on the number of switching nodes in the circuits connecting the points with the source. This problem is a constructing of MST with the limited radius, which can be stated as follows.

Given a complete undirected weighted graph $G = (V, E)$, $V = \{0,1, ..., n\}$, with non-negative edge weights $a_{ij} \geq 0$. Denote as $F = F(G)$ a set of spanning trees of graph $G$, and let $C_k(T)$ is a chain connecting node $k$ with a root node 0 – a signal source in the tree $T \in F$. It is required to construct a tree $T^* \in F$, which is the solution of the problem

$$\sum_{(i,j)\in T} a_{ij} \to \min_{T\in F}; \tag{1.1}$$

$$|C_k(T)| \leq R,\ \ k = 1, \dots, n, \tag{1.2}$$

where $R \leq n$ is a given positive integer, and $|C_k(T)|$ is the number of edges in the chain $C_k(T)$. The number $\max_{k\in V}|C_k(T)|$ is a radius of the tree $T$.

The problem (1.1) - (1.2) is NP-hard when $R \geq 2$, that naturally follows from the NP-hardness of the problem of building the MST of limited diameter [4]. In [5] showed the NP-hardness of the maximization problem, which is closely related to the problem (1.1) - (1.2). Indeed, if $a_{ij} \in [a, A]$, then the problem

$$\sum_{(i,j)\in T} b_{ij} \to \max_{T\in F}$$

with the constraint (1.2), where $b_{ij} = A - a_{ij}$, is equivalent to (1.1) - (1.2).

In [5] for the maximization problem a series of polynomial algorithms for constructing the solutions with a guaranteed relative error 1/2, is proposed. If the weights of the edges are satisfy the triangle inequality, then the relative error is improved to the

$$\min\left\{\frac{2}{5}, \frac{4}{R+7}, \frac{2}{R+1}\right\}.$$

For the problem (1.1) - (1.2) a priori estimates of accuracy depends on the parameters of the problem and are not guaranteed (constants) [5].

Another example of the practical problem is selecting of radio transmission ranges of the elements in the wireless networks. This problem is known in the literature as a “Min-Power Symmetric Connectivity Problem” and it is as follows.

A simple weighted undirected graph $G = (V, E)$ with vertex set $V$, $|V| = n$, and a set of edges $E$ is given. Let $c_{ij} \geq 0$ is the weight of the edge $(i, j) \in E$. It is required to find a spanning tree $T^*$ in graph $G$, which is a solution of the problem:

$$W(T) = \sum_{i \in V} \max_{j \in N_i(T)} c_{ij} \to \min_T , \tag{1.3}$$

where $N_i(T)$ is the set of vertices adjacent to the vertex $i$ in the tree $T$. Any feasible solution to the problem (1.3) – spanning tree – also called a *communication tree*.

Problem (1.3) is strongly NP-hard even in the case when the nodes are the points in $R^2$, and the edge weight is the Euclidean distance between the corresponding points. In general, the NP-hardness follows naturally from the polynomial reducibility of the minimal covering (MC) to the problem (1.3).

For any spanning tree $T$ are just the obvious inequalities holds

$$\sum_{(i,j) \in T} c_{ij} \leq W(T) \leq 2 \sum_{(i,j) \in T} c_{ij} ,$$

which implies that the MST is a 2-approximate solution of the problem (1.3). That is

$$\frac{W_{\text{MST}}}{W^*} \leq 2 ,$$

where $W_{\text{MST}}$ is the value of the objective of the MST, and $W^*$ is the optimum of the objective function. We proved the

**Theorem 1.1.** Let the weights of the edges that are included in the MST, belong to the interval [*a*, *b*], then

$$\frac{W_{\text{MST}}}{W^*} \le 2 - \frac{2a}{a+b+2b/(n-2)};$$

and when $n \to +\infty$, the next inequality holds

$$\frac{W_{\text{MST}}}{W^*} \le \frac{2b}{a+b}.$$

**Theorem 1.2.** If the problem of constructing *R*-approximate solution of the minimal vertex cover (MVC) in a graph, which degree do not exceed *k*, is NP-hard, then the problem of building $\left(1+\frac{R-1}{k+1}\right)$-approximate solution of the problem (1.3) is also NP-hard.

**Corollary 1.1.** It's known that the problem of building $(1+1/52)$-approximate solution for the MVC when $k=4$ is NP-hard. Then from Theorem 1.2, in particular, follows the NP-hardness of construction of the $(1+1/260)$-approximate solution to the problem (1.3).

## 1.3. Examples and exercises

**Example1.1.** Build a MST in the graph shown in Fig. 1.1*a* (next to the edges are their weight), using Prim's algorithm.

Solution. Set $T = (\{1\}, \varnothing)$. Then the labels of the vertices $(\alpha_j, \beta_j)$, $\alpha_j = 1$, $j = 2, \ldots, 7$, $\beta_2 = \beta_3 = 2$, $\beta_5 = 4$, other $\beta_j = +\infty$. The nodes 2 and 3 are nearest to *T*, add, for example, vertex 2, obtain $T = (\{1, 2\}, \{(1, 2)\})$ and recalculate the labels of the nodes 3, …, 7. Then $(\alpha_3, \beta_3) = (1, 2)$,

$(\alpha_4, \beta_4) = (2, 1)$, $(\alpha_5, \beta_5) = (2, 3)$, $(\alpha_6, \beta_6) = (1, +\infty)$, $(\alpha_7, \beta_7) = (1, +\infty)$. Now the closest to $T$ the node 4, add it: $T = (\{1, 2, 4\}, \{(1, 2), (2, 4)\})$. Continuing the process, add successively vertices 5, 3, 6 and 7 together with the edges (4, 5), (1, 3), (5, 7) and (3, 6). MST is shown in Fig. 1.1*b* and its weight is 14.

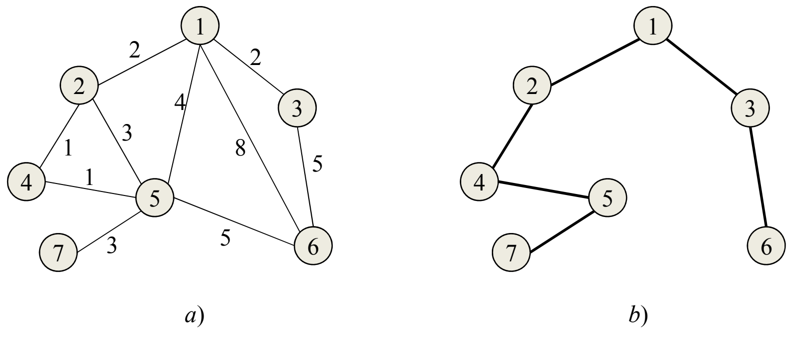


Fig. 1.1. *a*) Graph; *b*) MST.

**Example 1.2.** Find the number of spanning trees of the graph shown in Fig. 1.1*a* of degree at most 2.

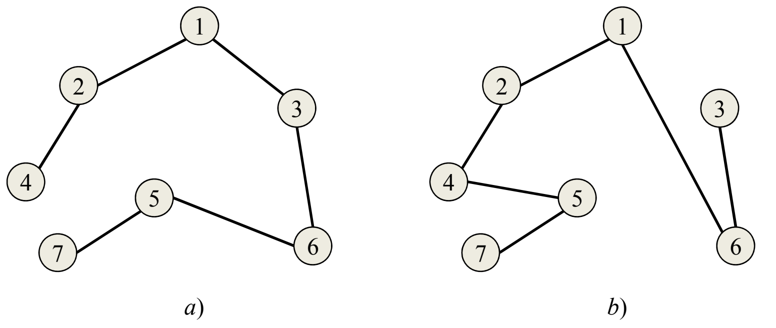


Fig. 1.2. Two spanning trees of degree 2.

Solution. Number the edges. Obviously, the edge 1 = (5, 7) will be included in all spanning trees. One can build a binary decision tree, starting from the edge 1, including (if it does not lead to a cycle, or that exceeds the degree of vertices), or not including the next edge. As a result, we obtain three Hamiltonian paths. One of them is shown in Fig. 1.1*b*, the other two – in Fig. 1.2.

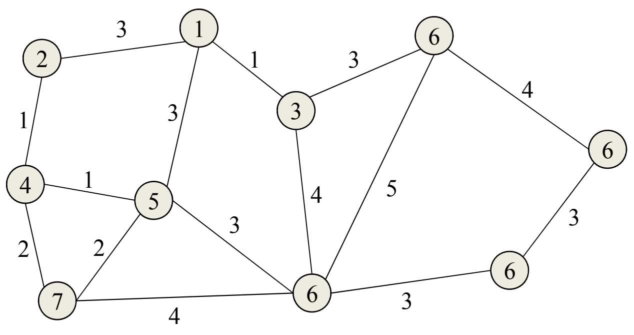


Fig. 1.3. Next to each edge lists its weight.

**Exercise 1.1.** Prove that the algorithms Prima, Kruskal and Borůvka build a MST.

**Exercise 1.2.** Show that in the tree, which has two or more nodes, there are at least two vertices of degree 1.

**Exercise 1.3.** Find a spanning tree in the graph shown in Fig. 1.1*a*, where the maximum weight of the edge is minimal.

**Exercise 1.4.** Find MST in the graph shown in Fig. 1.1*a*, using the algorithms of Kruskal and Borůvka.

**Exercise 1.5.** Find all MST in the graph shown in Fig. 1.3.

# Chapter 2. Construction of the shortest paths

Given a directed (or undirected) graph $G = (V, A)$, where $V = \{1, \ldots, n\}$ is the set of vertices and $A$ – the set of arcs. Assign to each arc $(i, j) \in A$ the “length” $d_{ij} \geq 0$.

**Definition 2.1.** The *path* $P_{ij}$ from vertex $i$ to node $j$ is defined as a sequence of arcs that starts at $i$ and ends at $j$, in which the end of the previous arc is the beginning of the next one. The sum of the lengths of the arcs in a path is called the *length of the path*.

The problem of constructing the shortest paths arises naturally in a variety of applications. For example, if in the communication network an information from the source should reach the receivers during the minimal time, then we can construct a graph in which the length of the arc coincides with the time of passage of the signal along this arc and reduce problem to the problem of construction a shortest path tree (SPT) from source to receiver.

If one wants to build a network of roads connecting a given set of cities so that the travel time from one city to another is minimal, then one need to find the shortest path between all pairs of cities.

## 2.1. Dijkstra’s algorithm

Algorithm invented by the Dutch scientist E. Dijkstra in 1959 [1]. It builds a SPT from the initial vertex (the root) to all other vertices in the graph with the non-negative lengths of the arcs (edges).

Let $s$ be the initial vertex (source node). The algorithm, at each step adds to the partially constructed tree $T$ rooted at $s$ one closest to $s$ vertex which is not in $T$. If one wants to find the shortest path to the vertex $t$, then algorithm stops after the addition of this node. Otherwise, a spanning tree that connects all the vertices by the shortest paths from $s$, i.e. SPT, will be constructed.

Denoted by $N_j = \{i \in V \mid (i, j) \in A\}$. Set the initial tree $T = (s, \varnothing)$. Assign a label $d_i$ to each vertex $i \in V$, which is equal to the (current) minimum length of the path from $s$ to $i$. Source label is $d_s = 0$ and it does not change during operation of the algorithm. Initially $d_i = +\infty$ for all $i \neq s$. At each step of the algorithm looks for an arc:

$$(i, j) = \arg \min_{p \in N_q; p \in T, q \notin T} \{d_p + d_{pq}\},$$

which is added to $T$, and update the label of $j$-th vertex $d_j = d_i + d_{ij}$. Thereafter (with the constant complexity) a label of each node not in $T$ is updated: $d_k = \min\{d_k, d_j + d_{jk}\}$, $k \notin T$.

The algorithm stops when all vertices are included in the tree $T$ (after $n$ steps). The complexity of one step (selection of a node for inclusion in the $T$) is equal to $O(n)$. Therefore, the overall complexity of the Dijkstra's algorithm is $O(n^2)$.

## 2.2. Bellman-Ford algorithm

If the lengths of some edges (arcs) take negative values in the absence of cycles of negative length, shortest path from one vertex to all other builds the Bellman-Ford algorithm [1].

We introduce $p(v)$ as a node number immediately preceding the vertex $v$ in the path $P_{sv}$. Then the Bellman-Ford algorithm can be written in steps as follows.

<u>Step 1.</u> For each vertex $v \in V$: if $v = s$, then set $d_v = 0$, otherwise set $d_v = +\infty$ and $p(v) = $ null.

Step 2. For each vertex $i = 1, \ldots, n - 1$ and for each edge $(u, v) \in A$: if $d_u + d_{uv} < d_v$, then set $d_v = d_u + d_{uv}$ and $p(v) = u$.

Step 3. For each edge $(u, v) \in A$: if $d_u + d_{uv} < d_v$, then graph contains negative cycle.

**Lemma 2.1.** After performing $i$ iterations:

1) If $d_u < +\infty$, then $d_u$ is the length of some path from $s$ to $u$;

2) If there is a path from $s$ to $u$, with at most $i$ edges, then $d_u$ does not exceed the length of the shortest path from $s$ to $u$, containing at most $i$ edges.

**Proof**. At zero iteration ($i = 0$) path length containing no edges $d_s = 0$. For other nodes $d_u = +\infty$, because there is no path from source to the $u$ with zero edges. Let us prove the first statement. Consider the case where the length of the path to $v$ is changed, i.e. executed the assignment $d_v = d_u + d_{uv}$. By the induction hypothesis, $d_u$ is the length of a path from source node to $u$. Hence, the value of $d_u + d_{uv}$ is the length of the path $P_{su} \cup \{(u, v)\}$.

To prove the second statement, consider the shortest path from $s$ to $u$, with at most $i$ edges. Let $v$ be the penultimate vertex of this path. Then the part of the path from $s$ to $v$ is the shortest path from the source to $v$, containing at most $i - 1$ edges. By the induction hypothesis, $d_v$ after $i - 1$ iterations does not exceed the length of this path. Consequently, $d_{uv} + d_v$ not exceed the length of the path from $s$ to $u$. At the iteration $i$, the value $d_u$ is compared to the $d_{uv} + d_v$, and it is assigned a new value if $d_{uv} + d_v$ is less. Therefore, after $i$-th iteration, $d_u$ not does exceed the length of the shortest path from the source to $u$, comprising no more than $i$ edges. □

If graph has no negative cycles, then every vertex will have a single path of minimum length, and in step 3 it cannot occur decreasing of the length of any path. Otherwise a negative cycle exists.

## 2.3. Floyd–Warshall algorithm

Floyd-Warshall algorithm was developed in 1962 [1] and is used to find the shortest distances between all pairs of vertices of a directed weighted graph.

We denote $d_{ij}^k$ the length of the shortest path from $i$ to $j$, which passes through the vertices of the set $\{1, 2, \ldots, k\}$. Then $d_{ij}^0 = d_{ij}$ is the arc length if $(i, j)$ exists; otherwise $d_{ij}^0 = +\infty$.

If the shortest path from $i$ to $j$ does not pass through the vertex $k$, then $d_{ij}^k = d_{ij}^{k-1}$. If there exists a shortest path from $i$ to $j$, passing through $k$, then $d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$. Recursive formula for the calculation of $d_{ij}^k$ is:

$$d_{ij}^0 = d_{ij};$$

$$d_{ij}^k = \min\left\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right\}.$$

Floyd-Warshall algorithm consistently calculates the values $d_{ij}^k$ for all $i, j, k = 1, \ldots, n$. The values $d_{ij}^n$ are the lengths of the shortest paths between any nodes $i$ and $j$. It is easy to see that the complexity of the algorithm is equal to $O(n^3)$. If, in addition, for each pair of vertices we store the information about the first vertex in the path, in addition to the distance between two nodes, we find ourselves able to recover the shortest path.

There is the shortest path between the pair of vertices $i, j$, which is a part of the negative cycle as path length from $i$ to $j$ can be arbitrarily small (negative). By default, the application of the algorithm means the absence of negative cycles in the graph. However, if there is a negative cycle, the Floyd-Warshall algorithm can be used to find it:

- algorithm iteratively calculates lengths of shortest paths between each pair of nodes *i*, *j*, including $i = j$;
- first, all the length of the path from *i* to *i* are equal to zero;
- path $P_{ik} \cup P_{ki}$ length may be smaller if it is negative, i.e. this is negative cycle;
- after stopping of algorithm, path length from *i* to *i* is negative in case if there is a cycle of negative length.

The presence of negative numbers $d_{ii}$ indicates the presence of negative cycles in the graph.

## 2.4. Examples and exercises

**Example 2.1.** Build a SPT from vertex *s* to all the vertices of the directed graph shown in Fig. 2.1*a* (next to the arcs are their length) using Dijkstra's algorithm.

Solution. First we set the length of the shortest path to the node – vertex labels equal $d_s = 0$, $d_i = +\infty$, $i \neq s$, and partially built the shortest path tree setting $T = (s, \varnothing)$. Find

$$\arg \min_{p \in N_q; p \in T, q \notin T} \{d_p + d_{pq}\} = \arg \min_{s \in N_q; q \notin T} \{d_s + d_{sq}\} = (s,4)$$

and set $T = (\{s, 4\}, \{(s, 4)\})$, $d_4 = 1$. The labels of other nodes do not changed, because vertex 4 is a sink. Then find

$$\arg \min_{p \in N_q; p \in T, q \notin T} \{d_p + d_{pq}\} = (s,2)$$

and set $T = (\{s, 2, 4\}, \{(s, 4), (s, 2)\})$, $d_2 = 3$. Update the label of the node 1: $d_1 = \min\{d_1, d_2 + d_{21}\} = \min\{+\infty, 3+2\} = 5$. Labels of other nodes do not changed. Continuing the process, add successively in the constructed tree edges (*s*, 5), (2, 1), (1, 3), (3, 6) and the nodes 5, 1, 3, 6. As a result, a tree will be constructed as shown in Fig. 2.1*b*, in which the label next to the node is the length of the shortest path from the source *s*.

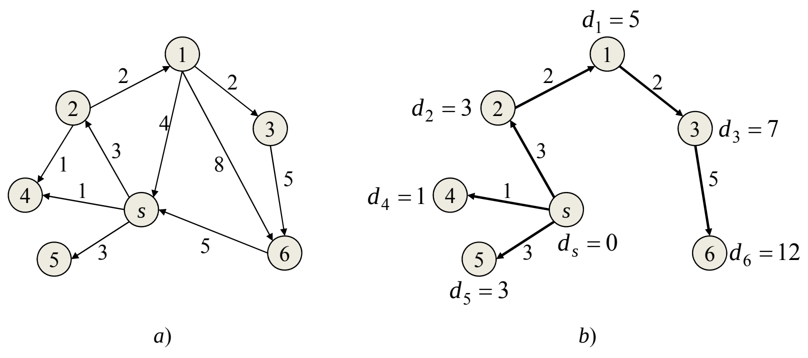


Fig. 2.1.

**Example 2.2.** Find the length of the shortest paths between all pairs of vertices of the graph shown in Fig. 2.1*a* using the Floyd-Warshall algorithm.

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | - | 2 | - | - | 8 | 4 |
| 2 | 0 | - | 1 | - | - | - |
| - | - | 0 | - | - | 5 | - |
| - | - | - | 0 | - | - | - |
| - | - | - | - | 0 | - | - |
| - | - | - | - | - | 0 | 5 |
| - | 3 | - | 1 | 3 | - | 0 |

*a*) $d_{ij}^0$

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 7 | 2 | 5 | 8 | 8 | 4 |
| 2 | 0 | 4 | 1 | - | 10 | 6 |
| - | - | 0 | - | - | 5 | 10 |
| - | - | - | 0 | - | - | - |
| - | - | - | - | 0 | - | - |
| - | 8 | - | 6 | 8 | 0 | 5 |
| 5 | 3 | - | 1 | 3 | - | 0 |

*b*) $d_{ij}^1$

Table 2.1.

<u>Solution.</u> We prescribe the initial lengths of the paths between the nodes as a matrix with the elements $d_{ij}^0 = d_{ij}$, setting $s = 7$ (see Table. 2.1*a*, in

which the symbol "-" corresponds to infinity). We use the recurrence relations $d_{ij}^{1} = \min\{d_{ij}^{0}, d_{ik}^{0} + d_{kj}^{0}\}$ for calculating the path lengths after the first iteration (see Table. 2.1*b*). There are no paths from the nodes 4 and 5 to the other nodes, so the infinity in the lines 4 and 5 will continue as after the first iteration, and further.

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 7 | 2 | 5 | 8 | 8 | 4 |
| 2 | 0 | 4 | 1 | 9 | 10 | 6 |
| 17 | 13 | 0 | 11 | 13 | 5 | 10 |
| - | - | - | 0 | - | - | - |
| - | - | - | - | 0 | - | - |
| 10 | 8 | 12 | 6 | 8 | 0 | 5 |
| 5 | 3 | 7 | 1 | 3 | 13 | 0 |

Table 2.2. $d_{ij}^{4}$

After the fourth iteration, we obtain the final values of minimum lengths of paths connecting all pairs of vertices (Table 2.2).

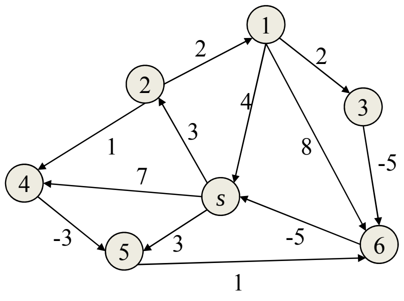


Fig. 2.1. Next to the arcs are their lengths.

**Exercise 2.1.** Prove that Dijkstra's algorithm builds a tree of shortest paths.

**Exercise 2.2.** Find the shortest path between all pairs of vertices of the graph shown in Fig. 2.1*a*.

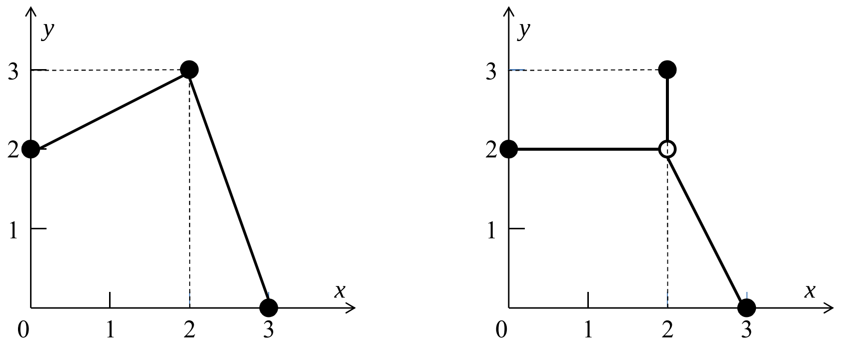
**Exercise 2.3.** Whether there is a negative cycle in the graph shown in Fig. 2.2?

**Exercise 2.4.** Use the Bellman-Ford algorithm to construct the shortest paths from the vertex $s$ in the graph shown in Fig. 2.2, removing the arcs ending in $s$.

**Exercise 2.5.** Prove the correctness of the Floyd-Warshall algorithm.

# Chapter 3. Steiner tree problem

Suppose there are three points in the plane, the distance between them is determined by the Euclidean metric, and we want to connect these points through the MST. The corresponding tree is shown in Fig. 3.1*a*, and its length (the sum of the lengths of the edges) is equal $\sqrt{5}+\sqrt{10}$.



*a*) MST *b*) Steiner tree

Fig. 3.1.

Let's introduce an additional (intermediate) point and connect the original points with intermediate point. The result is a tree of length $\sqrt{5}+3$ (Fig. 3.1*b*). This tree is a *geometric* (or *metric*) Steiner tree, and it contains one extra point (2, 2) – the *Steiner node*. In order to construct the minimal Steiner's tree, one can use additional points in the plane to reduce the length of the tree.

Next, let's consider the Steiner tree problem in graph, the statement of which is given in the next section. In the last problem the Steiner nodes are selected from a given set of vertices.

## 3.1. Steiner tree problem and its complexity

Given a simple undirected weighted graph $G = (V, E)$, where the vertex set is the union of two sets $V = S \cup I$. The vertices in $S$ are called the *terminals*, and the vertices in the set $I$ are *intermediate* vertices. Each edge $(i, j) \in E$, $i, j \in V$, attributed to the "weight" (or "length") $c_{ij} \geq 0$. The goal is to link the vertices in $S$ with the minimum weight tree, which is called the minimum-weight Steiner tree (MWST). In this case, the required tree may include the intermediate vertices in the set $I$.

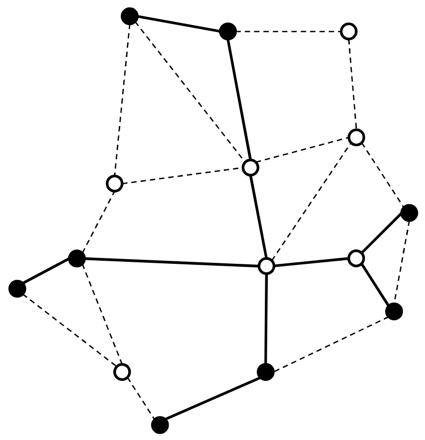


Fig. 3.2. Example of the Steiner tree in a graph.

In Fig. 3.2 by dashed lines are shown the edges of the graph $G$, which connect terminals – painted nodes and intermediate (not painted) vertices. The solid lines show the Steiner tree, which include three intermediate nodes.

For a MWST on the plane with the Euclidean metric we have the following properties.

**Property 3.1.** Steiner point has degree 3.

**Property 3.2.** If vertex *i* has degree 3 in the MWST, then the angle between any two edges incident to *i*, is equal to 120°.

**Property 3.3.** The number of Steiner points in MWST is $k \in [0, |S| - 2]$.

It is known that the Steiner tree problem in graphs (as well as the geometric Steiner tree problem) is strongly NP-hard [4], so in practice, various approximation algorithms for constructing MWST are used.

## 3.2. Approximate algorithms

Obviously, MST is a fesible solution to the Steiner tree problem, and the ratio $W_{\text{MST}} / W^* \le 2$ holds in the case of Euclidean distance, where $W_{\text{MST}}$ is the weight of the minimum spanning tree, and $W^*$ is the minimum length of the Steiner tree. That is, Prim's (or Kruskal's) algorithm builds a 2-approximate solution to the Steiner tree problem by polynomial complexity.

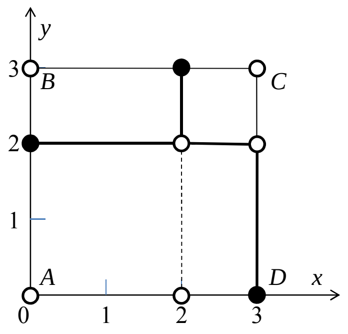


Fig. 3.3. The bold lines show MWST in the Hannan's grid

For the geometric Steiner tree problem (with the Euclidean metric) there is stronger result: $W_{\text{MST}} / W^* \le 2/\sqrt{3}$. It is also proposed several algorithms that build a 2-approximate solution to the Steiner tree problem for arbitrary graphs. For example, it is sufficient to find the shortest path between each pair of vertices of the graph, go to the new graph without intermediate nodes with edge lengths equal to the lengths of the shortest paths, and build a MST. This will be a 2-approximate solution to the original Steiner tree problem.

Published a number of papers in which  the algorithms with a lower ratio are proposed, for example, the algorithm of constructing 1.55-approximate solution for the case where the points are located on the plane and the distance between them is given by a rectilinear (Manhattan) metric $L_1$ [6]. In this case, the Steiner tree consists of a set of vertical and horizontal segments connecting terminals and intermediate points. Uniquely determined the smallest rectangle containing all terminals, and its sides are parallel to the axes (rectangle *ABCD* in Fig. 3.3). Obviously, all the edges of the MWST are within the the rectangle. In 1966, Hannan showed [7] that there is a MWST, which includes only the nodes of the lattice, resulting from the intersection of the horizontal and vertical lines passing through terminals – Hannan's grid (Fig. 3.3). In 1976, Hwang [8] showed that the MST is a 3/2-approximate solution in thia case. In 1992, Zelikovsky [9] developed an algorithm for constructing a rectangular Steiner tree with ratio 11/8, the first heuristic algorithm that builds a better solution than MST.

## 3.3. Some problems of network synthesis using Steiner trees

The problems of constructing the Steiner trees that have certain properties arise in the synthesis of data networks, the design of communications, roads, pipelines, routing very large scale integrated circuits (VLSI), etc.

For example, consider the following problem of constructing MWST with restrictions on the lengths of the paths from a given node – the root or source of the signal, which arises in connection with a routing in VLSI. In [10] problem is set as follows.

Given a weighted graph $G = (V, E)$, $V = \{0, 1, \dots, n\}$, where the selected node 0 is called the root. For each terminal $k \in S \subseteq V$ the maximum permissible length of the path from the root $d_k \geq 0$ is given. Each edge $(i, j) \in E$ attributed integer "weight" $c_{ij} \in [c, C]$ and "length" $d_{ij} \in [d, D]$, $c \geq 0$, $d \geq 0$. The problem

$$W(T) = \sum_{(i,j)\in T} c_{ij} \to \min_{T\in F}; \tag{3.1}$$

$$\sum_{(i,j)\in P_k(T)} d_{ij} \leq d_k, \quad k \in S, \tag{3.2}$$

where $F$ is a set of Steiner trees spanning nodes in $S$, and $P_k(T)$ is a path from the root to the vertex $k$ in the tree $T$.

The problem (3.1) – (3.2) is NP-hard even in the case $S = V$. In [10] the following heuristic algorithm is proposed to solve the problem (3.1) – (3.2).

First, the tree is being constructed from a single root, i.e. $T_0 = (\{0\}, \varnothing)$. At each iteration $k$ of the algorithm to a tree built in the previous steps $T_{k\text{-}1}$ one vertex $j \notin T_{k\text{-}1}$ and one edge $(i, j) \in E$ are added:

$$(i, j) = \arg \min_{i\in T_{k-1},\, j\notin T_{k-1}} \{c_{ij} + q(d_{ij} + R_i)\},$$

where $R_i$ is a length of path $P_i(T_{k\text{-}1})$, $R_j = d_{ij} + R_i$, the parameter $q \geq 0$, and get a tree $T_k$.

The variation of the parameter $q$ allows to control the "quality" of the solution. In particular, we show that in the case of integer weights and lengths of the edges the mentioned algorithm builds a tree of shortest paths when $q \geq Cn$. When $q = 0$, the algorithm obviously builds a kind of Steiner tree without restrictions on the lengths of the paths. By changing parameter's value, one can get different trees and remember the best feasible tree as an approximate solution. It is shown that for the construction of various trees it is enough to run the algorithm with a *finite* number of values of $q$, which leads to the polynomial complexity of $O(n^2 \log(CDn))$.

The above algorithm does not have a guaranteed ratio and for its analysis the numerical experiment was performed, which showed a high efficiency of the approach.

Let us turn to the next example. One of the stages of designing large-scale integrated circuits (after placement of elements in VLSI) is a routing. This phase is divided into global and detailed routing. Global routing is one of the most important stages of VLSI design, where for each circuit the routing areas under resourse and delay constraints is determined. In the literature there are several formulations of the global routing problem (GRP) with different criteria and constraints. The main objective of the global routing is tracing all of VLSI circuits without constraint violations. However, even the simplest setting in which to carry out the routing of two-terminal circuit with limited routing resources (without time delay) is an NP-hard problem.

To solve the GRP the researchers proposed various approaches where the routing is usually performed on only two layers. At the core of these approaches are sequential routing algorithms, routing and rerouting algorithms, algorithms based on the solution of of multicommodity flow problems, hierarchical methods, and different metaheuristics.

In the modern VLSI at the design stage of the global routing along with the trace resource increasing attention gets the signal propagation time. The wire density and the time delay are generally competing criteria and practically absent in the literature publications, which are considered together these criteria.

VLSI *logical scheme* can be represented by an acyclic graph with several *primary inputs* and *primary outputs*. It includes heterogeneous elements connected by partially ordered circuits. Each element of the circuit implements a Boolean function and has several inputs and one output, which, along with the primary inputs and outputs are called *terminals*. The circuit is defined by a single terminal-source and multiple terminals – the recipients of the signal. For example, in Fig. 3.4 the circuit shown in bold lines, has one source – the output of element 5 and the four terminals that are the inputs of the 7, 8, 9 and 11.

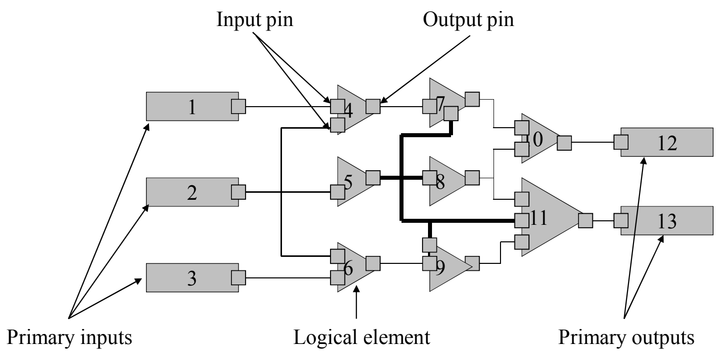


Fig. 3.4. An example of a logical network

For each primary input the *arrival time* (AT) of signal is set, and for each primary output the most later allowed time to get a signal (RT – *required time*) is given.

Layout routing area consist of identical rectangles, one above the other, called layers. The deployment of elements of the integrated circuit is given, so the coordinates of all the terminals are assumed to be known.

The terminals of adjacent layers are connected by “vias”, which will be considered parallel to the axis 0*z*. Trace each layer is either parallel to the axis 0*x*, or parallel to the axis 0*y*. However, some layers can be used for placement of circuit elements only.

On the stage of the global routing all the layers are split into identical global cell-shaped rectangles.

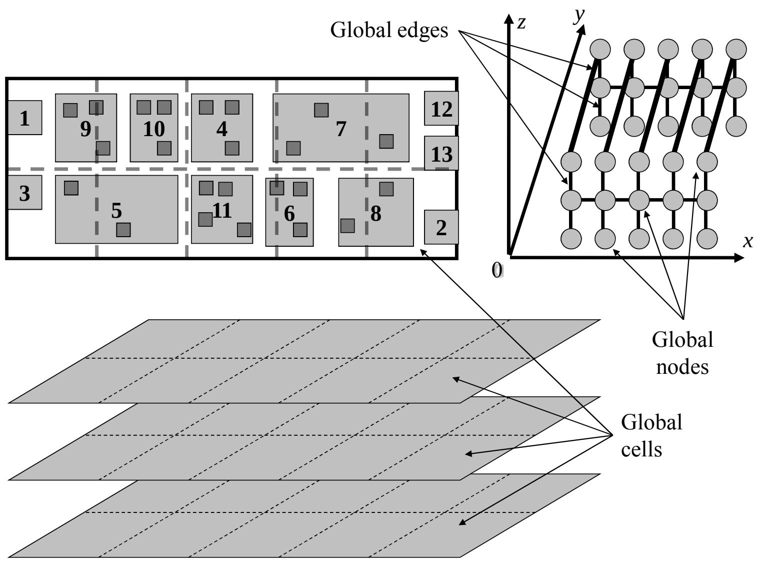


Fig. 3.5. An example of building a global graph

As a result, each terminal falls into one of these cells. Then built the next global graph. Each global cell is assigned a global vertex. A pair of global vertices are connected with a global edge in the following cases (see Fig. 3.5):

- if they are located on the same layer and corresponding global cells have a common side which is perpendicular to the given routing direction;

- if the vertices are located on neighboring layers, and $(x, y)$-coordinates coincide.

To connect the terminals of every circuit a Steiner tree in the global graph is used. Each edge is attributed to the global trace resource or capacity – the maximum number of occurrences of this edge in the set of Steiner trees.

So, let us assume that the global graph $G = (V, E)$ with vertex set $V$ and edge set $E$ is given. To each edge $(i, j) \in E$ is assigned the length $l_{ij} > 0$, the capacitance $c_{ij} \geq 0$, the electrical resistance (resistance) $r_{ij} \geq 0$ and capacity of $q_{ij} \geq 0$.

Circuit $s$ (denote it just the number $s = 1, \ldots, S$) is given as a set of terminals $V^s \subseteq V$. For the terminals, which are the primary inputs of VLSI, the time of receipt of signals from the outside (ATs) is given. For each of the primary outputs a later feasible time of signal reception (RT) is specified. Terminal $i \in V^s$ has a capacity $c_i^s \geq 0$, which is determined by the type of the element. Each circuit $s$ has one source $0^s \in V^s$ with the resistance $r_0^s \geq 0$.

For the analytical calculation of the delay, in the modern VLSI commonly the Elmore model is used. Let tree $T$ rooted in the node $0$ is given. Denoted by:

- $P_k(T)$ – path from the root to the vertex $k$ in the tree $T$;
- $C_j$ – capacity of the subtree $T_j$ rooted in the node $j$, i.e. $C_j = \sum_{(i,j)\in T_j} c_{ij} + \sum_{i\in T_j} c_i$ .

Then the propagation time along the arc $(i, j) \in T$ is calculated by the as

$$d_{ij} = d_{ij}(T) = r_{ij}\left(\frac{c_{ij}}{2} + C_j\right). \tag{3.3}$$

The propagation time from the root to any vertex *k* in tree *T* is given as

$$t_k = t_k(T) = r_0 C_0 + \sum_{(i,j)\in P_k(T)} d_{ij} . \tag{3.4}$$

Let the time delays are computed using Elmore formulas (3.3)-(3.4). In a given *n*-vertex tree, the complexity of these calculations is *O*(*n*).

In GRP is necessary to link each set of vertices $V^s$, $s = 1, \ldots, S$, using the Steiner tree so that each edge of the global graph $(i, j) \in E$ included not more than in $q_{ij}$ different trees, and the arrival time in the primary outputs do not exceed the RTs.

In practice, the permissible tracing can not exist, or it is difficult to find due to the large dimension of the problem and its NP-hardness. Therefore, the problem of perform routing *close to the admissible* and further namely this problem we call the global routing problem. In other words, the GRP is to find the routes which results in compromise between the excess over tracer resources and time delay.

To solve the GRP, the following iterative procedure is proposed. Sorting the circuits is performed first, based on one of the empirical criteria, which can be used as the number of terminals in the chain, a minimum perimeter of a rectangle containing all terminal circuit and others. Then, for each circuit *s* in the ordered list, taking into account the time delays and residual capacities of edges of the global graph, we construct a set of *candidate trees* $Q^s$. It then selects one tree from each set $Q^s$, $s = 1, \ldots, S$. To do that, we solve the problem of minimizing the sum of penalties for exceeding the trace of resources which is formulated in the form of quadratic integer programming. Adapted gradient algorithm is proposed which yields a solution of continuous relaxation of the latter problem, and then using various heuristics we build an integer solutions, the best of which is selected as an approximate solution of GRP.

To construct a timing-driven tree we proposed the algorithm MAD. Consider arbitrary circuit $s$ with the sourse node $0 \in V^s$. MAD builds a Steiner tree for the circuit $s$ in some connected subgraph $G' = (V', E')$ of the graph $G$, where $V' \supseteq V^s$. From $G'$ the edges which capacity is less than some integer $q$ (a parameter of the algorithm) are removed. For each of the remaining edges $(i, j) \in E'$ (using (3.3)) delay $d_{ij}$ is calculated. It can be calculated in a tree constructed at the previous iterations (initially this tree has no edges). The different ways of calculating delays $d_{ij}$ are discussed below.

Algorithm MAD is a modification of Dijkstra's algorithm, but does not guarantee the construction of a tree with a minimum delay due to the specific Elmore model. Note that the construction of the tree with the minimum transmission time from the root to the terminals is NP-hard problem, because special case where the resistance of the edges are zero, that is a Steiner tree problem in graphs.

The topology of the constructed tree depends on the sub-graph $G'$, the values of the parameter $q$ and the edge delays $d_{ij}$. As $G'$ can be used the entire graph $G$, the Hannan’s graph, the minimal grid containing the set of terminals, or other sub-graphs of the global graph.

For the construction of a variety of trees according to Elmore delays proposed to apply the algorithm MAD iteratively using the previously constructed trees to calculate $d_{ij}$. As can be seen from (3.3), the value of $d_{ij}$ depends on the capacitance $C_j$ of the subtree $T_j$. Since the capacitance of the subtree $C_j$ in the process of constructing the tree is not known to compute $d_{ij}$, one can use the values of this magnitude in the previously constructed trees. For example, $C_j$ can be a capacitance of the subtree $T_j$, built at the previous iteration, or the arithmetical mean of the capacitances in all subtrees $T_j$ in previously constructed trees.

The diversity of trees (using algorithm MAD) one can achieves by changing the values of the parameter *q*. Furthermore, one can use the trees constructed by other algorithms. For example, if the resistance and capacitance is proportional to the length of the links, one can build a minimal Steiner trees as a candidate tree, and trees that contain the shortest paths from the root to the terminals. Since the problem of construction a minimum Steiner tree belongs to the class of NP-hard problems, it is necessary to use different approximation algorithms that build *different* trees. This will expand the set of candidate trees, which may allow to get a better solution with the trace of the global resources of the graph.

Suppose that for each circuit *s* the set of candidate trees is found $Q^s$. If for some *s* the set $Q^s$ consists of a single tree, it included into solution, and then the capacities of global edges and circuit is recalculated and *s* is excluded from the list.

Consider the problem of optimal allocation of routing resources, that is the problem of choosing the trees of the sets $Q^s$, $|Q^s| > 1$. For brevity, the subscript $e \in E$ is used for the edges of a graph *G*, and the index $t \in J = \bigcup_{s=1}^{S} Q^s$ – for the trees. Let $a_{et} = 1$, if edge *e* belongs to the tree *t*, and $a_{et} = 0$ otherwise. The variable $x_t$ takes value $x_t = 1$, if tree *t* is selected, and $x_t = 0$ otherwise.

In our notation, consider the problem:

$$f(x) = \sum_{e \in E} \left( \max \left\{ 0, \sum_{t \in J} a_{et} x_t - q_e \right\} \right)^2 \to \min_{x_t \in \{0,1\}} ; \tag{3.5}$$

$$\sum_{t \in Q^s} x_t = 1, \quad s = 1, \ldots, S . \tag{3.6}$$

The objective function (3.5) is the total penalties for exceeding the trace resources $q_e$, $e \in E$. Tracing without exceeding the capacities of global edges corresponds to zero value of the objective function.

Replacing the integrality condition of variables on the condition of non-negativity, we get a continuous relaxation of the problem (3.5) – (3.6). Smoothness and convexity of the objective function allows us to solve the last problem with the gradient algorithm. We propose a gradient algorithm in which the complexity of each iteration is $O(|E|\cdot|J|)$, and the total number of iterations is at most $O(\varepsilon^{-1}\ln\varepsilon^{-1})$. In the proposed method, at each iteration we are not trying to find the direction of steepest descent, which would increase the complexity of the algorithm, but choose the direction of descent from the current point to the *integer* point.

Choosing one tree $t_s$ from each set $Q^s$ gives a set of Steiner trees, which is an approximate solution of the problem (3.5) – (3.6). This set of trees depends on the current point and, therefore, may vary from iteration to iteration. During the descent we keep the best (for example, in the sense of the criterion (3.5), or the density of connections) of the found solutions.

The last example is the construction of the signal tree in the computing system. Signal network is responsible for the synchronization of the whole system. Command signals are generated outside the system and fed into it through the entrance (the root). Each functional element is connected with the root through the signal network. It performs a series of logical operations (functions) and waiting for the signal to transmit the results to the other elements, until the next cycle of calculations. Thus, there is control information flow within the computer system. *Clock skew* is the maximum time difference between receiving the signals by the various components of the system. The increase in clock skew in computing systems leads to decrease in the rate of calculation. In the modern systems, where the size of elements is significantly less than a micron, clock skew is one of the main factors that determine the functioning of the system. Clock skew reduces the clock frequency, as between two successive signals a period should be increased so that all circuit components have time to receive a signal. It is believed that high-speed circuitry for clock skew should not exceed 5% of the maximum transmission time.

Synchronization problem can be formulated as follows. Each terminal (circuit element) performs certain operations. All terminals operate part of the overall program and they need to work in concert. The requirement for receiving signals of all terminals at the same time can be satisfied, if one build a tree in which all the paths from the source to the terminals have the same delay. Such a tree is said to be *feasible*. Furthermore, among the feasible trees a minimum weight tree should be selected, which would minimize the space occupied by the tree.

This routing is done on the edges of the uniform planar rectangular grid (in particular, have that degree of each vertex in a graph does not exceed four). Consequently, not always possible to construct a valid tree.

From a mathematical point of view, the problem is to construct a rectlinear minimum Steiner tree in which the distance (in the tree) from the root to each terminal is equal to the radius of the grid graph. It is known that this problem is NP-hard. Approximation algorithm for solving the problem is proposed, for example, in [11].

## 3.4. Examples and exercises

**Example 3.1.** Find MWST to link up four points located in the plane in pairs symmetrically with respect to the horizontal line (Fig. 3.6).

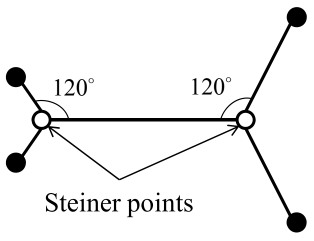
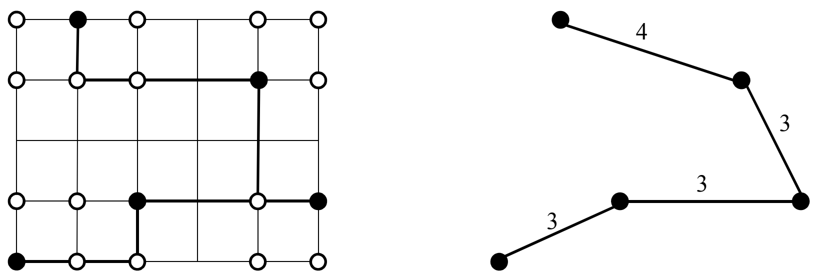


Fig. 3.6.

Solution. From the properties of metric MWST follows that the number of Steiner points is not greater than 2, and the edges incident to these points form an angles of 120°. Draw a horizontal line halfway between the left and right points. From each point draw a line that intersects the horizontal line at 120°. The points of intersection – Steiner points. MWST built, it included 2 Steiner points (Fig. 3.6).

**Example 3.2.** Suppose the graph vertices are located in the unit grid as shown in Fig. 3.7. It is necessary to link terminals by 2-approximate Steiner tree using a rectangular metric $L_1$.



*a*) Hannan's grid and MWST *b*) MST

Fig. 3.7

Solution. We first construct a Hannan's grid, having horizontal and vertical lines passing through the terminals (Fig. 3.7*a*). Find the length of the shortest path between each pair of terminals and construct a MST (Fig. 3.7*b*). On a grid representation of the constructed tree is not unique, but any representation is a 2-approximate solution of the Steiner tree problem. The Fig. 3.7*a* shows a tree that is MWST.

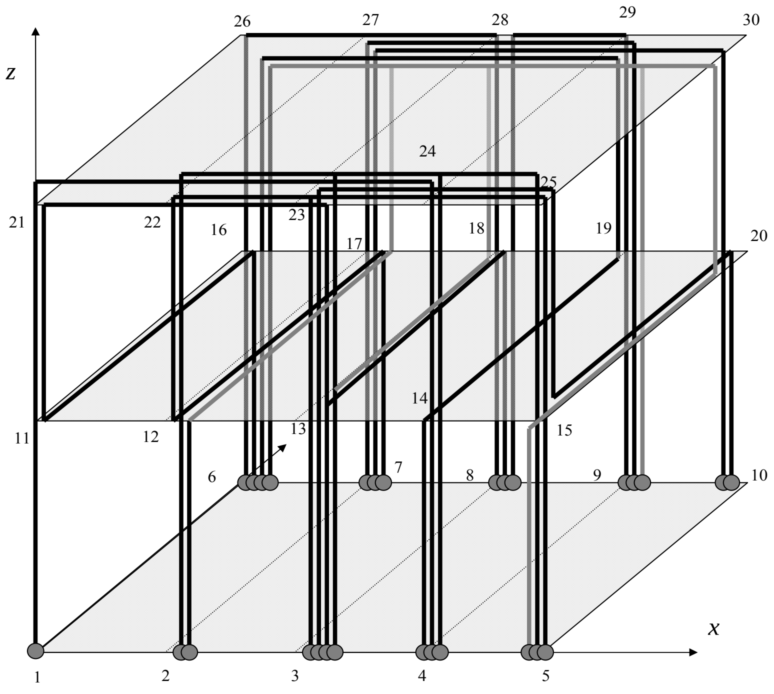


Fig. 3.8. Example of routing in 3-layer global graph.

**Example 3.3.** To illustrate the method to solve the GRP, we give an example. Suppose there are 11 circuits (networks), 3 layers and 5×2 grid. All elements of VLSI are located on the ground (bottom) layer, the vias have a high capacity, a zero capacitance and the resistance 0.00937. The second layer is used for vertical wires (i.e., parallel to the axis 0y), the link (corresponding to the global edge of the graph) has a capacity 16, a specific capacitance 0.470431, and specific resistance 0.016209. A vias between the layers 2 and 3 have a high capacity, a zero capacitance and a resistance 0.00781. The third layer is used for connections parallel to the axis 0x and has a capacity 12, a specific capacitance 0.456122 and a specific resistance 0.010763.

Fig. 3.8 shows the result of the method, which built a trace with the density (the maximum number of wires at one edge of the graph) equals 5. Arrival time in the primary output 12 is 34 ps (picoseconds), and AT in the primary output 13 is 36 ps. Fig. 3.9 shows the projection of the trees on the plane (*x*, *y*).

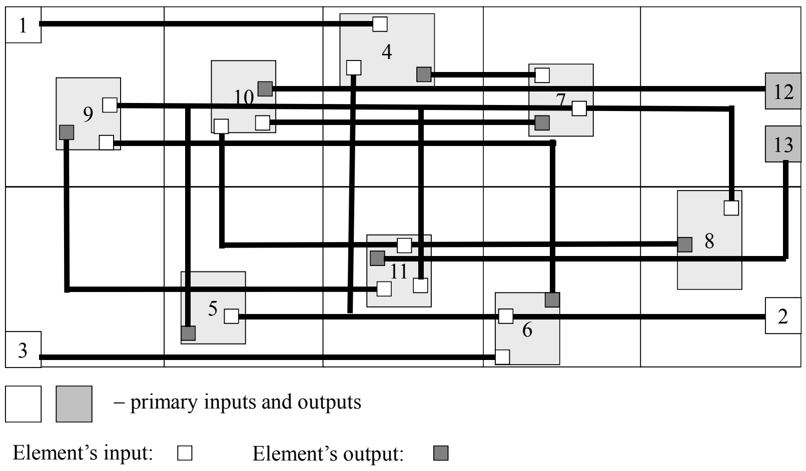


Fig. 3.9. Projections of trees

The similar example was resolved after the addition of two layers (resistivity layers 4 and 5, we reduced 10 times as compared with the layers 2 and 3, the specific capacitance and via characteristics left unchanged as layers 2 and 3 respectively). The algorithm has built the solution shown in Fig. 3.10 with a routing density of 3. Here we do not take into account the density of the vias, as the capacity of such edges in the example is not limited. ATs in the primary outputs 12 and 13 are comparable with similar times for the 3-layer routing.

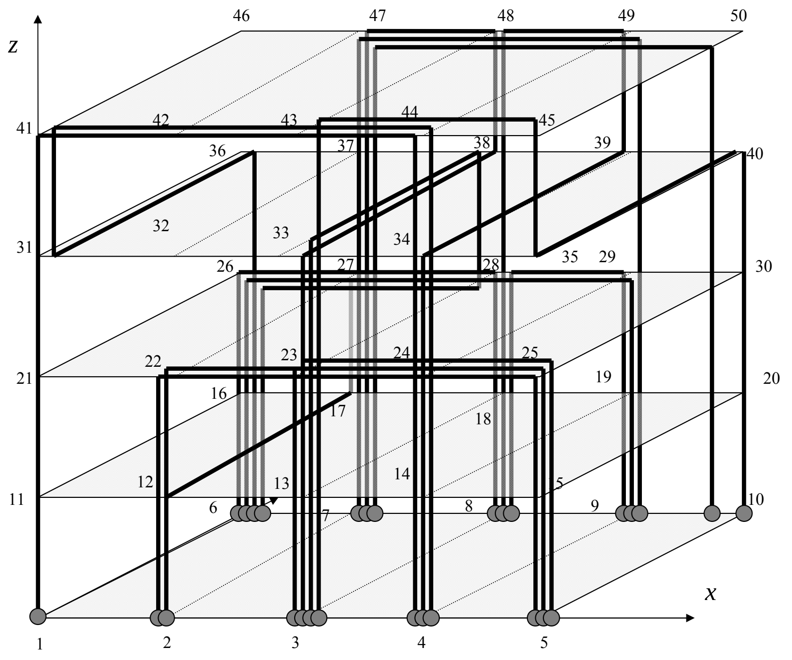


Fig. 3.10. 5-layer routing. Density is 3

**Exercise 3.1.** Build a MWST in the graph shown in Fig. 3.7*a* for the case of Euclidean metric.

**Exercise 3.2.** Build a MWST in the graph shown in Fig. 3.7*a*, where the path lengths from the bottom left of the terminal is minimal. Euclidean metric. Rectangular metric.

# Chapter 4. Traveling salesman problem

In the traveling salesman problem we are given a matrix of pairwise distances between $n$ cities. You want to find an order of cities that minimizes the total distance of a journey when a salesman visits each city exactly once and comes back to the initial city. In other words, in complete weighted undirected graph we wish to find a Hamiltonian cycle of a minimum weight. In the directed graph, we need to find a Hamiltonian tour of minimum weight. The traveling salesman problem has many applications, from VLSI chip fabrication to X-ray crystallography, and a long history. Decades of research, combined with the rapid growth in computer speeds and memory capacities, have led to one new record after another. Over the past 25 years, the record for the largest nontrivial TSP instance solved to optimality has increased from 100 cities to 1000000 cities. It is NP-hard problem. So, any exact algorithm must have a worst-case running time that grows faster than any polynomial (assuming P ≠ NP). This leaves researchers with two alternatives: either look for heuristics that merely find *near*-optimal tours, but do so quickly, or attempt to develop optimization algorithms that work well on ''real-world,'' rather than worst-case instances. Because of its simplicity and applicability, the TSP has for decades served as an initial proving ground for new ideas related to both these alternatives. These new ideas include most of the local search variants covered in this chapter, which makes the TSP an ideal subject for a case study. In addition, the new ideas include many of the important advances in the related area of combinatorial optimization, and to keep our discussions of optimization algorithms in perspective, let us begin from the well-known definitions of the complexity classes P and NP.

## 4.1. Computational complexity

The theory of NP-completeness is perhaps one of the most interesting topics in computer science. The major investigator of this field, Professor S. A. Cook of Toronto University received a Turing Award for his contribution in this field of research. There is no doubt at all, among many interesting research results in computer science, the theory of NP-completeness is one of the most exciting, and also puzzling, theories. The main theorem, now called Cook's theorem, is perhaps the most widely cited theorem. Now we will try to explain the real meaning of the theorem.

The theory of computational complexity is important because it identifies a large class of difficult problems. Here, by a difficult problem, we mean a problem whose algorithmic lower bound seems to be in order of an exponential function. In other words, the theory of NP-completeness has identified a large class of problems which do not seem to have any polynomial time algorithms to solve them.

Roughly speaking, we may say that the theory of NP-completeness first points out that many problems are called NP problems. The notation NP denotes non-deterministic polynomial. Let us first define a non-deterministic algorithm as follows: A non-deterministic algorithm is an algorithm consisting of two phases:  guessing and checking. Furthermore, it is assumes that a non-deterministic algorithm always makes a correct guessing.

For instance, given the satisfiability problem with a particular Boolean formula, a non-deterministic algorithm at first guesses an assignment and then checks whether this assignment satisfies  the formula or not. An important concept to note here is that a correct solution is always obtained by guessing.  In other words, if the formula is satisfiable, them a non-deterministic algorithm always guesses correctly and obtains an assignment satisfying this formula.

Consider the traveling salesperson decision problem: for given constant $c$, we need to check is there a Hamiltonian tour with length at most $c$. A non-deterministic algorithm will always guess a tour and check whether this tour is shorter than the constant $c$.

The reader may by outraged by this concept of non-deterministic algorithms because it is physically impossible to have such an algorithm. How can we always make a correct guess? Actually, non-deterministic algorithms do not exist and they will never exist. The concept of non-deterministic algorithms is useful only because it will help us later define a class of problems, called NP problems.

**Definition 4.1**. *If the checking stage of a non-deterministic algorithms is of polynomial time complexity, then this non-deterministic algorithms is called a non-deterministic polynomial algorithms. If a decision problem can be solved by a non-deterministic polynomial algorithms, this problem is called a non-deterministic polynomial (NP for short) problem.*

From the above definition, we may immediately conclude that *every problem which can be solved in polynomial time (by deterministic algorithms, of course) must be a non-deterministic polynomial problem.* Typical cases are searching, merging, sorting, and minimum spanning tree problems. The reader is reminded here that we are talking about decision problems. Searching is a decision problem; sorting is obviously not. But we can always create a decision problem out of the sorting problem. The original sorting problem is to sort $a_1, a_2, \dots, a_n$ into an ascending or descending sequence. We may construct a decision problem as follows: Given $a_1, a_2, \dots, a_n$, and $C$, determine whether there exists a permutation of $a_i$-s $(a_1, a_2, \dots, a_n)$ such that $|a'_2 - a'_1| + |a'_3 - a'_2| + \cdots + |a'_n - a'_{n-1}| < C$. All of the problems which can be solved in polynomial time are called P problems.

The satisfiability problem and the traveling salesman decision problem are both NP problems because the checking stage for the both problems is of polynomial time complexity. In fact, most solvable problems that one can think of are NP problems.

A famous decision problem which is not an NP problem is the halting problem. The halting problem is defined as follows: Given an arbitrary program with an arbitrary input data, will the program terminate or not? Another problem is the first-order predicate calculus satisfiability problem. Both problems are the first-order predicate calculus satisfiability problem. Both problems are so called undecidable problems.

Undecidable problems  cannot be solved by guessing and checking. Although they are decision problems, somehow they cannot be solved by exhaustively examining the solution space. The reader should notice that the Boolean logic (or also called propositional logic), an assignment is characterized by an $n$-tuple. But, for first-order predicate calculus, an assignment is not bounded. It may be of infinite length. This is why the first-order predicate calculus is not an NP problem. It suffices to remind the reader that undecidable problems are even more difficult than NP problems.

Let us be more explicit. For the satisfiability problem and the traveling salesman decision problem, the number of solutions is finite. There are $2^n$ possible assignments for the satisfiability problem and $(n-1)!$ possible tours for the traveling salesman decision problems. Therefore, although these problems are difficult, they at least have some upper bounds for them. For instance, for the satisfiability problem, we can at least use an algorithm with $O(2^n)$ time complexity to solve it. There is, however, no such upper bound for the undecidable problems. It can be shown that the upper bounds never exist. Intuitively, we may say that we can let the problem run, say 1 million years, and still cannot make any conclusion because it is still possible that in the next step, the program halts. Similarly, for the first-order predicate calculus satisfiability problem, we have the same situation. Suppose after running the program for a very long time, we still have not produced an empty clause. But, it is actually possible that the next clause being generated is an empty clause.

Now we shall introduce Cook's theorem. We shall only give an informal proof because a formal proof is very complicated. Cook' theorem can be stated follows.

**Theorem 4.0 (Cook)** *NP=P if and only if the satisfiability problem is a P problem.*

The proof of the above theorem consists of two parts. The first part is "If NP=P, then the satisfiability problem is a P problem". The part is obvious because the satisfiability problem is an NP problem. The second part is "If the satisfiability problem is a P problem, then NP=P". This is a crucial part of Cook's theorem and we shall elaborate this in the rest of this section.

Let us now explain the main spirit of Cook's theorem. Suppose that have an NP problem $A$ which is quite difficult to solve. Instead of solving this problem $A$ directly, we shall create another problem $A'$ and by solving that problem $A'$, we shall obtain the solution of $A$. It is important to note here that every problem is a decision problem. Our approach is the follows:

(1) *Since problem* $\boldsymbol{A}$ *is an NP problem, there must exist an NP algorithm* $\boldsymbol{B}$ *which solves this problem. An NP algorithm is a non-deterministic polynomial algorithm. It is physically impossible and therefore we cannot use it. However, as we shall see, we can still use* $\boldsymbol{B}$ *conceptually in the following steps.*

(2) *We shall construct a Boolean formula* $\boldsymbol{C}$ *corresponding to* $\boldsymbol{B}$ *such that* $\boldsymbol{C}$ *is satisfiable if and only if the non-deterministic algorithm* $\boldsymbol{B}$ *terminates successfully and returns an answer "yes". If* $\boldsymbol{C}$ *is unsatisfiable, then algorithm B would terminate unsuccessfully and return the answer "no".*

We shall note at this point that when we mention a problem, we mean an instance of a problem, That is, we mean a problem with a particular input. Otherwise, we cannot say that the algorithm terminates.

We shall also delay the discussion about how $\boldsymbol{C}$ is constructed. This part is the crucial part of Cook's theorem.

> (3) *After constructing formula $\boldsymbol{C}$, we shall temporarily forget about our original problem $\boldsymbol{A}$ and the non-deterministic algorithm $\boldsymbol{B}$. We shall try to see whether $\boldsymbol{C}$ is satisfiable or not. If it is satisfiable, then we say that the answer of problem $\boldsymbol{A}$ is "yes"; otherwise, the answer is "no". That we can do so is due to the property of formula $\boldsymbol{C}$ stated in Step (2). That is, $\boldsymbol{C}$ is satisfiable if and only if $\boldsymbol{B}$ terminates successfully.*

All thing are beautiful. The above approach seems to suggest that we only have to pay attention to the satisfiability problem. For instance, we never have to know how to solve the traveling salesman decision problem; we merely have to know how to determine whether the Boolean formula corresponding to the traveling salesman problem is satisfiable or not. Yet, there is a big and serious problem here. If the satisfiability problem is hard to solve, the original the traveling salesman problem is still hard to solve. This is the heart of Cook's theorem. In indicates that if the satisfiability problem can be solved in polynomial number of steps, essentially because of the above approach.

**Definition 4.2** *Let $A_1$ and $A_2$ be two problems. $A_1$ reduces to $A_2$ (written as $A_1 \propto A_2$) if and only if $A_1$ can be solved in polynomial time, by using a polynomial time algorithm which solves $A_2$.*

From the above definition, we can say that if $A_1 \propto A_2$, and there is a polynomial time algorithm solving $A_2$, then there is a polynomial time algorithm to solve $A_1$. Using Cook's theorem, we can say that *every NP problem reduces* to the satisfiability problem, because we can always solve this NP problem by first solving the satisfiability problem of the corresponding Boolean formula. From this definition, we can easily see the following: If $A_1 \propto A_2$ and $A_2 \propto A_3$, then $A_1 \propto A_3$. Having defined "*reduce to*", we can now define NP-complete problems.

**Definition 4.3** *A problem $A$ is NP-complete if $A \in NP$ and every NP problem reduces to $A$.*

From the above definition, we know that if $A$ is NP-complete problem and $A$ can be solved in polynomial time, then every NP problem can be solved in polynomial time. Clearly, the satisfiability problem is an NP-complete because of Cook's theorem. By definition, if any NP-complete problem can be solved in polynomial time, then P=NP.

The satisfiability problem was the first found NP-complete problem. Later, R. Karp showed 21 NP-complete problems. These NP-complete problems include node cover, feedback ark set, Hamiltonian cycle, etc. Karp received a Turing Award in 1985.

To show that a problem $A$ is NP-complete, we do not have to prove that all NP problems reduce to $A$. This is what Cook did when he showed the NP-completeness of the satisfiability problem. Nowadays, we merely have to use the transitive property of "reduce to". If $A_1$ is NP-complete problem, $A_2$ is an NP problem and we can prove that $A_1 \propto A_2$, then $A_2$ is NP-complete problem. The reasoning is rather straightforward. If $A$ is NP-complete problem, then all NP problems reduce to $A$. If $A \propto B$, then all NP problems reduce to $B$ because of the transitive property of "*reduce to*". Therefore, $B$ must be NP-complete.

In the previous discussion, we made the following statements:

1. The satisfiability problem is the most difficult problem among all NP problems.
2. When we prove the NP-completeness of a problem $A$, we often try to prove that the satisfiability problem reduces to $A$. Thus, it appears that $A$ is more difficult than the satisfiability problem.

To see that there is no inconsistency in these statements, let us note that every NP problem reduces to the satisfiability problem. Thus, if we are interested in an NP problem $A$, then certainly $A$ reduces to the satisfiability problem. However, we must emphasize here that saying a problem $A$ reduces to the satisfiability problem is not significant at all because it only means that the satisfiability problem is more difficult than $A$, which is well-established fact. If we successfully proved that the satisfiability problem reduces to $A$, then $A$ is even more difficult than the satisfiability problem, which is a highly significant statement. Note that $A \propto$ the satisfiability problem and the satisfiability problem $\propto A$. Therefore, so far as the degree of difficulty is concerned, $A$ is equivalent to the satisfiability problem.

We may extend the above arguments to all NP-complete problems. If $A$ is an NP-complete problem, then by definition every NP problem, say $B$, reduces to $A$. If we further prove that $B$ is NP-complete by proving $A \propto B$, then $A$ and $B$ are equivalent to each other. In summary, all NP-complete problems form an equivalent class.

Note that we have restricted NP problems to be decision problems. We may now extend the concept of NP-completeness to optimization problems by defining "*NP-hardness*".

**Definition 4.4** *A problem $A$ is NP-hard if every NP problem reduces to $A$.*

Note that $A$ is not necessarily an NP problem. In fact, $A$ may be an optimization problem. Thus, a problem $A$ is NP-complete if $A$ is NP-hard and $A$ is an NP. Through this way, an optimization problem is NP-hard if its corresponding decision problem is NP-complete. For instance, the traveling salesman problem is NP-hard.

The concept of NP-completeness is perhaps one of the most difficult concepts to understand in computer science. The most important paper concerning with this concept was written by Cook in 1971 which showed the significance of the satisfiability problem. In 1972, Karp proved many combinatorial problems are NP-complete. These two papers have been considered landmark papers on this subject. Since then, numerous problems have been proved to be NP-complete. There are so many of them that a database is maintained by David Johnson of AT&T. The best book totally devoted to NP-completeness is Garey and Johnson, 1979. It gives the history of the development of NP-completeness and can always be used as an encyclopedia of it.

Note that NP-completeness is in worst case analysis only. That a problem is NP-complete should not discourage anyone from developing efficient algorithms to solve it in average cases. For instance, there are many algorithms developed to solve the satisfiability problem. Many of them are based on the resolution principle. This principle was invented by Robinson. Recently, several algorithms for the satisfiability problem have been found to be polynomial for average cases. Another famous NP-hard problem is the traveling salesman problem, whose NP-hardness can be established by reducing the Hamiltonian cycle problem to it. There are several books devoted to the algorithms for the traveling salesman problem [1, 2, 3].

## 4.2. Theoretical results

Two fundamental complexity-theoretic results constrain the behavior of *any* heuristic for the TSP. For a given heuristic $A$ and TSP instance $I$, let $A(I)$ denote the length of the tour produced by $A$ and let $OPT(I)$ denote the length of an optimal tour. The first result concerns the best performance guarantee that is possible when there are no restrictions on the types of instances considered.

**Theorem 4.1.** *Assuming $P \neq NP$, no polynomial-time TSP heuristic can guarantee $A(I)/OPT(I) \leq 2^{p(n)}$ for any fixed polynomial p and all instances $I$.*

Fortunately, most applications impose substantial restrictions on the types of instances allowed. In particular, in most applications distances must obey what is called the *triangle inequality*. This says that the direct path between two cities is always the shortest route. Thus much of the theoretical work on TSP heuristics is predicated on the assumption that the triangle inequality holds. In this case the result of Sahni and Gonzalez (1976) no longer applies, and the only known constraint is the following much more limited (and recent) one, derived as a consequence of the deep connection between approximation and the characterization of NP in terms of probabilistically checkable proof systems.

**Theorem 4.2.** *Assuming P≠NP, there exists an* $\varepsilon > 0$ *such that no polynomial-time TSP heuristic can guarantee* $A(I)/OPT(I) \leq 1 + \varepsilon$ *for all instances I satisfying the triangle inequality.*

Even this lower bound evaporated if one is willing to consider the further restriction to instances consisting of points in the plane under a geometrical norm such as the Euclidean metric, a condition holding in many applications. For such instances, Arora (1996) has proved that polynomial time *approximation schemes* exist. Stated in Euclidean terms, Arora's result is as follows.

**Theorem 4.3.** *There is an algorithm A that, given an Euclidean TSP instance and a constant* $\varepsilon > 0$*, runs in time* $n^{O(1/\varepsilon)}$ *and guarantee* $A(I)/OPT(I) \leq 1 + \varepsilon$.

Unfortunately, Arora's result appears to be of only theoretical significance, both because of the constants built into the O( ) notation and because the algorithm also requires space $n^{O(1/\varepsilon)}$. Thus in practice, we may have to settle for algorithms with worse guarantees but better time and space requirements. In this chapter we shall concentrate on four such tour constructive algorithms: Nearest Neighbor, Greedy, Clarke-Write, and Chritofides-Serdukov. The first three all provide a substantially better guarantee than would be possible without the triangle inequality. The fourth provides a far better guarantee and is the current champion with respect to this worst-case measure.

## 4.3. Tour construction algorithms

### *Nearest Neighbor*

Perhaps the most natural heuristic for the TSP is the famous *Nearest Neighbor* algorithm (NN). In this algorithm one mimics the traveler whose rule of thumb is always to go next to the nearest as-yet-unvisited location. We construct an ordering $\pi(1), \dots, \pi(n)$ of the cities, with the initial city $\pi(1)$ chosen arbitrarily and in general $\pi(i+1)$ chosen to be the city $k$ that minimizes the distance to the next city. The corresponding tour traverses the cities in the constructed order, returning to $\pi(1)$ after visiting city $\pi(n)$ . The running time for NN as described is $O(n^2)$. If we restrict attention to instances satisfying the triangle inequality, NN does substantially better than the general upper bound of Theorem 4.1, although it is still far worse than the limit provided by Theorem 4.2. In particular, we are guaranteed that $NN\,(I)/OPT\,(I) \leq 0.5\,(\lfloor \log_2 n \rfloor + 1)$. No substantially better guarantee is possible, however, as there are instances for which the ratio grows as $O(\log_2 n)$.

### *Greedy*

Some authors use the name *Greedy* for Nearest Neighbor, but it is more appropriately reserved for the following special case of the ‘‘greedy algorithm’’ of matroid theory. In this heuristic, we view an instance as a complete graph with the cities as vertices. A tour is then simply a Hamiltonian cycle in this graph, i.e., a connected collection of edges in which every city has degree 2. We build up this cycle one edge at a time, starting with the shortest edge, and repeatedly adding the shortest remaining available edge, where an edge is *available* if it is not yet in the tour and if adding it would not create a degree-3 vertex or a cycle of length less than $n$. In view of the intermediate partial tours typically constructed by this heuristic, it is called the *multi-fragment* heuristic.

The Greedy heuristic can be implemented to run in time $O(n^2 \log n)$ and is thus somewhat slower than NN. On the other hand, its worst-case tour quality may be somewhat better. As with NN, it can be shown that $Greedy(I)/OPT\,(I) \leq 0.5(\lceil \log_2 n \rceil + 1)$ for all instances $I$ obeying the triangle inequality, but the worst examples known for Greedy only make the ratio grow as $(\log_2 n)/(3 \log \log_2 n)$.

***Clarke-Wright***

The *Clarke-Wright savings heuristic* (Clarke-Wright or simply CW for short) is derived from a more general vehicle routing algorithm due to Clarke and Wright. In terms of the TSP, we start with a pseudo-tour in which an arbitrarily chosen city is the *hub* and the salesman returns to the hub after each visit to another city. In other words, we start with a multi-graph in which every non-hub vertex is connected by two edges to the hub. For each pair of non-hub cities, let the *savings* be the amount by which the tour would be shortened if the salesman went directly from one city to the other, bypassing the hub. We now proceed analogously to the Greedy algorithm. We go through the non-hub city pairs in non-increasing order of savings, performing the bypass so long as it does not create a cycle of non-hub vertices or cause a non-hub vertex to become adjacent to more than two other non-hub vertices. The construction process terminates when only two non-hub cities remain connected to the hub, in which case we have a true tour. As with Greedy, this algorithm can be implemented to run in time $O(n^2 \log n)$. The best performance guarantee currently known (assuming the triangle inequality) is $CW(I)/OPT\,(I) \leq \lceil \log_2 n \rceil + 1$ (a factor of 2 higher than that for Greedy), but the worst examples known yield the same $(\log_2 n)/(3 \log \log_2 n)$ ratio as obtained for Greedy.

***Christofides–Serdukov***

The previous three algorithms all have worst-case ratios that grow with $n$ even when the triangle inequality holds. Theorem 4.2 does not rule out much better performance, however, and in fact a large class of algorithms do perform much better. As observed by Rosenkrantz, Stearns, and Lewis, there are at least three simple polynomial-time tour generation heuristics, *Double Minimum Spanning Tree*, *Nearest Insertion*, and *Nearest Addition*, that have worst-case ratio 2 under the triangle inequality. That is, they guarantee $A(I)/OPT\,(I) \leq 2$ under that restriction, and there exist instances with arbitrarily large values of $n$ that show that this upper bound cannot be improved. We do not discuss these heuristics in detail since they are all dominated in practice by NN, Greedy, and CW, despite the fact that their worst-case performance is so much better. One tour construction heuristic with a constant worst-case performance ratio is not so dominated, however. This is the algorithm of Christofides and Serdukov the current champion as far as performance guarantee is concerned, having a worst-case ratio of just 3⁄2 assuming the triangle inequality. This bound is tight, even for Euclidean instances. The heuristic proceeds as follows. First, we construct a minimum spanning tree $T$ for the set of cities. Note that the length of such a tree can be no longer than $OPT(I)$, since deleting an edge from an optimal tour yields a spanning tree. Next, we compute a minimum-length matching $M$ on the vertices of odd degree in $T$. It can be shown by a simple argument that assuming the triangle inequality this matching will be no longer than $OPT(I)/2$. Combining $M$ with $T$ we obtain a connected graph in which every vertex has even degree. This graph must contain an Euler tour, i.e., a cycle that passes through each edge exactly once, and such a cycle can be easily found. A traveling salesman tour of no greater length can then be constructed by traversing this cycle while taking shortcuts to avoid multiply visited vertices. A *shortcut* replaces a path between two cities by a direct edge between the two. By the triangle inequality the direct route cannot be longer than the path it replaces. Not only does this algorithm provide a better worst-case guarantee than any other currently known tour construction heuristic, it also tends to find better tours in practice, assuming care is taken in the choice of shortcuts. Its running time cost is substantial, however, compared to those for Nearest Neighbor, Greedy, and Clarke-Wright. This is primarily because the best algorithms currently available for its matching step take time $O(n^3)$, whereas none of the other three algorithms takes more than $O(n^2 logn)$ time. In theory this running time gap can be reduced somewhat: A modification of the Christofides-Serdukov algorithm with the same worst-case guarantee and an $O(n^{2.5})$ running time can be obtained by using a scaling based matching algorithm and halting once the matching is guaranteed to be no longer than $1 + (1/n)$ times optimal. As far as we know, however, this approach has never been implemented, and as we shall see, the competition from local search algorithms is sufficiently strong that the programming effort needed to do so would not be justified.

## 4.4. Lin-Kernighan algorithm

For over a decade and a half, from 1973 to about 1989, the world champion heuristic for the TSP was generally recognized to be the local search algorithm of Lin and Kernighan. This algorithm is both a generalization of 3-Opt and an outgrowth of ideas the same authors had previously applied to the graph partitioning problem, ideas that have much in common with tabu search. In this section we will give a more complete description of the one for the TSP.

An LK search is based on 2-Opt moves: replace two edges by two other edges to form another tour. The Lin-Kernighan heuristic allows the replacement of an arbitrary number of edges in moving from a tour to a neighboring tour, where again a complex greedy criterion is used in order to permit the search to go to an unbounded depth without an exponential blowup. We can sketch the basic idea as follows. Given a tour, we can remove an edge $(a, b)$ to obtain a Hamiltonian path with endpoints $a$ and $b$. Regard one of the endpoints as fixed, say $a$, and the other, $b$, as variable. If we add an edge $(b, c)$ from the variable endpoint $b$, a cycle is formed; there is an unique edge $(c, d)$ incident to $c$ whose removal breaks the cycle, producing a new Hamilton path with a new variable endpoint $d$. This operation is called rotation. We can always close a tour by adding the edge connecting the fixed endpoint $a$ with the current variable endpoint $d$. A move of the Lin-Kernighan heuristic from one tour to a neighbor consist of first removing an edge to form a Hamiltonian path, then performing a sequence of greedy rotations, and finally reconnecting the two endpoints to form a tour. There are different variants of this basic scheme depending on how exactly the rotations in each step is chosen, and on the restrictions on edges to enter and leave the tour. We define more precisely a variant which was shown some interesting theoretical properties about corresponding local optima.

Let $T$ be a tour. All 2-opt and 3-opt neighbors of $T$ are included in the LK neighborhood. In addition, for every pair of adjacent edges $x_1 = (a, b)$ from $T$ and $y_1 = (b, c)$ outside from $T$ with weights of these edges $w(y_1) < w(x_1)$, we have a sequence of neighboring tours $T_1, T_2, \dots$ defined as follows. Let $H = T - x_1$ be a Hamiltonian path with fixed endpoint $a$ and free endpoint $b$. Perform the first rotation that adds the edge $y_1$ and deletes a uniquely determined edge $x_2$ to form a new Hamiltonian path $H_1$, then close the path to obtain the first neighboring tour $T_1$. Edges $x_1$ and $x_2$ are marked "ineligible" and cannot reenter the tour during the sequence.

For $i$-th step, $i = 2, \dots$ consider all eligible edges $y_i \notin H_{i-1}$ incident to the variable endpoint such that $\sum_{j=1}^{i}[w(y_j) - w(x_j)] < 0$; if none exists, the sequence is terminated. If there are some such edges, choose among them that edge $y_i$ whose addition to $H_{i-1}$ and rotation leads to the least expensive new Hamilton path $H_i$ with ties broken lexicographically; i.e., $w(y_i) - w(x_{i+1})$ is minimized, where $x_{i+1}$ is the unique edge that has to be removed after adding $y_i$. Edge $x_{i+1}$ becomes ineligible. The $i$-th neighbor $T_i$ is the tour obtained by closing the path $H_i$. Observe that the sequence terminates after less than $n^2$ steps because once an edge leaves the tour, it becomes ineligible and cannot reenter. This concludes the formal description of the neighborhood LK.

The main difference with the original Lin-Kernighan definition is that an edge is allowed to enter the tour and later depart again, whereas in the original algorithm it cannot: the set of departing edges and entering edges are required there to be disjoint.

The simplest way to accelerate the Lin-Kernighan algorithm is to introduce the neighbor-list 3-Opt. More specifically, we proceed as in 3-Opt, considering all possibilities for $t_1, t_2, t_3, t_4$, and $t_5$ that satisfy

$$w(t_2, t_3) < w(t_1, t_2)$$

$$w(t_2, t_3) + w(t_4, t_5) < w(t_1, t_2) + w(t_3, t_4).$$

For each such choice, we use the tour that would result from performing the corresponding 3-Opt move as the starting point for an LK search. Note that for some choices of $t_1$ through $t_5$ there will be two possible ways of generating a corresponding 3-Opt move (two potential candidates for $t_6$ that yield legal 3- Opt moves), and both are considered. For other choices, there may be no way to generate a legal 3-Opt move, and in these cases we attempt to find cities $t_6$, $t_7$, and $t_8$ that together with $t_1$ through $t_5$ yield a legal 4-Opt move that meets the one-tree restriction (with $t_7$ restricted to those cities on the neighbor list for $t_6$). The first such move found is used to produce the starting tour. In addition, we preload the tabu lists for the LK search with the edges added by the 3- or 4-Opt move that yielded the initial path for the search.

The algorithm proceeds in a series of phases based on the notion of a *champion* tour, i.e., the best tour seen so far. Initially, this is just the tour produced by a starting heuristic, such as the Greedy algorithm. Until a new champion is crowned, all of our LK searches are based on tours obtained from this champion by 3-Opt (4-Opt) moves, with the search proceeding systematically through possible choices of $t_1$ through $t_5$ as in 3-Opt. Note that this method for restarting an LK search naturally embodies the tabu search concept of *intensification*, since it ensures that we keep exploring the vicinity of the current champion tour.

Whenever a tour is found that is better than the current champion, we complete the current LK search and then take as our new champion the best tour found during that search. If a given choice of $t_1$ through $t_4$ yields a legal 2-Opt move that improves the current champion, and none of the LK searches derived from this choice yields a better improving move, we take as our new champion the tour resulting from performing that 2-Opt move. Whenever a new champion is crowned, we enter a new phase based on this new champion, restarting the basic 3-Opt loop with the next available value for $t_1$. The Johnson implementation uses don't-look bits to restrict the choices for $t_1$, as in neighbor-list 2- and 3-Opt. The algorithm terminates when all possible choices of $t_1$ through $t_5$ have been considered for a given champion without yielding an improvement, i.e., when a tour is found that is locally optimal with respect to the expanded neighborhood structure implicit in the Lin-Kernighan algorithm itself.

Variants on this restart strategy have been considered in the literature, mostly with the idea of speeding up the algorithm. A common approach is to restrict attention to choices of $t_1$ through $t_6$ that yield valid 3-Opt moves and are such that $t_1$ through $t_4$ also yield a valid 2-Opt move. In addition, the number of LK searches made for each choice of $t_1$ can be more directly restricted. Whereas the Johnson et al. implementation considers both tour neighbors of $t_1$ as candidates for $t_2$, many implementations consider only one possibility for $t_2$, typically the successor of $t_1$ in the current tour. Given this, the ''no backtracking'' strategy of Mak and

Morton, simply starts an LK search as soon as $t_1$ has been chosen and its successor $t_2$ identified. Reinelt suggest considering only the first three possibilities for $t_3$ and starting an LK search as soon as $t_3$ has been chosen. This yields at most 3 LK searches for each choice of $t_1$. Mak and Morton suggest allowing alternatives for both $t_5$ and $t_3$, but only considering the first 5 choices for each (for a bound of 25 on the total number of LK searches for a given choice of $t_1$). In another variant, two options each are allowed for $t_3$, $t_5$ and $t_7$, yielding 8 possible LK searches for each choice of $t_1$. Only the more draconian of these approaches are likely to provide substantial speedups by themselves, however. With neighbor lists of length 20, the Johnson et al. implementation can theoretically perform 800 or more LK searches per choice of $t_1$, but the actual average number of calls is more like 6 for random Euclidean instances and 8 for the larger of our random distance matrix instances.

Another common modification that has been proposed is to limit the depth of LK searches, say to 50 steps. This again is unlikely to cause any significant speedup of the algorithm by itself, given that the average depth searched even when no bounds are imposed is only 3 moves beyond the level at which the LK search is initiated. It does however, enable use of the *Segment-Tree* data structure for tour representation proposed by Applegate, Chvatal, and Cook, which for certain classes of instances is a serious competitor to the two-level tree data structure used in the Johnson implementation.

Finally, there have been various proposals to modify the LK- search method more drastically, either by using shorter neighbor-lists to further limit the alternatives considered for $t_{2i+1}$, or conversely, by augmenting the class of possible moves considered. Mak and Morton suggest allowing 2-Opt moves that flip a prefix of the current path as well as the standard ones that flip a suffix. Reinelt and Rinaldi suggest allowing moves in which a single city is moved from its current position to the end of the path. Dam and Zachariasen have spelled out an even more flexible variant called a *flower transition.* In this search method, the base configuration is not a Hamiltonian path, as in LK search, but a one-tree consisting of a cycle (the *blossom*) attached to a path (the *stem*). Such a graph has more flexibility as there are two ways it can be turned into a tour, depending on which of the cycle edges adjacent to the stem is deleted. Individual steps of a search again involve adding an edge from the free end of the path, but now that edge may go either to another stem city or to a city in the cycle, and in the latter case there are again two choices for which edge to delete.

## 4.5 Metaheuristics

Unlike exact methods, metaheuristics allow to tackle large-size problem instances by delivering satisfactory solutions in a reasonable time. There is no guarantee to find global optimal solutions or even bounded solutions. Metaheuristics have received more and more popularity in the past 20 years. Their use in many applications shows their efficiency and effectiveness to solve large and complex problems.

It is unwise to use metaheuristics to solve problems where efficient exact algorithms are available. An example of this class of problems is the P class. In the case where those exact algorithms give "acceptable" search time to solve the target instances, metaheuristics are useless. For instance, one should not use a metaheuristic to find a minimum spanning tree or a shortest path in a graph. Polynomial-time exact algorithms exist for those problems. Hence, for easy optimization problems, metaheuristics are seldom used.

Many combinatorial optimization problems belong to the NP-hard class of problems. This high-dimensional and complex optimization class of problems arises in many areas of industrial concern: telecommunication, computational biology, transportation and logistics, planning and manufacturing, engineering design, and so on. Moreover, most of the classical optimization problems are NP-hard in their general formulation: traveling salesman, set covering, vehicle routing, graph partitioning, graph coloring, and so on. For an NP-hard problem where state-of-the-art exact algorithms cannot solve the handled instances within the required search time, the use of metaheuristics is justified. For this class of problems, exact algorithms require (in the worst case) exponential time. Nevertheless, the NP-completeness of a problem does not imply anything about the complexity of a particular class of instances that has to be solved.

Many classification criteria may be used for metaheuristics:

**Nature inspired versus nonnature inspired:** Many meta-heuristics are inspired by natural processes: evolutionary algorithms and artificial immune systems from biology; ants, bees colonies, and particle swarm optimization from swarm intelligence into different species (social sciences); and simulated annealing from physics.

**Memory usage versus memoryless methods:** Some metaheuristic algorithms are memoryless; that is, no information extracted dynamically is used during the search. Some representatives of this class are local search, GRASP, and simulated annealing. While other metaheuristics use a memory that contains some information extracted online during the search. For instance, short-term and long-term memories in tabu search.

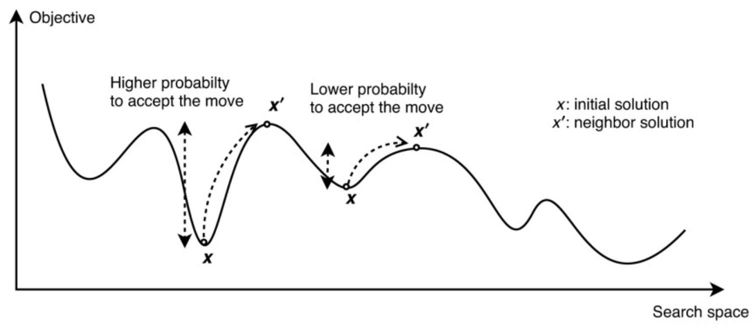
**Deterministic versus stochastic:** A deterministic metaheuristic solves an optimization problem by making deterministic decisions (e.g., local search, tabu search). In stochastic metaheuristics, some random rules are applied during the search (e.g., simulated annealing, evolutionary algorithms). In deterministic algorithms, using the same initial solution will lead to the same final solution, whereas in stochastic metaheuristics, different final solutions may be obtained from the same initial solution. This characteristic must be taken into account in the performance evaluation of metaheuristic algorithms.

**Population-based search versus single-solution based search:** Single-solution based algorithms (e.g., local search, simulated annealing) manipulate and transform a single solution during the search while in population-based algorithms (e.g., particle swarm, evolutionary algorithms) a whole population of solutions is evolved. These two families have complementary characteristics: single-solution based meta-heuristics are exploitation oriented; they have the power to intensify the search in local regions. Population-based metaheuristics are exploration oriented; they allow a better diversification in the whole search space. In the next chapters of this book, we have mainly used this classification. In fact, the algorithms belonging to each family of metaheuristics share may search mechanisms.

**Iterative versus greedy:** In iterative algorithms, we start with a complete solution (or population of solutions) and transform it at each iteration using some search operators. Greedy algorithms start from an empty solution, and at each step a decision variable of the problem is assigned until a complete solution is obtained. Most of the  metaheuristics are iterative algorithms. Below we present some of them.

### 4.4.1. Simulated annealing

Simulated annealing applied to optimization problems emerges from the work of S. Kirkpatrick et al. and V. Cerny. In these pioneering works, SA has been applied to graph partitioning and VLSI design. In the 1980s, SA had a major impact on the field of heuristic search for its simplicity and efficiency in solving combinatorial optimization problems. Then, it has been extended to deal with continuous optimization problems. SA is based on the principles of statistical mechanics whereby the annealing process requires heating and then slowly cooling a substance to obtain a strong crystalline structure. The strength of the structure depends on the rate of cooling metals. If the initial temperature is not sufficiently high or a fast cooling is applied, imperfections (metastable states) are obtained. In this case, the cooling solid will not attain thermal equilibrium at each temperature. Strong crystals are grown from careful and slow cooling. The SA algorithm simulates the energy changes in a system subjected to a cooling process until it converges to an equilibrium state (steady frozen state). This scheme was developed in 1953 by Metropolis.



Objective
Higher probabilty to accept the move
x'
x
Lower probabilty to accept the move
x'
x
x: initial solution
x': neighbor solution
Search space

Fig.4.1 Behavior of SA

SA is a stochastic algorithm that enables under some conditions the degradation of a solution. The objective is to escape from local optima and so to delay the convergence. SA is a memoryless algorithm in the sense that the algorithm does not use any information gathered during the search. From an initial solution, SA proceeds in several iterations. At each iteration, a random neighbor is generated. Moves that improve the cost function are always accepted. Otherwise, the neighbor is selected with a given probability that depends on the current temperature and the amount of degradation of the objective function. As the algorithm progresses, the probability that such moves are accepted decreases. This probability follows, in general, the Boltzmann distribution. It uses a control parameter, called temperature, to determine the probability of accepting nonimproving solutions. At a particular level of temperature, many trials are explored. Once an equilibrium state is reached, the temperature is gradually decreased according to a cooling schedule such that few nonimproving solutions are accepted at the end of the search (see Fig.4.1).

The SA algorithm is described below:

**SA Algorithm**

```
1.  Input: Cooling schedule: $x := x_0$ ; $T := T_{\max}$ ;
2.  Repeat Until Stopping criteria satisfied e.g. $T < T_{\min}$
    2.1. Repeat Until Equilibrium condition
         2.1.1. Generate a random neighbor $s'$
         2.1.2. $\Delta E = f(x') - f(x)$;
         2.1.3. If $\Delta E \leq 0$ Then $x := x'$
    Else Accept $x'$ with a probability $e^{-\Delta E/T}$
    2.2. $T = g(T)$;
3.  Output: Best solution found.
```

In the SA algorithm, the temperature is decreased. There is always a compromise between the quality of the obtained solutions and the speed of the cooling schedule. If the temperature is decreased slowly, better solutions are obtained but with a more significant computation time. The temperature can be updated in different ways:

• **Linear:** In the trivial linear schedule, the temperature is updated as follows: $T = T - \beta$, where $\beta$ is a specified constant value.

Hence, we have

$$T_i = T_0 - i \times \beta$$

where $T_i$ represents the temperature at iteration $i$.

• **Geometric:** In the geometric schedule, the temperature is updated using the formula

$$T = \alpha T,$$

where $\alpha \in ]0, 1[$.

It is the most popular cooling function. Experience has shown that $\alpha$ should be between 0.5 and 0.99.

• **Logarithmic:** The following formula is used:

$$T_i = \frac{T_0}{\log(i)}$$

This schedule is too slow to be applied in practice but has the property of the convergence proof to a global optimum.

• **Very slow decrease:** The main trade-off in a cooling schedule is the use of a large number of iterations at a few temperatures or a small number of iterations at many temperatures. A very slow cooling schedule such as $T_i + 1 = T_i/(1 + \beta T_i)$, where $\beta = T_0 - \frac{T_F}{L-1} T_0 T_F$ and $T_F$ is the final temperature.

Only one iteration is allowed at each temperature in this very slow decreasing function.

• **Nonmonotonic:** Typical cooling schedules use monotone temperatures. Some nonmonotone scheduling schemes where the temperature is increased again may be suggested. This will encourage the diversification in the search space. For some types of search landscapes, the optimal schedule is nonmonotone.

• **Adaptive:** Most of the cooling schedules are static in the sense that the cooling schedule is defined completely *a priori*. In this case, the cooling schedule is "blind" to the characteristics of the search landscape. In an adaptive cooling schedule, the decreasing rate is dynamic and depends on some information obtained during the search. A dynamic cooling schedule may be used where a small number of iterations are carried out at high temperatures and a large number of iterations at low temperatures.

Concerning the stopping condition, theory suggests a final temperature equal to 0. In practice, one can stop the search when the probability of accepting a move is negligible. The following stopping criteria may be used:

• Reaching a final temperature is the most popular stopping criteria.

• Achieving a predetermined number of iterations without improvement of the best found solution.

• Achieving a predetermined number of times a percentage of neighbors at each temperature is accepted; that is, a counter increases by 1 each time a temperature is completed with the less percentage of accepted moves than a predetermined limit and is reset to 0 when a new best solution is found. If the counter reaches a predetermined limit, the SA algorithm is stopped.

SA compared to local search is still simple and easy to implement. It gives good results for a wide spectrum of optimization problems: the historical ones such as TSP and VLSI design in different domains of application. A good survey on SA can be found in [ ].

Other similar methods of simulated annealing have been proposed in the literature, such as threshold accepting, great deluge algorithm, record-to-record travel, and demon algorithms. The main objective in the design of those SA-inspired algorithms is to speed up the search of the SA algorithm without sacrificing the quality of solutions.

***Record-to-Record Travel*** This algorithm is also a deterministic optimization algorithm inspired from simulated annealing [229]. The algorithm accepts a nonimproving neighbor solution with an objective value less than the RECORD minus a deviation D. RECORD represents the best objective value of the visited solutions during the search. The bound decreases with time as the objective value RECORD of the best found solution improves. The RRT algorithm is described below:

**RRT Algorithm**

```
1. Input: Deviation $D > 0$; $x := x_0$; $RECORD := f(x)$ ;
2. Repeat Until Stopping criteria satisfied
      2.1. Generate a random neighbor $x'$;
      2.2. If $f(x') < RECORD + D$ Then $x := x'$;
      2.3. If $RECORD > f(x')$ Then $RECORD := f(x')$ ;
3. Output: Best solution found.
```

The RRT algorithm has the advantage to be dependent on only one parameter, the DEVIATION value. A small value for the deviation will produce poor results within a reduced search time. If the deviation is high, better results are produced after an important computational time.

***Great Deluge Algorithm*** The great deluge algorithm was proposed by Dueck in 1993. The main difference with the SA algorithm is the deterministic acceptance function of neighboring solutions. The inspiration of the GDA algorithm comes from the analogy that the direction a hill climber would take in a great deluge to keep his feet dry. Finding the global optimum of an optimization problem may be seen as finding the highest point in a landscape. As it rains incessantly without end, the level of the water increases. The algorithm never makes a move beyond the water level. It will explore the uncovered area of the landscape to reach the global optimum. GDA describes the template of the algorithm in a minimization context. A generated neighbor solution is accepted if the absolute value of the objective function is less than the current boundary value, named level. The initial value of the level is equal to the initial objective function. The level parameter in GDA operates somewhat like the temperature in SA. During the search, the value of the level is decreased monotonically. The decrement of the reduction is a parameter of the algorithm.

**GD Algorithm**

```
1. Input: Level $L$
   $x := x_0$ ;
   Choose the rain speed $UP$;
   Choose the initial water level $LEVEL$;
2. Repeat Until Stopping criteria satisfied
   2.1. Generate a random neighbor $x'$ ;
   2.2. If $f(x') < LEVEL$ Then $x := x'$
        $LEVEL := LEVEL - UP$ ;
3. Output: Best solution found.
```

The great deluge algorithm needs the tuning of only one parameter, the UP value that represents the rain speed. The quality of the obtained results and the search time will depend only on this parameter. If the value of the UP parameter is high, the algorithm will be fast but will produce results of poor quality. Otherwise, if the UP value is small, the algorithm will generate relatively better results within a higher computational time. An example of a rule that can be used to define the value of the UP parameter may be the following: a value smaller than 1% of the average gap between the quality of the current solution and the water level.

***Demon Algorithms*** Since 1998 many metaheuristics based on the demon algorithm (DA) have been proposed. The demon algorithm is another simulated annealing-based algorithm that uses computationally simpler acceptance functions. The acceptance function is based on the energy value of the demon (credit). The demon is initialized with a given value D. A nonimproved solution is accepted if the demon has more energy (credit) than the decrease of the objective value. When a DA algorithm accepts a solution of increased objective value, the change value of the objective is credited to the demon. In the same manner, when a DA algorithm accepts an improving solution, the decrease of the objective value is debited from the demon.

**DA Algorithm**

```
1. Input: Demon initial value $D$,  $x := x_0$ ;
2. Repeat until stopping criteria satisfied
    2.1 Generate a random neighbor $'$;
    2.2 $E := f(x') - f(x)$;
    2.3 If $E \leq D$ Then $x := x'$ ;
    2.4 $D := D - E$ ;
3. Output: Best solution found.
```

The acceptance function of demon algorithms is computationally simpler than in SA. It requires a comparison and a subtraction, whereas in SA it requires an exponential function and a generation of a random number. Moreover, the demon values vary dynamically in the sense that the energy (credit) depends on the visited solutions (Markov chain) during the search, whereas in SA and TA the temperature (threshold) is not dynamically reduced. Indeed, the energy absorbed and released by the demon depends mainly on the accepted solutions. Different variants of the DA algorithm can be found in the literature. They differ by the annealing schedule of the acceptance function:

• **Bounded demon algorithm:** This algorithm imposes an upper bound $D_0$ for the credit of the demon. Hence, once the credit of the demon is greater than the upper bound, no credit is received even if improving solutions are generated.

• **Annealed demon algorithm:** In this algorithm, an annealing schedule similar to the simulated annealing one is used to decrease the credit of the demon. The credit of the demon will play the same role as the temperature in simulated annealing.

• **Randomized bounded demon algorithm:** A randomized search mechanism is introduced in the BDA algorithm. The credit of the demon is replaced with a normal Gaussian random variable, where the mean equals the credit of the demon ($D_m$), and a specified standard deviation $Dsd$. Hence, the energy associated with the demon will be $D := D_m + Gaussian\, noise$.

• **Randomized annealed demon algorithm:** The same randomized search mechanism of the RBDA algorithm is introduced as in the ADA algorithm. Compared to simulated annealing, the application of demon algorithms to academic and real-life problems show competitive quality of results within a reduced search time. Moreover, they are very easy to design and implement and they need tuning of few parameters.

### 4.4.2. Tabu Search

Tabu search algorithm was proposed by Glover. In 1986, he pointed out the controlled randomization in SA to escape from local optima and proposed a deterministic algorithm. In a parallel work, a similar approach named "steepest ascent/mildest descent" has been proposed by Hansen. In the 1990s, the tabu search algorithm became very popular in solving optimization problems in an approximate manner. Nowadays, it is one of the most widespread metaheuristics. The use of memory, which stores information related to the search process, represents the particular feature of tabu search.

TS behaves like a steepest LS algorithm, but it accepts nonimproving solutions to escape from local optima when all neighbors are nonimproving solutions. Usually, the whole neighborhood is explored in a deterministic manner, whereas in SA a random neighbor is selected. As in local search, when a better neighbor is found, it replaces the current solution. When a local optima is reached, the search carries on by selecting a candidate worse than the current solution. The best solution in the neighborhood is selected as the new current solution even if it is not improving the current solution. Tabu search may be viewed as a dynamic transformation of the neighborhood. This policy may generate cycles; that is, previous visited solutions could be selected again.

To avoid cycles, TS discards the neighbors that have been previously visited. It memorizes the recent search trajectory. Tabu search manages a memory of the solutions or moves recently applied, which is called the *tabu list*. This tabu list constitutes the short-term memory. At each iteration of TS, the short-term memory is updated. Storing all visited solutions is time and space consuming. Indeed, we have to check at each iteration if a generated solution does not belong to the list of all visited solutions. The tabu list usually contains a constant number of tabu moves. Usually, the attributes of the moves are stored in the tabu list.

By introducing the concept of solution features or move features in the tabu list, one may lose some information about the search memory. We can reject solutions that have not yet been generated. If a move is "good," but it is tabu, do we still reject it? The tabu list may be too restrictive; a nongenerated solution may be forbidden. Yet for some conditions, called *aspiration criteria*, tabu solutions may be accepted. The admissible neighbor solutions are those that are nontabu or hold the aspiration criteria.

In addition to the common design issues for metaheuristics such as the definition of the neighborhood and the generation of the initial solution, the main design issues that are specific to a simple TS are

• **Tabu list:** The goal of using the short-term memory is to prevent the search from revisiting previously visited solutions. As mentioned, storing the list of all visited solutions is not practical for efficiency issues.

• **Aspiration criterion:** A commonly used aspiration criteria consists in selecting a tabu move if it generates a solution that is better than the best found solution. Another aspiration criterion may be a tabu move that yields a better solution among the set of solutions possessing a given attribute.

Some advanced mechanisms are commonly introduced in tabu search to deal with the intensification and the diversification of the search:

• **Intensification (medium-term memory):** The medium-term memory stores the elite (e.g., best) solutions found during the search. The idea is to give priority to attributes of the set of elite solutions, usually in weighted probability manner. The search is biased by these attributes.

• **Diversification (long-term memory):** The long-term memory stores information on the visited solutions along the search. It explores the unvisited areas of the solution space. For instance, it will discourage the attributes of elite solutions in the generated solutions to diversify the search to other areas of the search space. The TS algorithm is described below:

**TS Algorithm**

```
1. Input: Initialize starting solution $x := x_0$ and the tabu list, medium-term and long-term memories;
2. Repeat until stopping criteria satisfied
    1.1. Find best admissible neighbor $x'$ ;
    1.2. $x := x'$ ;
    1.3. Update tabu list, aspiration conditions, medium and long term memories ;
    1.4. If intensification criterion holds Then intensification ;
    1.5. If diversification criterion holds Then diversification ;
3. Output: Best solution found.
```

Theoretical studies carried out on tabu search algorithms are weaker than those established for simulated annealing. A simulated annealing execution lies within the convex hull of a set of tabu search executions. Therefore, tabu search may inherit some nice theoretical properties of SA. In addition to the search components of local search (hill climbing), such as the representation, neighborhood, initial solution, we have to define the following concepts that compose the search memory of TS: the tabu list (short-term memory), the medium-term memory, and the long-term memory.

**Short-Term Memory** The role of the short-term memory is to store the recent history of the search to prevent cycling. The naive straightforward representation consists in recording all visited solutions during the search. This representation ensures the lack of cycles but is seldom used as it produces a high complexity of data storage and computational time. For instance, checking the presence of all neighbor solutions in the tabu list will be prohibitive. The first improvement to reduce the complexity of the algorithm is to limit the size of the tabu list. If the tabu list contains the last $k$ visited solutions, tabu search prevents a cycle of size at most $k$. Using hash codes may also reduce the complexity of the algorithms manipulating the list of visited solutions. In general, attributes of the solutions or moves are used. This representation induces a less important data storage and computational time but skips some information on the history of the search. For instance, the absence of cycles is not ensured.

The most popular way to represent the tabu list is to record the move attributes. The tabu list will be composed of the reverse moves that are forbidden. This scheme is directly related to the neighborhood structure being used to solve the problem. If the move is applied to the solution to generate new solution, then the move (or its reverse) is stored in the list. This move is forbidden for a given number of iterations, named the *tabu tenure* of the move. If the tabu list contains the last $k$ moves, tabu search will not guarantee to prevent a cycle of size at most $k$.

In tabu search, many different tabu lists may be used in conjunction. For instance, some ingredients of the visited solutions and/or the moves are stored in multiple tabu lists. A move is not tabu if the conditions for *all* tabu lists are satisfied. In many optimization problems, it is more and more popular to use simultaneously various move operators, and hence different neighborhoods are defined.

The size of the tabu list is a critical parameter that has a great impact on the performances of the tabu search algorithm. At each iteration, the last move is added in the tabu list, whereas the oldest move is removed from the list. The smaller is the value of the tabu list, the more significant is the probability of cycling. Larger values of the tabu list will provide many restrictions and encourage the diversification of the search as many moves are forbidden. A compromise must be found that depends on the landscape structure of the problem and its associated instances. The tabu list size may take different forms:

• **Static:** In general, a static value is associated with the tabu list. It may depend on the size of the problem instance and particularly the size of the neighborhood. There is no optimal value of the tabu list for all problems or even all instances of a given problem. Moreover, the optimal value may vary during the search progress. To overcome this problem, a variable size of the tabu list may be used.

• **Dynamic:** The size of the tabu list may change during the search without using any information on the search memory.

• **Adaptive:** In the adaptive scheme, the size of the tabu list is updated according to the search memory. For instance, the size is updated upon the performance of the search in the last iterations.

**Medium-Term Memory** The role of the intensification is to exploit the information of the best found solutions (elite solutions) to guide the search in promising regions of the search space. This information is stored in a medium-term memory. The idea consists in extracting the (common) features of the elite solutions and then intensifying the search around solutions sharing those features. A popular approach consists in restarting the search with the best solution obtained and then fixing in this solution the most promising components extracted from the elite solutions.

The main representation used for the medium-term memory is the *recency memory*. First, the components associated with a solution have to be defined; this is a problem specific task. The recency memory will memorize for each component the number of successive iterations the component is present in the visited solutions. The most commonly used event to start the intensification process is a given period or after a certain number of iterations without improvement.

The intensification of the search in a given region of the search space is not always useful. The effectiveness of the intensification depends on the landscape structure of the target optimization problem. For instance, if the landscape is composed of many basins of attraction and a simple TS without intensification component is effective to search in each basin of attraction, intensifying the search in each basin is useless.

**Long-Term Memory** As mentioned many times, metaheuristics are more powerful search methods in terms of intensification. Long-term memory has been introduced in tabu search to encourage the diversification of the search. The role of the long-term memory is to force the search in nonexplored regions of the search space. The main representation used for the long-term memory is the *frequency memory*. As in the recency memory, the components associated with a solution have first to be defined. The frequency memory will memorize for each component the number of times the component is present in all visited solutions. The diversification process can be applied periodically or after a given number of iterations without improvement. Three popular diversification strategies may be applied:

• **Restart diversification:** This strategy consists in introducing in the current or best solution the least visited components. Then a new search is restarted from this new solution.

• **Continuous diversification:** This strategy introduces during a search a bias to encourage diversification. For example, the objective function can integrate the frequency occurrence of solution components in the evaluation of current solutions. Frequently applied moves or visited solutions are penalized.

• **Strategic oscillation:** Introduced by Glover in 1989, strategic oscillation allows to consider (and penalize) intermediate solutions that are infeasible. This strategy will guide the search toward infeasible solutions and then come back to a feasible solution.

As for the intensification, the diversification of the search is not always useful. It depends on the landscape structure of the target optimization problem. For instance, if the landscape is a "massif central" where all good solutions are localized in the same region of the search space within a small distance, diversifying the search to other regions of the search space is useless. The search time assigned to the diversification and the intensification components of TS must be carefully tuned depending on the characteristics of the landscape structure associated with the problem. TS has been successfully applied to many optimization problems. Compared to local search and simulated annealing, various search components of TS are problem specific and must be defined. The search space for TS design is much larger than for local search and simulated annealing. The degree of freedom in designing the different ingredients of TS is important. The representation associated with the tabu list, the medium-term memory, and the long-term memory must be designed according to the characteristics of the optimization problem at hand. This is not a straightforward task for some optimization problems. Moreover, TS may be very sensitive to some parameters such as the size of the tabu list.

### 4.4.3. Variable neighborhood search

Variable neighborhood search has been recently proposed by P. Hansen and N. Mladenovic. The basic idea of VNS is to successively explore a set of predefined neighborhoods to provide a better solution. It explores either at random or systematically a set of neighborhoods to get different local optima and to escape from local optima. VNS exploits the fact that using various neighborhoods in local search may generate different local optima and that the global optima is a local optima for a given neighborhood. Indeed, different neighborhoods generate different landscapes (see Fig.4.2).

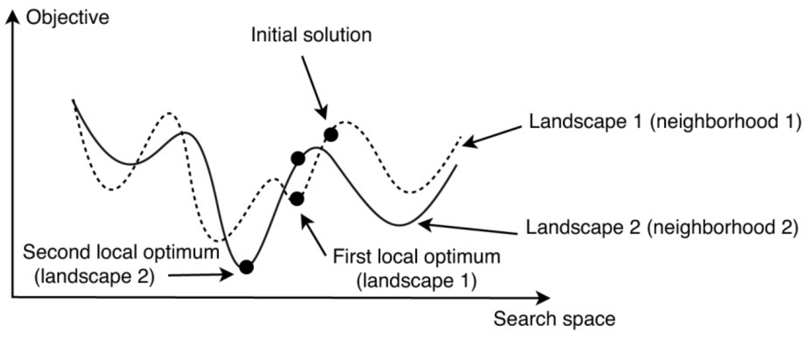


**Fig. 4.2** Behavior of VNS

**Variable Neighborhood Descent** The VNS algorithm is based on the variable neighborhood descent, which is a deterministic version of VNS. VND uses successive neighborhoods in descent to a local optimum. First, one has to define a set of neighborhood structures $N_l$ $(l = 1, \dots, l_{\max})$. Let $N_1$ be the first neighborhood to use and $x$ the initial solution. If an improvement of the solution $x$ in its current neighborhood $N_l(x)$ is not possible, the neighborhood structure is changed from $N_l$ to $N_{l+1}$. If an improvement of the current solution $x$ is found, the neighborhood structure returns to the first one $N_1(x)$ to restart the search. This strategy will be effective if the different neighborhoods used are complementary in the sense that a local optima for a neighborhood $N_i$ will not be a local optima in the neighborhood $N_j$.

**VND Algorithm**

```
1. Input: a set of neighborhood structures $N_l$ for $l = 1, \ldots, l_{\max}$
   $x := x_0$ ; $l := 1$.
2. While $l \leq l_{\max}$ Do
   2.1. Find the best neighbor $x'$ of $x$ in $N_l(x)$;
   2.2. If $f(x') < f(x)$ Then $x := x'$; $l := 1$;
   2.3. Otherwise $l := l + 1$;
3. Output: Best solution found.
```

The design of the VND algorithm is mainly related to the selection of neighborhoods and the order of their application. The complexity of the neighborhoods in terms of their exploration and evaluation must be taken into account. The larger are the neighborhoods, the more time consuming is the VND algorithm. Concerning the application order, the most popular strategy is to rank the neighborhoods following the increasing order of their complexity.

**General Variable Neighborhood Search** VNS is a stochastic algorithm in which, first, a set of neighborhood structures are defined. Then, each iteration of the algorithm is composed of three steps: shaking, local search, and move. At each iteration, an initial solution is shaked from the current neighborhood $N_k$. For instance, a solution $x'$ is generated randomly in the current neighborhood.

A local search procedure is applied to the solution $x'$ to generate the solution $x''$. The current solution is replaced by the new local optima $x''$ if and only if a better solution has been found. The same search procedure is thus restarted from the solution $x''$ in the first neighborhood. If no better solution is found, the algorithm moves to the next neighborhood, randomly generates a new solution in this neighborhood, and attempts to improve it. Let us notice that cycling is possible.

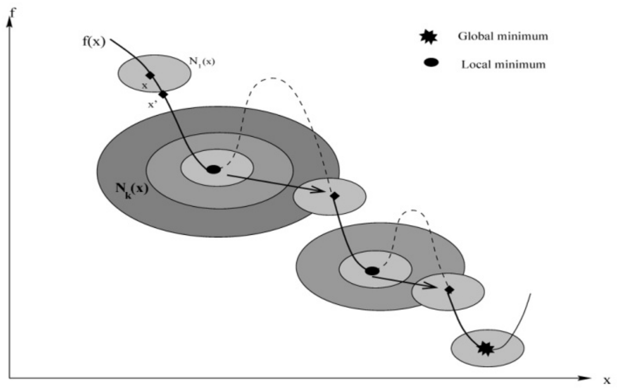


Fig.4.3. Idea of VNS

**VNS Algorithm**

```
1. Input: a set of neighborhood structures $N_k$ for $k = 1, \dots, k_{max}$
   $x = x_0$ ;
2. Repeat until stopping criteria
   2.1. $k = 1$ ;
   2.2. Repeat Until $k = k_{max}$
        2.2.1. Shaking: pick a random solution $x'$ from the $k^{\text{th}}$ neigh-
               borhood $N_k(x)$ of $x$ ;
        2.2.2. $x'' :=$ local search $(x')$ ;
        2.2.3. If $f(x'') < f(x)$
               Then $x := x''$; Continue to search with $N_1$; $k := 1$ ;
               Otherwise $k := k + 1$ ;
3. Output: Best solution found.
```

In addition to the design of a simple local search, the design of the VNS algorithm is mainly related to the selection of neighborhoods for the shaking phase. Usually, nested neighborhoods are used, where each neighborhood contains the previous one. A compromise must be found between intensification of the search and its diversification through the distribution of work between the local search phase and the shaking phase. An increased work in the local search phase will generate better local optima (more intensification), whereas an increased work in the shaking phase will lead to potentially better regions of the search space (more diversification).

As in local search, VNS requires a small number of parameters. For the shaking phase, the single parameter is the number of neighborhood structures. If large values are used (i.e., very large neighborhoods are considered), VNS will be similar to a multi-start local search. For small values, VNS will degenerate to a simple local search algorithm.

### 4.4.4 GRASP

The GRASP metaheuristic is an iterative greedy heuristic to solve combinatorial optimization problems. It was introduced in 1989. Each iteration of the GRASP algorithm contains two steps: construction and local search. In the construction step, a feasible solution is built using a randomized greedy algorithm, while in the next step a local search heuristic is applied from the constructed solution. A similar idea, known as the *semigreedy heuristic*, was presented in 1987, where a multi-start greedy approach is proposed but without the use of local search. The greedy algorithm must be randomized to be able to generate various solutions. Otherwise, the local search procedure can be applied only once. This schema is repeated until a given number of iterations and the best found solution are kept as the final result. We notice that the iterations are completely independent, and so there is no search memory. This approach is efficient if the constructive heuristic samples different promising regions of the search space that makes the different local searches generating different local optima of "good" quality. In the GRASP algorithm, the *seed* is used as the initial seed for the pseudorandom number generator.

**GRASP Algorithm**

```
1. **Input:** Number of iterations.
2. **Repeat** until stopping criteria
    2.1. $x := Random\ Greedy(seed)$ ;
    2.2. $x' := Local\ \ Search(x)$ ;
3. **Output:** Best solution found.
```

The main design questions for GRASP are the greedy construction and the local search procedures:

• **Greedy construction:** In the constructive heuristic, as mentioned, at each iteration the elements that can be included in the partial solution are ordered in the list (decreasing values) using the local heuristic. From this list, a subset is generated that represents the *restricted candidate list*. The RCL list is the key component of the GRASP meta-heuristic. It represents the probabilistic aspect of the metaheuristic. The restriction criteria may depend on

**Cardinality-based criteria:** In this case, the RCL list is made of the $p$ best elements in terms of the incremental cost, where the parameter $p$ represents the maximum number of elements in the list.

**Value-based criteria:** It is the most commonly used strategy. It consists in selecting the solutions that are better than a given threshold value.

At each iteration, a random element is picked from the list RCL. Once an element is incorporated in the partial solution, the RCL list is updated. To update the RCL list, the incremental costs of each element composing the RCL list must be reevaluated.

• **Local search:** Since the solutions found by the construction procedure are not guaranteed to be local optima, it is beneficial to carry out a local search step in which the constructed solution is improved. Traditionally, a simple local search algorithm is applied. Nevertheless, any metaheuristic can be used: tabu search, simulated annealing, noisy meth-od, and so on.

## 4.6. Exercises

1. Determine whether the following statements are correct or not

    (1) If a problem is NP-complete, then it cannot be solved by any polynomial algorithm in worst cases.

    (2) If a problem is NP-complete, then we have not found any polynomial algorithm to solve it in worst cases.

    (3) If a problem is NP-complete, then it is unlikely that a polynomial algorithm can be found in the future to solve it in worst cases.

    (4) If a problem is NP-complete, then it is unlikely that we can find a polynomial algorithm to solve it in average cases.

    (5) If we can prove that the lower bound of an NP-complete problem is exponential, then we have proved that P≠NP.

2. We know how to prove that a problem is NP-complete. How can we prove that a problem is not NP-complete.

3. Consider the following problem. Given two input variables $a$ and $b$, return "Yes" if $a>b$ and "No" if otherwise. Design a non-deterministic polynomial algorithm to solve this problem. Transform it into a Boolean formula such that the algorithm returns "Yes" if and only if the transform Boolean formula is satisfiable.

4. Maximal clique decision problem: A maximal clique is a maximal complete subgraph of a graph. The size of a maximal clique is the number of vertices in it. The clique decision problem is to determine whether there is a maximal clique at least size $k$ for some $k$ in a graph or not. Show that the maximal clique decision problem is NP-complete by reducing the satisfiability problem to it.

5. Traveling salesman decision problem: Show that the traveling salesman decision problem is NP-complete by proving that the Hamilton cycle decision problem reduces polynomially to it.
6. Bottleneck traveling salesman decision problem: Given a graph and a number M, the bottleneck traveling salesman decision problem is to determine whether there exist a Hamiltonian cycle in this graph such that the longest edge of this cycle is less that M. Show that the bottleneck traveling decision problem is NP-complete.
7. Read the book of Papadimitriou and Steiglitz for the NP-completeness of the 3-dimential matching problem.
8. Let there be a set of points densely distributed on a circle. Apply the approximation Euclidean traveling salesman algorithm to find an approximation tour for this set of points. Is this result also an optimal one?
9. Consider the following bottleneck optimization problem: We are given a set of points in the plane and we are asked to find *k* points such that among these *k* points, the shortest distance is maximized. This problem is NP-hard. Design a Lin-Kernighan type heuristic for this problem.
10. Consider the bottleneck traveling salesman problem in Euclidean plane. Design the SA, VNS, GD, TS, and GRASP meta-heuristics for this NP-hard problem.

# Bibliography

1. Christofides N. Graph theory. An algorithmic approach. New York, London, San Francisco: Academic press, 1975.
2. Wichmann B.A. Ackermann's function: a study in the efficiency of calling procedures // BIT, 1976, 16, 103-110.
3. Monma C., Suri S. Transitions in geometric minimum spanning trees // Discrete ans Computational Geometry, 1992, 8(3), 265-293.
4. Garey M.R., Johnson D.S. Computers and intractability. A guide to the theory of NP-completeness. San Francisco: W.H. Freeman and company, 1979.
5. Demet'ev V. T., Erzin A. I., Larin R. M., Shamardin Yu. V. The optimization problem of hierarchical structures. Novosibirsk: Publishing House of Novosibirsk University, 1996. (in Russian)
6. Robins G., Zelikovsky A. Improved Steiner tree approximation in graphs // Proc. 10th Ann. ACM-SIAM Symp. on Discrete Algorithms, ACM-SIAM, 2000, 770-779.
7. Hanan M. On Steiner's problem with rectilinear distance // SIAM J. Applied Math., 1966, 14, 255–265.
8. Hwang F. K. On steiner minimal trees with rectilinear distance // SIAM J. Applied Math., 1976, 30 (1), 104–114.
9. Zelikovsky A. Z. An 11/8-approximation algorithm for the Steiner problem on networks with rectilinear distance // Janos Bolyai Mathematica Societatis Conf.: Sets, Graphs, and Numbers, 1992, 733–745.
10. Erzin A. I., Cho J. D. A Deep-Submicron Steiner Tree // Mathematical and Computer Modelling, 2000, 31, 215-226.

11. Erzin A. I., Cho J. D. Skew Minimization Problem with Possible Sink Displacement // Automation and Remote Control, 2003, 64(3), 493-504

12. Korte B, Vigen J. Combinatorial optimization. Berlin: Springer, 2007.

13. Deineko V., Tiskin A. Fast minimum-weight double-tree shortcutting for metric TSP: is the best one good enough? // ACM Journal on Experimental Algorithmics. 2009. Vol. 14. N 4.6.

14. Papadimitriou, C. H. and Steiglitz K. Combinatorial Optimization. Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

15. Yannakakis M. Computational complexity. In: Aarts E., Lenstra J. (Eds.) Local search in combinatorial optimization. NY:Wiley, 1997.

16. Grover L.K. Local search and the local structure of NP-complete problems // Operations Research Letters. 1992. Vol. 12, N 4. P. 235–244.

17. Angel E., Zissimopoulos V. On the classification of NP-complete problems in terms of their correlation coefficient // Discrete Appl. Math. 2. V. 9. P. 261–277.

18. Burkard R.E., Deineko V.G., Woeginger G.J. The traveling salesman problem and the PQ-tree // Lecture Notes in Computer Science, Vol. 1084. Berlin: Springer, 1996. P. 490–504.

19. Gutin G. Exponential neighborhood local search for the traveling salesman problem // Comput. Oper. Res. 1999. Vol. 26. P. 313–320.

20. Gutin G., Yeo A. Small diameter neighborhood graphs for the traveling salesman problem: four moves from tour to tour // Comput. Oper. Res. 1999. Vol. 26. P. 321–327.

21. Talbi El-G. Metaheuristics: From Design to Implementation (Wiley Series on Parallel and Distributed Computing) Wiley, 2009.

22. Kirkpatrick, S. Gelatt, C.D. Vecchi M.P. Optimization by simulated annealing // Science. 1983. Vol. 220, P.671–680.

23. Glover, F. Laguna M. Tabu Search. Dordrecht: Kluwer Acad. Publ., 1997.

24. Mladenović, N. Hansen P. Variable neighbourhood search, Computers and Operations Research, 1997. Vol. 24. P. 1097–1100.

25. Wolsey L. A. Integer Programming. N. Y.: John Wiley & Sons, Inc, 1998.

26. Erzin A. I. Introduction to Operations Research: Textbook. Novosibirsk: Novosibirsk State University, 2006. (in Russian)

Textbook

**Erzin Adil Ilyasovich, Kochetov Yury Andreevich**

Routing problems

Prepared to print by S.V. Isakova

Signed to print: 24.06.2015
Format 60×84 1/16. Publ. sh. 5,6. Cond. publ. sh. 5,2.
Circulation 200 copies.
Order N

Editorial and Publishing Center, NSU
630090, Novosibirsk, Pirogova str., 2.