ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Προσεγγιστικοί Αλγόριθμοι και Προσεγγιστικά Σχήματα για ειδικές περιπτώσεις του Προβλήματος του Πλανόδιου Πωλητή (Traveling Salesman Problem)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Γιαννακάκης

Επιβλέπουσα: Φώτω Αφράτη
Καθηγήτρια Ε.Μ.Π.

Αθήνα, Ιούνιος 2006

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Προσεγγιστικοί Αλγόριθμοι και Προσεγγιστικά Σχήματα για ειδικές περιπτώσεις του Προβλήματος του Πλανόδιου Πωλητή (Traveling Salesman Problem)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Ν. Γιαννακάκης

Επιβλέπουσα: Φώτω Αφράτη
Καθηγήτρια Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 9η Ιουνίου 2006.

.....	.....	.....
Φώτω Αφράτη	Ευστάθιος Ζάχος	Γεώργιος Κολέτσος
Καθηγήτρια Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2006

.....
Ιωάννης Ν. Γιαννακάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ο σκοπός της παρούσας διπλωματικής εργασίας είναι η μελέτη προσεγγιστικών αλγορίθμων και προσεγγιστικών σχημάτων που επιλύουν σε πολυωνυμικό χρόνο κάποιες ειδικες περιπτώσεις του NP-complete Προβλήματος του πλανόδιου πωλητή (Traveling Salesman Problem, TSP). Η εργασία ξεκινά με μια εισαγωγή στη θεωρία της πολυπλοκότητας, τη θεωρά γράφων και τους προσεγγιστικούς αλγορίθμους. Ορίζεται στη συνέχεια το πρόβλημα στη γενική του μορφή (general TSP), και ακολουθεί ο ορισμός του Metric TSP (MTSP), με παρουσίαση και ανάλυση προσεγγιστικών αλγορίθμων που το επιλύουν σε πολυωνυμικό χρόνο. Ακολουθεί η περίπτωση του Ευκλείδιου TSP (Euclidean TSP), για το οποίο δίνεται ένα πολυωνυμικό προσεγγιστικό σχήμα (PTAS). Η τελευταία περίπτωση που μελετάται είναι το (1,2)-TSP, για το οποίο δίνονται οι πιο σημαντικοί προσεγγιστικοί αλγόριθμοι, μεταξύ των οποίων και ένας πολύ πρόσφατος (2005). Η εργασία ολοκληρώνεται με την παράθεση κάποιων σημαντικών πρακτκών εφαρμογών του TSP.

Λέξεις κλειδιά

Πρόβλημα του Πλανόδιου Πωλητή, TSP, πολυπλοκότητα, προσεγγιστικοί αλγόριθμοι, προσεγγιστικά σχήματα, PTAS, FPTAS, Metric TSP, Ευκλείδιο TSP, (1,2)-TSP.

Abstract

The aim of this thesis is the research on approximation algorithms and approximation schemes, which solve in polynomial time some special cases of the NP-complete Traveling Salesman Problem (TSP). The thesis begins with an introduction to complexity theory, to graph theory and to approximation algorithms. Subsequently, the general TSP is defined, followed by the definition of the Metric TSP (MTSP) with the presentation of some approximation algorithms solving it. Next comes the Euclidean TSP, for which a polynomial-time approximation scheme (PTAS) is given. The last case which is studied is the (1,2)-TSP, for which the most important algorithms are given; among them a very recent one. The thesis is completed with the presentation of some important applications of the TSP.

Key words

Traveling Salesman Problem, TSP, complexity, approximation algorithms, approximation schemes, PTAS, FPTAS, Metric TSP, Euclidea TSP, (1,2)-TSP.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION TO THE THEORY OF COMPLEXITY

From the beginning of their existence the human beings were trying to solve various problems in order to make their life easier. The development of the science of mathematics enabled people to proceed to the representation of the problems in mathematical terms and the creation of suitable conditions for their efficient solution.

Various algorithms were invented for the solution of these problems. But what do we mean by the term *algorithm*? Although a strict definition for the algorithm does not exist, we usually call "algorithm" a finite set of rules which describe a method for the solution of a specific problem. The method described is consisting of a sequence of computational processes.

However, research is not limited to the mere representation of an algorithm solving a specific problem but proceeds to the study of the measurable properties which characterize the efficiency of a computational method. Such measurable quantities are, for example, the computation time, the amount of memory space of the computer and several others, which characterize the complexity of the algorithm. Furthermore, the term *complexity* is used to characterize, beyond algorithms, the problems themselves. Namely, the complexity of a problem is represented by the complexity of the optimal algorithm which solves the specific problem.

Two aspects of the behavior of an algorithm can be studied; the average efficiency (average-case analysis) and also the worst case (worst-case analysis), including all possible instances of the specific problem which the algorithm solves.

An algorithm can be deterministic or non-deterministic. For the deterministic algorithms the computation proposed is linear. For every computational iteration of

them there exists exactly one legal next iteration. Their computational procedure proceeds step by step and is capable to stop for any possible input. Concerning the non-deterministic algorithms, for every one of their computational iterations there can exist many, one or none legal next computational iteration. The deterministic algorithms constitute, therefore, a subcase of the corresponding non-deterministic. The non-deterministic algorithms are consisting of two phases. In the first phase they guess a sequence of computational iterations, whereas in the second phase, under a clearly deterministic procedure, they check whether the result given by the first phase constitutes a solution of the problem.

1.1 The cost of an algorithm

The cost of an algorithm depends on the input of the problem. It usually increases with the increase of the size n of the input.

In most cases it is more practical to compute the order of the cost of an algorithm than it's exact value. What is interesting is its asymptotic behavior. In other words, we are looking for the marginal trend of increase of the function which represents the complexity of the algorithm while the size of the input is increased.

In the following we are going to define some symbols, which are used extensively for the study and the analysis of algorithms which solve the Traveling Salesman Problem (TSP).

Let $f : N\{0\} \to N\{0\}$ and $g : N\{0\} \to N\{0\}$, where N are the natural numbers. Then, we have the following notations:

$$O(g) = \{f | \exists c > 0, \exists n_0 : \forall n > n_0 : f(n) \leq cg(n)\}$$

$$o(g) = \{f | \forall c > 0, \exists n_0 : \forall n > n_0 : f(n) \leq cg(n)\}$$

$$\Omega(g) = \{f | \exists c > 0, \exists n_0 : \forall n > n_0 : f(n) \geq cg(n)\}$$

$$\omega(g) = \{f | \forall c > 0, \exists n_0 : \forall n > n_0 : f(n) \geq cg(n)\}$$

$$\Theta(g) = \{f | \exists c_1 > 0, \exists c_2 > 0 \exists n_0 : \forall n > n_0 : c_1 \leq \frac{f(n)}{g(n)} \leq c_2\}$$

If $g \in O(f)$, we usually write $g(n) = O(f(n))$ to denote that function g is of order f.

A function g is of polynomial degree referred to n, when $g \in O(poly) = \bigcup O(n^k)$.

The most basic criterion of whether an algorithm is efficient or not, is the time cost it presents or, better, its time complexity. This is represented by a function $T(n)$. $T(n)$ represents the number of steps that an algorithm makes, as a function of the size of its input n.

1.2 Optimization problems and decision problems

The problems are divided in various categories depending on their characteristics. We will refer here to two of the most basic categories, the optimization problems and the decision problems.

In the optimization problems, we are searching for the solution which maximizes or minimizes an objective function. For each instance of such a problem, there is a set of feasible solutions $F(x)$. For each solution $s \in F(x)$ we correspond, by means of an objective function c, a positive integer $c(s)$ and we are looking for the specific solution s which optimizes[1] $c(s)$. The Traveling Salesman Problem (TSP) is an optimization problem and, more specifically, a minimization problem.

Another category of problems are the decision problems. In this category the possible answers are two: *yes* and *no*. If we assume that all answers giving a "yes" belong to a set A, then these problems, for every input x, are taking the form : "$x \in A$?"

Suppose that we have an optimization problem with input x and we are looking for a feasible solution s which optimizes the objective function $c(s)$. This problem, like any optimization problem, can be transformed to the corresponding decision problem: For input x, is there a feasible solution s for x such that $c(s)$ is greater to n? For minimization problems we examine whether $c(s) < n$, whereas for maximization problems whether $c(s) > n$.

In general, it is particularly practical to transform problems of any category to decision problems, as these can be studied very effectively by means of the formal languages. If we take into account the fact that to introduce an instance of a problem to a computational model, this must be coded into a finite set of symbols, then we can understand why we want our problem to have a direct relation to the formal languages. Let's refer to some basic elements of them.

Suppose we have a finite set Σ of symbols. We define Σ^* as the set of the finite length strings consisting of symbols in Σ. If, for example, $\Sigma = \{0, 1\}$, then

[1] It makes it maximum or minimum, depending on whether we have maximization or minimization problem

$\Sigma^* = \{\varepsilon, 1, 0, 10, 1001, 0111, ...\}$. Apparently Σ^* has an infinite number of elements, as there exist infinite combinations of symbols in Σ. We say that L is a language of the alphabet[2] Σ if $L \subseteq \Sigma^*$.

For every decision problem, there exist instances for which the answer is "yes" and others for which the answer is "no". Suppose that for a problem Π the "yes" instances belong to the set N_Π whereas the "no" belong to O_Π. If a language L consists of the strings, say x, which constitute codings of the problem for which the answer is "yes", then this is defined as following:

$L(\Pi, k) = \{x \in \Sigma^* |\ x \text{ represents an instance } I \in N_\Pi\}$

By k we denote the way of coding the problem. Depending on the coding selected, we have a differentiation between N_Π and O_Π. A string x, therefore, belongs to the language L if the instance of the decision problem Π, which is represented by x by means of coding k, gives the answer "yes".

1.3 The classes P and NP

It has been stated above that a problem belongs to a complexity class depending on the complexity of the optimal algorithm which solves it. We will refer here to two very important classes of problems, P and NP. The problems belonging to class P can be solved by means of efficient algorithms. On the other hand, the corresponding class NP problems are considered as "hard" problems and no efficient[3] algorithms have been determined yet to solve them. We will refer in more detail to these two classes.

We call P (*polynomial*) the class of the problems which can be solved in polynomial time by some deterministic algorithm.

It has been said that every algorithm is realized in some computation model. The model to which we will be referring from now on is the *Deterministic Turing Machine* (DTM) in which we assume that the deterministic algorithm of the above definition is realized. A DTM program with an alphabet Σ accepts a string x, if it stops in state of acceptance q_N for x. An equivalent definition, therefore, for class P is:

$P = \{L | \exists \text{ polynomial time DTM which accepts language } L\}$

[2]We call alphabet a finite set of symbols like Σ.

[3]When we say efficient algorithms we mean polynomial time algorithms.

Let's clarify here that the time of a DTM for some n^4 is defined as the maximum time required for DTM to respond for any input string, let it be x, with length equal to n. A DTM is called of polynomial time, if there exists a polynomial p such that $\forall n \in Z_+$: time (DTM) $\leq p(n)$.

We call NP (Non-deterministic Polynomial Time) the class of problems which are solved in polynomial time by some non-deterministic algorithm.

If we recall the definition of non-deterministic algorithms, we can say that a problem belongs to the NP class when, after "guessing" a possible solution s for the specific problem, a verification of this solution can be made in polynomial time.

An equivalent definition for NP class is:

$NP = \{L | \exists$ polynomial time NDTM which accepts the language $L\}$

NDTM is a *Non-deterministic Turing Machine*. A NDTM selects each time the shortest path leading to a state of acceptance for a specific input string, let it be x.

Which is however the relation between the classes P and NP? Apparently P $\subseteq$ NP.We have said that the deterministic algorithm is a special case of the non-deterministic. Thus, a problem which is solved in polynomial time by a deterministic algorithm is also solved in polynomial time by a non-deterministic algorithm. What is searched until now is whether the problems which are solved in polynomial time by a NDTM can also be solved in polynomial time by a DTM. That is, if P $\supseteq$ NP and, subsequently, P = NP.

1.4 The concept of reduction

The problems belonging to classes P and NP are extremely numerous. Many times the question arises which one among several problems is more complex or whether one problem can be related to another. Answers to such questions are given by the concept of reduction. Let's see its exact definition:

We say that a problem B is polynomially reduced to a problem A and we denote this by $B \preceq A$, if there exists a transformation of polynomial time R, which, for every $x \in B$, produces an equivalent $R(x) \in A$.By the word "equivalent" we mean that if the input x of problem B[5] leads to the answer "yes", then an only then the input $R(x)$ of problem A leads also to an answer "yes". The same holds for a negative answer.

[4]where n is the length of the input string

[5]We consider that both A and B are decision problems

In a few words, instead of solving the problem B for an input x, we can compute $R(x)$ and solve the problem A for input $R(x)$. In both cases we find the same solution.

1.5 NP-Hard and NP-Complete Problems

When a problem B is reduced to a problem A then we can say that A is at least equally "hard" to B. This holds because if an efficient algorithm is found for A, then, automatically, B can also be solved efficiently. The more problems are reduced to a problem, say A, the more "hard" this problem is characterized.

NP-Hard are called the problems to which all problems belonging to class NP are reduced. Namely:

If $\forall \Pi' \in NP$ holds $\Pi' \preceq \Pi$ then Π is NP-Hard.

NP-Hard problems which belong to the class NP are called *NP-Complete* problems.

We consider from the above that NP-Hard and NP-Complete problems are characterized by their extreme difficulty. For example, if an algorithm of polynomial time is found which solves one of these problems, then all problems belonging to NP class can be solved polynomially. It will have thus been proved that P = NP. The Traveling Salesman Problem is one of the NP-Complete problems.

Chapter 2

INTRODUCTION TO GRAPH THEORY

In this chapter we will present some things about graph theory that are very useful to the description of the Traveling Salesman Problem (TSP) and the algorithms we will present about this problem and the special instances we will examine. We will give in this chapter some very useful definitions and some algorithms about graph theory.

2.1 Graphs

This section presents two kinds of graphs: directed and undirected. Certain definitions in the literature differ from those given here, but for the most part, the differences are slight. Section 2.3 shows how graphs can be represented in computer memory.

A **directed graph** (or **digraph**) G is a pair (V, E), where V is a finite set and E is a binary relation on V. The set V is called the vertex set of G, and its elements are called **vertices** (singular: **vertex**). The set E is called the edge set of G, and its elements are called **edges**. Figure 2.1(a) is a pictorial representation of a directed graph on the vertex set {1,2,3,4,5,6}. Vertices are represented by circles in the figure, and edges are represented by arrows. Note that **self-loops** (edges from a vertex to itself) are possible.

In an **undirected graph** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices, rather than ordered pairs. That is, an edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$. By convention, we use the notation (u, v) for an edge, rather than the set notation $\{u, v\}$, and (u, v) and (v, u) are considered to be the same edge. In an undirected graph, self-loops are forbidden, and so every edge consists of exactly two distinct vertices. Figure 2.1(b) is a pictorial representation of an undirected graph

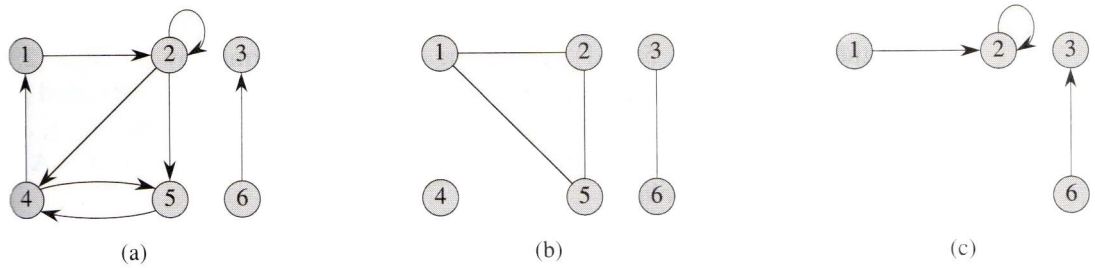


Figure 2.1: *Directed and undirected graphs.* ***(a)*** *A directed graph* $G = (V, E)$*, where* $V = \{1, 2, 3, 4, 5, 6\}$ *and* $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$*. The edge (2, 2) is a self loop.* ***(b)*** *An undirected graph* $G = (V, E)$*, where* $V = \{1, 2, 3, 4, 5, 6\}$ *and* $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$*. The vertex 4 is isolated.* ***(c)*** *The subgraph of the graph in part (a) induced by the vertex set* $\{1, 2, 3, 6\}$*.*

of the vertex set {1,2,3,4,5,6}.

Many definitions for directed and undirected graphs are the same, although certain terms have slightly different meanings in the two contexts. If (u, v) is an edge in a directed graph $G = (V, E)$, we say that (u, v) is **incident from** or **leaves** vertex u and is incident to or enters vertex v. For example, the edges leaving vertex 2 in Figure 2.1(a) are (2,2), (2,4) and (2,5). The edges entering vertex 2 are (1,2) and (2,2). If (u, v) is an edge in an undirected graph $G = (V, E)$, we say that (u, v) is **incident on** vertices u and v. In Figure 2.1(b), the edges incident on vertex 2 are (1,2) and (2,5).

If (u, v) is an edge in a graph $G = (V, E)$, we say that vertex v is **adjacent** to the vertex u. When the graph is undirected, the adjacency relation is symmetric. When the graph is directed, the adjacency relation is not necessarily symmetric. If v is adjacent to u in a directed graph, we sometimes write $u \rightarrow v$. In parts (a) and (b) of Figure 2.1, vertex 2 is adjacent to vertex 1, since the edge (1,2) belongs to both graphs. Vertex 1 is *not* adjacent to vertex 2 in Figure 2.1(a), since the edge (2,1) does not belong to the graph.

The **degree** of a vertex in an undirected graph is the number of edges incident on it. For example, vertex 2 in Figure 2.1(b) has degree 2. A vertex whose degree is 0, such as vertex 4 in Figure 2.1(b), is **isolated**. In a directed graph, the **out-degree** of a vertex is the number of edges leaving it, and the **in-degree** of a vertex is the number of edges entering it. The **degree** of a vertex in a directed graph is its in-degree plus its out-degree. Vertex 2 in figure 2.1(a) has in-degree 2, out-degree

3, and degree 5.

A **path** of **length** k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2, ..., v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, ..., k$. The length of the path is the number of edges in the path. The path **contains** the vertices $v_0, v_1, ..., v_k$ and the edges $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. (There is always a 0-length path from u to u.) If there is a path p from u to u', we say that u' is **reachable** from u via p, which we sometimes write as $u \rightsquigarrow u'$ if G is directed. A path is **simple** if all vertices in the path are distinct. In Figure 2.1(a), the path $\langle 2, 5, 4, 5 \rangle$ is not simple.

A **subpath** of path $p = \langle v_0, v_1, ..., v_k \rangle$ is a contiguous subsequence of its vertices. That is, for any $0 \leq i \leq j \leq k$, the subsequence of vertices $\langle v_i, v_{i+1}, ..., v_j \rangle$ is a subpath of p.

In a directed graph, a path $\langle v_0, v_1, ...v_k \rangle$ forms a **cycle** if $v_0 = v_k$ and the path contains at least one edge. The cycle is **simple** if, in addition, $v_1, v_2, ..., v_k$ are distinct. A self-loop is a cycle of length 1. Two paths $\langle v_0, v_1, v_2, ..., v_{k-1}, v_0 \rangle$ and $\langle v'_0, v'_1, v'_2, ..., v'_{k-1}, v'_0 \rangle$ form the same cycle if there exists an integer j such that $v'_i = v_{(i+j) mod k}$ for $i = 0, 1, ..., k-1$. In Figure 2.1(a), the path $\langle 1, 2, 4, 1 \rangle$ forms the same cycle as the paths $\langle 2, 4, 1, 2 \rangle$ and $\langle 4, 1, 2, 4 \rangle$. This cycle is simple, but the cycle $\langle 1, 2, 4, 5, 4, 1 \rangle$ is not. The cycle $\langle 2, 2 \rangle$ formed by the edge (2,2) is a self-loop. A directed graph with no self-loops is **simple**. In an undirected graph, a path $\langle v_0, v_1, ..., v_k \rangle$ forms a **(simple) cycle** if $k \geq 3$, $v_0 = v_k$, and $v_1, v_2, ..., v_k$ are distinct. For example, in Figure 2.1(b), the path $\langle 1, 2, 5, 1 \rangle$ is a cycle. A graph with no cycles is **acyclic**.

An undirected graph is connected if every pair of vertices is connected by a path. The connected components of a graph are the equivalence classes of vertices under the "is reachable from" relation. The graph in Figure 2.1(b) has three connected components: {1, 2, 5}, {3, 6} and {4}. Every vertex in {1, 2, 5} is reachable from every other vertex in {1, 2, 5}. An undirected graph is connected if it has exactly one connected component, that is, if every vertex is reachable from every other vertex.

A directed graph is **strongly connected** if every two vertices are reachable from each other. The **strongly connected components** of a directed graph are the equivalence classes of vertices under the "are mutually reachable" relation. A directed graph is strongly connected if it has only one strongly connected component. The graph in Figure 2.1(a) has three strongly connected components: {1, 2, 4, 5}, {3} and {6}. All pairs of vertices in {1, 2, 4, 5} are mutually reachable. The vertices {3, 6} do not form a strongly connected component, since vertex 6 cannot be reached from vertex 3.

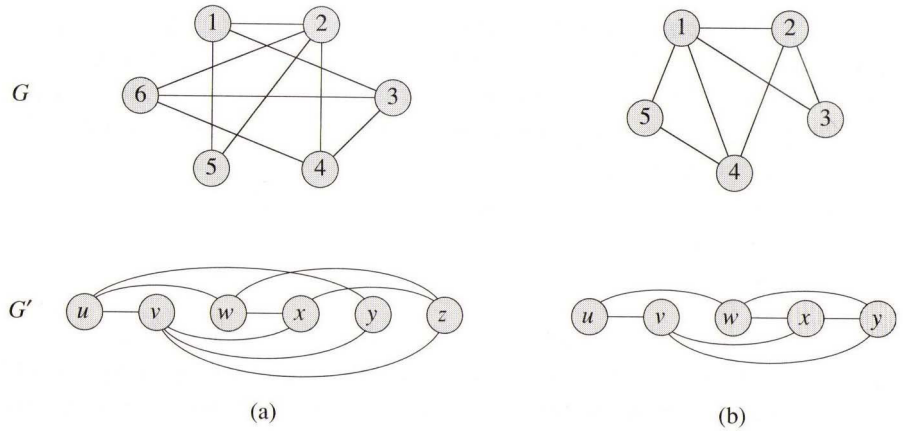


Figure 2.2: ***(a)*** *A pair of isomorphic graphs. The vertices of the top graph are mapped to the vertices of the bottom graph by* $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$. ***(b)*** *Two graphs that are not isomorphic, since the top graph has a vertex of degree 4 and the bottom graph does not.*

Two graphs $G = (V, E)$ and $G' = (V', E')$ are **isomorphic** if there exists a bijection $f : V \to V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. In other words, we can relabel the vertices of G to be vertices of G', maintaining the corresponding edges in G and G'. Figure 2.2(a) shows a pair of isomorphic graphs G and G' with respective vertex sets $V = \{1, 2, 3, 4, 5, 6\}$ and $V' = \{u, v, w, x, y, z\}$. The mapping from V to V' given by $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$ is the required bijective function. The graphs in Figure 2.2(b) are not isomorphic. Although both graphs have 5 vertices and 7 edges, the top graph has a vertex of degree 4 and the bottom graph has not.

We say that a graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Given a set $V' \subseteq V$, the subgraph of G **induced** by V' is the graph $G' = (V', E')$, where

$$E' = \{(u, v) \in E : u, v \in V'\}.$$

The subgraph induced by the vertex set {1, 2, 3, 6} in Figure 2.1(a) appears in Figure 2.1(c) and has the edge set {(1, 2), (2, 2), (6, 3)}.

Given an undirected graph $G = (V, E)$, the **directed version** of G is the directed graph $G' = (V, E')$, where $(u, v) \in E'$ if and only if $(u, v) \in E$. That is, each undirected edge (u, v) in G is replaced in the directed version by the two directed

edges (u, v) and (v, u). Given a directed graph $G = (V, E)$, the **undirected version** of G is the undirected graph $G' = (V, E')$, where $(u, v) \in E'$ if and only if $u \neq v$ and $(u, v) \in E$. That is, the undirected version contains the edges of G "with their directions removed" and with self-loops eliminated. (Since (u, v) and (v, u) are the same edges in an undirected graph, the undirected version of a directed graph contains it only once, even if the directed graph contains both edges (u, v) and (v, u).) In a directed graph $G = (V, E)$, a neighbor of a vertex u is any vertex that is adjacent to u in the undirected version of G. That is, v is a **neighbor** of u if $u \neq v$ and either $(u, v) \in E$ or $(v, u) \in E$. In an undirected graph, u and v are neighbors if they are adjacent.

Several kinds of graphs are given special names. A **complete graph** is an undirected graph in which every pair of vertices is adjacent. A **bipartite graph** is an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2. An acyclic, undirected graph is a forest, and a connected, acyclic, undirected graph is a **(free) tree** (see next section). We often take the first letters of "directed acyclic graph" and call such a graph a **dag**.

Weighted graphs are graphs for which each edge has an associated ***weight***, typically given by a ***weight function*** $w : E \to R$.

A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V. Figure 2.3 illustrates this notion. We say that an edge $(u, v) \in E$ ***crosses*** the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$. We say that a cut ***respects*** a set A of edges if no edge in A crosses the cut. An edge is a ***light edge*** crossing a cut if its weight is the minimum of any edge crossing the cut. There can be more than one ***light edge*** crossing a cut in the case of ties. More generally, we say that an edge is a light edge satisfying a given property if its weight is the minimum of any edge satisfying the property.

An ***Euler tour*** of a connected graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once. It has been proved that a graph has an Euler tour if and only if each vertex of the graph is of even degree. A ***hamiltonian cycle*** of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V. A graph that contains a hamiltonian cycle is said to be ***hamiltonian***; otherwise, it is ***nonhamiltonian***.

There are two variants of graphs that we may occasionally encounter. A **multigraph** is like an undirected graph, but it can have both multiple edges between vertices and self-loops. An **hypergraph** is like an undirected graph, but each **hy-**

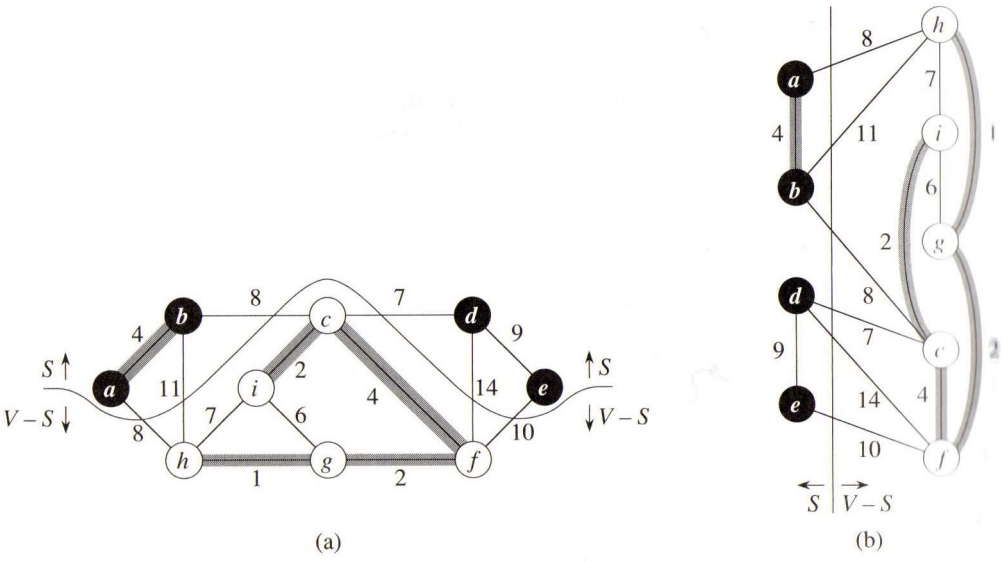


Figure 2.3: *Two ways of viewing a cut $(S, V - S)$ of a graph.* ***(a)*** *The vertices in the set S are shown in black, and those in $V - S$ are shown in white. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge crossing the cut. A subset A of the edges is shaded; note that the cut $(S, V - S)$ respects A, since no edge of A crosses the cut.* ***(b)*** *The same graph with the vertices in the set S on the left and the vertices in the set $V - S$ on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.*

peredge, rather than connecting two vertices, connects an arbitrary subset of vertices. Many algorithms written for ordinary directed and undirected graphs can be adapted to run on these graphlike structures.

The contraction of an undirected graph $G = (V, E)$ by an edge $e = (u, v)$ is a graph $G' = (V', E')$, where $V' = V - \{u, v\} \bigcup \{x\}$ and x is a new vertex. The set of edges E' is formed from E by deleting the edge (u, v) and, for each vertex w incident to u or v, deleting whichever of (u, w) and (v, w) is in E and adding the new edge (x, w).

2.2 Trees

As with graphs, there are many related, but slightly different, notions of trees. This section presents definitions and mathematical properties of several kinds of trees. Section 2.3 describes how trees can be represented in computer memory.

2.2.1 Free trees

As defined in the previous section (2.1), a **free tree** is a connected, acyclic, undirected graph. We often omit the adjective "free" when we say that a graph is a tree. If an undirected graph is acyclic but possibly disconnected, it is a **forest**. Many algorithms that work for trees also work for forests. Figure 2.4(a) shows a free tree, and Figure 2.4(b) shows a forest. The forest in Figure 2.4(b) is not a tree because it is not connected. The graph in Figure 2.4(c) is neither a tree nor a forest, because it contains a cycle.
The following theorem captures many important facts about free trees.

Theorem 2.1 (Properties of free trees) Let $G = (V, E)$ be an undirected graph. The following statements are equivalent.

1. G is a free tree.
2. Any two vertices on G are connected by a unique simple path.
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected.
4. G is connected, and $|E| = |V| - 1$
5. G is acyclic, and $|E| = |V| - 1$
6. G is acyclic, but if any edge is added to E, the resulting graph consists a cycle.

Proof (1) $\Rightarrow$ (2) : Since a tree is connected, any two vertices in G are connected by at least one simple path. Let u and v be vertices that are connected by two distinct simple paths p_1 and p_2, as shown in Figure 2.5. Let w be the vertex at which the paths first diverge; that is, w is the first vertex on both p_1 and p_2 whose successor on p_1 is x and whose successor on p_2 is y, where $x \neq y$. Let z be the first vertex at which the paths reconverge; that is, z is the first vertex following w on p_1 that is also on p_2. Let p' be the subpath of p_1 from w through x to z, and let p'' be the subpath of p_2 from w through y to z. Paths p' and p'' share no vertices except their endpoints. Thus, the path obtained by concatenating p' and the reverse of p'' is a cycle. This contradicts our assumption that G is a tree. Thus, if G is a tree, there can be at most one simple path between two vertices.

(2) $\Rightarrow$ (3) : If any two vertices in G are connected by a unique simple path, then G is connected. Let (u, v) be any edge in E. This edge is a path from u to v, and so it must be the unique path from u to v. If we remove (u, v) from G, there is no path from u to v, and hence its removal disconnects G.

(3) $\Rightarrow$ (4) : By assumption, the graph G is connected, and we know that $|E| \geq |V| - 1$. We shall prove $|E| \leq |V| - 1$ by induction. A connected graph with $n = 1$ or $n = 2$ vertices has $n - 1$ edges. Suppose that G has $n \geq 3$ vertices and that all graphs satisfying (3) with fewer than n vertices also satisfy $|E| \leq |V| - 1$. Removing an arbitrary edge from G separates the graph into $k \geq 2$ connected components (actually $k = 2$). Each component satisfies (3), or else G would not satisfy (3). Thus, by induction, the number of edges in all components combined is at most $|V| - k \leq |V| - 2$. Adding in the removed edge yields $|E| \leq |V| - 1$.

(4) $\Rightarrow$ (5) : Suppose that G is connected and that $|E| = |V| - 1$. We must show that G is acyclic. Suppose that G has a cycle containing K vertices $v_1, v_2, ..., v_k$, and without loss of generality assume that this cycle is simple. Let $G_k = (V_k, E_k)$ be the subgraph of G consisting of the cycle. Note that $|V_k| = |E_k| = k$. If $k < |V|$, there must be a vertex $v_{k+1} \in V - V_k$ that is adjacent to some vertex $v_i \in V_k$, since G is connected. Define $G_{k+1} = (V_{k+1}, E_{k+1})$ to be the subgraph of G with $V_{k+1} = V_k \bigcup \{v_{k+1}\}$ and $E_{k+1} = E_k \bigcup \{(v_i, v_{k+1})\}$. Note that $|V_{k+1}| = |E_{k+1}| = k+1$. If $k + 1 < |V|$, we can continue, defining G_{k+2} in the same manner, and so forth, until we obtain $G_n = (V_n, E_n)$, where $n = |V|$, $V_n = V$, and $|E_n| = |V_n| = |V|$. Since G_n is a subgraph of G, we have $E_n \subseteq E$, and hence $|E| \geq |V|$, which contradicts the assumption that $|E| = |V| - 1$. Thus, G is acyclic.

(5) $\Rightarrow$ (6) : Suppose that $G$is acyclic and that $|E| = |V| - 1$. Let k be the number of connected components of G. Each connected component is a free tree by definition, and since (1) implies (5), the sum of all edges in all connected components of G is $|V| - k$. Consequently, we must have $k = 1$, and G is in fact a tree. Since (1) implies (2), any two vertices in G are connected by a unique simple path. Thus, adding any edge to G creates a cycle.

(6) $\Rightarrow$ (1) : Suppose that G is acyclic but that if any edge is added to E, a cycle is created. We must show that G is connected. Let u and v be arbitrary vertices in G. If u and v are not already adjacent, adding the edge (u, v) creates a cycle in

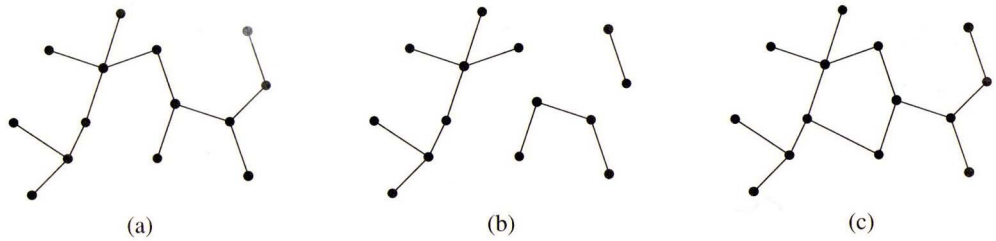


Figure 2.4: ***(a)*** *A free tree.* ***(b)*** *A forest.* ***(c)*** *A graph that contains a cycle and is therefore neither a tree nor a forest.*

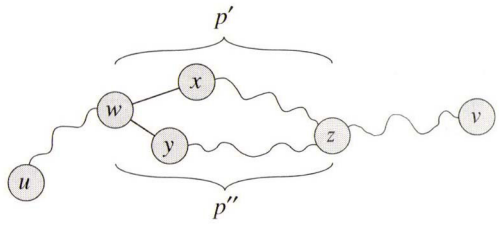


Figure 2.5: *A step in the proof of Theorem 2.1: if (1) G is a free tree, then (2) any two vertices in G are connected by a unique simple path. Assume for the sake of contradiction that vertices u and v are connected by two distinct simple paths p_1 and p_2. These paths first diverge at vertex w, and they first reconverge at vertex z. The path p' concatenated with the reverse of the path p'' forms a cycle, which yields a contradiction.*

which all edges but (u, v) belong to G. Thus, there is a path from u to v, and since u and v were chosen arbitrarily, G is connected.

2.2.2 Rooted and ordered trees

A **rooted tree** is a free tree in which one of the vertices is distinguished from the others. The distinguished vertex is called the **root** of the tree. We often refer to a vertex of a rooted tree as a **node**[1] of the tree. Figure 2.6(a) shows a rooted tree on a set of 12 nodes with root 7.

[1]The term "node" is often used in the graph theory literature as a synonym for "vertex". We shall reserve the term "node" to mean a vertex of a rooted tree.

Consider a node x in a rooted tree T with root r. Any node y on the unique path from r to x is called **ancestor** of x. If y is an ancestor of x, then x is a **descendant** of y. (every node is both an ancestor and a descendant of itself). If y is an ancestor of x and $x \neq y$, then y is a **proper ancestor** of x and x is a **proper descendant** of y. The **subtree rooted at** x is the tree induced by descendants of x, rooted at x. For example, the subtree rooted at node 8 in Figure 2.5(a) contains nodes 8, 6, 5, and 9.

If the last edge on the path from the root r of a tree t to a node x is (y, x), then y is the **parent** of x, and x is a **child** of y. The root is the only node n T with no parent. If two nodes have the same parent, they are **siblings**. A node with no children is an **external node** or **leaf**. A nonleaf node is an **internal node**.

The number of children of a node x in a rooted tree T is called the degree of x.[2] The length of the path from the root r to a node x is the **depth** of x in T. The **height** of a node in a tree is the number of edges on the longest simple downward path from the node to a leaf, and the height of a tree is the height of its root. The height of a tree is also equal to the largest depth of any node in the tree.

An ordered tree is a rooted tree in which the children of each node are ordered. That is, if a node has k children, then there is a first child, a second child, ..., and a kth child. The two trees in Figure 2.6 are different when considered to be ordered trees, but the same when considered to be just rooted trees.

2.2.3 Binary and positional trees

Binary trees are defined recursively. A **binary tree** T is a structure defined on a finite set of nodes that either

- contains no nodes, or
- is composed of three disjoint sets of nodes: a **root** node, a binary tree called its **left subtree**, and a binary tree called **right subtree**.

The binary tree that contains no nodes is called the **empty tree** or **null tree**, sometimes denoted NIL. If the left subtree is nonempty, its root is called the **left child** of the root of the entire tree. Likewise, the root of a nonnull right subtree is the **right child** of the root of the entire tree. If a subtree is the null tree NIL, we

[2]The degree of a node depends on whether T is considered to be a rooted tree or a free tree. The degree of a vertex in a free tree is, as in any undirected graph, the number of adjacent vertices. In a rooted tree, however, the degree is the number of children-the parent of a node does not count toward its degree.

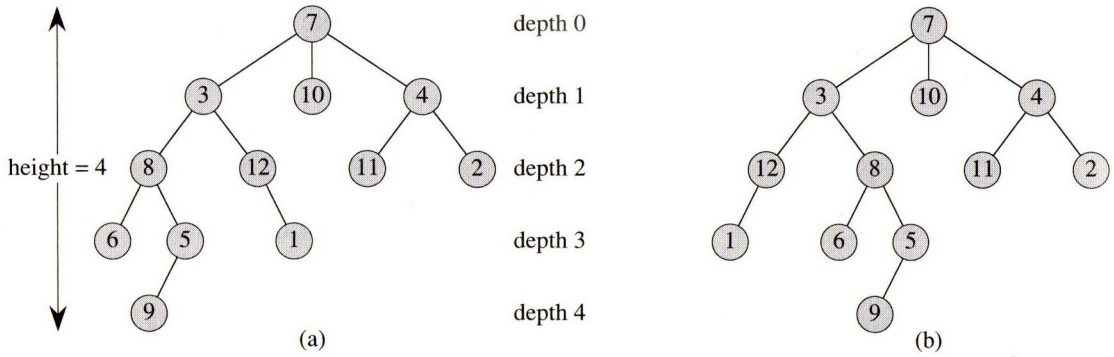


Figure 2.6: *Rooted and ordered trees.* ***(a)*** *A rooted tree with height 4. The tree is drawn in a standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left-to-right order of the children of a node matters; otherwise it doesn't.* ***(b)*** *Another rooted tree. As a rooted tree, it is identical to the tree in (a), but as an ordered tree it is different, since the children of node 3 appear in a different order.*

say that the child is **absent** or **missing**. Figure 2.7(a) shows a binary tree.

A binary tree is not simply an ordered tree in which each node has degree at most 2. For example, in a binary tree, if a node has just one child, the position of the child-whether it is the left child or the right child- matters. In an ordered tree, there is no distinguishing a sole child as being either left or right. Figure 2.7(b) shows a binary tree that differs from the tree in Figure 2.7(a) because of the position of one node. Considered as ordered trees, however, the two trees are identical.

The positioning information in a binary tree can be represented by the internal nodes of an ordered tree, as shown in Figure 2.7(c). The idea is to replace each missing child in the binary tree with a node having no children. These leaf nodes are drawn as squares in the figure. The tree that results is a **full binary tree**: each node is either a leaf or has degree exactly 2. There are no degree-1 nodes. Consequently, the order of the children of a node preserves the position information.

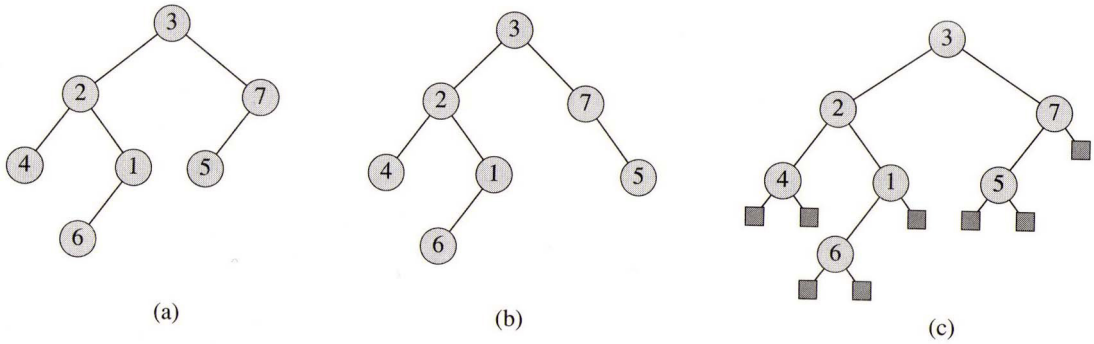


Figure 2.7: *Binary trees.* ***(a)*** *A binary tree drawn in a standard way. The left child of a node is drawn beneath the node and to the left. The right child is drawn beneath and to the right.* ***(b)*** *A binary tree different from the one in (a). In (a), the left child of node 7 is 5 and the right child is absent. In (b), the left child of node 7 is absent and the right child is 5. As ordered trees, these trees are the same, but as binary trees, they are distinct.* ***(c)*** *The binary tree in (a) represented by the internal nodes of a full binary tree: an ordered tree in which each internal node has degree 3. The leaves in the tree are shown as squares.*

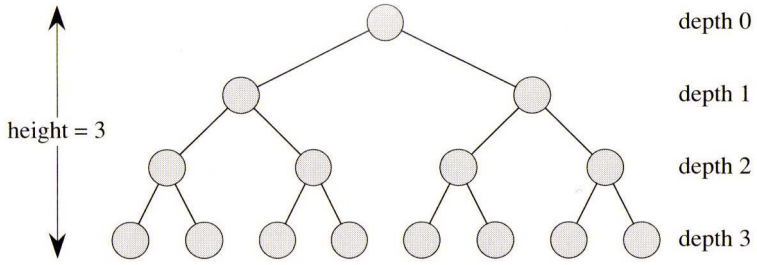


Figure 2.8: *A complete binary tree of height 3 with 8 leaves and 7 internal nodes.*

The posiioning information that distinguishes binary trees from ordered trees can be extended to trees with more than 2 children per node. In a **positional tree**, the children of a node are labeled with distinct positive integers. The ith child of a node is **absent** if no child is labeled with integer i. A k-ary tree is a positional tree in which for every node, all children with labels greater than k are missing. Thus, a binary tree is a k-ary tree with $k = 2$.

A **complete k-ary tree** is a k-ary tree in which all leaves have the same depth and all internal nodes have degree k. Figure 2.8 shows a complete binary tree of height 3. How many leaves does a complete k-ary tree of height h have? The root has k children at depth 1, each of which has k children at depth 2, etc. Thus, the number of leaves at depth h is k^h. Consequently, the height of a complete k-ary tree with n leaves is $\log_k n$. The number of internal nodes of a complete k-ary tree of height h is

$$1 + k + k^2 + ... + k^{h-1} = \sum_{i=0}^{h-1} k^i = \frac{k^h - 1}{k - 1}$$

Thus, a complete binary tree has 2^{h-1} internal nodes.

2.2.4 Traversing a binary tree

There are three ways to traverse a tree:

- inorder: The key of the root of a subtree is printed between the values in its left subtree and those in its right subtree.
- preorder: The root is printed before the values in either subtree.
- postorder: The root is printed after the values in its subtrees.

All these ways of traversing are working recursively.

For the tree in Figure 2.9, the sequences of the visiting nodes for each of the three ways are the following:

- inorder: 4 2 5 1 6 3
- preorder: 1 2 4 5 3 6
- postorder: 4 5 2 6 3 1

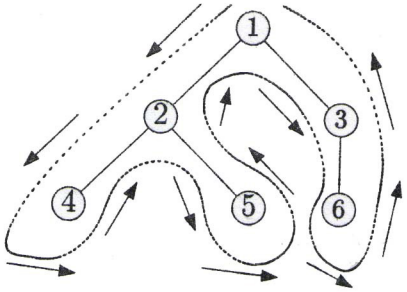


Figure 2.9: *Tree Traversal*

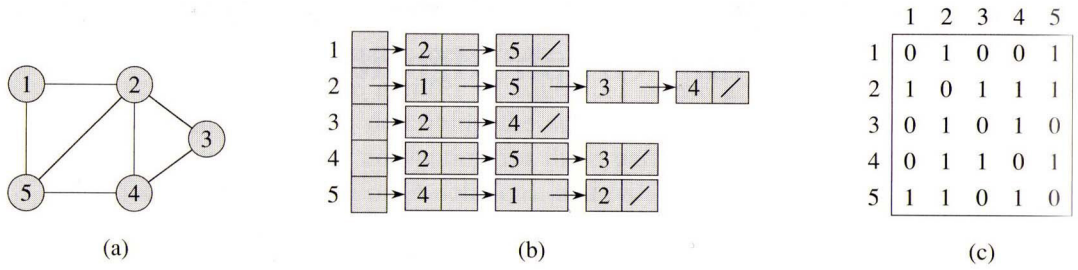


Figure 2.10: *Two representations of an undirected graph.* ***(a)*** *An undirected graph G having five vertices and seven edges.* ***(b)*** *An adjacency-list representation of G.* ***(c)*** *The adjacency-matrix representation of G.*

2.3 Representation of graphs

There are two standard ways to represent a graph $G = (V, E)$: as a collection of **adjacency lists** or as an **adjacency matrix**. Either way is applicable to both directed and undirected graphs. The adjacency-list representation is usually preferred, because it provides a compact way to represent ***sparse*** graphs-those for which $|E|$ is much less than $|V|^2$. Most of the graph algorithms which will be presented assume that an input graph is represented in adjacency-list form. An adjacency-matrix representation may be preferred, however, when the graph is ***dense***-$|E|$ is close to $|V|^2$-or when we need to be able to tell quickly if there is an edge connecting two given vertices.

The ***adjacency-list representation*** of a graph $G = (V, E)$ consists of an array

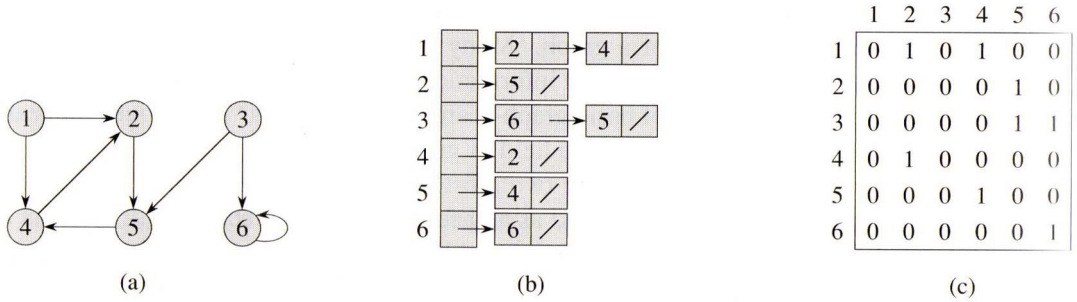


Figure 2.11: *Two representations of a directed graph.* ***(a)*** *A directed graph G having six vertices and eight edges.* ***(b)*** *An adjacency-list representation of G.* ***(c)*** *The adjacency-matrix representation of G.*

Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G. (Alternatively it may contain pointers to these vertices.) The vertices in each adjacency list are typically stored in an arbitrary order. Figure 2.10(b) is an adjacency-list representation of the undirected graph in Figure 2.10(a). Similarly, Figure 2.11(b) is an adjacency-list representation of the directed graph in Figure 2.11(a).

If G is a directed graph, the sum of the lengths of all the adjacency lists in $|E|$, since an edge of the form (u, v) is represented by having v appear in $Adj[u]$. If G is an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa. For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$.

Adjacency lists can readily be adapted to represent ***weighted graphs***. For example, let $G = (V, E)$ be a weighted graph with weight function w. The weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored with vertex v in u's adjacency list. The adjacency-list representation is quite robust in that it can be modified to support many other graph variants.

A potential disadvantage of the adjacency-list representation is that there is no quicker way to determine if a given edge (u, v) is present in the graph than to search for v in the adjacency list $Adj[u]$. This disadvantage can be remedied by an adjacency-matrix representation of the graph, at the cost of using asymptotically more memory.

For the ***adjacency-matrix representation*** of a graph $G = (V, E)$, we assume

that the vertices are numbered 1,2,.....,$|V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij}$ such that

$$a_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

Figures 2.8(c) and 2.10(c) are the adjacency matrices of the undirected and directed graphs in Figures 2.10(a) and 2.11(a), respectively. The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph.

Observe the symmetry along the main diagonal of the adjacency matrix in Figure 2.10(c). We define the ***transpose*** of a matrix $A = (a_{ij})$ to be the matrix $A^T = (a_{ij}^T)$ given by $a_{ij}^T = a_{ji}$. Since in an undirected graph, (u, v) and (v, u) represent the same edge, the adjacency matrix A of an undirected graph is its own transpose: $A = A^T$. In some applications, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half.

Like the adjacency-list representation of a graph, the adjacency-matrix representation can be used for weighted graphs. For example, if $G = (V, E)$ is a weighted graph with edge-weight function w, the weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored as the entry in row u and column v of the adjacency matrix. If an edge does not exist, a NIL value can be stored as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or ∞.

Although the adjacency-list representation is asymptotically at least as efficient as the adjacency-matrix representation, the simplicity of an adjacency matrix may make it preferable when graphs are reasonably small. Moreover, if the graph is unweighted, there is an additional advantage in storage for the adjacency-matrix representation. Rather than using one word of computer memory for each matrix entry, the adjacency matrix uses only one bit per entry.

2.4 Matching

Given an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is ***matched*** by matching M if some edge in M is incident on v; otherwise, v is ***unmatched***.

- A ***maximal matching*** is a matching that is not a proper subset of any other matching.
- A ***maximum matching*** is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M| \geq |M|$.

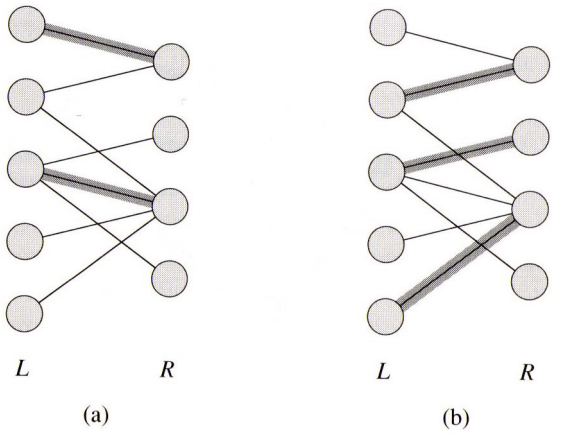


Figure 2.12: *A bipartite graph $G = (V, E)$ with vertex partition $V = L \cup R$.* ***(a)*** *A matching with cardinality 2.* ***(b)*** *A maximum matching with cardinality 3.*

- A ***perfect matching*** is a matching in which every vertex is matched.

Figure 2.12 illustrates the notion of a matching.

2.5 Minimum Spanning Tree

One of the most important problems in graph theory is the finding of the Minimum Spanning Tree (MST) of a graph. In this problem, we are given an undirected weighted graph and we try to find a tree whose edges belong to the edges of the graph, such that the total weight of the tree (the sum of the weights of all his edges) to be the minimum we can have.

We will now describe the problem in a formal language. We are given an undirected graph $G(V, E)$ and a matrix $c_{|V|\times|V|}$, which contains for each edge $(u, v) \in E$ the weight of the edge $c[u, v] > 0$. We have to find a tree $T(V', E')$ such that:

$$\begin{cases} V' = V \\ E' \subseteq E \\ \Sigma_{(u,v)\in E'} c[u, v] = MINIMUM \end{cases}$$

We will present briefly two algorithms that solve this problem (finding the Minimum Spanning Tree) in polynomial time.These are: ***Prim's algorithm*** and ***Kruskal's***

algorithm. Both algorithms are using the greedy strategy, that is, at each step, they make the choice that is best at the moment. Such strategy is not generally guaranteed to find globally optimal solutions to problems. For the Minimum Spanning Tree problem, however, it can be proved that the greedy strategies, which are used in the algorithms we will present, do yield a spanning tree with minimum weight.

2.5.1 Prim's Algorithm

Prim's algorithm begins with the empty solution. In each step it chooses the edge with the minimum weight from the remaining edges (the edges that have not been selected in previous steps of the algorithm) so that the resulting subgraph remains a tree (connected and acyclic graph). When there is a tie between the edges of the minimum cost, it doesn't matter which one of them we will select. The algorithm stops when there is no edge to choose, so that the resulting graph remains a tree. Then, we have as a result the Minimum Spanning Tree $T(V', E')$ of the given graph $G(V, E)$.
Figure 2.13 shows the execution of Prim's algorithm in a given graph step-by-step.

2.5.2 Kruskal's Algorithm

Like Prim's algorithm, Kruskal's algorithms begins also with the empty solution. In each step it selects the edge with the minimum weight from the remaining edges (the edges that have not been selected in previous steps of the algorithm) so there are not cycles in the resulting subgraph. It doesn't matter if there are two or more connected components in the graph which results after each step, and this is the difference with Prim's algorithm. When there is a tie between the edges of the minimum cost, it doesn't matter which one of them we will select. The algorithm stops when there is no edge to choose, so that the resulting graph has no cycles. Then, we have as a result the Minimum Spanning Tree $T(V', E')$ of the given graph $G(V, E)$.
Figure 2.14 shows the execution of Kruskal's algorithm in a given graph step-by-step.

Each of these algorithms (Prim's and Kruskal's) can easily be made to run in time $O(E \lg V)$ using ordinary binary heaps. By using Fibonacci heaps, Prim's algorithm can be sped up to run in time $O(E + V \lg V)$, which is an improvement if $|V|$ is much smaller than $|E|$.

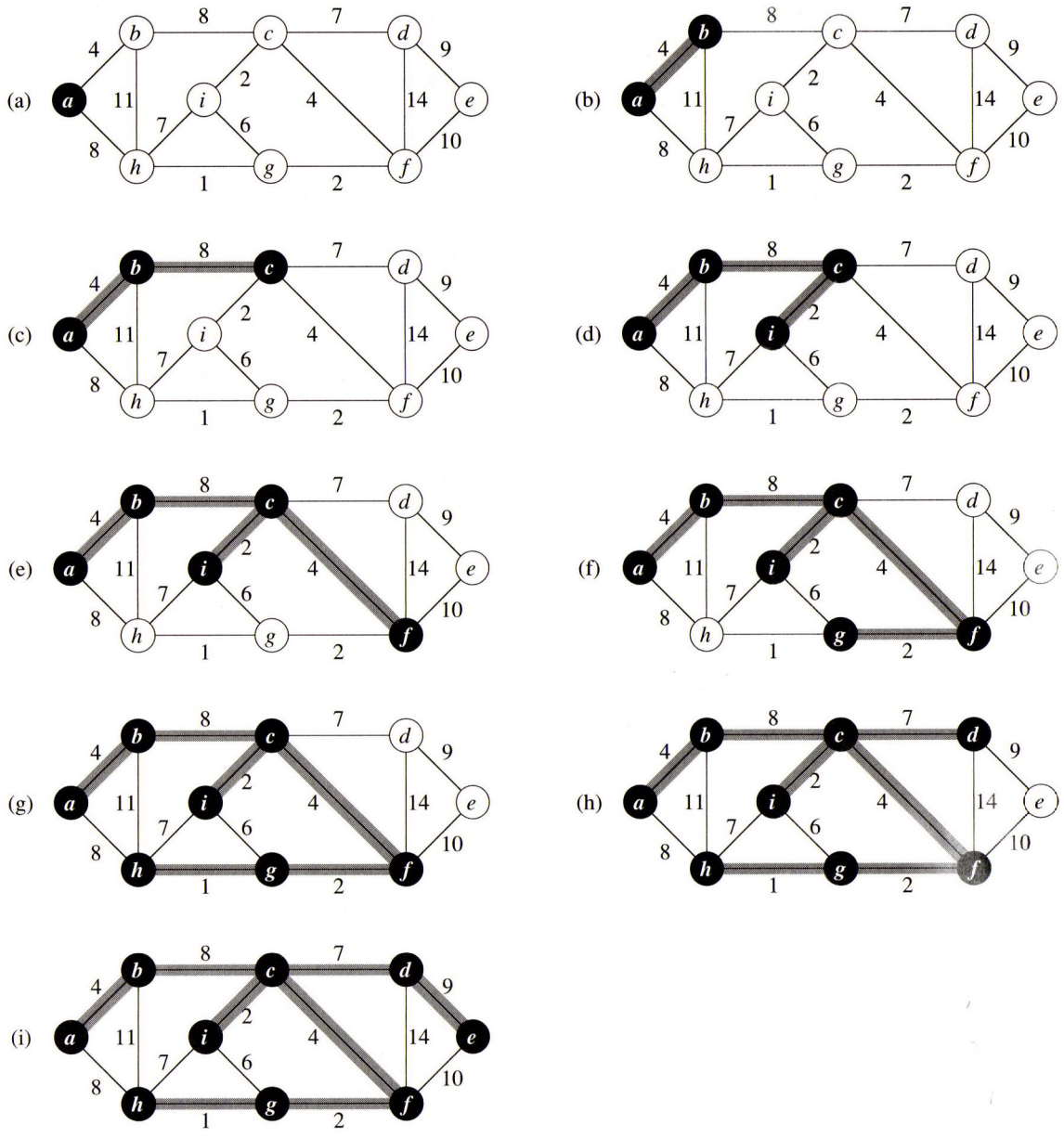
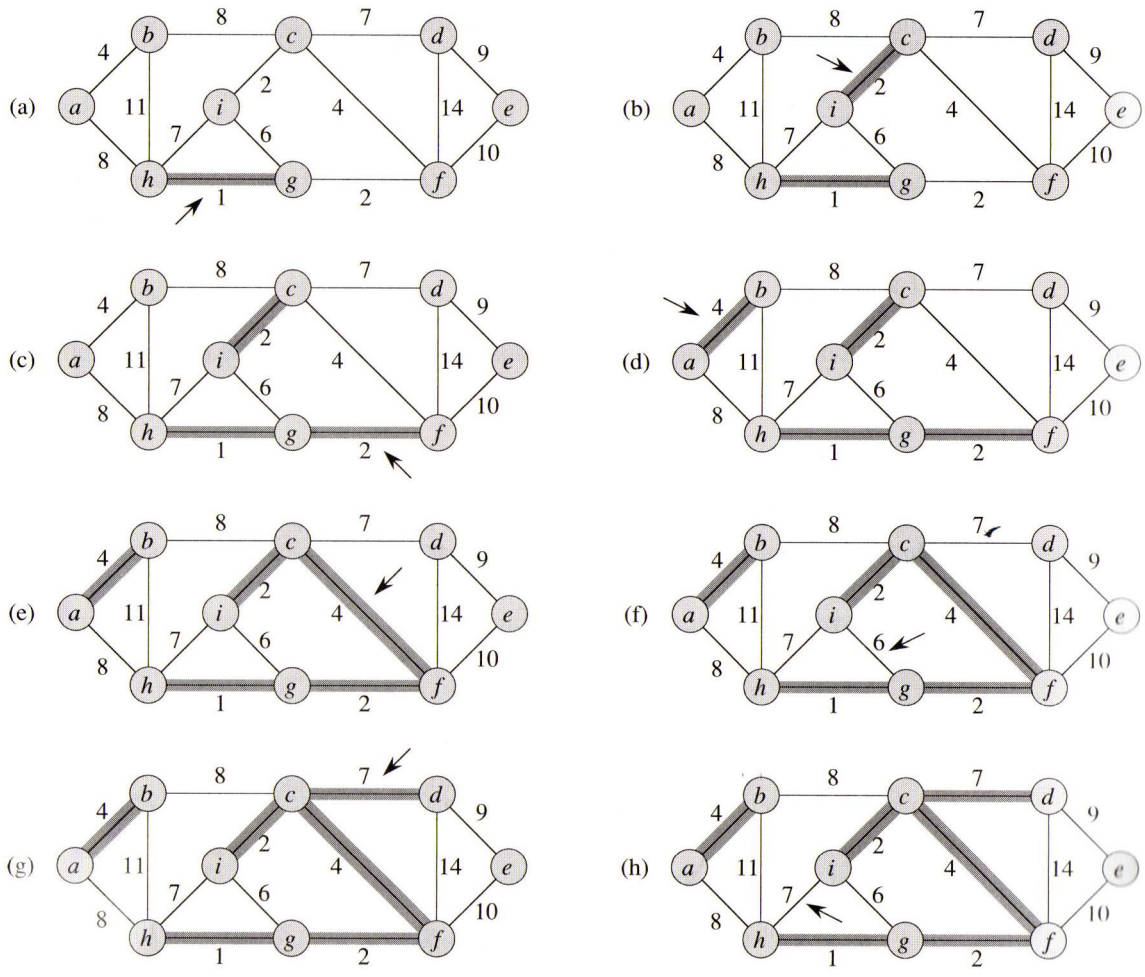


Figure 2.13: *The execution of Prim's algorithm on the graph from Figure 2.3. The root vertex is a. Shaded edges are in the tree being grown, and the vertices in the tree are shown in black. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice af adding either edge* (b, c) *or edge* (a, h) *to the tree since both are light edges crossing the cut.*


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
7
6
14
9
10
1
2

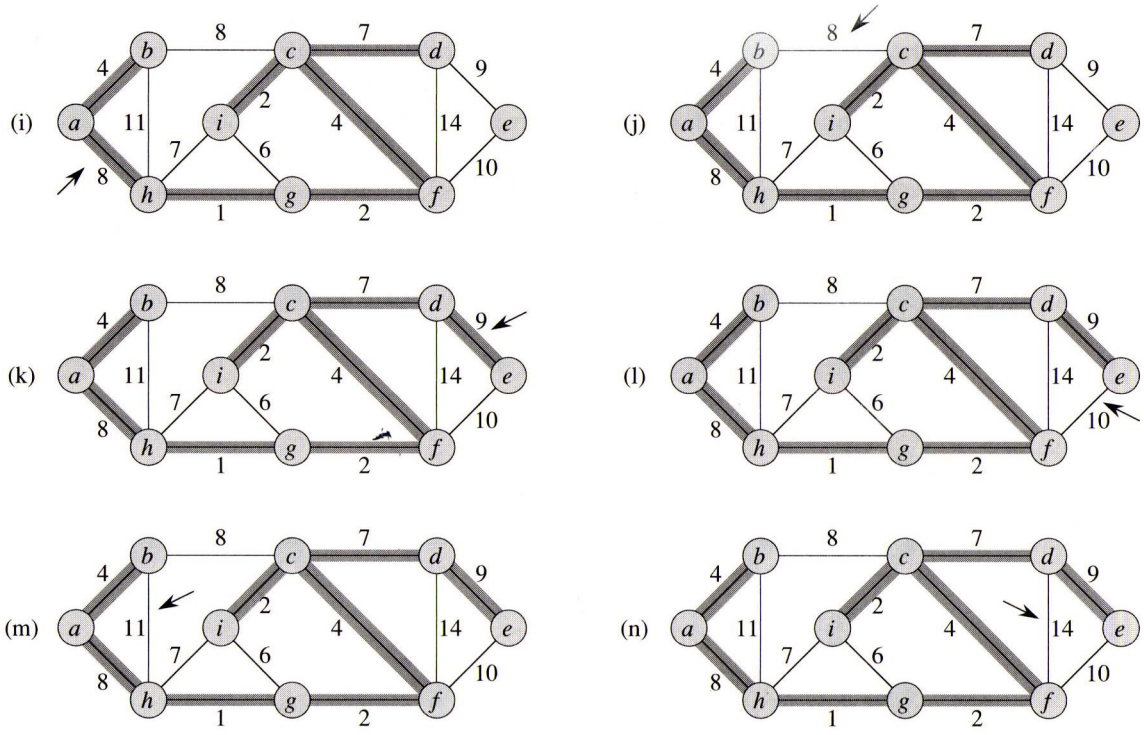


Figure 2.14: *The execution of Kruskal's algorithm on the graph from Figure 2.3. Shaded edges belong to the forest A being grown. The edges are considered by the algorithm in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.*

Chapter 3

INTRODUCTION TO APPROXIMATION ALGORITHMS

Many problems of practical significance are NP-complete but are too important to abandon merely because obtaining an optimal solution is intractable. If a problem is NP-complete, we are unlikely to find a polynomial-time algorithm for solving it exactly, but even so, there may be hope. There are at least three approaches to getting around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory. Second, we may be able to isolate important special cases that are solvable in polynomial time. Third, it may still be possible to find *near-optimal* solutions in polynomial time (either in the worst case or on average). In practice, near-optimality is often good enough. An algorithm that returns near-optimal solutions is called ***approximation algorithm***. This book presents polynomial-time approximation algorithms for special cases of the most difficult and most important and useful NP-complete problem, the ***Traveling Salesman Problem (TSP)***.

3.1 Performance ratios for approximation algoritms

Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution. Depending on the problem, an optimal solution may be defined as one with maximum possible cost or one with minimum possible cost; that is, the problem may be either a maximization or a minimization problem.

We say that an algorithm for a problem has an ***approximation ratio*** of $\rho(n)$

if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$max(\frac{C}{C^*}, \frac{C^*}{C}) \leq \rho(n).$$

We also call an algorithm that achieves an approximation ratio of $\rho(n)$ a $\rho(n)$-***approximation algorithm***. The definitions of approximation ratio and of $\rho(n)$-approximation algorithm apply for both minimization and maximization problems. For a maximization problem, $0 < C \leq C^*$, and the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution. Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution. Since all solutions are assumed to have positive cost, these ratios are always well defined. The approximation ratio of an approximation algorithm is never less than 1, since $C/C^* < 1$ implies $C^*/C > 1$. Therefore, a 1-approximation algorithm[1] produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

For many problems, polynomial-time approximation algorithms with small constant approximation ratios have been developed, while for other problems, the best known polynomial-time approximation algorithms have approximation ratios that grow as functions of the input size n. An example of such a problem is the set-cover problem.

3.2 Approximation Schemes (PTAS-FPTAS)

Some NP-complete problems allow polynomial-time approximation algorithms that can achieve increasingly smaller approximation ratios by using more and more computation time. That is, there is a trade-off between computation time and the quality of the approximation. An example is the subset-sum problem. This situation is important enough to deserve a name of its own.

An ***approximation scheme*** for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1+\epsilon)$-approximation algorithm. We say that an approximation scheme is a ***polynomial-time approximation scheme (PTAS)*** if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

[1]When the approximation ratio is independent of n, we will use the terms approximation ratio of ρ and ρ-approximation algorithm, indicating no dependance on n.

The running time of a polynomial-time approximation scheme can increase very rapidly as ϵ decreases. For example, the running time of a polynomial-time approximation scheme might be $O(n^{2/\epsilon})$. Ideally, if ϵ decreases by a constant factor, the running time to achieve the desired approximation should not increase by more than a constant factor. In other words, we would like the running time to be polynomial in $1/\epsilon$ as well as in n.

We say that an approximation scheme is ***fully polynomial-time approximation scheme (FPTAS)*** if it is an approximation scheme and its running time is polynomial both in $1/\epsilon$ and in the size n of the input instance. For example, the scheme might have a running time of $O((1/\epsilon)^2 n^3)$. With such a scheme, any constant-factor decrease in ϵ can be achieved with a corresponding constant-factor increase in the running time.

There are two ways of defining polynomial time approximation schemes, depending on whether the performance guarantee is absolute or holds only asymptotically as the optimum value becomes large (tends to infinity). The first (which is the usual one), called an *absolute polynomial time approximation scheme*, requires that for any ϵ there be an algorithm producing solutions with cost $\mathrm{SOL} \leq (1+\epsilon)\mathrm{OPT}$, where OPT is the optimum value (for minimization problems; for maximization problems the requirement is that $\mathrm{SOL} \geq (1-\epsilon)\mathrm{OPT}$). The other, called an *asymptotic polynomial time approximation scheme*, requires just that $\mathrm{SOL} \leq (1+\epsilon)\mathrm{OPT} + C$, where C is a constant (for minimization problems; for maximization problems the analogous requirement is that $\mathrm{SOL} \geq (1-\epsilon)\mathrm{OPT} - C$).

Chapter 4

THE TRAVELING SALESMAN PROBLEM (TSP)

In this chapter, we will give a presentation of the the NP-complete problem Traveling Salesman Problem (TSP), we will give an algorithm who solves the TSP using dynamic programming (these algorithms will be exponential, since the problem is NP-complete), and we will show that the TSP cannot be approximated, so we cannot have a polynomial-time algorithm to solve it is $P = NP$.

4.1 Description of the Traveling Salesman Problem (TSP)

In the ***Traveling Salesman Problem (TSP)***, which is closely related to the hamiltonian-cycle problem[1], a salesman must visit n cities. Modeling the problem as a complete graph with n vertices, we can say that the salesman wishes to make a tour, or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is an integer cost $c(i, j)$ to travel from city i to city j, and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour. For example, in Figure 4.1, a minimum-cost tour is $\langle u, w, v, x, u\rangle$, with cost 7. The formal language for the corresponding decision problem is

TSP $= \{\langle G, c, k\rangle : G = (V, E)$ is a complete graph, c is a function from $V \times V \to Z$, $k \in Z$, and G has a traveling-salesman tour with cost at most k $\}$.

As we said before, it has been proved that the Traveling Salesman Problem (TSP) is NP-Complete.

[1]The hamiltonian-cycle problem says:"Does a graph G have a hamiltonian cycle?". It has been proved that the hamiltonian-cycle problem is NP-complete.

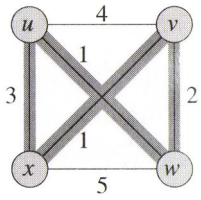


Figure 4.1: *An instance of the Traveling Salesman Problem. Shaded edges represent a minimum-cost tour, with cost 7.*

4.2 Using Dynamic Programming to solve the Traveling Salesman Problem (TSP)

In this section we will show how we can solve the Traveling Salesman Problem (TSP) using dynamic Programming.Before doing that, we will describe briefly the basic ideas of dynamic programming.

Dynamic programming solves problems by combining the solutions to subproblems. ("programming" in this context refers to a tabular method, not to writing computer code). Dynamic Programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered.

Dynamic programming is typically applied to ***optimization problems***. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.[2] We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.

[2]As we have said, the TSP is an optimization problem

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution in a bottom-up fashion.

4. Construct an optimal solution from computed information.

Steps 1-3 form the basis of a dynamic-programming solution to the problem. Step 4 can be omitted if only the value of an optimal solution i required. When we do perform step 4, we sometimes maintain additional information during the computation in step 3 to ease the construction of an optimal solution.

We will now present the algorithm for the TSP. We define $g(i, S)$ to be the cost of the shortest path from the node i to the node 1, which crosses all the nodes of S. Then, we have:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{cost[1, k] + g(k, V - \{1, k\})\}$$

, and generally:

$$g(i, S) = \min_{j \in S} \{cost[i, j] + g(j, S - \{j\})\}$$

Also: $g(i, \emptyset) = cost[i, 1]$.

Using this recursive function, we can step-by-step compute the shortest tour which starts from the node 1 and finishes in the same node.

Example 4.1. Suppose we have a directed graph, when the array of costs between his edges is the following:

$$C = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

We can see that if we compute $g(1, \{2, 3, 4\})$, the problem will have been solved. We start to compute all the g for $|V| = 0$, $|V| = 1$, $|V| = 2$ and $|V| = 3$.

When $|V| = 0$ we have:
$g(2, \emptyset) = Cost[2, 1] = 5$
$g(3, \emptyset) = Cost[3, 1] = 6$
$g(4, \emptyset) = Cost[4, 1] = 8$

When $|V| = 1$ we have:
$g(2, \{3\}) = Cost[2, 3] + g(3, \emptyset) = 9 + 6 = 15$
$g(2, \{4\}) = Cost[2, 4] + g(4, \emptyset) = 10 + 8 = 18$
$g(3, \{2\}) = Cost[3, 2] + g(2, \emptyset) = 13 + 5 = 18$
$g(3, \{4\}) = Cost[3, 4] + g(4, \emptyset) = 12 + 8 = 20$

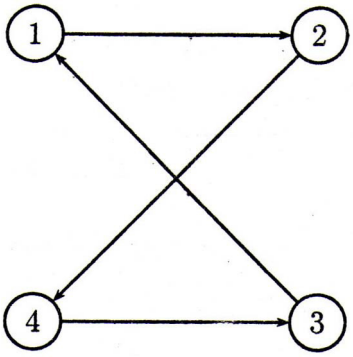


Figure 4.2: *Minimum cost tour.*

$g(4, \{2\}) = Cost[4, 2] + g(2, \emptyset) = 8 + 5 = 13$
$g(4, \{3\}) = Cost[4, 3] + g(3, \emptyset) = 9 + 6 = 15$

When $|V| = 2$ we have:
$g(2, \{3, 4 \swarrow\}) = \min(Cost[2, 3] + g(3, \{4\}), Cost[2, 4] + g(4, \{3\})) = \min(9 + 20, 10 + 15) = 25$
$g(3, \{2, 4 \swarrow\}) = \min(Cost[3, 2] + g(2, \{4\}), Cost[3, 4] + g(4, \{2\})) = \min(13 + 18, 12 + 13) = 25$
$g(4, \{2 \swarrow, 3\}) = \min(Cost[4, 2] + g(2, \{3\}), Cost[4, 3] + g(3, \{2\})) = \min(8 + 15, 9 + 18) = 23$

Finally, when $|V| = 3$ we have:
$g(1, \{2 \swarrow, 3, 4\}) = \min(Cost[1, 2]+g(2, \{3, 4\}), Cost[1, 3]+g(3, \{2, 4\}), Cost[1, 4]+ g(4, \{2, 3\})) = \min(10 + 25, 15 + 25, 20 + 23) = 35$

The arrows ($\swarrow$) show the node which is chosen in each case. So, in this graph, beginning from the node 1 the pointer leads us to 2. From 2 we go to 4 and from there it's obligatory to go to 3, since it's the only node we haven't visited, and the cycle closes in 1. Finally, the tour we ask for is: $1 \to 2 \to 4 \to 3$ and, of course, it doesn't matter which node we start from.

In each step k we compute the $g(i, S)$ for $n - 1$ different i. All $n - 2$ remaining elements (apart from $(1, j)$) are combined over $|S| = k$, that is, there are $\binom{n-2}{k}$ possible combinations (different values than g). So, the space required is:

$$\text{Space} = (n-1)\sum_{k=0}^{n-2}\binom{n-2}{k} = (n-1)(1+1)^{n-2} = (n-1)2^{n-2}$$

So, the space complexity is $O(n2^n)$. The time complexity is also exponential, since the problem is NP-complete and the algorithm is not effective.

4.3 Inapproximability of the Traveling Salesman Problem (TSP)

As we referred many times in the previous pages of this book, the Traveling Salesman Problem (TSP) is NP-complete, so there is no algorithm which solves this problem in polynomial time. It would be good if we could find an approximation algorithm for the TSP, this is, an algorithm that would return near-optimal solution of the problem. The bad news is that not only the TSP is NP-Complete, but it also cannot be approximated by any algorithm unless $P = NP$.

Theorem 4.1 (Inapproximability of the TSP) If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general Traveling Salesman Problem.

Proof The proof is by contradiction. Suppose to the contrary that for some number $\rho \geq 1$, there is a polynomial-time approximation algorithm A with approximation ratio ρ. Without loss of generality, we assume that ρ is an integer, by rounding it up if necessary. We shall then show how to use A to solve instances of the hamiltonian-cycle problem in polynomial time. Since the hamiltonian-cycle problem is NP-complete, solving it in polynomial time implies that $P = NP$.[3]

Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem. We wish to determine efficiently whether G contains a hamiltonian cycle by making use of the hypothesized approximation algorithm A. We turn G into an instance of the Traveling Salesman Problem as follows. Let $G' = (V, E')$ be the complete graph on V; that is,

$$E' = \{(u, v) : u, v \in V \text{ and } u \neq v\}.$$

Assign an integer cost to each edge in E' as follows:

$$c(u, v) = \begin{cases} 1, & \text{if } (u, v) \in E, \\ \rho|V| + 1, & \text{otherwise.} \end{cases}$$

Representations of G' and c can be created from a representation of G in time polynomial in $|V|$ and $|E|$.

[3]Due to a theorem, if any NP-complete problem is polynomial-time solvable, then $P = NP$. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

Now, consider the Traveling Salesman Problem in the graph G' with cost function c. If the original graph G has a hamiltonian cycle H, then the cost function c assigns to each edge of H a cost of 1, and so the graph G' contains a tour of cost $|V|$. On the other hand, if G does not contain a hamiltonian cycle, then any tour of G' must use some edge not in E. But any tour that uses an edge not in E has a cost of at least

$$(\rho|V| + 1) + (|V| - 1) = \rho|V| + |V| > \rho|V|.$$

Because edges not in G are so costly, there is a gap of at least $\rho|V|$ between the cost of a tour that is a hamiltonian cycle in G (cost $|V|$) and the cost of any other tour (cost at least $\rho|V| + |V|$).

What happens if we apply the approximation algorithm A to the Traveling Salesman Problem in the graph G' with cost function c? Because A is guaranteed to return a tour of cost no more than ρ times the cost of an optimal tour, if G contains a hamiltonian cycle, then A must return it. If G has no hamiltonian cycle, then A returns a tour of cost more than $\rho|V|$. Therefore, we can use A to solve the hamiltonian-cycle problem in polynomial time.[4]

[4]The proof of Theorem 4.1 is an example of a general technique for proving that a problem cannot be approximated well. Suppose that given a NP-hard problem X, we can produce in polynomial time a minimization problem Y such that "yes" instances of X correspond to instances of Y with value at most k (for some k), but that "no" instances of X correspond to instances of Y with value greater than ρk. Then we have shown that, unless $P = NP$, there is no polynomial-time ρ-approximation algorithm for problem Y.

Chapter 5

THE METRIC TRAVELING SALESMAN PROBLEM (MTSP)

In the previous chapter we presented the (general) Traveling Salesman Problem (TSP) and we proved that this problem, which is NP-complete, cannot be approximated. The good news is that, although the general TSP cannot be approximated, there are some special cases of the TSP, for which we can find approximation algorithms which solve them. In this chapter we will present one of these special cases, the Metric Traveling Salesman Problem (MTSP), giving some approximation algorithms for this problem.

5.1 Presentation of the Metric Traveling Salesman Problem (MTSP)

In the Traveling Salesman Problem, we are given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$, and we must find a hamiltonian cycle (a tour) of G with minimum cost. As an extension of our notation, let $c(A)$ denote the total cost of the edges in the subset $A \subseteq E$:

$c(A) = \sum_{(u,v)\in A} c(u, v)$.

In many practical situation, it is always cheapest to go directly from a place u to a place w; going by way of any intermediate stop v can't be less expensive. Putting it another way, cutting out an intermediate stop never increases the cost. We formalize this notion by saying that the cost function c satisfies the ***triangle inequality*** if for all vertices $u, v, w \in V$,

$c(u, w) \leq c(u, v) + c(v, w)$.

The triangle inequality is a natural one, and in many applications it is automatically satisfied. For example, if the vertices of the graph are points in the plane and the cost of the traveling between two vertices is the ordinary euclidean distance between them, then the triangle inequality is satisfied. (There are many cost functions other than the euclidean distance that satisfy the triangle inequality.)

The Traveling Salesman Problem (TSP) in a graph $G = (V, E)$ where the cost function c of his edges satisfies the triangle inequality, is called Metric Traveling Salesman Problem (MTSP). The MTSP cannot be solved by an exact algorithm in polynomial time, but can be approximated. In the next sections of this chapter we will give various approximation algorithms for the MTSP, with different approximation ratios.

5.2 A 2-approximation algorithm for the Metric Traveling Salesman Problem (MTSP)

We are given a full undirected graph $G = (V, E)$ and his cost function c which satisfies the triangle inequality. We need to find the optimal tour in this graph. We will first compute a structure- a minimum spanning tree (MST)- whose weight is a lower bound on the length of an optimal traveling salesman tour. We will then use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree's weight, as long as the cost function satisfies the triangle inequality. The following algorithm implements this approach:

2-Approx-MTSP-Tour

1. select a vertex $r \in V[G]$ to be a "root" vertex
2. compute a minimum spanning tree T for G from root r using Prim's algorithm
3. let L be the list of vertices visited in a preorder tree walk of T
4. **return** the hamiltonian cycle H that visits the vertices in the order L

Recall from section 2.2.4 that a preorder tree walk recursively visits every vertex in the tree, listing a vertex when it is first encountered, before any of its children are visited.

Figure 5.1 illustrates the operation of 2-Approx-MTSP-Tour. Part (a) of the figure shows the given set of vertices, and part (b) shows the minimum spanning

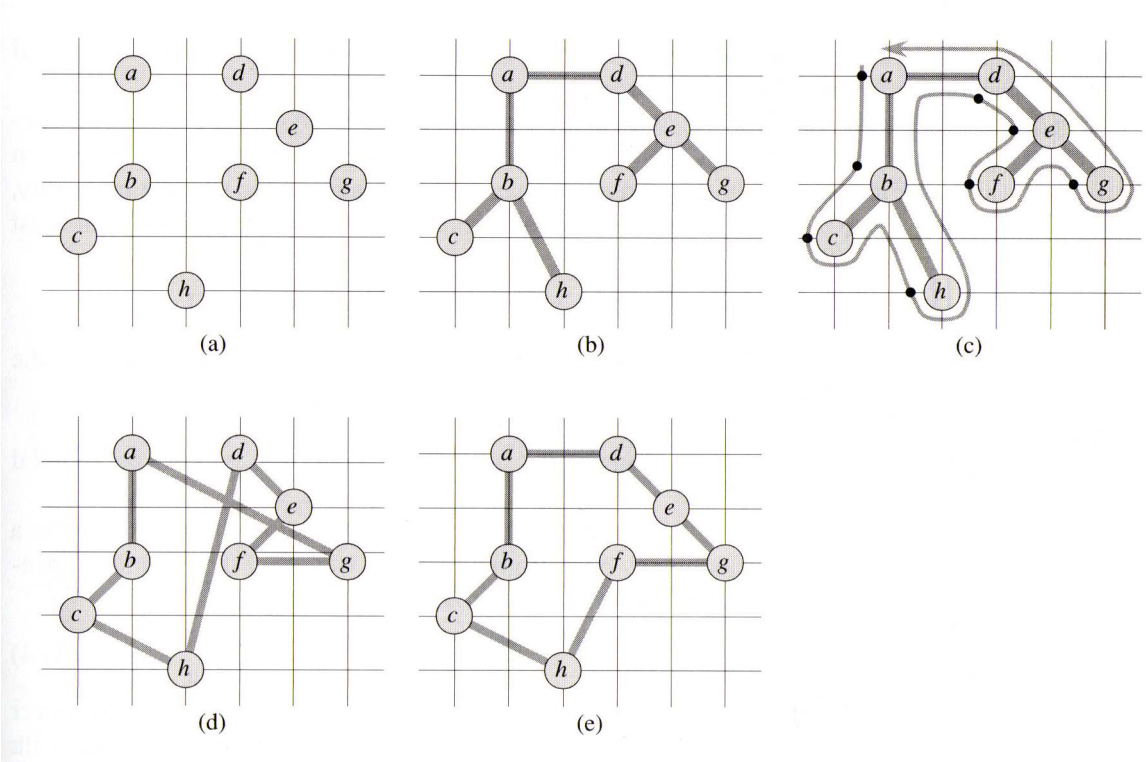


Figure 5.1: *The operation of 2-Approx-MTSP-Tour.* ***(a)*** *The given set of points, which lie on vertices of an integer grid. For example, f is one unit to the right and two units up from h. The ordinary euclidean distance is used as the cost function between two points.* ***(b)*** *A minimum spanning tree (MST) T of these points, as computed by Prim's Algorithm. Vertex a is the root vertex. The vertices happen to be labeled in such a way that they are added to the main tree by Prim's Algorithm in alphabetical order.* ***(c)*** *A walk of T, starting at a. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g.* ***(d)*** *A tour of the vertices obtained by visiting the vertices in the order given by the preorder walk. This is the tour H returned by 2-Approx-MTSP-Tour. Its total cost is approximately 19.074.* ***(e)*** *An optimal tour H^* for the given set of vertices. Its total cost is approximately 14.715.*

tree T grown from root vertex a by Prim's algorithm. Part (c) shows how the vertices are visited by a preorder walk of T, and part (d) displays the corresponding tour, which is the tour returned by 2-Approx-MTSP-Tour. Part (e) displays an optimal tour, which is about 23 percent shorter.

Even with a simple implementation of Prim's algorithm, the running time of 2-Approx-MTSP-Tour is $\Theta(V^2)$. We now show that if the cost function for an instance of the Traveling Salesman Problem satisfies the triangle equality, that is, if we have an instance of the Metric Traveling Salesman Problem, then 2-Approx-MTSP-Tour returns a tour whose cost is not more than twice the cost of an optimal tour. In other words, we show that the algorithm we presented above is 2-approximation.

Theorem 5.1 2-Approx-MTSP-Tour is a polynomial-time 2-approximation algorithm for the Traveling Salesman Problem with the triangle inequality (Metric TSP).

Proof We have already shown that 2-Approx-MTSP-Tour runs in polynomial time.

Let H^* denote an optimal tour for the given set of vertices. Since we obtain a spanning tree by deleting any edge from a tour, the weight of the minimum spanning tree T is a lower bound on the cost of an optimal tour, that is,

$$c(T) \leq c(H^*). \tag{5.1}$$

A ***full walk*** of T lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. Let us call this walk W. The full walk of our example gives the order

$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a.$

Since the full walk traverses every edge of T exactly twice, we have (extending our definition of the cost c in the natural manner to handle multisets of edges)

$$c(W) = 2c(T). \tag{5.2}$$

Equations (5.1) and (5.2) imply that

$$c(W) \leq 2c(H^*), \tag{5.3}$$

and so the cost of W is within a factor of 2 of the cost of an optimal tour.

Unfortunately, W is generally not a tour, since it visits some vertices more than once. By the triangle inequality, however, we can delete a visit to any vertex from W and the cost does not increase. (If a vertex v is deleted from W between visits to

u to w, the resulting ordering specifies going directly from u to w.) By repeatedly applying this operation, we can remove from W all but the first visit to each vertex. In our example, this leaves the ordering

a, b, c, h, d, e, f, g.

This ordering is the same as that obtained by a preorder walk of the tree T. Let H be the cycle corresponding to this preorder walk. It is a hamiltonian cycle, since every vertex is visited exactly once, and in fact it is the cycle computed by 2-Approx-MTSP-Tour. Since H is obtained by deleting vertices from the full walk W, we have

$$c(H) \leq c(W). \tag{5.4}$$

Combining inequalities (5.3) and (5.4) shows that $c(H) \leq 2c(H^*)$, which completes the proof.

The algorithm presented in this section is not the only 2-approximation algorithm for the MTSP. In the next section, we will present another approximation algorithm with approximation ratio 2 for the MTSP.

5.3 2-Approximation algorithm for the Metric Traveling Salesman Problem (MTSP) using closest-point heuristic

In this section we will present another approximation algorithm for the MTSP, and we will prove that this algorithm returns a tour whose total cost is not more than twice the cost of an optimal tour, that is, the approximation ratio of this algorithm equals 2. This algorithm is based on a ***closest-point heuristic***, and for this reason we will give it the name: ***closest-point algorithm***.

Closest-point algorithm

1. select a trivial cycle consisting of a single arbitrarily chosen vertex
2. **repeat**
3. identify the vertex u that is not on the cycle but whose distance to any vertex on the cycle is minimum. Suppose that the vertex on the cycle that is nearest u is vertex v
4. Extend the cycle to include u by inserting u just after v
5. **until** all vertices are on the cycle

6. **return** the cycle

We will now prove that this algorithm is 2-approximation algorithm for the MTSP.

Theorem 5.2 Closest-point algorithm is a polynomial-time 2-approximation algorithm for the Traveling Salesman Problem with the triangle inequality (Metric TSP).

Proof Let T be the tour produced by the closest-point algorithm, node u_i be the point added at the step i by the algorithm and let v_i be the point on cycle closest to u_i when u_i added. We can view the procedure as:

(i) Original cycle: ....., v', v_i, v'', ...
(ii) Adding u: ....., v', v_i, u, v_i, v'', ...
(iii) Removing the duplicate node: ...., v', v_i, u, v'', ...

The cycle resulting from (iii) is what is constructed by the algorithm. From the triangle inequality, the cost of (iii) is no larger than the cost of (ii). Thus, the cost incurred by adding u_i to the tour is at most $2c(v, u_i)$, so the total cost of the tour created by the algorithm satisfies $c(T) \leq 2\sum_i c(u_i, v_i)$.

In Prim's algorithm, nodes and edges are added in the exact same sequence as in the closest-point algorithm. Thus, the cost of the Minimum Spanning Tree (MST) produced by Prim's algorithm is $\sum_i c(u_i, v_i)$ and $c(T) \leq 2\sum_i c(u_i, v_i) \leq 2c(MST)$.[1] As we said in the previous section, the weight of the MST is a lower bound of the cost of an optimal tour, that is $c(MST) \leq c(OPT)$.[2]

Thus, we have shown that:

$c(T) \leq 2c(MST) \leq 2c(OPT)$

and we are done.

In spite of the nice approximation ratio provided by Theorems 5.1 and 5.2, ***2-Approx-MTSP-Tour*** and ***Closest-point algorithm*** are not the best practical choices for this problem. There are other approximation algorithms that typically perform much better in practice.

[1] $c(MST)$ is the total cost of the Minimum Spanning Tree (MST)
[2] $c(OPT)$ is the total cost of the optimal tour

5.4 Improving the factor to 3/2

We have seen till now two algorithms (2-Approx-MTSP-Tour and Closest-point algorithm), which both have an approximation ratio 2. As we said in the final words of the previous section, there are also approximation ratio with much better approximation ratios for this problem. From the known algorithms for the MTSP, this with the best ratio is Christofides' algorithm. In this section we will present this algorithm, and we will prove that he is 3/2-approximation.

Christofides' algorithm

1. Find an MST of G, say T
2. Compute a minimum cost perfect matching, M, on the set of odd-degree vertices of T. Add M to T and obtain an Eulerian graph.
3. Find an Euler tour, $\mathcal{T}$, of this graph.
4. Output the tour that visits vertices of G in order of their first appearance in $\mathcal{T}$. Let $\mathcal{C}$ be this tour.

Interestingly, the proof of this algorithm is based on a second lower bound on the optimal solution.[3].

Lemma 5.1 Let $V' \subseteq V$, such that $|V'|$ is even, and let M be a minimum cost perfect matching on V'. Then, $c(M) \leq c(OPT)/2$.

Proof Consider an optimal TSP tour of G, say τ. Let τ' be the tour on V' obtained by visiting the vertices of G in order of their first appearance in τ. By the triangle inequality, $c(\tau') \leq c(\tau)$. Now, τ' is the union of two perfect matchings on V', each consisting of alternate edges of τ'. Thus, the cheaper of these matchings has cost $\leq c(\tau')/2 \leq c(OPT)/2$. Hence the optimal matching also has cost at most $c(OPT)/2$.

Theorem 5.3 Christofides' algorithm achieves an approximation guarantee of 3/2 for Metric Traveling Salesman Problem (MTSP).

Proof The cost of the Euler tour is

$$c(\mathcal{T}) \leq c(T) + c(M) \leq c(OPT) + \frac{1}{2}c(OPT) = \frac{3}{2}c(OPT)$$

[3]The first lower bound of the cost of the optimal tour we have given is the weight of the minimum spanning tree

where the second inequality follows by using the two lower bounds on the cost of the optimal solution. Using the triangle inequality, $c(\mathcal{C}) \leq c(\mathcal{T})$, and the theorem follows.

Christofides' algorithm is the one with the best approximation ratio for the MTSP between the algorithms we know today. Finding a better approximation algorithm for Metric TSP is currently one of the outstanding open problems in this area. Many researchers have conjectured that an approximation factor of 4/3 may be achievable.

A very important point about the Metric TSP is that Arora, Lund, Motwani, Sudan and Szegedy have proved [2] that Metric TSP is one of the problems that there is no PTAS to solve them. In the next chapter, however, we will give a special case of the metric TSP (euclidean TSP) for which there is a PTAS, and we will describe this PTAS.

Chapter 6

THE EUCLIDEAN TRAVELING SALESMAN PROBLEM

In the previous chapter we presented a special case of the Traveling Salesman Problem, the Metric Traveling Salesman Problem (MTSP). We saw that the MTSP can be approximated, and we described various approximation algorithms, of which the best has approximation ratio 3/2. In this chapter we will present another case of TSP, the Euclidean TSP, and we will give a PTAS for this special case. The central idea of the PTAS we will give is to define a "coarse solution", depending on the error parameter ε, and to find it using dynamic programming. A feature is that we do not know a deterministic way of specifying the coarse solution - it is specified probabilistically.

6.1 Description of the Euclidean Traveling Salesman Problem

Euclidean TSP is an instance of the Metric TSP where the cities which must be visited by the salesman are points in $\mathbb{R}^d$, and the distance between two cities is the Euclidean distance between these cities.The description of the problem in a formal language follows.

For fixed d, given n points in $\mathbb{R}^d$, the problem is to find the minimum length tour of the n points. The distance between any two points x and y is defined to be the Euclidean distance between them, i.e., $(\sum_{i=1}^{d}(x_i - y_i)^2)^{1/2}$.

6.2 Description of the algorithm

We will give the algorithm for points in the plane, i.e., $d = 2$. The extension to arbitrary d is straightforward.

Define the ***bounding box*** of the instance to be the smallest axis-parallel square that contains all n points. Via a simple perturbation of the instance, we may assume that the length of this square, L, is $4n^2$ and that there is a unit grid defined on the square such that each point lies on a gridpoint. Further, assume without loss of generality that n is a power of 2, and let $L = 2^k$, $k = 2 + 2\log_2 n$.

The basic *dissection* of the bounding box is a recursive partitioning into smaller squares. Thus, the $L \times L$ square is divided into four $L/2 \times L/2$ squares, and so on. It will be convenient to view this dissection as a 4-ary tree, T, whose root is the bounding box. The four children of the root are the four $L/2 \times L/2$ squares, and so on. The nodes of T are assigned *levels*. The root is at level 0, its children at level 1, and so on. The squares represented by nodes get levels accordingly. Thus, squares at level i have dimensions $L/2^i \times L/2^i$. The dissection is continued until we obtain unit squares. Clearly, T has depth $k = O(\log n)$. By a ***useful square*** we mean a square represented by a node in T.

Next, let us define ***levels for horizontal and vertical lines*** that accomplish the basic dissection (these are all the lines of the grid defined on the bounding box). The two lines that divide the bounding box into four squares have level 1. In general, the 2^i lines that divide the level $i-1$ squares into level i squares each have level i. Therefore a line of level i forms the edge of useful squares at levels i, $i+1$, ... , i.e., the largest useful square on it has dimensions $L/2^i \times L/2^i$.

Each line will have a special set of points called ***portals***. The coarse solution we will be seeking is allowed to cross a line only at a portal. The portals on each line are equidistant points. On a line of level i, these points are $L/(2^i m)$ apart, where the parameter m is fixed to be a power of 2 in the range $[k/\varepsilon, 2k/\varepsilon]$. Clearly, $m = O(\log n/\varepsilon)$. Since the largest useful square on a level i line has dimensions $L/2^i \times L/2^i$, each useful square has a total of at most $4m$ portals on its four sides and corners. We have chosen m to be a power of 2 so that a portal in a lower level square is a portal for all higher level squares it lies in.

We will say that a tour τ is *well behaved with respect to the basic dissection and has limited crossings* if it is well behaved with respect to the basic dissection, and furthermore, it visits each portal at most twice.

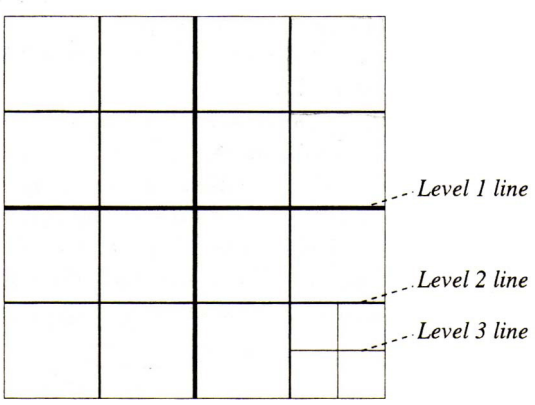


Figure 6.1: *Bounding box*

Lemma 6.1 Let tour τ be well behaved with respect to the basic dissection. Then, there must be a tour that is well behaved with limited crossings, whose length is at most that of τ.

Proof The basic reason is that removing self-intersections by "short-cutting" can only result in a shorter tour, since Euclidean distance satisfies the triangle inequality. If τ uses a portal on line l more than twice, we can keep "short-cutting" on the two sides of l until the portal is used at most twice. If this introduces additional self-intersections, they can also be removed.

Lemma 6.2 The optimal well behaved tour with respect to the basic dissection, having limited crossings, can be computed in time $2^{O(m)} = n^{O(1/\varepsilon)}$.

Proof We will build a table, using dynamic programming, that contains, for each useful square, the cost of each valid visit. We will sketch the main ideas of this.

Let τ be the optimal tour we wish to find. Clearly, the total number of times τ can enter and exit a useful square, S, is at most $8m$. The part of τ inside S is simply a set of at most $4m$ paths, each entering and exiting S at portals, and together covering all the points inside the square. Furthermore, the paths must be internally non-self-intersecting, i.e., two paths can intersect only at their entrance or exit points. This means that the pairing of entrance and exit points of the paths must form a balanced arrangement of parentheses.

Let us call such a listing of portals, together with their pairing as entrance and exit points, a ***valid visit***.

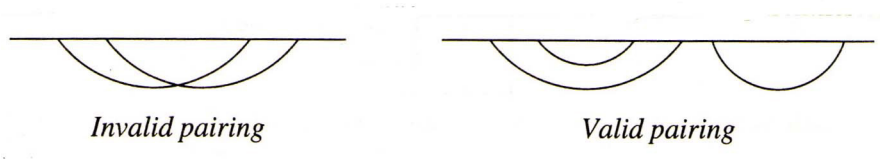


Figure 6.2: *Valid and invalid pairing*

The number of useful squares is clearly $poly(n)$. Let us first show that the number of valid visits in a useful square is at most $n^{O(1/\varepsilon)}$, thereby showing that the number of entries in the table is bounded by $n^{O(1/\varepsilon)}$.

Consider a useful square S. Each of its portals is used 0, 1, or 2 times, a total of $3^{4m} = n^{O(1/\varepsilon)}$ possibilities. Of these, retain only those possibilities that involve an even number of portal usages. Consider one such possibility, and suppose that it uses $2r$ portals. Next, we need to consider all possible pairings of these portals that form a balanced arrangement of parentheses. The number of such arrangements is the rth Catalan number, and is bounded by $2^{2r} = n^{O(1/\varepsilon)}$. Hence, the total number of valid visits in S is bounded by $n^{O(1/\varepsilon)}$

For each entry in the table, we need to compute the optimal length of this valid visit. The table is built up the decomposition tree, starting at its leaves. Consider a valid visit V in a square S. Let S be a level i square. We have already fixed the entrances and exits on the boundary of S. Square S has four children at level $i+1$, which have four sides internal to S, with a total of a most $4m$ more portals. Each of these portals is used 0, 1, or 2 times, giving rise again to $n^{O(1/\varepsilon)}$ possibilities. Consider one such possibility, and consider all its portal usages together with portal usages of a valid visit V. Obtain all possible valid pairings of these portals that are consistent with those of visit V. Again, using Catalan numbers, their number is bounded by $n^{O(1/\varepsilon)}$. Each such pairing will give rise to valid visits in the four squares. The cost of the optimal way of executing these valid visits in the four squares has already been computed. We compute their sum. The smallest of these sums is the optimal way of executing visit V in square S.

6.3 Proof of correctness

For the proof of correctness, it suffices to show that there is a well behaved tour with respect to the basic dissection whose length is bounded by $(1+\varepsilon)c(OPT)$. It turns out that this is not always the case. Instead, we will construct a larger family of dissections and will show that, for any placement of the n points, at least half these dissections have short well behaved tours with limited crossings. So, picking a random dissection from this set suffices.

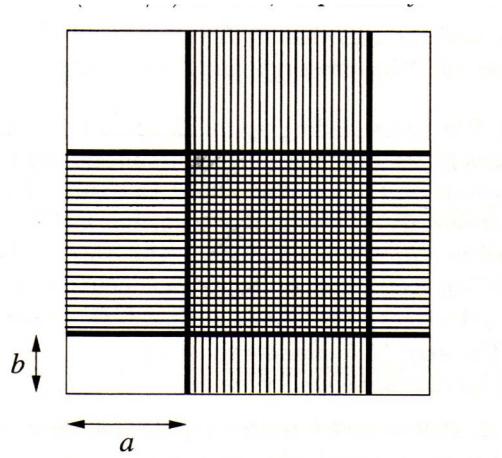


Figure 6.3: *Shifted bounding box*

Let us define L^2 different dissections of the bounding box, which are shifts of the basic dissection. Given integers a, b with $0 \leq a, b \leq L$, the (a, b)-shifted dissection is obtained by moving each vertical line from its original location x to $(a + x)$ mod L, and moving each horizontal line from its original location y to $(b + y)$ mod L. Thus, the middle lines of the shifted dissection are located at $(a + L/2)$ mod L and $(b + L/2)$ mod L, respectively.

The entire bounding box is thought of as being "wrapped around". Useful squares that extend beyond L in their x or y coordinates will thus be thought of as "wrapped around", and will still be thought of as a single square. Of course, the positions of the given n points remain unchanged; only the dissection is shifted.

Let π be the optimal tour, and $N(\pi)$ be the total number of times π crosses horizontal and vertical grid lines. If π uses a point at the intersection of two grid lines, then we will count it as two crossings. It can be proved the following:

Lemma 6.3 $N(\pi) \leq 2 \cdot c(OPT)$

Following is the central fact leading to the PTAS.

Theorem 6.1 Pick a and b uniformly at random from $[0, L)$. Then, the expected increase in cost in making π well behaved with respect to the (a, b)-shifted dissection is bounded by $2\varepsilon \cdot c(OPT)$.

Proof Given any dissection, consider the process of making π well behaved with respect to it. This involves replacing a segment of π that does not cross a line l at a portal by two segments so that the crossing is at the closest portal on l. The corresponding increase in the length of the tour is bounded by the interportal distance on line l.

Consider the expected increase in length due to one of the crossings of tour π with a line. Let l be this line. l will be a level i line in the randomly picked dissection with probability $2^i/L$. If l is a level i line, then the interportal distance on it is $L/(2^i m)$. Thus, the expected increase in the length of the tour due to this crossing is at most

$$\sum_i \frac{L}{2^i m}\frac{2^i}{L} = \frac{k}{m} \leq \varepsilon,$$

where we have used the fact that m lies in $[k/\varepsilon, 2k/\varepsilon]$. The theorem follows by summing over all $N(\pi)$ crossings and using Lemma 6.3.

Remark 6.1 The ideas leading up to Theorem 6.1 can be summarized as follows. Since lower level lines have bigger useful squares incident at them, we had to place portals on them further apart to ensure that any useful square had at most $4m$ portals on it (thereby ensuring that dynamic programming could be carried out in polynomial time). But this enabled us to construct instances for which there was no short well behaved tour with respect to the basic dissection. On the other hand, there are fewer lines having lower levels.

Now, using Markov's inequality we get:

Corollary 6.1 Pick a and b uniformly at random from $[0, L)$. Then, the probability that there is a well behaved tour of length at most $1+4\varepsilon \cdot c(OPT)$ with respect to the (a, b)-shifted dissection is greater or equal to $1/2$.

Notice that Lemma 6.1 holds in the setting of an (a, b)-shifted dissection as well. The PTAS is now straightforward. Simply pick a random dissection, and find an optimal well behaved tour with limited crossings with respect to this dissection using the dynamic programming procedure of Lemma 6.2. Notice that the same procedure holds even for a shifted dissection. The algorithm can be derandomized by trying all possible shifts and outputting the shortest tour obtained. Thus, we get:

Theorem 6.2 There is a PTAS for the Euclidean TSP problem in $\mathbb{R}^2$.

Chapter 7

THE TRAVELING SALESMAN PROBLEM WITH DISTANCES ONE AND TWO ((1,2)-TSP. A 7/6-APPROXIMATION ALGORITHM.

We have seen till now some very interesting special cases of the Traveling Salesman Problem. We saw firstly the Metric TSP, including some very important approximation algorithms for it. The best approximation ratio for this problem is 3/2. Then, we examined a special case of it, the Euclidean TSP, giving the most important algorithm (PTAS) we know about this problem. In this chapter we will present a special case of the Euclidean TSP, the (1,2)-TSP and will give a polynomial-time approximation algorithm with worst case ratio 7/6. In this special case of TSP, all distances of the graph are either one or two. We will also show that this special case of the Traveling Salesman Problem is MAX SNP-hard (we will determine what does it mean), and therefore it is unlikely that it has a polynomial-time approximation scheme.

7.1 Introduction. Description of the (1,2)-TSP.

As we have said, we will study an interesting further case of the Traveling Salesman Problem, namely the one in which the matrix is symmetric and the entries are either one or two. Since $a \leq b + c$ for all $a, b, c \in \{1, 2\}$, such instances of the Traveling Salesman Problem satisfy the triangle inequality. This special case of the

TSP can be considered as a generalization of the Hamiltonian Cycle problem, since we are asking for the tour of the nodes of the graph that contains the fewest possible nonedges (2 entries). It follows immediately that the problem is NP-complete. In fact, it is this version of the Traveling Salesman Problem that was shown NP-complete in the original reduction by Karp [25], and since then in every textbook on the subject.

In this chapter we will show that this special case of the Traveling Salesman Problem can be approximated in polynomial time by a ratio of 7/6. In the approximation algorithm we will present, we use the well-known technique of *subtour patching*: We start from the optimum 2-matching (subgraph with all degrees equal to 2, but not necessarily connected), and patch the cycles of the 2-matching together to form a tour. Two cycles can be "patched" together by picking one edge in each, deleting these two edges, and connecting the four endpoints in any one of the other two ways (Figure 7.1).

For example, the following simple algorithm achieves a ratio of $\frac{4}{3}$: We start with the optimum 2-matching, which contains at most $n/3$ cycles.[1] Consider any two cycles (Figure 7.1). We can always patch them together with an increase of 2 in the cost (see Figure 7.1). However, if there is any edge of weight 2 in one of the cycles, the cost is increased by at most 1, since we can choose to replace two edges, with total weight at least 3, with two more edges of total weight at most 4. So, we start with the cycles of the optimum 2-matching, and patch them together, one after the other. The first time, patching two cycles may cost 2 but, since an edge of weight 2 is now present, from now on it is going to cost at most 1 per patching (more precisely, at most 2, and each patching with cost 2 is either the first, or is preceded immediately by a patching of cost at most zero). Since at most $n/3 - 1$ patchings are needed, the cost of the tour finally created is at most the cost of the optimum 2-matching plus $n/3$, at most $\frac{4}{3}$ times the optimum tour.

In what follows, we use a more complex argument to develop a polynomial algorithm with worst case ratio of $\frac{11}{9}$. The basic idea is to patch the cycles in an order specified by the solution of yet another matching problem. We also show that, if we combine this idea with a polynomial-time algorithm for finding the optimal triangle-free 2-matching of a graph [22], the bound becomes $\frac{7}{6}$.

[1] because each cycle has at least three nodes.

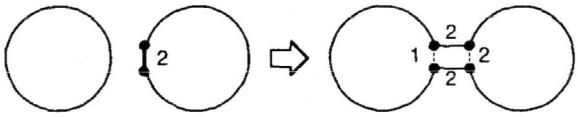


Figure 7.1: *Simple Patching Heuristic*

7.2 The approximation algorithm

We are given an $n \times n$ symmetric distance matrix d_{ij}, with entries 1 and 2. Matrix d_{ij} defines a graph $G_d = (V, E)$, where $V = \{1, 2, ..., n\}$, and $[i, j] \in E$ if and only if $d_{ij} = 1$. We are asked to find the cyclic permutation π that minimizes $\sum_{i=1}^{n} d_{i,\pi(i)}$.

We first describe the algorithm and argument for the special case in which G_d has a Hamiltonian cycle (i.e., the optimal cost is n). We first find an optimal 2-matching of d, that is to say, the shortest possible subgraph of degree 2. This can be done in time $O(n^{2.5})$ [35].Naturally, since G_d has a Hamiltonian cycle, the optimal 2-matching has weight n. It has a set C of connected components (cycles); in general, $|C| > 1$ (otherwise we are done).

We form a bipartite graph B with node sets C and V, and with an edge from component c to node v if v does not belong to c and there is an edge $[u, v]$ in E going from a node u of cycle c to v.

Lemma 7.1 There is a matching in B covering all cycles in C.

Proof Consider the Hamiltonian cycle, $(v_1, v_2, ..., v_n)$, and match cycle c with the first node v_i not in c, for which v_{i-1} is in c.

Consider now the directed graph $F = (C, A)$ with $(c, c') \in A$ whenever c is matched according to Lemma 7.1 with node v of c'. Obviously, F has out-degree 1, that is to say, it is a *function*. We need the following lemma about functions.

Lemma 7.2 Any function F has a spanning subgraph consisting of node-disjoint in-trees[2] of depth 1 and paths of length 2.

Proof A weak connected component of F is a cycle, with certain in-trees converging into it. We shall consider each weak component separately. We start with a leaf

[2]An *in-tree* is an oriented tree in which a single vertex is reachable from every other one. Likewise, an arborescence, or *out-tree* or branching, is an oriented tree in which all vertices are reachable from a single vertex

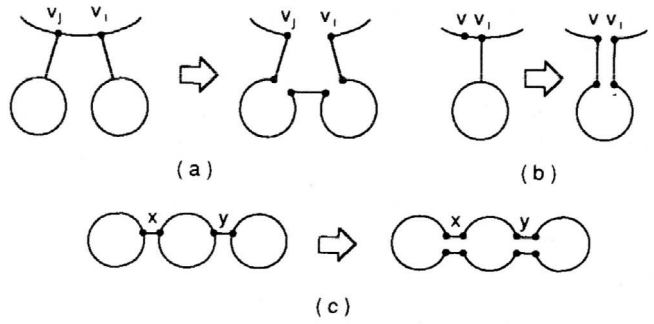


Figure 7.2: *The Three Cases*

l that has a maximum distance from a node in the cycle. We consider l's successor s. If s is not on the cycle, we define an in-tree by s and all of its immediate predecessors (also leaves). We remove this tree, and repeat. We end up with the cycle, plus certain immediate predecessors of nodes in the cycle. Notice that, for the nodes of the cycle that have predecessors outside the cycle, we have a choice whether or not to include in its in-tree the predecessor on the cycle. Well, we do whatever leaves an even number, possibly zero, of nodes between this node and the previous such node in the cycle. These even-length paths are trivially decomposable into paths of length 1 (which are indeed in-trees of height 1). This leaves us with the case in which there is no node outside the cycle. This cycle can be easily decomposed into paths of length 1 and a single path of length 2 (notice that F has no self-loops).

With every arc (c, c') of the directed graph F we can associate an undirected edge (v, v') of the graph G_d where v is a node of the cycle c and v' is the node of c' to which c is matched. Note that any two arcs of F with the same head c' are associated with edges of G_d that are incident with distinct nodes of the cycle c'. However, if c' is the head of one arc and the tail of another arc of F, then the edges of G_d associated with these two arcs may be incident to the same node of c'.

The approximation algorithm we present proceeds as follows: We decompose the directed graph F as in Lemma 7.2, and then we look at the resulting components. We first consider the in-trees of height 1. They consist of a cycle c (the root) together with other cycles $c_1, ..., c_m$, and, for $i = 1, ..., m$, there is an edge of G_d from a node of c_i to node v_i of c, where $v_i \neq v_j$ for $i \neq j$. We go around the cycle c in clockwise order starting from any node that is not a v_i if there is such a node, or from an arbitrary node otherwise. If we encounter two adjacent v_i's, we merge the corresponding c_i's with c as shown in Figure 7.2(a). If the clockwise next node to v_i, call it v, is not a v_j, or if v_i is the last unexamined node in the clockwise traversal,

then we merge c_i with c as shown in Figure 7.2(b).

Finally, consider a path of length 2 in the decomposition of F. It corresponds to three cycles of G_d connected as in the left part of Figure 7.2(c). Note that the two indicated nodes x, y of the middle cycle may coincide because the middle cycle is the head of one arc and the tail of the other arc of length-2 path of F, and thus, this is not a special case of Figure 7.2(a). In this case we merge the corresponding cycles as in Figure 7.2(c). Notice that, by now, all cycles have edges of length 2 (if the edges added in Figure 7.2 are of length 1, we simply increase their length to 2, which cannot decrease the length of the tour constructed) and can thus be merged together at no extra cost.

Lemma 7.3 The tour produced by the algorithm is of length at most $\frac{11}{9}n$.

Proof The cost of the tour produced is n (the cost of the 2-matching) plus the cost of all mergings of cycles according to Figure 7.2. We shall show that each of the mergings of Figure 7.2 cost at most $\frac{2}{9}$ per node involved. This certainly holds in the case of paths of length 2 (Figure 7.2(c)), in which at least nine nodes are involved (at least three for each of the three cycles), and an extra cost of 2 is incurred. For in-trees, we charge the merging of Figure 7.2(a) to the vertices of the two nonroot cycles, which are at least 6, and thus the cost is $\frac{1}{6}$ per vertex. For Figure 7.2(b), we charge the nodes of the nonroot cycle plus the corresponding v_j and the next node v of the root cycle, and thus the cost is $\frac{1}{5}$ per node. It is clear that n node of the graph is charged in two mergings, and thus the bound is established.

We next deal with the case in which the optimum cost is bigger than n. Again, we start with the optimum 2-matching. In fact, we pick a 2-matching which (a) has at most one cycle with a length-two edge (all other cycles are called *pure*), and (b) there is no length-two edge on the nonpure cycle incident upon a length-one edge of the graph going to another cycle. If either condition is violated, then we can merge two cycles at no added cost to obtain a 2-matching with fewer cycles, and this must end.

We now form the bipartite graph B as before, only restricting the left-hand side to the *pure* cycles, and find the maximum matching (since we do assume a Hamilton cycle, this matching may not be perfect with respect to the pure cycles). As before, let $F = (C, A)$ be the directed graph with $(c, c') \in A$ whenever c is matched to a node of c'. Now F is only a partial function; i.e., some nodes (namely, the nonpure cycle and the unmatched pure cycles) have out-degree 0.

From F we can form as in Lemma 7.2 a spanning subgraph F' whose nontrivial components are in-trees of depth 1 and paths of length 2; F' may have also some trivial components (isolated nodes) which are unmatched pure cycles or the nonpure

cycle of the 2-matching. We merge the cycles in the nontrivial components of F' as before, then we merge the remaining pure cycles that are isolated at a cost of 1 each, and finally the nonpure cycle, if it is isolated, at no cost.

Suppose that the optimum 2-matching has k edges of weight 2; obviously, the 2-matching has cost $n + k$, and the optimum traveling salesman tour has cost at least that. Fix an optimum tour $v_1, v_2, ..., v_n, v_1$, let U be the set of nodes v_i such that the edge $[v_i, v_{i+1}]$ has length 2, and let c_2 be the number of pre cycles that contain a node from U. Clearly, the optimum tour has cost at least $n+c_2$. Let r_2 be the number of pure cycles that are isolated in F', and let n_2 be the total number of nodes in these cycles. Since these r_2 pure cycles are unmatched, and one can easily obtain from the optimum tour a matching of B with at most c_2 unmatched cycles, it follows that $c_2 \geq r_2$.

The cost for merging the cycles in the nontrivial components of F' is $\frac{2}{9}$ per node charged. We claim that the number of charged nodes is at most $n - n_2 - k$. To see this, observe that even if the nonpure cycle is in a nontrivial component of F', it is the root of such a component, and every node of the nonpure cycle that is charged is preceded by a length-one edge, because of property (b) above. Therefore, at least k nodes of the nonpure cycle are not charged. Thus, the total cost of the tour constructed by the algorithm is now

$$cost \leq n + k + \frac{2}{9}(n - n_2 - k) + r_2.$$

However, since $r_2 \leq n_2/3, c_2$,

$$cost \leq \frac{11}{9}n + \frac{7}{9}k + \frac{1}{3}c_2 \leq \frac{11}{9}\max\{n + c_2, n + k\},$$

which is at most $\frac{11}{9}$ times the optimum traveling salesman tour. We have shown the following

Theorem 7.1 We can find a tour of length no worse than $\frac{11}{9}$ times the optimum in $O(n^{2.5})$ time.

We can improve on this bound by using a powerful result due to Hartvigsen [24]. He developed a polynomial-time algorithm for finding the optimum 2-matching which contains no triangles. Running the algorithm we presented starting from the optimum triangle-free 2-matching, the previous calculation becomes

Theorem 7.2 We can find a tour of length no worse than $\frac{7}{6}$ times the optimum in polynomial time.

Notice that there are two worst-case types of instances for our algorithm: The one in which the bipartite matching results in a set of cycles of length 3, where all nodes are themselves cycles of length four; and the one in which we have k cycles of length 4 arranged around a cycle of length $2k$.

7.3 Lower bound

Is there a limit to how close to 1 we can get by polynomial-time approximation algorithms (assuming that $P \neq NP$), or is it the case that any ratio can be achieved? We show next that the Traveling Salesman Problem with distances 1 and 2 ((1,2)-TSP) is ***MAX SNP-hard*** [38]. This implies that, if such an approximation scheme existed, then one would exist for a wide class of optimization problems, including several variants of maximum satisfiability, node cover, independent set in bounded-degree graphs, etc. Due to the PCP theorem, there is no PTAS for the maximum satisfiability problem (assuming that $P \neq NP$), so there is no PTAS for any of the problems that belong to the MAX SNP class, including the TSP (and the (1,2)-TSP).

MAX SNP is the class of maximization problems that can be expresses as

$$\max_{S \subseteq V^r} |\{\overline{x} : \phi(\overline{x}, S, G)\}|$$

where $G \subseteq V^r$ is a given relation (typically a graph, $r = 2$), S is the optimal relation sought, and ϕ is a first-order formula involving the relations G, S and the first-order variables $\overline{x}$. For example, the maximum satisfiability problem with 3-clauses (or 2-clauses), the independent set[3] and node cover problem for bounced degree graphs, the max-cut problems, and others, are in MAX SNP.

Furthermore, all these problems turns out to be MAX SNP-complete. Consider two optimization (maximization or minimization) problems A and B. We say that A L-reduces (for *linearly reduces*) to B if there are two polynomial-time algorithms f and g and constants $\alpha, \beta > 0$ such that:

1. Given any instance a of A, f produces an instance b of B, such that the cost of the optimum of b, $\text{opt}(b)$, is at most $\alpha\text{opt}(a)$; and furthermore
2. Given any (feasible) solution y of b, g can produce in polynomial time a solution x of a such that $|cost(x) - opt(a)| \leq \beta|cost(y) - opt(b)|$.

The basic facts about L-reductions are these: First, they are indeed reductions (property (2) says that we can find the optimum of A given the optimum of B).

[3]An independent set of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that each edge in E is incident on at most one vertex in V'. The independent set problem is to find a maximum-size independent set in G

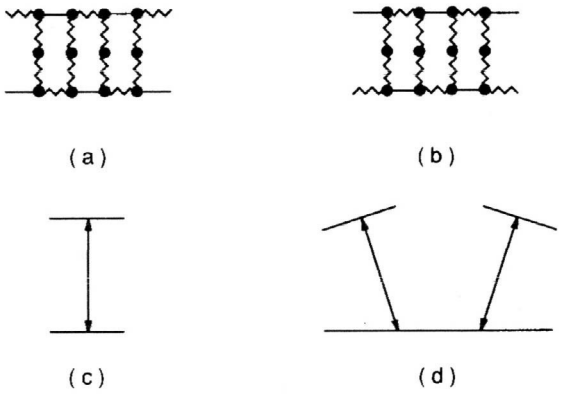


Figure 7.3: *The Exclusive OR (XOR) Device*

Second, the composition of L-reductions is an L-reduction. Third, if A L-reduces to B and there is a polynomial-time approximation algorithm for B achieving a ratio of $1+\epsilon$, then there is a polynomial-time approximation algorithm for A with ratio $1+\alpha\beta\varepsilon$. In view of this, we can think that MAX SNP also contains certain minimization problems: Those that are L-reducible to a problem in MAX SNP.

A problem in MAX SNP is MAX SNP-complete if all problems in MAX SNP L-reduce to it. As usual, if this condition is met but the problem is not known to be in MAX SNP (such a problem is the Travelling Salesman Problem with distances 1 and 2), it is called MAX SNP-hard. It follows from the above that if a MAX SNP-hard problem had a polynomial-time approximation scheme, then every problem in MAX SNP would had one. As we mentioned above, there is no polynomial-time approximation scheme for any of these problems.

We show the following theorem:

Theorem 7.3 The Traveling Salesman Problem with distances 1 and 2 is MAX SNP-hard.

Proof We shall L-reduce 3SAT with at most four occurrences of each literal (a problem known to be MAX SNP-complete [38]) to the (1,2)-Traveling Salesman Problem. We are given a formula with n variables $x_1, ..., x_n$ and m clauses with three literals each $C_1, ..., C_m$, such that each variable appears a most four times positively and at most four times negatively. We are asked to find the truth assignment which maximizes the number of clauses satisfied.

Given such a formula, we shall construct an instance of the Traveling Salesman

Problem with all distances either one or two. In fact, we shall exploit the connection between the (1,2)-TSP and the Hamilton cycle problem and, instead of an instance of the (1,2)-TSP, we shall construct a graph G. The Traveling Salesman Problem instance will be implied if we consider all edges as having length one, and all nonedges as two. The construction is based on several specialized devices.

One basic device is the exclusive-or shown in Figure 7.3(a) (from [37]). If the graph is to have a Hamilton circuit, and there are n connections of the nodes in the device to other parts of the graph, except of the four endpoints, then this graph behaves as if two edges 1-1' and 2-2' are constrained so that exactly one of them is traversed by the Hamilton circuit. The two possible traversals of this subgraph by a Hamilton circuit are shown in Figures 7.3(a) and 7.3(b).

We shall sometimes think of an exclusive-or subgraph as two edges connected by an extrinsic device (Figure 7.3(c)). We can arrange exclusive-or's in series by, say, identifying the lower-right edge of one with the lower-left edge of the next (Figure 7.3(d)); we think of this as two consecutive edges connected by exclusive-or with two other ones. We use a lot in the next device.

For each variable we have the device shown in Figure 7.4. It consists of two paths, presenting alternative traversals, which capture the choice of truth value for the variable. Each of these paths is an arrangement of 29 exclusive-ors. In particular, there are five "batteries" of five edges each on each path (a total of 50 edges). Each of these 25 edges of one path is connected b exclusive-ors with one of the 25 edges of the other path, so that for each battery on one path and each battery on the other there is an exclusive-or connection between two edges of these batteries. The precise order is not important.

In any portion of a Hamilton path from 1 to 2, one of these two paths will be traversed, and this corresponds to the two possible truth values for the variable. Between any two consecutive batteries on one path, say the one corresponding to $x_i = true$, there is an additional edge. Thus, there are four such edges on the path, which correspond to the four occurrence of the literal $\overline{x}_i$ (similarly for $x_i = false$ and x_i). These four edges are called *occurrence edges*.

For each clause we have the triangle device shown in Figure 7.5. Each edge corresponds to the occurrence of a literal in this clause, and it is connected by an exclusive-or to an occurrence edge of the device for the variable which corresponds to that occurrence.

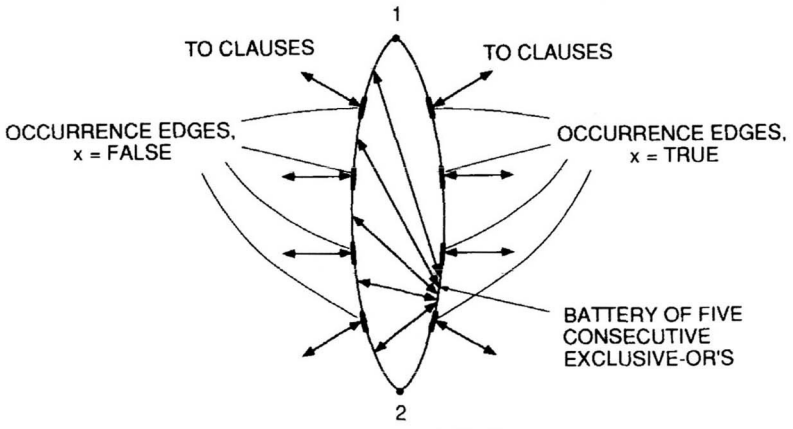


Figure 7.4: *The Variable Device*

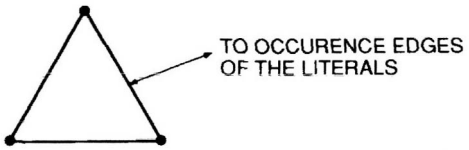


Figure 7.5: *The Clause Device*

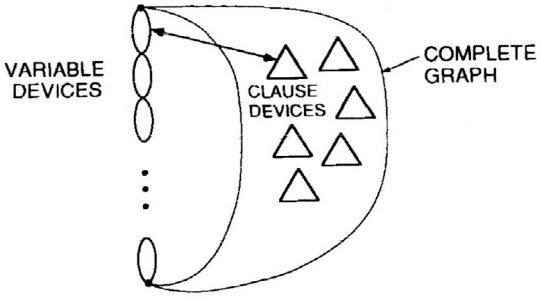


Figure 7.6: *The Overall Graph*

As for the overall graph, it is a set of n variable devices connected in series, and a set of m clause devices connected so that all $3m$ nodes of the m triangles are pairwise connected among themselves, and are also connected to the first and last nodes of the variable part (Figure 7.6).

It has been showed that graph G has a Hamilton circuit if and only if the formula is satisfiable. In particular, the clause device is such that, if all three literals are false by the choices at the variable devices, then all three edges of the triangle will be traversed, and thus a "short circuit" will be closed, contradicting Hamiltonicity. But here our interest lies in the more involved optimization problem: we must show that the construction is an L-reduction. Condition (1) of the definition of L-reduction is immediate. The optimum of the satisfiability problem is $\Theta(m) = \Theta(n)$, and the optimum of the Traveling Salesman Problem is at most $c(m+n)$, for some $c < 1000$.

For condition (2), we should first notice that any tour of this instance of the Traveling Salesman Problem uses certain edges and perhaps certain nonedges of G. Since all nonedges have identical cost 2, we can identify a tour with its parts that are paths of G. That is, certain nodes of the graph, called the ***endpoints***, are traversed by the tour through an edge of length one and an edge of length two (there may also be certain *double endpoints*, nodes that are traversed through two edges of length two; these endpoins will count as two endpoins each). Thus, a tour has cost $N + \lceil E/2 \rceil$, where N is the number of cities and E is the number of endpoints. Now, given a tour with E endpoints, we shall show how to exhibit a truth assignment with at most E clauses unsatisfied; and vice versa. This would establish condition (2) for L-reduction.

Suppose that we are given a truth assignment that satisfies all but k of the clauses. We can construct a tour with either $k + 1$ or k endpoints (depending on

whether k is odd or even) as follows: The variable devices are traversed in the way suggested by the assignment. The tour then traverses triangles corresponding to the satisfied clauses, and then the unsatisfied ones. In the latter part, an endpoint is introduced at each unsatisfied clause, and perhaps an extra one to complete the tour.

Conversely, suppose that we are given a tour with E endpoints. We shall show that we can assume that this tour has a certain structure that makes it very similar to the tour constructed above (otherwise, we can produce a better tour).

First, we can assume that all exclusive-or graphs are traversed either in the intended way (Figure 7.3), plus in three more ways (Figure 7.7) and their symmetric counterparts. We show this by a case analysis, based on the number of boundary edges (the four edges connecting the exclusive-or subgraph with the rest of the graph.) If no boundary edge is traversed, then the optimal traversal of the nodes of the subgraph s obviously as in Figure 7.7(a); there are two endpoints within the subgraph. If one boundary edge is traversed, then it is suboptimal to exit the subgraph before we traverse all its nodes, and thus Figure 7.7(b) results. If two boundary edges are traversed, suppose first that they are both lower boundary edges. Then it is suboptimal to traverse the subgraph in any other way than the ones shown in Figure 7.3. Finally, suppose one of them is a lower boundary edge, and the other is an upper one. Then, at least two endpoints exist within the subgraph (one for each boundary edge traversed), and we may replace this traversal with the one in Figure 7.7(b) at no extra cost.

If all four boundary edges are traversed, then again there are at least two endpoints in the subgraph, and we can replace this traversal at no extra cost by the traversal of Figure 7.3(a) or 7.3(b). Finally, suppose that three boundary edges are traversed. It is impossible to achieve anything better than Figure 7.7(b), since at least one endpoint must occur. However, if the two lower boundary edges are not connected as in Figure 7.7(c), then we can transform this into a traversal in Figure 7.3 by omitting the upper left boundary edge, and adding the horizontal edge next to it.

Hence, we can assume that each exclusive-or of G is traversed according to one of these four traversals (Figure 7.3, Figures 7.7(a), 7.7(b) and 7.7(c)). It is useful for following the arguments below to consider each exclusive-or as an extrinsic mechanism that connects two "edges" of G (the upper edge and the lower edge in Figure 7.3(c)). If an exclusive-or is traversed as in Figure 7.3(a) we shall say that the upper edge is *traversed*, and the lower edge is *untraversed*. In Figure 7.7(a) we say that both edges are untraversed. In Figure 7.7(b) the upper edge is untraversed,

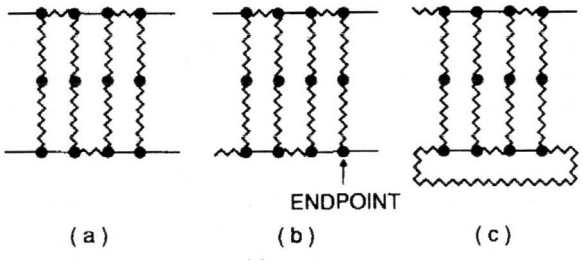


Figure 7.7: *Semi (a) and Sesqui- (b) Traversal*

and the lower edge is semitraversed. Finally, in Figure 7.7(c) we say that the lower edge is sesquitraversed, and the upper edge semitraversed.

Next, we shall show that we can assume that all devices corresponding to variables are traversed in a way consistent with a truth assignment. The difficulty in showing this is the following: Suppose that these devices are traversed so that all eight occurrence edges are traversed. This may have the effect of making the variable both true and false, and thus saving up to four endpoints at clauses that are otherwise unsatisfied. We shall have to show that more than four extra endpoints must be introduced elsewhere in the variable subgraph, so that such traversals can be ruled out. The batteries of exclusive-or's are our way of ensuring this.

Consider an occurrence of a literal. We assign to it a truth value of *true*, *false*, or *expensive*. An occurrence is expensive if either the occurrence edge corresponding to it is semitraversed, or an edge in the battery preceding it is semitraversed. Otherwise, it is true if it is traversed or sesquitraversed, and false if it is untraversed. Notice that expensive occurrences cost us one endpoint each.

We claim that there are no two contradicting literals x and $\overline{x}$ that are both true. In proof, suppose they were. Then all exclusive-or edges on the batteries preceding them are either traversed or sesquitraversed. However, two of these edges are, by our construction, connected by an exclusive-or, and hence this exclusive-or has both edges traversed or sesquitraversed, which is absurd.

Now consider the occurrence edges corresponding to a clause. If they are false or expensive, there is at least an endpoint that can be attributed exclusively to the clause: If at least one is expensive, the endpoint is the one of the semitraversal of the occurrence edge or at the battery before; if they are all false, then one of the edges of the triangle must be semitraversed, otherwise a short cycle would be created. Hence, the truth assignment that makes a literal true if there is a true occurrence in the above sense (and makes an arbitrary decision if no occurrence of the literal or its negation is true) satisfies all but at most E of the clauses.

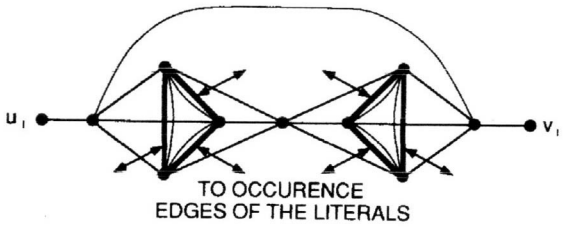


Figure 7.8: *The Modified Clause Device*

The graph G of length-one edges in the above construction has unbounded degree. A simple modification of the reduction can be used to show that the problem is hard even if the graph has bounded degree, a fact which is useful for further reductions [13].

Corollary 7.1 The Traveling Salesman Problem with distances 1 and 2 is MAX SNP-hard even if restricted to instances where the graph formed by the length-one edges has bounded degree.

Proof We sketch the modifications in the construction for a bound of six on the degree. With a little more care the bound can be reduced to four. Let F be a formula in 3CNF with clauses $C_1, ..., C_m$ where each literal occurs at most four times. Let F' be the formula that contains two copies C_i, C_i' of each clause of F. Starting from F', we use the same variable device as in the previous construction, except that of course now the number of occurrences has doubled, so the device is doubled accordingly. For the clause device we use the graph shown in Figure 7.8. The two triangles of heavy edges correspond to the two clauses C_i, C_i' and their edges are connected with exclusive-or gadgets to the corresponding occurrence edges in the variable devices. The variable devices are connected in series as before, followed by the clause devices which are also connected in series, i.e., u_1 is identified with the last node of the last variable device, and u_i for $i > 1$ is identified with v_{i-1}. Finally, we have node s connected only to the first node of the first variable device, and another node t connected only to v_m.

If the optimum truth assignment leaves k clauses of F unsatisfied (thus, $2k$ clauses of F'), then as in the proof of Theorem 7.3, every tour must have at least $2k+2$ endpoints: the nodes s, t, and one other endpoint for every unsatisfied clause of

F'. Conversely, we can construct such a tour by connecting the following $k+1$ paths. One path starts at s, traverses the variable devices according to the optimal truth assignment, goes through all the u_i and v_i nodes traversing the satisfied clauses, and terminates at node t. For every unsatisfied clause C_i of F, there is a path that traverses all the (other) nodes in the corresponding clause device; the two endpoints of this path are a node in the triangle of C_i and a node in the triangle of C_i'.

It follows from the above constructions that the usual variants of the TSP problem are also MAX SNP-hard: find the shortest Hamilton path (instead of tour), find the best path when in addition we have specified a start node or an end node or both.

As we have already said in section 3.2, there are two ways of defining polynomial time approximation schemes depending on whether the performance guarantee is absolute or holds only asymptotically as the optimum value becomes large (absolute polynomial time approximation scheme and asymptotic polynomial time approximation scheme). Theorem 7.3 implies that the Traveling Salesman Problem with distances one and two most likely does not have a scheme of the first kind. We show now that, under conditions satisfied by our problem the two kinds of approximation schemes coincide.

Theorem 7.4 Let A be an optimization problem. Suppose that for any constant k there is a polynomial algorithm (with the polynomial arbitrarily depending on k) for finding an optimum solution for instances of A with optimum value at most k. Then A has an absolute polynomial time approximation scheme if and only if it has an asymptotic polynomial time approximation scheme.

Proof Clearly, if A has an absolute scheme then it has also an asymptotic one. Suppose that A has an asymptotic scheme with constant C. The absolute scheme is this: Given ϵ, first run the algorithm that tests whether the optimal value is at most $2C/\epsilon$ (a constant). If successful, we have the optimum. Otherwise, we run the asymptotic scheme with approximation ratio $\epsilon/2$. If A is a minimization problem, then

$$SOL \leq (1 + \frac{\epsilon}{2})OPT + C \leq (1 + \epsilon)OPT$$

because OPT $\geq 2C/\epsilon$. Similarly, if A is a maximization problem, then

$$SOL \geq (1 - \frac{\epsilon}{2})OPT - C \leq (1 - \epsilon)OPT.$$

As a corollary, for the Traveling Salesman Problem with distances one and two, either there is no asymptotic scheme, or there is an absolute one(an unlikely event by Theorem 7.3). The same result is true (with similar proof) if we allow in the definition of an asymptotic scheme the additive term to be any function, possibly

depending on ϵ, that grows as little-oh of OPT, instead of just being a constant C, i.e., if, for example in the case of minimization problems, we require only that $SOL \leq (1+\epsilon)OPT + f_\epsilon(OPT)$, where $f_\epsilon(n)/n$ goes to 0 as n goes to infinity. The distinction between absolute and asymptotic schemes is meaningful only for problems such as graph coloring ore bin packing, whose restriction to constant cost is difficult.

So, it has now been proved that (1,2)-TSP has not an approximation scheme.

Another fact we have to say here is that the currently best known explicit inapproximability bound for (1,2)-TSP is 741/740 [20].

Chapter 8

A RECENT 8/7-APPROXIMATION ALGORITHM FOR (1,2)-TSP

In the previous chapter we presented the Traveling Salesman Problem with distances one and two ((1,2)-TSP), and we gave a very interesting approximation algorithm, which approximates the problem by a ratio of 7/6. We also showed that we cannot have a PTAS for this problem. Finally, we said that the currently best known explicit inapproximability bound for (1,2)-TSP is 741/740. In this chapter, we will present another approximation algorithm, which is very recent (introduced in 2005) and improves the approximation ratio of the problem to 8/7.

8.1 Introduction

We formulate now our main theorem:

Theorem 8.1 There exists a polynomial time approximation algorithm for the (1,2)-TSP with approximation ratio 8/7.

The formulation of the main algorithm of Theorem 8.1 (Section 8.3) and its analysis is contained in the subsequent sections of the chapter.

We present in this chapter a new method of so-called ***small step improvements*** on the sets of path covers, and the auxiliary notions of justifications, consistency and a color alternating path. This method has also other algorithmic applications.

8.2 The Equivalent Statement

We can represent an instance of (1,2)-TSP as a graph G in which nodes are points of the metric and edges are pairs of points in distance 1. Suppose that G has n nodes and we can find a path cover with k paths (these paths have to be simple and node-disjoint). Then, these paths have $n-k$ edges and we can connect them into a tour, with steps from a path end to a path beginning having cost 2; thus the cost of this tour is $n+k$. Thus, our problem is to minimize k. Moreover, if an optimum solution has cost $n+k^*$, and our goal is to approximate it within factor $\frac{8}{7}$, it suffices to find a path cover with no more than $\frac{1}{7}n + \frac{8}{7}k^*$ paths.

8.3 Small Step Improvement Algorithm

We will investigate the following approach. We maintain a tentative solution that is represented as edge set A (A stands for algorithm's solution), A is a 2-matching (i.e. no more than two edges of A are incident to any given node) that defines, say, k_A paths and cycles with m_A nodes in the cycles. We can alter this solution using an edge set C (C stands for change) into a new solution $A \oplus C$ (here $\oplus$ is the symmetric difference). We say that C improves A if

1. $A \oplus C$ is a 2-matching,
2. either $k_{A\oplus C} < k_A$ or
3. $k_{A\oplus C} = k_A$ and $m_{A\oplus C} > m_A$.

Suppose that for a certain constant K the following holds true:

($\bigstar$) either $k_A \leq \frac{1}{7}n + \frac{8}{7}k^*$, or there exists a C that improves A and $|C| \leq K$.

Then we can use the following algorithm:

K-IMPROV Algorithm

- start with $A = \emptyset$
- while you can find C of size at most K that improves A replace A with $A \oplus C$.

Clearly, we cannot perform n improvements of kind (2) because we would get zero as the number of paths and cycles. We also cannot perform n improvements of kind (3) without an improvement of kind (2) because we would get more than n nodes in the cycles. Hence we cannot perform n^2 improvements. Each search for an improvement takes a polynomial time (where the polynomial depends on K) and when it fails, we terminate and A is a satisfactory solution.

Obviously, K-IMPROV Algorithm runs time $O(n^K + 4)$. In the remaining part of the chapter we will prove ($\bigstar$) for $K = 21$.

8.4 Alternating paths

8.4.1 Definitions

To analyze the algorithm, we introduce the notions of the *auxiliary graph* $\mathcal{G}$, *paths*, *cycles*, possible *initial* edges and nodes, *consistency* of initial nodes, $\mathcal{AP}$s and *justification points*.

In the analysis of Small Improvement Algorithm we fix an optimum solution, a 2-Matching B such that $k_B = k^*$ (B stands for the *best*). Using B define graph $\mathcal{G}$ which has all the same nodes as G. Let D be the set of edges that have both ends in the same cycle of A. $\mathcal{G}$ has edge set $A \bigcup B - D$ which we divide into three *colors*, white color $A - B - D$, black color $B - A - D$ and gray color $A \bigcap B - D$.

An *alternating path*, $\mathcal{AP}$ for short, is a path that starts and ends with a black edge and in which black and white edges alternate. At some point we will relax this notion by allowing to substitute white edges with gray ones.

For A-objects we define *initial nodes*. A node is *initial* if we allow it to be the first or last node of an $\mathcal{AP}$ and it is owned by an A-object. For an A-path, the initial nodes are the endpoints. For an A-cycle C we designate a pair of initial nodes such that C has a Hamiltonian path with these nodes as the endpoints, and this path can be extended with two black edges to another two nodes. We will show that such node pairs can be found in A-cycles with fewer than 8 nodes.

In this proof we will use the notion of *justification points*. If B consists of k^* paths, the optimum cost is $n+k^*$ and A is good enough if it has cost at most $\frac{8}{7}(n+k^*)$, i.e. it consists of at most $\frac{1}{7}n + \frac{8}{7}k^*$ A-objects. We create $n + 8k^*$ *justification points* and to prove that solution A is good enough each A-object has to collect 8 points. A node that is incident to $2 - a$ edges of B has $1 + 4a$ points (the sum of these a's equals $2k^*$). A path starts with the justification points of its endpoints ad a cycle starts with the justifications of all its nodes. The remaining points will be collected by $\mathcal{AP}$s; an $\mathcal{AP}$ gives the collected points to the A-objects that contain its initial nodes. After we "break" certain $\mathcal{AP}$s, they may contain only one initial node and thus deliver the collected points to only one A-object.

Typically, an A-path has two endpoints and each of them is an initial node of an $\mathcal{AP}$ that should give it $2\frac{1}{2}$ points. Similarly, a typical cycle has $4 + a$ nodes, it has two initial nodes of $\mathcal{AP}$s that should give it $(3 - a)/2$ points.

There can be several deviations from the typical case. An A-object can have fewer than two initial nodes; in such a case it collects more justifications from the nodes it contains. If an A-object is an A-singleton, then each initial edge should give it 3 points. If a cycle has more than 6 nodes, it will own no initial nodes. A

node incident to only 1 edge of B has 4 additional points and we omit easy special cases provided by such nodes.

8.4.2 Very Small Improvements

In some situations we have small improvements that insert only one edge. We will discuss this cases, and in further analysis we may assume that they do not occur.

A black edge e that connects two initial nodes is one such case. If e connects initial nodes from two different A-objects, inserting e merges these two objects into one; when we merge an A-cycle we have to remove one of its edges. If e connects initial nodes of a single A-object, this must be an A-path and inserting e converts that path into a cycle.

An edge e that connects an A-singleton with another A-object is another such case, except when A connect an A singleton with a midpoint of an A-path with exactly 3 nodes. Otherwise we have an improvement of kind (4).

Now suppose that we do not have a very small improvement and we have an $\mathcal{AP}$, say $\mathcal{R}$, that starts at u, and $\{u\}$ is an A-singleton. Then for an A-path (v, w, x) and some y, path $\mathcal{R}$ starts with (u, w, v, y). When we consider $\mathcal{R}$ as a possible part of an improvement, we have an option of using an "abbreviated" version that starts with (v, y); on one hand we will "forget" that $\mathcal{R}$ starts at an A-singleton and thus needs to collect an extra $\frac{1}{2}$ point; one the other, we will forget that $\mathcal{R}$ collected $\frac{1}{2}$ point at node w.

8.4.3 Examples with $\mathcal{AP}$s

An alternating path by itself can define an improvement. The $\mathcal{AP}$ at the left of Figure 8.1 consists of 4 black edges and 3 white ones. If we apply its set of edges as a change, we will get 4 paths (shown underneath) where before we had 5. What happened is that the $\mathcal{AP}$ changed the number of solution edges that are incident to path ends from 1 to 2, ad it did not change this number for intermediate nodes. Therefore we decreased the number of path ends by 2, hence the number of paths by 2. If the ends of the $\mathcal{AP}$ belong to cycles, the situation is similar; in the second example in Figure 8.1 the number of path ends does not change but we have one less cycle, and therefore fewer objects.

An interesting special case (see Figure 8.2 right) occurs when $\mathcal{AP}$ starts and ends at the same cycle. Then we obtain an improvement if the first and the last nodes of this $\mathcal{AP}$ are the initial nodes of the cycle; because these nodes are consistent, we can include the traversal of this cycle in the improved solution.

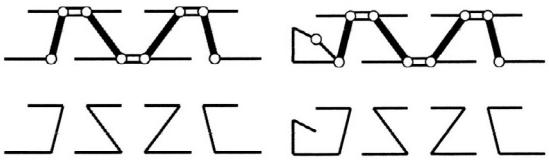


Figure 8.1: *Example of $\mathcal{AP}$*

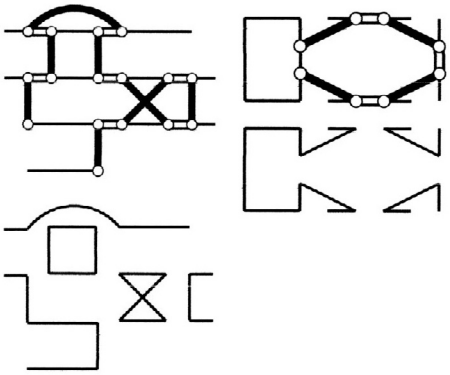


Figure 8.2: *Interesting special case: $\mathcal{AP}$ starts and ends at the same cycle*

Finally, an $\mathcal{AP}$ may fail to provide an improvement if it creates cycles. Even if the number of path ends decreases by two the number of paths and cycles may increase if we create two new cycles in the process. In the example at the left of Figure 8.2 we started with 4 paths and we changed them into 3 paths and 2 cycles.

8.4.4 Initial Edges of Cycles

Let C is a cycle of A with at most 7 nodes with $|C|$ *justification* points (i.e. with all nodes adjacent to two edges of B).

Let $\widehat{C}$ be the set of nodes of C and $\widehat{K} \subset \widehat{C}$ be the set of nodes incident to black edges. In this subsection we show that certain two nodes of $\widehat{K}$ are *consistent* in the sense that they are endpoints of a Hamiltonian path of $\widehat{C}$.

If $|\widehat{K}| = 2$, then $\widehat{K}$ is a *consistent pair* because the set of edges of B that are contained in C forms a single path. Hence we assume that $\widehat{K} \geq 3$.

Suppose that two nodes of $\widehat{K}$, u and v, are adjacent on the cycle C, ten u

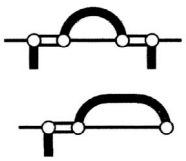


Figure 8.3: *Two different S-arcs*

and v are consistent because we can form a path by removing edge $\{u, v\}$ from C. Therefore we can assume that nodes of $\widehat{K}$ are not adjacent on C, hence $|\widehat{K}| \leq |\widehat{C}|/2$; this means that $|\widehat{K}| = 3$ and $|\widehat{C}| = 6$.

Because $\widehat{K}$ has 3 nodes, B covers $\widehat{C}$ with two paths, one of these paths has 1 node and no edges, the other has 5 nodes and 4 edges, hence 2 edges of B are contained in $\widehat{C} - \widehat{K}$. Without loss of generality C is a cycle $(u_0, ..., u_5)$, $\widehat{K} = \{u_0, u_2, u_4\}$ and $\{u_1, u_3\} \in B$; thus we can traverse $\widehat{C}$ with $(u_0, u_5, u_4, u_3, u_1, u_2)$.

8.4.5 $\mathcal{AP}$s with Deficit-General Method

According to our rules, an $\mathcal{AP}$, say $\mathcal{AR}$, collects $\frac{1}{2}$ point for every of its nodes except endpoints of paths and cycle nodes; the reason we do not *a priori* collect more is that each of these nodes may belong to two different $\mathcal{AP}$s.

We will use several methods, we break $\mathcal{AP}$s that traverse through a cycle created by $\mathcal{R}$; if one of the broken $\mathcal{AP}$s, say $\mathcal{P}$, in short, we can merge this cycle with the A-object that owns the initial point of $\mathcal{P}$, if $\mathcal{P}$ is long, it does not need to collect points from its edge in the cycle, and we can transfer this point to $\mathcal{R}$.

8.4.6 Avoiding Bad Cases

S-arcs-Avoiding Them or Finding Extra Points for Them

Arcs, the black edges contained in paths, are potentially troublesome because they make it possible for a short $\mathcal{AP}$ to create more cycles as they allow to obtain a cycle from a single A-path fragment and one black edge (the arc). In this case, this arc is preceded and followed by a white edge directed away from it - or by a path endpoint. We will call it S-arc, for S(hort cycle making) arc. The number of nodes on the path fragments that connects the endpoints of an arc will be called the *length* of this arc.

We will avoid the creation of S-arcs by making decisions about the decomposition of black and white edges into a set of $\mathcal{AP}$s. When we will not be able to make such a decision, we will be able to endow such S-arcs with gray edges that will provide them with extra points.

To apply these techniques we consider a *chain* of arcs, say $\mathcal{Q}$, which is a path formed by arcs that are contained in an A-path, say $\mathcal{P}$. For the sake of uniformity, we extend $\mathcal{P}$ in both direction with a single *phony white edge*. We will make decisions how to connect the elements of a chain with the adjacent white edges when we are forming $\mathcal{AP}$s. Connecting an arc to a phony white edge implicitly designates it to be an initial edge of its $\mathcal{AP}$.

The decisions about the decomposition within a chain are dependent: a node u incident to two arcs of $\mathcal{Q}$, say a_0 and a_1, is also incident tov two white edges of $\mathcal{P}$, say e_0 and e_1. The decision at u may connect a_0 with e_0 and a_1 with e_1, or, alternatively, it may connect a_0 with e_1 and a_1 with e_0.

We first pick an arc a_0 of $\mathcal{Q}$ as the "least priority"; next, we assure that no other arc in $\mathcal{Q}$ is an S-arc. We start from any end of $\mathcal{Q}$ and proceed toward a. Initially we make an arbitrary decision at the endpoint of $\mathcal{Q}$. Inductively, we consider the endpoint of arc $a \neq a_0$ for which we made decision at its other end; if the latter decision connected a with a white edge directed away from it, at the other and we connect a with a white edge directed inward, otherwise we make an arbitrary choice.

Good case (1). An arc a in chain $\mathcal{Q}$ is *directly hit* from an endpoint of an A-path, i.e. a node inside arc a is connected by a black edge with this endpoint. We give a the least priority and we have no adverse consequences. If we want to create an improvement using an $\mathcal{AP}$ that contains S-arc a, then we increase the number of objects by creating a cycle, but then we decrease it back by inserting the edge of the direct hit, and removing an adjacent edge from the cycle.

Good case (2). An endpoint of $\mathcal{Q}$, adjacent to its first arc a, is adjacent to a white edge b directed inward a; we can leave a as the "least priority" and the final decision connecting a with b will assure that it is not an S-arc.

Good case (3). A node u is adjacent to two arcs of $\mathcal{Q}$ and two white edges, positioned as shown in Figure 8.4. We can give a_0 the "least priority", and the decision at u will connect a_0 with e_0 and a_1 with e_1 which assures that neither becomes an S-arc.

Good case (4). $\mathcal{Q}$ consists of one arc only, a, of length 4, and a becomes an S-arc, i.e. a is adjacent to two gray edges that are inside it; we must have the configuration as shown in Figure 8.5. We have two $\mathcal{AP}$s, one with the sequence of edges b_0, e_0, a, e_2, b_3 and another with sequence b_1, e_1, b_2. We can

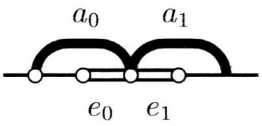


Figure 8.4: *Good case 3*

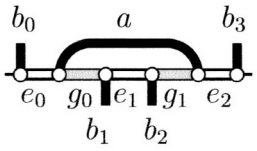


Figure 8.5: *Good case 4*

alter the connections to have sequences b_0, e_0, a, g_1, b_2 and b_1, g_0, a, e_2, b_3, and each of these two $\mathcal{AP}$s can get $2\frac{1}{2}$ points.

Good case (5). $\mathcal{Q}$ contains two consecutive arcs that differ in length by exactly 2, so we have a situation from Figure 8.6. We can give b_0 the least priority because it is in a similar situation to a direct hit. In particular, in an $\mathcal{AP}$ we can replace the edge sequence e_0, b_0, e_1 with a "detour" e_0, b_1, e_2.

Consequently, if $\mathcal{Q}$ consists of only one arc a and we make it an S-arc then a has length at least 5 and is adjacent to two gray edges inside it.

Good case (6). $\mathcal{Q}$ contains two consecutive arcs, say b_0 and b_1, where b_1 is adjacent to an end of path $\mathcal{P}$ and b_0 is not - imagine that in Figure 8.6 edge e_2 is the first edge of $\mathcal{P}$. Then we can give the lowest priority to b_0, because b_1 delivers a direct hit to b_0 in case the latter becomes an S-arc.

Good case (7). $\mathcal{Q}$ contains an arc b adjacent to an endpoint of $\mathcal{P}$, as shown in Figure 8.7. If we do not have case (2), edge g is gray and edge e_0 is white. Suppose that e_1, the other edge of $\mathcal{P}$ that is adjacent to g, is white. We will use a similar method to case (4). We have two $\mathcal{AP}$s, one starts with edge sequence b, e_0, b_0, the other has a fragment b_1, e_1, b_2. We replace them by connecting b with b_1 (rather than b_0), edge sequence b, g, b_1, as well as connecting

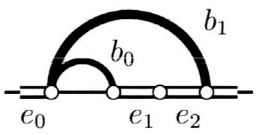


Figure 8.6: *Good case 5*

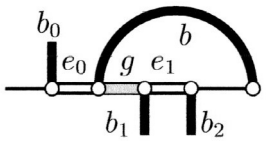


Figure 8.7: *Good case 7*

b_0 with b_2, edge sequence b_0, e_0, b, e_1, b_2. Both of these new $\mathcal{AP}$s use edge b, but the former removes g from $\mathcal{P}$ and collects 1 point, and the latter removes e_0 and e_1 and collects 2 points.

The chains which do not fall into either of the seven good cases are *troublesome*. They may form a *superchain* connected with gray edges. If such a chain contains an endpoint of $\mathcal{P}$, we say that it is *terminal*. Note that a troublesome terminal chain cannot be connected into a superchain with others, because it provides neighboring chains with direct hits.

A terminal troublesome chain contains only arcs adjacent to the endpoint, say u, because we cannot apply the methods of cases (3) and (6), and because (2) and (7) are not applicable, and the other end(s) of its arc(s) are adjacent to a pair of gray edges. Thus if this chain consists of one arc, the $\mathcal{AP}$ starting at this edge collects $2\frac{1}{2}$ points from 3 gray edges, and this arc has length at least 5; and if this chains consists of two arcs, it can collect points from 4 gray edges and the longer arc has length a least 7, the $\mathcal{AP}$ starting at the longer arc collects 4 points.

A nonterminal troublesome chain may have one arc only, in this case it is an arc of length 5 or more and is adjacent to two gray edges, both directed inward. We can pick any of these gray edges to provide an extra point to this arc.

A nonterminal troublesome chain with more than one arc is adjacent to two gray edges at its endpoints. Let us decide that we will collect an extra point from one of them. We will give the least priority to an arc of length at least 6 that contains this gray edge in its interior. Say that the arc adjacent to this gray edge is the first. If the first arc has length 6 or more, we can choose it (to give the least priority). If the first arc has length below 6, we follow the chain until we have the first length increase of more than 1, or from 5 to 6. Observe that after length increase of 1, we cannot finish the chain, and after length increase of 2, we can apply the method of case (5), so after the first larger increase we must have length at least $3 + 3 = 6$.

As we follow the chain, we visit nodes, starting from the two nodes of the initial gray edge, and we cannot return to a visited node, as black and gray edges cannot

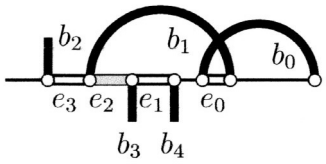


Figure 8.8: *Avoiding terminal short cycles*

form a short cycle. If we increase the arc length by 1 we increase the span of the visited nodes without introducing a new hole, and if we decrease the arc length, we fill a hole. If we start with an arc of length 3, we do not have a hole so we cannot decrease arc length before the first large increase. If we start with an arc of length 4, we have a single hole, so we can decrease the arc length once, but this cannot be the end of the chain, ad we cannot start with a decrease (this would make an arc of length 2, which is not an arc), so if we avoid length 6 we have length sequence 4, 5, 4, and then only increases. If we start with length 5, the first decrease cannot be 1, that closes the black/gray cycle, it cannot be 2, case (5), and it cannot be (3) - no arcs of length 2, so we start with an increase.

Avoiding Terminal Short Cycles

Let $\mathcal{R}$ be an $\mathcal{AP}$ starting at an endpoint of an A-path, say $\mathcal{P}$. We wish to avoid a situation when the first 4 edges of $\mathcal{R}$ - two black and two white- define a change that creates a cycle, say $\mathcal{C}$. If $\mathcal{C}$ consists of one fragment of an A-path, it contains an S-arc and receive extra point(s), and this we accept. If $\mathcal{C}$ consists of two fragments, they need to be flanked on 4 sides; one flank can be provided by the beginning node of $\mathcal{R}$, but three flanks would have to be provided by white edges, so one white edge is used twice, hence $\mathcal{R}$ starts as shown in Figure 8.8 with edges b_0, e_0, b_1, e_3. In this case we treat the triple of edges b_0e_0, b_1 as if it was a single arc, and we apply the methods of the previous section.

If e_2 is a white edge, we change $\mathcal{R}$ to follow e_2 rather than e_3; if that conflicts S-arc avoidance at that node, the other black edge at this node is an arc that is hit either by b_0 or by b_0, e_0, b_1. If e_2 is a gray edge, we give its points to $\mathcal{R}$ and we consider e_1 (if e_2 connects b_1 with b_0, we define e_1 as the edge on $\mathcal{P}$ adjacent to e_0). The reason why e_2 and e_1 can be considered in this fashion is that we can "reconfigure" $\mathcal{P}$ in such a way that e_2 (or e_1) becomes its initial edge; we replace the pair of $\mathcal{AP}$s - $\mathcal{R}$ and the $\mathcal{AP}$ that contains e_1 with: an $\mathcal{AP}$ that uses b_2 and e_3, thus merging some object with one part of $\mathcal{P}$ and creating a cycle from the initial

part, and follows with (quite arbitrarily chosen) black edge that is incident to that cycle; the second $\mathcal{AP}$ makes reconfiguration that makes e_2 the new initial edge and follows with the black edge incident to the new endnode.

If both e_2 and e_1 are gray, $\mathcal{R}$ gets 2 extra points.

8.4.7 $\mathcal{AP}$s with Deficit-Cycle to Cycle

An $\mathcal{AP}$, say $\mathcal{R}$, that connects two cycles should collect $1\frac{1}{2} + 1\frac{1}{2} = 3$ points. Consider the cases when $\mathcal{R}$ does not form an improvement.

In that case $\mathcal{R}$ creates a cycle, so it creates c path fragments and uses b S-arcs where $c + b \geq 2$. In turn, a path fragments must have $2a$ flanks where they are separated from their paths, and these flanks have to be created by white edges of $\mathcal{R}$; at least two white edges create one flank, and no white edge creates more than 2, so we need at least $c + 1$ white edges, hence $\mathcal{R}$ collects at least $c + 1 + b \geq 3$ points.

8.4.8 $\mathcal{AP}$s with Deficit-Large Cycle to Path Endpoint

In this section we consider an $\mathcal{AP}$, say $\mathcal{R}$, that connects a path $\mathcal{P}$ and a cycle $\mathcal{C}_\prime$ of length 6 and which does not define an improvement. It should collect at least $2\frac{1}{2} + \frac{1}{2} = 3$ points.

If $\mathcal{R}$ creates two cycles, we can show that $\mathcal{R}$ collects at least 4 points. Two cycles require b S-arcs and c separated path fragments where $b + c \geq 4$. To separate c path fragments we need to create $2c$ fragment endpoints; only one fragment end can be created by the endpoint of $\mathcal{P}$, and the rest, $2c - 1$ of them, requires $a \geq c$ white edges, hence $\mathcal{R}$ collects $a + b \geq 4$ points.

Now we can assume that $\mathcal{R}$ creates exactly one cycle, say $\mathcal{C}$. The reasoning from the cycle-to-cycle case does not apply only if $\mathcal{C}$ is created from fragments of A-paths and one of the endpoints of these fragments is the initial point of $\mathcal{R}$, say u, that is an endpoint of $\mathcal{P}$. If $\mathcal{R}$ does not collect 3 points, it has at most 2 white edges and $\mathcal{C}$ is a *terminal cycle* as we discussed in case (7) of subsection 8.4.6. As we showed there, we can choose a decomposition of white and black edges into $\mathcal{AP}$s so that either a terminal cycle is avoided, or it receives additional 2 points from a gray edge (both points are needed if $\mathcal{C}$ is created with an S-arc).

8.4.9 $\mathcal{AP}$s with Deficit-Small Cycle to Path Endpoint

In this section we consider an $\mathcal{AP}$, say $\mathcal{R}$, that connects a path $\mathcal{P}$ and a cycle $\mathcal{C}_0$ of length 4 or 5; $\mathcal{R}$ should collect $2\frac{1}{2} + 1\frac{1}{2} = 4$ points (or $3\frac{1}{2}$ points if $\mathcal{C}_0$ has length

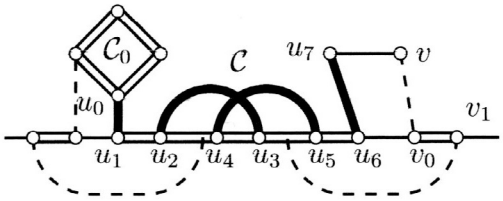


Figure 8.9: $\mathcal{AP}$ with three white edges

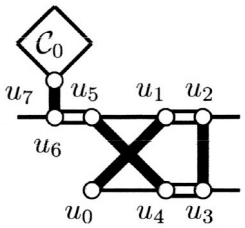


Figure 8.10: $\mathcal{AP}$ with three edges different than the previous

5). If $\mathcal{R}$ creates two cycles, we have a situation already discussed in the previous subsection.

Suppose that $\mathcal{R}$ creates only one cycle $\mathcal{C}$; we would have an improvement if $\mathcal{C}$ contains more than 5 nodes.

Suppose that $\mathcal{C}$ is created with an S-arc and that it has exactly 5 nodes; imagine that (u_2, u_5) in Figure 8.9 is a black arc; note that at both ends of $\mathcal{C}$ we have gray edges. If the left of these edges is adjacent to another possibly troublesome chain of arcs (besides the arc that defines $\mathcal{C}$), the first arc of this chain is directly hit by edge (u_7, u_6) and thus that it does not need an extra point. If the right of these edges is adjacent to another possibly troublesome chain of arcs, then the first arc of this chain is hit by (u_7, u_6, u_5, u_2). Therefore $\mathcal{R}$ can collect points from both of these gray edges, and thus it has 4 points.

If $\mathcal{R}$ contains three white edges, we must have a configuration of Figure 8.9 or Figure 8.10 with $\mathcal{R} = (u_0, ..., u_7)$.

We will break every $\mathcal{AP}$, say $\mathcal{Q}$, containing an edge of $\mathcal{C}$; we claim that there will be enough points for all these $\mathcal{AP}$s plus one point for $\mathcal{R}$.

Suppose that we have broken $\mathcal{Q}$ into $\mathcal{Q}_0$ and $\mathcal{Q}_0$ by removing an edge e that

belongs to $\mathcal{C}$. We consider $\mathcal{Q}_i$ as a possible improvement for $A \oplus \mathcal{R}$.

If $\mathcal{Q}_i$ starts at an A-cycle and it is not an improvement, $\mathcal{Q}_i$ must create a cycle from a fragment(s) of an A-path, and that requires at least two white edges; in that case $\mathcal{Q}_i$ collects at least $\frac{1}{2}$ point more than needed for that A-cycle. There is one exception: $\mathcal{Q}_i$ may start at $\mathcal{C}_0$ that in solution $A \oplus \mathcal{R}$ is en ending portion of a path and we may have an arc that was not an arc in solution A; then $\mathcal{Q}_i$ may have exactly one white edge and it still creates a cycle. However, this case describes an improvement, as the new cycle must have more nodes than $\mathcal{C}_0$.

If $\mathcal{Q}_i$ starts at an A path and it is not an improvement, $\mathcal{Q}_i$ must create a cycle from a fragment(s) of an A path, and this requires collecting at least 3 points; if $\mathcal{Q}_i$ creates a cycle with 2 points, it has at most 2 white edges, so this is a terminal cycle that gets another 2 points. Again, $\mathcal{Q}_i$ has a surplus of $\frac{1}{2}$ point. An exception occurs if $\mathcal{Q}_i$ has at most two white edges and it uses an arc that was not an arc in solution A, i.e. when $\mathcal{Q}_i$ creates a cycle from a single fragment of a path that contains edge (u_6, u_7).

If there exists two such exceptions, such $\mathcal{Q}_i$ and $\mathcal{Q}_j$ form an improvement in conjunction with $\mathcal{R}$. The reason is that both exceptional $\mathcal{AP}$ fragments create a cycle from a single path fragment that contains a particular edge; it follows that one of these fragments must have an endpoint inside the cycle created by $\mathcal{Q}_i$. Now, applying $\mathcal{R}$ as a change replaces cycle $\mathcal{C}_0$ with $\mathcal{C}$ and creates a "composite" path with edge (u_6, u_7); applying $\mathcal{Q}_i$ fuses $\mathcal{C}$ with a path but creates a cycle that contains (u_6, u_7) edge; now the initial portion of $\mathcal{Q}_j$, with at most one white edge, reaches the latter cycle, so it must be an improvement.

Note that a exception $\mathcal{Q}_i$ must have at least one white edge.

Now we can perform the balance of points. If we break $\mathcal{Q}$ and one of the paths is an exception, ne branch of $\mathcal{Q}$ has a surplus of $\frac{1}{2}$, the second branch has a deficit of $1\frac{1}{2}$, and we have also the white edge that was removed from $\mathcal{Q}$; as a result we have neither a deficit nor a surplus. If none of the paths is an exception, we have a surplus of 2 points. Finally, if we had a gray edge, we have 1 point to collect. Note that the interior of $\mathcal{C}$ has a least two edges, so at least one of them brings the surplus.

The difference between the case of Figure 8.9 and the case of Figure 8.10 is the following: One of the nodes of $\mathcal{C}$ is an endpoint of an A-path, so we cannot collect $\frac{1}{2}$ point from that node - this A-path already collected both halfs; as a result we can be $\frac{1}{2}$ point short. This requires that everything is tight: $\mathcal{C}_0$ has 4 nodes only, the interior of $\mathcal{C}$ has only two edges, one is gray,so we break only one $\mathcal{AP}$, say $\mathcal{Q}$, and one of the created branches, say $\mathcal{Q}_0$, starts at a pat endpoint, creates a cycle and

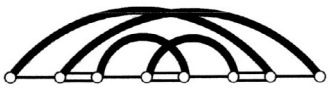


Figure 8.11: $\mathcal{AP}$ with two cycles where none of the cycles is a terminal one.

has one white edge only.

This means that $\mathcal{Q}_0$ starts at an endpoint of a new path joined with edge (u_2, u_3), uses a new arc - that surely is not an arc in solution A - and that its 4th node is u_0, u_1, u_4 or u_5. If it is u_0, $\mathcal{Q}_0$ forms an $\mathcal{AP}$ between two endpoints of A-path, with one white edge only - surely an improvement. If it is u_5, we extend $\mathcal{Q}_0$ with (u_5, u_6, u_7) and we get an $\mathcal{AP}$ from an A-path to an A-cycle with two white edges and no arc - again an improvement. If it is u_4, we extend $\mathcal{Q}_0$ with $(u_4, u_3, u_2, u_1, u_0)$ and we get an $\mathcal{AP}$ between two A-path endpoints with no S-arc, and 3 white edges that are located on two A-paths, in the next subsection we will see that it has to be an improvement. The case of u_1 is similar: we extend $\mathcal{Q}_0$ with $(u_1, u_5, u_4, u_3, u_2, u_1, u_0)$.

8.4.10 $\mathcal{AP}$s with Deficit-Path Endpoint to Path Endpoint

An $\mathcal{AP}$ that connects two path endpoints should collect 5 points or form a small improvement. Otherwise, we have an $\mathcal{AP}$, say $\mathcal{R}$, that creates at least 2 cycles and collects at most 4 points.

If $\mathcal{R}$ creates c cycles, they must contain $2c-a$ fragments of A-paths and a S-arcs. To separate the fragments of A-paths from their paths we need to create $2(2c-a)$ flanks, and at most 2 of them can be the path endpoints, so at least $2(2c-a-1)$ of them is created by white edges, which requires at least $2c-a-1$ white edges.

Suppose that $\mathcal{R}$ creates 3 cycles. Then it collects at least $2 \times 3 - 1 = 5$ points from white edges and S-arcs.

Suppose that $\mathcal{R}$ creates 2 cycles and it collects only 3 points. Then, none of the cycles is a terminal one, and we have a configuration from Figure 8.11, or a similar one, with 2 white edges and one S-arc. We can collect the extra point using the $\mathcal{AP}$ breaking method from the previous subsection.

Note that $\mathcal{R}$ converted an A-path, say $\mathcal{P}$, into a pair of cycles. Suppose that there exists an $\mathcal{AP}$ that starts at another A-path and extends to $\mathcal{P}$, and that it collects less than 3 points before reaching $\mathcal{P}$. Then, this $\mathcal{AP}$ fragment does not

create a new cycle but it merges one of the cycles that replaced $\mathcal{P}$ with another A-path; as a result we have the same number of objects as before the change, but we have more nodes in cycles, an improvement. Now we can assume that such an $\mathcal{AP}$ collects at least 3 points, so it has $\frac{1}{2}$ point surplus before reaching $\mathcal{P}$.

Suppose that there exists an $\mathcal{AP}$ that starts at an A-cycle and extends to $\mathcal{P}$ and it collects less than 2 points before reaching $\mathcal{P}$, this $\mathcal{AP}$ decreases the number of objects in solution $A \oplus \mathcal{R}$; if there are such $\mathcal{AP}$s reaching each of the two new cycles that cover $\mathcal{P}$, we have an improvement. Now we can assume that one of the new cycles, say $\mathcal{C}$, is not reached by any such $\mathcal{AP}$.

We can break all $\mathcal{AP}$s that contain an edge of $\mathcal{C}$ and collect points from its gray edges. In the worst case, $\mathcal{C}$ is the "outer" cycle, and it has only 2 edges on $\mathcal{P}$, one of them gray; we collect surplus of two partial $\mathcal{AP}$s, 1 point, and the points from the edges themselves that were not allocated already to $\mathcal{P}$, 2nd point.

The configuration of Figure 8.11 was based on an assumption that every white edge provides two endpoints for the path fragments of the cycles created by $\mathcal{R}$. In every other case $\mathcal{R}$ collects at least 4 points, and we can use the techniques of the last two subsections to collect another one.

Now suppose that $\mathcal{R}$ creates 2 cycles and collects only 4 points. We can repeat the above arguments and collect points from one of the cycles created by $\mathcal{R}$, say $\mathcal{C}$, that is nt reached by a partial $\mathcal{AP}$ with a deficit - that extends from an A-cycle. However, one $\mathcal{R}$ has a "spare" black edge that merges two fragments of A-paths into a new path, and as a result new arcs and new terminal cycles may exist. Thus we experience the same problems as in the previous subsection. As a result, we need a more detailed analysis for the case when $\mathcal{C}$ contains only one white edge, only one gray edge, and it contains a endpoint of an A-path.

In a case analysis we can omit the case when $\mathcal{R}$ includes an S-arc, because it is treated identically to the case when $\mathcal{R}$ creates a cycle like the inner cycle in Figure 8.11. Therefore we will assume that $\mathcal{R}$ contains exactly 4 white edges, creates 4 path fragments combined into 2 cycles hence it creates 8 fragment flanks, and ***either*** there are 3 white edges that create two flanks each, 1 white edge creating 1 flank, and 1 flank is created by a common endpoint of $\mathcal{R}$ and an A-path, hence or 4 path fragments are contiguous as in Figure 10.11, ***or*** there are 2 white edges that create two flanks each, 2 white edges creating 1 flank each and 2 flanks are created by endpoints of $\mathcal{R}$.

In the first case, it is as if we added to Figure 10.11 a part of Figure 10.9 that is to the left of u_5. Suppose that edge incident to the beginning of $\mathcal{R}$ is white,then we are breaking $\mathcal{Q}$ that contains that edge. Let $\mathcal{Q}_0$ be the branch starting at the

beginning of $\mathcal{R}$, we can show that it should have a surplus of $1\frac{1}{2}$ point. Observe that $\mathcal{Q}_0$ can be used as a complete $\mathcal{AP}$, and suppose that it does not create an improvement. If $\mathcal{Q}_0$ connects to another endpoint of an A-path, it can collect only 3 points only if it is exactly as in Figure 8.11, so it converts an A-path int two cycles. However, edge (u_7, u_6) delivers a direct hit to one of these cycles, so combined with $\mathcal{Q}_0$ it does not change the number of objects but it creates one new cycle. If $\mathcal{Q})_0$ connects to an A-cycle and has at most 2 white edges, then it creates a terminal cycle and gets at least 3 points. Thus we got the extra point that we need.

If that edge is gray, we break $\mathcal{Q}$ that contains the rightmost edge of Figure 8.11, in the position of (u_3, u_5) of Figure 8.9. Suppose that the exception $\mathcal{Q}_0$ comes to u_5, we can extend it to u_3 and then to the beginning of $\mathcal{R}$, and this is a complete $\mathcal{AP}$ with 2 white edges, an improvement. Suppose that $\mathcal{Q}_0$ comes to u_3; we view it as a change to solution $A \oplus \mathcal{R}$; from the outer cycle w remove (u_3, u_5), we add back edge (u_5, u_6) and we remove (u_6, u_7). As a result, we merge $\mathcal{C}$ with the cycle created by $\mathcal{Q}_0$, and the net change is one more cycle.

We skip the case analysis for the second case because it involves the same ideas.

8.4.11 Largest Improvement

The largest improvement that this analysis needs occurs when we have an $\mathcal{AP}$, say $\mathcal{R}$, that connects two ends of A-paths, collects 4 points and creates 5 cycles with "good arcs", where the arcs are good because they have "direct hits" from other path ends; using 4 such hits we merge 4 of the resulting cycles and thus we get an improvement; $\mathcal{R}$ has 4 white edges and 5 blacks, and each "hit" merges two cycles by inserting an edge and removing two, for the total of $5 + 4 + 4(2 + 1) = 21$ edges.

Chapter 9

SUMMARY OF THE PRESENTED ALGORITHMS

We will now summarize the algorithms we presented in this book.

First of all, we presented a *dynamic algorithm*, which solves the general Traveling Salesman Problem. This algorithm is exact, but the time complexity is exponential, as well as the space complexity. We also proved that there cannot be an approximation algorithm for the general TSP.

After this, we presented the case of the Metric Traveling Salesman Problem (MTSP) with the most interesting approximation algorithms for it. We showed two algorithms with approximation ratio 2, *2-Approx-MTSP-Tour* and *Closest-Point Algorithm*, and we also gave *Christofides' Algorithm*, whose approximation ratio equals 3/2. Christofides' Algorithm is the one with the best approximation ratio for the MTSP between the algorithms we know today.

Then, we showed the special case of the Euclidian Traveling Salesman Problem. For this case, we presented the most important *PTAS*, which solves it.

The last case which was examined was the (1, 2)-TSP. Firstly, we presented three similar approximation algorithms for this problem. Their approximation ratios are 4/3, 11/9 and 7/6 respectively. Then, we presented a very recent algorithm (2005), which improves the approximation ratio to 8/7. We also showed that (1, 2)-TSP is MAX-SNP-Hard, so there is no PTAS for it, and it was mentioned that the best known explicit inapproximability bound for this case of the TSP is 741/740.

In the following table (Table 9.1) we summarize all the algorithms presented. The first column (problem) refers to the case of the TSP which is solved by the algorithm. The second column (name) incudes te name of the algorithm which was

problem	name	approximation ratio
general TSP	dynamic algorithm	-
metric TSP	2-Approx-MTSP-Tour	2
metric TSP	Closest-point algorithm	2
metric TSP	Christofides' algorithm	3/2
euclidean TSP	PTAS	$1 + \varepsilon$
(1, 2)-TSP	4/3-approx	4/3
(1, 2)-TSP	11/9-approx	11/9
(1, 2)-TSP	7/6-approx	7/6
(1, 2)-TSP	8/7-approx	8/7

Table 9.1: Summary of the presented algorithms

used in this book. In the third column there is the approximation ratio of the algorithm, if applicable.

In the next and final chapter of this book, we present the most important applications of the Traveling Salesman Problem.

Chapter 10

APPLICATION OF THE TRAVELING SALESMAN PROBLEM

Much of the work on the TSP is not motivated by direct applications, but rather by the fact that the TSP provides an ideal platform for the study of general methods that can be applied to a wide range of discrete optimization problems. This is not to say, however, that the TSP does not find applications in many fields. Indeed, the numerous direct applications of the TSP bring life to the research area and help to direct future work.

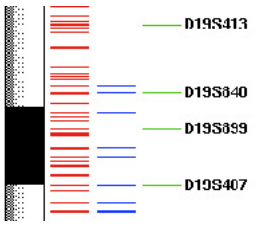
The TSP naturally arises as a subproblem in many transportation and logistics applications, for example the problem of arranging bus routes to pick up the children in a school district. This bus application is of important historical significance to the TSP, since it provided motivation for Merrill Flood, one of the pioneers of TSP research in the 1940s. A second TSP application from the 1940s involved the transportation of farming equipment from one location to another to test soil, leading to mathematical studies in Bengal by P.C. Mahalanobis and in Iowa by R.J. Jessen. More recent applications involve the scheduling of service calls at cable firms, the delivery of meals to homebound persons, the scheduling of stacker cranes in warehouses, the routing of trucks for parcel post pickup, and a host of others.

Although transportation applications are the most natural settings for the TSP, the simplicity of the model has led to many interesting applications in other areas. A classic example is the scheduling of a machine to drill holes in a circuit board or other object. In this case the holes to be drilled are the cities, and the cost of travel is the time it takes to move the drill head from one hole to the next. The technology for drilling varies from one industry to another, but whenever the travel time of the drilling device is a significant portion of the overall manufacturing process then the

TSP can play a role in reducing costs.

To provide the reader a sample of some current applications of the TSP, we provide on the following pages of this chapter a list of some of the applied (and not-so-applied, but still fun) work that has involved modules from the Concorde[1] TSP library.

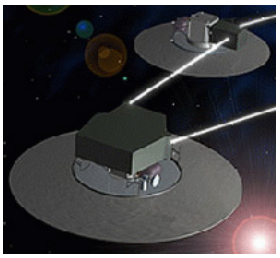
10.1 Genome Sequencing



Researchers at the National Institute of Health have used Concorde's TSP solver to construct radiation hybrid maps as part of their ongoing work in genome sequencing. The TSP provides a way to integrate local maps into a single radiation hybrid map for a genome; the cities are the local maps and the cost of travel is a measure of the likelihood that one local map immediately follows another. A report of the work is given in [1].

This application of the TSP has been adopted by a group in France developing a map of the mouse genome. The mouse work is described in [52].

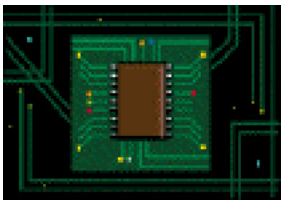
10.2 Starlight Interferometer Program



[1]Concorde is a computer code for the symmetric traveling salesman problem (TSP) and some related network optimization problems. It has been developed in Georgia Institute of Technology and it has been used in many significant applications of the Traveling Salesman Problem. The code is written in the ANSI C programming language.

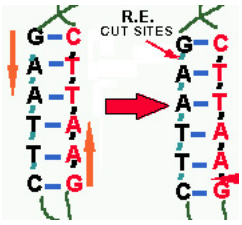
A team of engineers at Hernandez Engineering in Houston and at Brigham Young University have experimented with using Chained Lin-Kerninghan to optimize the sequence of celestial objects to be imaged in a proposed NASA *Starlight* space interferometer program. The goal of the study is to minimize the use of fuel in targeting and imaging maneuvers for the pair of satellites involved in the mission (the cities in the TSP are the celestial objects to be imaged, and the cost of travel is the amount of fuel needed to reposition the two satellites from one image to the next). A report of the work is given in [7].

10.3 Scan Chain Optimization



A semi-conductor manufacturer has used Concorde's implementation of the Chained Lin-Kernighan heuristic in experiments to optimize scan chains in integrated circuits. Scan chains are routes included on a chip for testing purposes and it is useful to minimize their length for both timing and power reasons.

10.4 DNA Universal Strings

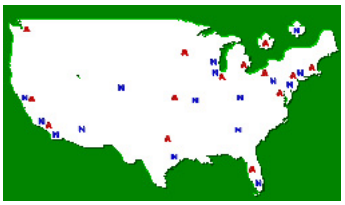


A group at AT&T used Concorde to compute DNA sequences in a genetic engineering research project. In the application, a collection of DNA strings, each of length k, were embedded in one universal string (that is, each of the target strings is contained as a substring in the universal string), with the goal of minimizing the length of the universal string. The cities of the TSP are the target strings, and the cost of travel is k minus the maximum overlap of the corresponding strings.

10.5 Whizzkids '96 Vehicle Routing

A modified version of Concorde was used to solve the Whizzkids'96 vehicle routing problem, demonstrating that the winning solution in the 1996 competition was in fact optimal. The problem consists of finding the best collection of routes for 4 newsboys to deliver papers to their 120 customers. The team of David Applegate, William Cook, Sanjeeb Dash, and Andre Rohe received a 5,000 Gulden prize for their solution in February 2001 from the information technology firm CMG.

10.6 A Tour Through MLB Ballparks



A baseball fan found the optimal route to visit all 30 Major League Baseball parks using Concorde's solver. The map above is a link to an interesting site devoted to current and past ballparks.

10.7 Coin Collection

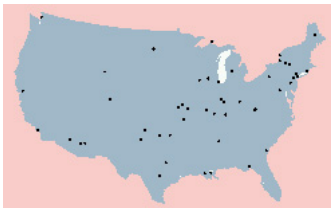
An old application of the TSP is to schedule the collection of coins from payphones throughout a given region. A modified version of Concorde's Chained Lin-Kernighan heuristic was used to solve a variety of coin collection problems. The modifications were needed to handle 1-way streets and other features of city-travel that make the assumption that the cost of travel from x to y is the same as from y to x unrealistic

in this scenario.

10.8 Touring Airports

Concorde is currently being incorporated into the Worldwide Airport Path Finder web site to find shortest routes through selections of airports in the world. The author of the site writes that users of the path-finding tools are equally split between real pilots and those using flight simulators.

10.9 USA Trip



The travel itinerary for an executive of a non-profit organization was computed using Concorde's TSP solver. The trip involved a chartered aircraft to visit cities in the 48 continental states plus Washington, D.C. (Commercial flights were used to visit Alaska and Hawaii.) It would have been nice if the problem was the same as that solved in 1954 by Dantzig, Fulkerson, and Johnson, but different cities were involved in this application (and somewhat different travel costs, since flight distances do not agree with driving distances). The data for the instance was collected by Peter Winker of Lucent Bell Laboratories.

10.10 Designing Sonet Rings

An early version of Concorde's tour finding procedures was used in a tool for designing fiber optical networks at Bell Communications Research (now Telcordia). The TSP aspect of the problem arises in the routing of sonet rings, which provide communications links through a set of sites organized in a ring. The ring structure provides a backup mechanism in case of a link failure, since traffic can be rerouted in the opposite direction on the ring.

10.11 Power Cables

Modules from Concorde were used to locate cables to deliver power to electronic devices associated with fiber optic connections to homes. Some general aspects of this problem area are discussed in [44].

Bibliography

[1] R. Agawala, D.L. Applegate, D. Maglott, G.D. Schuler, and A.A. Schaffler, *A Fast and Scalable Radiation Hybrid Map Construction and Integration Strategy*, Genome Research 10: pp.350-364, 2000.

[2] E.M. Arkin and R. Hassin, *Approximation algorithms for the geometric covering salesman problem*, Discrete Applied Mathematics, 55: pp.197-218, 1994.

[3] S. Arora, *Polynomial Time Approximation Schemes for Euclidean TSP and other Geometric Problems*, Proc. 37th IEEE FOCS (1996).

[4] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and hardness of approximation algorithms*, Journal of the ACM, 45(3): p. 501-555, 1998. Preliminary version in Proc. of FOCS' 92.

[5] S. Arora, *Probabilistic Checking of Proofs and Hardness of Approximation Problems*, Ph.D. Thesis, University of California at Berkeley, 1994.

[6] Y. Azar, *Lower bounds for insertion methods for TSP*, Combinatorics, Probability, and Computing, 3: pp.285-292, 1994.

[7] C.A. Bailey, T.W. McLain, R.W. Beard , *Fuel Saving Strategies for Separated Spacecraft Interferometry* , AIAA Guidance, Navigation, and Control Conference, 2000.

[8] B.S. Baker, *Approximation algorithms for NP-complete problems on planar graphs*, J. ACM, 41: pp.153-180, 1994.

[9] J.L. Bentley, *Fast algorithms for geometric traveling salesman problems*, ORSA J. Computing, 4: pp.387-411, 1992.

[10] P. Berman, S. Hannenhalli and M. Karpinski, *1.375-Approximation Algorithm for Sorting by Reversals*, Proc. 10th ESA (2002), LNCS Vol. 2461, Springer, 2002, pp. 200-210.

[11] P. Berman and M. Karpinski, *8/7-Approximation Algorithm for (1,2)-TSP*, Electronic Colloquium on Computational Complexity, Report No. 69 (2005).

[12] M. Blaeser and B. Seifert, *Computatuional Cycle Coves without Short Cycles*, Proc. 9th ESA (2001), LNCS Vol. 2161, Springer, 2001, pp. 368-379.

[13] Blum A., Jiang T., Li M., Tromp J., and Yannakakis M. (1991), *Linear Approximation of Shortest Superstrings*, Proc. 23th Annual ACM Sympos. on Theory of Computing.

[14] D.P. Bovet and P. Crescenzi, *Introduction to the Theory of Complexity*, Prentice Hall, 1993.

[15] N. Christofides (1976), *Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem*, Technical Report, GSIA, Carnegie Mellon University.

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, "*Introduction to Algorithms*," Second Edition, The MIT Press.

[17] P. Crescenzi, V. Kann, R. Silvestri, and L. Trevisan, *Structure in approximation classes*, In Proceedings of the 1st Combinatorics and Computing Conference, pages 539-548, LNCS 959, Springer-Verlag, 1995.

[18] J. Edmonds, *Optimum branchings*, Journal of Research National Bureau of Standards, Part B, 17B(4):pp.233-240, 1996.

[19] L. Engebretsen, *An explicit lower bund for TSP with distances one and two*, . Technical Report TR98-046, Electronic Colloquium on Computational Complexity, 1998.

[20] L. Engebretsen and M. Karpinski, *Approximation Hardness of TSP with Bounded Metrics*, Proc. 28th ICALP (2001), LNCS Vol. 2076, Springer, 2001, pp. 201-212; journal version to appear in JCSS.

[21] W. Fernandez de la Vega and M. Karpinski, *On the Approximation Hardness of Dense TSP and other Path Problems*, Information Processing Letters 70 (1999), pp. 53-55.

[22] M.R. Garey, R.L. Graham, and D.S. Johnson, *Some NP-complete geometric problems*, In Proceedings of the 8th ACM Symosium on Theory of Computing, pp.10-22, 1976.

[23] M. Grigni, E. Koutsoupias, and C.H. Papadimitriou, *An approximation scheme for planar graph TSP*, In Proceedings of the 36th IEEE Symposium on Foundations of Computer Science, pp.640-645, 1995.

[24] D.B. Hartvigsen (1984), *Extensions of Matching Theory*, Ph.D. Thesis, Carnegie Mellon University.

[25] Dorit S. Hochbaum, *Approximation Algorithms for NP-Hatd Problems*, PWS Publishing Company.

[26] P. Indyk and R. Motwani, *Approximate nearest neighbors: Towards removing the curse of dimensionality*, In Proceedings of the 30th ACM Symposium on Theory of Computing, p.604-613, 1998.

[27] Karp R. M. (1972), Reducibility among Combinatorial Problems in *Complexity of Computer Computations*, R. E. Miller, and J. W. Thatcher (Eds.), Plenum, New York.

[28] Jon M. Kleinberg, *Two algorithms for nearest-neighbor search in high dimensions*, In Proceedings of the 29th ACM Symposium on Theory of Computing, pp.599-608, 1997.

[29] S. Khanna, R. Motwani, M. Sudan, and U. Vazirani, *On syntactic versus computational views of approximability*, SIAM Journal on Computing, 28(1): pp.164-191, 1999.

[30] E. Kushilevitz,R. Ostrovski, and Y. Rabani, *Efficient search for approximate nearest neighbor in high dimensional spaces*, In Proceedings of the 30th ACM Symposium on Theory of Computing, pp.614-623, 1998.

[31] Lawler E. L., Lenstra J. K., Rinnooy Kan, A. H. G., and Shmoys D. B. (1986), *The Traveling Salesman Problem*, Wiley Interscience, New York.

[32] C.S. Mata and J.S.B. Mitchel, *Approximation algorithms for geometric tour and network design problems*, In Proc. 11th ACM Symp. Computational Geometry, pp.360-369, 1995.

[33] N. Megiddo and K.J. Supowit, *On the complexity of some common geometric locatio problems*, SIAM Journal of Computing, 13: pp.182-196, 1984.

[34] J.S.B. Mitchel, *Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, K-MST, and related problems*, SIAM Journal on Computing, 1997.

[35] C.H. Papadimitriou, *Euclidean TSP is NP-complete*, Theoretical Computer Science, 4: pp.237-244, 1977.

[36] C.H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

[37] C.H. Papadimitriou and Steiglitz K. (1982), *Combinatorial Optimization, Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ.

[38] C.H. Papadimitriou C.H. and Yannakakis M. (1988), *Optimization, Approximation, and Complexity Classes*, Proc. 20th Annual ACM Sympos. Theory of Computing, 229-234.

[39] C. H. Papadimitriou and M. Yannakakis, *The Traveling Salesman Problem With Distances One and Two*, Math. Oper. Res. 18 (1993).

[40] F.P. Preparata and M.I. Shamos, *Computational Geometry: An introduction*, Springer-Verlag, 1985.

[41] S.B. Rao and W.D. Smith, *Approximating geometrical graphs via "spanners" and "banyans"*, In Proceedings of the 29th ACM Symposium on Theory of Computing, pp.540-550, 1998.

[42] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, *An analysis of several heuristics for the travelng salesman problem*, SIAM Journal of Computing, 6: pp.563-581, 1977.

[43] S. Sahni and Gonzalez T. (1976), *P-complete Approximation Problems*, J. Assoc. Comput. Math., 23, 555-565.

[44] T.H. Sloane, F. Mann, H. Kaveh, *Powering the Last Mile: An Alternative to Powering FITL*, Alpha Technologies. Bellingham, WA, USA .

[45] W.D. Smith, *Finding the optimum N-city traveling salesman tour in the Euclidean plane in subexponential time and polynomial space*, Manuscript, 1988.

[46] L. Trevisan, *When Hamming Meets Euclid: The Approximability of Geometric TSP and MST*, Proc. 29th ACM STOC (1997), pp. 21-29.

[47] J.H. van Lint and R.M. Wilson, *A Course in Combinatorics*, Cambridge University Press, 1992.

[48] Vijay V. Vazirani, "*Approximation Algorithms*," Springer Verlag.

[49] S. Vishwanathan, *An Approximation Algorithm for the Asymetric Travelling Salesman Problem with Distances One and Two*, Information Processing Letters 44 (1992), pp. 297-302.

[50] A.R. Warburton, *Worst-case analysis of some convex hull heuristics for the Euclidean traveling salesman problem*, Operations Research Lett., 13: pp.37-42, 1993. 297-302.

[51] H.T. Wareham, *On the computational complexity of inferring evolutionary trees*, Master's Thesis, Memorial University of Newfoundland, Canada, 1993.

[52] *A Radiation Hybrid Transcript May of the Mouse Genome*, Nature Genetics 29, pp.194-200, 2001.