MIT/LCS/TM-209

COMPUTATIONAL COMPLEXITY AND THE TRAVELING SALESMAN PROBLEM

David Johnson

Christos Papadimitriou

December 1981

COMPUTATIONAL COMPLEXITY AND THE TRAVELING SALESMAN PROBLEM

David S. Johnson and Christos H. Papadimitriou

This is a draft of the Third Chapter in the forthcoming book *The Traveling Salesman Problem*, edited by E.L. Lawler, J.K. Lenstra, and A.G. Rinooy Kan, to be published by John Wiley and Sons.

Keywords and Phrases: Traveling salesman problem (TSP), computational complexity, polynomial-time algorithms, P, NP, and NP-complete problems, Hamilton circuit problem, traveling salesman polytope.

COMPUTATIONAL COMPLEXITY AND THE TRAVELING SALESMAN PROBLEM

David S. Johnson and Christos H. Papadimitriou

In the last decade or so a theory of *computational complexity* has developed, based on rigorous methods for evaluating algorithms and for classifying problems as "hard" or "easy". This theory is deeply indebted to the field of Combinatorial Optimization, which has provided it with invaluable motivation, insight, and paradigms. The TSP is probably the most important among the latter. It has served as a testbed for about every new algorithmic idea, and was one of the first optimization problems conjectured to be "hard" in a specific technical sense.*

In this chapter we shall be studying the complexity of the TSP, while providing an introduction to the general area of computational complexity. Section 1 provides an introduction to the notion of an algorithm and to different measures of its "complexity". Section 2 shows how we can restrict our attention to *decision problems* (problems with a "yes" or "no" answer), and divide these into equivalence classes according to their difficulty. Section 3 then proves that even very restricted versions of the TSP are likely to be very hard (they are, in the technical jargon, *NP-complete*). Section 4 concludes the chapter by examining the complexity of problems related to the TSP, as well as complexity aspects of various proposed *approaches* to the TSP.

*The conjecture was made in 1965 by Jack Edmonds, whose work of this period contains many seeds of present-day complexity theory

1. Algorithms and Complexity

1.1 Algorithms

An algorithm is a step-by-step procedure for solving a problem. This concept can be formalized in a number of ways, for instance in terms of *Turing machines* or of *computer programs* in some programming language such as FORTRAN or ALGOL (assuming the semantics of the language have been precisely specified). However, the above informal notion of "algorithm" will suffice for most of our discussions, as will the following informal notion of "problem" (the thing an algorithm "solves").

A problem can be viewed as consisting of a *domain* containing the *instances* of the problem, together with a *question* that can be asked about any one of the instances. For example, an instance of the TSP is composed of a number n of cities and an $n \times n$ distance matrix C, and the question asked by the TSP is "What is the shortest tour for the n cities?" Our standard format for describibg problems will consist of a generic description of an instance in the problem's domain, followed by a statement of the problem's question, posed in terms of the generic instance. A description of the TSP in this format goes as follows:

TSP

INSTANCE: Integer $n \geq 3$ and $n \times n$ matrix $C = (c_{ij})$, where each c_{ij} is a non-negative integer.

QUESTION: Which permutation $\pi = (\pi(1), \pi(2), \ldots \pi(n))$ of the integers from 1 to n minimizes the sum $\sum_{i=1}^{n-1} c_{\pi(i),\pi(i+1)} + c_{\pi(n),\pi(1)}$?

Note that in this definition we restrict the inter-city distances to be integers, a convention we shall follow throughout this chapter. In practice, one might expect inter-city distances to occasionally have fractional parts, or even be irrational numbers. However, irrational numbers are approximated by rationals for computer representation, and one can always convert the rational distances to integral values simply by multiplying by an appropriate scale factor. We shall assume that all this has already been done.

An algorithm is said to *solve* a problem P if, given any instance I of P as input data, it will generate the answer of P's question for I. An algorithm for the TSP must thus, given any n and C as above, generate the corresponding minimum length tour (permutation).

One classical result in the theory of algorithms is that there are well-defined mathematical problems for which *no* algorithms can exist (for a discussion see [Lewis and Papadimitriou 1981]). Although, given a particular instance, one may be able to come up with an answer for that instance, there is no general procedure that will apply to *all* instances. By comparison, the situation with the TSP is much better (at least in theory). There definitely is an algorithm for the TSP, namely, Algorithm A of Figure 1. It generates

all tours, evaluates each, and picks the best. This algorithm embodies the sophisticated technique known to practitioners as "brute force", and even though we shall postpone a detailed analysis of its efficiency, it should already be clear that this is not its strong point.

A better algorithm is Algorithm B in Figure 2. This algorithm, finds, for each i, the shortest path from city 1 to city i that visits all the other nodes in $\{2, 3, \ldots, n\}$. Once these paths are found it is a simple matter to compute the shortest tour (this is what the last two assignments of Algorithm are doing). To find these paths, the algorithm solves a more general problem. For any $S \subseteq \{2, \ldots, n\}$ and $i \in S$, let an (S, i) path be a path which starts at city 1,visits each city in S exactly once, and no other city, and ends up in city i. Let cost$[S, i]$ stand for the length of the shortest (S, i) path. Then cost$[S, i]$ satisfies the following equation, where $|S| \geq 2$:

$$\text{cost}[S, i] = \min_{k \in S - \{i\}} (\text{cost}[S - \{i\}, k] + c_{ki})$$

What Algorithm B does should now be clear. It "builds-up" the values cost$[S, i]$ for larger and larger sets S until cost$[\{2, 3, \ldots, n\}, i]$ is obtained for $i = 2, 3, \ldots n$. The array *bestpath* is used to record for each S and i the path which achieves cost$[S, i]$.

Putting together the solutions of bigger and bigger sub-problems to obtain an overall solution is an algorithmic technique of wide applicability, known as *dynamic programming* (see [Bellman and Dreyfus 1962]). This particular algorithm is due to [Held and Karp 1962].

For another example of this use of dynamic programming, let us consider the following problem that would seem to be closely related to the TSP.

SHORTEST PATH
INSTANCE: As in TSP
QUESTION: What is the shortest path from city 1 to city n?

Note that, unlike the TSP, the shortest path problem makes no insistence that cities other than 1 and n be visited by the path. Here we define cost$[i, j]$, for each city $i \neq 1$ and integer $j \geq 1$, to be the length of the shortest path from 1 to i containing j or fewer edges. We have, for $j \geq 1$

$$\text{cost}[i, j] = \min \{ \text{cost}[i, j-1], \min_{k \neq i, 1} (\text{cost}[k, j-1] + c_{ki}) \}$$

This equation is the key to the dynamic programming embodied in Algorithm C, shown in Figure 3.

Superficially, Algorithms B and C look very similar. As we shall see, from the standpoint of computational complexity they are worlds apart. Algorithm B, and in fact every known algorithm for the TSP, has much more in common with the brute force of Algorithm A than it does with the dynamic programming of Algorithm C.

1.2 Estimating Running Time

Let us estimate the running time of Algorithms A, B, and C. In order that our analysis provide general information, we shall express running time as a function of the number n of cities, rather than just determine it for a particular instance. In order to make our analysis independent of the speed of the particular computer which might be used to execute the algorithm, we shall count "steps" instead of machine cycles, and shall use O-notation ("big O" notation) to express the running time function. To say that an algorithm's running time is "$O(f(n))$" means that there is some constant c such that the algorithm's running time on all inputs of size n is bounded by $cf(n)$. The precise value of c would depend on the computer used.

Algorithm A takes time $O(n!)$. The main loop is repeated $(n-1)!$ times, once for each permutation of $\{2,\ldots,n\}$. Each execution of the loop takes $O(n)$ time in generating the next permutation, evaluating *cost*, and updating *besttour* (if necessary). Multiplying, we obtain the $O(n!)$ estimate.

Much of the effort of Algorithm B is concentrated in the triply nested loop. Let us assume that j is fixed. Then the innermost loop takes time at most cj for some $c > 0$ (the minimization is over $j-1$ items, and the sequences *bestpath* are about j long). The loop is repeated for each subset of $\{2,3,\ldots,n\}$ that contains j elements, and for each of the j elements in each subset. Summing over all j's we have

$$\sum_{j=2}^{n-1} \binom{n-1}{j} cj^2 \leq c[2^{n-1}(n-1)^2] = O(n^2 2^n)$$

The rest of Algorithm B (the initial loop and the computation of *besttour*) are of linear complexity, and their contribution is absorbed in the O-notation. Thus, Algorithm B takes time $O(n^2 2^n)$.

For Algorithm C, the body of the inner loop is executed $(n-1)^2$ times, and takes $O(n)$ times per execution. This yields are overall bound of $O(n^3)$

1.3 Worst-case and Average Case Analysis

Algorithms A, B and C are all very well-behaved, in the following sense: One can predict quite accurately the number of steps required for an instance, and the prediction does not vary from one instance to another with the same size n. On the other hand, there are many algorithms that can take a number of steps which varies wildly among instances of the same size (a famous example is the Simplex algorithm for linear programming [Klee and Minty 1972]). If we are to have a complexity function which takes on a single value for each size, there are essentially two approaches we can take. We can either count the *average* or the *maximum* number of steps, over all inputs of size n.

The second method of analysis, called *worst-case analysis*, has an obvious disadvantage: "Pathological" cases are allowed to determine the complexity, even though they may be exceedingly rare in practice. *Average-case analysis* on the other hand also has its drawbacks. We must choose a probability distribution for the instances, and finding a mathematically tractable one that also models the instances encountered in practice is not always easy (or even possible). Most clever algorithms introduce a great deal of *statistical dependence* among their individual stages, and average case analysis becomes a very complicated task when independence cannot be assumed. Finally, whereas worst-case analysis at least provides a "guarantee" to the user of an algorithm, average-case analysis often provides a prediction that one can have confidence in only if one has to solve a large number of instances. If you must solve a particular instance of a problem, and your algorithm performs abysmally on it, there is little consolation in knowing that you have just encountered a statiscally insignificant exception.

Thus we shall take the worst-case approach in the rest of this chapter, a choice that rewards us with the elegant theory of NP-completeness. The average case approach wil be pursued in Chapter 6.

1.4 Polynomial Time

We have chosen a measure of the performance of an algorithm: the number of steps spent on an instance, maximized over all instances of size n, and expressed as a function of n (up to a constant multiple). What we have yet to do is specify a relationship between this measure and the question of whether the given algorithm is a "good" one or not.

Ideally, we would like to call an algorithm "good" when it is sufficiently efficient to be useable in practice, but this is a rather vague and slippery notion. An idea that has gained wide acceptance in recent years is to consider an algorithm "good" if its worst-case complexity is bounded by a polynomial function of n. For example, algorithm C is a "polynomial-time algorithm" and hence "good", but algorithm A and B are not. They are both "exponential" in the sense that their worst-case complexity grows at least as fast as c^n for some $c > 1$. (Although there are "sub-exponential" functions such as $n^{\log n}$, which are neither polynomially bounded nor exponential, for simplicity in what follows we shall often use the term "exponential" to refer to any algorithm whose worst-case complexity is not bounded by a polynomial).

The difference between polynomials and exponentials becomes clear if one takes an "asymptotic" point of view. Although a given exponential function may initially yield smaller values than a given polynomial function, there always is an N such that for all $n \geq N$, the polynomial is the winner. See Table 1. Furthurmore, polynomial time algorithms are in a better position to take advantage of technological improvements in the speed of

computers. Suppose that we have two algorithms, one running in time $O(n^3)$, the other in time $O(2^n)$, both of which can solve a problem with $n = 100$ in one day. If we get a new computer which is twice as fast, the polynomial-time algorithm can now in one day solve instances with $n = 126$, a *multiplicative* factor of 1.26, whereas our exponential algorithm can only obtain an *additive* improvement, to $n = 101$.

There are also a number of mathematical niceties that reward us for concentrating on the distinction between polynomial and exponential. Polynomials have mathematical properties that make them appealing as the basis for a theory. You can add two polynomials, multiply them, and compose them, and the result will still be a polynomial. Polynomial *time* has a long standing place in the abstract theory of computation: There are a wide variety of "reasonable" theoretical models for computers, from Turing Machines to Random Access Machines, with many varieties of each, but each model can simulate any other one with only a *polynomial* loss in efficiency. Thus a polynomial-time algorithm for one model corresponds to a polynomial-time algorithm in each of the others. (For details, see [Aho, Hopcroft, and Ullman 1974, Hopcroft and Ullman 1979]). Furthermore, if one concentrates on polynomial time, one can ignore differences in the choice of input size parameters, an issue that is worth further discussion.

Our definition of *input size* can have a significant effect on the function we derive for the worst-case complexity of an algorithm. For instance, consider an instance of the TSP with n cities. We have been using the number of cities as the variable in our complexity measures. A reasonable alternative would be the number $m = n^2$ of entries in the distance matrix. If we use *this* as our measure of input size, algorithm A suddenly "speeds-up" from $O(n!)$ to $O(\sqrt{m}!)$, and algorithm C "speeds up" from $O(n^3)$ to $O(m^{3/2})$. Notice, however, that algorithm A remains exponential and algorithm C remains polynomial. Thus, if we are only interested in the distinction between polynomial and exponential algorithms, we are given a wide latitude in our choice of input measures and hence can ignore fine details.

There is, however, one proviso: Our measure of input size must reflect (to within a polynomial) the actual length of a concise encoding of the instance in question, as it would be input to a computer (as a sequence of symbols). Assuming thal all the inter-city distances are integers that fit into a single computer word, say of size 32, the number of cities is a measure of input size which meets this requirement, since the actual encoding length would be $32n^2$. If the inter-city distances are allowed to be arbitrarily large integers, we would need to use a more sophisticated measure of input size, for example $m = n^2 \cdot log_2(\max\{c_{ij}\})$, that would take into account the number of bits required to represent those distances (the requirement that we consider only concise encodings means that numbers must be represented in a positional notation such as binary, so that the size

of a number is proportional to its logarithm, rather than its value). For a more detailed discussion of input size and encodings, see [Garey and Johnson 1979].

Thus there are both practical and theoretical reasons for identifying the distinction between "good" and "bad" algorithms with the distinction between those algorithms which run on polynomial time and those that do not. Proposals to this effect were first made, in the mid-1960's, by Alan Cobham [Cobham 1964] and Jack Edmonds [Edmonds 1965], and the significance of polynomial time was hinted at much earlier in the work of [John von Neumann 1953]. It was Edmonds who proposed the precise terminology "good" = "polynomial-time". Subsequent work and in particular the theory of NP-completeness, which we shall be discussing shortly, have widened the acceptance of this equation as the best available way of tying the empirical notion of a "practical algorithm" to a precisely formalized mathematical concept.

Of course, as with all mathematical modeling of real-world phenomena, there is room for "experimental errors". The Simplex algorithm for linear programming, although it has exponential worst case complexity [Klee and Minty 1972], has been a constant success in practice. The recently discovered Ellipsoid algorithm for linear programming [Khachian 1979] on the other hand, is provably polynomial, but for now it seems to have been abandoned as a practical alternative, despite much initial enthusiasm.

There is also, of course, the possibility of such contrasts as an $O(1.001^n)$ exponential algorithm with a polynomial-time algorithm that is $O(n^{80})$ or $O(10^{100}n)$. However, for the most part such contrasts are more potential than real. Algorithms with such running times do not appear that often in practice. Moreover, once a polynomial algorithm is discovered for a problem, it is often the case that a sequence of improvements produces a version that is also of practical use. The real breakthrough is in finding that *first* polynomial-time algorithm. In a sense, finding a better exponential-time algorithm is just a matter of more cleverly organizing a brute force approach, whereas to make the jump to polynomial time requires a real insight into the nature of the problem at hand, *if* the jump can be made at all.

2. *P* and *NP*

2.1 Decision Problems

Just as we can classify an algorithm as good or bad, depending on whether or not it has polynomial time complexity, we can classify a *problem* as "hard" or "easy" depending on whether or not it can be solved by an algorithm with polynomial time complexity. It is on the basis of this distinction that an elegant theory of the complexity of problems has been developed.

For uniformity, this theory formally restricts itself to *decision problems*, that is, problems whose question requires only a "yes" ot "no" answer. (As we shall see, this does not restrict the applicability of the theory.) Many problems are naturally expressed as decision problems. An example, about which we will have much to say later, is the following well-known problem from graph theory.

HAMILTON CIRCUIT

INSTANCE: A graph $G = (V,E)$

QUESTION: Is there a circuit (closed sequence of edges) in G passing through each vertex in V exactly once?

The TSP is not a decision problem, as its question asks for the shortest tour, not a "yes" or "no" answer. However, the TSP (and *any* optimization problem) can be easily reformulated as a decision problem, by adding a *bound* B to the input data. For example, the decision version of the TSP would be as follows:

TSP(decision)

INSTANCE: Integer $n \geq 3$, an $n \times n$ matrix $C = (c_{ij})$, where each c_{ij} is a non-negative integer, and an integer $B \geq 0$.

QUESTION: Is there a permutation $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$ of the integers from 1 to n such that

$$\sum_{i=1}^{n-1} c_{\pi(i),\pi(i+1)} + c_{\pi(n),\pi(1)} \leq B?$$

It is clear that if there is a polynomial-time algorithm for the TSP, then there is a polynomial time algorithm for its decision problem version. This is true for all optimization problems, as long as the cost of a given feasible solution can be computed in polynomial time (a quite reasonable assumption). What is a bit more remarkable is that, for the TSP, the converse is also true.

Consider the algorithm TSPTOUR depicted in Figure 4. TSPTOUR generates an optimal tour for any TSP instance, so long as it is provided with a subroutine TSPDECISION that solves TSP(decision).

The first part of TSPTOUR computes the length of an optimal tour, using "binary search". We start out knowing that the optimal length lies between $low = 0$ and $high = n \cdot c_{max}$, where $c_{max} = max_{ij}\{c_{ij}\}$. This is because a tour contains n links, none of which is longer than the maximum inter-city distance. We use TSPDECISION to determine in which half of this range the optimum length actually resides, by applying it to the original instance with the bound $B = \lfloor \frac{low+high}{2} \rfloor$. If the answer is "yes" we know that B is a new upper bound on the optimal length; otherwise $B + 1$ is a new lower bound. In either case, the size of the range has been halved. The process is repeated until the upper and lower bounds coincide, in which case they both equal the optimum length. The total number of iterations is no more than $\lceil log_2(n \cdot c_{max}) \rceil$, since the size of the range is halved at each iteration.

At this point we know the optimal tour length but do not yet have an optimal tour. The second part of TSPTOUR constructs such a tour. It does this by checking which inter-city links may be deleted, again using TSPDECISION. If a tour of the optimum length still exists when c_{ij} is increased to $nc_{max} + 1$, we know that there is an optimum tour not containing a link between cities i and j. Otherwise, an optimal tour *must* contain a link between i and j, so we return c_{ij} to its original value. After we have tested all the c_{ij}, the only ones that will retain their original values will be a set corresponding to the links on an optimal tour. (Since no optimal tour can contain one of the $(n \cdot c_{max} + 1)$-length links, there must be an optimal tour using only links which had their original length restored, and any unnecessary link would *not* have had its length restored).

The running time of TSPTOUR depends on that of TSPDECISION. This subroutine is called $\lceil log_2(n \cdot c_{max}) \rceil + n^2$ times, which is a polynomial function of $n \cdot log_2(c_{max})$ the "accurate" input size measure for the TSP described in the previous section. Thus if TSPDECISION runs in polynomial time, TSPTOUR will have a time complexity function which is the product of two polynomials and hence polynomial itself. We thus have the following result:

Theorem 1 There is a polynomial time algorithm for the TSP if and only if there is a polynomial time algorithm for TSP(decision).

Analogous results can be proved for most optimization problems. Thus, if we are only interested in the polynomial vs. non-polynomial distinction, we lose little in the way of generality by restricting attention to decision problems.

2.2 Polynomial Reductions

We now introduce a method for relating the complexities of different decision problems. It is every mathematician's dream to reduce the problem he is working on to a problem

that has already been solved. In computational complexity there is an analogous notion: the *polynomial reduction.*

We say a problem A is *polynomial-time reducible* to a problem B if there is an algorithm for A which uses a subroutine for B, and the algorithm for A runs in polynomial time when the time for executing each call of the subroutine is counted as a single step (note that this implies that the subroutine can only be called a polynomially bounded number of times). For example, the algorithm TSPTOUR of Fig. 4 is a polynomial reduction from the TSP to TSP(decision). Exercise 2 asks for a reduction of the TSP to the problem of finding the longest path between two vertices in a graph.

The usefulness of this notion of reduction stems from the following lemma, which can be proved easily using the fact that the composition of two polynomials is a polynomial.

Lemma 1 If there is a polynomial-time reduction from A to B, and there is a polynomial-time algorithm for B, then there is a polynomial-time algorithm for A.

There are basically three kinds of reductions:

(1) *Reductions that prove a problem is easy*, by reducing it to a problem already known to be solvable by a polynomial-time algorithm. This is the traditional use of reduction; for example [Ford and Fulkerson 1959] describe many such reductions.

(2) *Reductions that prove a problem is hard*, by showing that some known "hard" problem reduces to it. Some early examples are in [Dantzig et al. 1967] Section 3 will contain a series of such reductions.

(3) *Reductions that prove nothing*, because they reduce problem A of unknown complexity to a problem B that is (or is suspected to be) hard.

A great many reductions of this third kind have appeared in the last three decades. The problem that played the role of problem B most frequently is the following problem, for which all known algorithms have exponential-time complexity (and work poorly in practice too):

INTEGER PROGRAMMING

INPUT: An $m \times n$ integer matrix $A = a_{ij}$, an m-vector $b = (b_1, b_2, \ldots, b_m)$ of integers.

QUESTION: Is there an n-vector x of non-negative integers such that $Ax = b$, i.e. $\sum_{j=1}^{n} a_{ij}x_j = b_i, 1 \leq i \leq m$

Versions of this problem with inequalities instead of equalities, or which ask for an x which minimizes a linear function, $\sum_{j=1}^{n} c_i x_j$ can all be shown to be equivalent to this decision problem version [Papadimitriou and Steiglitz 1982]. However, quite different-looking problems can also be reduced to it.

As an example, let us show how to reduce HAMILTON CIRCUIT to INTEGER PROGRAMMING. Given a graph $G = (V,E)$ where $V = \{1, 2, \ldots, N\}$, we can write an integer program $I(G)$ such that $I(G)$ has a solution if and only if G has a Hamilton circuit. $I(G)$ has N^2 variables x_{ik} $i,k = 1, \ldots, N$. The intended meaning of x_{ik} is the following: $x_{ik} = 1$ if vertex i is the kth vertex of the Hamilton circuit, and $x_{ik} = 0$ if otherwise. This can be expressed by the following equations and inequalities:

(1) $\sum_{k=1}^{N} x_{ik} = 1 \quad i = 1, \ldots, N$
(2) $\sum_{i=1}^{N} x_{ik} = 1 \quad k = 1, \ldots, N$
(3) $x_{ik} + x_{j,k+1} \leq 1 + e_{ij} \quad i,j,k = 1, \ldots, N$

where $e_{ij} = 1$ if $\{i,j\} \in E$ and $e_{ij} = 0$ otherwise; and $x_{i,N+1}$ stands for $x_{i,1}$, for all i. The inequalities of (3) can be turned into equalities by using *slack variables* $s_{ijk} \geq 0$ and replacing (3) by $x_{ik} + x_{j,k+1} + s_{ijk} = 1 + e_{ij}$, $1, j, k = 1, \ldots, N$. The requirement that the x_{ik}'s have only 0 or 1 as potential values is automatically ensured by equationa (1) and (2). Writing all these equations in matrix form we get an instance $I(G)$ of INTEGER PROGRAMMING in which $m = N^3 + 2N$ and $n = N^3 + N^2$. It is easy to verify that the desired solution x exists for $I(G)$ if and only if G has a Hamilton circuit.

The polynomial reduction from HAMILTON CIRCUIT to INTEGER PROGRAMMING is now straightforward. Given G we construct $I(G)$, which clearly can be done in polynomial time, and then call the hypothetical algorithm for INTEGER PROGRAMMING, using the "yes" or "no" outcome of that algorithm as our answer. There is a special name for polynomial reductions of this simple type, which are between decision problems and use the hypothetical subroutine exactly once, returning its answer as their own. They are called *polynomial transformations.* We say that HAMILTON CIRCUIT is *polynomially transformable* to INTEGER PROGRAMMING (notation: HAMILTON CIRCUIT α INTEGER PROGRAMMING).

The integer program $I(G)$, which was the result of the transformation above, belongs to the important special class of *0–1 programs*, that is, integer programs with the additional constraint $x_j \in \{0, 1\}$ for all variables x_j. 0–1 PROGRAMMING is a special case of INTEGER PROGRAMMING, since the constraint $\dashv x_j \in \{0, 1\} \vdash$ can be expressed by introducing a new variable x'_j and the equation $x_j + x'_j = 1$. Thus 0–1 PROGRAMMING can be no "harder" than the general problem. We show below that it is no "easier" either, by giving a polynomial transformation from INTEGER PROGRAMMING to it. The proof requires a relatively recent result, which we state without proof (see [Borosh & Treybig, 1973] or [Papadimitriou 1981]).

Lemma 2 Suppose A and b make up an instance of INTEGER PROGRAMMING, and let s be the sum of the logarithms of the absolute values of the entries in A and b, plus

$m + n$. Then the instance (A, b) has a solution if and only if it has a solution satisfying $x_j < 2^{p(s)}$, for all variables x_j, where p is a fixed polynomial independent of A and b.

Note that s, as defined in the lemma, constitutes a valid measure of input size, as discussed in Subsection 1.4. Thus, the lemma in effect says that, if there is a solution, then there is one whose size is bounded by a polynomial in the instance size (each x_j requires only $O(p(s))$ symbols).

Theorem 2 INTEGER PROGRAMMING α 0–1 PROGRAMMING

Proof Given an integer program I (an instance I of INTEGER PROGRAMMING), we transform I to an equivalent 0–1 program $Z(I)$ as follows: for each variable x of I we create $p(s)$ 0–1 variables $x_0, x_1, \ldots, x_{p(s)-1}$ for $Z(I)$, where s and p are as in Lemma 2, with the interpretation that $x = \sum_{j=0}^{p(s)-1} x_j 2^j$. We then rewrite the equations of I using this substitution, and obtain $Z(I)$. That $Z(I)$ is equivalent to I (i.e., has a solution if and only if I does) follows from Lemma 2 and the fact that a non-negative integer less than $2^{p(s)}$ can be written (in a unique way) as the sum of powers of 2 from $2^0 = 1$ to $2^{p(s)-1}$. That this is a polynomial transformation follows from the fact that s, as defined in Lemma 2, is a valid measure of input size for INTEGER PROGRAMMING.

In Section 3 we shall show that INTEGER PROGRAMMING is polynomial-time transformable to even more restricted special cases of itself. In doing so we shall use the following particularly desirable property of polynomial transformations, again proved using the fact that the composition of two polynomials is a polynomial.

Lemma 3 If there are polynomial transformations T_1 from A to B and T_2 from B to C, then there is a polynomial transformation T_3 from A to C.

In other words, polynomial transformability is transitive. (A similar result holds for polynomial reducibility.)

2.3 The classes P and NP

We now define the first of two important classes of decision problems. The class P consists of all those decision problems for which a polynomial-time algorthm exists. In view of Theorem 1 and the discussion in Section 1.4, the crucial question involved in determining the complexity of the TSP can be formalized as follows: "Is TSP(decision) in P?"

In this Chapter we present evidence that strongly suggests the answer to this question is no, but we shall not be able to provide a rigorous proof. Proofs that problems are not in P do exist. If a problem is undecidable and hence not solvable by *any* algorithm, it is certainly not in P. In addition, many decidable problems can be shown to require exponential time (see, for example [Lewis and Papadimitriou 1981]). Unfortunatly, there

is a sense in which TSP(decision) is qualitatively "easier" than any of the problems so far shown to be outside of P, and so new techniques would seem to be required if it is to be proved inherently exponential.

The characteristics of TSP(decision) which make it "easier" than the provably hard problems, are shared with a wide variety of other natural combinatorial optimization problems. Over the past two decades three different but equivalent formulations have been proposed for these characteristics. Together they define a class of decision problems we call NP.

The first definition involves the *succinct certificate property*. HAMILTON CIRCUIT, 0–1 PROGRAMMING, and TSP(decision) all have this property. Any "yes" instance (i.e., a graph that has a Hamilton circuit; a system of linear equations that has a 0–1 solution; a distance matrix whose optimal tour length is less than a given bound B) has a succinct "certificate": a mathematical object of small size that establishes beyond doubt that the instance indeed warrants a "yes" answer. (For the above problems the certificates are: a Hamilton circuit; a 0–1 vector satisfying the equations; and a tour of length B or less, respectively). The certificate must be *succinct*, i.e., of size bounded by a polynomial in the instance size, and there must be a polynomial-time *certificate checking algorithm* which, given an instance and a supposed certificate, determines whether the certificate is indeed valid. All "yes" instances must possess at least one such certificate, and no "no" instance can have any. It may be very difficult to discover a certificate for a "yes" instance, but, once discovered, it can be exhibited and checked in an efficient manner.

This can be made a bit more formal as follows. A decision problem A has the succinct certificate property if and only if there is another decision problem $C \in P$ (the certificate-checking problem for A) whose instances are of the form (I, S), where I is an instance of A and S is an object whose size is bounded by a polynomial in the size of I, and such that the following are equivalent:

(1) I is a "yes" instance of A

(2) There is an S such that (I, S) is a "yes" instance of C.

There are clearly many interesting and important decision problems, in addition to the ones already mentioned, that have the succinct certificate property: By Lemma 2, INTEGER PROGRAMMING, though more general than 0–1 PROGRAMMING, has the succinct certificate property itself. Furthermore, it can be argued that the decision problem versions of all "reasonable" combinatorial optimization problems have the property. Such problems ask about the existence of certain combinatorial objects (vectors, circuits, tours, etc.) whose costs, lengths, etc. obey a certain bound. Such objects are "succinct" according to our definition, and their costs can be computed efficiently (if the cost function is "reasonable"). Hence the objects themselves can play the role of the

certificates for these problems. The class of problems that satisfies the succinct certificate property was introduced by Edmonds [Edmonds 1965], who called them problems with *good characterizations.*

A second way of defining a broad class of decision problems including TSP(decision) involves the concept of a *nondeterministic algorithm*, an unrealistic but theoretically useful tool, with a long history in complexity theory. A nondeterministic algorithm is like an ordinary algorithm, except that it is equipped with one additional, extraodinarily powerful instruction:

go to both label1, label2

Executing this instruction divides the computation into two parallel processes, one continuing from each of the two instructions indicated by *label1* and *label2*. By encountering more and more such instructions, the computation will branch like a tree into a number of parallel computations that potentially can grow as an exponential function of the time elasped. (See Fig. 5). If *any* of these branches reports "yes" then we say that the overall nondeterministic algorithm has answered "yes". The answer is "no" if none of the branches ever reports "yes". The asymmetry parallels that of the succinct certificate property: "Yes" instances have a succinct proof of their "yes-ness", but "no" instances may not have a succinct proof of their "no-ness"—how can one succinctly certify that a graph has *no* Hamilton circuit?

We say a nondeterministic algorithm solves a decision problem in polynomial time if (a) for each instance it gives the correct answer, as defined above, and (b) the number of steps used by the first of the branches to report "yes" (counting steps from the start of the overall computation) is bounded by a polynomial in the size of the instance. For example, the non-deterministic algorithm N of Figure 6 solves 0–1 PROGRAMMING in polynomial time by examining, in parallel, all 0–1 vectors.

The class of decision problems that can be solved in polynomial time by nondeterministic algorithms was introduced by S.A. Cook [Cook 1971] and by R.M. Karp [Karp 1972]. The latter called the class *NP*, for Non-deterministic Polynomial time.

A third class of problems was studied by G.B. Dantzig in 1960 [Dantzig 1960]. He wrote "It is worthwhile to systematically review and classify problems that can be reduced to [INTEGER PROGRAMMING] and thereby solved."

Of course, the last statement would now be considered overly optimistic, given the poor performance of the best available algorithms for INTEGER PROGRAMMING. In 1960 the euphoria over the discovery of the Simplex algorithm for linear programming, and the new but untested cutting plane approach for INTEGER PROGRAMMING, were fueling such optimism. Still, decision problems that can be polynomially transformed to INTEGER PROGRAMMING constitute a broad, important class of problems. As

Dantzig and numerous researchers after him have showed, this class contains many graph problems, some non-linear programming problems, and the decision problem versions of the TSP and a host of other combinatorial problems.

The following remarkable theorem, a paraphrase of results due to Cook, states that the three classes we have just defined are *one and the same class.*

Theorem 3 [Cook 1971] Let A be a decision problem. Then the following are equivalent.

(1) A has the succinct certificate property.

(2) $A \in NP$

(3) A is polynomial-time transformable to INTEGER PROGRAMMING.

Sketch of Proof We shall not give a complete proof of this theorem (for a rigorous proof, see [Cook 1971] or any one of [Aho, Hopcroft, and Ullman, 1974], [Garey and Johnson 1979], [Lewis and Papadimitriou 1981], [Papadimitriou and Steiglitz 1981]). Two of the three implications are easy, however.

(1) $\Rightarrow$ (2) If A has the succinct certificate property, then a non-deterministic algorithm for A can be designed as follows. First, branch out the computation so that all potential certificates can be examined in parallel. Since the certificate is succinct, this can be accomplished while the computation tree still has only polynomial depth. Then, for each of the potential certificates (still in parallel) deterministically check its validity using the polynomial-time certificate-checking algorithm for the problem. If a succinct certificate exists, at least one of the branches of the computation will return a "yes" answer in polynomial time. (This is exactly what the nondeterministic algorithm N of Fig. 6 is doing).

(3) $\Rightarrow$ (1) If A is polynomially transformable to INTEGER PROGRAMMING, then every "yes" instance of A has a succinct certificate: the solution, as specified in Lemma 2, to the integer program that is the result of the transformation. By Lemma 2 this solution will be succinct, and the certificate-checking algorithm merely needs to construct the integer program corresponding to the given instance of A (using the polynomial transformation) and then check to see whether the proposed certificate is indeed a solution.

(2) $\Rightarrow$ (3) This is the hard part. We need first to define more precisely our model of non-deterministic algorithms. This can be done in a number of ways, for instance, in terms of *non-deterministic Turing machines* (see [Aho, Hopcroft and Ullman 1974], [Garey and Johnson 1979], [Lewis and Papadimitriou 1981]). Our model will essentially be a set of *rules* whereby we can move from one "state" of the computation to the next (possibly in more than one way because of non-determinism). What the proof would then show is that INTEGER PROGRAMMING is expressive enough to enable us to represent these rules by linear equations in appropriate 0–1 variables (not unlike the transformation from

HAMILTON CIRCUIT to INTEGER PROGRAMMING, although considerably more involved). The full details are omitted.

The equivalences shown in Theorem 3 establish the class NP as an important, stable concept. A crucial question thus arises, concerning the relationship between NP and the previously introduced class P of problems solvable in polynomial time. Since ordinary "deterministic" algorithms are a special case of non-deterministic algorithms (those non-deterministic algorithms which do not use the **go to both** instruction), it follows immediately that $P \subseteq NP$. Is $P = NP$? In other words, can we simulate deterministically any non-deterministic algorithm without sacrificing more than a polynomial amount of time? This seems extremely unlikely. Any direct way of performing such a simulation takes exponential time (due to the potential a non-deterministic algorithm has for generating an exponential number of parallel computations after only a polynomial number of non-deterministic "steps"). Furthermore, researchers havefor years been attempting without success to find polynomial-time algorithms for certain problems in NP, such as the TSP.If it were discovered that $P = NP$, and, as if by some master stroke, all those frustrating problems suddenly became polynomially solvable, these researchers (and many others) would be greatly surprised. Thus it is widely conjectured that $P \neq NP$. However, no proof of this conjecture has yet been found. This question is the central open problem in computer science today, and one of the most important open problems in mathematics.

Since NP contains TSP(decision), we cannot hope to show that the TSP cannot be solved in polynomial time without first proving $P \neq NP$, an apparently awesome task. We can, however, do something almost as good. In the next section we show that the two conjectures, $P \neq NP$, and TSP(decision) $\notin P$, are equivalent. The key concept is that of *NP-completeness.*

3. *NP*-Completeness

3.1 Definition

We say that a decision problem is *NP*-complete if (a) A $\in NP$ and (b) every problem in NP is polynomially transformable to A. The conplexity of an *NP*-complete problem is intimately related to our conjecture that $P \neq NP$. By (a), if $P = NP$ and A is *NP*-complete, then A $\in P$. On the other hand, by (b) and Lemma 1, if A $\in$ P then $NP = P$. Thus if A is *NP*-complete, it is solvable in polynomial time if and only if $P = NP$. If, as is widely believed, $P \neq NP$, then none of the *NP*-complete problems can be in P.

During the past decade, many problems have been proved *NP*-complete. [Garey and Johnson 1979] contains a list of over 300 examples. We have already discovered one in this chapter: it follows directly from Theorem 3 that INTEGER PROGRAMMING is *NP*-complete! In this section we shall show that TSP(decision) is also *NP*-complete. This is our main negative result concerning the TSP; it is the strongest negative result one can hope to prove, short of establishing that $P \neq NP$.

One need not prove a new version of Theorem 3 every time one wants to show a new problem A $\in NP$ to be *NP*-complete. By the transitivity of polynomial transformability, all we need show is that some *already known* *NP*-complete problem B transforms to A.

At this point we have only one "known" *NP*-complete problem: INTEGER PROGRAMMING. However, by Theorem 2 a second one can be quickly added to our list, since that theorem provides a polynomial transformation from INTEGER PROGRAMMING to 0–1 PROGRAMMING. From this start a whole family tree of *NP*-complete problems can be generated, although we will concentrate on that part of the tree that leads to the TSP. (Historically, the "grandaddy" problem for *NP*-completeness was not exactly INTEGER PROGRAMMING, but a problem in mathematical logic called SATISFIABILITY –see exercise 5 –, and [Cook 1971], [Aho, Hopcroft and Ullman 1974], [Garey and Johnson 1979], [Papadimitriou and Steiglitz 1981]).

We note in passing that there are many problems which can be solved in polynomial time if and only if $P = NP$, but which do not make it into the family tree for technical reasons. For example, the TSP itself (optimization version) cannot be *NP*-complete because it is not a decision problem, even though it is equivalent in complexity to TSP(decision) which *is* *NP*-complete. A problem in NP to which an *NP*-complete problem is polynomially *reducible* (but not known to be transformable) also may not be called *NP*-complete, even though it has the same claim to intractability as any *NP*-complete problem. Finally, there are decision problems which satisfy part (b) of the definition but not (a) (they are not known to be in NP). For all these kinds of problems we have reserved the term *NP-hard*. A problem (decision or otherwise) is *NP*-hard if all problems in NP are polynomially reducible (not necessarily transformable) to it.

3.2 Special Cases

We shall actually be proving complexity results for a number of different special cases of the TSP. We say problem A is a special case of problem B if both problems ask the same question and the domain of A (the set of possible instances) is a subset of the domain of B. A special case may be easier than the general problem (Chapter 4 presents polynomial-time algorithm for a number of special cases of the TSP), or it may be just as hard (*most* interesting special cases of TSP(decision) are just as NP-complete as the general problem, as we shall see shortly). A special case cannot be harder (as long as there is an efficient way to tell instances that belong to the special case domain from those that don't).

Fig. 7 depicts the position of the TSP within a hierarchy of special cases, with specilization increasing as we go down the figure. The general, asymmetric TSP is near the top. Above it we have *generalizations* of the TSP, some of which will be discussed in Chapter 5. A standard special case is the symmetric TSP (restricted to only those instances where $c_{ij} = c_{ji}$ for all cities i and j)—this special case is often what people mean when they refer to the "TSP".

Another way to restrict the general TSP is to require that the distance matrix satisfy the *triangle inequality*: $c_{ij} + c_{jk} \leq c_{ik}$ for all i, j, k. Of course, we may ask that *both* conditions be satisfied, in which case we obtain an important special case, examined in Chapter 5. The *mixed Chinese postman problem* (defined in exercise– and discussed more fully in Chapter 5) can be considered as a (further) special case of the TSP with asymmetric distance matrices that obey the triangle inequality.

HAMILTON CIRCUIT, a purely graph-theoretic problem, can be viewed as a special case of the symmetric TSP with triangle inequality. Given a graph $G = (V, E)$ we can think of it as a $|V| \times |V|$ distance matrix with distance 1 between vertices that are connected by an edge, and 2 otherwise (any path involving two or more edges has length 2 or more, so the triangle inequality is obeyed). A Hamilton circuit exists if and only if there is a tour of length $|V|$. Similarly the HAMILTON CIRCUIT problem for directed graphs is a special case of the (asymmetric) TSP with triangle inequality, and a generalization of the undirected HAMILTON CIRCUIT problem.

Another important special case of the symmetric TSP with triangle inequality is the *Euclidean* TSP. The cities are given as points with integer coordinates in the two-dimensional plane, and their distances are computed according to the Euclidean metric: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ for two cities (x_1, y_1) and (x_2, y_2).

There are technical problems with the Euclidean TSP as stated. The inter-city distances can be irrational numbers, and so the distance matrix might require infinite preci-

sion if input directly. Hence it must be given implicitly (either by the values of c_{ij}^2, or by merely giving the coordinates of the cities). Secondly, even if such alternative formats are used, there is still the problem of evaluating tour lengths which are the sums of square roots. See Fig. 8. In comparing two tour lengths to see which is better, (or in comparing a tour length to a given bound in TSP(decision)), how much work is involved? Each additional decimal place computed costs only a polynomial amount of time, but *will* a polynomial number of decimal places suffice? All that is currently known is that an exponential number of places will do the trick. Thus there is some question as to whether TSP(Decision), for the Euclidean version, is even in NP.

We bypass this difficulty by re-defining the distance between (x_1, y_1) and (x_2, y_2) to be $\lceil\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}\rceil$, the value of the standard Euclidean distance rounded up to the next integer. This (discrete) Euclidean distance still obeys the triangle inequality (see Exercise 19). There is some loss of precision, but this can be moderated if in the beginning we multiply all city coordinates by an appropriately large number, so as to guarantee the degree of accuracy desired.

A related variant is the *rectilinear* (or *Manhattan*) TSP: the distance between (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$ (no problems of precision here).

At the bottom of our chart is a final special case. Suppose that in Euclidean TSP(decision), the, bound B equals the number n of cities. Since each city has integer coordinates, each pair of cities is at least distance 1 apart. Thus a tour of the desired length exists if and only if there is a Hamilton circuit in the graph G whose vertices are the cities, with edges only between those cities that are distance one apart in the plane. Such a graph is called a *grid graph* (because it is a vertex-induced subgraph of the infinite grid).

We shall show that HAMILTON CIRCUIT for grid graphs, the ultimate special case of the TSP, is NP-complete. Since complexity propagates upwards, this will imply that all the decision problems in the tower of Figure 7 are NP-complete. (A separate proof is needed for the mixed Chinese Postman problem, which is also NP-complete—see [Papadimitriou 1976]). Our proof will follow from a series of transformations, starting with 0–1 PROGRAMMING and proceeding through 2 main intermediate problems: EXACT COVER and HAMILTON CIRCUIT (the unrestricted version).

3.3 From 0–1 PROGRAMMING to EXACT COVER

Theorem 2 showed how to polynomially transform INTEGER PROGRAMMING to 0–1 PROGRAMMING. We now show how to transform the latter problem to a further special case: the one in which the matrix A is a 0–1 matrix and the vector b is all ones. This special case has a name: the EXACT COVER problem. It can have the following interpretation. We are given a family $F = \{S_1, \ldots, S_n\}$ of subsets of a set $U = \{u_1, \ldots, u_m\}$.

U corresponds to the set of rows A, and the columns of A are the characteristic vectors of the S_j's. We are asked to find an *exact cover* of U, i.e., a subfamily $C \subseteq F$ such that a C corresponds to a 0–1 solution x for $Ax = b$.

We shall show how to transform any 0–1 program $Ax = b$ to an equivalent one that conforms to the restriction of EXACT COVER. First, we must get rid of the negative entries in A. For each variable x we introduce a new variable x' and a new equation $x + x' = 1$. Then we replace x by $(1 - x')$ in each equation from A where x had a negative coefficient, yielding a new equation with x' instead of x and a positive coefficient instead of a negative one. The equations resulting after all such substitutions have been made may have new values for the constant on their right-hand side. If any right hand side is negative, we know that the 0–1 program has no solution (all coefficients and variables must be non-negative, so the result cannot be negative). We can then construct a trivial instance of EXACT COVER having no solution, such as $0 = 1$, and be guaranteed that the answer for a constructed instance is the same as that for $Ax = b$.

Assuming that all right hand sides are non-negative, let $A'x = b'$ be the new 0–1 program. This is not yet an instance of EXACT COVER, since A' could have fairly large integers as entries, as could b'. In order to replace these with 0–1 entries, we consider the binary representations of the coefficients in each equation, and write appropriate equations to capture binary additions.

We illustrate this by an example. Suppose we had the equation

$$6x_1 + 5x_2 + 7x_3 + 11x_4 = 12$$

The binary expansions of 6, 5, 7, 11, and 12 are 0110, 0101, 0111, 1011, and 1100, respectively. We write the following four equations, one for each *bit* of the coefficients, the least significant first

$$\begin{aligned} x_2 + x_4 &= 0 + 2z_1 \\ x_1 + x_3 + x_4 + z_1 &= 0 + 2(z_2 + z_3) \\ x_1 + x_2 + x_3 + z_2 + z_3 &= 1 + 2(z_4 + z_5) \\ x_4 + z_4 + z_5 &= 1 \end{aligned}$$

The z_i's represent *carry bits*, and we may need up to n of these for each row. When we re-write equations so that all variables are on the left-hand side, we once again get negative coefficients (-2's) which we remove by replacing $-2z$ by $z' + z'' - 2$ in its equation and adding the new equations $z + z' = 1$ and $z + z'' = 1$. After having performed these substitutions we at last obtain a collection of equations, all of whose coefficients are 0 or 1. The right hand sides, however, can be anywhere from 0 to $2n + 1$, and our definition of EXACT COVER requires that they all equal 1.

We get around this final obstacle as follows. If the right-hand side is 0, the equation can simply be dropped, and all its variables set to 0 in (and hence dropped from) the other equations. Suppose we have an equation whose right-hand side is equal to $k > 1$ and whose left-hand side has l non-zero coefficients. We replace the equation by $k + 2l$ new equations with right-hand side equal to 1. For example, the equation

$$x + y + z + w = 3$$

would be replaced by the following 11 equations

$$\begin{aligned}
x + x' &= 1 \\
x_1 + x_2 + x_3 + x' &= 1 \\
y + y' &= 1 \\
y_1 + y_2 + y_3 + y' &= 1 \\
z + z' &= 1 \\
z_1 + z_2 + z_3 + z' &= 1 \\
w + w' &= 1 \\
w_1 + w_2 + w_3 + w' &= 1 \\
x_1 + y_1 + z_1 + w_1 &= 1 \\
x_2 + y_2 + z_2 + w_2 &= 1 \\
x_3 + y_3 + z_3 + w_3 &= 1
\end{aligned}$$

Consider the first two equations. If $x = 0$ then all three of x_1, x_2, and x_3 must be 0. If $x = 1$ then exactly two of x_1, x_2 and x_3 must be 0 and the third must be 1. Analogous statements hold for the y_i's, z_i's and w_i's, due to the next six equations. Thus the three final equations can be satisfied if and only if exactly 3 of x, y, z and w are equal to 1, precisely the requirement of our original equation.

To summarize our transformation: we first remove the negative coefficients from A, halting prematurely if this leaves a negative entry in b. We then reduce the coefficients of A to 0, 1 and -2 by replacing each equation by a set of equations that simulate it using bit-wise addition. Then the -2 coefficients are removed, leaving only certain entries in b as potential trouble-makers. If an entry in b is 0 the corresponding equation is dropped along with its variables. If an entry exceeds 1 the corresponding equation is again replaced by a set of equations that simulates it, and we at last have an instance in which all entries in A are 0 or 1 and all entries in b are 1, as required by EXACT COVER. Each step only adds a number of equations and/or variables that is bounded by a polynomial in m and n (the dimensions of A) and in the number of bits required to represent the largest entry in A or b. Thus the transformation is a polynomial transformation and we have proved the following result.

Theorem 4 0–1 PROGRAMMING α EXACT COVER

Since EXACT COVER is clearly in NP, this means that EXACT COVER is NP-complete. Something even stronger can be said:

Corollary 4.1 EXACT COVER is NP-complete even when restricted to instances where each row of A contains exactly 2 or 3 entries equal to 1 (set terminology: each element is contained in either 2 or 3 sets).

Proof We show how to transform EXACT COVER to this restricted version of itself. Equations with no non-zero coefficients are either tautologically true ($0 = 0$), in which case they can be dropped, or tautalogically false ($0 = 1$), in which case the instance has no solution and we transform it into a trivial one in the required format which has no solution (e.g., $x + y = 0, y + z = 0, x + y + z = 1$). Equations with one non-zero coefficient (e.g., $x = 1$) give the value of the corresponding variable, which then can be substituted in all other equations. The resulting A will still have all 0–1 entries, and all entries of b are one or less. If any such entry is negative, there is again no solution, and we proceed as above. If an entry is 0, then we drop the equation, and also drop all variables that appear to this equation from all other equations.

Finally, consider an equation in which four or more variables have coefficient 1:

$$x_1 + x_2 + x_3 + \ldots + x_k = 1, k \geq 4$$

We replace this equation by the following system of $2k - 5$ equations, having y_i's as new variables.

$$\begin{aligned} x_1 + x_2 + y_1 &= 1 \\ y'_1 + x_3 + y_2 &= 1 \\ &\vdots \\ y'_{k-3} + x_{k-1} + x_k &= 1 \\ y_1 + y'_1 &= 1 \\ &\vdots \\ y_{k-3} + y'_{k-3} &= 1 \end{aligned}$$

It is left as an exercise to verify that this system is equivalent to the above equation.

3.4 From EXACT COVER to HAMILTON CIRCUIT

We shall present a transformation from the restricted form of EXACT COVER given in Corollary 4.1 to HAMILTON CIRCUIT. Our construction employs several "special-purpose subgraphs" (*gadgets* in the jargon of NP-completeness). Take, for example, the graph H shown in Fig. 9(a). It has a very useful property, discribed by the following Lemma:

Lemma 4 Suppose that a graph $G = (V,E)$ contains the graph $H = (W,F)$ of Fig. 9(a) in such a way that no vertex in $V - W$ is adjacent to any of the vertices 3 through 14. Then if C is a Hamilton circuit of G, C must traverse H in one in the two ways shown in Figures 9(b) and 9(c).

Proof Since C is a Hamilton circuit, it must traverse all five paths $[3, 4, 5]$, $[6, 7, 8]$, $[9, 10, 11]$, and $[12, 13, 14]$ (in one direction or the other) as this is the only way to pick up the vertices 4, 7, 10, and 13. Also, exactly one of the edges $\{6, 9\}$ and $\{8, 11\}$ must be traversed. If both are traversed then the cycle $[6, 7, 8, 11, 10, 9, 6]$ is formed. If neither, then both $\{3, 6\}$ and $\{5, 8\}$ must be traversed, and the cycle $[3, 4, 5, 8, 7, 6, 3]$ is formed. If edge $\{6, 9\}$ is traversed, then the traversal shown in 9(b) results, all choices being forced; if $\{8, 11\}$, then the one in 9(c).

What Lemma 4 really says, is that subgraph H behaves as a *pair of edges* $\{1, 15\}, \{2, 16\}$, with the additional constraint that a Hamilton circuit of G must traverse *exactly one* of these edges. For this reason we call H an *exclusive or* subgraph, using the symbolic representation of Fig. 9(d).

For our construction we shall use two more special-purpose subgraphs, graphs J and K shown in Figures 10(a) and (b). Their "crucial properties" are stated in Lemma 5.

Lemma 5 (a) In graph J (Fig. 10(a)), any Hamilton path from vertex 1 to vertex 2 traverses exactly two of the edges in $\{e_1, e_2, e_3\}$; furthermore, for any subset of two of these edges there is such a Hamilton path traversing them. (b) In graph K (Fif. 10(b)), any Hamilton path from vertex 1 to vertex 2 traverses exactly one of the edges in $\{e_1, e_2\}$, and for each of these edges there is a Hamilton path traversing it.

The proof is a straightforward exhaustion of all possibilities, similar to the proof of Lemma 4.

We can now describe our transformation from the restricted form of EXACT COVER to HAMILTON CIRCUIT. Suppose we are given an instance $Ax = b$ of EXACT COVER, where not only are all entries of b 1 and of A 0–1, but also there are either two or three entries equal to 1 in each row of A (i.e., each equation involves two or three variables). We need to construct a graph $G = (V,E)$ such that G has a Hamilton circuit if and only if $Ax = b$ has a 0–1 solution.

Graph G contains, for each of the m equations in $Ax = b$, a copy of the graph J or the graph K, depending on whether the equation has three or two variables, respectively. These copies are linked together in a series (see Fig. 11). For each variable x, the graph contains a pair of parallel edges. These pairs are also linked into a series, and two more edges are added to join the equation series and the variable series into a loop.

Note that, thus far, G certainly has a Hamilton circuit. It traverses the J and K graphs one after another (leaving out one e_j edge in each, according to Lemma 5), and then the pairs of parallel edges, one after the other, leaving out either the left or the right copy in each pair. As Fig. 11 indicates, however, we are not finished constructing G. Using copies of the *exclusive or* subgraph of Fig. 9(a), we are going to connect edges of G in such a way that G no longer automatically has a Hamilton circuit, but has one if and only if $Ax = b$ has the desired solution.

There are 2^n ways in which the Hamilton circuit can traverse the series of parallel edges on the right of Fig. 11. Each of these will correspond to the 2^n possible solution vectors x. If the left edge from the pair corresponding to variable x_j is in the circuit, we take this to mean that $x_j = 1$ (as with x_3 and x_4 in the figure); if the right, then $x_j = 0$ (as with x_1 and x_2).

We use the *exclusive or*s and the copies of J and K to assure that the resulting solution vector actually satisfies the equations: If the kth variable of an equation is x_j, we use an *exclusive or* to connect the e_k edge of the J or K graph corresponding to the equation to the left copy of the pair corresponding to x_j (see again Fig. 11). A Hamilton circuit now will exist if and only if there is a way of traversing the pairs of parallel edges (i.e., a $0-1$ n-vector x) such that in each copy of a J or a K on the left (i.e., in each equation) exactly one of the e_j's is not traversed (i.e., exactly one of the corresponding variables has value 1). Thus G has a Hamilton circuit if and only if $Ax = b$ has a 0-1 solution, and we have proved the following Theorem.

Theorem 5 (Restricted) EXACT COVER α HAMILTON CIRCUIT.

Hence HAMILTON CIRCUIT is NP-complete. As with EXACT COVER, we can actually show a restricted version of HAMILTON CIRCUIT remains NP-complete, using a slight modification of our proof. The graphs that we constructed are already quite constrained. All vertex degrees are either 2 or 3. Also the graph is "almost planar": it can be arrange on the plane so that only exclusive-or subgraphs cross each other (there are four such crossings in Fig. 11). Each crossing can be replaced by the "crossover" configuration of Fig. 12 (right), thus obtaining a planar graph with the same degree bounds. Finally, we can render our graph bipartite (a graph is bipartite if the vertices can be assigned colors black and white, in such a way that no edge joins two vertices with the same color). Each copy of H, J, and K is already bipartite, as is the series of parallel edges. The only trouble spot could be the *exclusive or*'s of Fig. 11. But these can be made bipartite by using the model of *exclusive or* shown in Fig. 13 instead of that in Fig. 9(a) (the crossovers of Fig. 12 can be dealt with similarly). We thus have proved the following:

Corollary 5.1 HAMILTON CIRCUIT is NP-complete even when restricted to planar,

bipartite graphs with vertex degrees either 2 or 3.

3.5 From HAMILTON CIRCUIT to HAMILTON CIRCUIT FOR GRID GRAPHS

Recall that a *grid graph* is a finite, vertex-induced subgraph of the *infinite grid*, i.e., the infinite graph with set of vertices $Z \times Z$ and set of edges $\{\{(x, y), (x', y')\}: |x-x'|+|y-y'| = 1\}$. (See Figure 14). In this Subsection we show that HAMILTON CIRCUIT is *NP*-complete even in the extremely special case of grid graphs. We have already argued that this problem is an extremely restricted special case of the TSP.

We first need a result concerning graph embeddings. An *embedding* of a planar graph G (with all degrees 2 or 3) into the grid is a function which maps the vertices of G to distinct vertices of the grid, and edges of G to vertex-disjoint (except for common endpoints) paths in the grid. Figure 15 shows an example. The *extent* of the embedding is the side of the smallest square that contains the images of all vertices and all edges of the graph –the extent of the embedding shown in Fig. 15 is 4.

Lemma 6 Any planar graph with n vertices and f faces (and all vertices of degree 2 or 3) has an embedding of extent $n + f$ or less.

Proof Recall that, if a graph is planar, then for each of its faces there is a planar representation in which that face is the *external* face (i.e., the one whose boundary is the boundary of the representation of G). We prove, by induction on the number of faces f, that, for any graph G as above, there is an embedding of extent $n + f$ or less, such that (a) a designated face is the external face of the embedding, and (b) each degree-2 vertex in the external face is embedded so that it is the rightmost point of the embedding along the horizontal line drawn through it.

This is certainly true when $f = 2$ (i.e., G is a polygon as in Fig. 16(a)). If $f > 2$, consider the graph G' obtained from G by choosing a face adjacent to the designated one, and deleting all edges and degree-2 vertices common to the two faces. G' has $n' \leq n$ vertices and $f' = f-1$ faces. By induction, we can embed G' on an $(n'+f') \times (n'+f')$ square so that the combined face is external and (b) holds. We then add new vertices and edges to obtain an embedding of G, possibly replacing certain edges of G' by vertical paths, as shown in Fig. 16(b). The addition increases the extent by at most 1 plus the number of new vertices. The Lemma follows.

We shall show that HAMILTON CIRCUIT for planar, bipartite graphs with degrees 2 or 3 is polynomially transformable to HAMILTON CIRCUIT for grid graphs. Given a graph G in the former class, we shall construct in polynomial time a grid graph G' such that G' has a Hamilton circuit if and only if G has one.

We assume that G is given to us along with a planar representation of itself (such a representation is available as a result of our construction in the proof of Theorem 5, and,

if missing, can be generated in linear time from a standard description of the graph using the techniques of [Hopcroft and Tarjan 1974]). Our first step is to embed G onto the grid as per Lemma 6 –it is easy to see that the construction in the proof of Lemma 6 can be implemented in polynomial time.

We wish to do this so that the embedding preserves the bipartiteness of G, in the following sense: Recall that in a bipartite graph the nodes can be divided into "black" and "white" classes, with the edges going from one class to the other. Assume that such a coloration has been assigned to G (this can be easily done in linear time). The grid is *itself* bipartite –we can color all vertices black whose coordinates have an even sum, and color all other vertices white. We wish our embedding to be "color–preserving", i.e., white vertices of G go to white vertices of the grid, and black to black. To do this, we first embed G as per Lemma 6, then multiply the scale by two so that all the vertices of G have black images, and then move all the images of vertices one position to the right, as in Fig 17.

This embedding will be our basic "plan" for the construction of the grid graph G. For our final construction we shall blow up the scale again (by a factor of 9), and replace the vertex and edge images by *boxes* and *tentacles*, respectively, which will simulate the vertices and edges of G.

A *box* is the 3×3 grid graph shown in Fig. 18. As is easily checked, the box has the following agreeable property.

Lemma 7 For all i, j, $1 \leq i < j \leq 4$, there is a Hamilton path from p_i to p_j (as identified in Fig. 18) containing all four edges e_1, e_2, e_3 and e_4.

Tentacles are built from *strips*. A strip is a grid graph like the one shown in Fig. 19a. Its *endpoints* are a, b, c, and d. Two otherwise disjoint strips can be combined by identifying two adjacent endpoints of one with an endpoint and an adjacent vertex of the other, as in Fig. 19b. The endpoints of the combination are the two pairs of adjacent degree-2 vertices. We can continue adjoining new strips as long as we like, as in Fig. 19c. The result is called a *tentacle* if the only induced edges are those of the original strips. The grid graph in Fig. 19c is a tentacle. A key property of tentacles is the following, proved by a parity argument.

Lemma 8 In a tentacle there is a Hamilton path between two endpoints if and only if one is black and the other is white.

A Hamilton path between two adjacent endpoints in a tentacle is called a *return path*; between twp non-adjacent ones, a *cross path.*

We are now ready to describe the grid graph G'. As mentioned above, we first multiply the scale of our previous imbedding by 9, a factor big enough to open up space for the construction, and odd enough to preserve parity. We then expand each vertex which is

the image of a node in G to a box, centered at that vertex (see Fig. 20). Note that the corners of a box correspond to a white vertex of G are all white, and similarly for the black vertices. The paths corresponding to edges of G are then expanded into tentacles. If the edge was (u, v), where u is a white vertex of G and v is a black vertex, the tentacle has two endpoints adjacent to vertices of the box corresponding to u, and only one endpoint adjacent to the vertex of the box corresponding to v. (For example, take $u = x_2$ and $v = y_2$ in Fig. 20). It is easy to see that this can always be done, because of the parity-preserving property of our embedding.

We now claim that G' has a Hamilton circuit if and only if G has one. First suppose that G has a Hamilton circuit C. The corresponding tour of G' follows the image of C in G'. A tentacle corresponding to an edge of C is traversed by a cross path, thus linking the two boxes corresponding to its endpoints. Boxes corresponding to black vertices of G are traversed by a Hamiltonian path from an endpoint of the tentacle corresponding to the "incoming edge" of C to an endpoint of the tentacle corresponding to the "outgoing edge".

A box corresponding to a white vertex is traversed analogously, except that a detour is made (around e_1, e_2, e_3 or e_4) to traverse by a return path any tentacle corresponding to an edge of G with the given white vertex as endpoint that is *not* in C (there can be at most one such detour per box since the maximum vertex degree in G is three). The existence of the required paths through boxes and tentacles is guaranteed by Lemmas 7 and 8. See Figure 20.

Conversely, suppose G' has a Hamilton circuit C'. Each black box of G' has at most three edges to the outside world in G' (*external edges*). Hence exactly two of these must be used by C', with the corresponding tentacles traversed by cross paths. In the case where a black box has three external edges in G', the tentacle corresponding to its untraversed external edge must be picked up by a return path from its other endpoint.

Now consider the white boxes of G'. Each is adjacent to two or three tentacles, at least one of which must be traversed by a cross path if the box is to be connected to the outside world by C'. Since C' must contain an even number of the external edges for any box, and since a cross path used only one of a pair of external edges, this means that exactly two of the tentacles associated with a white box must be traversed by cross paths.

Finally, consider the subgraph C of G consisting of those edges of G that correspond to tentacles of G' traversed by cross paths of C'. All vertices in this subgraph have degree two. Moreover, the subgraph is connected since tentacles traversed by return paths in C' do not connect boxes in C'. Thus C is the desired Hamilton circuit for G.

This completes the proof of the theorem which has been our ultimate goal in this section (a goal that has taken us so long to reach that the reader may wish to go back to

Subsections 3.1 and 3.2 for a reminder of its significance.)

Theorem 6 [Itai et al. 1982] HAMILTON CIRCUIT is NP-complete even when restricted to grid graphs.

4. More Bad News

In the previous section we saw even very special cases of the TSP were NP-complete and hence likely to be intractable. In this section we examine the complexity of a number of problems related to the TSP, and of some *approaches* to solving the TSP. As we shall see, the outlook continues to be bleak.

4.1 Suboptimality

In Chapter 5, we shall prove some negative results concerning the possibility of even *approximating* the optimal solution of the general TSP. A similar issue is that of *suboptimality*. As we have seen, finding an optimal tour is likely to be a hopeless task. What about recognizing an optimal tour when we see one? This can be formulated as the following decision problem.

TSP SUBOPTIMALITY

INSTANCE: An instance (n, C) of the TSP and a tour π of the n cities.

QUESTION: Is π suboptimal, i.e., is there a second tour π' of shorter length?

TSP SUBOPTIMALITY is clearly in NP—an optimal tour can, as in TSP(decision), serve as a certificate for any "yes" instance. Unfortunately, this is about the best we can say, as the problem is itself NP-complete. This is a consequence of the following result derived from [Papadimitriou & Steiglitz 1977].

Theorem 7 Given an undirected graph $G = (V, E)$ with a Hamilton circuit C, it is NP-complete to determine whether G has a *second* Hamilton circuit.

Proof We use a transformation from the HAMILTON CIRCUIT problem. Suppose $G = (V, E)$ is a graph. We show how to add an artificial Hamilton Circuit to G, such that the edges of this circuit cannot "mix" with other edges of G to form a new Hamilton circuit. This is done by replacing each vertex v of G by the subgraph S pictured in Fig. 21a, which uses the exclusive-or of the previous section. We then add edges $\{A(u), B(v)\}$ and $\{A(v), B(u)\}$ for each edge $\{u, v\}$ in the original graph. We obtain our artificial Hamilton circuit by ordering the vertices of G (arbitrarily) as $v_1, v_2, \ldots, v_n$ and adding the edges $\{C(v_i), D(v_{i+1})\}, 1 \leq i < n$, and $\{C(v_n), D(v_1)\}$.

Subgraph S can be traversed by a Hamilton circuit in just two ways, shown in Fig. 21b and 21c. By our construction, all copies of S must be traversed in the same way. If they are traversed as in Fig. 21c, then we get our artificial circuit. If as in Fig. 21b, then we get a circuit if and only if G had one.

Corollary 7.1 TSP SUBOPTIMALITY is NP-complete even in the case of symmetric distance matrices that obey the triangle inequality.

Proof Let $G' = (V', E')$ be the graph produced by the above construction. The vertices in V' will be the cities, and the inter-city distances $d(u, v)$ will be defined as follows:

$$d(u,v) = \begin{cases} 4, & \text{if } \{u,v\} \notin E'; \\ 3, & \text{if } \{u,v\} = \{C(v_n), D(v_1)\}; \\ 2, & \text{if } \{u,v\} \in E' - \{C(v_n), D(v_1)\}. \end{cases}$$

Note that the corresponding distance matrix will obey the triangle inequality.

The shortest possible tour length is $2|V'|$, and furthermore a tour cannot use any non-edges of G' or any "artificial" edges (if it used one artificial edge, it would have to use $\{C(v_n), D(v_1)\}$. Hence a tour of this length exists (and our artificial tour, having length $2|V'| + 1$ is suboptimal) if and only if the original graph had a Hamilton circuit.

The NP-completeness of TSP SUBOPTIMALITY has some interesting consequences concerning the effectiveness of *local search heuristics* for the TSP. These issues will be examined in Chapter 5.

4.2 Restricted Spanning Tree Problems

The traveling salesman problem can be viewed as a special case of the following spanning tree optimization problem. Let us fix a family F of *prototype trees*, such as the family of paths (Fig. 22a), that of stars (Fig. 22b), or the remaining families shown in Fig. 22. (The notion of a family of trees can be made formal; see [Papadimitriou & Yannakakis 1981]). The *optimization problem for* F is defined as follows: Given a symmetric distance matrix C, find the shortest spanning tree which is isomorphic to a tree in the family F of prototypes.

If F is the family of paths, then we have the TSP (actually, the "path TSP", or "wandering salesman" problem, which is equivalent to the TSP, see Exercise 4). For the family of stars (Fig. 22b) the problem is trivial—just try each possible center. For the family of Fig. 22c, the problem turns out to be equivalent to the polynomial-time solvable *weighted matching* problem (see Exercise 20 and the Appendix). On the other hand, the problem corresponding to the family of trees in Fig. 22d is NP-hard, as it is intimately related to 3-DIMENSION MATCHING (see [Garey & Johnson 1979]). The question that arises is, "How can we tell the hard families of prototypes from the easy ones?" What distinguishes the families of 22a and 22d from those of 22b and 22c?

The answer involves what is called the *dissociation number* $d(T)$ of a tree T. Given a tree T, $d(T)$ is the smallest number of vertices whose deletion reduces the tree into a collection of isolated edges and vertices. For example, a path with n vertices has dissociation number $\lfloor n/3 \rfloor$. The following theorem becomes a generalization of the result that the TSP is NP-hard.

Theorem 8 [Papadimitriou and Yannakakis 1981] If F is a family of trees and there is an $\epsilon > 0$ such that for each $T = (V, E)$ in F, $d(T) > |V|^{\epsilon}$, then the optimization problem for F is NP-hard.

The (unconstrained) minimum spanning tree (MST) problem can, of course, be solved in polynomial time (see the Appendix). The discussion above reveals that this tractability is not particularly robust. Add a few side conditions on the desired trees and you may well find yourself facing the TSP. Another example of this non-robustness comes from the following variants, one for each $k \geq 2$.

DEGREE-k MST

INSTANCE Integer $n \geq 3$, $n \times n$ distance matrix C.

QUESTION What is the shortest spanning tree for T in which no vertex has degree exceeding k?

The DEGREE-2 MST is the TSP (again the wandering salesman version) and hence it is NP-hard. It is not difficult to show that the DEGREE-k MST is NP-hard for *any* $k \geq 2$. (Exercise 11(a)) An interesting question remains however: What is the complexity of the *Euclidean* special cases of these problems?

It can be shown (see Exercise 11(b)) that the minimum spanning tree of a set of integer coordinate points on the plane *never* contains a vertex of degree 6 or greater. Hence there is a polynomial-time algorithm for the EUCLIDEAN DEGREE-k MST for all $k \geq 5$: namely, the algorithm which simply finds the shortest unconstrained MST. On the other hand from Section 3 and Exercise 4 we have that the EUCLIDEAN DEGREE-2 MST (the Euclidean TSP, that is) is NP-hard. This leaves us with two cases: $k = 3$ and $k = 4$. In [Papadimitriou and Vazirani 1981] it is shown that the EUCLIDEAN DEGREE-3 MST is NP-hard; the EUCLIDEAN DEGREE-4 MST is still open.

4.3 More on Complexity Classes

In this section we discuss some distinctions that will be meaninglesss if it turns out that $P = NP$, but in the more likely event that $P \neq NP$ give us ways of showing that certain variants of the TSP are even "harder" than the basic decision problem version. To do this we must first revisit the classes P and NP and their vicinity. The general hypothesis is that P and NP are related as in Fig. 23, that is, $P \neq NP$ and all the NP-complete problems belong to $NP - P$. There are other classes of interest. The class $coNP$ contains the *complements* of all problems in NP. For instance, the complement of TSP(decision) is given as follows:

TSP(complement)

INSTANCE Integer $n \geq 3$, $n \times n$ distance matrix C, integer B.

QUESTION Is it true that all tours have costs exceeding B?

This is the same as TSP(decision) except that the "yes" and "no" answers have been reversed. Similarly, HAMILTON CIRCUIT(complement) asks whether a given graph has *no* Hamilton circuit, INTEGER PROGRAMMING(complement) asks whether a given integer program has *no* solution, etc.

Due to the basic asymmetry in the definition of NP, it seems quite likely that $NP \neq coNP$. (What could be a succinct certificate of the fact that every tour has cost *exceeding* B? A listing of all tours together with their lengths would be a certificate, but it certainly would not be succinct). However, no proof that $NP \neq coNP$ is yet known, and for good reason: it would imply $P \neq NP$. (Exercise 13(c)). What is certain, however, is that the complements of the NP-complete problems are the least likely members of $coNP$ to be in NP, exactly as the NP-complete problems are the least likely members of NP to be in P (see Exercise 13(a)).

We can use NP and $coNP$ to define a class that apparently includes even harder problems, namely the class D^P. This class contains just those problems which are the *intersection* of a problem in NP with one in $coNP$. That is, each problem X in D^P is defined by two problems X_1 and X_2 over the same set of instances, with $X_1 \in NP$ and $X_2 \in coNP$, such that the answer for X is "yes" if and only if the answers for *both* X_1 and X_2 are "yes". Here are two typical problems from this class.

EXACT TSP

INSTANCE: Integer $n \geq 3$, $n \times n$ distance matrix C, integer B.

QUESTION: Is the cost of an optimal tour *exactly* equal to B?

MAXIMUM NON-HAMILTON GRAPH

INSTANCE: Graph $G = (V, E)$

QUESTION Is it true that (a) G has no Hamilton circuit, and (b) if we add *any* edge to E then a Hamilton circuit results?

EXACT TSP is the intersection of TSP(decision), which is in NP, with the variant of TSP(complement) which asks if all tours have length B or greater and is in $coNP$. MAXIMUM NON-HAMILTON GRAPH is the intersection of the two problems suggested by two parts, (a) and (b) of its question. Problem (a) is in $coNP$ and (b) is in NP. Note that NP and $coNP$ are both contained in D^P, since any problem is equal to itself intersected with the trivial problem that always answers "yes" and hence is in both NP and $coNP$ (since it is in P).

In analogy with NP, a problem in D^P is said to be D^P-complete if all problems in D^P are transformable to it. Since D^P contains all the problems in both NP and $coNP$, D^P-complete problems are even worse than the NP-complete ones. [Papadimitriou & Yannakakis 1982] show the following:

Theorem 9 EXACT TSP is D^P-complete.

We do not know whether MAXIMUM NON-HAMILTON GRAPH is also D^P- complete. However, it does have a property (shared by the D^P-complete problems) that shows that it is even "harder" than the NP-complete problems:

Theorem 11 If MAXIMUM NON-HAMILTON GRAPH is in NP, then $NP = coNP$.

Proof Suppose that it in NP. We show that this would imply HAMILTON CIRCUIT (complement) is also in NP, which, by Exercise 13(a), implies that $NP = coNP$. HAMILTON CIRCUIT (complement) would be in NP because any graph with no Hamilton circuit would have the following succinct certificate: a set of additional edges that makes it *maximal* non-Hamiltonian, together with a succinct certificate that the augmented graph is indeed maximal non-Hamiltonian. The latter exists by our hypothesis.

It can be shown that the D^P-complete problems have an even stonger property: if they are in either NP *or* $coNP$, then $NP = coNP$ (see Exercise 13(b)). Thus the sense in which EXACT TSP and MAXIMAL NON-HAMILTONIAN GRAPH are "harder" than TSP(decision) and the other NP-complete problems is as follows: Assuming $NP \neq coNP$, these problems do not have the succinct certificate property (nor does the complement of EXACT TSP), whereas all the NP-complete problems do. Note, however, that this is about the only sense in which these problems are "harder". All of the above problems (including the TSP) are polynomially *reducible* to each other (if not transformable) and hence they can all be solved within the same running time bounds (to within a polynomial). If $P = NP$ they will all be in P.

Nevertheless, the distinctions made above are of importance from a theoretical point of view, and we shall make further use of them in the next subsection.

4.4 The Complexity of the TSP Polytope

The TSP is one of many combinatorial optimization problems, easy and hard, which can be reformulated as follows

minimize $c'x$

subject to $x \in V$,

where c is a cost vertor (in the case of the TSP, a representation of the distance matrix C in vector form), x is the vector to be optimized (an $\binom{n}{2}$-dimensional vector of unknowns for the n-city symmetric TSP), and V is a finite set of "feasible" vectors (for the TSP, V consists of the characteristic vectors of all tours, considered as sets of edges).

The advantage of this formulation is that, because the objective function is linear, the optimization problem above is equivalent to the following problem:

minimize $c'x$

subject to $x \in CH(V)$

where $CH(V)$ is the *convex hull* of the set V, i.e., the smallest convex polytope that contains all the vectors in V. (See [Grunbaum 1967] and chapters 8 and 9). Minimizing linear functionals over convex polytopes is supposed to be well-understood —this is what linear programming and the simplex algorithm are all about [Dantzig 1959], not to mention the Ellipsoid algorithm [Khachian 1979].

One catch is that it takes exponentially many inequalities to describe $CH(V)$ in the case of the TSP. The polyhedron $CH(V)$ has, literally, just too many faces. This does not immediatley disqualify us, however. The *polyhedral* approach has worked before in the presence of this impediment, most notably in the case of the weighted matching problem [Edmonds 1965a]. What seems more serious is that the convex polytope for the TSP is tarnished by a number of negative complexity results, that strike at the very core of the known polyhedral techniques. These results suggest that, despite early promise, the polyhedral approach may prove to be no more of an "answer" for the TSP than any of the other methods that have been tried so far (with so little success).

Let us first consider the complexity of *facets*. A facet of an n-dimensional polytope is a face of dimension $n-1$. Alternatively, it is an inequality, satisifed by all vertices of the polytope, and is satisfied with equality by n *affinely independent* vertices (see [Grunbaum] and chapter 8). If the TSP were to be solved, like weighted matching, by applying a variant of the simplex algorithm to the optimization problem over $CH(V)$, we would need at the very least a complete "characterization" of all the facets of the TSP polytope. Such a characterization has been for a long time the "Holy Grail" for researchers in the field. Increasingly more complex classes of facets were discovered (e.g. [Dantzig et al. 1954], [Chvatal 1973] [Maurras 1975] [Groetschel 1980] [Groetschel and Padberg 1979]), but no satisfactory conclusion of the search is in sight. Complexity theory yields an explanation for this failure. A satisfactory characterization of the facets of the TSP polytope would mean, at the very least, that the following problem is in NP:

TSP FACETS

INSTANCE An instance (n, C) of the symmetric TSP, plus an inequality $\sum_{j=1}^{\binom{n}{2}} a_j x_j \leq b$ with b and the coefficients a_j all integers.

QUESTION Is the given inequality a facet of the TSP polytope?.

The following result was recently shown by [Papadimitriou & Yannakakis 1981]

Theorem 12 MAXIMUM NON-HAMILTONIAN GRAPH α TSP FACETS

Thus, by Theorem 11, we have that TSP FACETS is "harder" than the NP-complete problems.

Corollary 12.1 If TSP FACETS $\in NP$ then $NP = coNP$

The same corollary can be proved using linear programming duality [Karp & Papadimitriou 1980]. The proof of Theorem 12 has as a by-product the construction of a new natural and "dense" class of facets of the TSP polytope.

A variant on the TSP FACETS problem is TSP SUPPORTING HYPERPLANES, the problem of recognizing valid inequalities for the TSP polytope that are tight for at least one vertex (not necessarily $\binom{n}{2}$ affinely independent ones). Facets are always supporting hyperplanes, but not vice-versa. For this problem there is a stronger result, proved by a transformation from EXACT TSP.

Theorem 13 [Papadimitriou & Yannakakis 1981] TSP SUPPORTING HYPERPLANES is D^P-complete.

Let us now turn to the question of *adjacency* on the TSP polytope. An *edge* of a convex polytope is a one-dimensional face of the polytope. Two vertices are adjacent if they are the endpoints of an edge. Since the Simplex algorithm moves toward optimality by crawling along the edges of polytope (see [Papadimitriou & Steiglitz 1981] Section 2.9), an understanding of the edges of the TSP polyotpe should be a prerequisite for attacking the TSP polyhedrally. One way of formulating this problem is the following:

TSP NON-ADJACENCY

INSTANCE: An integer n, plus two tours π and π' of n cities.

QUESTION Are π and π' non-adjacent vertices of the TSP polytope for n cities?

Unfortunately, the prospects are not good here either:

Theorem 14 [Papadimitriou 1978] TSP non-adjacency is NP-complete.

A final problem is that of deciding whether a given rational point lies within the TSP polytope or not. Answering such a question would be essential if we wished to attack the TSP using the Ellipsoid algorithm, as suggested in [Browne 1979] for example. As might be expected, this problem too is NP-complete [Papadimitriou & Yannakakis 1981].

For a different perspective on the polyhedral approach, see Chapters 8 and 9. We suspect readers have had enough of bad news by now, and so we shall stop here. Subsequent chapters will show that, despite an overall bleak picture, there are many corners of the world where a traveling salesman can find an occasional ray of light.

References

A.V. Aho, J.E. Hopcroft, J.D. Ullman *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading Mass., 1974.

R. Bellman and S.E. Dreyfus *Applied Dynamic Programming*, Princeton Univ. Press, Princeton, NJ, 1962.

I. Borosh and L.B. Treybig "Bounds on Positive Inbtegral Solutions to Linear Diophantine Equations", *Proc. Amer. Math. Soc.*, 55, pp. 299-304, 1976.

M.W. Browne "An Approach to Difficult Problems", *The New York Times*, 29 November 1979.

A. Cobham "The Intrinsic computational Difficulty of Functions", *Proc. of the 1964 International Congress for Logic, Methodology and Philosophy of Science*, pp. 24-30, ed. Y. Bar-Hillel, North Holland, Amsterdam, 1964.

S.A. Cook "On the Complexity of Theorem-Proving Procedures", *Proc. 3rd ACM Symposium on the Theory of Computing*, pp. 151-158, 1971.

G.B. Dantzig "On the Significance of Solving Linear Programming Problems with Some Integer Variables", *Econometrica*, pp. 30-44, 1960.

G.B. Dantzig *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

G.B. Dantzig, W.O. Blattner, M.R. Rao "All Shortest Routes from a Fixed Origin in a Graph", in *Theory of Graphs: An international Symposium*, Gordon & Breach, New York 1967.

G.B. Dantzig, D.R. Fulkerson, S.M. Johnson "Solution of a Large-scale Travelling Salesman Problem", *Opera:ions Research, 2*, pp. 393-410, 1954.

J. Edmonds "Paths, Trees, and Flowers", *Canadian J. Math., 17*, pp. 449-467, 1965.

J. Edmonds "Matching and a Polyhedron with Zero-One Vertices" *J. Research of the NBS, 69B*, pp.125-130, 1965.

L.R. Ford, Jr. and D.R. Fulkerson *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

M.R. Garey, D.S. Johnson *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, New York, 1979.

M. Groetschel "On the Monotone Symmetric Travelling Salesman Problem: Hypohamiltonian-Hypotraceable Graphs", *Math. of Operations Research, 5*, pp. 285-292, 1980.

M. Groetschel, M. Padberg "On the Symmetric Travelling Salesman Problem II: Lifting Theorems and Facets", *Math. Progrlmming, 16*, pp. 281-302, 1979.

B. Grunbaum *Convex Polytopes*, John Wiley and Sons, 1967.

M. Held, R.M. Karp "A Dynamic Programming Approach to Sequencing Problems", *SIAM J., 10*, pp. 196-210, 1962.

J.E. Hopcroft, R.E. Tarjan "Efficient Planarity Testing", *J.ACM, 21*, pp. 549-558, 1974.

J.E. Hopcroft and J.D., Ullman *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading Mass., 1979.

A. Itai, C.H. Papadimitriou and J. Szwarcfiter "Hamilton Paths in Grid Graphs", *SIAM j. Computing*, in press, 1982.

R.M. Karp "Reducibility among Combinatorial Problems", in *Complexity of Computer Computations*, ed. R.E. Miller and J.W. Thatcher, Plenum Press, 1972.

R.M. Karp and C.H. Papadimitriou "On Linear Characterizations of Combinatorial Optimization Problems", *Proc. 21st FOCS Conference*, pp. 1-9, 1980.

L.G. Khachian "A Polynomial Algorithm for Linear Programming", *Doklady Akad. Nauk USSR*, 244, no.5, pp. 1093-1096, 1979. Translated in *Soviet Math. Doklady*, 20, pp. 191-194.

V. Klee and G.J. Minty "How good is the Simplex Algorithm?", pp. 159-175 in *Inequalities III*, ed. O. Shisha, Academic Press, New York, 1972.

H.R. Lewis and C.H. Papadimitriou *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

J.F. Maurras "Some Results on the Convex Hull of Hamiltonian Cycles of Symmetric Complete Graphs", in *Combinatorial Programming: Methods and Applications* B. Roy (ed.), Reidel, 1975.

C.H. Papadimitriou "On the Complexity of Edge Traversing", *J.ACM, 23*, pp. 544-554, 1976.

C.H. Papadimitriou "On the Complexity of Integer Programming", *J.ACM, 28*, 4, pp. 765-768, 1981.

C.H. Papadimitriou and K. Steiglitz "On the Complexity of Local Search for the Traveling Salesman Problem", *SIAM J. Computing, 6*, 1, pp. 76-83, 1977.

C.H. Papadimitriou and K. Steiglitz *Combinatorial optimization: Algorithms and Complexity*, Prentice-Hall, 1982.

C.H. Papadimitriou and U. Vazirani "On two Geometric Problems Related to the Traveling Salesman Problem", manuscript, 1981.

C.H. Papadimitriou and M. Yannakakis "The Complexity of Restricted Spanning Tree Problems", *J.ACM, 28*, in press, 1981.

C.H. Papadimitriou and M. Yannakakis "The Complexity of Facets (And some Facets of Complexity)", manuscript, 1982.

J. von Neumann, "A Certain Zero-Sum Two-Person Game Equivalent to the Assignment Problem", in *Contributions to the Theory of Games II*, ed. A.W. Kuhn and H.W. Tucker, Princeton Univ. Press, Princeton, NJ, 1953.

Exercises

1. (a) How much *space* is required by algorithm B? Describe a version of algorithm B that requires only polynomial space.

(b) Show that the space requirements of Algorithm DP are $O(n \cdot 2^n)$.

(c) Give an $O(n^2)$ algorithm for SHORTEST PATH.

2. Consider the version of SHORTEST PATH in which we are asked to find the *longest* simple path (path with no repetitions of nodes) between s and t. Show that there is a polynomial-time algorithm for this problem if and only if there is one for the TSP.

3. We are asked to determine an unknown number x between 0 and M by asking questions of the form "is $x > a$?" for different a's of our choice. Show that this cannot be done with fewer than $\lceil \log_2(M+1) \rceil$ questions in the worst case. (Therefore the "binary search" technique employed in the algorithm of Fig. 4 is optimal.)

4. The *wandering salesman problem* (WSP) is the TSP, only that the salesman can start his tour in any city, and does not have to return to the first city in the end.

(a) Give a polynomial-time transformation from TSP(decision) to WSP(decision).

(b) The other way around.

5. A *Boolean variable* is a variable assuming only the values **true** and **false**. Boolean variables can be combined by the operations **or**, **and** and **not** very much the same way that real variables can be combined by $+$, $\times$ and $\surd$. A *literal* is either a boolean variable or the negation of one (an expression of the form **not** x, where x is a Boolean variable). A *clause* is the **or** of several literals. A *Boolean expression in conjunctive normal form (CNF)* is the **and** of a set of clauses. A CNF expression is *satisfiable* if there is an assignment of values to the Boolean variables in it that make the expression **true**. The SATISFIABILITY problem is the following: Given a CNF expression, is it satisfiable?

(a) Show that SATISFIABILITY can be considered a special case of 0-1 PROGRAMMING.

(b) Show that SATISFIABILITY is NP-complete.

(c) Show that SATISFIABILITY remains NP-complete even if there are three literals in each clause. (This problem is called 3-SATISFIABILITY.)

(d) Give a polynomial-time transformation from 3-SATISFIABILITY to the CLIQUE problem (given a graph G and an integer k, is there a completely connected set of k nodes in G?). Hint: Construct a graph with seven nodes for each clause of the Boolean expression.

6. LINEAR PROGRAMMING is the same as INTEGER PROGRAMMING, only the solution is only constrained to be *rational* instead of integer.

(a) Show that LINEAR PROGRAMMING $\in NP$.

(b) Use the duality theory of linear programming to show that LINEAR PROGRAMMING $\in coNP$. (Note that both (a) and (b) follow trivially from the Ellipsoid algorithm for linear programming [Khachian 1979], so do not use that result.)

7. (a) The DIAL-A-RIDE problem is the WSP of Exercise 4, only now the cities are divided into *origins* and the corresponding *destinations*, and the tour is required to visit each origin before the corresponding destination. Show that this problem is NP-complete, and that the Dynamic Programming algorithm DP can be modified to solve this problem. What is the time complexity of this algorithm?

(b) Repeat part (a) for the version of the TSP in which we are also given n integers $k_1, \ldots, k_n$, and we are asked for the shortest tour that visits city 1 k_1 times, city 2 k_2 times, and so on. (It may cost to go from a city to itself.)

8. There are four versions of the HAMILTON CIRCUIT problem: HAMILTON CIRCUIT, HAMILTON PATH, DIRECTED HAMILTON CIRCUIT, DIRECTED HAMILTON PATH (the obvious definitions). Give polynomial-time transformations between all these problems.

9. The *Chinese Postman Problem* is the following: Given a *mixed* graph (i.e., a graph with both directed and undirected edges), with a cost for each directed and undirected edge, find the shortest closed walk of the graph (path with possible repetitions of nodes and edges) that visits each directed and undirected edge at least once. Show that this problem is a special case of the asymmetric TSP with triangle inequality. (Hint: The cities are the midpoints of the edges.)

10. (a) Show that the Euclidean WSP (Exercise 5) is NP-complete.

(b) Show that determining whether a set of points on the plane has a minimum spanning tree (see the Appendix) which is a path, is NP-complete.

(Hint: The construction in the proof of Theorem 6 is useful in both.)

11. (a) Show that the DEGREE-k MST problem is NP-complete, for all $k > 1$.

(b) Show that the Euclidean DEGREE-5 MST is polynomial (Hint: Can the MST of a planar set of points with integer coordinates have a node of degree 6 or more?).

(c) Show that the Euclidean DEGREE-3 MST problem is NP-complete.

12. Show that the HAMILTON CIRCUIT problem remains NP-complete even when restricted to planar graphs with all nodes of degree *exactly* three.

13. (a) Show that, if the complement of an NP-complete problem is in NP, then $NP = coNP$.

(b) Show that, if a D^P-complete problem is in NP *or* in $coNP$, then $NP = coNP$.

(c) Show that, if $P = NP$, then $NP = coNP$.

14. The *bottleneck* TSP is the version in which one tries to minimize the *maximum* among the distances traversed, not their sum. Show that the bottleneck TSP is NP-complete even in the Euclidean case.

15. (a) Show that any problem in NP can be solved in time $O(2^{p(n)})$ for some polynomial p.

(b) Show that any problem in NP can be solved in polynomial *space*.

(c) Repeat (a) and (b) for the class D^P.

16. (a) Show that $NP, coNP \subseteq D^P$.

(b) Show that if $NP \neq D^P$ then all areas shown in Fig. 23 are nonempty.

17. Show, by using the duality theory of linear programming that, if TSP FACETS $\in NP$, then $NP = coNP$.

18. (a) Show that TSP FACETS $\in D^P$.

(b) Show that the problem of telling whether a given rational point is an interior point of the TSP polytope is in NP.

(c) Show that the problem of telling whether two given TSP tours on n cities are nonadjacent vertices of the TSP polytope is in NP.

19. Show that the Euclidean distances among integer points on the plane, rounded up to the next integer, satisfy the triangle inequality.

20. For definitions concerning the weighted matching problem [Edmonds 1965a] see the Appendix. Show that the problem of finding, given a distance matrix over an odd number of cities, the shortest spanning tree isomorphic to the "double star" of Fig.22c, can be reduced to the weighted matching problem.

21. Show that the problem UNIQUE TOUR, asking whether there is exactly one tour in a given instance of the TSP that has a specified length B, is in D^P. How about the problem UNIQUE OPTIMAL TOUR (does the instance have a unique optimum?).

ALGORITHM A

Input: An integer $n \leq 1$ and an $n \times n$ distance matrix C of nonnegative integers.

Output: The shortest tour of n cities.

begin
  min : $= \infty$;
  for all permutations π of $\{2, 3, \ldots, n\}$ **do**
  begin
    cost : $= c_{1\pi(2)} + c_{\pi(n)1} + \sum_{j=2}^{n} c_{j\pi(j)}$;
    if cost < min **then** min : = cost, besttour : $= (1)//\pi$
  end;
  output besttour
end

Note: // Stands for concatenation of sequences of cities. For example, (1,3)//(2,5,4) = (1,3,2,5,4). Permutations are represented as sequences: $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$.

Figure 1

ALGORITHM B

Input and Output: As in Algorithm A

```
begin
    for i = 2, 3, ..., n do
        cost[{i}, i] := c_1i, bestpath[{i}, i] := (1, i);
    for j := 2, 3, ..., n − 1 do
        for each S ⊆ {2, 3, ..., n} with |S| = j do
            for each i ∈ S do
            begin
            cost[S, i] := min_{k∈S−{i}} (cost [S − {i}, k] + c_ki);
            let k be the city that achieves this minimum;
            bestpath[S, i] := bestpath [S − {i}, k]//(k)
            end
    mincost := min_{k≠1} (cost [{2, 3, ..., n}, k] + c_ki);
    let k be the city that achieves this minimum;
    besttour := bestpath [{2, 3, ..., n}, k]//(k);
    output besttour
end
```

Figure 2

ALGORITHM C

Input: As in Algorithm A
Output: The shortest path from city 1 to city n

```
begin
    for i = 2, ..., n do cost[i, 1] := c_1i, bestpath [i, 1] := (1, i);
    for j = 2, 3, ..., n do
        for i = 2, 3, ..., n do
        begin
        cost [i, j] := min_{k≠1,i} (cost[k, j − 1] + c_kj)
        let k be the city that achieves this minimum;
        bestpath[i, j] := bestpath[k, j − 1]//(i);
        if cost [i, j − 1] < cost [i, j] then
            cost[i, j] := cost[i, j − 1], bestpath[i, j] := bestpath[i, j − 1]
        end;
    output bestpath[n, n − 1]
end
```

Figure 3

ALGORITHM TSPTOUR

Input: An integer n, and an $n \times n$ distance matrix C with nonnegative integer entries.

Output: A matrix C with entries either nonnegative integers or ∞. The non-∞ entries are exactly the edges used in the optimum tour.

```
begin
    comment: The algorithm TSPTOUR calls as a subroutine an assumed algorithm
        TSPMEMBERSHIP(n, C, L) which solves the membership version of the TSP.
    low := 0;
    high := n · max_ij[c_ij];
    while low ≠ high do
        if TSPMEMBERSHIP (n, C, ⌊(low+high)/2⌋) = "yes"
            then high := ⌊(low+high)/2⌋
            else low := ⌊(low+high)/2⌋ + 1;
    comment: At this point high = low contains the cost of the optimal tour, found by
        binary search;
    optimum := high;
    for i = 1, 2, ..., n do
    for j = 1, 2, ..., n do
        begin
        remember := c_ij;
        c_ij := ∞;
        if TSPMEMBERSHIP(n, C,optimum) = "no"
            then c_ij := remember
        end
end
```

Figure 4

Function	Approximate Values		
n	10	100	1000
$n \log n$	33	664	9966
n^3	1000	1,000,000	10^9
$10^6 n^8$	10^{14}	10^{22}	10^{30}
2^n	1024	1.27×10^{30}	1.05×10^{301}
$n^{\log n}$	2099	1.93×10^{13}	7.89×10^{29}
$n!$	3,628,800	10^{158}	4×10^{2567}

Table 1

A comparison of the growth of some polynomial functions (above) to that of certain exponential functions.

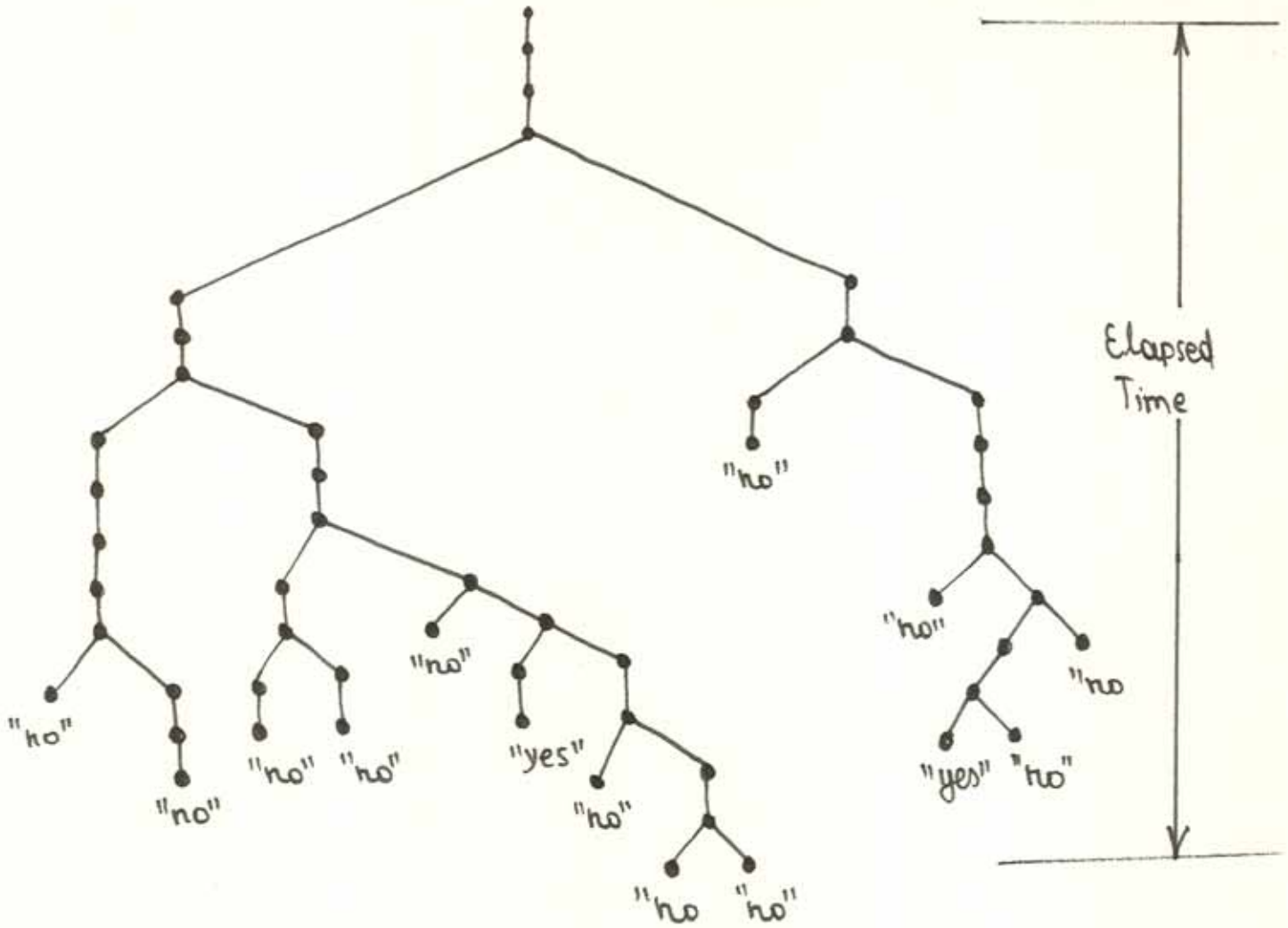


Figure 5

A possible execution history of a nondeterministic algorithm. Branching points are executions of go to both instructions. The answer of this computation is "yes", because there is at least one leaf with answer "yes". The time used by this computation is 16, the depth of the deepest leaf.

ALGORITHM N

Input: An $m \times n$ integer matrix A, integer m-vector b.
Output: "Yes" if there is an $x \in \{0, 1\}^n$ such that $Ax = b$, and "no" otherwise.

```
begin
for j: = 1, ..., n do
    begin
    goto both zero, one;
    zero: x_j := 0; goto again;
    one: x_j := 1;
    again: continue
    end;
if x = (x_1, ..., x_n) satisfies Ax = b
    then output "yes"
    else output "no"
end
```

Figure 6

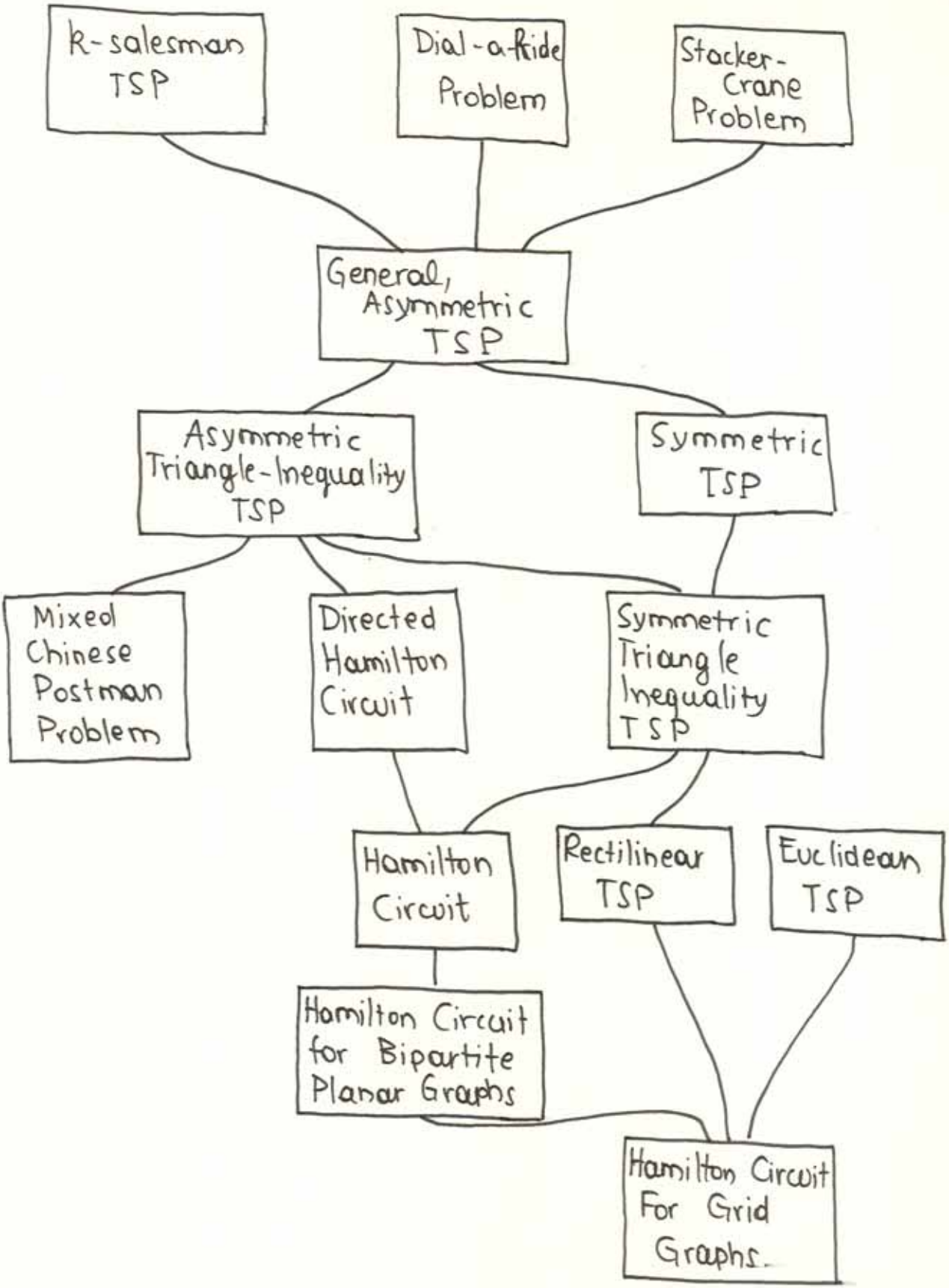


Figure 7

Special cases and generalizations of the TSP.

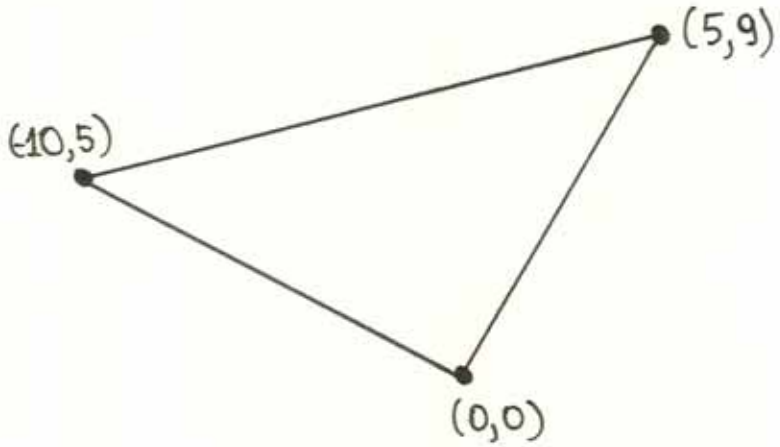
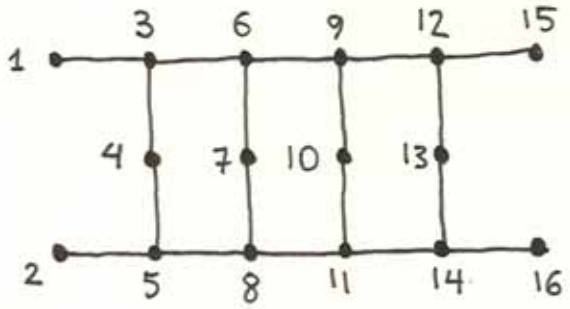
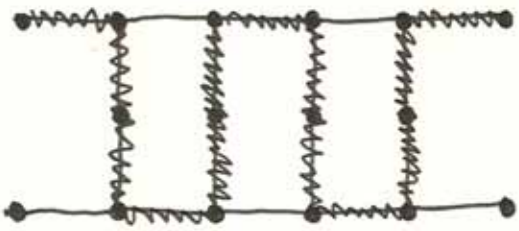


Figure 8

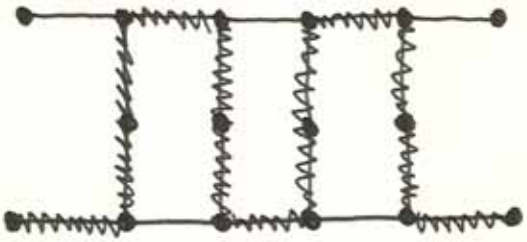
What is the length of the perimeter of this triangle? If calculated with the results truncated to the fifth significant digit, the answer is 36.999; to the eighth significant digit it is 37.000145. There is no known method to predict the number of significant digits that are necessary in order to compare the sum of square roots to an integer.



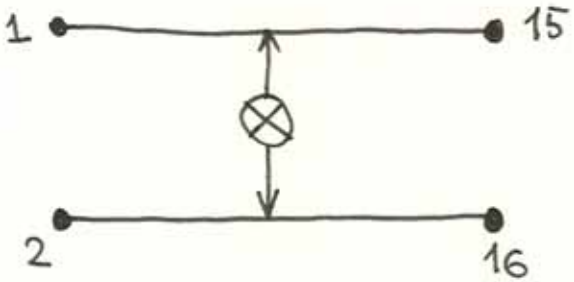
(a)



(b)



(c)



(d)

Figure 9

The exclusive-or subgraph H

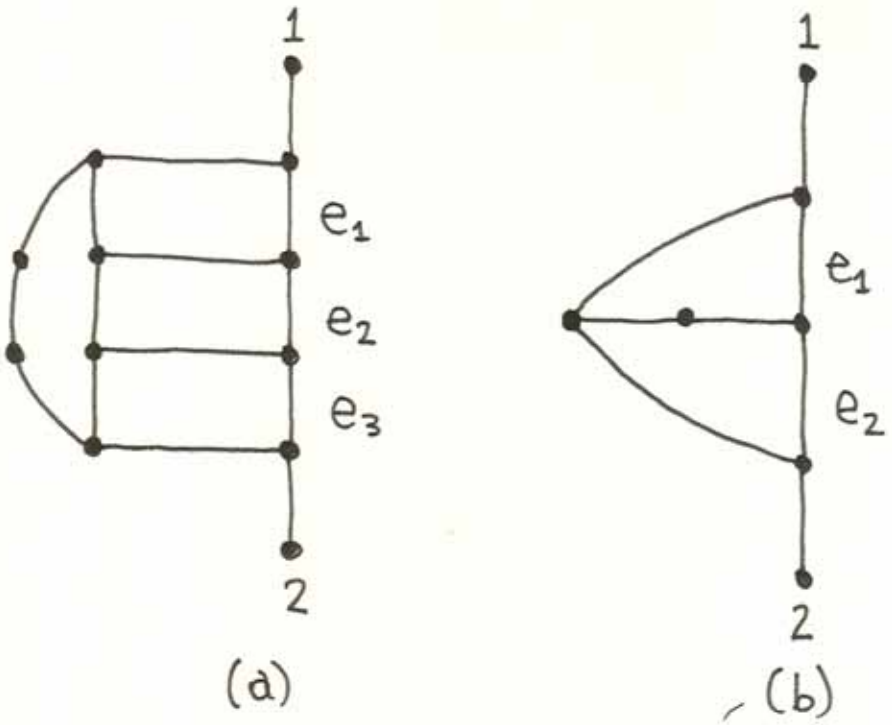


Figure 10

The graphs J and K

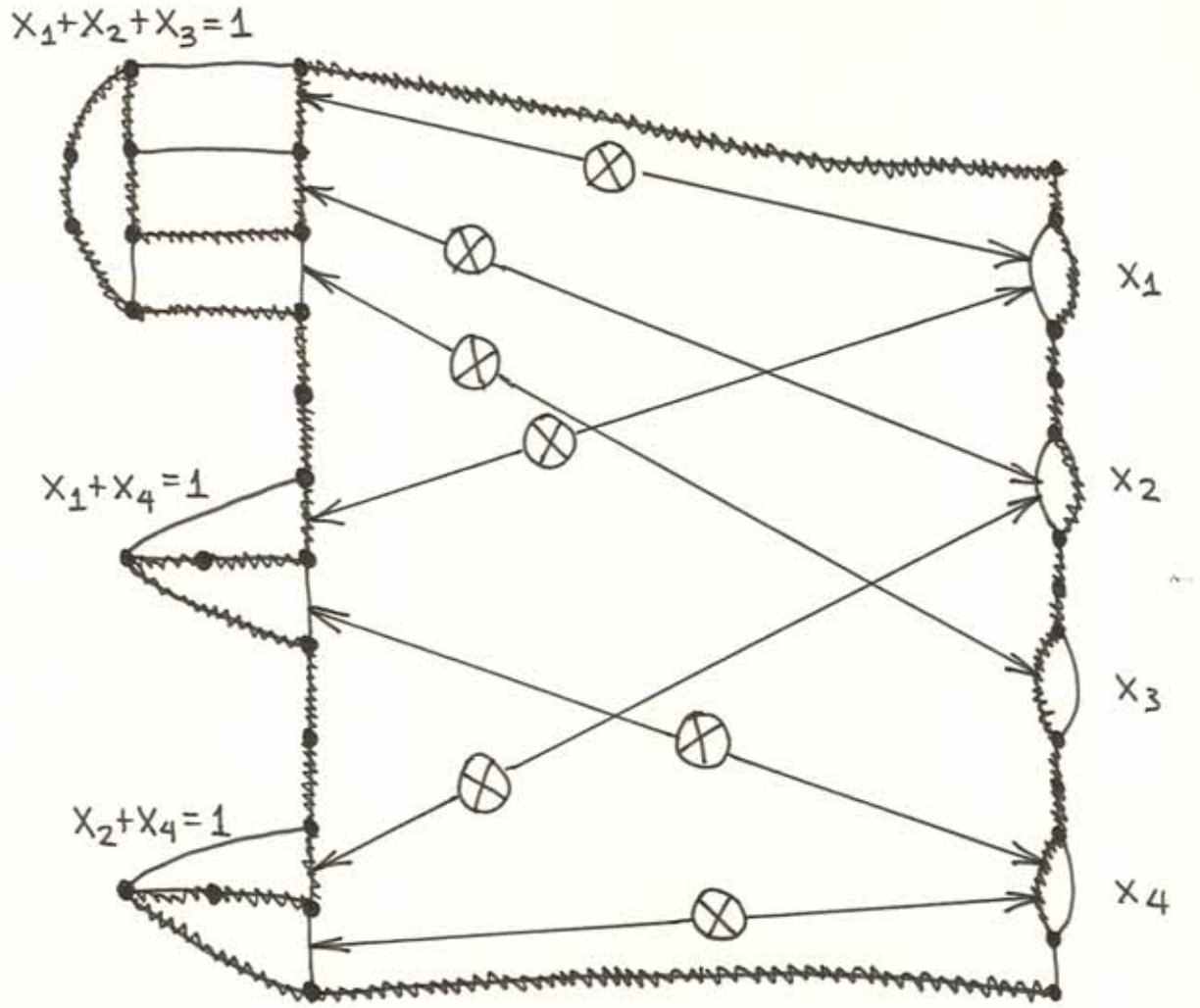


Figure 11

An example of the transformation from EXACT COVER to HAMILTON CIRCUIT.

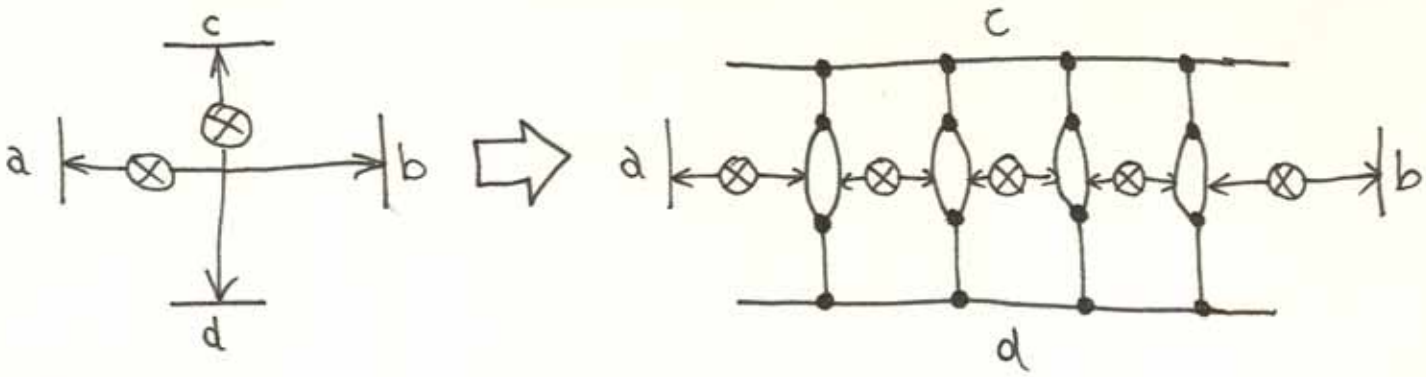


Figure 12

The "crossover".

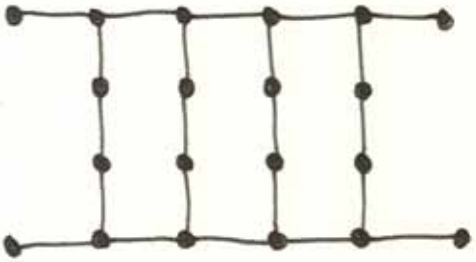


Figure 13

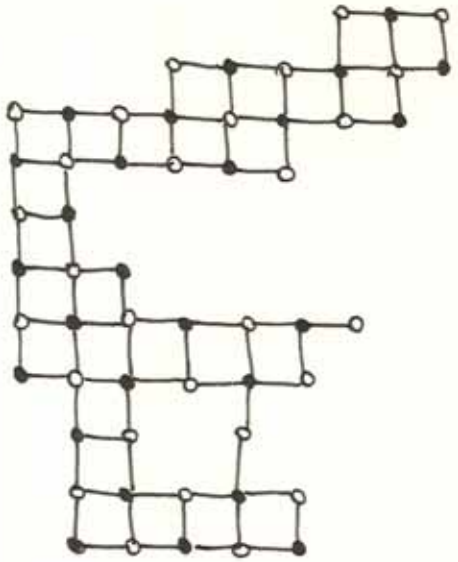


Figure 14

A grid graph

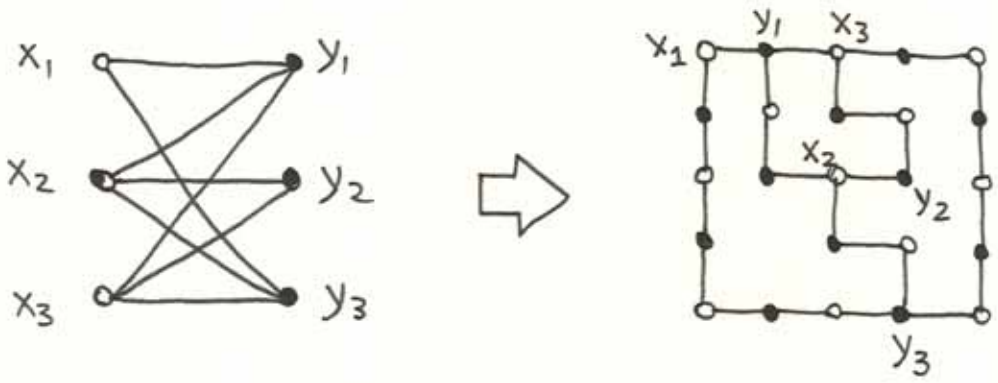


Figure 15

A color-preserving embedding of a planar bipartite graph.

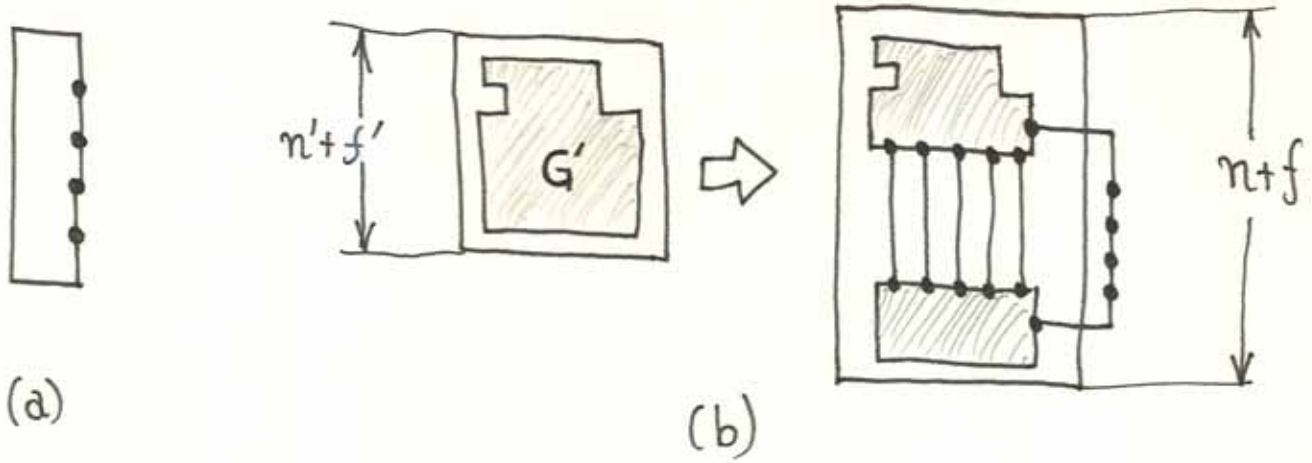


Figure 16

The basis (a) and the induction step (b) in the proof of Lemma 6.

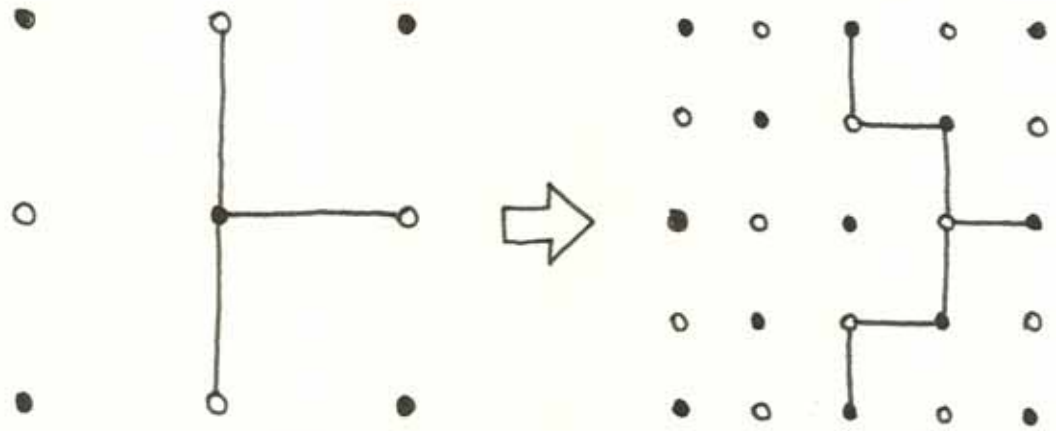


Figure 17

How to make an embedding color-preserving.

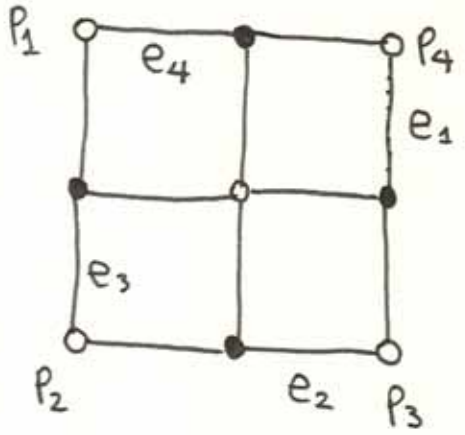


Figure 18

The box

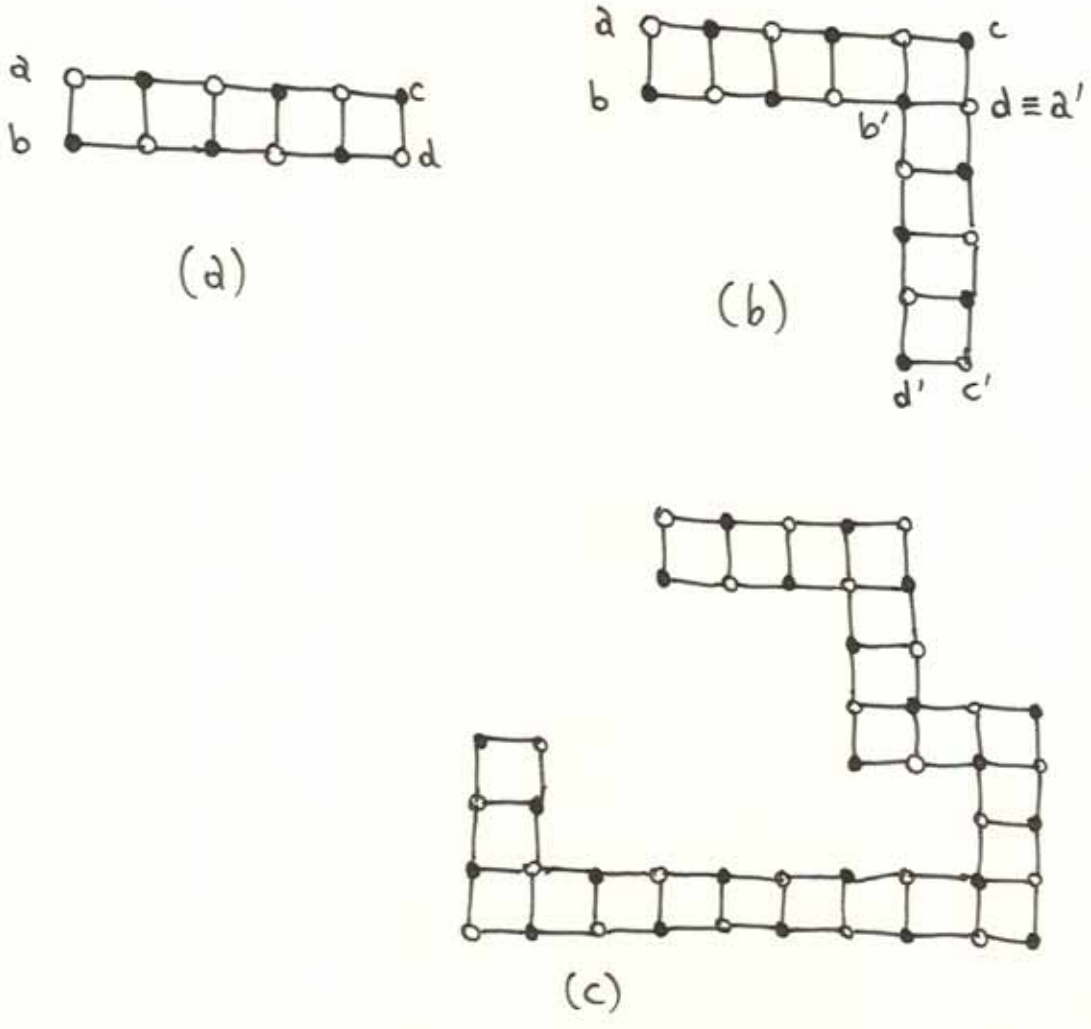


Figure 19

A strip (a), a combination of two strips (b), and a tentacle (c).

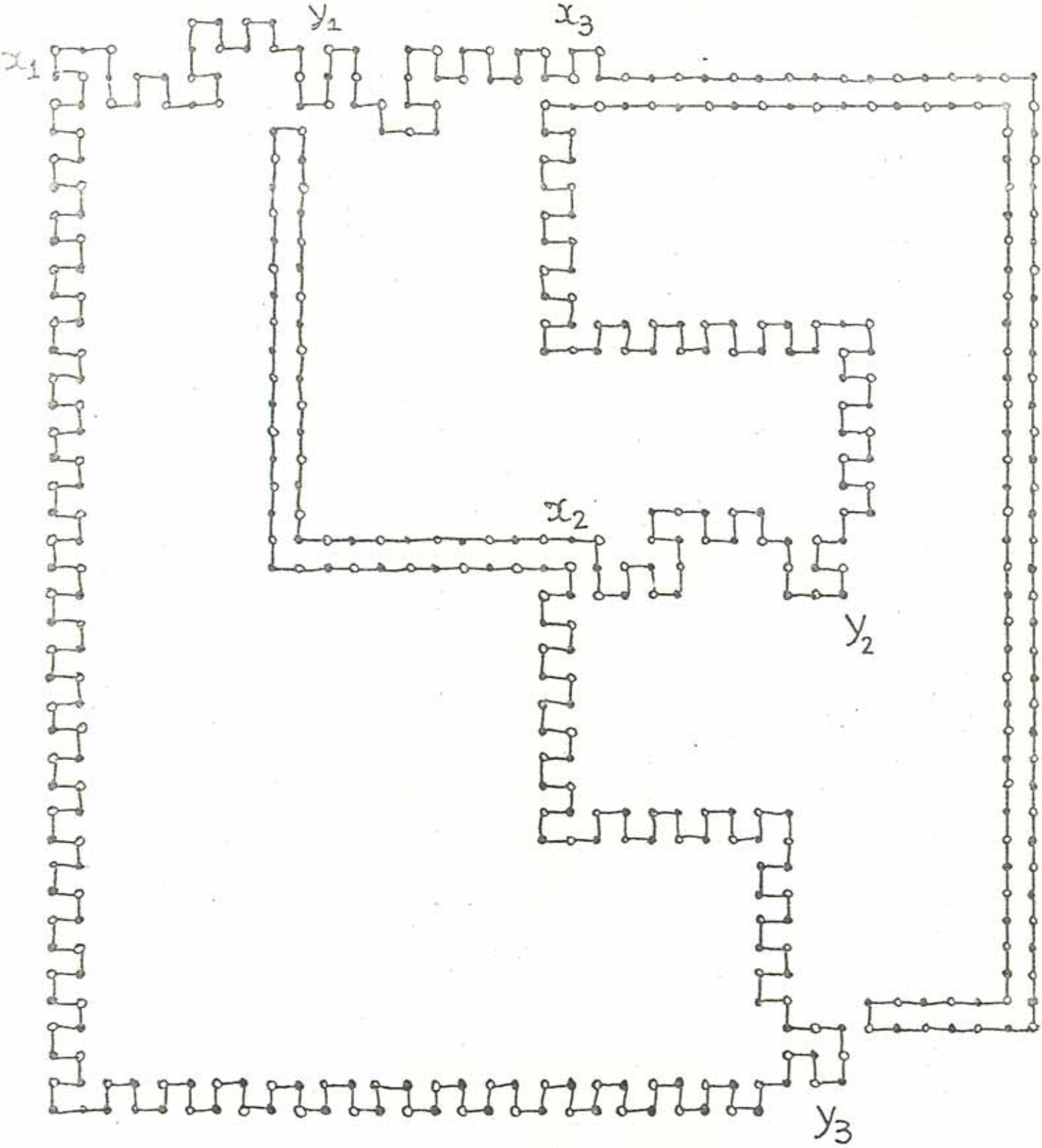


Figure 20

The grid graph G' corresponding to the embedding of Fig. 15, and a Hamilton circuit, correponding to $[x_1, y_1, x_3, y_2, x_2, y_3]$.

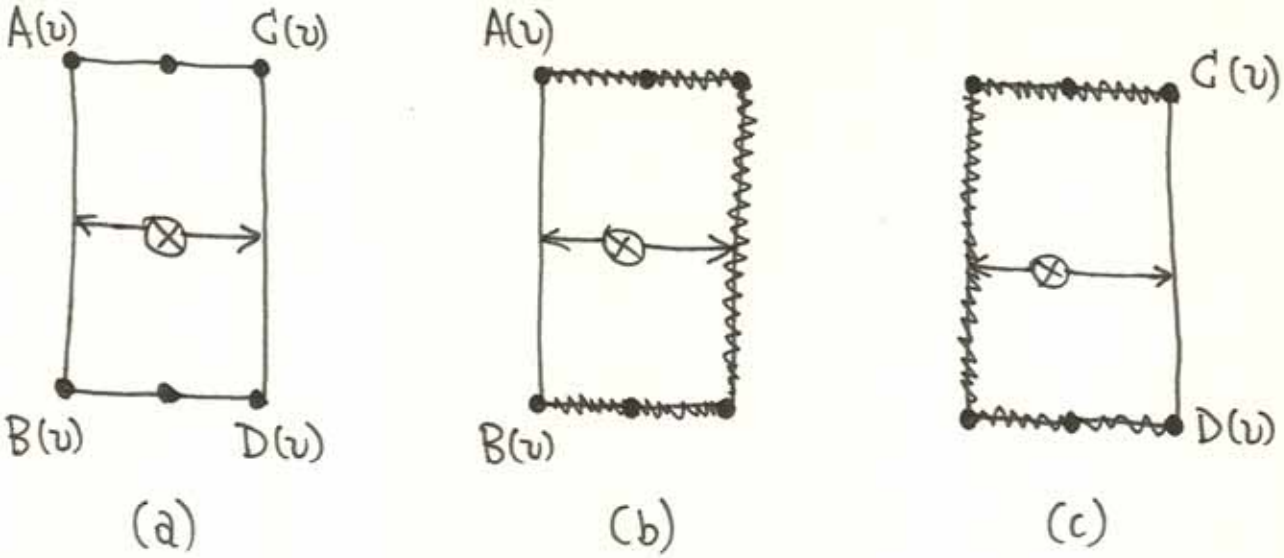


Figure 21

Vertex substitute for the proof of Theorem 7, with the two possible ways it can be traversed.

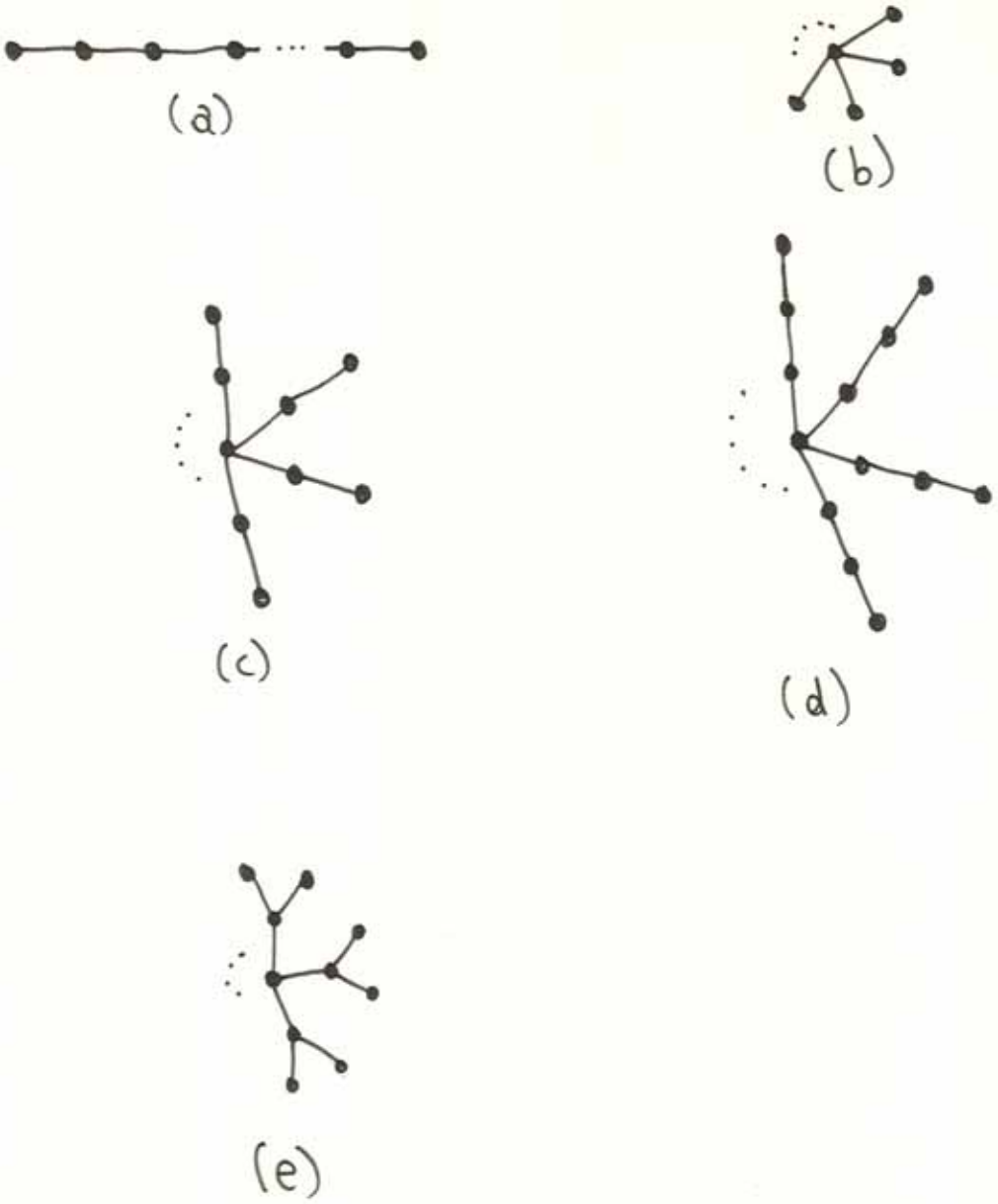


Figure 22

Families of trees.

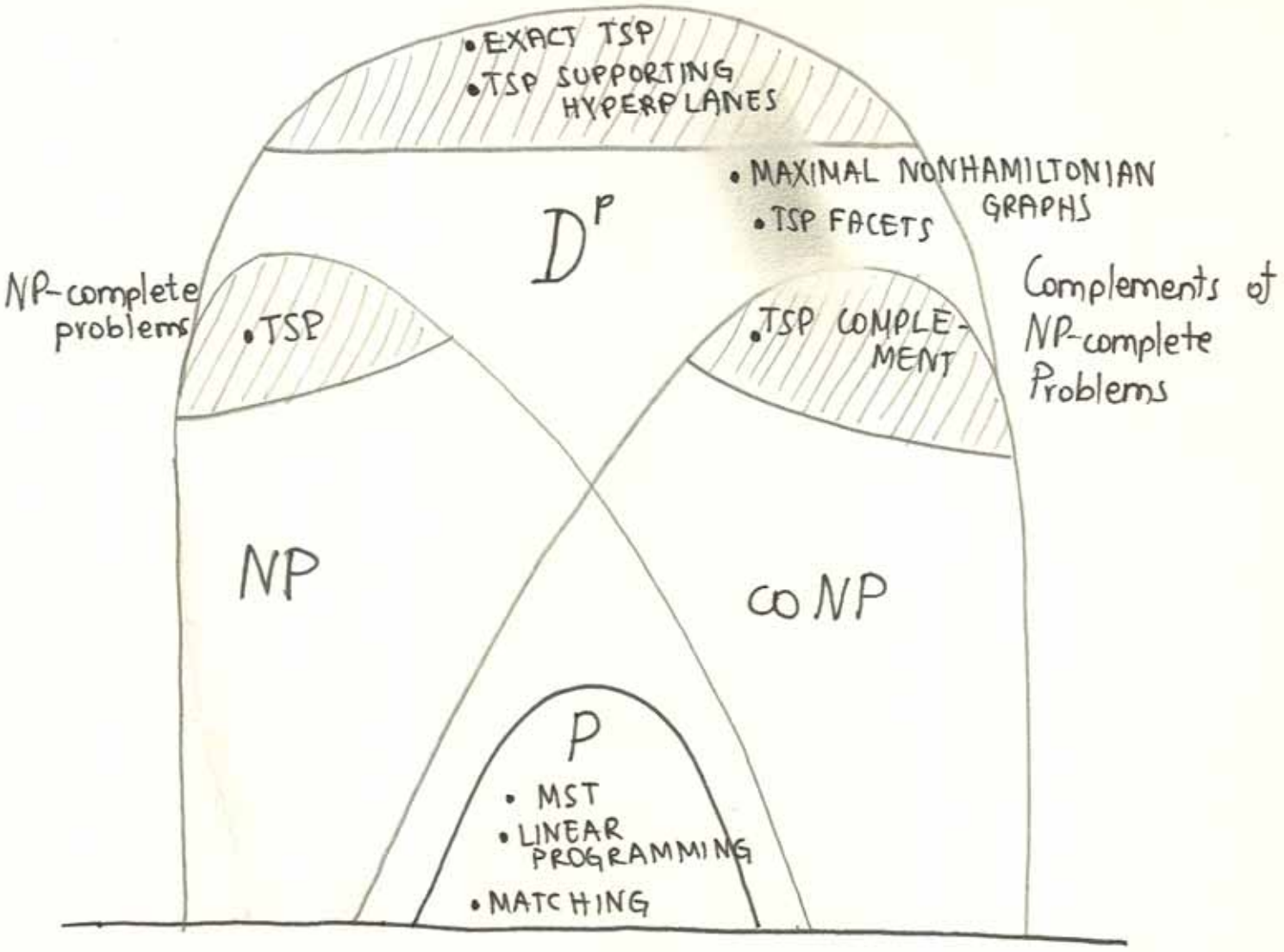


Figure 23

A conjectured topography of P, NP, and their vicinity.