Interfaces

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Tutorial on Computational Complexity

Craig A. Tovey,

To cite this article:
Craig A. Tovey, (2002) Tutorial on Computational Complexity. Interfaces 32(3):30-61. http://dx.doi.org/10.1287/inte.32.3.30.39

Tutorial on Computational Complexity

Craig A. Tovey
School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332
ctovey@isye.gatech.edu

This paper was refereed.

Computational complexity measures how much work is required to solve different problems. It provides a useful classification tool for OR/MS practitioners, especially when tackling discrete deterministic problems. Use it to tell, in advance, whether a problem is easy or hard. Knowing this won't solve your problem, but it will help you to decide what kind of solution method is appropriate. Complexity analysis helps you to understand and deal with hard problems. It can pinpoint the nasty parts of your problem, alert you to a special structure you can take advantage of, and guide you to model more effectively. You will solve your problem better when you know the borders between hard and easy. Locating the difficulty can indicate where to aggregate, decompose, or simplify. To detect and prove computational difficulty, show that a known hard problem from the literature is embedded within your problem. Fix parameters of your problem to arrive at the known hard problem, or use specialization, padding, forcing, or the more difficult gadget proofs. Study contrasting pairs of easy and hard problems to develop your intuitive ability to assess complexity.
(*Analysis of algorithms*: *computational complexity*.)

Computational complexity is the measurement of how much work is required to solve different problems. It provides a useful classification tool for OR/MS practitioners, especially when tackling discrete deterministic problems. Use it to tell, in advance, whether a problem is easy or hard. Knowing this won't solve your problem, but it will help you decide what kind of solution method is appropriate. If the problem is easy, you can probably solve it as a linear program (LP) or network model or with some other canned method. If the problem is hard, finding an exact solution is apt to be costly or impractical, and you will probably have to resort to enumerative methods (which may be slow or useless for large cases) or settle for an approximate solution obtained with heuristics.

Complexity theory can help you to understand and deal with hard problems. It can pinpoint the nasty parts of your problem, alert you to a possible special structure you can take advantage of, and help you model more effectively.

I've written this tutorial for two audiences. For practitioners who want to improve their intuitive ability to assess complexity, I recommend this introduction and §§1, 4, 6, 8, and 10. Those who want more, for example, to be able to use standard references, should read the entire tutorial.

What can you expect to learn? A basic way to solve problems in OR/MS is to have a toolbox of standard well-solved easy problems, such as maximum flow and shortest path. You take your real problem and model it with the right choice of problem from the toolbox. That process classifies your problem as easy and lets you solve it by a standard method. This tutorial teaches how to classify problems in the opposite way. You have a second toolbox, containing basic hard problems. You make the right choice of problem from the toolbox and model it with your real problem. That process classifies your problem as hard. The model in reverse is a *complexity proof*, and it sometimes pinpoints the difficulty in real problems. Pinpointing the difficulty can indicate where to aggregate, decompose, or simplify. If you know where the borders are between hard and easy, you will be better able to deal with your problem.

0092-2102/02/3203/0030$05.00
1526-551X electronic ISSN

The abbreviated reading I've suggested should expand your knowledge of the toolbox of basic hard problems and improve your ability to distinguish hard problems from easy problems. It includes examples of finding the complexity within problems (§4) and of how to use complexity to cope with hard problems (§8). You will need to read the entire tutorial to use the huge toolbox found in the standard references and to insure that you use complexity theory correctly and appropriately.

There is no way around doing complexity proofs if you want to use computational complexity. Therefore, much of this tutorial is more mathematical than most *Interfaces* articles. Fortunately, once you can read the standard references, you can often find a known hard problem similar enough to yours that you can use one of the easier proof techniques, such as specialization or padding. The tutorial also covers gadget proofs, which are more difficult.

Some other complexity classifications (§9) get short shrift because I have not found them very useful. Finally, I discuss how complexity limits our ability to solve problems and what kinds of trade-offs we as a community might consider.

You do not need to know any complexity theory to read this tutorial. It will help if you know basic LP, networks, and integer programming (IP) at the undergraduate or MBA level, since I will draw several examples and analogies from these areas.

I've included two kinds of problems, imitating Knuth (1973) and McConnell (1989). *Questions* are to help you to understand what you are reading. Questions ordinarily won't require pencil and paper; the answers are short and all are given at the end of each section. *Exercises* may require some scribbling and a few minutes of thought. They should help you to develop skill in computational complexity. You should answer all the questions as you read. If you want to develop the ability to recognize complexity or do complexity proofs yourself, do the exercises as well.

1. Complexity and NP-Hardness

This section explains the ideas of time bounds for algorithms, $O(\ )$ notation, and polynomial and exponential time. It introduces the crucial concepts of instances, problems, and realistic cases.

In the heady years following the invention of LP and the simplex method, researchers explored many applications and extensions. One of these was integer programming, which is the same as LP except for the seemingly minor additional requirement that certain variables take on only integer values. Two patterns soon emerged, one good, one bad.

The good news was that many problems could be modeled as IPs that (apparently) could not be modeled as LPs. Problem after problem, from fixed charge to traveling salesman, fell before the onslaught of IP modelers.

The bad news was that all the solution methods proposed turned out to be poor at solving IPs, except small cases. Gomory's cutting-plane algorithm is a quintessential example. It was an extension of the simplex method, just as IP was an extension of LP. Like the simplex method, it could be proved to converge in a finite number of steps. By analogy, one would have expected Gomory's algorithm to solve IPs effectively. And it did, on small instances with, say, fewer than 20 constraints. But on moderate-sized instances, the algorithm frequently bogged down, making many tiny ineffectual cuts and sometimes failing to converge at all because of precision problems.

Researchers tried cutting planes, dynamic programming, branch and bound, and group theoretical methods, but all failed to solve the medium-sized cases.

The theory of computational complexity gives us a magic lens through which we can look at these two patterns and see them, the good and the bad, as a whole.

Integer programming is an NP-hard problem. Like all NP-hard problems, it shares two properties: that it can be used to model a host of important problems, and that no known solution method has proved to consistently and exactly solve large instances efficiently.

Running Time of Algorithms

> Sooner or later any problem will get big enough that you can't solve it. But it's nicer if you make it later (Rosenberg 1993).

How long does it take to find the maximum in a set of numbers? A numerical answer, such as "0.66 seconds," is meaningless, because the length of time depends on the hardware and software and, more important, varies depending on how big the set is. The

answer should be a function of the size of the particular instance to be solved. The function tells how the algorithm run time grows as the input size increases. In this case, suppose the numbers are $A(i)$: $i = 1, \ldots, n$. The obvious algorithm finds the maximum in time proportional to n. We say the algorithm runs in linear time, or is $O(n)$. If instead we wish to sort the numbers in ascending order, the fastest algorithms require time proportional to $n \log n$. We say that sorting is done in $O(n \log n)$ time.

A precise definition of $O(\ )$ time bounds is that an algorithm has time bound $O(f(n))$ if there exist constants N and K such that for every input of size $n \geq N$ the algorithm will not take more than $Kf(n)$ processing time. Notice two points: First, this is a worst-case definition, since the bound applies to all sufficiently large inputs. Some algorithms that are fast in practice but not in theory, such as the simplex method for LP or the Lin-Kernighan heuristic for the traveling-salesman problem, will have misleading time bounds. To get around this difficulty, we might say, for example, that the simplex method is $O(m^3)$ on average or in practice even though the best theoretical bound is exponentially large. Second, what exactly do we mean by size of an input? The precise meaning is the actual length, for example, the number of bits, of the input data. This is the mathematical definition, but it is nonintuitive and awkward to work with. Instead, usually one or two natural parameters describe the size of the particular instance to be solved, and we state an algorithm's time requirements as functions of these parameters. You can usually get away with using these natural parameters instead of input length, because they usually give the right answer.

For example, consider more carefully the time required to find the maximum of n numbers. The natural parameter to use is n. However, the bound $O(n)$ is not technically correct, because comparing two numbers takes more time if the numbers are very large (many digits long). The correct analysis must be performed in terms of the combined input length of the numbers, L, which takes into account the number of digits. There might be a few very large numbers or many small numbers of total length L. Fortunately, you can compare two numbers in time proportional to the length of the shorter. Hence, it takes time $O(L)$ to find the maximum of a list of numbers L bits long, regardless of whether it contains many small numbers or a few large numbers.

To find the maximum of a set of numbers, $O(L)$ is technically correct while $O(n)$ is not. On the other hand, since numbers have limited precision in practice, the more natural $O(n)$ bound is a good surrogate. As a rule of thumb, use the natural parameters without fear, unless the individual numbers in the data (for example, cost coefficients) are very long or very short. If they are very long, your problem may be harder than it looks; if they are very short, your problem may be easier because of dynamic programming. (Question: How much time would we say is required to add n numbers? Exercise: why is it trickier to determine the time required to divide a pair of numbers? Why is $O(1)$ the practical answer?)

The major distinction we make between algorithms is whether they take polynomial time or not. An algorithm requires polynomial time if for some k it has a time bound of $O(n^k)$. Since the heapsort algorithm has a time bound of $O(n \log n)$, it is polynomial with $k = 1.5$. (Question: Would $k = 1.001$ be correct? How about $k = 2$?) Most algorithms that are not polynomial are exponential, requiring more than c^{dn} time in the worst case, for some constants $c > 1$ and $d > 0$. Any exponential function will eventually exceed any polynomial function as n increases. From a theoretical point of view, that is why polynomial time algorithms are preferred to exponential time algorithms. In practice, exponential time algorithms are usually slower than polynomial time algorithms even for quite modest values of n.

In ordinary English usage, the term *problem* can refer to a specific numeric case or to a class of cases having the same form. When employing computational complexity, we always call a specific case an *instance*. We use the term *problem* to mean a class of instances of the same type, for example, the integer programming problem, or the min cost network flow problem. Maximize $c \cdot x$ subject to the constraints $Ax \leq b$; $x \geq 0$, is a problem. Maximize $4x_1 + 3x_2$ subject to the constraints $5x_1 + x_2 \leq 10$; $x_i \geq 0$, is an instance of that problem. (Question: Is Maximize $\Sigma_i x_i$ subject to the constraints $Ax \leq 0$; $x \geq 0$, a problem or an instance?)

The Straight Dope: Theory and Practice

In complexity theory, we make the following sweeping generalizations: If an algorithm runs in polynomial time, it is *fast*; otherwise it is *slow*. Fast is good; slow is bad. A problem that we can solve by a fast algorithm is *easy*; a problem that we can't is *hard*.

These generalizations work very well in practice, on the whole. Most polynomial-time algorithms are fast, and most exponential-time algorithms are useless on large cases in practice. Most easy problems can be solved quickly in practice (not necessarily by a theoretically fast algorithm—the simplex method is a good example); most hard problems cannot be solved quickly in practice if the cases are large.

Examples of easy problems include LP, minimum cost network flow, matching, minimum spanning tree, and sorting. Examples of hard problems include IP, traveling salesman, and job-shop scheduling. The class of easy problems is denoted by P in Figure 1. In Figure 1, a problem is represented by a single point.

Most uses of complexity in OR/MS involve a theoretically defined class of problems, the NP-complete problems. Theoreticians (like me) worry about whether the NP-complete problems really are hard. Indeed, the major unsolved question in theoretical computer science is to prove that no fast algorithm exists for an NP-complete problem. Practitioners (like me) don't worry about this question, because with the algorithmic technology available at present, the NP-complete problems are hard.

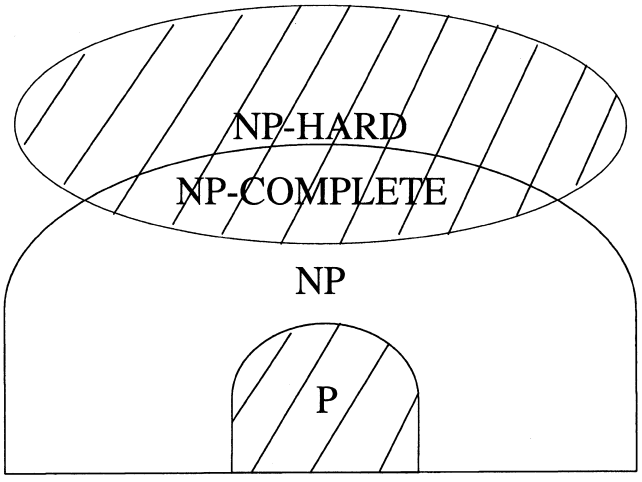

Figure 1: *P* is the set of easy problems. The NP-hard problems include the NP-complete problems and many hard problems that are not in NP. Almost all real problems are either easy or NP-hard.

Figure 1 depicts the class P of easy problems, the class of NP-complete problems, and the class of NP-hard problems, which contains the NP-complete class. You do not need to know the definitions to read and use this tutorial, but I've included them in §9.

This tutorial focuses on recognizing NP-hardness rather than NP-completeness for two reasons. First, it is just as useful in practice to prove membership in the larger class, NP-hard, as in the smaller class, NP-complete, and the NP-hard class is simpler to work with. Second, most problems encountered in practice that cannot be solved by any known fast algorithm are NP-hard. Few naturally occurring problems lie between the set of easy problems P and the set of NP-hard problems, in the unshaded region of Figure 1.

Thousands of problems are already known to be NP-hard. If you want to be sure your problem is NP-hard, you must either determine that someone has already proved it to be hard or use a known NP-hard problem to prove that your problem is hard as well.

If a problem is hard, you may be able to solve small instances readily, but any exact solution method will be in essence enumerative and will require exorbitant amounts of time on large enough instances in the worst case. Classifying a problem as hard means you cannot expect to solve exactly every possible case quickly but not that you should despair of solving it for practical purposes.

As problem solvers, we do not give up on a problem because it is hard. Our task does not end with an accurate diagnosis. I used to encounter a few computer science (CS) students each year who had been taught to stop work on a problem if they discovered it was hard. For years this has been an amusing contrast: many CS papers end with NP-hardness, while many OR papers begin with NP-hardness. That is, the authors of a typical CS paper define a problem that is simple enough to solve quickly, add generalizations that still permit quick solution, then end by showing that the next level of generality makes the problem NP-hard. Unfortunately, they are ending the story just as it is starting to become realistic. In a typical OR paper, the authors describe a model (presumably) based on a

real problem and show that it is NP-hard: this justifies the heuristic or math-programming solution methods that form the core of the paper. The CS community instead has focused on developing sophisticated and powerful sets of data structures and algorithms to solve fundamental, recurring problems with surprising efficiency. Recently they have also developed many fast algorithms that find approximate solutions to hard problems.

There are many reasons why a hard problem might still be solved in practice. NP-hard means only that it takes a long time to solve exactly all cases of sufficiently large size. This leaves us with many loopholes. Most cases might be solvable quickly, or all could be solvable quickly to within a few percent of optimality, or the cases you will have to solve might be small enough to be manageable.

In practice, you will usually have to deal with something more general than a specific instance but less general than a problem. We can call the set of instances of a problem that you expect to deal with *realistic cases*. The realistic cases are usually a family of related instances, often sharing structural or other properties. You don't know in advance exactly what is in the family. Identifying and taking advantage of features of the realistic cases is usually crucial to successful problem solving. Suppose, for example, that the realistic cases can be decomposed into a series of moderate-sized instances. Then you may do perfectly well using, say, canned software for nonlinear or mixed-integer programming.

(Question: Why do we usually describe a problem rather than the realistic cases?)

Answers:

—$O(n)$, although $O(L)$ is technically correct.

—Yes, yes. Any $k > 1$ will do.

—A problem.

—The realistic cases take too long to explain and aren't fully known.

2. Yes-No Form

This section shows how to convert problems into the standard instance, yes-no question format used in complexity analysis. The conversion is usually achieved by including a threshhold value in the input, but some care may be needed so as not to alter the problem's complexity. When we solve a problem, we almost always produce an explicit solution, for example, a schedule or a design. We laugh at the story of the mathematician who, seeing a fire in his wastebasket, points to the water faucet and a bucket, and exclaims, "A solution exists!," letting the house burn down. When I optimize the inventory policy for a corrugated-paper plant to minimize trim, I don't tell the manager the problem is solved and average trim can be reduced to 1.82 inches without providing the policy that achieves this value.

To do complexity analysis, we must suppress our inclination to provide solutions. The first skill you must acquire is to convert optimization problems into yes-no form.

In some applications, problems naturally occur as feasibility or satisfice questions. These include many classroom- and other facility-scheduling problems. Converting these to yes-no form is simple. For example, if we are scheduling jobs with various processing times and deadlines on a machine, it may be appropriate to seek a schedule with no late jobs. Following Garey and Johnson's (1979) popular format, we define this scheduling problem in two parts. The first part describes the form of the input data. The second part is a yes-no question. The format is as follows:

One-Machine Deadline Scheduling

Instance: Processing times t_i and deadlines $d_i : i = 1, \ldots, n$.

Question: Is there a set of start times $s_i \geq 0 : i = 1,\ldots,n$ that is feasible, $s_j \notin [s_i, s_i + t_i)\ \forall i \neq j$, and that meets all deadlines, $s_i + t_i \leq d_i\ \forall i$?

Usually the conversion is not that simple. Consider the optimization version of LP, which is to find x to maximize $c \cdot x$ subject to the constraints $Ax \leq b$; $x \geq 0$. To make a yes-no version, we introduce a threshhold value v as part of the input data, and ask whether there is a feasible x with objective value v or better. For threshhold LP, we write the following:

Instance: m by n matrix A; m-vector b; n-vector c; scalar v.

Question: Does there exist x such that $c \cdot x \geq v$ and $Ax \leq b$?

(Question: Consider the optimization scheduling

problem of minimizing the number of late jobs on a machine. Write a yes-no version in the standard format.) In general, you can convert an optimization problem to yes-no form by including a threshhold value with the input data.

We analyze the yes-no version of a problem to classify its optimization version. Therefore, hard problems should stay hard and easy problems should stay easy when converted to yes-no form. That is, the yes-no version should be hard if and only if the optimization version is hard. For example, the yes-no version of LP given above does not mislead us about the complexity of the optimization version: both are easy.

If you correctly convert a problem to yes-no form, the yes-no version will never be more difficult to solve than the original version. (Question: What erroneous conclusion might we reach from an incorrect conversion?) The converse property usually holds as well, but it does not need to be established in an NP-hardness proof.

Often it is possible to restrict a yes-no threshhold problem by fixing the threshhold value, without affecting the complexity of the problem. The two-machine line-balancing problem provides a good example. The problem is to divide jobs with known processing times as evenly as possible between two machines. This is the threshhold version:

Two-Machine Line Balancing
Instance: A set of processing times $t_1, \ldots, t_n$, and a number v.
Question: Can the indices $i = 1, \ldots n$ be partitioned into two sets, K and J, such that $|\Sigma_{i\in J} t_i - \Sigma_{i\in K} t_i| \leq v$?

A restricted version of the threshhold problem occurs when we require perfect balance between the machines, that is, $v = 0$. In standard format, we would write the following:

Two-Partition
Instance: A set of processing times $t_1, \ldots, t_n$.
Question: Can the indices $i = 1, \ldots n$ be partitioned into two sets, K and J, such that $\Sigma_{i\in J} t_i = \Sigma_{i\in K} t_i$?

It turns out that this more narrowly defined problem is just as hard as the two-machine-line-balancing problem. (Question: Why couldn't it be harder?) Since it is simpler to state and more restricted, the two-partition is often better to work with.

On the other hand, an extreme restriction can be much easier than the threshhold version. For example, one of the classic NP-hard problems on graphs, *independent set*, seeks a maximum-size subset of vertices, no two of which are connected by an edge. (Exercise: State the threshhold yes-no version of this problem in standard format.) What happens if we fix the threshhold at an extreme value? We get the following problem:
Instance: A graph $G = (V, E)$.
Question: Does G contain an independent set of cardinality $|V|$, that is, does there exist $S \subseteq V$; $|S| = |V|$; $(i, j) \notin E\ \forall i \in S, j \in S$?

In other words, does the given graph have no edges? By fixing the threshhold at an extreme value, we've substituted a very easy problem for a hard one. We could be misled into thinking that independent set was easy.

Often it is not simple to determine whether or not you can fix a threshhold without altering the problem's complexity. You have to figure out what makes the problem hard. A problem may be hard because it is difficult to get everything to balance out or fit together perfectly. Two-partition is an example. But in other problems, it is easy to find out if you can get everything perfect; the hard part is to minimize the ill effects when you can't get everything to fit. Independent set is an example.

When you convert a problem to yes-no form, imagine fixing the threshhold to an extreme value (usually 0). Focus on the restricted problem if it seems as computationally hard as the original. It will be simpler to work with. Otherwise, focus on a different range of threshhold values, but keep the easy restricted case in mind as a source of heuristic solution ideas. Either way, you will gain useful insight into your problem. If you trust your judgment, all you risk is spending a few hours thinking that your problem is easier than it actually is.

Sometimes you can remove the objective function without affecting the complexity of the problem. You should trust your judgment here too. Integer-programming feasibility is just as hard as threshhold integer programming. (Question: State each of these problems in standard format, and explain why the former is just as hard as the latter.) LP optimization is just

as easy as LP feasibility (combine the primal, dual, and strong duality constraints). (Question: Think of a problem that changes from hard to easy if the objective is removed.) (Exercise: For each yes-no problem defined in this section, write a specific instance for which the answer is yes and an instance for which the answer is no.) Answers:

—Besides the threshhold value v, the input contains the matrix A and the vectors c and b.

—We might conclude that the optimization problem is hard, when in fact it is easy.

—Any algorithm that solved two-machine-line-balancing would also solve two-partition.

—The threshold problem is as follows:

Instance: m by n matrix A, m-vector b and n-vector c, scalar v.

Question: Is there an integer n-vector x such that $Ax \leq b$ and $c \cdot x \geq v$?

—The feasibility problem is as follows:

Instance: m by n matrix A, m-vector b and n-vector c.

Question: Is there an integer n-vector x such that $Ax \leq b$?

Any threshhold instance can be turned into an equivalent feasibility instance by including $c \cdot x \geq v$ in the feasibility constraints.

—Knapsack, clique maximization, graph coloring, line balancing, and many others. (See §4 for problem definitions.)

3. The Nuts and Bolts of NP-Hardness Proofs

This section introduces NP-hardness proofs. The main idea is to model a known NP-hard problem as your problem. Figure 2 illustrates a particular proof, and Figure 3 shows the general form of a proof. The proof in Figure 2 actually arose during a practical application. We were sequencing placements of electronic components on a circuit board to minimize the production cycle time per board. The machine took at least 0.25 seconds to place a component on the board; if the previous component was far away on the board or was a type of part located far away on the machine, it could take much longer. The machine had to begin and end at a fixed location (a *fiducial*), so our problem was a traveling-salesman problem (TSP), a famous NP-hard problem.

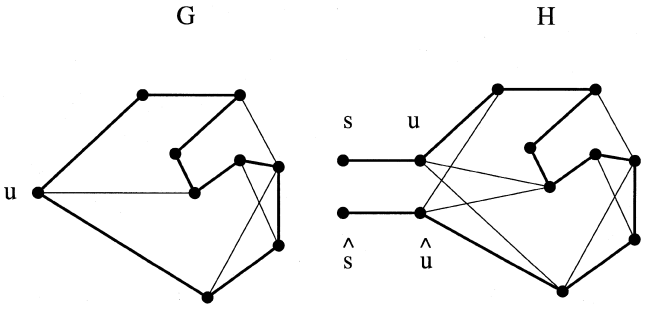

Figure 2: $\mathcal{T}$ transforms *G* into a related graph *H* by splitting and extending a vertex *u*. *G* has a Hamiltonian cycle if and only if *H* has a Hamiltonian path starting at *s*.

In our TSP, the vertices represent placements and the cost of edge from i to j is the time required to place j if the previous placement is i. So many edges cost 0.25 that for practical purposes we could slightly simplify our problem: discard the arcs costing more than 0.25 and seek a Hamiltonian cycle in the simplified graph. (A cycle or path in a graph is Hamiltonian if it visits each vertex exactly once.) (Question: How could we arrive at this Hamiltonian cycle model as a yes-no version of our TSP?)

The Hamiltonian cycle problem is another famous NP-hard problem, and we used heuristic methods to solve it. Then the problem changed. We found that some of the circuit boards had parts of varying sizes. The overall speed of the machine depended on the largest part it had picked up so far. We decided to sort the parts into size classes and solve separate sequencing problems for each class in turn. However, these

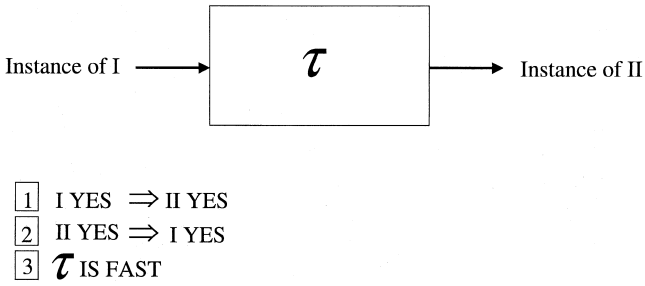

Figure 3: Show your problem (II) is hard by constructing $\mathcal{T}$, which transforms instances of a known hard problem (I) to instances of yours. $\mathcal{T}$ must satisfy the three properties listed.

new sequencing problems were not Hamiltonian cycle problems! Consider the smallest size parts: the fiducial forces a fixed starting point, and we must visit each vertex (make each placement) once, but we no longer have to return to our starting point.

Would this extra bit of freedom make the problem any easier? Should we still have been using heuristics to solve it? We demonstrate that the new problem is NP-hard. This will be our first example of an NP-hardness proof.

We already know that the following problem is hard.

Ham Cycle
Instance: A graph G.
Question: Does G contain a Hamiltonian cycle?

We would like to show that our new problem is hard. In words, our problem is to find a Hamiltonian path that starts at a particular vertex. The starting point is referred to as distinguished because it is known in advance. We precisely define our new yes-no problem:

s-Ham Path
Instance: A graph $H = (V, E)$ with distinguished vertex $s \in V$.
Question: Does H contain a Hamiltonian path beginning at s?

Suppose s-Ham Path were easy. Then software to solve it quickly would exist. We use this imaginary software to solve Ham cycle quickly, giving a contradiction. (Question: What will be the contradiction, and what will we be entitled to conclude?) The way to use our imaginary software is to build a fast front end:
Input: A graph G with vertex set U and edge set E.
Output: A graph H with distinguished vertex s.
(Question: What are the input and output instances of?) Our imaginary software would work as follows: given input graph G, select any vertex u of U and make a copy of it, $\hat{u}$, connected to the same vertices u is connected to. Create a new vertex s connected only to u and likewise a new vertex $\hat{s}$ connected only to $\hat{u}$. The resulting graph, H, is depicted in Figure 2. The front end is fast, with most of its time spent in copying G.

If G has a Hamiltonian cycle, we can think of it as starting and ending at u. This cycle corresponds to a path in H starting at u, ending at $\hat{u}$, and visiting all the vertices in H except s and $\hat{s}$. Extend this path at the beginning and end to reach s and $\hat{s}$, respectively. This is a Hamiltonian path in H starting at s. Therefore, if we input a graph G that has a Hamiltonian cycle, our front end will output a graph H that has a Hamiltonian path starting at s, and the imaginary software for s-Ham path, coupled with our front end, will correctly answer yes.

We also need to verify that, if G has no Hamiltonian cycle, our software will correctly answer no. We prove this by the contrapositive. (Question: State the contrapositive.) If our software answers yes then H contains a Hamiltonian path starting at s. Since the path is Hamiltonian, it must visit $\hat{s}$. By construction, only one edge is incident to $\hat{s}$. Therefore, the path has to visit $\hat{s}$ last. Truncating the first and last edges in the path gives a path from u to $\hat{u}$, which corresponds to a Hamiltonian cycle in G.

We've shown that our software answers yes if and only if G contains a Hamiltonian cycle. The front end is fast, the imaginary software is fast, and thus we have fast software to solve the Ham cycle problem. By contradiction, s-Ham path is NP-hard.

We have completed our first NP-hardness proof. (Suggested Exercise: Without referring to the above, write down the proof showing s-Ham path is NP-hard. This is not a rote-memory exercise; it is a standard method to master proof techniques. Reconstructing a proof you've read, in detail, reinforces your understanding and makes you much more able to do proofs on your own.) (Exercise: Prove s-Ham path is NP-hard using a modified front end as follows. Make the copy $\hat{u}$ of u as above, and make a new vertex $\hat{s}$ connected only to $\hat{u}$ as above, but do not make a new vertex s. Instead, change the label u to s.)

The General Form of an NP-Hardness Proof
Figure 3 shows the basic outline of an NP-hardness proof. You must follow this structure to show that your problem is NP-hard. Select a known hard problem and model it as a case of your problem. This is counterintuitive because it is the opposite of what you might do to solve your problem, namely, to model your problem as an IP or as a case of some other better-known problem.

You must concoct a transformation, $\mathcal{T}$, essentially a computer program like the front end in our s-Ham path proof, which has

Input: Instance of I, a known NP-hard problem.
Output: Instance of II, your problem.

Your transformation $\mathcal{T}$ must satisfy three properties:
(1) Yes-to-Yes: $\mathcal{T}$ maps yes instances of I to yes instances of II.
(2) Yes-from-Yes: $\mathcal{T}$ maps no instances of I to no instances of II.
(3) Fast: $\mathcal{T}$ is fast.

Finding the right $\mathcal{T}$ often requires a flash of insight because it has to satisfy all three properties. NP-hardness proofs are more akin to integration in calculus than to differentiation: they require creative recognition of patterns rather than following cookbook procedures. (Question: Suppose you know IP is NP-hard and you wish to prove two-machine line balancing is hard. For this specific proof, describe precisely the input form, the output form, and the three required properties of the transformation $\mathcal{T}$.)

One could state properties (1) and (2) more succinctly. $\mathcal{T}$ must output a yes instance of II if and only if it is input a yes instance of I. I have separated this into two properties because, in my experience, the most common mistake is not to satisfy property (2). Always verify property (2) by the contrapositive: prove that if the output of $\mathcal{T}$ is a yes instance of II, the input must have been a yes instance of I. I call property (2) Yes-from-Yes instead of No-to-No because it is always verified by the contrapositive.

Occasionally a proof fails because it violates property (3). Property (3) means that $\mathcal{T}$ runs in polynomial time in the length of the input instance.

Beginners are often confused about the direction of the transformation $\mathcal{T}$. They want to input an instance of their problem, II, and output an instance of some known hard problem, I. That is the natural direction of a transformation if you are trying to solve your problem. You would convert your problem into some better known mathematical form, such as IP or TSP. Complexity proofs require you to do the opposite, which at first seems counterintuitive.

The explanation is as follows. Suppose I is IP, a known hard problem. If you had fast software to solve II, you could put in $\mathcal{T}$ as a front end and have a fast IP solver! So it must be at least as difficult to solve II as to solve IP. (Question: Why does it help to know many NP-hard problems when trying to prove a new problem is NP-hard?)
Answers:

—We find it as a threshhold question with a particular fixed value. Does there exist a Hamiltonian cycle with total cost $\leq n(0.25)$?

—Being able to solve Ham cycle quickly contradicts the fact that Ham cycle is hard. We are entitled to conclude that s-Ham path is hard.

—The input is an instance of Ham cycle; the output is an instance of s-Ham path.

—The contrapositive is this: if our software answers yes, then G has a Hamiltonian cycle.

—Input: an instance of IP consisting of m by n matrix A and n-vector b. Output: an instance of two-machine line balancing consisting of k integers $s_1, \ldots, s_k$, and an integer v. The three required properties: (1) If there exists integer x such that $Ax \leq b$, then the s_i can be partitioned into two sets, so that the sum of each set is $\leq v$. (2) If the s_i can be partitioned into two sets, so that the sum of the s_i in each set is $\leq v$, then there exists integer x such that $Ax \leq b$. (3) $\mathcal{T}$ runs in time polynomial in the size of A and b.

—$\mathcal{T}$ runs in time polynomial in the length of the input instance.

—Yes, it is consistent.

—Knowing more hard problems gives you more choices of what to model as your problem. The more choices, the more likely you are to find one that you can do readily.

4. Spotting Complexity

This section will help you get better at spotting complexity. I begin with some general tips. Then I invite you to spend some time studying a collection of contrasting hard and easy problems. I don't know of a more effective way to develop your intuitive ability to recognize what is hard and what is easy. Reading this will also give you important practice understanding the concise style of Garey and Johnson (1979) and other references.

Here are some things that tend to make problems hard: dividing up work or resources perfectly evenly; making sequencing decisions that depend not just on where you are but also on where you've been; splitting

a set of objects into subsets, where each subset must satisfy a constraint; finding a largest substructure that satisfies some property; maximizing or minimizing intersections or unions of sets; interactions among three or more objects or classes of constraints.

If you limit a threshhold value, you produce a restricted version of a problem. When a problem involves a bunch of interlocking decisions, and you are trying to minimize the number of bad events, such as late jobs, mismatches, or items out of place, consider the restricted problem that tries to eliminate all bad events. Frequently the restricted problem will be hard if the original problem is hard. On the other hand, if you are trying to minimize a sum of penalties, such a restriction is more likely to alter the complexity of the problem. For example, in one-machine scheduling (see §2), it does no harm to restrict the problem of minimizing the number of late jobs to the problem of getting all jobs done on time. Both problems are easy. On the other hand, minimizing the total tardiness (sum of amounts by which late jobs are late) is hard. The problem complexity would change if it were restricted to achieving zero tardiness.

How do I evaluate the complexity of a problem? I don't follow a rigid method, but this is how I tend to proceed. First, I get a clear idea of the situation. I state a precise formulation of the problem, usually in words, sometimes with a picture. Second, I strip away the story part of the problem and abstract it down to a math problem: I turn people into vertices, processing times into numbers, retrieval orders into subsets, and so forth. Third, I visualize several different cases of the problem. I tinker with a few small instances by hand. What is the problem like if all the weights are equal or if all subsets are the same size or if there are only two subsets? If the problem has several complicating features, such as deadlines and precedence constraints, I take turns eliminating them. Often I immediately recognize the simpler case of the problem as NP-hard. It is important to know where the complexity of your problem resides. Fourth, after I've visualized some cases, I usually get a feeling that the problem is easy or that it is hard. If the former, I try to solve it with standard methods, such as greed, and to model it as an LP, network, or other known easy problem. If the latter, I focus on the simplest case that seems hard. Why does it seem hard? What does it remind me of? If it is simple to state, I pore through the references and try to find it or something similar. If this doesn't work, I either take my simplest case back to the third step or I roll up my sleeves and try a gadget proof.

Following is a list of NP-hard problems. In each entry, I first describe a hard problem and then describe an easy problem to contrast with it. Visualize each problem as you read. This is the best way I know for you to develop your own intuitive sense of complexity. Compare problems in the same entry; observe the alteration that makes an easy problem hard. Later, as you explore a new problem, you may find that it seems hard, or easy. What other problem does it remind you of? You may be on your way to resolving its complexity. (Exercises: For each problem in the following collection, think of an instance whose answer is yes and an instance whose answer is no. I've written some of the problem descriptions in the less compact Instance-Question format, to give you practice for reading the references. Some of the other problems will give you practice converting to yes-no form.)

Integer programming (IP): Maximize $c \cdot x$ subject to $Ax \leq b$, x integer. *Contrast*: LP is easy, as is IP when the constraint matrix is totally unimodular or has one of its two dimensions fixed. Its dimensions are the number of variables and the total number of constraints, including nonnegativity.

Two-machine weighted flow-time minimization: Given two parallel processors and a set of jobs with individual processing times and weights, find a schedule to minimize the weighted sum of completion times. *Contrast*: If all weights are equal or all times are equal, the problem is easy even for more than two processors (run shorter or weightier jobs first).

Steiner tree: Given a graph $G = (V, E)$, edge lengths, and a subset $S \subseteq V$ of vertices, find a tree in G of minimum total length that contains S. (The tree may contain vertices in $V - S$; try connecting the vertices of a square in the plane.) *Contrast*: The minimum spanning tree problem, in which $S = V$, is easy.

Maximum clique:

Instance: Graph $G = (V, E)$ and integer k.

Question: Does G contain a clique of size k, that is, does there exist $C \subset V$ such that $|C| = k$ and for all $i \in C$, $j \in C$, $i \neq j$, the edge $(i, j) \in E$? This is the same as

seeking a maximum independent set (no pair of vertices connected) in the complementary graph. *Contrast*: Maximum clique is easy on planar graphs (graphs that can be drawn in the plane without any edges crossing), since planar graphs cannot contain cliques of size 5. Finding a maximum independent set in a planar graph is hard.

Approximate solution of maximum clique: Given a graph G, it is hard to find a clique that is at least $1/100$ the size of the largest clique in G. The value $1/100$ could be set to any $1 \geq \in > 0$ and the problem stays hard. (Note: this particular problem cannot be phrased in yes-no form (Papadimitriou 1994).) *Contrast*: It is easy to find a Steiner tree whose length is within 1.5 times the minimum possible length.

3-matching (3DM): Also called three-dimensional or tripartite matching, this is the matchmaking problem if there are three sexes.

Instance: A base set of three distinct sets, X, Y, and Z, with $n = |X| = |Y| = |Z|$, and a set $S \subseteq X \times Y \times Z$ of acceptable triples.

Question: Is there a collection of n disjoint triples from S?

Contrast: When there are two sexes, this is easy. (Question: Why?)

Tip for reading: In the standard reference on complexity by Garey and Johnson (1979), as well as in most other texts, problems are defined compactly. Get used to puzzling over a definition for a moment before you understand the problem. For example, the three-matching problem asks if we can match up all $3n$ creatures into n triples. But the definition asks only if we can find n disjoint triples. It takes a moment to see that every creature will end up in a triple, as there are only $3n$ altogether.

Knapsack: This is essentially IP optimization with a single functional constraint. Given nonnegative vectors c and w and scalars v and K, does there exist a 0–1 vector x such that $w \cdot x \leq K$ and $c \cdot x \geq v$? *Contrast*: The knapsack problem is theoretically hard only when the coefficients occur to great precision. Otherwise it is solved fast with dynamic programming.

Partition: Given a set of n numbers $x_1, \ldots, x_n$, can the indices $i = 1, \ldots n$ be partitioned into two sets, K and J, such that $\Sigma_{i \in J} x_i = \Sigma_{i \in K} x_i$?

Bin packing: We are given a set of numbers $0 \leq x_i \leq 1$, and an unlimited number of bins. The sum of the numbers placed in each bin may not exceed 1. The problem is to minimize the number of bins used to hold all the numbers. (Question: State bin packing in instance-question format.) *Contrast*: There are fast approximate solutions to the same problem.

3-partition: This is very similar to bin packing. Given a set of $3n$ numbers $1/4 < x_i < 1/2$, with total sum n, can they be partitioned into n subsets (of three numbers each) with subset sum 1? *Contrast*: It is easy to partition $2n$ numbers into n subsets of two each.

3-SAT (three-satisfiability): This is the canonical NP-hard problem. It is similar to 0–1 IP with 0–1 coefficients and only three nonzero coefficients per constraint. We are given a set of Boolean (true-false) variables, $X_1, \ldots, X_n$. A *literal* is a variable X_i or its complement $\bar{X}_i$. We are also given a set of clauses C_j, in which each clause contains three literals. The question is, can each variable be set to true or false so that each clause is *satisfied*, that is, each clause contains at least one true literal?

Several important variations on 3-SAT are also hard. In not-all-equal 3-SAT, NAE-3-SAT, the instance is defined the same as in 3-SAT, but the question is, can the variables be set so that each clause contains at least one true and one false literal? In Exact-3-SAT, the question is, can the variables be set so that each clause contains exactly one true literal? *Contrast*: 3,3-SAT, the same problem as 3-SAT except that no variable appears in more than three clauses, is easy (Tovey 1984). 2-SAT, the same as 3-SAT except that every clause contains two literals, is easy, too. However, Max 2-SAT, which maximizes the number of satisfied clauses, is hard. It is easy to set the variables in 3-SAT so that the number of satisfied clauses is at least $7/8$ the maximum possible; it is hard to satisfy more than $7/8$ the maximum (Karloff and Zwick 1997, Håstad 1997).

Ham cycle, Ham path: Given a graph, find a cycle (respectively path) that visits each vertex exactly once. Even for the cases in which the graph is a grid graph (could be cut out of square mesh), these problems are hard. You should be able to see that if Ham cycle is hard, then the traveling-salesman problem is hard even when all costs are 0 or 1. Also, since Ham path is hard, so is finding a longest (respectively shortest) path in a graph whose distances are all 0 or 1 (respectively

-1). *Contrast*: Finding the shortest path in a graph with nonnegative costs is easy. A traversal in a graph may revisit vertices and edges. Finding a shortest traversal that visits all the edges (respectively vertices) in a graph is easy (respectively hard).

Max cut: Given a graph, partition the vertices into two sets S, T to maximize the number of edges crossing between the sets. *Contrast*: Min cut is easy; it is the dual of maximum flow where we take all edge capacities $= 1$. Max bisection cut is also hard. It is the same as Max cut, except we require $|S| = |T|$. Min bisection cut is hard, too. This seems surprising until we see that it is the same problem as Max bisection cut on the complementary graph.

Minimum feedback arc set: Given a directed graph G, arrange the vertices in a line so that as few arcs point backwards as possible. (Exercise: Show that this is equivalent to finding a minimum size subset S of arcs, with the property that every directed cycle in G contains at least one arc in S. The related problem of finding a subset of vertices is also hard.) *Contrast*: It is easy to determine whether G has no directed cycles. (Question: How is this easy problem related to the hard one?)

Graph 3-coloring: Given a graph, partition the vertices into three sets such that no two vertices connected by an edge are in the same set. This problem is hard even if the graph is planar. *Contrast*: 2-coloring a graph is easy. 4-coloring a planar graph is easy, since the answer is always yes. But 4-coloring a graph in general is hard.

Shortest path with obstacles: Finding the shortest path between two points in three dimensions that avoids a collection of polyhedral obstacles is NP-hard (Canny and Reif 1987). *Contrast*: The two-dimensional problem is easy (Latombe 1991).

Answers:

—It is a simple kind of assignment problem, also called bipartite matching.

—Bin-packing instance: numbers $x_1, \ldots, x_n$ and integer B. Question: Can the x_i be partitioned into B subsets each with sum ≤ 1?

—Set the threshhold value of minimum feedback arc set to zero to get the easy problem.

5. Illustrations of Common Pitfalls

One common pitfall is to make $\mathcal{T}$ a simple but lengthy procedure, violating property (3)—$\mathcal{T}$ is fast. You can often detect this error by noticing that $\mathcal{T}$ produces exponentially long instances of II. For example, here is a very easy problem:

Find zero

Instance: A list L of nonnegative integers.

Question: Does L contain a 0?

Let us construct an invalid transformation from s-Ham path to find zero (Figure 4). For each permutation of the vertices $V - s$, we start at s, traverse G, and record on a list the number of mistakes, or missing edges. Obviously G contains a Hamiltonian path from s if and only if we record a 0. The error is that L is not polynomially long in the size of G. Precisely, if G has n vertices, then it has $O(n^2)$ edges and the input to $\mathcal{T}$ is $O(n^2)$. But the list L contains $(n-1)!$ integers, and $(n-1)!$ grows faster than any polynomial function of n^2. If $\mathcal{T}$ produces more than a polynomial length output, it cannot possibly run in polynomial time, violating property (3)—$\mathcal{T}$ is fast.

Although this mistake seems too obvious to make, people make it surprisingly often. You are most likely to make this mistake when transforming from a problem involving numbers. A number does not have to be very long (for example, 30 binary characters) to be very big ($>10^9$).

A related error is to make $\mathcal{T}$ solve the instance of problem I in some enumerative fashion. I've seen this error in print. Some years ago, a well-known applied mathematician claimed to have a new solution method for LP. His algorithm amounted to enumerating the extreme points of the polyhedron, sorting them by objective value, and selecting the maximum (Brockett 1991). (Question: How could you construct a valid transformation from LP feasibility to find zero?)

Suppose s-Ham path is a known NP-hard problem, and we wish to use it to prove Ham cycle is NP-hard. This is the opposite of the first transformation described earlier in §3. Now we must take as input an instance H of s-Ham path, and transform it into an instance G of Ham cycle. I will use this situation to illustrate two of the most common pitfalls.

Many beginners take the graph H and connect s by new edges to every other node of H to make G. They reason that they can extend a Hamiltonian path in H starting at s to form a cycle in G simply by returning to s. They further reason that a Hamiltonian cycle in G

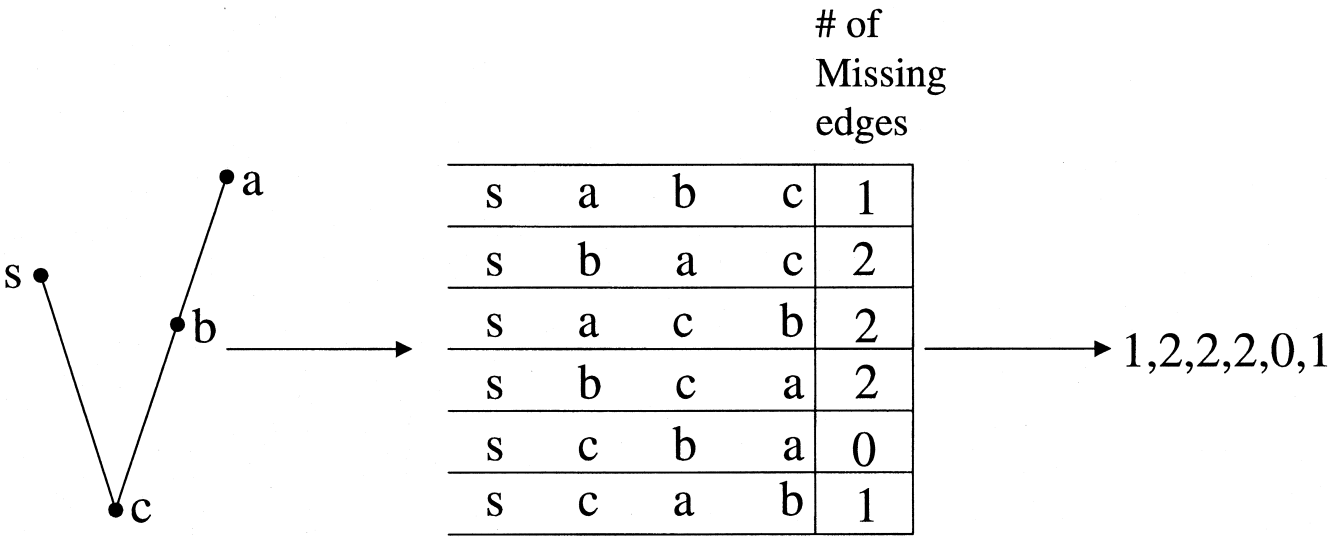

Figure 4: We can transform the hard *s*-Hamiltonian path problem to the trivial find-zero problem by enumerating possibilities, but property (3) is violated because the output of $\mathcal{T}$ is exponentially long.

can be thought of as beginning at *s*; simply cut out the new edge and what remains is a Hamiltonian path from *s* in *H*. (Exercise: What is wrong with this reasoning?) Figure 5 shows that this transformation will sometimes convert a no instance of *s*-Ham path to a yes instance of Ham cycle (a definite no-no). The beginner here is guilty of wishful thinking when verifying property (2)—convert Yes-from-Yes. All that we may assume is that *G* has a Hamiltonian cycle. We may not assume that this cycle uses exactly one of the new edges. The beginner is dazzled by the vision of the Hamiltonian path turning into a cycle by means of the new edge and perforce imagines all Hamiltonian cycles consist of a path from *H* extended by a new edge. But the cycle might contain two new edges and not correspond to a Hamiltonian path in *H*. (Question: Do cases in which the cycle contains no new edges cause an error?) (Exercise: Find a counterexample similar to Figure 5 in which graph *H* has only three vertices.) You might have noticed that if *H* is a 2-vertex graph with one edge, the transformation will convert this yes instance into a no instance. This is not a serious error. It only occurs on graphs with fewer than 3 vertices, and transformations are permitted to fail on a finite number of instances.

This kind of wishful thinking is a very common source of error in verifications of property (2)—Yes-from-Yes. Don't get so caught up in your vision of how solutions to I transform to solutions to II that you think all solutions to II have that form. To verify property (2), you may assume only that a solution exists; you may not make assumptions about its form.

Let's dig another pit for our beginner to fall into. The previous transformation didn't work because we connected *s* to too many vertices. This time we add only one new edge, so a cycle in *G* can't contain two new edges. If *H* contains a Hamiltonian path starting at *s*, take the vertex *v* at the end of that path and add a

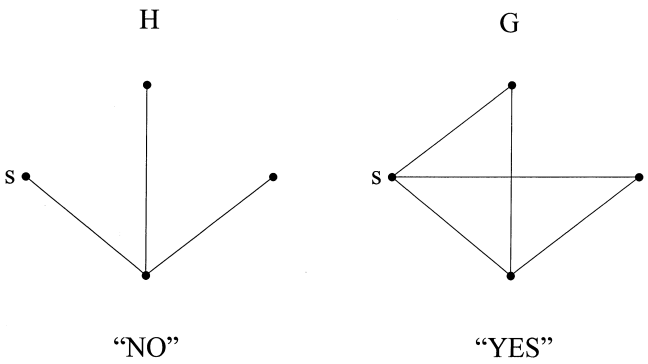

Figure 5: Connecting *s* to all nodes in *H* violates property (2). In the case depicted, *H* has no Hamiltonian path from *s* but *G* has a Hamiltonian cycle.

single new edge between v and s. The resulting graph G has a Hamiltonian cycle. This verifies property (1)—Yes-to-Yes; this time we can also verify property (2)—Yes-from-Yes. A Hamiltonian cycle in G can't contain more than one new edge. Remove the new edge if it is present, otherwise remove either edge incident on s. The remaining edges of the cycle form a Hamiltonian path from s in H. (Question: What is wrong with this transformation?)

The error is that we have violated property (3)—$\mathcal{T}$ is fast. How does the transformation $\mathcal{T}$ determine the vertex v? $\mathcal{T}$ is just dumb, fast front-end software. When it gets an instance as input, $\mathcal{T}$ doesn't know whether the answer is yes or no, much less can it solve the instance. But when we "take" the vertex v at the end of the Hamiltonian path in H, we are implicitly assuming that $\mathcal{T}$ has been clever enough to find that Hamiltonian path.

When you verify property (1)—$\mathcal{T}$ maps yes instances of I to yes instances of II—you are permitted to assume the answer to the instance of I is yes. When you verify property (2)—$\mathcal{T}$ maps yes instances of II from yes instances of I—you are permitted to assume the answer to the instance of II is yes. But the transformation $\mathcal{T}$ is never permitted to make such an assumption. If a no instance is input to $\mathcal{T}$, it must produce a no instance. We often don't see no instances because we prove property (2) by the contrapositive, but these instances are crucial. (Question: Why isn't $\mathcal{T}$ permitted to know whether the answer to the input instance is yes or no?) If instead $\mathcal{T}$ selects an arbitrary vertex x in H to connect to s, property (1) may fail (Figure 6).

Finally, let us construct a valid transformation. Add two new nodes, t and u, to H (Figure 7). Connect t to every vertex except s in the new graph G, but connect u only to s and t. If there is a Hamiltonian path from s in H, you can extend to a Hamiltonian cycle in G by appending t, then u. The required edges from unknown vertex x to t, from t to u, and from u back to s, are all in G. Conversely, if there is a Hamiltonian cycle in G, it must use the edges from t to u and from u to s. This is because G doesn't have any other edges incident to u. Removing these edges and the other new edge (the other one from t) leaves a Hamiltonian path

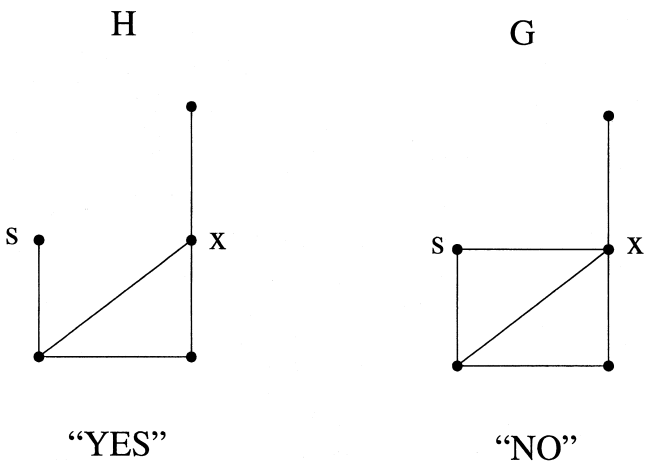

Figure 6: Connecting *s* to any one node *x* can convert a yes instance of *s*-Hamiltonian path to a no instance of Hamiltonian cycle, violating property 1. $\mathcal{T}$ may not calculate which *x* to use without violating property (3).

from s in H. (Question: If we omitted u from the transformation and connected s directly to t, which part of the proof would become invalid?) (Exercise: Find an instance for which this invalid proof fails.)

The key to avoiding the wishful-thinking error is to put into the instance an extreme substructure that forces a solution to have the desired characteristic. In problems involving Hamiltonicity, vertices with only one or two incident edges have strong forcing power. In other problems, it may be helpful to use very large or very small costs or sizes, or other extreme conditions. (Question: In a scheduling problem with precedence constraints, what kind of job might have strong forcing power?)

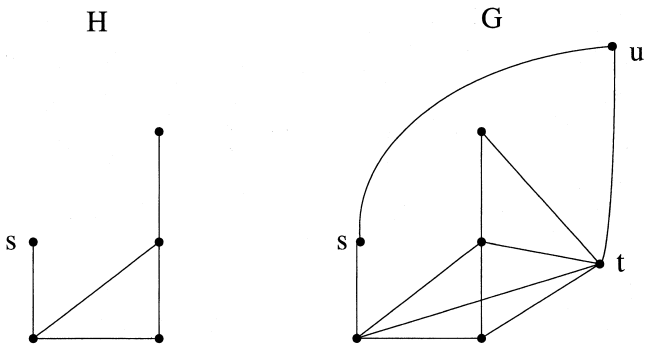

Figure 7: A valid transformation of *s*-Ham path to Ham cycle allows any Hamiltonian path from *s* in *H* to extend to a cycle in *G*, and forces any Hamiltonian cycle in *G* to traverse from *t* to *u* to *s*.

We implemented the transformation of Figure 7 for the circuit-card-assembly project described in §3. Because our code already contained a module to solve Ham cycle, the most practical way to solve *s*-Ham path was to add a front end to that module. Also, Ham cycle is the more widely studied problem of the two. We planned to acquire better software for Ham cycle in the future and swap out our own module.

In the ordinary undirected graphs we have encountered so far, an edge (i, j) permits travel in either direction between i and j. In a directed graph, an arc (i, j) permits travel only from i to j.

Directed Ham Cycle
Instance: A directed graph $G = (V, A)$.
Question: Does G contain a directed Hamiltonian cycle, that is, a permutation $\pi(i)$ of the vertices V such that $(\pi(i), \pi(i + 1)) \in A \ \forall i = 1, \ldots, |V| - 1$ and $(\pi(|V|), \pi(1)) \in A$?

Suppose we know that the directed Ham cycle problem is hard and wish to prove Ham cycle is hard. For each vertex v of G, make three copies, v_{in}, v_{middle}, and v_{out}, connected in a path (Figure 8). For each arc (v, w) of G, make an edge from v_{out} to w_{in}. Call the new undirected graph H.

Property (1)—Yes-to-Yes: Suppose G contains a Hamiltonian cycle $v^1, v^2, \ldots$. This implies that for all i, G contains the arc (v^i, v^{i+1}). Therefore for all i, H contains the edge $(v^i_{out}, v^{i+1}_{in})$. H also contains the edge between the in-copy and the middle-copy and the edge between the middle-copy and the out-copy of v^i, for all i. Therefore H contains the Hamiltonian cycle v^1_{in}, v^1_{middle}, v^1_{out}, v^2_{in}, v^2_{middle}, $v^2_{out}, \ldots$.

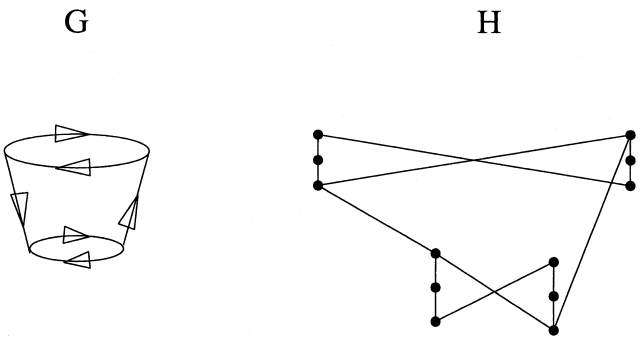

Figure 8: To transform the problem on a directed graph *G* into a problem on an undirected graph *H*, split each vertex into three vertices. A tour of the vertices of *G* must correspond to a tour of the vertices of *H*.

Property (2)—Yes-from-Yes: If H contains a Hamiltonian cycle C, we can think of C as starting at some vertex v^1_{in} of H. The cycle C is Hamiltonian and must visit v^1_{middle}. But that vertex has only two edges incident. So C must go from $v^1_{in} \to v^1_{middle} \to v^1_{out}$. From v^1_{out} the cycle must go to an in-copy of some other vertex, since all its edges go to in-copies (besides the edge to its own middle-copy, which C has already used). From that in-copy of, say, v^2, the cycle C must go to the middle- and out-copies of v^2 by the same reasoning as before (v^2_{middle} has only two edges). Thus C traverses the in-, middle-, and out-copies of each vertex of G, returning to v^1_{in}. This corresponds to a Hamiltonian cycle in the directed graph G.

Property (3)—$\mathcal{T}$ is fast: The transformation requires time proportional to the size of the graph G, which is a polynomially bounded function of the size.

A transformation that omits the middle copies of the vertices is flawed (Figure 9). (Question: Which pit have we fallen into? Exactly where does the above proof become invalid?)

If you are a visual thinker, you will find the following exercise an excellent way to achieve a firm understanding of correct NP-hardness proofs. (Hamiltonian variations exercise: Find transformations between all pairs of these four variations on the Hamiltonian theme. All graphs are undirected, although you could do all the directed variations, too.)

(1) Ham cycle

(2) *s*, *t*-Ham path: **Instance**: Graph $G = (V, E)$ with two distinguished vertices s and t. **Question**: Does G contain a Hamiltonian path with end points s and t?

(3) *s*-Ham path

(4) Ham path

To summarize, the most common pitfall in NP-hardness proofs is wishful thinking, in which you groundlessly assume a solution to II must have certain properties or structure. This makes your proof of property (2) invalid. You can often get around this difficulty by introducing extreme values or substructures in the instances of II, which force a solution to take a particular form. Other common pitfalls are to expect $\mathcal{T}$ to guess something that actually depends on knowing the

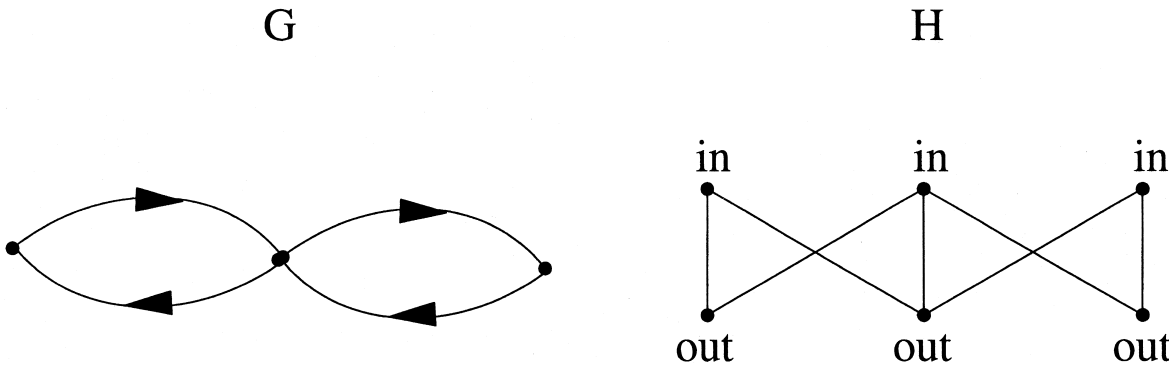

Figure 9: If the "middle" vertices are omitted, the no instance *G* transforms to the yes instance *H*, violating property (2).

solution or to do more than a polynomial amount of work.

Answers:

—Given an instance of LP feasibility, solve it with a polynomial time algorithm, such as Karmarkar's or the ellipsoid. If the instance is feasible, output the list $L = \{0\}$; otherwise, output the list $L = \{1\}$.

—No, those cases don't cause an error. The input instance has answer yes.

—Property (3) is violated, as explained in the text following.

—If it knew, it could output 1 if yes and 0 if no and thereby transform any problem into the trivial problem: instance, a binary number x; question, is $x = 1$?

—The contrapositive proof of property (2) would be invalid. G might have a Hamiltonian cycle that doesn't use the edge (t, s).

—A job that must be performed after all the other jobs or before all the other jobs.

—Wishful thinking. Property (2) fails because we can't be sure that C uses the edge (v^1_{in}, v^1_{out}).

6. Examples of NP-Hardness Proofs

This section contains a series of NP-hardness transformations arranged in order of increasing difficulty. The easiest method is *parameter specialization*: restrict the natural parameters of your problem, and arrive at a known NP-hard problem. The more problems you know are NP-complete, the more powerful a weapon you have. So many problems are known to be hard, and real problems are so complicated, that this method often suffices.

If the definition of a problem contains a parameter, you can fix the value of the parameter to define another problem that is no harder than the original one. For example, an instance of IP (integer programming) consists of an m by n constraint matrix A, an m-vector of right-hand-side values b, an n-vector objective c, and a threshhold value T. Fix $m = 1$ and you get the definition of the knapsack problem. Since knapsack is hard, IP must also be hard. As another example, the graph-coloring problem, given a graph $G = (V, E)$ and integer K, asks whether the vertices in V can be assigned colors $1 \ldots K$ such that vertices connected by an edge are differently colored. Fix $K = 3$ and you get the definition of graph 3-coloring (§4). Since graph 3-coloring is hard, graph coloring is hard. The transformation $\mathcal{T}$ in both of these examples is the identity, $\mathcal{T}(I) = I$. The three properties must be satisfied by the identity transformation, so I would write down the parameter restriction and the name of the resulting problem but not the rest of the proof.

You can also fix a general threshhold parameter at a particular value. I hinted at this kind of transformation in §2, describing partition as a restricted case of 2-machine line balancing. Intuitively, when $V = \Sigma_i t_i/2$, the line-balancing problem is the partition problem. To be more precise, consider all the instances of line balancing in which $V = \Sigma_i t_i/2$. These instances comprise the set of all instances of the partition problem.

The transformation, which I would not ordinarily write down, is as follows. Given an arbitrary instance

$x_1, \ldots, x_n$ of partition, set $t_i = x_i \; \forall \; i = 1, \ldots, n$ and set $V = \Sigma_i t_i/2$. The resulting instance of line balancing has answer yes if and only if the given instance of partition does. (Question: In a graph with nonnegative edge costs, the shortest-path problem is a very well-known easy problem. How would you prove that the longest-path problem is hard?)

Garey and Johnson (1979) describe *specialization*, a proof method one step up from parameter specialization. In a formal sense, almost every NP-completeness proof is of this type, but this is what it means in practice: restrict the data of your problem in a simple way, and show that the resulting special class of your problem is the same as a known NP-hard problem.

For example, in scheduling a set of n jobs with deadlines on a single processor, the ith job requires time t_i and incurs penalty p_i if it is not completed by its deadline d_i. The problem is to minimize the total penalty incurred or equivalently to maximize the sum of the p_i values of jobs completed on time.

When all jobs have the same deadline D, the problem is precisely the knapsack problem with knapsack capacity D, item sizes t_i, and values p_i. (Question: What transformation $\mathcal{T}$ are we implicitly using?) (Exercise: Prove the following problem is NP-hard: There are two processors, and each job must be run on one processor. Job i has processing time t_i and deadline d_i. The problem is to minimize the number of late jobs.)

The next example of specialization transforms Ham cycle to directed Ham cycle. The latter problem is the same as Ham cycle, except the graph's arcs are directed (one-way) rather than undirected (two-way). The cases of directed Ham cycle in which all arcs come in pairs (i, j) and (j, i) are equivalent to the undirected Ham cycle problem.

As another example, we transform Ham cycle to the threshhold version of the traveling-salesman problem.
Instance: Complete graph $G = (V, E)$ with integer edge costs $c_e : e \in E$ and integer K.
Question: Does there exist a Hamiltonian cycle in G with total cost $\leq K$?

Restrict this problem to cases in which all edge costs c_e are 0 or 1 and $K = 0$. The question is then equivalent to whether G has a Hamiltonian cycle consisting only of 0-cost edges. Thus it is the same as the Hamiltonian cycle problem. The transformation, which ordinarily we would not explicitly describe, is as follows. Take as input an instance of Ham cycle, a graph $H = (U, F)$. Set the cost of each edge in F to 0; then fill in the graph with the rest of the possible edges, each costing 1. Call the resulting complete graph G. Finally, set $K = 0$.

Next I show that 3-matching transforms to exact 3-cover, one of my favorite problems. On a base set we are given a collection of acceptable triples. The exact 3-cover problem asks if there is a partition, or exact cover, of the base set comprised of acceptable triples.

Exact 3-Cover
Instance: A collection S of subsets of $X = x_1, \ldots, x_{3n}$, each subset size 3.
Question: Is there a subcollection of n subsets in S whose union is X?

3-matching is the special case in which all the acceptable triples x_i, x_j, x_k happen to be of form $1 \leq i \leq n < j \leq 2n < k \leq 3n$. (Exercise: Show that the set-covering problem is hard by transforming exact 3-cover to it.)

Most NP-hard problems arising in practical applications are fancy versions of one or more canonical NP-hard problems in the literature. If you strip away most of the clutter, you often find that the much simpler problem underneath is still hard. Consider, for example, most vehicle-fleet-planning problems. There is a set of vehicles (for example, trucks or ships) with various travel ranges and capacities for conveying one or more commodities. There are one or more supply points, demands at various locations, and possibly restrictions on the timing of pickups, deliveries, and so forth. You seek a low cost or perhaps just a feasible schedule to meet demand. (Question: Where do you spot potential complexity in this problem?)

—If you seek a low-cost plan, and travel distances affect cost, then even if there were only one supply point and one vehicle with plenty of capacity and range, you would have a traveling-salesman problem.

—It is NP-hard to split a set of numbers into two groups whose sums are equal. This is the partition problem. If there are only two identical vehicles, a single supply point, and it is either infeasible or costly to split deliveries between vehicles, then the problem is hard. Why? In the case that total demand equals total fleet capacity, each vehicle has to go out full, hence a

partition problem must be solved. This transformation works even if travel costs are relevant: just make them all equal.

—Partition is not as bad as some other hard problems because it is solvable by dynamic programming in practice. If there are m identical vehicles, however, splitting deliveries equally contains the 3-partition problem, which is hard and not efficiently solvable by dynamic programming.

—If deliveries may not be split between vehicles, then the problem of minimizing the number of vehicles to meet demand is the hard bin-packing problem. I'm assuming that fleet size affects costs.

—If there are multiple commodities, even the associated network-flow problems are NP-hard.

If deliveries may not be shared among vehicles, it is hard merely to parcel out the deliveries to minimize cost or to meet demand without exceeding capacity. If deliveries may be split among vehicles, then assigning deliveries among vehicles is like a transportation or network-flow problem. However, even in this situation, the delivery cost to one location depends on where else the vehicle goes. This is the part of the problem that is similar to the traveling-salesman problem, and it makes things difficult.

The point is that regardless of vehicle differences, customer peculiarities, and multiple commodities, just routing vehicles or partitioning the orders is apt to be hard.

Many real applications are made more difficult by uncertainty in future demand, salvage (estimated future) values of vehicle locations, inventory status, and so forth. As a practical matter you can usually count on such factors making your problem substantially harder.

Padding and forcing transformations are a step up in difficulty from specialization. In general, these methods require you to do a little tinkering with small examples. They do not require great ingenuity or a big flash of insight. After you do the exercises in this tutorial, you should be able to perform this kind of transformation. If you know a few dozen NP-hard problems from the literature, you will find that padding, forcing, and specialization are enough to resolve complexity questions in most practical situations.

Often we want to make the solution to an instance have a particular form or property to get a transformation to work. One method, *forcing*, is to use an *enforcer*, a substructure that forces something to happen. I used enforcers in some transformations earlier (§§3 and 5). For example, a vertex incident to only one edge in an instance of Ham path forces the solution (if there is one) to have that vertex as an end point. Similarly, a vertex with only two edges in an instance of Ham cycle forces the solution (if there is one) to contain those two edges. It is usually simple to contrive enforcers for an NP-hard problem. If you cannot do so, this may be a clue that your problem is not NP-hard.

The *subset-sum* problem takes a set of integers $X = \{x_1, \ldots, x_n\}$ and integer K as an instance. The question is to find a subset of X whose elements sum to exactly K. (Question: Prove the subset sum problem is hard by using specialization.) To transform the subset sum problem to the partition problem, include two additional elements in X. The first has value $y = M + K$; the second $z = M + \Sigma_{i=1}^{n} x_i - K$. Here M, like a penalty term, must be large enough to dwarf the sum of the original terms. The big value forces y and z to appear in different subsets in any solution to the partition instance $\{X, y, z\}$.

Almost any scheduling problem that is hard on K machines is also hard on $K + 1$ machines. To transform the K machine problem, you can usually add a single large job (or a set of highly interconnected jobs) to the instance. These added jobs completely tie up the $K +$ 1st machine, leaving a de facto K-machine problem.

I chose the last example of forcing because it could just as well be thought of as *padding*, our next method. The idea is to pad an instance, typically with trivial elements, to permit a property to be satisfied. This may seem like cheating, but it is perfectly legal. For example, some years ago, Ammons, Lofgren, McGinnis, and I encountered a machine-configuration problem that looked almost like a partition problem (Lofgren 1986). A manufacturing company had two identical flexible assembly machines, each of which could be configured with up to 24 tools. Associated with each of the 48 tools was a certain amount of work. The problem was to divide the tools among the machines to balance their workload. This is identical to the partition problem, except the two subsets must have the same number of

elements. We can define this *equipartition* problem as follows:

Instance: Set of integers $X = \{x_1, \ldots, x_{2n}\}$.

Question: Is there a subset $S \subset X$ with $|S| = n$ such that $\Sigma_S x_i = \Sigma_X x_i/2$?

How could we show that equipartition is hard? Given an instance $x_1, \ldots, x_n$ of partition, pad the instance with n additional zero elements $x_{n+1} = \ldots = x_{2n} = 0$. (Question: Show that this transformation satisfies the three properties.) (Exercise: Suppose every tool carries a nonzero amount of work. Prove equipartition is hard even if all x_i must be strictly positive. Hint: Increase each term by 1.)

Exercise: As another example of forcing, transform equipartition to partition. Hint: Given an instance $y_1, \ldots, y_n$ of equipartition, add a large constant M to each y_i, giving $x_i = y_i + M$.

Exercise: Use padding to transform max cut to bisection max cut.

Exercise: 4-SAT is defined the same as 3-SAT, except each clause contains exactly 4 literals. Transform 3-SAT to 4-SAT. (Hint: Force a dummy variable to be false; use it to pad the 3-clauses into 4-clauses.)

Answers:

—On an n-vertex graph with edge costs of 0 and 1, set the threshhold value to $n - 1$ to specialize to the Ham path problem. This shows it is hard to find the longest path.

—Given in the text following.

—The restriction of subset sum, in which $K = \Sigma_i x_i/2$, is partition.

7. Gadgets: Transforming 3-SAT to 3-Coloring

In this section, I will construct a more complicated transformation, from 3-SAT to 3-coloring. The transformation requires the invention of one or two gadgets, in this case, small graphs to use as parts of the 3-coloring instance. I will show some of the blind alleys I went down while developing this proof to show the kind of thinking one does when inventing gadgets.

3-coloring a graph takes a graph G as an instance. Question: Can G be legally colored using not more than three colors? That is, can we assign to each vertex of G a color A, B, or C, in such a way that we assign different colors to vertices linked by an edge? 3-SAT takes a set of clauses as an instance. Each clause is a set of three different terms called *literals*, each of which is a complemented or uncomplemented true-false variable. For example, $\{X_1, \bar{X}_3, X_6\}$ is a possible clause. Question: Can the variables be assigned values of true or false so that at least one literal in each clause is true?

First, we need a gadget or component to represent the Boolean variables X_i. What portion of a 3-coloring problem could correspond to a true-false, yes-no decision? A single vertex does not work, since it could have any of three values. But if we could forbid one color, the other two possible colors could correspond to T and F.

Let's call the colors A, B, and C. Add a triangle, which we may assume is colored as shown in Figure 10, since there is no a priori difference between colors.

Connect X_1 to the vertex colored C. Now X_1 is A can mean X_1 = true, and X_1 colored B can mean X_1 = false. Conveniently we can get an $\bar{X}_1$, too (Figure 11). Vertex $\bar{X}_1$ must be A or B, and it is B if and only if X_1 is A.

The same construction gives us all of our variables X_1 through X_n, along with their complements. (Question: Why wouldn't it work for each X_i to have its own triangle marked ABC?) We will call these $2n$ vertices the variables' vertices.

Now we need a graph structure to act as a clause-gadget. It must meet the following conditions:

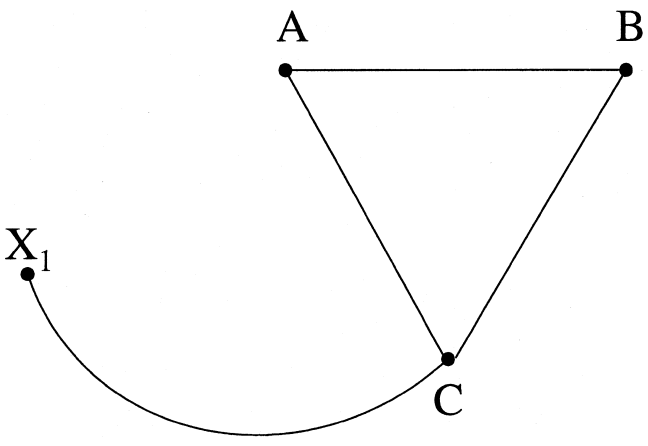

Figure 10: The triangle's edges force the coloring A, B, C, allowing for symmetry. Force vertex X_1 to be colored A or B by connecting it to vertex C.

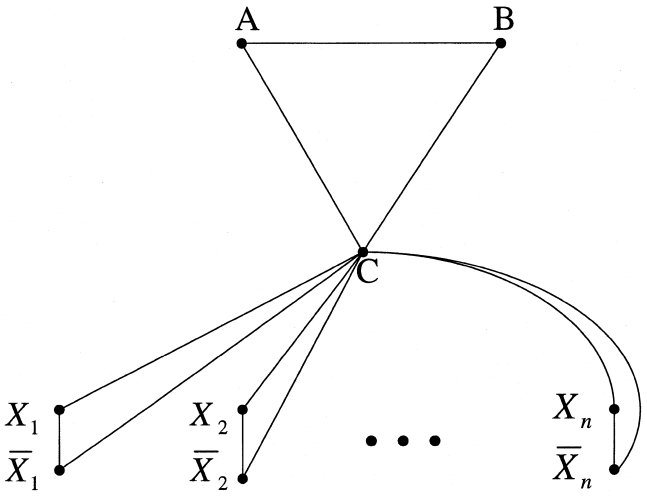

Figure 11: This gadget converts true-false variables to 3-coloring substructures. In any legal coloring, either $X_i = A$ and $\bar{X}_i = B$, or vice-versa. The first choice corresponds to setting X_i true.

(1) Clause-gadget connects to three variables' vertices in some way. To help us think, let's work with $(X_1 \vee X_2 \vee \bar{X}_3)$ as a generic example. This example intentionally contains both complemented and uncomplemented variables.

(2) If one or more of the three variables' vertices is colored A, then clause-gadget can be colored. (Question: What does this condition mean?)

(3) If all three variables' vertices are colored B, then clause-gadget cannot be legally colored. (Question: What does this condition mean?)

My first idea for clause-gadget did not work. I tried a triangle (Figure 12). The good news is that this structure satisfies condition (3). If all three variables' vertices are colored B, the triangle can't be legally colored since none of its vertices can be B. The bad news is that if those three vertices were colored A, then the triangle could not be legally colored. So condition (2) fails. My second idea for clause-gadget (Figure 13) failed as well. (Question: Which condition fails?)

The first idea prohibited all three variables' vertices from having the same color. The problem was that all A behaved the same as all B. To make A different from B, we must know which is which. This would mean using the triangle labelled ABC again. That gave me an idea.

In Figure 14, if X_1 is colored A, then its partner vertex Y_1 may be colored B or C (freedom of choice). But if X_1 is colored B, then Y_1 must be colored C (no freedom of choice). Put this together with the first try to get a valid clause-gadget (Figure 15). If all three variables' vertices are B, then there is no freedom of choice. Their partners are all C and the inner triangle cannot be legally colored. This gives condition (3). But as long as at least one variable's vertex is A, then one of the partners can be B. The other partners can be C, so the inner triangle can be legally colored, giving condition (2).

The pictures say it all, but we still must write the algebra. One of the awkward parts in writing down the proof is trying to describe correspondences in algebraic notation. You can often make your formal proof more readable by making a definition and lemma about your gadget.

Clause-Gadget Definition. A clause-gadget is a six-vertex subgraph, associated with a particular clause C, containing vertices $d^l, p^l : l = 1, 2, 3$ and edges (d^1, d^2), (d^2, d^3), (d^3, d^1), and $(d^l, p^l) : l = 1, 2, 3$. In addition, the clause-gadget must be attached to the rest of the graph by edges $(a, p^l) : l = 1, 2, 3$, where a is the special vertex colored A, and by edges $(L^l, p^l) : l = 1, 2, 3$, where L^l is C's lth literal's vertex.

Clause-Gadget Lemma. *A clause-gadget subgraph for clause C can be 3-colored legally if at least one of C's literals' vertices is colored with A. Conversely, if the clause-gadget subgraph is legally 3-colored, then at least one of C's literals' vertices $L^l : l = 1, 2, 3$, is not colored with B.*

The proof of the lemma could be as follows: If L^t is colored with A, then color p^t with B, the other two p^l with C, d^t with C, and the other two d^l with A and B. This is a legal 3-coloring regardless of the coloring of the other L^l. Conversely, in any legal 3-coloring of the clause-gadget the three d_l must be colored differently, because of the edges between them. Hence, one d^t must be colored with C. Then p^t must be colored with B, because of the edges (d^t, p^t) and (a, p^t), and so L^t must not be colored with B.

The NP-hardness proof of 3-coloring would then be written as follows:

Given an arbitrary instance of 3-SAT with n variables X_i and m clauses C_j, create vertices $\{a, b, c; x_i, \bar{x}_i : i = 1, \ldots, n\}$ and edges (a, b), (b, c), (c, a) and $(x_i, \bar{x}_i)$, (c, x_i) and $(c, \bar{x}_i) : i = 1, \ldots, n$. Vertices x_i and $\bar{x}_i$ correspond to the literals X_i and $\bar{X}_i$, respectively. For each

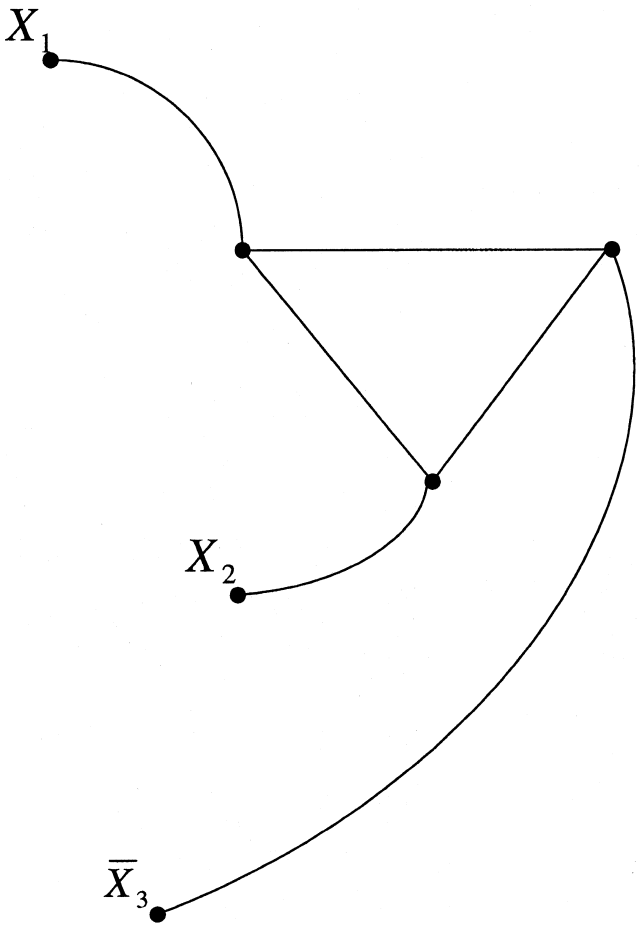

Figure 12: My first idea for a 3-coloring clause-gadget violated the Yes-to-Yes property. If all three literals are colored A (true), the triangle cannot be legally colored.

clause C_j, attach a clause-gadget. Let G denote the resulting graph.

Property (1)—Yes-to-Yes: Suppose the instance of 3-SAT has a satisfying truth assignment. If $X_i = T$ in this assignment, color vertex x_i with A and $\bar{x}_i$ with B; otherwise color vertex x_i with B and $\bar{x}_i$ with A. Color a with A, b with B, and c with C. Since the assignment is satisfying, each clause contains at least one literal whose vertex is colored with A. By the lemma, the clause-gadgets can be colored legally. Therefore G can be legally 3-colored.

Property (2)—Yes-from-Yes: Suppose there exists a valid 3-coloring of G. Without loss of generality a is colored with A, b with B, and c with C. For each i, one of the two vertices x_i, $\bar{x}_i$ must be colored with A and

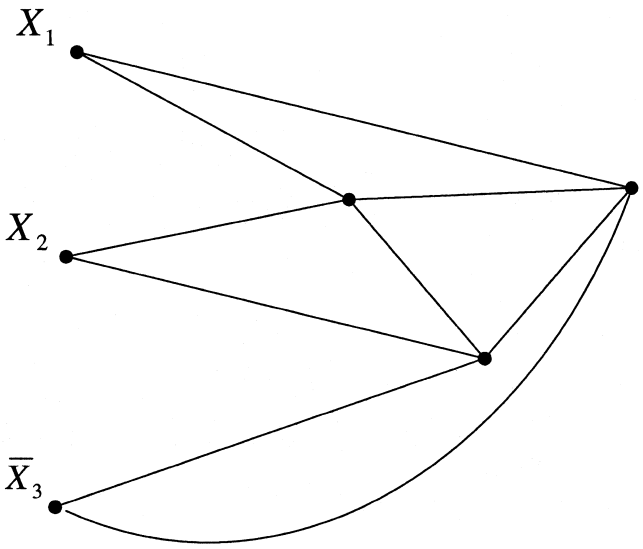

Figure 13: My second idea for a 3-coloring clause-gadget failed too.

the other with B because of the edges $(x_i, \bar{x}_i)$, (x_i, c), and $(\bar{x}_i, c)$. Then by the lemma, at least one of each clause's literal's vertices must be colored with A. In the 3-SAT instance, set X_i = true if and only if x_i is colored with A. This truth assignment is satisfying because at least one of each clause's literals is true.

Property (3)—$\mathcal{T}$ is fast: The construction of G takes polynomial time.

In a typical published NP-hardness proof, the explanatory references to edges would be omitted, as would the entire motivating discussion preceding the algebraic description. The gadget would probably not be illustrated, since it is not complicated enough. All this makes NP-hardness proofs so hard to read that many experts rarely read other people's proofs! It is usually easier to construct one's own proof, perhaps

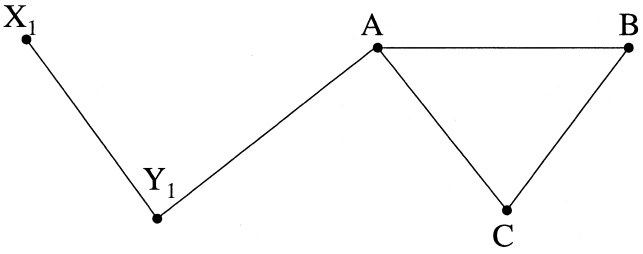

Figure 14: An idea for part of the clause-gadget. If X_1 is colored B (false) we are forced to color Y_1 with C, but if X_1 is true, we have a choice.

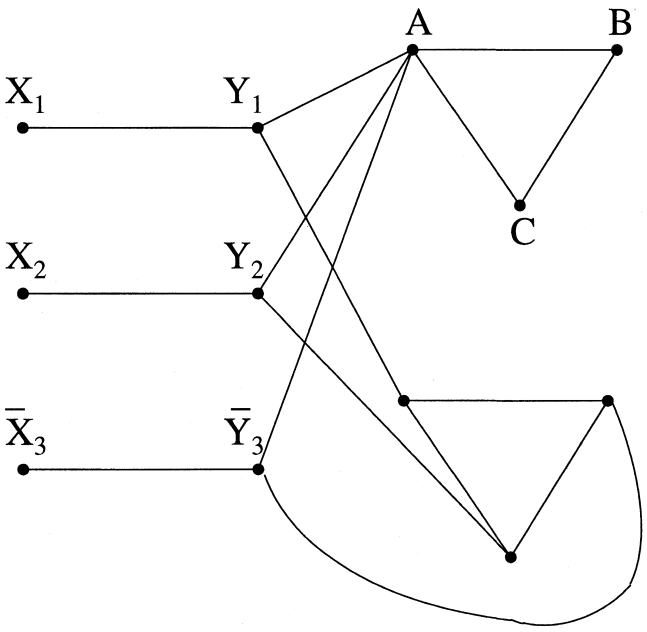

Figure 15: This valid clause-gadget can be legally 3-colored if and only if at least one of the three literals X_1, X_2, $\bar{X}_3$ is true (colored *A*).

with the hint of knowing what problem to transform from.

In my experience, it usually is not too much trouble to cook up a gadget for an NP-hard problem. As with enforcers, if you can't build gadgets for a problem, it could well be a sign that the problem is not NP-hard. Answers:

—There would not be any consistent meaning to the colors of variables. Indeed, X_1 could be A, X_2 could be B, and X_3 could be C, so colors of variables would not be restricted to two choices.

—If at least one of the literals in the clause is true, hence the clause is satisfied, then the corresponding clause-gadget portion of the graph can be colored.

—If the clause is false, the corresponding clause-gadget cannot be colored.

—Condition (2) fails again.

8. Dealing with NP-Hard Problems

> Complexity theory is usually an excuse for laziness. Give me a problem: I'll solve it (Dantzig 1979).

How do you solve a hard problem? Fortunately, the NP-hard classification leaves several important loopholes. Some NP-hard problems can be solved well with dynamic programming. The realistic cases might have special properties that permit effective solution. For example, if the realistic cases are not very large, when they are modeled as IPs, you should consider solving your problem with commercial math-programming software. One of the major accomplishments in our field over the past couple of decades is the solution of the medium-size IP, partly due to increases in computer power, and partly due to the enormous improvement of our algorithms.

If you can't readily solve your problem with off-the-shelf software, you may need to choose between heavy-duty mathematical programming methods and heuristics. Since tools for the former now accomodate the latter, you may also pursue both alternatives.

When is it appropriate to use mathematical programming? If the problem is stable, not apt to change within a year or two, and of great economic value, so getting that extra 0.1 or one percent is worth a great deal, and you are confident of the accuracy of your model and data, it may be worth the investment in time and money to pursue a mathematical programming solution. Many contenders for the Franz Edelman Award fall into this category.

When are heuristics appropriate? When you are unsure of the model, for example, when you can only approximately quantify the objective. When your data are not accurate and you don't need an exact solution to a guesstimated problem. When the problem is transient or unstable, and robustness in the solution method becomes important. When the instances are really enormous or the problem very difficult in practice (for example, job shop scheduling, quadratic assignment). When solution speed is critical. When your client prefers a decent solution now to a great solution later. When the amount of money at stake doesn't justify using other techniques. And when the expertise necessary for other methods is not available. In these situations, you may choose to pursue heuristic solutions. If they are successful, you can always reassess the option of employing a more costly but more exact solution method. Complexity theory can sometimes help you to assess the prospects for an approximate solution.

Finally, you can often solve a hard problem by changing your model. Turn a constraint into an objective or vice versa; aggregate or extract what matters most; approximate on the model level rather than solution level. Computational complexity can help you to choose among models by assessing their difficulty.

Pseudo-Polynomial Algorithms, Dynamic Programming, and Unary NP-Hardness

A few NP-hard problems can often be solved in practice via dynamic programming. These problems are hard only in the sense of getting exact solutions when the numbers are many digits long. If your problem naturally occurs as a knapsack, partition, or other such problem, you are in luck if the data do not occur to great precision or if you do not require an absolutely precise solution. However, many other NP-hard problems, including 3-partition, bin-packing, and all number-free problems, are not susceptible to quick solution by dynamic programming. Complexity theory provides an additional tool to classify susceptibility to dynamic programming.

How big a table do we need to solve the knapsack problem by using dynamic programming? If the data are integers and the knapsack capacity is K, the table is n by $K + 1$ large, requiring $O(nK)$ work. It suffices to store one row of the table at a time, but there is no way to avoid the factor of K (in the worst case) work and space requirements. The algorithm is called pseudo-polynomial because it is only fast when K is small. This ties back to my discussion in §1 about length. In that section, I mentioned that the natural parameters of a problem are usually good surrogates for the input length. If the dynamic programming algorithm is polynomial in those natural parameters, we say the algorithm runs in pseudo-polynomial time.

However, theoretically speaking, K can be enormous. This is because it takes only log K digits to write the number K. If K were an 80-digit number, the required table could not fit into the combined disk memory of all Internet-linked computers. In the proof that knapsack is NP-hard, the transformation takes a problem like exact 3-cover on a ground set of n objects and makes numbers that are each about $4n$ digits long. This is polynomial in length, but the numbers are of extraordinarily high precision. No naturally occurring knapsack instance would require that much precision. So, most real problems that are naturally modeled by knapsack have fairly small K, but other problems when transformed to knapsack will turn out to have impractically large K. In practice, dynamic programming algorithms are apt to run out of memory before they bog down with CPU usage.

There is another good thing about naturally occurring knapsack cases. Even if K is large, dynamic programming will quickly find an approximate solution to knapsack. Simply round the sizes to a few digits of precision; round them up if the capacity constraint isn't at all soft. (There are more effective heuristics too.) This technique is also appropriate when the data are given to high precision, but you don't have confidence in the low order bits.

The bad news is that many NP-hard problems are not susceptible to dynamic programming. These problems are called unary NP-hard or strongly NP-hard. All NP-hard problems that don't involve numbers at all, such as 3-SAT or Ham path, are unary NP-hard. Problems that involve numbers, but are obviously hard even when the numbers are small, are unary NP-hard, too. For instance, the TSP is hard when edge costs are 0 or 1 (Ham cycle), so TSP is unary NP-hard. Finally, some problems, such as bin packing and 3-partition, are very numerical yet are unary NP-hard.

Watch out for problems that are theoretically solvable in pseudo-polynomial time but are like unary NP-hard problems in practice. Dynamic programming is great for 2-machine line balancing; it is OK but slower by a factor of n for 3-machine line balancing; it is impractical for 5-machine line balancing. Theoretically, if m is allowed to vary and get large, the m-machine problem is like 3-partition or bin packing, and is unary NP-hard. Although the problem is not unary NP-hard for any fixed value of m, $m = 5$ is large, practically speaking.

Special Structure

> All good statisticians cheat by looking at the data (Chernoff 1975).

The realistic cases have special structure if the data cannot be arbitrary but are restricted in some way. Special structure often derives from geometric or other physical considerations, common causes, or other factors obvious to the problem-domain expert. One trick

to elicit this information is to show the expert some pathological instances and discover why they can't occur. Structure can make a problem easy. Unfortunately, many problems remain hard when restricted. Even so, special structure often permits solution of larger instances or better quality heuristic solutions.

A network flow problem is a kind of IP with special structure. While this particular structure is very well known, other helpful structures can be less obvious.

Here is a true story in which a problem that appeared difficult turned out to be quite easy.

Recently I was having dinner with my friend Ivan Chase, a biologist. Perhaps because the soup contained fish, he remembered a question he had meant to ask me. He had a hundred or so fish in his lab and wanted to run as many trials as possible of an experiment that required a small group of fish. No fish could participate in more than one trial, or the trials would not be independent. The difficulty was that some groupings among the fish were OK, and some were not. Ivan wanted to know what to do.

Immediately I visualized, quite incorrectly, that certain groups of fish somehow possessed a bad social dynamic, didn't get along well, and would spoil the experiment. Obviously, Ivan had a 3DM problem (3-dimensional matching, §4). He was trying to form, say, 33 groups of three out of 99 fish, when only certain groups of three were permitted. Then it occurred to me that what went wrong might be pairs of fish that didn't get along, rather than a complicated dynamic among three fish. In 3DM the triples *abc*, *abd*, and *bcd* might all be acceptable, while *acd* would not. But if the problems were due to pairwise interference this could not occur. Fortunately I remembered a problem proved NP-hard by Garey and Johnson (1979), called *Partition into Triangles*. Given a graph $G = (V, E)$, the problem is to partition V into $|V|/3$ triples, such that for each triple, G contains all three edges between the members of that triple. So even if groups were unsuitable because of pairwise conflicts, Ivan's problem was still NP-hard.

I hesitated to tell Ivan that his problem was hard to solve, perhaps because I had a glimmer of the truth, but probably because I needed more time to think of a good heuristic for partition into triangles. I asked what made groups of fish unacceptable. Ivan explained that he couldn't put big fish together with little fish—differences of more than 15 percent in weight were not allowed. "I see," I responded, and discarding my complexity proof and heuristic ideas, gave him a simple procedure to maximize the number of groups. If the three heaviest fish weigh within 15 percent of each other, make them a group. Otherwise eliminate the heaviest fish, which belongs to no acceptable group. Recurse.

The moral of the story is that a problem may look NP-hard but be so highly structured as to be easy. The problem domain expert may tell you that certain jobs must be performed before others: if you visualize an arbitrary acyclic precedence constraint graph, you may be ignoring hidden structure. People close to the problem often think it is obvious what causes precedence constraints, preferences, or mutual incompatibilities (two activities can't occur at the same time) and don't tell you about the natural structure of these constraints or objectives. In my experience, the operations researcher usually has to elicit information about special structure from the problem-domain experts, to whom the structure is either too obvious or too unimportant to bear mentioning.

In the case of Ivan's fish, there is an important intuitive explanation of why the special structure of the problem made it easy. 3DM is difficult because of the arbitrariness as to which triples are OK and which are not. If you decide to use the triple *abc*, this might force *d* and *e* together and simultaneously force *f*, *g*, and *h* apart. Solving 3DM is like putting together a puzzle in which what you assemble in one location affects what can fit together in many other locations. This looks difficult: making a bunch of interlocking yes-no decisions to simultaneously satisfy a collection of arbitrary-looking constraints. The constraint on weight difference so severely restricts the possible patterns of permissible triples that the problem becomes easy. (Question: What if the total mass of the fish in each group must equal 300 grams? Is this problem easy or hard? What if the total mass of the fish in each group may not exceed 300 grams?)

Here is a method that often works to elucidate special structure. Find a transformation $\mathcal{T}$ from 3-SAT, IP, knapsack, (or some other NP-hard problem you understand well) to your problem. Then apply $\mathcal{T}$ to a

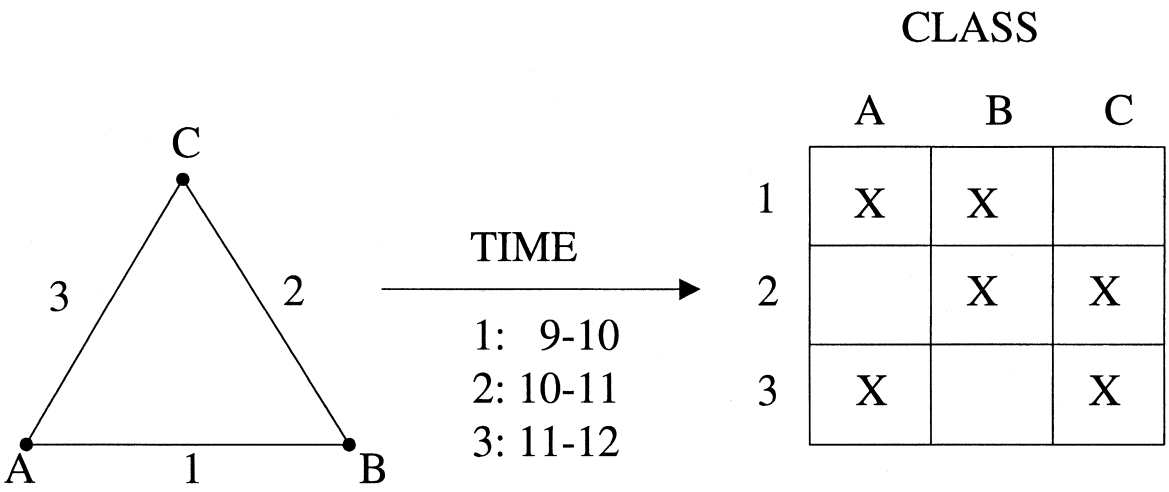

Figure 16: Transform coloring to classroom scheduling by turning colors into rooms. Each vertex becomes a class, and each edge (*i*, *j*) becomes a time period during which both classes *i* and *j* meet.

simple no instance of the hard problem whose LP relaxation is feasible. (For example, the constraints $2x \geq 1$; $2x \leq 1$, are such a no instance of IP feasibility.) The resulting instance of your problem is apt to be a small no instance. However, if special structure is making your problem easy, this instance likely cannot arise from the actual circumstances of your problem. Show this small instance to your problem-domain expert and ask whether it could happen and if not, why not? Whatever answer you get will deepen your understanding of the real problem.

In an example of the method at work, Mike Carter and I (1992) investigated classroom-scheduling problems, where classes that meet at various times must be assigned to rooms so that no classes use the same room at the same time. A simple transformation from graph coloring (§7) shows that finding a feasible schedule for classes is hard. Each vertex in the graph is a class; each edge is a time period during which the two incident classes meet; each color is a room. Assigning colors to vertices so no two vertices sharing an edge have the same color is the same as assigning rooms to classes so no two classes with overlapping time schedules have the same room.

To use the method, we began with the no instance of coloring a triangle with two colors (an odd cycle). The natural LP model would find a feasible fractional solution in which each vertex was assigned half of each color. This instance transforms to an infeasible classroom-scheduling instance (Figure 16). A practitioner criticized this infeasible instance, saying that one class met from 9:00 to 10:00 and 11:00 to 12:00, which didn't happen. This led us to realize that in many applications, each class uses one interval of time. Using properties of interval graphs, these problems may be easier to solve.

In a further application of this method (Carter and Tovey 1992, pp. S31–32), a practitioner criticized an infeasible instance we had generated because some teacher preferred every possible pair of the four rooms. This criticism made us realize that there could be a monotonicity structure among teacher preferences. This special structure permits scheduling to be done very easily in many applications, particularly in secondary schools.

Special structure doesn't always make a problem easier. The Ham-cycle problem (and therefore also the TSP) is hard even for grid-graphs in the Euclidean plane. Likewise, graph 3-coloring is hard on planar graphs. 3-SAT has a hard planar version as well: represent each variable and each clause by a vertex. If a variable or its complement appears in a clause, place an edge between the corresponding vertices. Impose the special structure that this graph must be planar and 3-SAT is still hard. 3,4-SAT, in which no variable may

appear in more than four clauses, is another useful hard restriction of 3-SAT (Tovey 1984).

Proofs of NP-hardness subject to special structure are often long and difficult. In general, these proofs either depend on gadgets or on what Garey and Johnson (1979) call the method of local replacement. It is often best to impose special structure step by step, generating a sequence of increasingly constrained NP-hard problems, finally arriving at the target problem. It can also help to start the transformation from one of the restricted NP-hard problems listed above.

While special structure may be a complexity theorist's headache, it may be just what you need to solve your problem. Even if special structure doesn't make a problem change from hard to easy, it will often give you an extra order of magnitude in instance size before your solution technique bogs down. Another thing that special structure frequently does is to improve the performance of heuristics. For example, 2-opting for the TSP has much better performance guarantees if the distances are Euclidean than if they are arbitrary or satisfy only the triangle inequality (Chandra, et al. 1999).

Approximate Solutions

> The devil's in the details. . . . But do the details matter? (Tovey 1993).

The idea of heuristic solutions brings up another complexity question. How hard is it to obtain approximate solutions to optimization problems? For some problems, this turns out to be NP-hard. For example, it is hard to find a solution to the traveling-salesman problem within a factor of 1,000 of optimality. Why? Because if there were a fast algorithm that always produced tours less than 1,000 times the length of the optimal tour, we could use it to solve Ham cycle quickly. (Question: What costs would the transformation assign to the edges?) A breakthrough in complexity theory has led to many recent results of this sort, including the max clique approximation problem listed in §4. Other problems can be quickly solved to within some constant factor of optimality although it is NP-hard to guarantee a closer approximation. For example, Goemans and Williamson (1995) have discovered a fast algorithm that finds a solution to max out that is within 48 percent of optimal, although it is NP-hard to find a solution that is within 44 percent of optimal. The NP-hardness results utilize the theory of probabilistically checkable proofs, and the techniques required are beyond the scope of this tutorial (Håstad 1997).

It is tougher to develop intuition about the complexity of approximability than about the complexity of exact solution. Nonetheless, sometimes you can couple your understanding of the real problem with your knowledge of NP-hardness to gain valuable insight.

Based on complexity theory and knowledge of the problem domain, one would expect the circuit partitioning problem to be fairly nasty. In this problem, the components of an electronic circuit must be partitioned into sets. Each set will be packaged as an integrated circuit, and the partition must minimize interconnections between partitions or optimize a similar objective. You should be able to see that the problem is hard by using a transformation from exact 3-cover (§6). The key to the transformation is that each component has to be placed in exactly one set (§4). We might be tempted to search for an approximate solution that does not require an exact partitioning, since that seems to be the source of the hardness. However, our problem-domain knowledge tells us that a solution would be useless if it placed some components in more than one set or omitted some components. The devil is in details that matter.

Bramel and Simchi-Levi (1995) provide a lovely example of a problem more susceptible to approximate solution using vehicle-routing heuristics. Vehicle-routing problems often consist of difficult routing decisions combined with difficult partitioning or bin-packing decisions. Although bin packing is hard, approximate bin packing is not hard, and indeed knowing the approximate value of the optimal solution to the bin packing is not hard. (This is partly because, in bin packing, the items don't have to fit together perfectly to produce a feasible solution, as they do in the partitioning problem above. They might have to fit perfectly to produce an optimal solution.) This insight suggests that the detailed routing be decoupled and performed *after* vehicles are assigned to regions, which helps lead to a theoretically effective heuristic solution. Bramel and Simchi-Levi then find a more practically effective heuristic by replacing the routing

problem with a location problem that can be approximately solved more easily.

Answers:

—Hard, this is the 3-partition problem.

—Hard. Proof by specialization. When the total mass of all n fish is $100n$ grams, the question of whether you can use all the fish is exactly the 3-partition problem. Same answer if I had required the mass of each group to be at least 300 g.

—Costs of 0 for edges in the original graph and 1 otherwise.

9. Other Complexity Classifications

Some problems cannot be solved by any algorithm. Software testing is a good example of a problem that is at this extreme level of complexity. It has been proved that no algorithm can correctly and thoroughly detect bugs in software.

This complexity gives us some insight into what is difficult and time consuming in our field. Generally speaking, it is an unsolvable problem to verify that a formal structure (for example, a model or a computer program) does what we want it to do. I believe this is a reason why OR/MS researchers have done little formal work in the important area of modeling and model validation—it is a very hard problem (Dantzig's (1963) activity analysis, Geoffrion's (1987, 1996) structured modeling, and Hackman and Leachman's (1989) continuous time framework for modeling production systems are exceptions).

Co-NP Completeness and Lack of Succinct Characterizations

Not all NP-hard problems are alike, although I've disguised that fact until this point in the tutorial. Figure 17 shows how the class of NP-hard problem subdivides into NP-complete, co-NP-complete, and other categories.

Here is an NP-complete problem: the set of all matrix-vector pairs (A, b) such that for some integer vector x, it is true that $Ax \leq b$. Visualize this problem as a point in the region NPC in Figure 17. Here is a co-NP-complete problem: the set of all matrix-vector pairs (A, b) such that for *no* integer vector x is it true that $Ax \leq b$. Visualize this problem as a point in the region co-NP-complete in Figure 17. The NP-complete problem consists of the yes instances to IP feasibility, while the co-NP-complete problem consists of the no instances.

NP-complete problems are not the same as co-NP-complete problems, though both are NP-hard. If you are lucky, you can solve an NP-complete problem quickly by guessing the answer and verifying its correctness. For example, to solve the NP-complete IP feasibility problem just defined, you could luckily guess a feasible vector v and quickly verify that $Av \leq b$ and that v is integer.

However, as far as is known, co-NP-complete problems cannot be solved quickly by good guessing. If an IP instance is infeasible, there isn't in general a way to prove infeasibility short of trying an exponential number of possibilities and showing that each fails. Even a cutting-plane proof is apt to be exponentially long. Chvatal (1973) demonstrates that finding a cutting-plane proof can be identical to finding an exact solution using branch and bound.

The class co-NP-complete gives us insight into how IP codes spend their time. It is commonplace for an IP solver code to find an optimal solution in a few minutes and then spend a few hours verifying optimality. What is happening? Let v^* be the optimal solution value. The IP solver is solving two problems. First, it solves the NP-complete problem of finding an x with objective value as good as v^*. Second, it solves the co-NP-complete problem of verifying that there isn't anything better than v^*. (Question: What no instance is being solved?) Complexity theory tells us that even if we hot-start our IP code with the optimal solution, it still can take a long time to verify optimality, because any proof of optimality may have to be exponentially long.

At present, we are better at heuristically solving hard searching problems (NP-complete) than at heuristically solving hard verification problems (co-NP-complete). If you require proofs of optimality, you are likely to add significantly to your computing requirements.

You can also use co-NP-completeness theory to show the hopelessness of a quest for a simple characterization. For example, researchers spent a couple of decades seeking concise necessary and sufficient conditions for the existence of a core in spatial voting, but the co-NP-completeness of the problem implies that

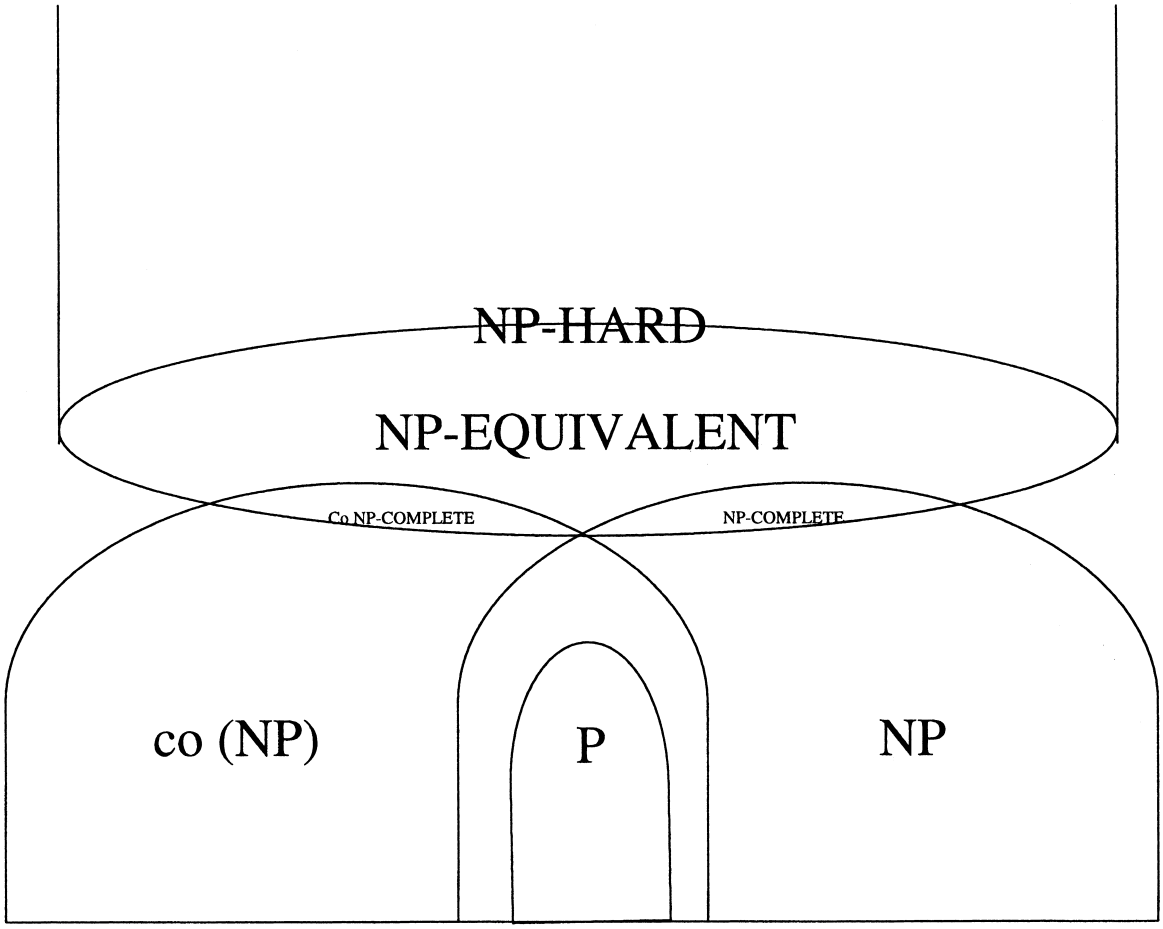

Figure 17: The co-NP-complete problems are complements of the NP-complete problems. Both classes are within the class of NP-equivalent problems, which is within the class of NP-hard problems.

this search was doomed from the start (Bartholdi et al. 1991). Coxson (1994) showed that attempts to find a concise test for *P*-matrices were similarly doomed.

Definitions of NP, Co-NP, and NP-complete

A string is a finite sequence of 0's and 1's. Any instance of any of our problems can be represented as a string. The set of all possible strings is {0, 1}.* A yes-no problem $L \subseteq \{0, 1\}^*$ is a set of strings that consists of all cases for which the answer is yes. For example, the problem of whether an integer is even is the set of strings whose rightmost character is 0. IP feasibility is the set of all strings representing a matrix-vector pair (A, b) such that $Ax \leq b$ for some integer vector x.

A computer program recognizes a problem L if, given a string $s \in \{0, 1\}^*$ as input, it outputs yes if $s \in L$ and outputs no if $s \notin L$. The class P of easy problems (§1) takes its name from "polynomial time." P is the set of all problems L for which there exists a computer program that recognizes L and runs in time polynomial in $|s|$, the length of the input string. Since a problem is itself a set of strings, we often refer to P as a class of problems rather than as a set of problems to avoid confusion between sets and sets of sets.

The complement of a problem L is denoted by $\bar{L} = \{0, 1\}^* - L$. If $L \in P$ then also $\bar{L} \in P$. (Question: Why must this be true?) The complement of the class P would be denoted $co(P)$; it consists of the complements of all problems in P. Given the above question, $co(P) = P$, so we don't use the term $co(P)$.

Now we define NP, which, contrary to popular belief, does not stand for "not polynomial." It means

"nondeterministic polynomial," because if you allow lucky guesswork, then the problems can be solved in polynomial time. A certificate-checking computer program C to *nondeterministically recognize* a language L takes two distinct strings as input, s and a "certificate" string c. The program C outputs yes or maybe. If $s \notin L$, then no matter what c is, C outputs maybe. On the other hand, if $s \in L$, then for at least one certificate string c, C outputs yes. NP is the set of all problems L for which there is a certificate-checking computer program that nondeterministically recognizes L and runs in time polynomially bounded in $|s|$. This implies $|c|$ is polynomially bounded in $|s|$, for otherwise the program would not have time to read all of c.

Determining that a number is composite (not prime) is a good example of a problem in NP. It might take you a long time to factor a number s, but once you have done so, the certificate c can be a divisor of s, and it is quick for C to check that c does divide s.

The class co-NP is the set of problems that are complements of problems in NP. As far as is known, co-NP is not the same as NP. Intuitively, this should make sense if you think about composite numbers. Just because there is a quick way to demonstrate that a number is composite doesn't mean there is a quick way to demonstrate that a number is prime. (It happens that there is.) Perhaps a better example is that no one has ever found a quick way to demonstrate that an IP is infeasible, even though there obviously is a quick way to demonstrate that an IP is feasible.

A problem $L1$ is NP-complete if (i) $L1 \in NP$, and (ii) every problem $L \in NP$ can be transformed to $L1$ in polynomial time, as defined in §3. In this tutorial, I have focused on (ii) because that property causes problems to be hard. Membership in NP is rarely important in practice.

The NP-complete problems are the hardest problems in NP in this sense: if you had a fast solution method for an NP-complete problem, you could solve any problem in NP quickly. (Question: How?) The co-NP-complete problems are similarly defined as the hardest problems in co-NP. They turn out to be precisely the complements of the NP-complete problems. They are NP-hard: if you had a fast solution method for a co-NP-complete problem, you could solve any problem in NP quickly.

P-space Completeness

There are other classes of problems that are widely believed to be harder than both the NP-complete and the co-NP-complete. The most important of these is the class of *P*-space complete problems, the hardest problems of those that can be solved using a polynomial amount of computer memory. Usually, if a problem is *P*-space complete, it will be harder to solve in practice than an NP-complete problem. The best-known exception to this rule is the problem of finding the Lin-Kernighan local optimum from a given starting point.

P-space complete problems include several natural questions about electronic circuits, such as the problem of finding a minimum size circuit to instantiate a given Boolean function. You can sometimes tell you are dealing with a problem of this type when it seems to involve an alternating sequence of difficult decisions between two opposing sides. Many games, including chess and checkers when generalized suitably to size $n \times n$ boards, are in this category. Other things that seem to make problems *P*-space complete are the presence of feedback, periodicity, or stochasticity.

Several important OR/MS problems are *P*-space complete. They include stochastic scheduling, periodic scheduling, Markov decision problems, queueing networks, and systems of differential equations. Several motion-planning problems for jointed robot arms are *P*-space complete as well (Latombe 1991).

In general, adding stochasticity or uncertainty makes hard problems a whole level harder. The classic illustration of this phenomenon is the problem of project planning under uncertainty. This problem motivated Dantzig (1963) to invent linear programming in the 1940s. Since that time, people have used LP to solve countless other problems, but only in the last decade have people reported real success in solving the original motivating problem.

Other Categories

Which problems can be solved faster with parallel processors, and which problems are inherently sequential, so that parallel computing does not help? We can try to use computational complexity to answer these questions by studying such problem classes as NC. Roughly speaking, the class NC is a subset of the easy

problems that can be solved very quickly, in polylogarithmic time ($O((\log n)^k)$ for some k), on a polynomial number of parallel processors.

At present, however, this portion of complexity theory is not useful to the practitioner. For one thing, only rarely used network software provides even $O(n)$ parallelism for real instance sizes. The main issue is that the distinctions made by the theory do not carry over to practice. Many problems that do not seem to be in NC nonetheless benefit quite handsomely from the degree of parallelism available now. For example, theory predicts that parallelism cannot help us solve IPs (or even LPs) much faster. But in practice, parallelism is helpful. A branch-and-bound code for k parallel processors solves IPs about k times faster than a single processor.

Computational complexity theory has little to say about continuous nonlinear problems. In practice, convexity is usually the dividing line between easy and hard (global) optimization, though nonsmoothness can make things difficult as well. The analog to determining whether the problem is easy or NP-hard is to determine whether or not the problem has multiple local optima. The answer roughly dictates what kind of solution method to use or solution quality to expect. However, when we apply NP-completeness theory to this domain, we find that problems thought of as simple in practice are theoretically difficult, for example, determining whether a point is a local minimum of a quadratic function. Thus traditional NP-completeness theory has not been particularly useful.

A problem can be hard in a noncomputational sense. It is not NP-hard for a human to run a three-minute mile. On a visit to a manufacturing site, I was told that the firm had a really hard problem assigning tools among the four machines in a production line. I thought the problem was hard because equalizing workload to avoid bottlenecks is a line-balancing problem, and the firm had enough machines to make dynamic programming impractical (§8). I was completely wrong! It turned out that, because of differences among the machines, the firm had no choice as to which tools to assign to each machine. The decision-making part of the problem was trivial. The problem was hard because one of the machines did not have enough capacity for its tools.

Real problems are often difficult because they are fuzzy and open-ended. Computational complexity theory does not explain this type of difficulty, because it presumes that the problem to be analyzed is well posed.

Answers:

—If we are minimizing, with integer coefficients, the instance is, given A, b, c, v^*, is there integer x such that $Ax \leq b$ and $c \cdot x \leq v^* - 1$?

—Take the computer program that recognizes L and change yes to no and vice-versa in the output statements.

—Attach the transformation to $L1$ as a front end.

10. Conclusions and Recommended Reading

Garey and Johnson's (1979) book is the deservedly classic reference. Papadimitriou and Steiglitz's (1982) book is excellent and is designed for readers who are familiar with the basics of optimization, rather than with computer science. For theoretical background, Sipser's (1997) book is a fine readable choice.

On the most fundamental level, understanding the basics of computational complexity should make you more conscious of how much time your solution methods require and how large your problem instances are. Your choice of algorithm is often more important than your choice of hardware or data structures.

On the next level, classifying problems as easy or NP-hard will tell what kinds of solution methods may be available. Deciding on an appropriate method depends on many other considerations, including size, financial stake, model accuracy, and time constraints.

Computational complexity is an ineluctable phenomemon. Many problems are hard to solve. Yet, I would set against that statement the following assertion: realistic cases are almost always easy if you study them long enough.

Study the realistic cases carefully enough, and you will find enough special structure to enable their solution. If we count modeling as part of the study process, this accounts for much of what we do in OR/MS. Here is another reason to understand computational complexity: improving your intuitive judgment of

hard and easy problems will make you a more effective modeler.

However, all is not rosy even if you can solve any problem if you study it enough. For one thing, it may not be worth doing. For another, what often happens when you give the user control of a model and solution procedure is that he runs them until they die. There are always things the model does not know about that the user does. The user tries to push the model into taking care of these extras until the solution procedure fails.

What happens is that the problem's inherent computational complexity resurges. You beat NP-hardness by knowing your problem really well and finding a solution procedure that is effective within a small zone. The user pushes the model out of that zone to a place where the procedure is ineffective.

It can be very tough explaining this to a user who thinks, "I only made a slight change and now the model doesn't run." People who believe the computer is magic may not believe in its limitations. Sometimes you can convince your users that a problem is hard by excitedly telling them what a fascinating research topic it makes.

One way or the other we are brought back to the reality of the inherent computational difficulty of problems. Experience tells us that most real problems are hard. Computational complexity then tells us that we cannot build tools that have all of the following properties: they solve realistic models, they require little expertise to use, they find an exactly optimum solution, they are always fast, and they apply to a broad range of cases.

We can't have all of these properties. What tradeoffs are we making or should we make? How we compromise on these issues gives us insight into the differences between practitioners and theoreticians and among other groups in our field.

I would like our professional community to have a discussion about tools. An ideal tool would have all the above qualities, but that is not possible. The tool user must make compromises with the complexity of reality; the tool developer must make compromises with the reality of complexity. But are the tool developers making compromises that are effective for the tool users, and are researchers exploring possibilities for more effective compromises? Perhaps researchers could have more impact on practice if they kept these different trade-offs in mind.

Acknowledgments

I am indebted to John Bartholdi, George Dantzig, Steve Hackman, George Nemhauser, Christos Papadimitriou, Don Ratliff, Adam Rosenberg, Richard Stone, and many others for teaching and thoughtful discussions about complexity and problem solving. I also owe many thanks to an exceptionally patient editor, Terry Harrison, and an exceptionally thoughtful and blunt anonymous referee. Mary Haight, the managing editor, encouraged me to add vigor without sacrificing rigor. David Tovey drafted the figures. Finally, I could not have completed the tutorial without the many perceptive suggestions of Dan Adelman, Gina Bisaha, Greg Grace, Bev Hutchinson, Dale Kolosna, Tianji Jiang, Sameer Mahajan, Ken Moorman, David O'Brien, Richard West, Chad Young, and my other students in CS 6155 at Georgia Tech.

References

Bartholdi, J., L. Narasimhan, C. Tovey. 1991. Recognizing equilibria in spatial voting games. *Social Choice and Welfare* **8**(3) 183–197.

Bramel, J., D. Simchi-Levi. 1995. A location based heuristic for general routing problems. *Oper. Res.* **43**(4) 649–660.

Brockett, R. W. 1991. Dynamical systems that sort lists, diagonalize matrices and solve linear programming problems. *Linear Algebra Appl.* **146**(1) 79–91.

Canny, J., J. Reif. 1987. New lower bound techniques for robot motion planning problems. *Proc. 28th Annual IEEE Sympos. in Foundations of Comput. Sci.* Los Angeles, CA, 49–60.

Carter, M., C. Tovey. 1992. When is the classroom assignment problem hard? *Oper. Res.* **40** (Supplement No. 1) S28–S39.

Chandra, B., H. Karloff, C. Tovey. 1999. New results on the old *k*-opt algorithm for the traveling salesman problem. *SIAM J. Comput.* **28**(6) 1998–2029.

Chernoff, H. 1975. Lecture on statistics, 18–304. Massachusetts Institute of Technology, Cambridge, MA.

Chvatal, V. 1973. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Math.* **4**(3) 305–337.

Coxson, G. 1994. The *P*-matrix problem is co-NP-complete. *Math. Programming* **64A**(2) 173–178.

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

Garey, M., D. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, CA.

Geoffrion, A. 1987. An introduction to structured modeling. *Management Sci.* **33**(5) 547–588.

——. 1996. Structured modeling. S. Gass, C. Harris, eds. *Encyclopedia of Operations Research and Management Science*. Kluwer Academic Publishers, Norwell, MA.

Goemans, M., D. Williamson. 1995. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM* **42**(4) 1113–1145.

Hackman, S., R. Leachman. 1989. A general framework for modeling production. *Management Sci.* **35**(4) 478–495.

Håstad, J. 1997. Some optimal inapproximability results. *Proc. 29th Annual ACM Sympos. on Theory of Comput. STOC '97*. 1–10.

Karloff, H., U. Zwick. 1997. A 7/8 approximation algorithm for MAX 3SAT. *Proc. 38th Annual IEEE Symposi. in Foundations of Comput. Sci.* 406–415.

Knuth, D. 1973. *Fundamental Algorithms*. Addison-Wesley, Reading, MA.

Latombe, Jean-Claude. 1991. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA.

Lofgren, C. 1986. Machine configuration of flexible printed circuit board assembly systems. Ph.D. thesis, School of ISyE, Georgia Institute of Technology, Atlanta, GA.

McConnell, J. V. 1989. *Understanding Human Behavior*, 6th ed. Holt, Rinehart, and Winston, New York.

Papadimitriou, C. 1994. *Computational Complexity*. Addison-Wesley, Reading, MA.

——, K. Steiglitz. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ.

Rosenberg, Adam. 1993. Conversation with author.

Sipser, M. 1997. *Introduction to the Theory of Computation*. PWS Publishing, Cambridge, MA.

Tovey, C. 1984. A simplified NP-complete satisfiability problem. *Discrete Appl. Math.* **8**(1) 85–89.

——. 1993. Lecture on heuristics in advanced topics course CS8113. Georgia Institute of Technology, Atlanta, GA.