

Algorithms and Complexity

Herbert S. Wilf
University of Pennsylvania
Philadelphia, PA 19104-6395

Copyright Notice

Copyright 1994 by Herbert S. Wilf. This material may be reproduced for any educational purpose, multiple copies may be made for classes, etc. Charges, if any, for reproduced copies must be just enough to recover reasonable costs of reproduction. Reproduction for commercial purposes is prohibited. This cover page must be included in all distributed copies.

Internet Edition, Summer, 1994

This edition of Algorithms and Complexity is available at the web site <<http://www/cis.upenn.edu/wilf>>. It may be taken at no charge by all interested persons. Comments and corrections are welcome, and should be sent to wilf@math.upenn.edu

Chapter 5: NP-completeness

5.1 Introduction

In the previous chapter we met two computational problems for which fast algorithms have never been found, but neither have such algorithms been proved to be unattainable. Those were the primality-testing problem, for which the best-known algorithm is delicately poised on the brink of polynomial time, and the integer-factoring problem, for which the known algorithms are in a more primitive condition.

In this chapter we will meet a large family of such problems (hundreds of them now!). This family is not just a list of seemingly difficult computational problems. It is in fact bound together by strong structural ties. The collection of problems, called the *NP-complete* problems, includes many well known and important questions in discrete mathematics, such as the following.

The travelling salesman problem ('TSP'): Given n points in the plane ('cities'), and a distance D . Is there a tour that visits all n of the cities, returns to its starting point, and has total length $\leq D$?

Graph coloring: Given a graph G and an integer K . Can the vertices of G be properly colored in K or fewer colors?

Independent set: Given a graph G and an integer K . Does $V(G)$ contain an independent set of K vertices?

Bin packing: Given a finite set S of positive integers, and an integer N (the number of bins). Does there exist a partition of S into N or fewer subsets such that the sum of the integers in each subset is $\leq K$? In other words, can we 'pack' the integers of S into at most N 'bins,' where the 'capacity' of each bin is K ?

These are very difficult computational problems. Take the graph coloring problem, for instance. We could try every possible way of coloring the vertices of G in K colors to see if any of them work. There are K^n such possibilities, if G has n vertices. Hence a very large amount of computation will be done, enough so that if G has 50 vertices and we have 10 colors at our disposal, the problem would lie far beyond the capabilities of the fastest computers that are now available.

Hard problems can have easy instances. If the graph G happens to have no edges at all, or very few of them, then it will be very easy to find out if a coloring is possible, or if an independent set of K vertices is present.

The real question is this (let's use 'Independent Set' as an illustration). Is it possible to design an algorithm that will come packaged with a performance guarantee of the following kind:

The seller warrants that if a graph G , of n vertices, and a positive integer K are input to this program, then it will correctly determine if there is an independent set of K or more vertices in $V(G)$, and it will do so in an amount of time that is at most $1000n^8$ minutes.

Hence there is no contradiction between the facts that the problem is hard and that there are easy cases. The hardness of the problem stems from the seeming impossibility of producing such an algorithm accompanied by such a manufacturer's warranty card. Of course the ' $1000n^8$ ' didn't have to be exactly that. But some quite specific polynomial in the length of the input bit string must appear in the performance guarantee. Hence ' $357n^9$ ' might have appeared in the guarantee, and so might ' $23n^3$,' but ' n^K ' would not be allowed.

Let's look carefully at why n^K would not be an acceptable worst-case polynomial time performance bound. In the 'Independent Set' problem the input must describe the graph G and the integer K . How many bits are needed to do that? The graph can be specified, for example, by its vertex adjacency matrix A . This is an $n \times n$ matrix in which the entry in row i and column j is 1 if $(i, j) \in E(G)$ and is 0 else.

Evidently n^2 bits of input will describe the matrix A . The integers K and n can be entered with just $O(\log n)$ bits, so the entire input bit string for the 'Independent Set' problem is $\sim n^2$ bits long. Let B denote the number of bits in the input string. Suppose that on the warranty card the program was guaranteed to run in a time that is $< n^K$.

Is this a guarantee of polynomial time performance? That question means 'Is there a polynomial P such that for every instance of 'Independent Set' the running time T will be at most $P(B)$?' Well, is T bounded

by a polynomial in B if $T = n^K$ and $B \sim n^2$? It would seem so; in fact obviously $T = O(B^{K/2})$, and that's a polynomial, isn't it?

The key point resides in the order of the qualifiers. We must give the polynomial that works for every instance of the problem *first*. Then that one single polynomial must work on every instance. If the 'polynomial' that we give is $B^{K/2}$, well that's a different polynomial in B for different instances of the problem, because K is different for different instances. Therefore if we say that a certain program for 'Independent Set' will always get an answer before $B^{K/2}$ minutes, where B is the length of the input bit string, then we would not have provided a polynomial-time guarantee in the form of a single polynomial in B that applies uniformly to all problem instances.

The distinction is a little thorny, but is worthy of careful study because it's of fundamental importance. What we are discussing is usually called a *worst-case* time bound, meaning a bound on the running time that applies to every instance of the problem. Worst-case time bounds aren't the only possible interesting ones. Sometimes we might not care if an algorithm is occasionally very slow as long as it is *almost always fast*. In other situations we might be satisfied with an algorithm that is fast *on average*. For the present, however, we will stick to the worst-case time bounds and study some of the theory that applies to that situation. In sections 5.6 and 5.7 we will study some average time bounds.

Now let's return to the properties of the NP-complete family of problems. Here are some of them.

- 1⁰: The problems all seem to be computationally very difficult, and no polynomial time algorithms have been found for any of them.
- 2⁰: It has not been proved that polynomial time algorithms for these problems do not exist.
- 2⁰: But this is not just a random list of hard problems. *If a fast algorithm could be found for one NP-complete problem then here would be fast algorithms for all of them.*
- 2⁰: Conversely, if it could be proved that no fast algorithm exists for one of the NP-complete problems, then there could not be a fast algorithm for any other of those problems.

The above properties are not intended to be a *definition* of the concept of NP-completeness. We'll get to that later on in this section. They are intended as a list of some of the interesting features of these problems, which, when coupled with their theoretical and practical importance, accounts for the intense worldwide research effort that has gone into understanding them in recent years.

The question of the existence or nonexistence of polynomial-time algorithms for the NP-complete problems probably rates as the principal unsolved problem that faces theoretical computer science today.

Our next task will be to develop the formal machinery that will permit us to give precise definitions of all of the concepts that are needed. In the remainder of this section we will discuss the additional ideas informally, and then in section 5.2 we'll state them quite precisely.

What is a decision problem? First, the idea of a *decision problem*. A decision problem is one that asks only for a yes-or-no answer: Can this graph be 5-colored? Is there a tour of length ≤ 15 miles? Is there a set of 67 independent vertices?

Many of them problems that we are studying can be phrased as decision problems or as *optimization problems*: What is the smallest number of colors with which G can be colored? What is the length of the shortest tour of these cities? What is the size of the largest independent set of vertices in G ?

Usually if we find a fast algorithm for a decision problem then with just a little more work we will be able to solve the corresponding optimization problem. For instance, suppose we have an algorithm that solves the decision problem for graph coloring, and what we want is the solution of the optimization problem (the chromatic number).

Let a graph G be given, say of 100 vertices. Ask: can the graph be 50-colored? If so, then the chromatic number lies between 1 and 50. Then ask if it can be colored in 25 colors. If not, then the chromatic number lies between 26 and 50. Continue in this way, using bisection of the interval that is known to contain the chromatic number. After $O(\log n)$ steps we will have found the chromatic number of a graph of n vertices. The extra multiplicative factor of $\log n$ will not alter the polynomial vs. nonpolynomial running time distinction. Hence if there is a fast way to do the decision problem then there is a fast way to do the optimization problem. The converse is obvious.

Hence we will restrict our discussion to decision problems.

What is a language?

Since every decision problem can have only the two answers ‘Y/N,’ we can think of a decision problem as asking if a given word (the input string) does or does not belong to a certain *language*. The language is the totality of words for which the answer is ‘Y.’

The graph 3-coloring language, for instance, is the set of all symmetric, square matrices of 0,1 entries, with zeroes on the main diagonal (these are the vertex adjacency matrices of graphs) such that the graph that the matrix represents is 3-colorable. We can imagine that somewhere there is a vast dictionary of all of the words in this language. A 3-colorability computation is therefore nothing but an attempt to discover whether a given word belongs to the dictionary.

What is the class P?

We say that a decision problem belongs to the class P if there is an algorithm \mathcal{A} and a number c such that for every instance I of the problem the algorithm \mathcal{A} will produce a solution in time $O(B^c)$, where B is the number of bits in the input string that represents I .

To put it more briefly, P is the set of easy decision problems.

Examples of problems in P are most of the ones that we have already met in this book: Are these two integers relatively prime? Is this integer divisible by that one? Is this graph 2-colorable? Is there a flow of value greater than K in this network? Can this graph be disconnected by the removal of K or fewer edges? Is there a matching of more than K edges in this bipartite graph? For each of these problems there is a fast (polynomial time) algorithm.

What is the class NP?

The class NP is a little more subtle. A decision problem Q belongs to NP if there is an algorithm \mathcal{A} that does the following:

- (a) Associated with each word of the language Q (*i.e.*, with each instance I for which the answer is ‘Yes’) there is a *certificate* $C(I)$ such that when the pair $(I, C(I))$ are input to algorithm \mathcal{A} it recognizes that I belongs to the language Q .
- (b) If I is some word that does not belong to the language Q then there is no choice of certificate $C(I)$ that will cause \mathcal{A} to recognize I as a member of Q .
- (c) Algorithm \mathcal{A} operates in polynomial time.

To put this one more briefly, NP is the class of decision problems for which it is easy to *check* the correctness of a claimed answer, with the aid of a little extra information. So we aren’t asking for a way to *find* a solution, but only to *verify* that an alleged solution really is correct.

Here is an analogy that may help to clarify the distinction between the classes P and NP. We have all had the experience of reading through a truly ingenious and difficult proof of some mathematical theorem, and wondering how the person who found the proof in the first place ever did it. Our task, as a reader, was only to *verify* the proof, and that is a much easier job than the mathematician who invented the proof had. To pursue the analogy a bit farther, some proofs are extremely time consuming even to check (see the proof of the four-color theorem!), and similarly, some computational problems are not even known to belong to NP, let alone to P.

In P are the problems where it’s easy to *find* a solution, and in NP are the problems where it’s easy to *check* a solution that may have been very tedious to find.

Here’s another example. Consider the graph coloring problem to be the decision problem Q . Certainly this problem is not known to be in P. It is, however, in NP, and here is an algorithm, and a method of constructing certificates that proves it.

Suppose G is some graph that *is* K -colorable. The certificate of G might be a list of the colors that get assigned to each vertex in some proper K -coloring of the vertices of G . Where did we get that list, you ask? Well, we never said it was easy to construct a certificate. If you actually want to find one then you will have to solve a hard problem. But we’re really only talking about *checking* the correctness of an alleged answer. To *check* that a certain graph G really is K -colorable we can be convinced if you will show us the color of each vertex in a proper K -coloring.

If you do provide that certificate, then our checking algorithm \mathcal{A} is very simple. It checks first that every vertex has a color and only one color. It then checks that no more than K colors have been used altogether.

It finally checks that for each edge e of G it is true that the two endpoints of e have different colors.

Hence the graph coloring problem belongs to NP.

For the travelling salesman problem we would provide a certificate that contains a tour, whose total length is $\leq K$, of all of the cities. The checking algorithm \mathcal{A} would then verify that the tour really does visit all of the cities and really does have total length $\leq K$.

The travelling salesman problem, therefore, also belongs to NP.

‘Well,’ you might reply, ‘if we’re allowed to look at the answers, how could a problem fail to belong to NP?’

Try this decision problem: an instance I of the problem consists of a set of n cities in the plane and a positive number K . The question is ‘Is it true that there is *not* a tour of all of these cities whose total length is less than K ?’ Clearly this is a kind of a negation of the travelling salesman problem. Does it belong to NP? If so, there must be an algorithm \mathcal{A} and a way of making a certificate $C(I)$ for each instance I such that we can quickly verify that *no such tour exists* of the given cities. Any suggestions for the certificate? The algorithm? No one else knows how to do this either.

It is not known if this negation of the travelling salesman problem belongs to NP.

Are there problems that *do* belong to NP but for which it isn’t immediately obvious that this is so? Yes. In fact that’s one of the main reasons that we studied the algorithm of Pratt, in section 4.10. Pratt’s algorithm is exactly a method of producing a certificate with the aid of which we can quickly check that a given integer is prime. The decision problem ‘Given n , is it prime?’ is thereby revealed to belong to NP, although that fact wasn’t obvious at a glance.

It is very clear that $P \subseteq NP$. Indeed if $Q \in P$ is some decision problem then we can verify membership in the language Q with the empty certificate. That is, we don’t even need a certificate in order to do a quick calculation that checks membership in the language because the problem itself can be quickly solved.

It seems natural to suppose that NP is larger than P. That is, one might presume that there are problems whose solutions can be quickly checked with the aid of a certificate even though they can’t be quickly found in the first place.

No example of such a problem has ever been produced (and proved), nor has it been proved that no such problem exists. The question of whether or not $P=NP$ is the one that we cited earlier as being perhaps the most important open question in the subject area today.

It is fairly obvious that the class P is called ‘the class P’ because ‘P’ is the first letter of ‘Polynomial Time.’ But what does ‘NP’ stand for? Stay tuned. The answer will appear in section 5.2.

What is reducibility?

Suppose that we want to solve a system of 100 simultaneous linear equations in 100 unknowns, of the form $Ax = b$. We run down to the local software emporium and quickly purchase a program for \$49.95 that solves such systems. When we get home and read the fine print on the label we discover, to our chagrin, that the system works only on systems where the matrix A is symmetric, and the coefficient matrix in the system that we want to solve is, of course, not symmetric.

One possible response to this predicament would be to look for the solution to the system $A^T Ax = A^T b$, in which the coefficient matrix $A^T A$ is now symmetric.

What we would have done would be to have *reduced* the problem that we really are interested in to an instance of a problem for which we have an algorithm.

More generally, let Q and Q' be two decision problems. We will say that Q' is *quickly reducible to* Q if whenever we are given an instance I' of the problem Q' we can convert it, with only a polynomial amount of labor, into an instance I of Q , in such a way that I' and I both have the same answer (‘Yes’ or ‘No’).

Thus if we buy a program to solve Q , then we can use it to solve Q' , with just a small amount of extra work.

What is NP-completeness?

How would you like to buy one program, for \$49.95, that can solve 500 different kinds of problems? That’s what NP-completeness is about.

To state it a little more carefully, *a decision problem is NP-complete if it belongs to NP and every problem in NP is quickly reducible to it.*

The implications of NP-completeness are numerous. Suppose we could prove that a certain decision problem Q is NP-complete. Then we could concentrate our efforts to find polynomial-time algorithms on just that one problem Q . Indeed if we were to succeed in finding a polynomial time algorithm to do instances of Q then we would automatically have found a fast algorithm for doing every problem in NP. How does that work?

Take an instance I' of some problem Q' in NP. Since Q' is quickly reducible to Q we could transform the instance I' into an instance I of Q . Then use the super algorithm that we found for problems in Q to decide I . Altogether only a polynomial amount of time will have been used from start to finish.

Let's be more specific. Suppose that tomorrow morning we prove that the graph coloring problem is NP-complete, and that on the next morning you find a fast algorithm for solving it. Then consider some instance of the bin packing problem. Since graph coloring is NP-complete, the instance of bin packing can be quickly converted into an instance of graph coloring for which the 'Yes/No' answer is the same. Now use the fast graph coloring algorithm that you found (congratulations, by the way!) on the converted problem. The answer you get is the correct answer for the original bin packing problem.

So, a fast algorithm for some NP-complete problem implies a fast algorithm for every problem in NP. Conversely suppose we can prove that it is impossible to find a fast algorithm for some particular problem Q in NP. Then we can't find a fast algorithm for any NP-complete problem Q' either. For if we could then we would be able to solve instances of Q by quickly reducing them to instances of Q' and solving them.

If we could prove that there is no fast way to test the primality of a given integer then we would have proved that there is no fast way to decide if graphs are K -colorable, because, as we will see, the graph coloring problem is NP-complete and primality testing is in NP. Think about that one for a few moments, and the extraordinary beauty and structural unity of these computational problems will begin to reveal itself.

To summarize: quick for one NP-complete problem implies quick for all of NP; provably slow for one problem in NP implies provably slow for all NP-complete problems.

There's just one small detail to attend to. We've been discussing the economic advantages of keeping flocks of unicorns instead of sheep. If there aren't any unicorns then the discussion is a little silly.

NP-complete problems have all sorts of marvellous properties. It's lovely that every problem in NP can be quickly reduced to just that one NP-complete problem. *But are there any NP-complete problems?* Why, after all, should there be a single computational problem with the property that every one of the diverse creatures that inhabit NP should be quickly reducible to it?

Well, there *are* NP-complete problems, hordes of them, and proving that will occupy our attention for the next two sections. Here's the plan.

In section 5.2 we are going to talk about a simple computer, called a Turing machine. It is an idealized computer, and its purpose is to standardize ideas of computability and time of computation by referring all problems to the one standard machine.

A Turing machine is an extremely simple finite-state computer, and when it performs a computation, a unit of computational labor will be very clearly and unambiguously describable. It turns out that the important aspects of polynomial time computability do not depend on the particular computer that is chosen as the model. The beauty of the Turing machine is that it is at once a strong enough concept that it can in principle perform any calculation that any other finite state machine can do, while at the same time it is logically clean and simple enough to be useful for proving theorems about complexity.

The microcomputer on your desktop *might* have been chosen as the standard against which polynomial time computability is measured. If that had been done then the class P of quickly solvable problems would scarcely have changed at all (the polynomials would be different but they would still be polynomials), but the *proofs* that we humans would have to give in order to establish the relevant theorems would have gotten much more complicated because of the variety of different kinds of states that modern computers have.

Next, in section 5.3 we will prove that there *is* an NP-complete problem. It is called *the satisfiability problem*. Its status as an NP-complete problem was established by S. Cook in 1971, and from that work all later progress in the field has flowed. The proof uses the theory of Turing machines.

The first NP-complete problem was the hardest one to find. We will find, in section 5.4, a few more NP-complete problems, so the reader will get some idea of the methods that are used in identifying them.

Since nobody knows a fast way to solve these problems, various methods have been developed that give approximate solutions quickly, or that give exact solutions in fast *average* time, and so forth. The beautiful

book of Garey and Johnson (see references at the end of the chapter) calls this ‘coping with NP-completeness,’ and we will spend the rest of this chapter discussing some of these ideas.

Exercises for section 5.1

1. Prove that the following decision problem belongs to P: Given integers K and a_1, \dots, a_n . Is the median of the a 's smaller than K ?
2. Prove that the following decision problem is in NP: given an $n \times n$ matrix A of integer entries. Is $\det A = 0$?
3. For which of the following problems can you prove membership in P?
 - (a) Given a graph G . Does G contain a circuit of length 4?
 - (b) Given a graph G . Is G bipartite?
 - (c) Given n integers. Is there a subset of them whose sum is an even number?
 - (d) Given n integers. Is there a subset of them whose sum is divisible by 3?
 - (e) Given a graph G . Does G contain an Euler circuit?
4. For which of the following problems can you prove membership in NP?
 - (a) Given a set of integers and another integer K . Is there a subset of the given integers whose sum is K ?
 - (b) Given a graph G and an integer K . Does G contain a path of length $\geq K$?
 - (c) Given a set of K integers. Is it true that not all of them are prime?
 - (d) Given a set of K integers. Is it true that all of them are prime?

5.2 Turing Machines

A Turing machine consists of

- (a) a doubly infinite *tape*, that is marked off into *squares* that are numbered as shown in Fig. 5.2.1 below. Each square can contain a single character from the character set that the machine recognizes. For simplicity we can assume that the character set contains just three symbols: ‘0,’ ‘1,’ and ‘ ’ (blank).
- (b) a *tape head* that is capable of either reading a single character from a square on the tape or writing a single character on a square, or moving its position relative to the tape by an increment of one square in either direction.
- (c) a finite list of *states* such that at every instant the machine is in exactly one of those states. The possible states of the machine are, first of all, the regular states q_1, \dots, q_s , and second, three *special states*
 - q_0 : the initial state
 - q_Y , the final state in a problem to which the answer is ‘Yes’
 - q_N : the final state in a problem to which the answer is ‘No’
- (d) a *program* (or *program module*, if we think of it as a pluggable component) that directs the machine through the steps of a particular task.

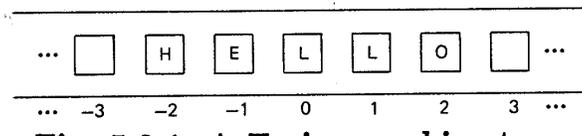


Fig. 5.2.1: A Turing machine tape

Let’s describe the program module in more detail. Suppose that at a certain instant the machine is in state q (other than q_Y or q_N) and that the symbol that has just been read from the tape is ‘*symbol*.’ Then from the pair (q, \textit{symbol}) the program module will decide

- (i) to what state q' the machine shall next go, and
- (ii) what single character the machine will now write on the tape in the square over which the head is now positioned, and
- (iii) whether the tape head will next move one square to the right or one square to the left.

One step of the program, therefore, goes from

$$(state, symbol) \text{ to } (newstate, newsymbol, increment). \quad (5.2.1)$$

If and when the state reaches q_Y or q_N the computation is over and the machine halts.

The machine should be thought of as part hardware and part software. The programmer's job is, as usual, to write the software. To write a program for a Turing machine, what we have to do is to tell it how to make each and every one of the transitions (5.2.1). A Turing machine program looks like a table in which, for every possible pair $(state, symbol)$ that the machine might find itself in, the programmer has specified what the *newstate*, the *newsymbol* and the *increment* shall be.

To begin a computation with a Turing machine we take the input string x , of length B , say, that describes the problem that we want to solve, and we write x in squares $1, 2, \dots, B$ of the tape. The tape head is then positioned over square 1, the machine is put into state q_0 , the program module that the programmer prepared is plugged into its slot, and the computation begins.

The machine reads the symbol in square 1. It now is in state q_0 and has read **symbol**, so it can consult the program module to find out what to do. The program instructs it to write at square 1 a **newsymbol**, to move the head either to square 0 or to square 2, and to enter a certain **newstate**, say q' . The whole process is then repeated, possibly forever, but hopefully after finitely many steps the machine will enter the state q_Y or state q_N , at which moment the computation will halt with the decision having been made.

If we want to watch a Turing machine in operation, we don't have to build it. We can simulate one. Here is a pidgin-Pascal simulation of a Turing machine that can easily be turned into a functioning program. It is in two principal parts.

The procedure **turmach** has for input a string x of length B , and for output it sets the Boolean variable **accept** to **True** or **False**, depending on whether the outcome of the computation is that the machine halted in state q_Y or q_N respectively. This procedure is the 'hardware' part of the Turing machine. It doesn't vary from one job to the next.

Procedure **gonextto** is the program module of the machine, and it will be different for each task. Its inputs are the present **state** of the machine and the **symbol** that was just read from the tape. Its outputs are the **newstate** into which the machine goes next, the **newsymbol** that the tape head now writes on the current square, and the **increment** (± 1) by which the tape head will now move.

```

procedure turmach(B:integer; x :array[1..B]; accept:Boolean);
{simulates Turing machine action on input string x of length B}
{write input string on tape in first B squares}
  for square := 1 to B do
    tape[square] :=x[square];
  {record boundaries of written-on part of tape}
  leftmost:=1; rightmost := B;
  {initialize tape head and state}
  state:=0; square:=1;
  while state  $\neq$  'Y' and state  $\neq$  'N' do
    {read symbol at current tape square}
    if square < leftmost or square > rightmost
      then symbol:= ' ' else symbol:= tape[square]
    {ask program module for state transition}
    gonextto(state,symbol,newstate,newsymbol,increment);
    state:=newstate;
    {update boundaries and write new symbol};
    if square > rightmost then leftmost:= square;
    tape[square] :=newsymbol;
    {move tape head}
    square := square+increment
  end;{while}
  accept:={ state='Y'}
end.{turmach}

```

Now let's try to write a particular program module `gonextto`. Consider the following problem: given an input string x , consisting of 0's and 1's, of length B . Find out if it is true that the string contains an odd number of 1's.

We will write a program that will scan the input string from left to right, and at each moment the machine will be in state 0 if it has so far scanned an even number of 1's, in state 1 otherwise. In Fig. 5.2.2 we show a program that will get the job done.

state	symbol	newstate	newsymbol	increment
0	0	0	0	+1
0	1	1	1	+1
0	blank	q_N	blank	-1
1	0	1	0	+1
1	1	0	1	+1
1	blank	q_Y	blank	-1

Fig. 5.2.2: A Turing machine program for bit parity

Exercise. Program the above as procedure `gonextto`, run it for some input string, and print out the state of the machine, the contents of the tape, and the position of the tape head after each step of the computation.

In the next section we are going to use the Turing machine concept to prove Cook's theorem, which is the assertion that a certain problem is NP-complete. Right now let's review some of the ideas that have already been introduced from the point of view of Turing machines.

We might immediately notice that some terms that were just a little bit fuzzy before are now much more sharply in focus. Take the notion of polynomial time, for example. To make that idea precise one needs a careful definition of what 'the length of the input bit string' means, and what one means by the number of 'steps' in a computation.

But on a Turing machine both of these ideas come through with crystal clarity. The input bit string x is what we write on the tape to get things started, and its length is the number of tape squares it occupies. A 'step' in a Turing machine calculation is obviously a single call to the program module. A Turing machine calculation was done 'in time $P(B)$ ' if the input string occupied B tape squares and the calculation took $P(B)$ steps.

Another word that we have been using without ever nailing down precisely is 'algorithm.' We all understand informally what an algorithm is. But now we understand formally too. An algorithm for a problem is a program module for a Turing machine that will cause the machine to halt after finitely many steps in state 'Y' for every instance whose answer is 'Yes,' and after finitely many steps in state 'N' for every instance whose answer is 'No.'

A Turing machine and an algorithm define a *language*. The language is the set of all input strings x that lead to termination in state 'Y,' i.e., to an *accepting* calculation.

Now let's see how the idea of a Turing machine can clarify the description of the class NP. This is the class of problems for which the decisions can be made quickly if the input strings are accompanied by suitable certificates.

By a *certificate* we mean a finite strip of Turing machine tape, consisting of 0 or more squares, each of which contains a symbol from the character set of the machine. A certificate can be loaded into a Turing machine as follows. If the certificate contains $m > 0$ tape squares, then replace the segment from square number $-m$ to square number -1 , inclusive, of the Turing machine tape with the certificate. The information on the certificate is then available to the program module just as any other information on the tape is available.

To use a Turing machine as a *checking* or *verifying* computer, we place the input string x that describes the problem instance in squares $1, 2, \dots, B$ of the tape, and we place the certificate $C(x)$ of x in squares $-m, -m + 1, \dots, -1$ of the tape. We then write a verifying program for the program module in which the program verifies that the string x is indeed a word in the language of the machine, and in the course of the verification the program is quite free to examine the certificate as well as the problem instance.

A Turing machine that is being used as a *verifying* computer is called a *nondeterministic* machine. The hardware is the same, but the manner of input and the question that is being asked are different from the

situation with a *deterministic* Turing machine, in which we decide whether or not the input string is in the language, without using any certificates.

The class NP (‘Nondeterministic Polynomial’) consists of those decision problems for which there exists a fast (polynomial time) algorithm that will verify, given a problem instance string x and a suitable certificate $C(x)$, that x belongs to the language recognized by the machine, and for which, if x does not belong to the language, *no* certificate would cause an accepting computation to ensue.

5.3 Cook’s Theorem

The NP-complete problems are the hardest problems in NP, in the sense that if Q' is any decision problem in NP and Q is an NP-complete problem, then every instance of Q' is polynomially reducible to an instance of Q . As we have already remarked, the surprising thing is that there is an NP-complete problem at all, since it is not immediately clear why any single problem should hold the key to the polynomial time solvability of every problem in the class NP. But there is one. As soon as we see why there is one, then we’ll be able to see more easily why there are hundreds of them, including many computational questions about discrete structures such as graphs, networks and games and about optimization problems, about algebraic structures, formal logic, and so forth.

Here is the *satisfiability problem*, the first problem that was proved to be NP-complete, by Stephen Cook in 1971.

We begin with a list of (Boolean) variables x_1, \dots, x_n . A *literal* is either one of the variables x_i or the negation of one of the variables, as \bar{x}_i . There are $2n$ possible literals.

A *clause* is a set of literals.

The rules of the game are these. We assign the value ‘True’ (T) or ‘False’ (F), to each one of the *variables*. Having done that, each one of the *literals* inherits a truth value, namely a literal x_i has the same truth or falsity as the corresponding variable x_i , and a literal \bar{x}_i has the opposite truth value from that of the variable x_i .

Finally each of the clauses also inherits a truth value from this process, and it is determined as follows. A clause has the value ‘T’ if and only if *at least one* of the literals in that clause has the value ‘T,’ and otherwise it has the value ‘F.’

Hence starting with an assignment of truth values to the variables, some true and some false, we end up with a determination of the truth values of each of the clauses, some true and some false.

Definition. *A set of clauses is satisfiable if there exists an assignment of truth values to the variables that makes all of the clauses true.*

Think of the word ‘or’ as being between each of the literals in a clause, and the word ‘and’ as being between the clauses.

The satisfiability problem (SAT). *Given a set of clauses. Does there exist a set of truth values (=T or F), one for each variable, such that every clause contains at least one literal whose value is T (i.e., such that every clause is satisfied)?*

Example: Consider the set x_1, x_2, x_3 of variables. From these we might manufacture the following list of four clauses:

$$\{x_1, \bar{x}_2\}, \quad \{x_1, x_3\}, \quad \{x_2, \bar{x}_3\}, \quad \{\bar{x}_1, x_3\}.$$

If we choose the truth values (T, T, F) for the variables, respectively, then the four clauses would acquire the truth values (T, T, T, F) , and so this would not be a *satisfying* truth assignment for the set of clauses. There are only eight possible ways to assign truth values to three variables, and after a little more experimentation we might find out that these clauses would in fact be satisfied if we were to make the assignments (T, T, T) (how can we recognize a set of clauses that is satisfied by assigning to every variable the value ‘T’?). ■

The example already leaves one with the feeling that SAT might be a tough computational problem, because there are 2^n possible sets of truth values that we might have to explore if we were to do an exhaustive search.

It is quite clear, however, that this problem belongs to NP. Indeed, it is a decision problem. Furthermore we can easily assign a certificate to every set of clauses for which the answer to SAT is ‘Yes, the clauses

are satisfiable.' The certificate contains a set of truth values, one for each variable, that satisfy all of the clauses. A Turing machine that receives the set of clauses, suitably encoded, as input, along with the above certificate, would have to verify only that if the truth values are assigned to the variables as shown on the certificate then indeed every clause does contain at least one literal of value 'T.' That verification is certainly a polynomial time computation.

Now comes the hard part. We want to show

Theorem 5.3.1. (*S. Cook, 1971*): *SAT is NP-complete.*

Before we carry out the proof, it may be helpful to give a small example of the reducibility ideas that we are going to use.

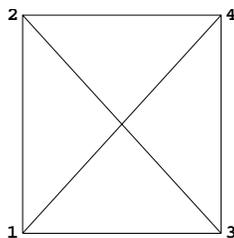


Fig. 5.3.1: A 3-coloring problem

Example. Reducing graph-coloring to SAT

Consider the graph G of four vertices that is shown in Fig. 5.3.1, and the decision problem 'Can the vertices of G be properly colored in 3 colors?'

Let's see how that decision problem can be reduced to an instance of SAT. We will use 12 Boolean variables: the variable $x_{i,j}$ corresponds to the assertion that 'vertex i has been colored in color j ' ($i = 1, 2, 3, 4; j = 1, 2, 3$).

The instance of SAT that we construct has 31 clauses. The first 16 of these are

$$\begin{aligned}
 C(i) &:= \{x_{i,1}, x_{i,2}, x_{i,3}\} & (i = 1, 2, 3, 4) \\
 T(i) &:= \{\bar{x}_{i,1}, \bar{x}_{i,2}\} & (i = 1, 2, 3, 4) \\
 U(i) &:= \{\bar{x}_{i,1}, \bar{x}_{i,3}\} & (i = 1, 2, 3, 4) \\
 V(i) &:= \{\bar{x}_{i,2}, \bar{x}_{i,3}\} & (i = 1, 2, 3, 4).
 \end{aligned} \tag{5.3.1}$$

In the above, the four clauses $C(i)$ assert that each vertex has been colored in at least one color. The clauses $T(i)$ say that no vertex has both color 1 and color 2. Similarly the clauses $U(i)$ (*resp.* $V(i)$) guarantee that no vertex has been colored 1 and 3 (*resp.* 2 and 3).

All 16 of the clauses in (5.3.1) together amount to the statement that 'each vertex has been colored in one and only one of the three available colors.'

Next we have to construct the clauses that will assure us that the two endpoints of an edge of the graph are never the same color. For this purpose we define, for each edge e of the graph G and color j ($=1,2,3$), a clause $D(e, j)$ as follows. Let u and v be the two endpoints of e ; the $D(e, j) := \{\bar{x}_{u,j}, \bar{x}_{v,j}\}$, which asserts that not both endpoints of the edge e have the same color j .

The original instance of the graph coloring problem has now been reduced to an instance of SAT. In more detail, *there exists an assignment of values T, F to the 12 Boolean variables $x_{1,1}, \dots, x_{4,3}$ such that each of the 31 clauses contains at least one literal whose value is T if and only if the vertices of the graph G can be properly colored in three colors.* The graph is 3-colorable if and only if the clauses are satisfiable. ■

It is clear that if we have an algorithm that will solve SAT, then we can also solve graph coloring problems. A few moments of thought will convince the reader that the transformation of one problem to the other that was carried out above involves only a polynomial amount of computation, despite the seemingly large number of variables and clauses. Hence graph coloring is quickly reducible to SAT.

Proof of Cook's theorem

We want to prove that SAT is NP-complete, *i.e.*, that every problem in NP is polynomially reducible to an instance of SAT. Hence let Q be some problem in NP and let I be an instance of problem Q . Since Q is in NP there exists a Turing machine that recognizes encoded instances of problem Q , if accompanied by a suitable certificate, in polynomial time.

Let TMQ be such a Turing machine, and let $P(n)$ be a polynomial in its argument n with the property that TMQ recognizes every pair $(x, C(x))$, where x is a word in the language Q and $C(x)$ is its certificate, in time $\leq P(n)$, where n is the length of x .

We intend to construct, corresponding to each word I in the language Q , and instance $f(I)$ of SAT for which the answer is 'Yes, the clauses are all simultaneously satisfiable.' Conversely, if the word I is not in the language Q , the clauses will not be satisfiable.

The idea can be summarized like this: *the instance of SAT that will be constructed will be a collection of clauses that together express the fact that there exists a certificate that causes Turing machine TMQ to do an accepting calculation.* Therefore, in order to test whether or not the word Q belongs to the language, it suffices to check that the collection of clauses is satisfiable.

To construct an instance of SAT means that we are going to define a number of variables, of literals, and of clauses, in such a way that the clauses are satisfiable if and only if x is in the language Q , *i.e.*, the machine TMQ accepts x and its certificate.

What we must do, then, is to express the accepting computation of the Turing machine as the simultaneous satisfaction of a number of logical propositions. It is precisely here that the relative simplicity of a Turing machine allows us to enumerate all of the possible paths to an accepting computation in a way that would be quite unthinkable with a 'real' computer.

Now we will describe the Boolean variables that will be used in the clauses under construction.

Variable $Q_{i,k}$ is true if after step i of the checking calculation it is true that the Turing machine TMQ is in state q_k , false otherwise.

Variable $S_{i,j,a} = \{\text{after step } i, \text{ symbol } a \text{ is in tape square } j\}$.

Variable $T_{i,j} = \{\text{after step } i, \text{ the tape head is positioned over square } j\}$.

Let's count the variables that we've just introduced. Since the Turing machine TMQ does its accepting calculation in time $\leq P(n)$ it follows that the tape head will never venture more than $\pm P(n)$ squares away from its starting position. Therefore the subscript j , which runs through the various tape squares that are scanned during the computation, can assume only $O(P(n))$ different values.

Index a runs over the letters in the alphabet that the machine can read, so it can assume at most some fixed number A of values.

The index i runs over the steps of the accepting computation, and so it takes at most $O(P(n))$ different values.

Finally, k indexes the states of the Turing machine, and there is only some fixed finite number, K , say, of states that TMQ might be in. Hence there are altogether $O(P(n)^2)$ variables, a polynomial number of them.

Is it true that every random assignment of true or false values to each of these variables corresponds to an accepting computation on $(x, C(x))$? Certainly not. For example, if we aren't careful we might assign true values to $T_{9,4}$ and to $T_{10,33}$, thereby burning out the bearings on the tape transport mechanism! (why?)

Our remaining task, then, will be to describe precisely the conditions under which a set of values assigned to the variables listed above actually defines a possible accepting calculation for $(x, C(x))$. Then we will be sure that whatever set of satisfying values of the variables might be found by solving the SAT problem, they will determine a real accepting calculation of the machine TMQ.

This will be done by requiring that a number of clauses be all true ('satisfied') at once, where each clause will express one necessary condition. In the following, the bold face type will describe, in words, the condition that we want to express, and it will be followed by the formal set of clauses that actually expresses the condition on input to SAT.

At each step, the machine is in at least one state.

Hence at least one of the K available state variables must be true. This leads to the first set of clauses,

At step $P(n)$ the machine is in state q_Y .

one for each step i of the computation:

$$\{Q_{i,1}, Q_{i,2}, \dots, Q_{i,K}\}$$

Since i assumes $O(P(n))$ values, these are $O(P(n))$ clauses.

At each step, the machine is not in more than one state

Therefore, for each step i , and each pair j', j'' of distinct states, the clause

$$\{\bar{Q}_{i,j'}, \bar{Q}_{i,j''}\}$$

must be true. These are $O(P(n))$ additional clauses to add to the list, but still more are needed.

At each step, each tape square contains exactly one symbol from the alphabet of the machine.

This leads to two lists of clauses which require, first, that there is at least one symbol in each square at each step, and second, that there are not two symbols in each square at each step. The clauses that do this are

$$\{S_{i,j,1}, S_{i,j,2}, \dots, S_{i,j,A}\}$$

where A is the number of letters in the machine's alphabet, and

$$\{\bar{S}_{i,j,k'}, \bar{S}_{i,j,k''}\}$$

for each step i , square j , and pair k', k'' of distinct symbols in the alphabet of the machine.

The reader will by now have gotten the idea of how to construct the clauses, so for the next three categories we will simply list the functions that must be performed by the corresponding lists of clauses, and leave the construction of the clauses as an exercise.

At each step, the tape head is positioned over a single square.

Initially the machine is in state 0, the head is over square 1, the input string x is in squares 1 to n , and $C(x)$ (the input certificate of x) is in squares 0, -1, ..., $-P(n)$.

At step $P(n)$ the machine is in state q_Y .

The last set of restrictions is a little trickier:

At each step the machine moves to its next configuration (state, symbol, head position) in accordance with the application of its program module to its previous (state, symbol).

To find the clauses that will do this job, consider first the following condition: the symbol in square j of the tape cannot change during step i of the computation if the tape head isn't positioned there at that moment. This translates into the collection

$$\{T_{i,j}, \bar{S}_{i,j,k}, S_{i+1,j,k}\}$$

of clauses, one for each triple $(i, j, k) = (\text{state, square, symbol})$. These clauses express the condition in the following way: either (at time i) the tape head is positioned over square j ($T_{i,j}$ is true) or else the head is not positioned there, in which case either symbol k is not in the j th square before the step or else symbol k is (still) in the j th square after the step is executed.

It remains to express the fact that the transitions from one configuration of the machine to the next are the direct results of the operation of the program module. The three sets of clauses that do this are

$$\begin{aligned} &\{\bar{T}_{i,j}, \bar{Q}_{i,k}, \bar{S}_{i,j,l}, \bar{T}_{i+1,j+INC}\} \\ &\{\bar{T}_{i,j}, \bar{Q}_{i,k}, \bar{S}_{i,j,l}, Q_{i+1,k'}\} \\ &\{\bar{T}_{i,j}, \bar{Q}_{i,k}, \bar{S}_{i,j,l}, S_{i+1,j,l'}\}. \end{aligned}$$

In each case the format of the clause is this: ‘either the tape head is not positioned at square j , or the present state is not q_k or the symbol just read is not l , but if they are then ...’ There is a clause as above for each step $i = 0, \dots, P(n)$ of the computation, for each square $j = -P(n), P(n)$ of the tape, for each symbol l in the alphabet, and for each possible state q_k of the machine, a polynomial number of clauses in all. The new configuration triple (INC, k', l') is, of course, as computed by the program module.

Now we have constructed a set of clauses with the following property. If we execute a recognizing computation on a string x and its certificate, in time at most $P(n)$, then this computation determines a set of (True, False) values for all of the variables listed above, in such a way that all of the clauses just constructed are simultaneously satisfied.

Conversely if we have a set of values of the SAT variables that satisfy all of the clauses at once, then that set of values of the variables describes a certificate that would cause TMQ to do a computation that would recognize the string x and it also describes, in minute detail, the ensuing accepting computation that TMQ would do if it were given x and that certificate.

Hence every language in NP can be reduced to SAT. It is not difficult to check through the above construction and prove that the reduction is accomplishable in polynomial time. It follows that SAT is NP-complete. ■

5.4 Some other NP-complete problems

Cook’s theorem opened the way to the identification of a large number of NP-complete problems. The proof that Satisfiability is NP-complete required a demonstration that every problem in NP is polynomially reducible to SAT. To prove that some other problem X is NP-complete it will be sufficient to prove that SAT reduces to problem X . For if that is so then every problem in NP can be reduced to problem X by first reducing to an instance of SAT and then to an instance of X .

In other words, life after Cook’s theorem is a lot easier. To prove that some problem is NP-complete we need show only that SAT reduces to it. We don’t have to go all the way back to the Turing machine computations any more. Just prove that if you can solve your problem then you can solve SAT. By Cook’s theorem you will then know that by solving your problem you will have solved every problem in NP.

For the honor of being ‘the second NP-complete problem,’ consider the following special case of SAT, called *3-satisfiability*, or 3SAT. An instance of 3SAT consists of a number of clauses, just as in SAT, except that the clauses are permitted to contain no more than three literals each. The question, as in SAT, is ‘Are the clauses simultaneously satisfiable by some assignment of T, F values to the variables?’

Interestingly, though, the general problem SAT is reducible to the apparently more special problem 3SAT, which will show us

Theorem 5.4.1. *3-satisfiability is NP-complete.*

Proof. Let an instance of SAT be given. We will show how to transform it quickly to an instance of 3SAT that is satisfiable if and only if the original SAT problem was satisfiable.

More precisely, we are going to replace clauses that contain more than three literals with collections of clauses that contain exactly three literals and that have the same satisfiability as the original. In fact, suppose our instance of SAT contains a clause

$$\{x_1, x_2, \dots, x_k\} \quad (k \geq 4). \quad (5.4.1)$$

Then this clause will be replaced by $k - 2$ new clauses, utilizing $k - 3$ new variables z_i ($i = 1, \dots, k - 3$) that are introduced just for this purpose. The $k - 2$ new clauses are

$$\{x_1, x_2, z_1\}, \{x_3, \bar{z}_1, z_2\}, \{x_4, \bar{z}_2, z_3\}, \dots, \{x_{k-1}, x_k, \bar{z}_{k-3}\}. \quad (5.4.2)$$

We now make the following

Claim. If x_1^*, \dots, x_k^* is an assignment of truth values to the x ’s for which the clause (5.4.1) is true, then there exist assignments z_1^*, \dots, z_{k-3}^* of truth values to the z ’s such that all of the clauses (5.4.2) are simultaneously satisfied by (x^*, z^*) . Conversely, if (x^*, z^*) is some assignment that satisfies all of (5.4.2), then x^* alone satisfies (5.4.1).

To prove the claim, first suppose that (5.4.1) is satisfied by some assignment x^* . Then one, at least, of the k literals x_1, \dots, x_k , say x_r , has the value ‘T.’ Then we can satisfy all $k - 2$ of the transformed clauses (5.4.2) by assigning $z_s^* := ‘T’$ for $s \leq r - 2$ and $z_s^* = ‘F’$ for $s > r - 2$. It is easy to check that each one of the $k - 2$ new clauses is satisfied.

Conversely, suppose that all of the new clauses are satisfied by some assignment of truth values to the x ’s and the z ’s. We will show that at least one of the x ’s must be ‘True,’ so that the original clause will be satisfied.

Suppose, to the contrary, that all of the x ’s are false. Since, in the new clauses none of the x ’s are negated, the fact that the new clauses are satisfied tells us that they would remain satisfied without any of the x ’s. Hence the clauses

$$\{z_1\}, \{\bar{z}_1, z_2\}, \{\bar{z}_2, z_3\}, \dots, \{\bar{z}_{k-4}, z_{k-3}\}, \{\bar{z}_{k-3}\}$$

are satisfied by the values of the z ’s. If we scan the list from left to right we discover, in turn, that z_1 is true, z_2 is true, \dots , and finally, much to our surprise, that z_{k-3} is true, and z_{k-3} is also false, a contradiction which establishes the truth of the claim made above.

The observation that the transformations just discussed can be carried out in polynomial time completes the proof of theorem 5.4.1. ■

We remark, in passing, that the problem ‘2SAT’ is in P.

Our collection of NP-complete problems is growing. Now we have two, and a third is on the way. We will show next how to reduce 3SAT to a graph coloring problem, thereby proving

Theorem 5.4.2. *The graph vertex coloring problem is NP-complete.*

Proof: Given an instance of 3SAT, that is to say, given a collection of k clauses, involving n variables and having at most three literals per clause, we will construct, in polynomial time, a graph G with the property that its vertices can be properly colored in $n + 1$ colors if and only if the given clauses are satisfiable. We will assume that $n > 4$, the contrary case being trivial.

The graph G will have $3n + k$ vertices:

$$\{x_1, \dots, x_n\}, \{\bar{x}_1, \dots, \bar{x}_n\}, \{y_1, \dots, y_n\}, \{C_1, \dots, C_k\}$$

Now we will describe the set of edges of G . First each vertex x_i is joined to \bar{x}_i ($i = 1, \dots, n$). Next, every vertex y_i is joined to every other vertex y_j ($j \neq i$), to every other vertex x_j ($j \neq i$), and to every vertex \bar{x}_j ($j \neq i$).

Vertex x_i is connected to C_j if x_i is *not* one of the literals in clause C_j . Finally, \bar{x}_i is connected to C_j if \bar{x}_i is not one of the literals in C_j .

May we interrupt the proceedings to say again why we’re doing all of this? You have just read the description of a certain graph G . The graph is one that can be drawn as soon as someone hands us a 3SAT problem. We described the graph by listing its vertices and then listing its edges. What does the graph do for us?

Well suppose that we have just bought a computer program that can decide if graphs are colorable in a given number of colors. We paid \$ 49.95 for it, and we’d like to use it. But the first problem that needs solving happens to be a 3SAT problem, not a graph coloring problem. We aren’t so easily discouraged, though. We convert the 3SAT problem into a graph that is $(n + 1)$ -colorable if and only if the original 3SAT problem was satisfiable. Now we can get our money’s worth by running the graph coloring program even though what we really wanted to do was to solve a 3SAT problem.

In Fig. 5.4.1 we show the graph G of 11 vertices that corresponds to the following instance of 3SAT:

$$C_1 := \{x_1, \bar{x}_2\}; \quad C_2 := \{x_1, x_2, \bar{x}_3\}.$$

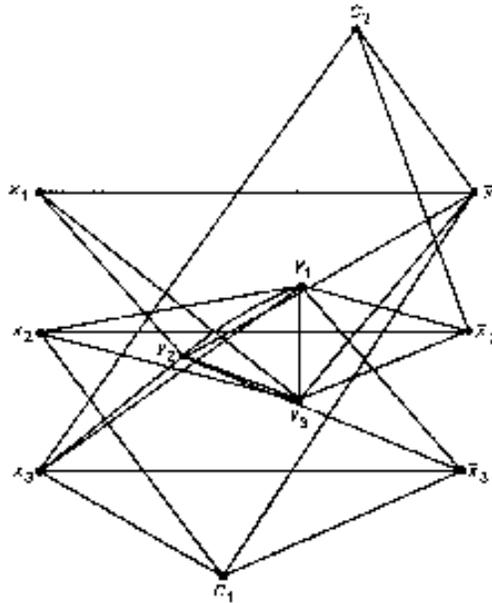


Fig. 5.4.1: The graph for a 3SAT problem

Now we claim that this graph is $n + 1$ colorable if and only if the clauses are satisfiable.

Clearly G cannot be colored in fewer than n colors, because the n vertices y_1, \dots, y_n are all connected to each other and therefore they alone already require n different colors for a proper coloration. Suppose that y_i is assigned color i ($i = 1, \dots, n$).

Do we need new colors in order to color the x_i vertices? Since vertex y_i is connected to every x vertex and every \bar{x} vertex except x_i, \bar{x}_i , if color i is going to be used on the x 's or the \bar{x} 's, it will have to be assigned to one of x_i, \bar{x}_i , but not to both, since they are connected to each other. Hence a new color, color $n + 1$, will have to be introduced in order to color the x 's and \bar{x} 's.

Further, if we are going to color the vertices of G in only $n + 1$ colors, the only way to do it will be to assign color $n + 1$ to exactly one member of each pair (x_i, \bar{x}_i) , and color i to the other one, for each $i = 1, \dots, n$. That one of the pair that gets color $n + 1$ will be called the *False* vertex, the other one is the *True* vertex of the pair (x_i, \bar{x}_i) , for each $i = 1, \dots, n$.

It remains to color the vertices C_1, \dots, C_k . The graph will be $n + 1$ colorable if and only if we can do this without using any new colors. Since each clause contains at most three literals, and $n > 4$, every variable C_i must be adjacent to both x_j and \bar{x}_j for at least one value of j . Therefore no vertex C_i can be colored in the color $n + 1$ in a proper coloring of G , and therefore every C_i must be colored in one of the colors $1, \dots, n$.

Since C_i is connected by an edge to every vertex x_j or \bar{x}_j that is not in the clause C_i , it follows that C_i cannot be colored in the same color as any x_j or \bar{x}_j that is not in the clause C_i .

Hence the color that we assign to C_i must be the same as the color of some 'True' vertex X_j or \bar{x}_j that corresponds to a literal that is in clause C_i . Therefore the graph is $n + 1$ colorable if and only if there is a 'True' vertex for each C_i , and this means exactly that the clauses are satisfiable.

It is easy to verify that the transformation from the 3SAT problem to the graph coloring problem can be carried out in polynomial time, and the proof is finished. ■

By means of many, often quite ingenious, transformations of the kind that we have just seen, the list of NP-complete problems has grown rapidly since the first example, and the 21 additional problems found by R. Karp. Hundreds of such problems are now known. Here are a few of the more important ones.

Maximum clique: We are given a graph G and an integer K . The question is to determine whether or not there is a set of K vertices in G , each of which is joined, by an edge of G , to all of the others.

Edge coloring: Given a graph G and an integer K . Can we color the *edges* of G in K colors, so that whenever two edges meet at a vertex, they will have different colors?

Let us refer to an edge coloring of this kind as a *proper* coloring of the edges of G .

A beautiful theorem of Vizing* deals with this question. If Δ denotes the largest degree of any vertex in the given graph, the Vizing's theorem asserts that the edges of G can be properly colored in either Δ or $\Delta + 1$ colors. Since it is obvious that *at least* Δ colors will be needed, this means that the edge chromatic number is in doubt by only one unit, for every graph G ! Nevertheless the decision as to whether the correct answer is Δ or $\Delta + 1$ is NP-complete.

Hamilton path: In a given graph G , is there a path that visits every vertex of G exactly once?

Target sum: Given a finite set of positive integers whose sum is S . Is there a subset whose sum is $S/2$?

The above list, together with SAT, 3SAT, Travelling Salesman and Graph Coloring, constitutes a modest sampling of the class of these seemingly intractable problems. Of course it must not be assumed that every problem that 'sounds like' an NP-complete problem is necessarily so hard. If for example we ask for an Euler path instead of a Hamilton path (*i.e.*, if we want to traverse *edges* rather than *vertices*) the problem would no longer be NP-complete, and in fact it would be in P, thanks to theorem 1.6.1.

As another example, the fact that one can find the edge connectivity of a given graph in polynomial time (see section 3.8) is rather amazing considering the quite difficult appearance of the problem. One of our motivations for including the network flow algorithms in this book was, indeed, to show how very sophisticated algorithms can sometimes prove that seemingly hard problems are in fact computationally tractable.

Exercises for section 5.4

1. Is the claim that we made and proved above (just after (5.4.2)) identical with the statement that the clause (5.4.1) is satisfiable if and only if the clauses (5.4.2) are simultaneously satisfiable? Discuss.
2. Is the claim that we made and proved above (just after (5.4.2)) identical with the statement that the Boolean expression (5.4.1) is equal to the product of the Boolean expressions (5.4.2) in the sense that their truth values are identical on every set of inputs? Discuss.
3. Let it be desired to find out if a given graph G , of V vertices, can be vertex colored in K colors. If we transform the problem into an instance of 3SAT, exactly how many clauses will there be?

5.5 Half a loaf ...

If we simply *have* to solve an NP-complete problem, then we are faced with a very long computation. Is there anything that can be done to lighten the load? In a number of cases various kinds of probabilistic and approximate algorithms have been developed, some very ingenious, and these may often be quite serviceable, as we have already seen in the case of primality testing. Here are some of the strategies of 'near' solutions that have been developed.

Type I: 'Almost surely ...'

Suppose we have an NP-complete problem that asks if there is a certain kind of substructure embedded inside a given structure. Then we may be able to develop an algorithm with the following properties:

- (a) It always runs in polynomial time
- (b) When it finds a solution then that solution is always a correct one
- (c) It doesn't always find a solution, but it 'almost always' does, in the sense that the ratio of successes to total cases approaches unity as the size of the input string grows large.

An example of such an algorithm is one that will find a Hamilton path in almost all graphs, failing to do so sometimes, but not often, and running always in polynomial time. We will describe such an algorithm below.

* V. G. Vizing, On an estimate of the chromatic class of a p -graph (Russian), *Diskret. Analiz.* **3** (1964), 25-30.

Type II: ‘Usually fast ...’

In this category of quasi-solution are algorithms in which the uncertainty lies not in whether a solution will be found, but in how long it will take to find one. An algorithm of this kind will

- (a) always find a solution and the solution will always be correct, and
- (b) operate in an *average* of subexponential time, although occasionally it may require exponential time. The averaging is over all input strings of a given size.

An example of this sort is an algorithm that will surely find a maximum independent set in a graph, will on the average require ‘only’ $O(n^{c \log n})$ time to do so, but will occasionally, *i.e.*, for some graphs, require nearly 2^n time to get an answer. We will outline such an algorithm below, in section 5.6. Note that $O(n^{c \log n})$ is not a polynomial time estimate, but it’s an improvement over 2^n .

Type II: ‘Usually fast ...’

In this kind of an algorithm we don’t even get the right answer, but it’s close. Since this means giving up quite a bit, people like these algorithms to be very fast. Of course we are going to drop our insistence that the questions be posed as decision problems, and instead they will be asked as optimization problems: find the shortest tour through these cities, or, find the size of the maximum clique in this graph, or, find a coloring of this graph in the fewest possible colors, etc.

In response these algorithms will

- (a) run in polynomial time
- (b) always produce some output
- (c) provide a guarantee that the output will not deviate from the optimal solution by more than such-and-such.

An example of this type is the approximate algorithm for the travelling salesman problem that is given below, in section 5.8. It quickly yields a tour of the cities that is guaranteed to be at most twice as long as the shortest possible tour.

Now let’s look at examples of each of these kinds of approximation algorithms.

An example of an algorithm of Type I is due to Angluin and Valiant. It tries to find a Hamilton path (or circuit) in a graph G . It doesn’t always find such a path, but in theorem 5.5.1 below we will see that it usually does, at least if the graph is from a class of graphs that are likely to have Hamilton paths at all.

Input to the algorithm are the graph G and two distinguished vertices s, t . It looks for a Hamilton path between the vertices s, t (if $s = t$ on input then we are looking for a Hamilton circuit in G).

The procedure maintains a partially constructed Hamilton path P , from s to some vertex ndp , and it attempts to extend P by adjoining an edge to a new, previously unvisited vertex. In the process of doing so it will delete from the graph G , from time to time, an edge, so we will also maintain a variable graph G' , that is initially set to G , but which is acted upon by the program.

To do its job, the algorithm chooses at random an edge (ndp, v) that is incident with the current endpoint of the partial path P , and it deletes the edge (ndp, v) from the graph G' , so it will never be chosen again. If v is a vertex that is not on the path P then the path is extended by adjoining the new edge (ndp, v) .

So much is fairly clear. However if the new vertex v is already on the path P , then we short circuit the path by deleting an edge from it and drawing in a new edge, as is shown below in the formal statement of the algorithm, and in Fig. 5.5.1. In that case the path does not get longer, but it changes so that it now has

enhanced chances of ultimate completion.

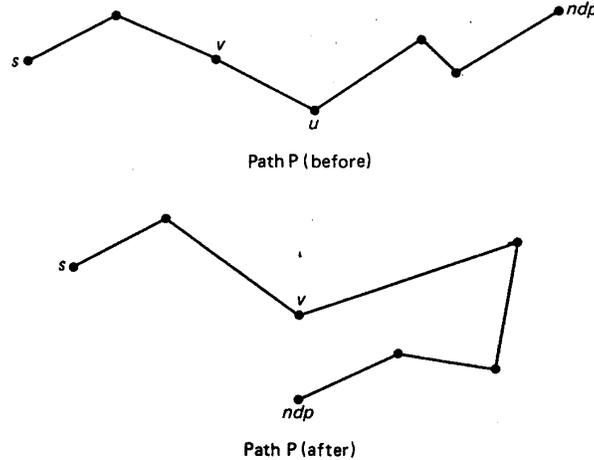


Fig. 5.5.1: The short circuit

Here is a formal statement of the algorithm of Angluin and Valiant for finding a Hamilton path or circuit in an undirected graph G .

```

procedure  $uhc(G:\text{graph}; s, t: \text{vertex})$ ;
{finds a Hamilton path (if  $s \neq t$ ) or a Hamilton
  circuit (if  $s = t$ )  $P$  in an undirected graph  $G$ 
  and returns 'success', or fails, and returns 'failure'}
 $G' := G$ ;  $ndp := s$ ;  $P :=$  empty path;
repeat
if  $ndp$  is an isolated point of  $G'$ 
then return 'failure'
else
  choose uniformly at random an edge  $(ndp, v)$  from
  among the edges of  $G'$  that are incident with  $ndp$ 
  and delete that edge from  $G'$ ;
if  $v \neq t$  and  $v \notin P$ 
then adjoin the edge  $(ndp, v)$  to  $P$ ;  $ndp := v$ 
else
  if  $v \neq t$  and  $v \in P$ 
  then
    {This is the short-circuit of Fig. 5.5.1}
     $u :=$  neighbor of  $v$  in  $P$  that is closer to  $ndp$ ;
    delete edge  $(u, v)$  from  $P$ ;
    adjoin edge  $(ndp, v)$  to  $P$ ;
     $ndp := u$ 
  end; {then}
end {else}
until  $P$  contains every vertex of  $G$  (except  $T$ , if
   $s \neq t$ ) and edge  $(ndp, t)$  is in  $G$  but not in  $G'$ ;
adjoin edge  $(ndp, t)$  to  $P$  and return 'success'
end. {uhc}

```

As stated above, the algorithm makes only a very modest claim: either it succeeds or it fails! Of course what makes it valuable is the accompanying theorem, which asserts that in fact the procedure almost always succeeds, provided the graph G has a good chance of having a Hamilton path or circuit.

What kind of graph has such a ‘good chance’? A great deal of research has gone into the study of how many edges a graph has to have before almost surely it must contain certain given structures. For instance, how many edges must a graph of n vertices have before we can be almost certain that it will contain a complete graph of 4 vertices?

To say that graphs have a property ‘almost certainly’ is to say that the ratio of the number of graphs on n vertices that have the property to the number of graphs on n vertices approaches 1 as n grows without bound.

For the Hamilton path problem, an important dividing line, or threshold, turns out to be at the level of $c \log n$ edges. That is to say, a graph of n vertices that has $o(n \log n)$ edges has relatively little chance of being even connected, whereas a graph with $> cn \log n$ edges is almost certainly connected, and almost certainly has a Hamilton path.

We now state the theorem of Angluin and Valiant, which asserts that the algorithm above will almost surely succeed if the graph G has enough edges.

Theorem 5.5.1. *Fix a positive real number a . There exist numbers M and c such that if we choose a graph G at random from among those of n vertices and at least $cn \log n$ edges, and we choose arbitrary vertices s, t in G , then the probability that algorithm UHC returns ‘success’ before making a total of $Mn \log n$ attempts to extend partially constructed paths is $1 - O(n^{-a})$.*

5.6 Backtracking (I): independent sets

In this section we are going to describe an algorithm that is capable of solving some NP-complete problems fast, *on the average*, while at the same time guaranteeing that a solution will always be found, be it quickly or slowly.

The method is called *backtracking*, and it has long been a standard method in computer search problems when all else fails. It has been common to think of backtracking as a very long process, and indeed it can be. But recently it has been shown that the method can be very fast on average, and that in the graph coloring problem, for instance, it functions in an average of *constant* time, *i.e.*, the time is independent of the number of vertices, although to be sure, the worst-case behavior is very exponential.

We first illustrate the backtrack method in the context of a search for the largest independent set of vertices (a set of vertices no two of which are joined by an edge) in a given graph G , an NP-complete problem. In this case the average time behavior of the method is not constant, or even polynomial, but is subexponential. The method is also easy to analyze and to describe in this case.

Hence consider a graph G of n vertices, in which the vertices have been numbered $1, 2, \dots, n$. We want to find, in G , the size of the largest independent set of vertices. In Fig. 5.6.1 below, the graph G has 6 vertices.

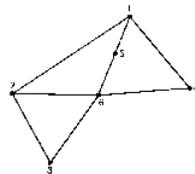


Fig. 5.6.1: Find the largest independent set

Begin by searching for an independent set S that contains vertex 1, so let $S := \{1\}$. Now attempt to enlarge S . We cannot enlarge S by adjoining vertex 2 to it, but we can add vertex 3. Our set S is now $\{1, 3\}$.

Now we cannot adjoin vertex 4 (joined to 1) or vertex 5 (joined to 1) or vertex 6 (joined to 3), so we are stuck. Therefore we backtrack, by replacing the most recently added member of S by the next choice that we might have made for it. In this case, we delete vertex 3 from S , and the next choice would be vertex 6. The set S is $\{1, 6\}$. Again we have a dead end.

If we backtrack again, there are no further choices with which to replace vertex 6, so we backtrack even further, and not only delete 6 from S but also replace vertex 1 by the next possible choice for it, namely vertex 2.

To speed up the discussion, we will now show the list of all sets S that turn up from start to finish of the algorithm:

$$\{1\}, \{13\}, \{16\}, \{2\}, \{24\}, \{245\}, \{25\}, \{3\}, \\ \{34\}, \{345\}, \{35\}, \{4\}, \{45\}, \{5\}, \{6\}$$

A convenient way to represent the search process is by means of the *backtrack search tree* T . This is a tree whose vertices are arranged on levels $L := 0, 1, 2, \dots, n$ for a graph of n vertices. Each vertex of T corresponds to an independent set of vertices in G . Two vertices of T , corresponding to independent sets S', S'' of vertices of G , are joined by an edge in T if $S' \subseteq S''$, and $S'' - S'$ consists of a single element: the highest-numbered vertex in S'' . On level L we find a vertex S of T for every independent set of exactly L vertices of G . Level 0 consists of a single root vertex, corresponding to the empty set of vertices of G .

The complete backtrack search tree for the problem of finding a maximum independent set in the graph G of Fig. 5.6.1 is shown in Fig. 5.6.2 below.

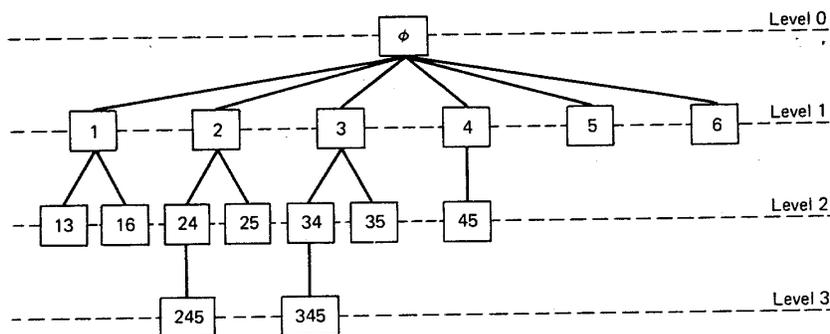


Fig. 5.6.2: The backtrack search tree

The backtrack algorithm amounts just to visiting every vertex of the search tree T , without actually having to write down the tree explicitly, in advance.

Observe that the list of sets S above, or equivalently, the list of nodes of the tree T , consists of exactly every independent set in the graph G . A reasonable measure of the complexity of the searching job, therefore, is the number of independent sets that G has. In the example above, the graph G had 19 independent sets of vertices, including the empty set.

The question of the complexity of backtrack search is therefore the same as the question of determining the number of independent sets of the graph G .

Some graphs have an enormous number of independent sets. The graph \overline{K}_n of n vertices and no edges whatever has 2^n independent sets of vertices. The backtrack tree will have 2^n nodes, and the search will be a long one indeed.

The complete graph K_n of n vertices and every possible edge, $n(n-1)/2$ in all, has just $n+1$ independent sets of vertices.

Any other graph G of n vertices will have a number of independent sets that lies between these two extremes of $n+1$ and 2^n . Sometimes backtracking will take an exponentially long time, and sometimes it will be fairly quick. Now the question is, *on the average* how fast is the backtrack method for this problem?

What we are asking for is the average number of independent sets that a graph of n vertices has. But that is the sum, over all vertex subsets $S \subseteq \{1, \dots, n\}$, of the probability that S is an independent set. If S has k vertices, then the probability that S is independent is the probability that, among the $k(k-1)/2$ possible edges that might join a pair of vertices in S , exactly zero of these edges actually live in the random graph G . Since each of these $\binom{k}{2}$ edges has a probability $1/2$ of appearing in G , the probability that none of them appear is $2^{-k(k-1)/2}$. Hence the average number of independent sets in a graph of n vertices is

$$I_n = \sum_{k=0}^n \binom{n}{k} 2^{-k(k-1)/2}. \quad (5.6.1)$$

Hence in (5.6.1) we have an exact formula for the average number of independent sets in a graph of n vertices. A short table of values of I_n is shown below, in Table 5.6.1, along with values of 2^n , for comparison. Clearly the average number of independent sets in a graph is a lot smaller than the maximum number that graphs of that size might have.

n	I_n	2^n
2	3.5	4
3	5.6	8
4	8.5	16
5	12.3	32
10	52	1024
15	149.8	32768
20	350.6	1048576
30	1342.5	1073741824
40	3862.9	1099511627776

Table 5.6.1: Independent sets and all sets

In the exercises it will be seen that the rate of growth of I_n as n grows large is $O(n^{\log n})$. Hence the average amount of labor in a backtrack search for the largest independent set in a graph grows subexponentially, although faster than polynomially. It is some indication of how hard this problem is that even on the average the amount of labor needed is not of polynomial growth.

Exercises for section 5.6

- What is the average number of independent sets of size k that are in graphs of V vertices and E edges?
- Let t_k denote the k th term in the sum (5.6.1).
 - Show that $t_k/t_{k-1} = (n - k + 1)/(k2^{k+1})$.
 - Show that t_k/t_{k-1} is > 1 when k is small, then is < 1 after k passes a certain critical value k_0 . Hence show that the terms in the sum (5.6.1) increase in size until $k = k_0$ and then decrease.
- Now we will estimate the size of k_0 in the previous problem.
 - Show that $t_k < 1$ when $k = \lfloor \log_2 n \rfloor$ and $t_k > 1$ when $k = \lfloor \log_2 n - \log_2 \log_2 n \rfloor$. Hence the index k_0 of the largest term in (5.6.1) satisfies

$$\lfloor \log_2 n - \log_2 \log_2 n \rfloor \leq k_0 \leq \lfloor \log_2 n \rfloor$$

- The entire sum in (5.6.1) is at most $n + 1$ times as large as its largest single term. Use Stirling's formula (1.1.10) and 3(a) above to show that the k_0 th term is $O((n + \epsilon)^{\log n})$ and therefore the same is true of the whole sum, *i.e.*, of I_n .

5.7 Backtracking (II): graph coloring

In another NP-complete problem, that of graph-coloring, the average amount of labor in a backtrack search is $O(1)$ (bounded) as n , the number of vertices in the graph, grows without bound. More precisely, for fixed K , if we ask 'Is the graph G , of V vertices, properly vertex-colorable in K colors?,' then the average labor in a backtrack search for the answer is bounded. Hence not only is the average of polynomial growth, but the polynomial is of degree 0 (in V).

To be even more specific, consider the case of 3 colors. It is already NP-complete to ask if the vertices of a given graph can be colored in 3 colors. Nevertheless, *the average number of nodes in the backtrack search tree for this problem is about 197*, averaged over all graphs of all sizes. This means that if we input a random graph of 1,000,000 vertices, and ask if it is 3-colorable, then we can expect an answer (probably 'No') after only about 197 steps of computation.

To prove this we will need some preliminary lemmas.

Lemma 5.7.1. *Let s_1, \dots, s_K be nonnegative numbers whose sum is L . Then the sum of their squares is at least L^2/K .*

Proof: We have

$$\begin{aligned} 0 &\leq \sum_{i=1}^K \left(s_i - \frac{L}{K}\right)^2 \\ &= \sum_{i=1}^K \left(s_i^2 - 2\frac{Ls_i}{K} + \frac{L^2}{K^2}\right) \\ &= \sum_{i=1}^K s_i^2 - 2\frac{L^2}{K} + \frac{L^2}{K} \\ &= \sum_{i=1}^K s_i^2 - \frac{L^2}{K}. \blacksquare \end{aligned}$$

The next lemma deals with a kind of inside-out chromatic polynomial question. Instead of asking ‘How many proper colorings can a given graph have?’, we ask ‘How many graphs can have a given proper coloring?’

Lemma 5.7.2. *Let \mathcal{C} be one of the K^L possible ways to color in K colors a set of L abstract vertices $1, 2, \dots, L$. Then the number of graphs G whose vertex set is that set of L colored vertices and for which \mathcal{C} is a proper coloring of G is at most $2^{L^2(1-1/K)/2}$.*

Proof: In the coloring \mathcal{C} , suppose s_1 vertices get color 1, \dots , s_K get color K , where, of course, $s_1 + \dots + s_K = L$. If a graph G is to admit \mathcal{C} as a proper vertex coloring then its edges can be drawn only between vertices of different colors. The number of edges that G might have is therefore

$$s_1s_2 + s_1s_3 + \dots + s_1s_K + s_2s_3 + \dots + s_2s_K + \dots + s_{K-1}s_K$$

for which we have the following estimate:

$$\begin{aligned} \sum_{1 \leq i < j \leq K} s_i s_j &= \frac{1}{2} \sum_{i \neq j} s_i s_j \\ &= \frac{1}{2} \left\{ \sum_{i,j=1}^K s_i s_j - \sum_{i=1}^K s_i^2 \right\} \\ &= \frac{1}{2} (\sum s_i)^2 - \frac{1}{2} \sum s_i^2 \\ &\leq \frac{L^2}{2} - \frac{1}{2} \frac{L^2}{K} \quad (\text{by lemma 5.7.1}) \\ &= \frac{L^2}{2} \left(1 - \frac{1}{K}\right). \end{aligned} \tag{5.7.1}$$

The number of possible graphs G is therefore at most $2^{L^2(1-1/K)/2}$. \blacksquare

Lemma 5.7.3. *The total number of proper colorings in K colors of all graphs of L vertices is at most*

$$K^L 2^{L^2(1-1/K)/2}.$$

Proof: We are counting the pairs (G, \mathcal{C}) , where the graph G has L vertices and \mathcal{C} is a proper coloring of G . If we keep \mathcal{C} fixed and sum on G , then by lemma 5.7.2 the sum is at most $2^{L^2(1-1/K)/2}$. Since there are K^L such \mathcal{C} ’s, the proof is finished. \blacksquare

Now let’s think about a backtrack search for a K -coloring of a graph. Begin by using color 1 on vertex 1. Then use color 1 on vertex 2 unless $(1, 2)$ is an edge, in which case use color 2. As the coloring progresses through vertices $1, 2, \dots, L$ we color each new vertex with the lowest available color number that does not cause a conflict with some vertex that has previously been colored.

At some stage we may reach a dead end: out of colors, but not out of vertices to color. In the graph of Fig. 5.7.1 if we try to 2-color the vertices we can color vertex 1 in color 1, vertex 2 in color 2, vertex 3 in color 1 and then we'd be stuck because neither color would work on vertex 4.

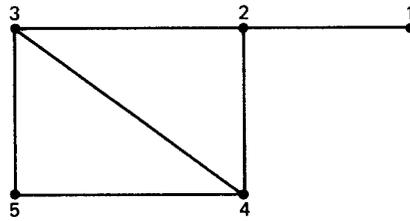


Fig. 5.7.1: Color this graph

When a dead end is reached, back up to the most recently colored vertex for which other color choices are available, replace its color with the next available choice, and try again to push forward to the next vertex.

The (futile) attempt to color the graph in Fig. 5.7.1 with 2 colors by the backtrack method can be portrayed by the *backtrack search tree* in Fig. 5.7.2.

The search is thought of as beginning at 'Root.' The label at each node of the tree describes the colors of the vertices that have so far been colored. Thus '212' means that vertices 1,2,3 have been colored, respectively, in colors 2,1,2.

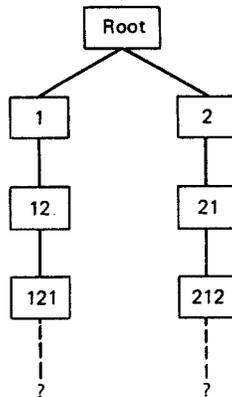


Fig. 5.7.2: A frustrated search tree

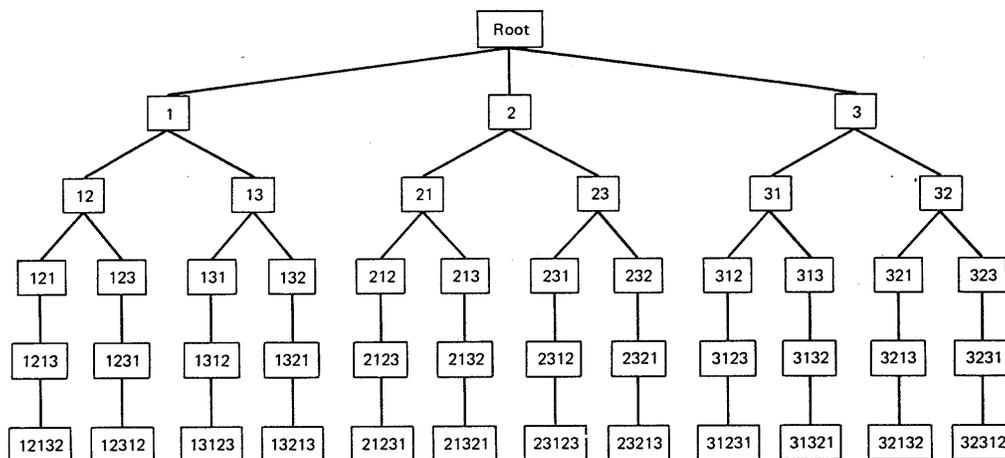


Fig. 5.7.3: A happy search tree

If instead we use 3 colors on the graph of Fig. 5.7.1 then we get a successful coloring; in fact we get 12 of them, as is shown in Fig. 5.7.3.

Let's concentrate on a particular *level* of the search tree. Level 2, for instance, consists of the nodes of the search tree that are at a distance 2 from 'Root.' In Fig. 5.7.3, level 2 contains 6 nodes, corresponding to the partial colorings 12, 13, 21, 23, 31, 32 of the graph. When the coloring reaches vertex 2 it has seen only the portion of the graph G that is induced by vertices 1 and 2.

Generally, a node at level L of the backtrack search tree corresponds to a proper coloring in K colors of the subgraph of G that is induced by vertices $1, 2, \dots, L$.

Let $H_L(G)$ denote that subgraph. Then we see the truth of

Lemma 5.7.4. *The number of nodes at level L of the backtrack search tree for coloring a graph G in K colors is equal to the number of proper colorings of $H_L(G)$ in K colors, i.e., to $P(K, H_L(G))$, where P is the chromatic polynomial. ■*

We are now ready for the main question of this section: what is the average number of nodes in a backtrack search tree for K -coloring graphs of n vertices? This is

$$\begin{aligned} A(n, K) &= \frac{1}{\text{no. of graphs}} \sum_{\text{graphs } G_n} \{\text{no. of nodes in tree for } G\} \\ &= 2^{-\binom{n}{2}} \sum_{G_n} \left\{ \sum_{L=0}^n \{\text{no. of nodes at level } L\} \right\} \\ &= 2^{-\binom{n}{2}} \sum_{G_n} \sum_{L=0}^n P(K, H_L(G)) \quad (\text{by lemma 5.7.4}) \\ &= 2^{-\binom{n}{2}} \sum_{L=0}^n \left\{ \sum_{G_n} P(K, H_L(G)) \right\}. \end{aligned}$$

Fix some value of L and consider the inner sum. As G runs over all graphs of N vertices, $H_L(G)$ selects the subgraph of G that is induced by vertices $1, 2, \dots, L$. Now lots of graphs G of n vertices have the same $H_L(G)$ sitting at vertices $1, 2, \dots, L$. In fact exactly $2^{\binom{n}{2} - \binom{L}{2}}$ different graphs G of n vertices all have the same graph H of L vertices in residence at vertices $1, 2, \dots, L$ (see exercise 15 of section 1.6). Hence (5.7.2) gives

$$\begin{aligned} A(n, K) &= 2^{-\binom{n}{2}} \sum_{L=0}^n 2^{\binom{n}{2} - \binom{L}{2}} \left\{ \sum_{H_L} P(K, H) \right\} \\ &= \sum_{L=0}^n 2^{-\binom{L}{2}} \left\{ \sum_{H_L} P(K, H) \right\}. \end{aligned}$$

The inner sum is exactly the number that is counted by lemma 5.7.3, and so

$$\begin{aligned} A(n, K) &\leq \sum_{L=0}^n 2^{-\binom{L}{2}} K^L 2^{L^2(1-1/K)/2} \\ &\leq \sum_{L=0}^{\infty} K^L 2^{L/2} 2^{-L^2/2K}. \end{aligned}$$

The infinite series actually converges! Hence $A(n, k)$ is bounded, for all n . This proves

Theorem 5.7.1. *Let $A(n, k)$ denote the average number of nodes in the backtrack search trees for K -coloring the vertices of all graphs of n vertices. Then there is a constant $h = h(K)$, that depends on the number of colors, K , but not on n , such that $A(n, k) \leq h(K)$ for all n .*

5.8 Approximate algorithms for hard problems

Finally we come to Type III of the three kinds of ‘half-a-loaf-is-better-than-none’ algorithms that were described in section 5.5. In these algorithms we don’t find the exact solution of the problem, only an approximate one. As consolation we have an algorithm that runs in polynomial time as well as a performance guarantee to the effect that while the answer is approximate, it can certainly deviate by no more than such-and-such from the exact answer.

An elegant example of such a situation is in the Travelling Salesman Problem, which we will now express as an optimization problem rather than as a decision problem.

We are given n points (‘cities’) in the plane, as well as the distances between every pair of them, and we are asked to find a round-trip tour of all of these cities that has minimum length. We will assume throughout the following discussion that the distances satisfy the triangle inequality. This restriction of the TSP is often called the ‘Euclidean’ Travelling Salesman Problem.

The algorithm that we will discuss for this problem has the properties

- (a) it runs in polynomial time and
- (b) the round-trip tour that it finds will never be more than twice as long as the shortest possible tour.

The first step in carrying out the algorithm is to find a minimum spanning tree (MST) for the n given cities. A MST is a tree whose nodes are the cities in question, and which, among all possible trees on that vertex set, has minimum possible length.

It may seem that finding a MST is just as hard as solving the TSP, but NIN (No, It’s Not). The MST problem is one of those all-too-rare computational situations in which it pays to be greedy.

Generally speaking, in a greedy algorithm,

- (i) we are trying to construct some optimal structure by adding one piece at a time, and
- (ii) at each step we make the decision about which piece will be added next by choosing, among all available pieces, the single one that will carry us as far as possible in the desirable direction (be greedy!).

The reason that greedy algorithms are not usually the best possible ones is that it may be better not to take the single best piece at each step, but to take some other piece, in the hope that at a later step we will be able to improve things even more. In other words, the *global* problem of finding the best structure might not be solvable by the *local* procedure of being as greedy as possible at each step.

In the MST problem, though, the greedy strategy works, as we see in the following algorithm.

```

procedure mst( $\mathbf{x}$  :array of  $n$  points in the plane);
{constructs a spanning tree  $T$  of minimum length, on the
 vertices  $\{x_1, \dots, x_n\}$  in the plane}
let  $T$  consist of a single vertex  $x_1$ ;
  while  $T$  has fewer than  $n$  vertices do
    for each vertex  $v$  that is not yet in  $T$ , find the
      distance  $d(v)$  from  $v$  to the nearest vertex of  $T$ ;
    let  $v^*$  be a vertex of smallest  $d(v)$ ;
    adjoin  $v^*$  to the vertex set of  $T$ ;
    adjoin to  $T$  the edge from  $v^*$  to the nearest
      vertex  $w \neq v^*$  of  $T$ ;
  end{while}
end.{mst}

```

Proof of correctness of *mst*: Let T be the tree that is produced by running *mst*, and let e_1, \dots, e_{n-1} be its edges, listed in the same order in which the algorithm *mst* produced them.

Let T' be a minimum spanning tree for \mathbf{x} . Let e_r be the first edge of T that does not appear in T' . In the minimum tree T' , edges e_1, \dots, e_{r-1} all appear, and we let S be the union of their vertex sets. In T' let f be the edge that joins the subtree on S to the subtree on the remaining vertices of \mathbf{x} .

Suppose f is shorter than e_r . Then f was one of the edges that was available to the algorithm *mst* at the instant that it chose e_r , and since e_r was the shortest edge available at that moment, we have a contradiction.

Suppose f is longer than e_r . Then T' would not be minimal because the tree that we would obtain by exchanging f for e_r in T' (why is it still a tree if we do that exchange?) would be shorter, contradicting the minimality of T' .

Hence f and e_r have the same length. In T' exchange f for e_r . Then T' is still a tree, and is still a minimum spanning tree.

The index of the first edge of T that does not appear in T' is now at least $r + 1$, one unit larger than before. The process of replacing edges of T that do not appear in T' without affecting the minimality of T can be repeated until every edge of T appears in T' , *i.e.*, until $T = T'$. Hence T was a minimum spanning tree. ■

That finishes one step of the process that leads to a polynomial time travelling salesman algorithm that finds a tour of at most twice the minimum length.

The next step involves finding an Euler circuit. Way back in theorem 1.6.1 we learned that a connected graph has an Euler circuit if and only if every vertex has even degree. Recall that the proof was recursive in nature, and immediately implies a linear time algorithm for finding Euler circuits recursively. We also noted that the proof remains valid even if we are dealing with a *multigraph*, that is, with a graph in which several edges are permitted between single pairs of vertices. We will in fact need that extra flexibility for the purpose at hand.

Now we have the ingredients for a quick near-optimal travelling salesman tour.

Theorem 5.8.1. *There is an algorithm that operates in polynomial time and which will return a travelling salesman tour whose length is at most twice the length of a minimum tour.*

Here is the algorithm. Given the n cities in the plane:

- (1) Find a minimum spanning tree T for the cities.
- (2) Double each edge of the tree, thereby obtaining a ‘multitree’ $T^{(2)}$ in which between each pair of vertices there are 0 or 2 edges.
- (3) Since every vertex of the doubled tree has even degree, there is an Eulerian tour W of the edges of $T^{(2)}$; find one, as in the proof of theorem 1.6.1.
- (4) Now we construct the output tour of the cities. Begin at some city and follow the walk W . However, having arrived at some vertex v , go from v directly (via a straight line) to the next vertex of the walk W that you haven’t visited yet. This means that you will often *short-circuit* portions of the walk W by going directly from some vertex to another one that is several edges ‘down the road.’

The tour Z' that results from (4) above is indeed a tour of all of the cities in which each city is visited once and only once. We claim that its length is at most twice optimal.

Let Z be an optimum tour, and let e be some edge of Z . Then $Z - e$ is a path that visits all of the cities. Since a path is a tree, $Z - e$ is a spanning tree of the cities, hence $Z - e$ is at least as long as T is, and so Z is surely at least as long as T is.

Next consider the length of the tour Z' . A step of Z' that walks along an edge of the walk W has length equal to the length of that edge of W . A step of Z' that short circuits several edges of W has length at most equal to the sum of the lengths of the edges of W that were short-circuited. If we sum these inequalities over all steps of Z' we find that the length of Z' is at most equal to the length of W , which is in turn twice the length of the tree T .

If we put all of this together we find that

$$\text{length}(Z) > \text{length}(Z - e) \geq \text{length}(T) = \frac{1}{2}\text{length}(W) \geq \frac{1}{2}\text{length}(Z')$$

as claimed (!) ■

More recently it has been proved (Cristofides, 1976) that in polynomial time we can find a TSP tour whose total length is at most $3/2$ as long as the minimum tour. The algorithm makes use of Edmonds’s algorithm for maximum matching in a general graph (see the reference at the end of Chapter 3). It will be interesting to see if the factor $3/2$ can be further refined.

Polynomial time algorithms are known for other NP-complete problems that guarantee that the answer obtained will not exceed, by more than a constant factor, the optimum answer. In some cases the guarantees apply to the *difference* between the answer that the algorithm gives and the best one. See the references below for more information.

Exercises for section 5.8

1. Consider the following algorithm:

```

procedure mst2( $\mathbf{x}$  :array of  $n$  points in the plane);
{allegedly finds a tree of minimum total length that
  visits every one of the given points}
  if  $n = 1$ 
    then  $T := \{x_1\}$ 
    else
       $T := mst2(n - 1, \mathbf{x} - x_n)$ ;
      let  $u$  be the vertex of  $T$  that is nearest to  $x_n$ ;
       $mst2 := T$  plus vertex  $x_n$  plus edge  $(x_n, u)$ 
  end.{mst2}

```

Is this algorithm a correct recursive formulation of the minimum spanning tree greedy algorithm? If so then prove it, and if not then give an example of a set of points where mst2 gets the wrong answer.

Bibliography

Before we list some books and journal articles it should be mentioned that research in the area of NP-completeness is moving rapidly, and the state of the art is changing all the time. Readers who would like updates on the subject are referred to a series of articles that have appeared in issues of the Journal of Algorithms in recent years. These are called ‘NP-completeness: An ongoing guide.’ They are written by David S. Johnson, and each of them is a thorough survey of recent progress in one particular area of NP-completeness research. They are written as updates of the first reference below.

Journals that contain a good deal of research on the areas of this chapter include the Journal of Algorithms, the Journal of the Association for Computing Machinery, the SIAM Journal of Computing, Information Processing Letters, and SIAM Journal of Discrete Mathematics.

The most complete reference on NP-completeness is

M. Garey and D. S. Johnson, *Computers and Intractability; A guide to the theory of NP-completeness*, W. H. Freeman and Co., San Francisco, 1979.

The above is highly recommended. It is readable, careful and complete.

The earliest ideas on the computational intractability of certain problems go back to

Alan Turing, On computable numbers, with an application to the *Entscheidungsproblem*, Proc. London Math. Soc., Ser. 2, **42** (1936), 230-265.

Cook’s theorem, which originated the subject of NP-completeness, is in

S. A. Cook, The complexity of theorem proving procedures, Proc., Third Annual ACM Symposium on the Theory of Computing, ACM, New York, 1971, 151-158.

After Cook’s work was done, a large number of NP-complete problems were found by

Richard M. Karp, Reducibility among combinatorial problems, in R. E. Miller and J. W. Thatcher, eds., *Complexity of Computer Computations*, Plenum, New York, 1972, 85-103.

The above paper is recommended both for its content and its clarity of presentation.

The approximate algorithm for the travelling salesman problem is in

D. J. Rosencrantz, R. E. Stearns and P. M. Lewis, An analysis of several heuristics for the travelling salesman problem, SIAM J. Comp. **6**, 1977, 563-581.

Another approximate algorithm for the Euclidean TSP which guarantees that the solution found is no more than $3/2$ as long as the optimum tour, was found by

N. Cristofides, Worst case analysis of a new heuristic for the travelling salesman problem, Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, 1976.

The minimum spanning tree algorithm is due to

R. C. Prim, Shortest connection networks and some generalizations, Bell System Tech. J. **36** (1957), 1389-1401.

The probabilistic algorithm for the Hamilton path problem can be found in

D. Angluin and L. G. Valiant, Fast probabilistic algorithms for Hamilton circuits and matchings, Proc. Ninth Annual ACM Symposium on the Theory of Computing, ACM, New York, 1977.

The result that the graph coloring problem can be done in constant average time is due to

H. Wilf, Backtrack: An $O(1)$ average time algorithm for the graph coloring problem, Information Processing Letters **18** (1984), 119-122.

Further refinements of the above result can be found in

E. Bender and H. S. Wilf, A theoretical analysis of backtracking in the graph coloring problem, Journal of Algorithms **6** (1985), 275-282.

If you enjoyed the average numbers of independent sets and average complexity of backtrack, you might enjoy the subject of random graphs. An excellent introduction to the subject is

Edgar M. Palmer, Graphical Evolution, An introduction to the theory of random graphs, Wiley-Interscience, New York, 1985.

adjacent 40
Adleman, L. 149, 164, 165, 176
Aho, A. V. 103
Angluin, D. 208-211, 227
Appel, K. 69
average complexity 57, 211*ff.*

backtracking 211*ff.*
Bender, E. 227
Bentley, J. 54
Berger, R. 3
big oh 9
binary system 19
bin-packing 178
binomial theorem 37
bipartite graph 44, 182
binomial coefficients 35
—, growth of 38
blocking flow 124
Burnside's lemma 46

cardinality 35
canonical factorization 138
capacity of a cut 115
Carmichael numbers 158
certificate 171, 182, 193
Cherkassky, B. V. 135
Chinese remainder theorem 154
chromatic number 44
chromatic polynomial 73
Cohen, H. 176
coloring graphs 43
complement of a graph 44
complexity 1
—, worst-case 4
connected 41
Cook, S. 187, 194-201, 226
Cook's theorem 195*ff.*
Cooley, J. M. 103
Coppersmith, D. 99
cryptography 165
Cristofides, N. 224, 227
cut in a network 115
—, capacity of 115
cycle 41
cyclic group 152

decimal system 19
decision problem 181
degree of a vertex 40
deterministic 193
Diffie, W. 176
digraph 105
Dinic, E. 108, 134
divide 137
Dixon, J. D. 170, 175, 177
domino problem 3

'easy' computation 1
edge coloring 206
edge connectivity 132

Index

- Edmonds, J. 107, 134, 224
- Enslin, K. 103
- Euclidean algorithm 140, 168
 - , complexity 142
 - , extended 144*ff.*
- Euler totient function 138, 157
- Eulerian circuit 41
- Even, S. 135
- exponential growth 13

- factor base 169
- Fermat's theorem 152, 159
- FFT, complexity of 93
 - , applications of 95 *ff.*
- Fibonacci numbers 30, 76, 144
- flow 106
 - , value of 106
 - , augmentation 109
 - , blocking 124
- flow augmenting path 109
- Ford-Fulkerson algorithm 108*ff.*
- Ford, L. 107*ff.*
- four-color theorem 68
- Fourier transform 83*ff.*
 - , discrete 83
 - , inverse 96
- Fulkerson, D. E. 107*ff.*

- Galil, Z. 135
- Gardner, M. 2
- Garey, M. 188
- geometric series 23
- Gomory, R. E. 136
- graphs 40*ff.*
 - , coloring of 43, 183, 216*ff.*
 - , connected 41
 - , complement of 44
 - , complete 44
 - , empty 44
 - , bipartite 44
 - , planar 70
- greatest common divisor 138
- group of units 151

- Haken, W. 69
- Hamiltonian circuit 41, 206, 208*ff.*
- Hardy, G. H. 175
- height of network 125
- Hellman, M. E. 176
- hexadecimal system 21
- hierarchy of growth 11
- Hoare, C. A. R. 51
- Hopcroft, J. 70, 103
- Hu, T. C. 136

- independent set 61, 179, 211*ff.*
- intractable 5

- Johnson, D. S. 188, 225, 226

- Karp, R. 107, 134, 205, 226
- Karzanov, A. 134
- Knuth, D. E. 102
- König, H. 103

Index

- k -subset 35
- language 182
- Lawler, E. 99
- layered network 120*ff.*
- Lenstra, H. W., Jr. 176
- LeVeque, W. J. 175
- Lewis, P. A. W. 103
- Lewis, P. M. 227
- L'Hospital's rule 12
- little oh 8
- Lomuto, N. 54

- Maheshwari, S. N. 108*ff.* , 135
- Malhotra, V. M. 108*ff.* , 135
- matrix multiplication 77*ff.*
- max-flow-min-cut 115
- maximum matching 130
- minimum spanning tree 221
- moderately exponential growth 12
- MPM algorithm 108, 128*ff.*
- MST 221
- multigraph 42

- network 105
 - flow 105*ff.*
 - , dense 107
 - , layered 108, 120*ff.*
 - , height of 125
- Nijenhuis, A. 60
- nondeterministic 193
- NP 182
- NP-complete 61, 180
- NP-completeness 178*ff.*

- octal system 21
- optimization problem 181
- orders of magnitude 6*ff.*

- P 182
- Palmer, E. M. 228
- Pan, V. 103
- Pascal's triangle 36
- path 41
- periodic function 87
- polynomial time 2, 179, 185
- polynomials, multiplication of 96
- Pomerance, C. 149, 164, 176
- positional number systems 19*ff.*
- Pramodh-Kumar, M. 108*ff.* , 135
- Pratt, V. 171, 172
- Prim, R. C. 227
- primality, testing 6, 148*ff.* , 186
 - , proving 170
- prime number 5
- primitive root 152
- pseudoprimalty test 149, 156*ff.*
 - , strong 158
- public key encryption 150, 165

- Quicksort 50*ff.*

- Rabin, M. O. 149, 162, 175
- Ralston, A. 103

Index

- recurrence relations 26*ff.*
- recurrent inequality 31
- recursive algorithms 48*ff.*
- reducibility 185
- relatively prime 138
- ring \mathbf{Z}_n 151*ff.*
- Rivest, R. 165, 176
- roots of unity 86
- Rosenkrantz, D. 227
- RSA system 165, 168
- Rumely, R. 149, 164, 176
- Runge, C. 103

- SAT 195
- satisfiability 187, 195
- scanned vertex 111
- Schönhage, A. 103
- Selfridge, J. 176
- Shamir, A. 165, 176
- slowsort 50
- Solovay, R. 149, 162, 176
- splitter 52
- Stearns, R. E. 227
- Stirling's formula 16, 216
- Strassen, V. 78, 103, 149, 162, 176
- synthetic division 86

- 3SAT 201

- target sum 206
- Tarjan, R. E. 66, 70, 103, 135
- Θ ('Theta of') 10
- tiling 2
- tractable 5
- travelling salesman problem 178, 184, 221
- tree 45
- Trojanowski, A. 66, 103
- 'TSP' 178, 221
- Tukey, J. W. 103
- Turing, A. 226
- Turing machine 187*ff.*

- Ullman, J. D. 103
- usable edge 111
- Valiant, L. 208-11, 227
- vertices 40
- Vizing, V. 206

- Wagstaff, S. 176
- Welch, P. D. 103
- Wilf, H. 60, 103, 227, 228
- Winograd, S. 99
- worst-case 4, 180
- Wright, E. M. 175