

**NP - COMPLETE PROBLEMS,
TURING MACHINES, AND
THE PROOF OF COOK'S THEOREM**

**SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR GRADUATION
WITH HONORS TO THE**

**DEPARTMENT OF
MATHEMATICS, COMPUTER SCIENCE,
ENGINEERING AND PHYSICS**

CARROLL COLLEGE

BY

LAURA D. BRUNKEN

HELENA, MONTANA

DECEMBER 1989



CORETTE LIBRARY
CARROLL COLLEGE

SIGNATURE PAGE

This thesis for honors recognition has been approved for the
Department of Mathematics, Computer Science, Engineering and Physics

Philip B. Rose

Director

Darrell Hagen

Reader

Marie M. Vanisko

Reader

12/07/89

Date



ACKNOWLEDGMENTS

I'd like to thank all of my teachers at Carroll College for giving me the background and motivation to attempt such a project. Very special and warm thanks goes to Philip Rose, without whose help and understanding I would never have been able to complete this endeavor. Also to my readers, Darrell Hagen and Marie Vanisko, who spent so much time expressing their ideas and comments to aid me along the way, I thank you.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	ii
LIST OF ILLUSTRATIONS.....	iv
CHAPTER	
1. INTRODUCTION	1
2. PRELIMINARIES.....	3
Sample Problems	
Polynomial vs. Superpolynomial Time	
The Classes P and NP	
NP-Complete Problems	
3. TURING MACHINES.....	17
Four Characteristics	
Deterministic vs. Nondeterministic Turing Machines	
Universal Turing Machines	
Parity Check Turing Machine	
4. COOK'S THEOREM.....	29
Satisfiability Problem	
Proof of Cook's Theorem	
5. POLYNOMIAL REDUCTIONS.....	43
3SAT is NP-Complete	
3-Coloring a Map is NP-Complete	
Traveling Salesman Problem is NP-Complete	
EPILOGUE	55
APPENDIX.....	57
BIBLIOGRAPHY	65

LIST OF ILLUSTRATIONS

Figure	Page
1. Coloring of a Map	5
2. Coloring Points.....	5
3. Polynomial vs. Superpolynomial growth.....	9
4. Polynomial Reduction.....	14
5. Turing Machine Tape.....	19
6. Program Operation.....	21
7. Deterministic Turing Machine Example.....	23
8. Nondeterministic Turing Machine Example.....	23
9. Input Tape for Parity Check Turing Machine	26
10. Program Table for Parity Check Turing Machine.....	26
11. State Diagram for Parity Check Turing Machine	28
12. Possibilities for Symbol k and Square j	40
13. Hamiltonian Circuit and Traveling Salesman Paths.....	51
14. Comparison of Vertices and Edges.....	52
15. Turing Machine Examples	57
16. Submachine 0.....	58
17. Submachine 1.....	59
18. Concatenation Submachine 2	60
19. Submachines to Concatenate Single Characters.....	61
20. Submachine 2.....	62
21. Submachine 3.....	62

CHAPTER ONE - INTRODUCTION

There exists a group of practical problems in computer science today for which no one has yet been able to find reasonable algorithmic solutions (those solvable in a reasonable amount of time), yet no one has been able to prove that *no* such solutions exist, either. These problems readily admit *naive*, or obvious solutions, but these solutions would take entirely unreasonable amounts of time, say thousands of years, to terminate. Scientists have simply not been able to find other algorithms that run faster and more efficiently, but, as noted, have also been unable to prove their nonexistence. This puzzle has teased computer scientists for decades now, and it continues to be of great interest.

Why are these problems so important? There are approximately 1000 problems known to fall into this uncertainty category¹, and most of them have many practical uses. For example, if we could solve the Traveling Salesman problem, we could immediately map out the most efficient route for delivery companies such as UPS or Federal Express. If we could figure out the Bin Packing problem, lumber yards would be able to more efficiently fill orders from large pieces of wood. Being able to solve the Graph Coloring problem would make scheduling final exams for a university much quicker and easier. These problems are discussed in length in the pages that follow.

One of the most interesting aspects of these problems is that *if* scientists could find a fast running algorithm to solve any *one* of this special category of problems, they would know that a fast algorithm necessarily exists for *all* of

¹David Harel, Algorithms: The Spirit of Computing. (Wokingham, England: Addison-Wesley Publishing Company, Inc., 1987), 160.

them! Conversely, if they could *prove* that no such algorithm exists for one of them, they would know that there will never be a fast algorithm to solve any of them, and they could concentrate their efforts on finding algorithms to approximate the solutions. This class of problems, commonly called NP-complete problems, will be a major topic of this paper.

Turing machines, which will be the second focus of this paper, play a unique role in the development of these NP-complete problems. It is through these theoretical machines that the computation process has been separated into its most basic elements. Stephen A. Cook used the theory of Turing machines to prove his landmark theorem, which established the existence of an actual NP-complete problem. In the third section, I will present and prove Cook's Theorem, and to conclude the paper I will pull many of these ideas together by presenting three reductions between various pairs of NP-complete problems.

CHAPTER TWO - PRELIMINARIES

Many problems which admit algorithmic solutions can be computed in reasonable amounts of time. Even if they take months or a few years, there is still light at the end of the tunnel. The group of problems that concerns us in this paper, however, does not afford such nice, reasonable time solutions, at least for the more interesting, important, or useful instances. These problems, if we tried to solve them the only way we currently know how, would take upwards of centuries, and this is assuming the problems are being done on the fastest supercomputers in the world today!

These problems are so prevalent in everyday society that it is of utmost concern to computer scientists to either find algorithmic solutions that will run fairly quickly, or *prove* that none exist so they can concentrate their efforts elsewhere. Before going further, let's take a moment to examine a few of these problems in detail.

Sample Problems

Traveling Salesman. In this problem, we are given a map with some number, n , of cities, and a target distance, x . The object is to find a path through these cities such that the total distance traveled is less than or equal to x .² The question we wish to answer when faced with this problem is, can this be done? Yes or no? Throughout my discussions of all of these problems, I will be stating

²Harel, Algorithms: The Spirit of Computing. 162.

them as **decision problems**, i.e. problems which admit a 'yes' or 'no' answer. Many times, though, it is the corresponding **optimization problem** that interests us, i.e. what is the shortest route, the smallest number of colors, etc. In most cases, an optimization problem can be restated as a decision problem so that a solution to the decision problem implies a solution to the optimization problem. The correspondence of these two types of problems is important, because being a decision problem is one criterion for membership in the NP-complete category of problems.

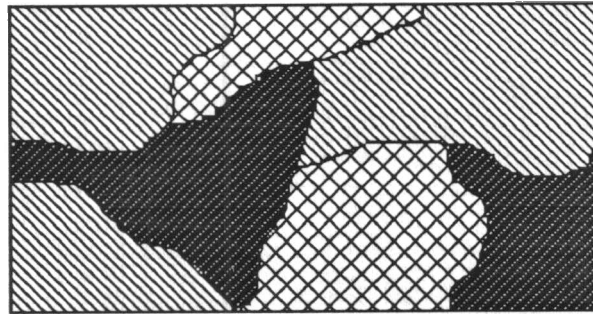
Now there are obvious practical uses for the Traveling Salesman problem, one being the routing of delivery trucks. Wouldn't UPS be delighted if someone could find a fast algorithm for this problem, so they could simply input the stops for each driver and out would come the map with the best possible route? Regrettably, no one has found such a nice solution!

Graph Coloring. Given a set of n points (or sections of a map) interconnected in some pattern, and a group of k colors, is it possible to color the points (or sections) in such a way that no two adjacent points (or sections) have the same color?³ See figures 1 and 2 for examples.

³Sara Baase, Computer Algorithms: Introduction to Design and Analysis. 2d ed., (Reading, MA: Addison-Wesley Publishing Company, Inc., 1988), 320.

Fig. 1. Coloring of a map.

There is no way, using only 2 patterns, that all adjacent sections will have different colorings, but with 3 patterns, this is easily attainable.



n=7
areas

k=2?   NO
k=3?    YES

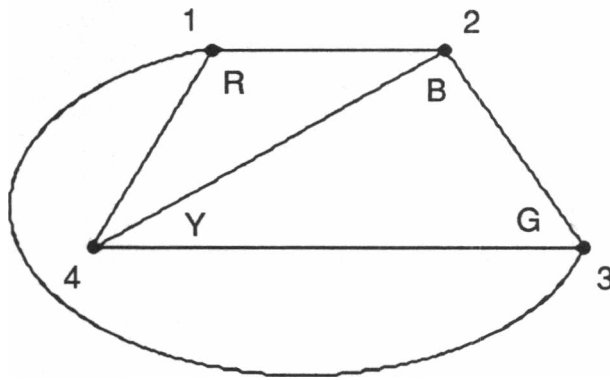


Fig. 2. Coloring points

Here, there are only 4 points, but because of the way they are connected, they can only be properly colored with a minimum of 4 colors!

k=2? R(ed), B(lue) NO
k=3? R(ed), B(lue), G(reen) NO
k=4? R(ed), B(lue), G(reen), Y(ellow) YES

The area in which graph coloring is most often used occurs with scheduling problems. For example, suppose a large university was scheduling their final exams for a certain 5-day week, and they wanted four exams per day, for a total of twenty time slots. Certain pairs of exams, such as Physics I and Calculus I, can't be scheduled at the same time because so many of the same students are in both classes. Therefore, if we let n represent the number of

courses, the borders between them reflect pairs of exams not to be at the same time, and $k=20$ stand for the time slots, this practical problem reduces to trying to properly color the n vertices with the $k=20$ colors.⁴

Bin Packing. Given a number, n , of bins that each have a capacity of one unit, and a set of k objects, s_1, \dots, s_k , each with size $0 \leq |s_i| \leq 1$ (for $i = 1 \dots k$), is there a way to pack these k items into the n bins?⁵ For example, suppose we have three buckets and two dozen each of oranges, grapefruit and bananas. Is there a way to arrange the fruit into these three buckets? Obviously, certain combinations of the fruit will provide the optimal arrangement for the bins, but the problem is finding those combinations.

One very important application of this problem is that of computer memory and the storage of data. As technology advances, data is becoming more complex and coming in greater quantities. We are certainly always looking for more efficient ways of storing this data; we need smaller storage areas to hold more information.

Primality Testing. Much of our military defense communication is encrypted so that other nations can't intercept and read our messages. Prime numbers, those whose only factors are 1 and the numbers themselves, play a central role in many cryptographic systems of today. Primality testing looks to see whether a given number is prime or composite (having other factors). At first glance, this seems to be a fairly quick and easy assignment. After all, here is an algorithm to complete this very task on some input 'number':

⁴Baase, Computer Algorithms: 321.

⁵Ibid.

```

prime = True
i = 1
while (prime) and (i <  $\sqrt{\text{num}}$ ) do
  begin
    i = i + 1
    if the remainder of num/i is 0, then prime = False
  end
if prime then output 'yes, the number is a prime'
else output 'no, the number is not a prime'

```

The problem with this algorithm is that it is *much* too slow for the size of the numbers that we wish to test; we're talking about testing 100 and 200 **digit** numbers! When you consider the fact that the number of microseconds since the Big Bang is thought to be a 24 digit number,⁶ you can easily show that using the above method for testing primality is absolutely unreasonable.

Researchers have come closer to solving this problem than the other three previously mentioned, but the methods are based on probabilism, so there is always a degree of uncertainty in the answer, 'yes, the number is a prime.'

These are but a few samples from the class of "hard" problems; hard because they take unreasonable amounts of time, namely **super-polynomial time** to solve. Let's look at this a bit closer.

Polynomial vs. Super-polynomial Time

A function, $f(n)$, is **bounded from above** by another function, $g(n)$, if there is some value, n_0 , such that $n > n_0$ implies $g(n)$ will always be greater than

⁶Harel, Algorithms: The Spirit of Computing, 156.

or equal to $f(n)$.⁷ For example, let $f(n) = n^2$ and $g(n) = n^4$. It is easy to see that $f(n)$ is bounded from above by $g(n)$, since for $n \geq 1$, $g(n) \geq f(n)$.

A **polynomial function of n** is one that is *bounded from above* by n^k , where k is some constant. Polynomial functions include n , n^2 , $4n^{10}$, and $\log_{10}n$. Notice in all of these examples that the variable n is never in the exponent position; one characteristic of *super-polynomial* functions such as 2^n , 1.001^n , and n^n , is that it is.

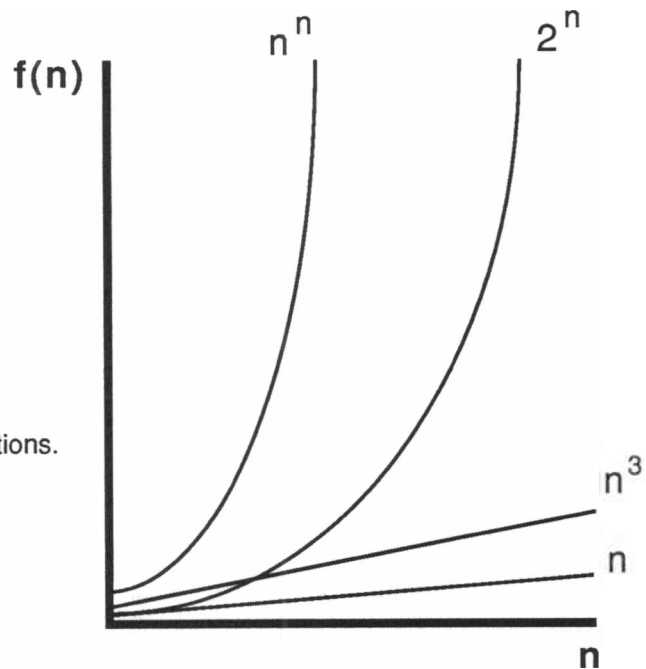
The distressing point about super-polynomial functions is that they grow very rapidly as n becomes large. The chart below gives some values for two polynomial functions and two super-polynomial functions as n increases. For a graphical representation of this data, see figure 3.

f(n) =	<u>n</u>	<u>n</u>³	<u>2</u>ⁿ	<u>n</u>ⁿ
	polynomial	polynomial	super-polynomial	super-polynomial
n = 1	1	1	2	1
2	2	8	4	4
4	4	64	16	256
8	8	512	256	16,777,216
16	16	4096	65,536	1.8×10^{18}
32	32	32,768	4.3×10^9	1.5×10^{48}
64	64	262,144	1.8×10^{19}	9.9×10^{99}

⁷Harel, Algorithms: The Spirit of Computing, 156.

Fig. 3. Polynomial vs. Superpolynomial growth

Notice how quickly the two superpolynomial functions grow compared to the polynomial functions.



Finally, we come to the definition of a **polynomial time algorithm**: an algorithm whose running time is bounded from above by a polynomial function of n , where n is the size of the input.⁸ Let's take a closer look at this notion of 'size of the input' by examining the primality testing algorithm I put forth earlier. If we look at the complexity of the algorithm, we see that for each number n we test, we do approximately \sqrt{n} units of work (really divisions of whole numbers). In this light, the algorithm seems to be polynomially bounded, until we look more closely at the actual size of the input, however. "Size of the input" really refers to the size of *storing* the input in a computer, i.e. the number of *bits* needed for storage. It doesn't take 128 bits to store the decimal number 128, in binary it takes only 8 bits. In general, it takes approximately $\log_2(n)$ bits to store a decimal number⁹, since the number of bits needed is the number of times that

⁸Harel, *Algorithms: The Spirit of Computing*, 157.

⁹Herbert S. Wilf, *Algorithms and Complexity*. (Englewood Cliffs, NJ: Prentice-Hall International, Inc., 1986), 5.

2 divides n . Now study the complexity of the primality testing algorithm again. For an input size of $\log_2(n)$, \sqrt{n} units of work suddenly seems like a great deal, and indeed it is. To see this more clearly, examine the following chart, which shows some sample numbers, input bits in binary, and work units, \sqrt{n} :

n (base 10)	input bits needed (base 2)	work units = \sqrt{n}
4	3 (e.g. 100)	2
16	5 (e.g. 10000)	4
64	7	8
256	9	16
1024	11	32
4096	13	64
16384	15	128

Notice the relationship between the second and third columns: for the number of input bits x , the work units are calculated with the formula $2^{\lfloor x/2 \rfloor}$. Thus, this actually has an *exponential* rather than polynomial time bound. The key to remember when examining algorithms is to look at the size of *storing* the input, and consider the complexity in terms of this.

Any algorithms that are polynomially bounded with respect to the size of storing the input are said to be **reasonable**, while super-polynomial time algorithms are **unreasonable**. Examining the charts above, I think you can see why we would like to stay with the polynomial time algorithms.

The Classes P and NP

The two classes, or families of problems known as P and NP provide us with some defining characteristics so that we may group many of the problems in Computer Science. Into the category of P falls any decision problem that has a polynomial time, deterministic solution.¹⁰ Let's dissect this definition so we can come to understand it better. First of all, a **decision problem** is one that can be stated so that the answer will either be *yes* or *no*. We've already discussed the meaning of polynomial time above, so that just leaves deterministic. A **deterministic solution** is one that has a definite step-by-step structure to it. The algorithm proceeds from one step to another by very structured, non-flexible rules, which are *determined* before the algorithm is executed; there is absolutely no guesswork involved.

So the class P (for polynomial time) is the set of 'easy' problems; those that have step-by-step procedures that run fairly quickly. None of the sample problems set forth at the beginning of this section belong to this class; those all belong to the class of NP problems.

There are four main characteristics of NP problems, the first being that they are decision problems, or can be at least re-stated as decision problems. The second characteristic is that these problems have obvious naive solutions, but would take super-polynomial time to execute. An example of this is the primality testing algorithm previously described. It gets the job done for small numbers, but becomes quickly useless for relatively large integers.

A third quality all NP problems have is that if we were given a proposed answer to the problem, we could easily verify (in polynomial time) the

¹⁰Robert Sedgewick, Algorithms. 2d ed, (Reading, MA: Addison-Wesley Publishing Company, Inc., 1988), 634.

correctness or incorrectness of this answer. For example, in the Traveling Salesman problem, if we were given a path through the cities, we could very quickly add up all of those distances to make sure they were less than our target distance. Likewise in the Graph Coloring problem, if someone gave us a list of the vertices and a proposed set of colorings, it wouldn't be hard to make sure no two adjacent vertices had the same color. These proposed solutions are called **certificates**,¹¹ because we certify that they are or are not correct. Thus, the difference between P and NP problems with regard to polynomial time is this: given any instance of a P problem, the computer can output the correct 'yes' or 'no' answer in polynomial time; given any instance of an NP problem **and** a proposed solution to the problem, the computer can output whether the proposed solution is correct or not, and it does *this* in polynomial time.

The fourth characteristic of NP problems is the most difficult to comprehend, but it is also the most fascinating. This is where we finally come to what the NP stands for, which is **nondeterministic polynomial** time solutions. To understand this, let's go back to our method of solving these problems. As we're trying to come up with a solution, we are faced with choices at each step: which city we should travel to next, which item we should next put into a bin, and so on. In other words, there is no single series of steps to take; no *deterministic* method of approach. We must 'guess' the next step to take, and this is where the *nondeterministic* variable comes into play.

Suppose that we were carrying with us a magic coin, and whenever we were faced with choices, we would simply flip the coin and proceed accordingly.¹² This coin, however, has the incredibly useful property that it magically always makes the 'right', or best possible choice. Wouldn't this be

¹¹Wilf, Algorithms and Complexity. 182.

¹²Harel, Algorithms: The Spirit of Computing. 167.

handy? If we had such a coin, we could certainly find a solution (if there is one) that runs in polynomial time, because with this aid, we would *know* the next step in which to proceed! It is the guessing, and the trying of all the possible steps that takes super-polynomial time; but if we eliminated that with our magic coin, we would cut our time down to polynomial. That's the good news. The bad news is that the only place that these much needed nondeterministic algorithms exist is in Computer Science theory! This idea is merely to help us recognize and classify problems into the NP category.

To recap, the basic difference between problems in the P and NP classes is this: P problems are those for which it is quick to *find* a solution; NP problems are those for which it is quick to *check* a proposed solution.¹³

Does $P = NP$? Do all of the problems in the P class also belong to NP, and visa versa, so that there exists only one group? This is the most widely researched question in Computer Science today.¹⁴ We don't yet know if there is a polynomial time, deterministic algorithm that will solve any of the problems in NP. No one has been able to find one, and most researchers are of the belief that none exist, so they lean toward the proposition $P \neq NP$, i.e. that $P \subset NP$. However, no one has been able to *prove* that polynomial time algorithms for these problems don't exist. Thus, the quest goes on!

NP-complete Problems

The last subject to discuss, and the one which will lead us into the succeeding sections, is that of the NP-complete group of problems. To be a

¹³Wilf, Algorithms and Complexity. 183.

¹⁴Ibid, 181.

member of this group, a problem must have two characteristics. First of all, it must be a member of the NP group, and thus have all four of the properties NP problems have, and secondly, every problem in NP must be polynomially reducible to it.¹⁵ Let's define polynomial reducibility more explicitly.

Let π_1 be some decision problem for which we would like an answer, but for which we don't have an algorithm, and x_1 be its input. Next, suppose π_2 is a problem for which we *do* have an algorithm, with x_2 representing its input. Note that this discussion is independent of the problem being in P or NP. If we can find some way to transform the x_1 input of π_1 into an x_2 input for π_2 such that a *yes* answer for π_2 would necessarily yield a *yes* for π_1 , and likewise for *no*, then we would have the problem π_1 solved! This is the idea of **reducibility**; reduce one problem to another so that the answer to the second, either 'yes' or 'no', will precisely be the answer to the first. If this reduction from one problem to another can be done algorithmically and in polynomial time, then we say π_1 is **polynomially reducible** to π_2 . See figure 4.

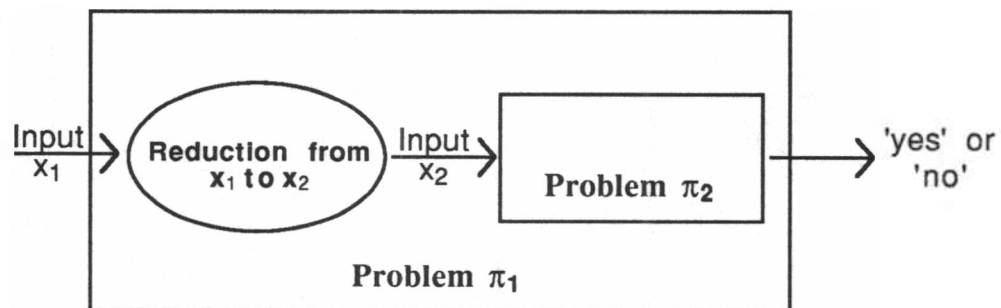


Fig. 4 Polynomial Reduction

Notice the requirement that the output from problem π_2 is necessarily the same as the output from problem π_1 . The reduction in the oval must be done in polynomial time to apply the term "polynomial reduction."

¹⁵Michael R. Garey and David S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. (San Francisco, CA: W.H. Freeman and Company, 1979), 13.

The magnificent property of these NP-complete, or NPC problems as they're called, is this: if we could find a polynomial time algorithm to solve any *one* NPC problem, then we would know that *every* problem in NP has a quick solution. Conversely, if one NP problem can be proven to *not* have a polynomial time solution, that would imply that *no* NPC problem does, since if an NPC problem *had* a quick solution, and because the NP problem is reducible to it, then it too would be solvable in polynomial time. This all or nothing characteristic is what the **complete** in NP-complete means: they are either *all* solvable in polynomial time or they are all not, and it makes sense, since each problem reduces to all the others.

The main relationship with which we should concern ourselves is this: a fast algorithm for any problem in NPC means there exists a fast algorithm for all NP problems; conversely, *proving* that any one NP problem is necessarily slow (super-polynomial time) means that every problem in NPC is also slow.¹⁶ Remember that before a problem can be termed NP-complete, it must first be a member of the NP class. Therefore, if we show that an NP problem is slow, then since it is reducible to *every* problem in NPC, then every problem in NPC must also be slow. Conversely, if any one of the NP-complete problems is shown to be fast, then of course all the rest of the NPC group will also be fast, but so will all of the problems in NP. Why? Remember the definition of an NP-complete problem - all members of the NP class must be reducible to it. So if an NPC problem is fast, and all the NP problems reduce to this problem, they also must be fast!

Where did all of this start? Indeed, how do we know there exists even one such NPC problem? Their existence is certainly counter-intuitive. Stephen

¹⁶Wilf, Algorithms and Complexity. 186.

A. Cook proved in 1971 that there is an NP-complete problem, namely the **satisfiability problem**.¹⁷ In the section after next, I will explain this problem thoroughly, and prove that it is NP-complete. The next section will give the building blocks for this proof, namely Turing Machines.

¹⁷Harel, Algorithms: The Spirit of Computing. 171.

CHAPTER THREE - TURING MACHINES

A **Turing machine** is not a physical machine with nuts and bolts; it is more of a programming environment, named a 'machine' because of the terms used to describe it. The Turing machine was invented in 1936 by Alan Turing¹⁸ as a theoretical aid to justify and help prove other theorems. It is the simplest of all computing environments, yet as powerful as the most sophisticated; a Turing machine can solve *any* algorithmic problem that any other computer and programming language can.

Alan Mathison Turing was one of the most brilliant mathematicians and computer scientists of all time, partly because of his analysis of the computation process and the resulting Turing machine in 1936. This was by no means his only accomplishment, however. Four years later in 1940, Turing went to work for the British Intelligence - Cipher Division, and it was through his genius that the British were able to break the ciphers coming from the German Enigma Machine.¹⁹ Largely due to this, the Allies were able to gain control of the Atlantic passageways and ultimately defeat Germany in WWII. After the war, Turing worked on the creation of computers for public use, and was one of the first to envision the concept of **computer programming**, in which programs *inside* the computer's memory would control the actions of the computer.²⁰

Quite opposite of his immense successes in the mathematics and computer science fields, Turing always had difficulty conforming with the

¹⁸R. Sommerhalder, and S.C. van Westrhenen, The Theory of Computability: Programs, Machines, Effectiveness and Feasibility. (Wokingham, England: Addison-Wesley Publishing Company, Inc., 1988), 32.

¹⁹Andrew Hodges, Alan Turing: The Enigma. (New York, NY: Simon and Schuster, Inc., 1983), 160.

²⁰Hodges, Alan Turing, 326.

general society. Sometimes treated like a leper by others, he found this part of his life too difficult to endure, and hence committed suicide in 1954 at the age of 42.²¹ The Turing Award, which has come to be recognized as the most prestigious award in computer science today, was established in 1966 to remember the great mind of Alan Turing. It is awarded annually by the Association of Computing Machinery to an individual who has significantly contributed in a technical manner to the computing community.²² There have been 23 recipients thus far, including such great minds as S.A. Cook (Cook's Theorem), Donald Knuth (The Art of Computer Programming), Dennis Ritchie and Ken Thompson (UNIX operating system and C programming language), and John Backus (FORTRAN programming language).²³

Let's take a closer look at Turing's first major accomplishment: the Turing machine. The way we are going to use Turing machines is to think of them as aiding in the solutions of decision problems. We could program a certain decision problem into the Turing machine, run it with inputs, and it would return a 'yes' or 'no' answer. Keep in mind that we're talking theoretically, *as if* we were really going to construct Turing machines to solve these decision problems. Our main purpose in using them is as a model to aid us in the proof of Cook's Theorem, which brings us to another very important use for Turing machines: their utilization in the proofs of some very important theorems. Turing machines do not have to be used just for decision problems or proofs as I will demonstrate later in this chapter, but for now let's take a closer look at exactly what a Turing machine is.

²¹Hodges, Alan Turing, 487.

²²Robert L. Ashenurst and Susan Graham, eds., ACM Turing Award Lectures - The First Twenty Years. (Reading, MA: Addison-Wesley Publishing Company, Inc., 1987), 17.

²³Ashenurst, ACM Turing Award Lectures.

Four Characteristics

There are four main characteristics (which really amount to simplifications of a normal programming environment) of all Turing machines. The four have to do with the data tape, tape head, set of states and the program.

Data Tape. Each Turing machine has an infinitely long tape consisting of squares that are capable of holding only one character at a time.²⁴ The squares are numbered with integers, as in figure 5.

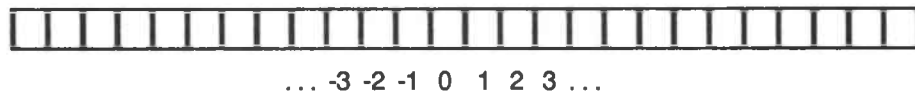


Fig. 5 Turing Machine Tape

There is a finite set of **symbols** that can be put in these squares and the set may vary, depending upon the particular program that is being developed. For example, the binary character set of { 0,1 } may be used, or perhaps the set { *,A,B,C }, as in the food chain problem given in the appendix.

Tape Head. Think of the **tape head** as a mechanical device (though it isn't *really* mechanical) that moves over the data tape, looking at only one square at a time. The head can only do three different activities: it can read a character from a square, it can write a character to a square, or it can move itself one square to the right or left (or stay stationary).²⁵ That is *all* it can do; it can't add two numbers, it can't branch to a subroutine, or perform any of the other

²⁴Wilf, Algorithms and Complexity, 189.

²⁵Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, (Reading, MA: Addison-Wesley Publishing Company, Inc., 1974), 26.

functions we would normally assume a programming environment could do. Yet, these Turing machines are ever so powerful.

States. A Turing machine is always in one of a finite number of **states**, which we can think of as certain configurations of the program.²⁶ For example, the *initial state* occurs at the start of the program before any characters have been read. The two final states, sometimes denoted q_y and q_n , stand for the Turing machine ending with a 'yes' or 'no', respectively.²⁷ These three are the only states that are usually named, but whatever configuration the Turing machine happens to be in at a certain moment is termed a state. This will become clearer with the examples to be shown.

Program. How does the Turing machine get from one state to another? A **program** directs the procedure through states. Depending on the state, q , of the Turing machine and the character, k , just read, the *program* decides three things: the new state, q' , the new symbol to write, k' , and the direction (if any) for the tape head to move. For example, this is how the Turing machine would typically work. Load the input string (of length B) into tape squares 1,2,3,..., B . Position the tape head over square 1 and set the state to q_0 (the initial state). Then enter the cycle:

1. Read a symbol from the tape
 2. Consult the program
 3. Write a symbol to the tape (at the same location)
 4. Move left or right one square (or not at all)
 5. Enter new state
- If new state = q_y or q_n then stop, otherwise start over at step 1.

This can be depicted graphically, as in figure 6.

²⁶Aho, Design and Analysis. 26.

²⁷Wilf, Algorithms and Complexity. 189.

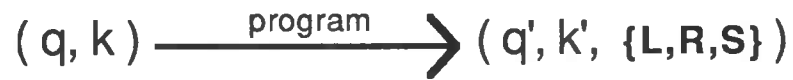


Fig. 6 Program Operation

After reading symbol k , the program decides the new state q' , new symbol k' , and direction, L(ef), R(ight), S(tationary), of the tape head.

With only these four traits, it is almost impossible to believe that Turing machines could be helpful in solving *any* algorithmic problem. Yet, from the minds of Alonzo Church and Alan Turing came the landmark **Church/Turing Thesis**, which demonstrates the incredible power of Turing machines. The Church/Turing Thesis states that a Turing machine can solve *any* problem that could be solved algorithmically on any computer using any programming language.²⁸ This is an incredible statement when you think about it. Any algorithmic problem, even the most complicated ones that guide our space travel, for example, could be accomplished by a system that can only read a symbol, write a symbol, and move one square left or right. It is because of this incredible power that we can use Turing machines as bases for proving complicated theorems, such as Cook's Theorem, which will come in the next chapter.

Deterministic vs. Nondeterministic Turing Machines

Before going directly to the difference between deterministic and nondeterministic Turing machines, we need a little bit of background

²⁸Harel, Algorithms: The Spirit of Computing, 221.

information. As we are utilizing Turing machines in the next chapter, we will need to concern ourselves with a few general aspects. The **length** of the input string is simply the number of characters in it. One **step** in the program consists of changing from one state to the next, as in figure 6. **Time** is measured in steps, and normally, polynomial time is depicted as $P(n)$, where n is the number of cells for the input. Thus, a calculation would be done in time $P(n)$ if it took the Turing machine $P(n)$ steps to complete it. Now we come to the two types of Turing machines.

In a **deterministic Turing machine**, each state has only *one* of each type of configuration leading to the next state.²⁹ For example, suppose the Turing machine is in state $q = 0$ and can either read an A or a B from the tape. Upon reading an A, the Turing machine will leave the A, move one square right and proceed to state $q = 1$, and upon reading a B, it will move one square left and go to state $q = 2$. Graphically, this would look like figure 7. Notice that there is only one line going out for each of A and B.

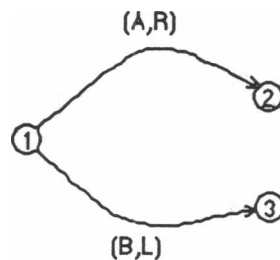


Fig. 7 Deterministic Turing Machine Example

²⁹Harel, Algorithms: The Spirit of Computing. 216.

In a **nondeterministic Turing machine**, which solves NP problems, there may be many lines emanating for each configuration, representing the many different choices, or routes the machine may take.³⁰ See figure 8.

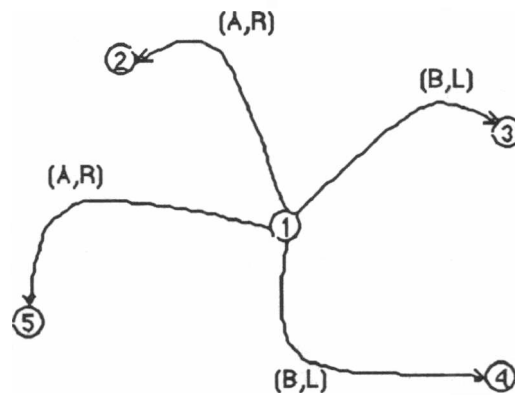


Fig. 8 Nondeterministic Turing Machine Example

From state $q = 1$, upon reading an A, the Turing machine could enter state 2 or 5, etc.

Remember that nondeterministic Turing machines will be used in the same theoretical way that nondeterministic algorithms are: to solve NP problems in polynomial time. In a nondeterministic Turing machine, it would somehow *know* the best route to take, and thus could solve the problem in polynomial time; if it didn't, it would take exponential time to try all the configurations.³¹

What do Turing machines have to do with the question, does $P = NP$? Since deterministic Turing machines can solve (in polynomial time) any member of the P class and nondeterministic Turing machines solve the NP problems, this question of $P = NP$ reduces to working with Turing machines alone. If we could find a deterministic Turing machine that runs in polynomial

³⁰Harel, Algorithms: The Spirit of Computing. 224.

³¹Ibid, 239.

time to solve any one of the NP problems, we would have conclusive evidence that $P = NP$; or, if we could prove that no such deterministic Turing machine exists, that would substantiate the view that $P \neq NP$.³² Obviously, though, no one has shown either of these, yet.

Universal Turing Machines

To complete our discussion of Turing machines, we need to introduce the idea of a Universal Turing machine (UTM). Basically, this is a Turing machine that will do the work of any other Turing machine. To accomplish this, we simply encode a particular Turing machine and input it, followed by its (the initial Turing machine's) inputs, to the UTM.³³ The UTM then simply imitates the Turing machine that was input to it and does exactly what the initial Turing machine would have done. At first, it may seem unreasonable to encode all the parts of a Turing machine (the tape, tape head, etc.) but all we actually need to encode is the Turing machine's *program*. Remember the program instructs the Turing machine through each of its various steps (see figure 6), so to represent the Turing machine, we need merely to represent the program in a form that the UTM can recognize.

The importance of this concept, much like that of Turing machines in general, lies in its theoretical application. The UTM represents a machine, *one* machine, capable of doing the work of *any* computer system, now or in the

³²Harel, Algorithms: The Spirit of Computing, 239.

³³Ibid, 232.

future. Granted, it would certainly not be overly efficient, but the power of such a machine should not be underestimated.

In Chapter 4, we will consider a *general* Turing machine, which should not be confused with the UTM. A general Turing machine simply means that machine which solves a particular problem, whereas the UTM is one that will solve *any* problem. We will not have further occasion to discuss UTMs in this paper; however a discussion of Turing machines would not be complete without introducing this concept.

Parity Check Turing Machine

To get a better feeling for Turing machines, I will present two examples. The first is a Turing machine that checks to see if there is an even number of 1's on an input tape consisting of a series of 1's and 0's, and the second, more extensive example can be found in the appendix.

The first Turing machine is named **parity check** because it could be used to check for even parity on a group of eight bits (a bit is a 1 or 0), as in figure 9. This is a very useful application used to check the accuracy of digitized data that is sent between two computers.



Fig. 9 Input Tape for Parity Check Turing Machine

The parity check Turing machine will answer Y or N to the question, 'Is there an even number of 1's in the series of 1's and 0's of the input tape?' It will

start with the tape head positioned over the first bit and end two cells after the last bit, with that cell containing either a Y or N. The next item we need is a program to tell the tape head what to do at each state when it encounters certain symbols (in our case, the symbols are 1, 0 and a blank square, which is signified by '-'). Remember that the program takes the information of (state, symbol) and outputs a (new state, new symbol, direction of movement). See figure 6 again. Therefore, all we need for the program is a table with precisely this information, as in figure 10.

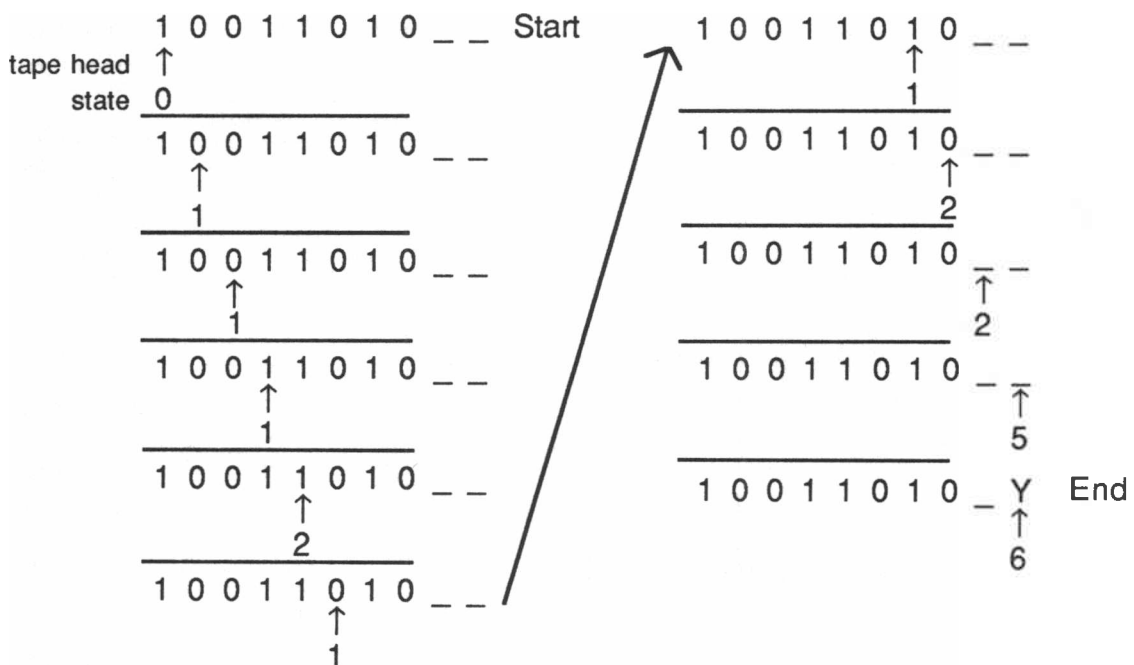
	State	Symbol read	New symbol	Direction	New state	
Initial	0	-	-	R	0	
	0	0	0	R	2	
	0	1	1	R	1	
	1	-	-	R	3	
	1	0	0	R	1	
	1	1	1	R	2	
	2	-	-	R	5	
	2	0	0	R	2	
	2	1	1	R	1	
	3	-	N	S	4	Stop with answer 'N'
	5	-	Y	S	6	Stop with answer 'Y'

Fig. 10 Program Table for Parity Check Turing Machine

Note that states 3 and 5 only account for the case of reading a blank in the cell. This is the only case that is necessary, since the tape head will only reach these states when it is at the end of the input. Thus, it will never be the case that a 1 or 0 is read while the Turing machine is in state 3 or 5.

To see how this Turing machine works, take the input tape of figure 9 and follow the series of steps outlined in the program section of the explanation of

the four characteristics of Turing machines previously described. For instance, at the initial set up, the Turing machine will be in state 0, with the tape head positioned over the first bit. By consulting the program table, we see that for the situation of being in state 0 and reading a 0, the Turing machine will leave the 0 in that cell, move one cell right (R) and proceed to state 2; however, if the symbol it read was a 1, the Turing machine would leave the 1 there, move one cell right and proceed to state 1. Here are what the steps of this Turing machine would look like, assuming an input of 10011010:



Another way to visualize Turing machines is through the use of **state diagrams**, as found in a software program called Turing's World (developed by Jon Barwise and John Etchemendy of Stanford University). The state diagram for the parity check Turing machine would be depicted as in figure 11.

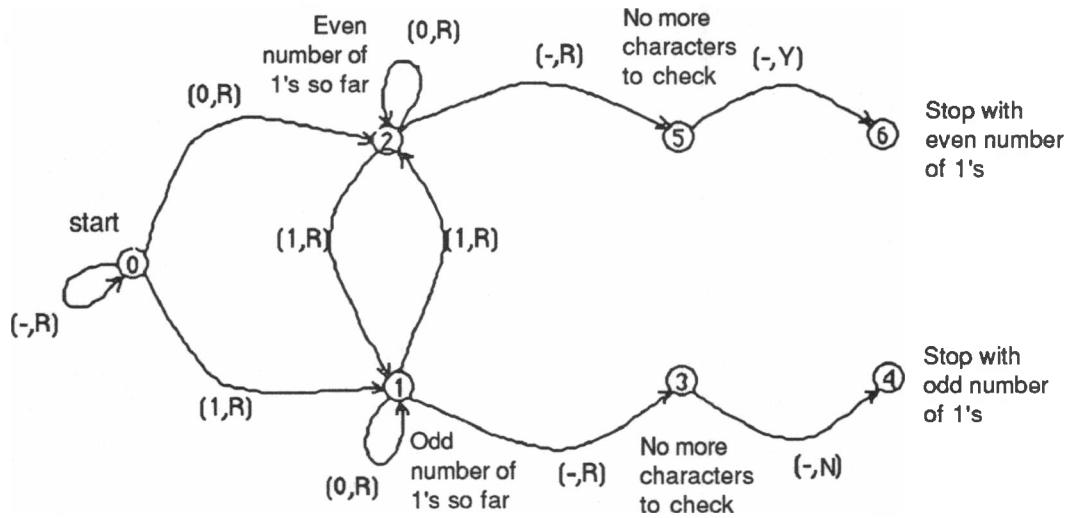


Fig. 11 State Diagram for Parity Check Turing Machine

In this diagram, the states are represented as circles and the steps of the program as ordered pairs linked to transition arrows. For example, the ordered pair $(1,R)$ joined with the arrow from state 0 to state 1 would be interpreted as, 'when the Turing machine is in state 0 and reads the symbol 1, it should leave that symbol there, move one cell to the right, and proceed to state 1.' If you step through the same example using this state diagram, you will find that you end up with the answer Y, just as you did with the program table.

Note that this Turing machine is a generic way of determining if an even or odd number of a certain character is present. It could be used in any application for which this information is needed; a parity check is merely one example. This machine is a fairly simple minded and relatively easy example to understand. For a more in-depth look at Turing machines and the software program Turing's World, consult the appendix.

CHAPTER FOUR - COOK'S THEOREM

We now turn to the proof of **Cook's Theorem**. We have been learning about NP-complete problems and their main characteristic that every other problem in the NP class is reducible to them, but where did it all start? How do we know there are any NPC problems at all? There must be one problem that is provably NPC and hence to which all other NP problems reduce; a very *first* NP-complete problem. There is indeed - it is called the *satisfiability problem* and was first proven to be NP-complete in 1971 by Stephen A. Cook. The rest of this chapter on Cook's Theorem will be divided into two parts: the first will describe the satisfiability problem and give examples, and the second will prove Cook's Theorem, that satisfiability is NP-complete.

Satisfiability Problem

The satisfiability problem has to do with determining the truth or falsity of sentences in propositional calculus. There are a number of terms we need to discuss before we can fully comprehend propositional sentences. Here is a list of definitions:

- Variable - $x_1, x_2, x_3, \dots, x_n$
each variable has a value of **T**(true) or **F**(false)

- Connective - not (\neg), and (\wedge), or (\vee)
each of these connects variables or groups of variables together

- Literal - variable either with or without the \neg connective
 $x_i, \neg x_i$
- Clause - group of literals connected with the \vee connective
 $(x_1 \vee x_2), (\neg x_3 \vee x_4)$
- Sentence - group of clauses connected with the \wedge connective
 $(x_1 \vee x_2) \wedge (\neg x_3 \vee x_4)$
 $(\neg x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_4 \vee x_5)$

Evaluation rules for connectives

- \neg (not) - negates the value of the variable
 $\neg T = F, \neg F = T$
- \wedge (and) - evaluates to T if and only if all of the literals are T
 $T \wedge T = T, T \wedge F = F, F \wedge T = F, F \wedge F = F$
- \vee (or) - evaluates to T if at least one of the literals is T
 $T \vee T = T, T \vee F = T, F \vee T = T, F \vee F = F$

With these in mind, we can construct a propositional sentence such as

$$(x_1 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_3 \vee x_4)$$

We're interested in assigning values to the literals so that the entire sentence turns out to be T. If this is possible (it may not be), we say the sentence is **satisfiable**.³⁴

Let's use the example above to demonstrate how this works. We will assign two different sets of values for the literals and work them through to see the outcome.

Ex. a) Let $x_1 = T$ $x_2 = T$ $x_3 = T$ $x_4 = F$
 Then
 $(x_1 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_3 \vee x_4)$ becomes
 $(T \vee T) \wedge (F \vee T \vee T) \wedge (F \vee F)$ by substitution
 $T \wedge T \wedge F$ by evaluation rules
 F

³⁴Baase, Computer Algorithms: 322.

Ex. b) Let $x_1 = T$ $x_2 = T$ $x_3 = F$ $x_4 = F$
 Then
 $(x_1 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_3 \vee x_4)$ becomes
 $(T \vee F) \wedge (F \vee T \vee T) \wedge (T \vee F)$ by substitution
 $T \wedge T \wedge T$ by evaluation rules
 T

In the first example, the sentence computes to F, and so is not yet satisfied. With the second set of assignments however, the sentence turns out to be T, and therefore, the sentence is satisfiable.

The **satisfiability problem**, or **SAT** for short, is stated in the following manner: given a sentence, is there an assignment of values to the literals such that the sentence evaluates to T?³⁵ This problem was proven to be the very first NP-complete problem, and in the next section, we will see how.

Cook's Theorem

Cook's Theorem is very easily stated, but much more difficult to prove. The theorem says:

Satisfiability is NP-complete

The proof of this theorem will consist of two parts, highlighting the two conditions of NP-completeness: (1) SAT is a member of the NP class, and (2) every problem in NP is reducible to SAT in polynomial time.

Member of NP class. Let's look at each of the four characteristics of NP problems (see page 10) to see that SAT is indeed a member of this class. SAT can be (and was) stated as a decision problem, so the first criterion is fulfilled.

³⁵Harel, Algorithms: The Spirit of Computing, 165.

The second condition states that there must be an obvious naive solution, but one that would take super-polynomial time. The naive solution to the SAT problem is simply to try each of the possible combinations of truth assignments to the variables to see if any of them make the sentence true. How many combinations are there? Each of the n variables has 2 possible choices (T or F), so there would be 2^n possible combinations, which we know is a super-polynomial number. So even given a polynomial time algorithm to check each combination, it would still take super-polynomial time to check a super-polynomial number of combinations!

A certificate that is checkable in polynomial time is the third criterion, and that is evident with the SAT problem. The certificate would be a certain assignment to the variables; the check out procedure simply to run the variables through using the evaluation rules for the connectives. Even for a large, polynomial number of variables, this is easily done in polynomial time.

The fourth and last criterion says that there must be a nondeterministic polynomial time solution to the problem. This is also not too difficult to conceive, if you imagine that during the assignment process, the algorithm just happens to *know* whether to assign T or F to each of the literals. Once this is done, and keep in mind this would only take polynomial time, the same polynomial time algorithm used earlier could be utilized to test the values and come up with an answer of T or F for the sentence. Hence, SAT satisfies the four conditions given earlier, and is in NP.

Polynomial Reducibility. In order for SAT to be NP-complete, every problem in NP must be reducible to it in polynomial time. Throughout the proof of this we will be examining NP problems, instances of the problems, and certificates for the instances. Let's be clear on what we mean by each of these:

- **NP problem** refers to an arbitrary problem in this class, for example the Traveling Salesman or Bin Packing problem
- **Instance** of the NP problem refers to a particular example of that problem, for example a certain map of cities or a certain number of bins and items
- **Certificate** for the instance of the NP problem refers to a proposed solution to that instance, for example a path through the cities or a list of items for each bin

To prove reducibility, we will use a general Turing machine that can solve any NP problem. We will take an instance and corresponding certificate of an arbitrary NP problem and consider a number of necessary conditions for a Turing machine that would check this certificate. By describing these conditions in terms of clauses of the SAT problem, we will come to the conclusion that the Turing machine will **accept** (reach a final state of 'yes') the instance and certificate of the NP problem *if and only if* all of the clauses are simultaneously satisfied. To prove *polynomial* reducibility, we will show that describing these Turing machine conditions in SAT terms takes at most polynomial time.³⁶ Thus we will have shown that any arbitrary NP problem is reducible to the SAT problem in polynomial time; therefore SAT is NP-complete. Before getting to this, there are a few concepts to learn.

As we proceed through this proof we will be concerned with time, since we need the reduction to be completed in polynomial time. To this end, we need to know a little bit about the **big-Oh notation**. This notation, symbolized by $O(f(n))$, provides an upper bound for the running time of algorithms. The formal definition is this:

³⁶Wilf, Algorithms and Complexity, 197.

A function, $g(n)$, is said to be $O(f(n))$, read “big-oh of $f(n)$ ”, if there exists constants n_0 and c_0 such that $|g(n)| < |c_0 f(n)|$ for all $n > n_0$.³⁷

This means that $g(n)$ is bounded above by $f(n)$, and consequently, the running time of $g(n)$ will be no longer than $f(n)$. We will be using this concept to distinguish polynomial from super-polynomial time, with $O(P^k(n))$ (k is a fixed constant) representing a bound of polynomial time.

Let's consider the relationship between a Turing machine and an NP problem. If we simply input an instance of the NP problem to the Turing machine, it will take the machine super-polynomial time to reach its final state, as we observed in Chapter Three. However, if we input a *certificate* along with the instance of the problem, the Turing machine merely has to *check* that this proposed solution does or does not work. This checking procedure, as was noted earlier in this chapter, is easily completed in polynomial time. In our proof of Cook's Theorem, we are considering instances and certificates of NP problems, thus our Turing machine runs in polynomial time.

Let's define explicitly the models we will be using in our proof:

- Let **Q** be an arbitrary NP problem
 - I** be any instance of Q
 - C** be a certificate corresponding to I
 - TMQ** be a Turing machine that accepts I and C in polynomial time
 - P(n)** be the maximum number of steps ($P(n)$ is a polynomial) TMQ takes to accept I and its certificate.³⁸
- Recall from Chapter 2 that n is *roughly* the size of the input, which in our case is the instance I

³⁷Sedgewick, Algorithms, 72.

³⁸Wilf, Algorithms and Complexity, 197.

Since we want to describe the Turing machine in terms of the SAT problem, we need *variables* to which we will assign T or F in such a way that the clauses constructed from these variables will be T if and only if TMQ accepts I and its certificate C. The three variables we need are described below:

1. $Q_{i,k}$ = T(rue) if after step i, TMQ is in state q_k
2. $T_{i,j}$ = T(rue) if after step i, the tape head is over square j
3. $S_{i,j,k}$ = T(rue) if after step i, symbol k is in square j³⁹

Let's take a moment to discover how many of these variables there are. We won't be able to calculate the exact number, but by using the big-Oh notation, we can determine whether the number is a polynomial or not, which is really what we're interested in, since we need *polynomial* reducibility. First of all, remember that since Q is an element of NP, there exists a Turing machine, TMQ, that will accept or reject an instance of Q and its certificate in at most, polynomial time. That is to say it will take TMQ less than or equal to $P(n)$ steps to come to a 'yes' or 'no' answer. Index i counts these steps, so it is on the order of $P(n)$, or $O(P(n))$ in big-Oh notation. A *step* in a Turing machine is really a movement of the tape head to an adjacent square, so if the maximum number of steps is $P(n)$, then the maximum number of squares the tape head could move is also $P(n)$. Thus the index, j, that counts the squares must be $O(P(n))$. Finally, index k counts the number of states of the Turing machine, which is a finite, constant number, so it doesn't change the order of the final tabulation of variables. Notice that we have been counting the indices i, j and k, yet we want the total number of variables Q,S and T. We can apply our findings for the indices to the actual variables since they are direct consequences of the

³⁹Wilf, Algorithms and Complexity. 198.

indices. That is to say that corresponding to each pair (i,k) is a certain value of Q , corresponding to each triplet (i,j,k) is a certain value of S , and so on. Thus stating that the number of i indices is $O(P(n))$ is the same as saying the number of variables of type $Q_{i,k}$ is $O(P(n))$. Both $S_{i,j,k}$ and $T_{i,j}$ are $O(P^2(n))$ because i and j are $O(P(n))$, and since they are subscripts to the same variable, we multiply the orders. Therefore the total number of variables is: $\text{order}(Q_{i,k}) + \text{order}(S_{i,j,k}) + \text{order}(T_{i,j}) = O(P(n)) + O(P^2(n)) + O(P^2(n)) = O(P^2(n))$, a polynomial number.⁴⁰ These big-Oh calculations are extremely important because they will help to determine whether the polynomial time factor actually exists.

Our task is to construct, from these three types of variables, a group of propositional clauses that are all simultaneously satisfied only when TMQ accepts an instance (along with its certificate) of an NP problem. This is where the simplicity of the TM is essential. It is because the steps of a Turing machine are so precisely defined that we can explicitly state all the necessary conditions that will lead the Turing machine to an accepting calculation.⁴¹

Once we state these conditions (there are seven in all) in terms of satisfiability clauses, we will be assured that whatever set of truth assignments to the variables we find to simultaneously satisfy all seven of these clauses will necessarily define an accepting calculation of TMQ on the instance (and certificate) of NP problem Q . Following is a list of the seven conditions:

1. At each step, TMQ is in at least one state
2. At each step, TMQ is in only one state
3. At each step, each cell contains exactly one symbol from the set of symbols
4. At each step, the tape head is above exactly one square

⁴⁰Wilf, Algorithms and Complexity. 198.

⁴¹Ibid.

5. Initially, TMQ is in state 0, the tape head is positioned over square 1, the instance, I , is encoded into squares $1, 2, \dots, n$ and the certificate, C , is encoded into squares $-1, -2, \dots, -P(n)$
6. The final step, $p(n)$, will find TMQ in state q_y , the accepting state
7. At each step, TMQ moves to its next(state, symbol, head position) according to the Turing machine's program

For each of these seven, I will first describe them in terms of SAT, then give the big-Oh for the clauses.

1. At each step, TMQ is in at least one state

This can be stated with the following clause:

$$(Q_{i,1} \vee Q_{i,2} \vee Q_{i,3} \vee \dots \vee Q_{i,K})$$

where K is the number of states.

This says that after step i , the Turing machine is either in state 1 ($Q_{i,1}$), or state 2 ($Q_{i,2}$), and so on, which describes exactly what we wanted to describe. How many clauses are there? Since i can take $O(P(n))$ values, and it is the only variable, there will be $O(P(n))$ clauses.

2. At each step, TMQ is in only one state

This can be stated in terms of two distinct states, j' and j'' , in this way:

$$(\neg Q_{i,j'} \vee \neg Q_{i,j''})$$

We can understand this much better with a table of truth values, seen below.

$Q_{i,j'}$	$Q_{i,j''}$	$(\neg Q_{i,j'} \vee \neg Q_{i,j''})$
T	F	T
F	T	T
F	F	T
T	T	F

Notice that the only time that $(\neg Q_{i,j'} \vee \neg Q_{i,j''}) = F$ occurs when TMQ is attempting to be in state j' and j'' at the same time, which is what we want to avoid. Again, the number of clauses

will be $O(P(n))$, since i is $O(P(n))$ and the number of states is constant.

3. At each step, each cell contains exactly one symbol from the alphabet

We will represent this as a collection of clauses much like those in characteristics 1 and 2 because if you think about it, characteristic 3 has two parts: there must be at least one symbol in each square, and there must be only one symbol in each square. This gives:

- 1) $(S_{i,j,1} \vee S_{i,j,2} \vee \dots \vee S_{i,j,r})$ - at least one
 where r is the number of symbols
 2) $(\neg S_{i,j,k'} \vee \neg S_{i,j,k''})$ - only one

Since both i and j vary in these clauses, the total number of clauses will be $O(P^2(n))$.

4. At each step, the tape head is above exactly one square⁴²

In the same spirit as above, we write this as:

- 1) $(T_{i,1} \vee T_{i,2} \vee T_{i,3} \vee \dots \vee T_{i,p(n)})$ - at least one
 where $p(n)$ is the max number of squares
 2) $(\neg T_{i,j'} \vee \neg T_{i,j''})$ - only one

The number of clauses will be, as you can probably guess, $O(P(n))$.

5. Initially, (a) TMQ is in state 0, (b) the tape head is positioned over square 1, (c) the instance, I , is encoded into squares $1, 2, \dots, n$, and (d) the certificate is encoded into squares $-1, -2, \dots, -P(n)$

(a) State 0 is expressed by:

$$(Q_{0,0})$$

(b) One literal is used for the initial position also:

$$(T_{0,1})$$

⁴²Wilf, Algorithms and Complexity, 199.

(c) The size of the input, l , is roughly n (from the given model assumptions), and since there is one symbol per square, this takes n squares. We will represent the particular symbols of the initial input as $k_{11}, k_{12}, \dots, k_{1n}$. This can be stated as:

$$(S_{0,1,k_{11}} \wedge S_{0,2,k_{12}} \wedge \dots \wedge S_{0,n,k_{1n}})$$

Notice the literals are connected with \wedge rather than \vee , since it must be the case that all of these are T.

(d) C takes $P(n)$ squares because C is the description of the *solution* to the instance, and from our model assumptions, we assumed that TMQ checks the solution in less than or equal to $P(n)$ steps. We will delineate the symbols of C as $k_{C1}, k_{C2}, \dots, k_{Cn}$, so this becomes:

$$(S_{0,-1,k_{C1}} \wedge S_{0,-2,k_{C2}} \wedge \dots \wedge S_{0,-P(n),k_{Cn}})$$

There are no variable subscripts in these initial conditions; both n and $P(n)$ are constants. Thus the total number of clauses for this condition is constant, which is represented in big-Oh notation as $O(1)$.

6. The final step, $p(n)$, will find TMQ in state q_y , the accepting state

This takes only one literal to express:

$$(Q_{p(n),y})$$

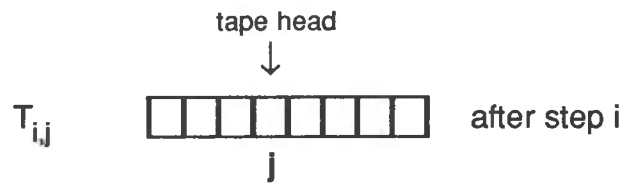
7. At each step, TMQ moves to its next(state, symbol, head position) according to the Turing machine's program⁴³

This also takes two parts to describe. The first is simply putting on paper what is very elementary and logical, that the symbol in the j^{th} square cannot change unless the tape head is positioned above that square. This can be stated as:

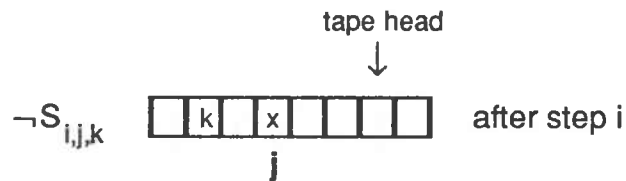
$$1) (T_{i,j} \vee \neg S_{i,j,k} \vee S_{i+1,j,k})$$

⁴³Wilf, Algorithms and Complexity, 200.

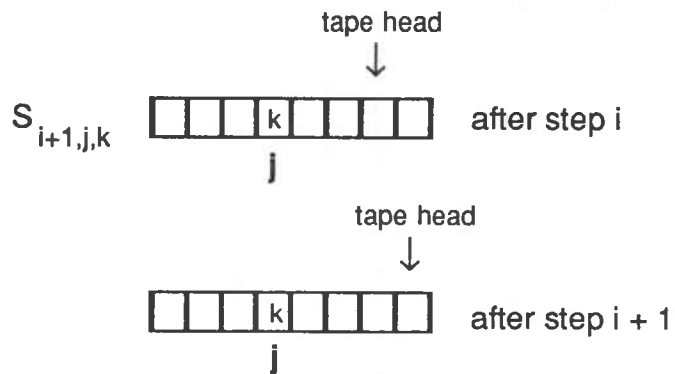
This doesn't seem to be saying the same thing at all, but let's look more closely. There are only two circumstances for the tape head: either it is over square j , or it isn't. $T_{i,j}$ symbolizes the case that it is; the other two literals mean it isn't. Now, if the tape head is *not* over square j , then one of two cases must be true: either symbol k is not in square j after step i ($\neg S_{i,j,k}$), or symbol k is *still* in square j after the *next* step ($S_{i+1,j,k}$). See figure 19. Once you think about it, you will realize that these take care of all the possible cases surrounding symbols and the tape head, and thus does express what we intended.



(a) tape head is over square j after step i



(b) symbol k is not in square j after step i



(c) symbol k is still in square j after step $i + 1$

Fig. 12 Possibilities for Symbol k and Square j

The second part expresses the point that each successive configuration comes through the program.

$$\begin{aligned}
 2) \text{ a) } & (\neg T_{i,j} \vee \neg Q_{i,k} \vee \neg S_{i,j,r} \vee T_{i+1,j+\text{INC}}) \wedge \\
 & \text{ b) } (\neg T_{i,j} \vee \neg Q_{i,k} \vee \neg S_{i,j,r} \vee Q_{i+1,k'}) \wedge \\
 & \text{ c) } (\neg T_{i,j} \vee \neg Q_{i,k} \vee \neg S_{i,j,r} \vee S_{i+1,j,r'})^{44}
 \end{aligned}$$

These three clauses are identical except for the last literal in each. The first three literals express the possibility that after step i , either the tape head is not over square j , or TMQ is not in state q_k , or square j does not contain symbol r . However, if this is the configuration, then after the next $((i+1)^{\text{st}})$ step,

- a) the tape head is over square $j+\text{INC}$ (INC means one square left or right)
- b) TMQ is in state $q_{k'}$
- c) symbol r' is in square j

The job of the program is to give the Turing machine its next symbol, state, and head position, and the last literal in each line does indeed express this fact. Thus, we have taken care of all of the possibilities for the program. There is certainly a polynomial number of clauses for this last characteristic of Turing machines; there is one clause for each step $i = 1, 2, \dots, p(n)$, each state q_k , each symbol r , and each square $-p(n), \dots, p(n)$. We know this is at most a polynomial number, since each of these parts is itself polynomial.

Now that we have come to the end of our transformations from Turing machines to SAT sentences, it is time to summarize our findings. The seven conditions above adequately describe a general Turing machine, and the SAT sentences show that these conditions can indeed be transformed to the SAT problem. The purpose of showing the polynomial size of all the conditions is to prove that this reduction can be accomplished in polynomial time. As long as the input size is polynomial, the reduction can take place in polynomial time.

Again, this is what we have shown through the construction of these seven conditions: if we input any instance (along with a certificate) of an NP

⁴⁴Wilf, Algorithms and Complexity. 200.

problem to TMQ and the Turing machine ends with a 'yes' answer, then it follows that there exists a set of truth assignments to the variables of these previous seven conditions such that all seven will be simultaneously satisfied; conversely, if we find a set of truth assignments to the variables such that all seven conditions are simultaneously satisfied, then we will know that this particular instance and certificate of the NP problem will cause the Turing machine to end in a 'yes' answer.

We have thus shown that SAT is a member of the NP class of problems, and that every other problem in NP is reducible to it in polynomial time. Hence, SAT is NP-complete.

CHAPTER FIVE - POLYNOMIAL REDUCTIONS

One purpose of a polynomial reduction is to show that a certain problem belongs to the NP-complete group of problems. In Chapter Two, we discovered that one of the characteristics of an NP-complete problem is that *every* problem in NP is reducible to it in polynomial time. From the proof of Cook's Theorem in the preceding chapter, we also know that every problem in NP is polynomially reducible to the SAT problem. Therefore, to show that a certain NP problem, A, is a member of NPC, all we need to show is that SAT reduces to A in polynomial time. Since every NP problem reduces to SAT, and SAT reduces to A, then every NP problem reduces to A. Actually, we could choose to reduce *any* NPC problem to A in order to prove A's NP-completeness, since every problem in NPC is reducible to each of the others (see Chapter Two). In this section, we're only interested in discovering the technique used in such reductions, so we will assume that the problems in our sample reductions are known to be members of the NP class, i.e. they all possess the four characteristics (described in Chapter Two) needed to be members of the NP class. The sample reductions I will present are: (1) SAT to 3SAT, (2) SAT to 3-Coloring a Graph, and (3) Hamiltonian Circuit to Traveling Salesman.

3SAT is NP-Complete

Discussion. We will show that the 3SAT problem (which we have assumed is a member of NP) belongs to the NPC class by reducing (in polynomial time) the SAT problem to it. The 3SAT problem differs from SAT in

only one way: 3SAT clauses may have at most three literals, whereas SAT may have any number of literals in each clause. Thus, (x_1) , $(x_1 \vee x_2)$, and $(x_1 \vee x_2 \vee x_3)$ are all valid clauses in 3SAT, but $(x_1 \vee x_2 \vee x_3 \vee x_4)$ is not, since it contains more than three literals.⁴⁵

Proof. Consider an instance of SAT in which at least one of the clauses has four or more literals, i.e. at least one clause would have the form:

$$(x_1 \vee x_2 \vee \dots \vee x_k) \quad \text{where } k \geq 4 \quad (5.1)$$

The procedure is to reduce this SAT problem to a 3SAT problem in such a way that the 3SAT problem is satisfied **if and only if** the SAT problem is. To reduce SAT to 3SAT, we need to replace each clause in SAT that contains four or more literals with a number of clauses that contain *exactly* three literals. To do this, we introduce $k-3$ new variables: $z_1, z_2, z_3, \dots, z_{k-3}$. Using these, we shall create $k-2$ new clauses to replace each clause of the form (5.1) in this way:⁴⁶

$$(x_1 \vee x_2 \vee z_1) \wedge (x_3 \vee \neg z_1 \vee z_2) \wedge \dots \wedge (x_r \vee \neg z_{r-2} \vee z_{r-1}) \wedge \dots \wedge (x_{k-1} \vee x_k \vee \neg z_{k-3}) \quad (5.2)$$

For example, if the clause was of the form $\alpha = (x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6)$ so $k=6$, we would define the $k-3 = 6-3 = 3$ new variables z_1, z_2, z_3 and then replace α by the $k-2 = 4$ clauses

$$(x_1 \vee x_2 \vee z_1) \wedge (x_3 \vee \neg z_1 \vee z_2) \wedge (x_4 \vee \neg z_2 \vee z_3) \wedge (x_5 \vee x_6 \vee \neg z_3).$$

⁴⁵Wilf, Algorithms and Complexity. 201.

⁴⁶Ibid, 202.

We want to prove that the 3SAT problem is satisfied if and only if the SAT problem is. For the “if” part of the proof, we will show that: **if SAT is satisfiable, then 3SAT is also satisfiable.** If we assume that SAT is satisfiable, we know there exists at least one literal in each clause that is T. Since the literals are connected with \vee and the clauses with \wedge , this is the minimum configuration that could make the sentence T. For definiteness, let's confine ourselves to considering only one clause of the SAT sentence, namely the one of (5.1) (the transformations of the other clauses would be handled in the same manner). Let x_r be the one x_i in (5.1) that is T. Define the z_i 's in this way:⁴⁷

$$z_i = \begin{cases} T & \text{if } i \leq r - 2 \\ F & \text{if } i > r - 2 \end{cases}$$

Let's substitute these back into the $k-2$ clauses of (5.2):

$$(x_1 \vee x_2 \vee z_1) \wedge (x_3 \vee \neg z_1 \vee z_2) \wedge \dots \wedge (x_r \vee \neg z_{r-2} \vee z_{r-1}) \wedge \dots \wedge (x_{k-1} \vee x_k \vee \neg z_{k-3})$$

After the appropriate substitutions, and assuming x_r is the only x_i that is T, (5.2) becomes:

$$\begin{array}{cccccccc} (F \vee F \vee T) & \wedge & (F \vee F \vee T) & \wedge & \dots & \wedge & (T \vee F \vee F) & \wedge & \dots & \wedge & (F \vee F \vee T) \\ T & & T & & \dots & & T & & \dots & & T \\ & & & & & & & & & & T \end{array}$$

⁴⁷Wilf, Algorithms and Complexity. 202.

Thus, all $k-2$ clauses of (5.2) are satisfied when the corresponding clause of the SAT problem is, which completes the "if" part of the proof that 3SAT is NPC.

For the second ("only if") half of the proof, we need to show that: **if 3SAT is satisfiable, then SAT is also satisfiable**. If 3SAT is satisfiable, then *each* clause in (5.2) must have at least one literal that is T, and to show that SAT is satisfiable, we need to show that at least one literal for each of its clauses is T. Again, let's examine only the clause of (5.1). We will prove SAT is satisfiable by contradiction, by assuming that SAT is *not* satisfiable. Assuming this is the case, we know there are no x_i 's in (5.1) that are T. Notice in (5.2), (which, in this part of the proof, we are assuming to be satisfied), that none of the x_i 's are negated ($\neg x_i$) and yet those $k-2$ clauses are still T. This means that the x_i 's play no role here, and thus we can write (5.2) as:⁴⁸

$$(z_1) \wedge (\neg z_1 \vee z_2) \wedge \dots \wedge (\neg z_{k-4} \vee z_{k-3}) \wedge (\neg z_{k-3}) \quad (5.3)$$

Since each of these clauses is T (from given), it must be the case that $z_1 = T$. Then $\neg z_1 = F$, so to make the second clause T, $z_2 = T$. Likewise in the third clause $z_3 = T$, and so on up to the $(k-4)^{\text{th}}$ and $(k-3)^{\text{rd}}$ clauses. Notice the pattern in the 2-member clauses: the first member is F, the second T. Thus in $(\neg z_{k-4} \vee z_{k-3})$, $\neg z_{k-4} = F$ and $z_{k-3} = T$. The only way for the last clause, $(\neg z_{k-3})$, to be true is if $\neg z_{k-3} = T$, which implies that $z_{k-3} = F$, but now we have a contradiction since our assumption led us to claim that $\neg z_{k-3}$ is both T and F. Therefore, we must *reject* our assumption that there are no x_i 's which are T, which leads us to the conclusion that at least one x_i per clause is T, thus establishing the

⁴⁸Wilf, Algorithms and Complexity, 203.

satisfiability of SAT when 3SAT is satisfied. This completes the reduction of SAT to 3SAT; now we need to show that this can be done in polynomial time.

Let y be the number of clauses that contain more than three literals. By replacing each of these y clauses by $k-2$ clauses, we arrive at a sum of y polynomial clauses, which is still a polynomial. The determination of which z_i 's are T and which are F requires one decision statement (if $i \leq r$) for each of the $k-3$ z_i 's. This is easily completed in polynomial time, even for large k , and so both parts of the reduction process take polynomial time. Hence SAT to 3SAT is a polynomial reduction, and we have shown that 3SAT is a member of the NP-complete class of problems.

3-Coloring a Graph is NP-Complete

Discussion. Recall from Chapter Two the graph coloring problem: given a graph consisting of n points, we would like to assign colors to those n points such that two conditions are satisfied: (i) each point is colored with one and only one color and (ii) no two adjacent points have the same color. In the 3-coloring a graph problem, we want to properly color the graph using only three colors.

Proof. To prove that the 3-coloring a graph NP problem is NPC, we will reduce the SAT problem to it, much like we did in the previous reduction. Let C be the graph with n points, labeled C_1, C_2, \dots, C_n , and the three colors used be R(ed), B(lue) and Y(ellow). Let F be the SAT sentence consisting of $3n$ literals, one for each combination of point-color. The literals will be depicted as $C_{1,R}$

(which means point 1 is colored red), $C_{1,B}$, $C_{1,Y}$, and so on.⁴⁹ Our task is to construct the SAT sentence F in such a way that it is satisfiable **if and only if** graph C is properly colored. We will do this by constructing F so that it reflects the two necessary conditions given above for 3-coloring a graph.

(i) Each point is colored with one and only one color

In terms of SAT, this leads us to a group of n subsentences:

$$\begin{aligned} & (C_{1,R} \wedge \neg C_{1,B} \wedge \neg C_{1,Y}) \vee (C_{1,B} \wedge \neg C_{1,Y} \wedge \neg C_{1,R}) \vee (C_{1,Y} \wedge \neg C_{1,R} \wedge \neg C_{1,B}) \\ \wedge & (C_{2,R} \wedge \neg C_{2,B} \wedge \neg C_{2,Y}) \vee (C_{2,B} \wedge \neg C_{2,Y} \wedge \neg C_{2,R}) \vee (C_{2,Y} \wedge \neg C_{2,R} \wedge \neg C_{2,B}) \\ \wedge & \cdot \\ & \cdot \\ \wedge & (C_{n,R} \wedge \neg C_{n,B} \wedge \neg C_{n,Y}) \vee (C_{n,B} \wedge \neg C_{n,Y} \wedge \neg C_{n,R}) \vee (C_{n,Y} \wedge \neg C_{n,R} \wedge \neg C_{n,B}) \end{aligned}$$

Each one of these subsentences expresses the condition that for each point m , either C_m is colored red and not blue or yellow, or it is colored blue and not red or yellow, or it is colored yellow and not blue or red.

(ii) No two adjacent points have the same color

This can be expressed with the following sentence, which must be true for all m and n such that C_m and C_n are adjacent points:

$$\neg((C_{m,R} \wedge C_{n,R}) \vee (C_{m,B} \wedge C_{n,B}) \vee (C_{m,Y} \wedge C_{n,Y}))$$

⁴⁹Harel, Algorithms: The Spirit of Computing, 171.

The above sentence means that it is not the case that both C_m and C_n are red, or C_m and C_n are blue, or C_m and C_n are yellow.⁵⁰

To apply the term *polynomial reduction*, the reduction outlined above must be completed in polynomial time, which will occur if the size of the sentence F is at most, polynomially larger than the number of points of C . Part (i) above is linear in n , ($O(n)$), since the number of subsentences is n . Part (ii) is quadratic in n , ($O(n^2)$), in its worst case. This occurs when every point of C is adjacent to every other point, yielding n^2 pairs of (m,n) . Hence, sentence F is only polynomially larger than n , the number of points in graph C , and thus the above reduction occurs in polynomial time.⁵¹

The only remaining task in proving NP-completeness for 3-coloring a graph is to confirm that C is 3-colorable if and only if F is satisfiable. Part I: **if C is 3-colorable, then F is satisfiable.** Since we are given that C is 3-colorable, we know there exists an assignment of colors to the n points such that the two conditions in the discussion above are met. Because of the way we have constructed F , we can make it satisfiable simply by transferring the point coloring scheme of C to the literals of F .⁵² For example, if point 4 has color red, then let $C_{4,R} = T$, $C_{4,B} = F$, and $C_{4,Y} = F$. By doing a similar transformation for the rest of the points, we will be assured that F is satisfiable.

Part II: **if F is satisfiable, then C is 3-colorable.** Since we are given that F is satisfiable, we know there exists a set of truth assignments to the literals of F such that parts (i) and (ii) are both true. Again, because of the way we have constructed F , we simply transfer the truth assignments to the points of the graph C and we will be guaranteed that C is 3-colorable.⁵³ For instance, if

⁵⁰Harel, Algorithms: The Spirit of Computing. 172.

⁵¹Harel, Algorithms: The Spirit of Computing. 172.

⁵²Ibid.

⁵³Ibid.

$C_{3,B} = T$, then assign point 3 the color blue, and likewise for the rest of the points.

We have shown that the reduction between SAT and 3-coloring a graph can be completed in polynomial time, and that a graph is 3-colorable if and only if the SAT sentence F is satisfiable. This, along with the fact that 3-coloring a graph is a member of NP (Chapter Two), proves the assertion that 3-coloring a graph is NP-complete.

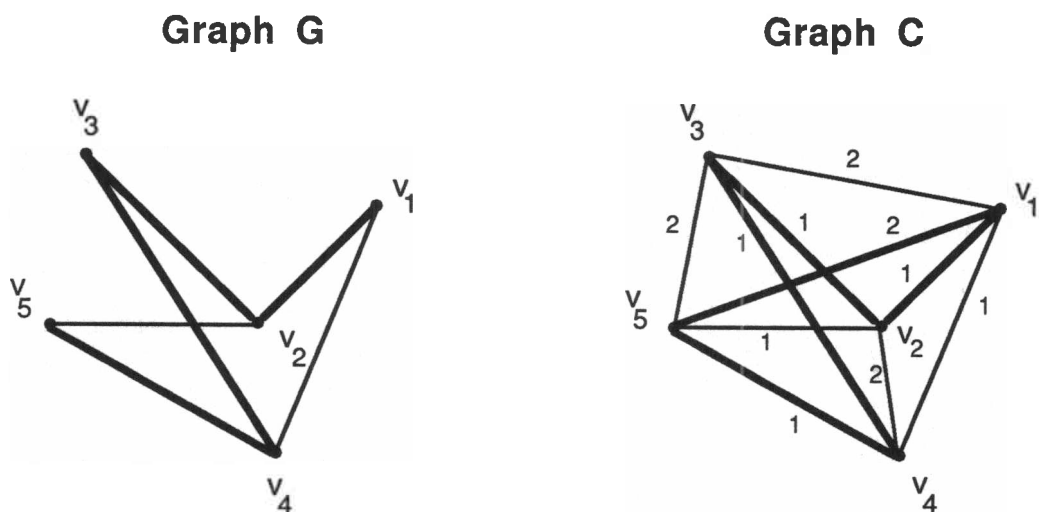
Traveling Salesman Problem is NP-Complete

Discussion. In the Traveling Salesman problem (as outlined in Chapter Two), the objective is to find a tour through a number of cities such that the total distance traveled is less than or equal to a target distance, x . To show that this problem is NP-complete, we shall reduce a known NPC problem, the Hamiltonian Circuit problem to it. The Hamiltonian Circuit problem is similar to the Traveling Salesman, but the objective here is not to find the *shortest* path, but to find a path through the cities such that each city is visited *once and only once*.⁵⁴ Another difference between these two problems is in their final destinations. In the Traveling Salesman problem, the starting point is also the final destination, so if we begin at city A and visit each of the other cities to the last city, Z , we then travel the last leg of the journey from Z to A . In the Hamiltonian Circuit problem, we end our journey at Z rather than going back to A , hence there is one less leg of the journey traveled.

⁵⁴Harel, Algorithms: The Spirit of Computing. 163.

Proof. Let G be a graph for which a Hamiltonian circuit exists. The vertices of G are denoted as v_1, v_2, \dots, v_m , where m is the number of vertices, or cities. We will create a graph C which is related to G and has a Traveling Salesman path whose length is less than or equal to $m+1$. We want to prove that the Traveling Salesman problem is NP-complete, so we will reduce the Hamiltonian Circuit problem to it in polynomial time and show that a Hamiltonian circuit for G exists **if and only if** the corresponding graph C (from the reduction) has a Traveling Salesman path whose distance is less than or equal to $x = m+1$. Here is the procedure we will use to reduce the Hamiltonian Circuit problem to the Traveling Salesman problem:

- (1) Copy the vertices from G to C
 - (2) On graph C , draw an edge between every pair of vertices, (v_i, v_k)
 - (3) If an edge of C also was present in G , then assign a distance of 1 to it; otherwise assign a distance of 2.
- (See figure 13 for an example)



Hamiltonian Circuit is emphasized with thick lines

Traveling Salesman path is emphasized with thick lines. Lines that were present in G have value 1, new lines have value 2.

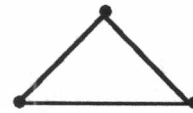
Fig. 13 Hamiltonian Circuit and Traveling Salesman Paths

Our first task is to confirm that this reduction can be completed in polynomial time. Our main concern is the number of edges we will have to draw for m vertices so that each vertex is connected to all of the others. Consider the following analysis: (see figure 14 for graphical representation)

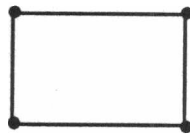
No. of vertices	No. of edges
2	1
3	3
4	6
5	10
⋮	⋮
⋮	⋮
⋮	⋮
m	$\frac{m(m-1)}{2}$



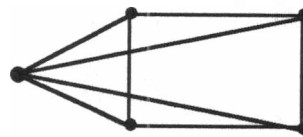
vertices = 2
edges = 1



vertices = 3
edges = 3



vertices = 4
edges = 6



vertices = 5
edges = 10

Fig. 14 Comparison of Vertices and Edges

Thus, for m vertices, we draw $\frac{m(m-1)}{2}$ edges, which is a polynomial number. All that is needed for the reduction is to decide whether each edge

should have a value of 1 or 2, which, for a polynomial number of edges will only take a polynomial amount of time. Hence, the reduction from the Hamiltonian Circuit to Traveling Salesman problem can occur in polynomial time.⁵⁵

Now we must show that G has a Hamiltonian circuit if and only if C has a Traveling Salesman path whose distance is less than or equal to $x = m+1$. **Part I: if G has a Hamiltonian circuit, then C has a Traveling Salesman path whose distance is less than or equal to $x = m+1$.** Let $\{v_1, v_2, \dots, v_m\}$ be the Hamiltonian circuit for G . Because of the way C is constructed, we know this path also exists on graph C , with each leg of the path having a value of 1. This makes a total distance so far of $m-1$ (there are $m-1$ edges for a path through m vertices). The only leg needed to complete a Traveling Salesman path is the one from v_m to v_1 , which has a value of 1 or 2. If the edge $v_m v_1$ existed on graph G , then it has a value of 1, making the total distance traveled = m . If the edge $v_m v_1$ did *not* exist on G , then its value is 2, which makes the total distance traveled = $m+1$. Thus, C has a Traveling Salesman path whose total distance is less than or equal to $m+1$.

Part II: if C has a Traveling Salesman path whose total distance is less than or equal to $m+1$, then G has a Hamiltonian circuit. Let $\{v_1, v_2, \dots, v_m, v_1\}$ be the Traveling Salesman path for C . There are m edges traversed along this path, each having a value of 1 or 2. Since the maximum distance traveled is given to be $m+1$, there are two possibilities for the distribution of values to the edges: (i) all m edges have values of 1, yielding a total distance of m , or (ii) the first $m-1$ edges have values of 1 and the m^{th} edge has a value of 2, yielding a total distance of $m+1$. In either case, the first $m-1$

⁵⁵Garey, Computers and Intractability. 36.

edges (which yield the path from v_1 to v_m) each have a value of 1, which implies they are also edges of graph G , and thus yield a Hamiltonian circuit for G .⁵⁶

We have now completed the proof that the Traveling Salesman problem is NP-complete. We reduced, in polynomial time, the Hamiltonian Circuit problem (which is NPC) to it, and demonstrated that a Hamiltonian circuit for a graph G exists if and only if a Traveling Salesman path (whose distance is less than or equal to a target, x) exists for the corresponding graph C .

⁵⁶Harel, Algorithms: The Spirit of Computing. 169.

EPILOGUE

In the previous chapter, the reductions we made connected similar NP-complete problems, but in fact, reductions can be made between *any* two NPC problems. In the multitude of NP-complete problems known to exist, there are problems that seem to have nothing at all to do with any of the others, yet we know there is a link, since they all reduce to each other. It is this link that makes these problems so very important for computer scientists to solve.

Research in the area of NPC problems is still flourishing today, with the main goal being to determine once and for all, if a polynomial time solution exists for these problems. Meanwhile, work continues in other directions too. One is the probabilistic approach mentioned earlier during the discussion of the primality problem. Another approach is to settle for algorithms that run in polynomial time but answer a less stringent question - i.e., relax the requirements of the problem slightly.

While it is true that few computer scientists believe that a polynomial time algorithm exists for these NPC problems, no one has yet been able to prove definitively that one does not. The person who finally answers this momentous question will most likely win the ACM Turing Award for that year, and rightfully so. While the answer itself may only interest a few people, the *consequences* of the answer will affect millions. If a polynomial time solution exists, the hundreds of seemingly disparate problems in this class that occur in our everyday lives will be instantly solved; if such a solution is proven not to exist,

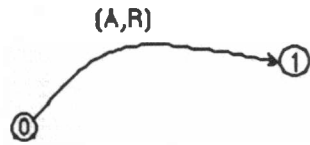
we can spend our research money on finding efficient approximation algorithms to help us deal with these problems. In either case, we will no longer have to divide our efforts between two areas of research for these problems, and consequently, we will be one step closer to solving these NPC problems.

APPENDIX

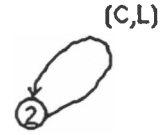
The following material is an extension of Chapter Three in which I gave an example of a problem (the parity check) that is solved using a Turing machine. In this section, I will discuss a more complicated example that is also solvable with a Turing machine. Using a computer program called Turing's World, I developed a Turing machine to solve a problem called the **Food Chain**. First, I'll give a bit more background on the Turing's World program, then on the food chain example itself.

Turing's World. There are a few fundamentals to learn before we get into the actual workings of the food chain program. First of all, the **machine** is simply the main routine that runs the program. A machine may (or may not) be made up of **submachines**, which we can associate with subroutines of a normal programming language; they carry out smaller, detailed tasks for the main machine. In Turing's World, a submachine is denoted by a square with a number in the center, which denotes the number of the submachine. As we noted in Chapter Three, each **state** in the machine is denoted by a circle with a number in it, and tasks get accomplished by moving among the various states. This moving, or change of states, is done through a **transition**, denoted by a curved arrow between two states. An ordered pair over the arrow tells the tape head its next move, and each ordered pair of a transition consists of two parts. The first component denotes the character just read, and thus is some member of the **alphabet** for that particular Turing machine, i.e. the set of symbols that can appear on the tape. In this case, the alphabet consists of { A,B,C,-,# }, where "-" signifies a blank square and "#" denotes the beginning and end of the input chain. The second half of the pair tells the tape head to do one of two

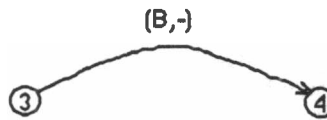
actions: move right or left (R or L), or write a character from the alphabet into the square and stay over that square. See figure 15 for some examples.



a) When the Turing Machine is in state 0 and reads an A, move the tape head one square right and go to state 1.



b) When the Turing Machine is in state 2 and reads a C, move one square left and stay in state 2.



c) When the Turing Machine is in state 3 and reads a B, replace the B with a blank and proceed to state 4. Note in this example the head does not move.

Fig. 15 Turing Machine Examples

Food Chain. This machine simulates the natural environment where the more dominant species eat the less dominant. In my program, there are only three species, A, B, and C, where A is the least dominant and C the most. The rules are as follows: any A preceded by a B will be eaten by the B because of its dominant status; likewise a B preceded by a C will also be eaten. Once an element has been eaten, the rest of the chain moves up one link to fill in the gap. The chain ends after all occurrences of the two combinations, BA and CB, have been eliminated. For example, the chain BACB will reduce to BC, since the A and last B will be eaten. The chain ABC will end as ABC because there are no target combinations.

At last, we get to the actual food chain Turing machine. My machine consists of four main subroutines:

- 0 - places the markers "#" at both ends of the initial input chain
- 1 - searches for BA sequences, deletes A, and concatenates
- 2 - searches for CB sequences, deletes B, and concatenates
- 3 - removes the markers from the ends of the chain

Submachine 0 adds a # marker to each end of the chain so that the Turing machine knows where the chain begins and ends. It starts at the beginning of the chain, which is input onto the tape before the program begins, and searches for the first blank. It keeps proceeding one square to the right until it reads a blank, which it replaces with a #. It then moves all the way left until it reaches the left hand blank, and replaces it with a #. See figure 16. You should walk through an example, perhaps the BACB or ABC from above, to see exactly how this works.

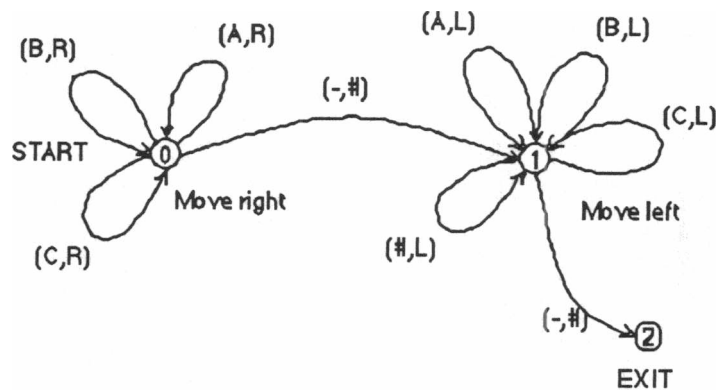


Fig. 16 Submachine 0

This adds the "#" marker to each end of the initial chain.

Submachine 1, pictured in figure 17, starts out by searching for a B, and if it doesn't find one, it simply passes control to the next submachine. Upon finding a B, though, it then looks to see if the next square holds an A, because that is the combination it needs to adjust; any other character following a B is of no concern.

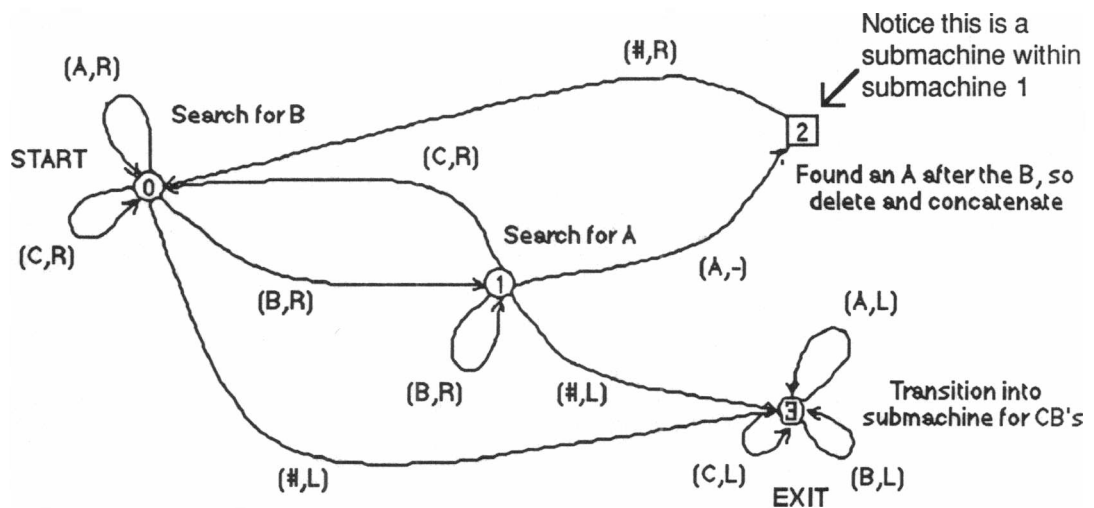


Fig. 17 Submachine 1

Notice the square numbered "2", indicating there is another submachine within this one. More on this in Fig. 18.

If it does find an A after a B, it deletes the A and goes to the next stage, which is to move all the rest of the characters one square to the left. This task is completed in another submachine *within* submachine 1. See figure 18.

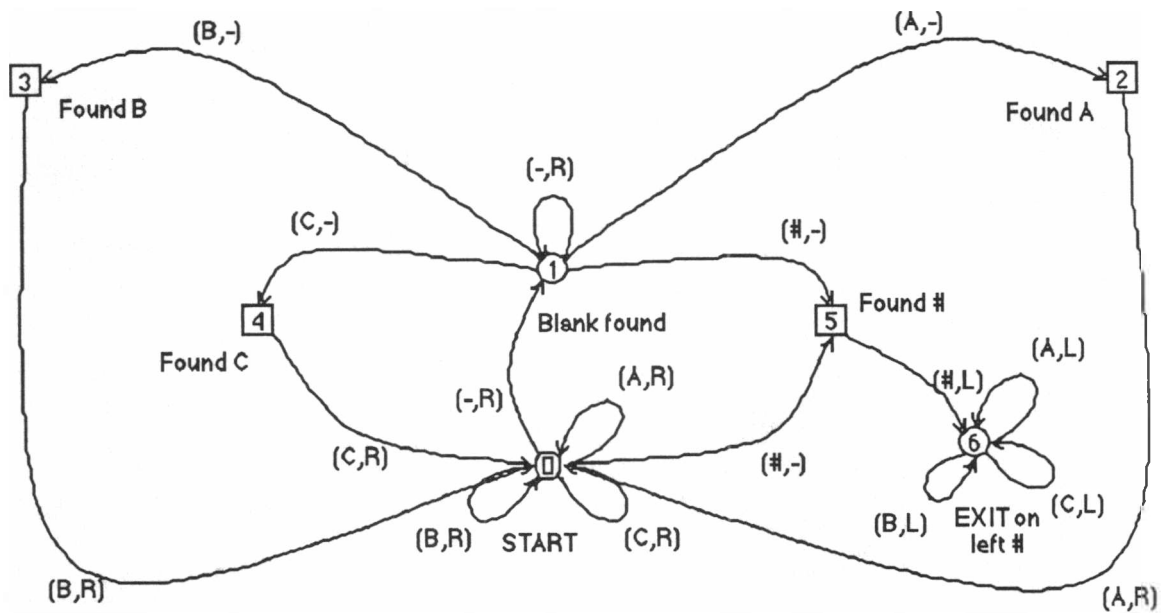


Fig. 18 Concatenation Submachine 2

This submachine deletes an A and concatenates the rest of the chain. It is located in square number 2 within submachine 1.

Notice in figure 18 that there are four other squares, indicating there are four more submachines within this submachine. Each of these does the same task, but on each of the characters A,B,C,#. When control comes to any of these submachines, one of the characters A,B,C, or # has just been replaced by a blank (-). Each submachine then moves the tape head left until it comes to any character; it then moves back one place to the right and inserts the character that was previously deleted. This has the effect of concatenating the characters after a member of the food chain has been eaten. See figure 19.

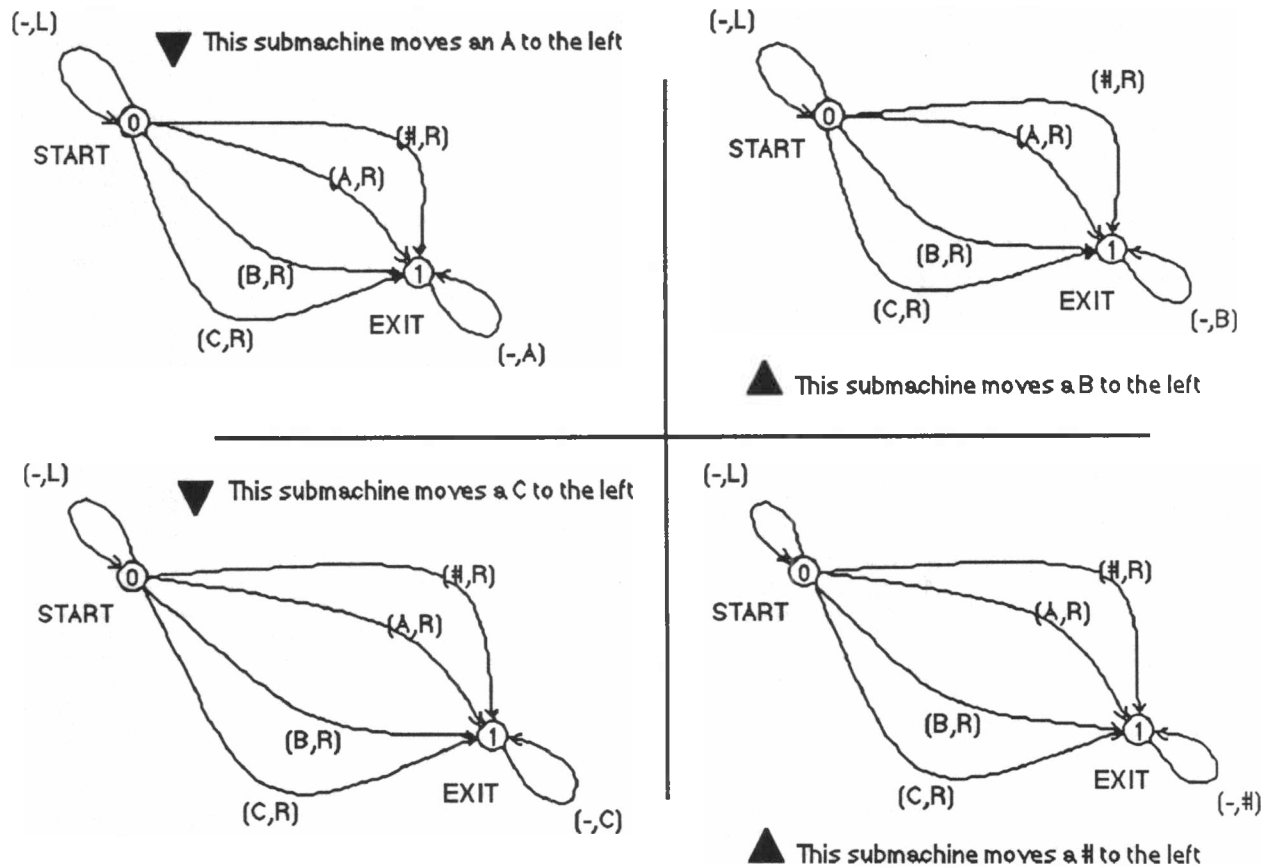


Fig. 19 Submachines to concatenate single characters

These are the farthest down in the levels of submachines, and are located in two places: within the concatenation submachine, which is within submachine 1, and within the concatenation submachine within submachine 2.

Submachine 2 works in the same way that submachine 1 does, except it searches for CB sequences. See figure 20. The submachine marked with a "2" in figure 20 works in exactly the same way as the concatenation submachine within submachine 1, as do the four submachines within "2". Thus, the two levels below submachine 2 are exactly the same as those in figures 18 and 19, so refer to these two figures for details.

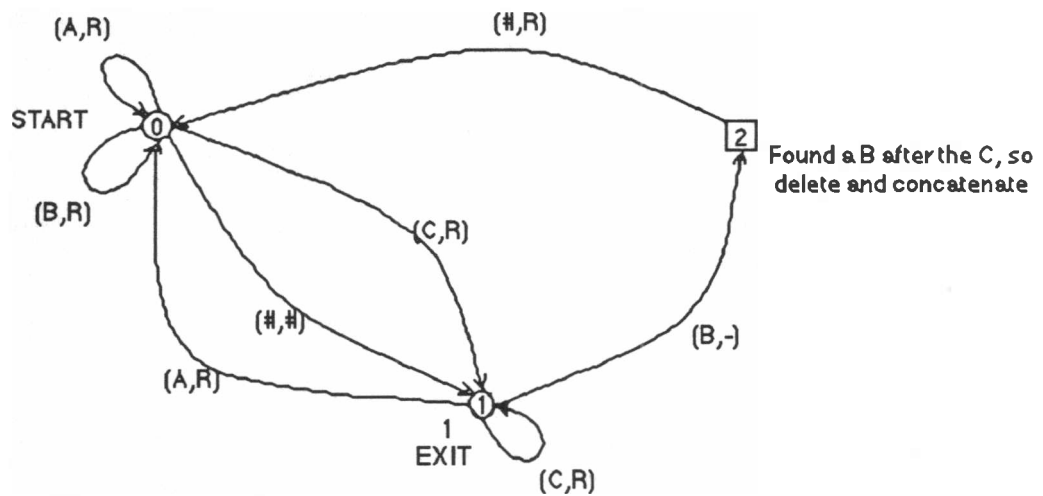


Fig. 20 Submachine 2

This submachine searches for CB's, and upon finding one, deletes the B and concatenates the rest. The submachine marked "2" is the same as the one in Fig. 18.

Submachine 3, the final one, simply searches for the markers at the ends of the chain and replaces them with blanks. See figure 21. Once this is done, the Turing machine halts with the ending chain visible on the tape.

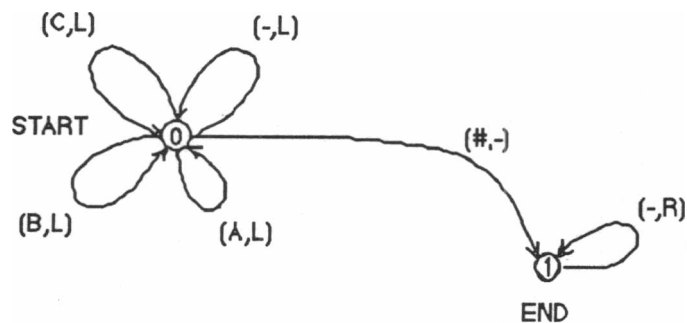


Fig. 21 Submachine 3

This submachine removes the two #'s from the ends of the chain.

As one final example of how this Turing machine works, I have put a sample input in Table 1 and shown all of the character changes the Turing machine will make before arriving at the end chain. You should use all of the diagrams on the previous pages and work through this example to get a better idea about how Turing machines work.

	<u>B</u>	<u>A</u>	<u>C</u>	<u>B</u>		Initial Chain
	B	A	C	B	#	
#	B	A	C	B	#	
#	B		C	B	#	
#	B			B	#	
#	B	C		B	#	
#	B	C			#	
#	B	C	B		#	
#	B	C	B			
#	B	C	B	#		
#	B	C		#		
#	B	C				
#	B	C	#			
#	B	C				
	B	C				Final Chain

Table 1 Sample input to the food chain Turing machine

BIBLIOGRAPHY

Books

- Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms. Reading, MA: Addison-Wesley Publishing Company, Inc., 1974.
- Ashenhurst, Robert L. and Susan Graham, eds. ACM Turing Award Lectures - The First Twenty Years. Reading, MA: Addison-Wesley Publishing Company, Inc., 1987.
- Baase, Sara. Computer Algorithms: Introduction to Design and Analysis. 2d ed. Reading, MA: Addison-Wesley Publishing Company, Inc., 1988.
- Barwise, Jon and John Etchemendy. Academic Courseware Exchange: Turing's World. Santa Barbara, CA: Kinko's Service Corporation, 1986.
- Brassard, Giles and Paul Bratley. Algorithmics: Theory and Practice. Englewood Cliffs, NJ: Prentice Hall, Inc., 1988.
- Garey, Michael R., and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. San Francisco, CA: W.H. Freeman and Company, 1979.
- Harel, David. Algorithms: The Spirit of Computing. Wokingham, England: Addison-Wesley Publishing Company, Inc., 1987.
- Hodges, Andrew. Alan Turing: The Enigma. New York, NY: Simon and Schuster, Inc., 1983.
- Sedgewick, Robert. Algorithms. 2d ed. Reading, MA: Addison-Wesley Publishing Company, Inc., 1988.
- Sommerhalder, R., and S.C. van Westrhenen. The Theory of Computability: Programs, Machines, Effectiveness and Feasibility. Wokingham, England: Addison-Wesley Publishing Company, Inc., 1988.
- Wilf, Herbert S. Algorithms and Complexity. Englewood Cliffs, NJ: Prentice-Hall International, Inc., 1986.

Articles

Cook, S.A. "The Complexity of Theorem Proving Procedures." Proc. Third Annual ACM Symposium on the Theory of Computing. ACM, New York, 1971: pp. 151-158.

Hopcroft, John E. "Turing Machines." Scientific American. 250 (May 1984), 86-99.