Lecture 25:
Cook's Theorem (1997)

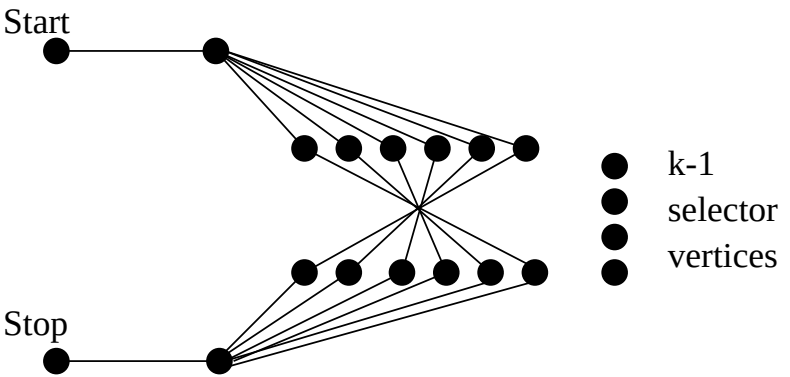
Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

Prove that Hamiltonian Path is NP-complete.

This is not a special case of Hamiltonian cycle! (G may have a HP but not cycle)
The easiest argument says that G contains a HP but no HC iff (x, y) in G such that adding edge (x, y) to G causes to have a HC, so $O(n^2)$ calls to a HC function solves HP.
The cleanest proof modifies the VC and HC reduction from the book:



This has a Hamiltown path from start to stop
iff the orignal graph had a vertex cover of size k.

Formal Languages and the Theory of NP-completeness

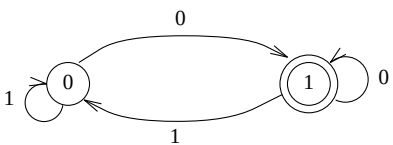
The theory of NP-completeness is based on formal languages and Turing machines, and so we will must work on a more abstract level than usual.

For a given alphabet of symbols $\Sigma = 0, 1, \&$, we can form an infinite set of *strings* or words by arranging them in any order: '&10', '111111','&&&', and '&'.

A subset of the set of strings over some alphabet is a *formal language*.

Formal language theory concerns the study of how powerful a machine you need to recognize whether a string is from a particular language.

Example: Is the string a binary representation of a even number? A simple finite machine can check if the last symbol is zero:

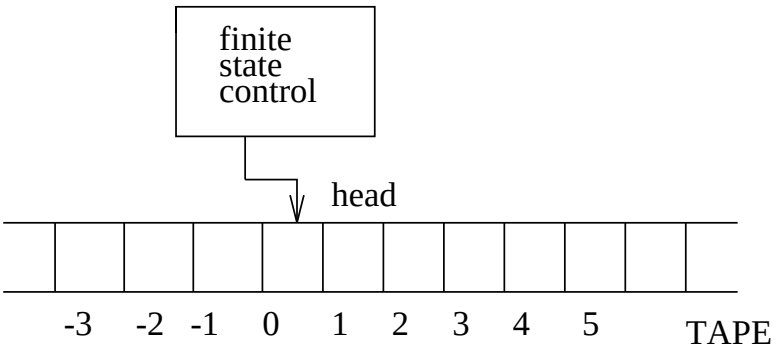


No memory is required, except for the current state.

Observe that solving decision problems can be thought of as formal language recognition. The problem instances are encoded as strings and strings in the language if and only if the answer to the decision problem is YES!

What kind of machine is necessary to recognize this language? A Turing Machine!

A Turing machine has a finite-state-control (its program), a two way infinite tape (its memory) and a read-write head (its program counter)


finite
state
control
head
-3
-2
-1
0
1
2
3
4
5
TAPE

So, where are we?

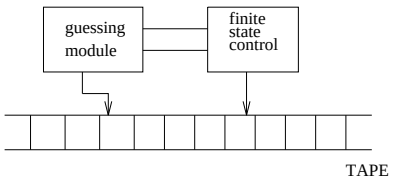
Each instance of an optimization or decision problem can be encoded as string on some alphabet. The set of all instances which return True for some problem define a language.
Hence, any problem which solves this problem is equivalent to a machine which recognizes whether an instance is in the language!
The goal of all this is going to be a formal way to talk about the set of problems which can be solved in polynomial time, and the set that cannot be.

Non-deterministic Turing Machines

Suppose we buy a *guessing module* peripherial for our Turing machine, which looks at a Turing machine program and problem instance and in polynomial time writes something it says is an answer. To convince ourselves it really is an answer, we can run another program to check it.

Ex: The Traveling Salesman Problem

The guessing module can easily write a permutation of the vertices in polynomial time. We can check if it is correct by summing up the weights of the special edges in the permutation and see that it is less than k.



The class of languages which we can recognize in time polynomial in the size of the string or a deterministic Turing Machine (without guessing module) is called P.

The class of languages we can recognize in time polynomial in the length of the string or a non-deterministic Turing Machine is called NP.

Clearly, $P \in NP$, since for any DTM program we can run it on a non-deterministic machine, ignore what the guessing module is doing, and it will just as fast.

P ?= NP

Observe that any NDTM program which takes time $P(n)$ can simulated in $P(N)2^{P(n)}$ time on a deterministic machine, by running the checking program $2^{P(n)}$ times, once on each possible guessed string.

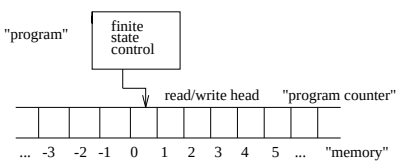
The $10,000 question is whether a polynomial time simulation exists, or in other words whether $P = NP?$. Do there exist languages which can be verified in polynomial time and still take exponential time on deterministic machines?

This is the most important question in computer science. Since proving an exponential time lower bound for a problem in NP would make us famous, we assume that we cannot do it.

What we can do is prove that it is at least as hard as any problem in NP. A problem in NP for which a polynomial time algorithm would imply all languages in NP are in P is called NP-complete.

Turing Machines and Cook's Theorem

Cook's Theorem proves that satisfiability is NP-complete by reducing all non-deterministic Turing machines to SAT.
Each Turing machine has access to a two-way infinite tape (read/write) and a finite state control, which serves as the program.



A program for a non-deterministic TM is:

1. Space on the tape for guessing a solution and certificate to permit verification.

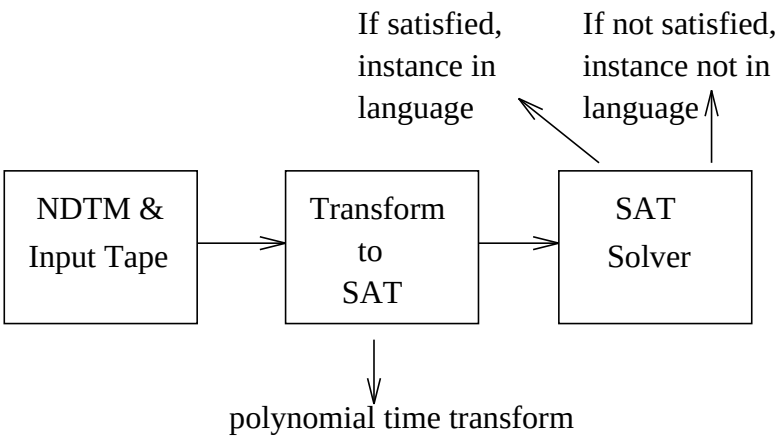
2. A finite set of tape symbols
3. A finite set of states Θ for the machine, including the start state q_0 and final states Z_{yes}, Z_{no}
4. A transition function, which takes the current machine state, and current tape symbol and returns the new state, symbol, and head position.

We know a problem is in NP if we have a NDTM program to solve it in worst-case time $p[n]$, where p is a polynomial and n is the size of the input.

Cook's Theorem - Satisfiability is NP-complete!

Proof: We must show that any problem in NP is at least as hard as SAT. Any problem in NP has a non-deterministic TM program which solves it in polynomial time, specifically $P(n)$.

We will take this program and create from it an instance of satisfiability such that it is satisfiable if and only if the input string was in the language.



If a polynomial time transform exists, then SAT must be NP-complete, since a polynomial solution to SAT gives a polynomial time algorithm to anything in NP.

Our transformation will use boolean variables to maintain the state of the TM:

Variable	Range	Intended meaning
$Q[i,j]$	$0 \leq i \leq p(n)$ $0 \leq k \leq r$	At time i, M is in state q_k
$H[i,j]$	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n)+1$	At time i, the read-write head is scanning tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n)+1$	At time i, the contents of tape square j is symbol S_k
	$0 \leq k \leq v$	

Note that there are $rp(n) + 2p^2(n) + 2p^2(n)v$ literals, a polynomial number if $p(n)$ is polynomial.

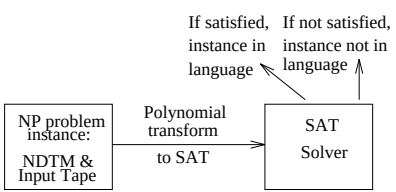
We will now have to add clauses to ensure that these variables takes or the values as in the TM computation.

The group 6 clauses enforce the transition function of the machine. If the read-write head is not on tape square j at time i, it doesn't change ....

There are $O(p(^2(n))$ literals and $O(p^2(n))$ clauses in all, so the transformation is done in polynomial time!

Polynomial Time Reductions

A decision problem is NP-hard if the time complexity on a deterministic machine is within a polynomial factor of the complexity of any problem in NP.
A problem is NP-complete if it is NP-hard and in NP.
Cook's theorem proved SATISFIABILITY was NP-hard by using a polynomial time reduction translating each problem in NP into an instance of SAT:

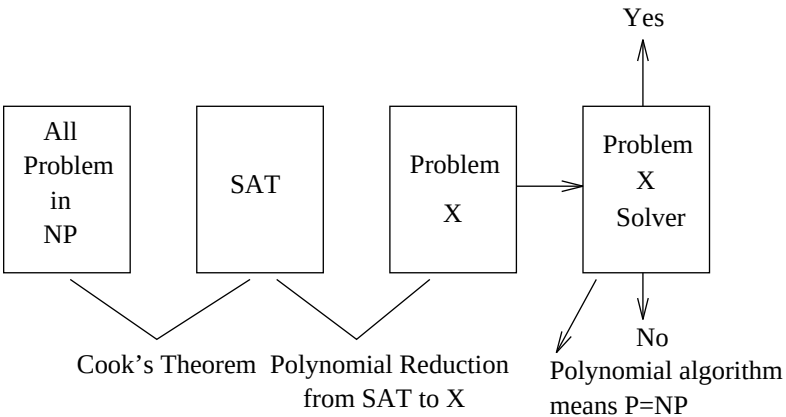


Since a polynomial time algorithm for SAT would imply a

polynomial time algorithm for everything in NP, SAT is NP-hard. Since we can guess a solution to SAT, it is in NP and thus NP-complete.

The proof of Cook's Theorem, while quite clever, was certainly difficult and complicated. We had to show that all problems in NP could be reduced to SAT to make sure we didn't miss a hard one.

But now that we have a known NP-complete problem in SAT. For any other problem, we can prove it NP-hard by polynomially transforming SAT to it!



Since the composition of two polynomial time reductions can be done in polynomial time, all we need show is that SAT, ie. any instance of SAT can be translated to an instance of x in polynomial time.