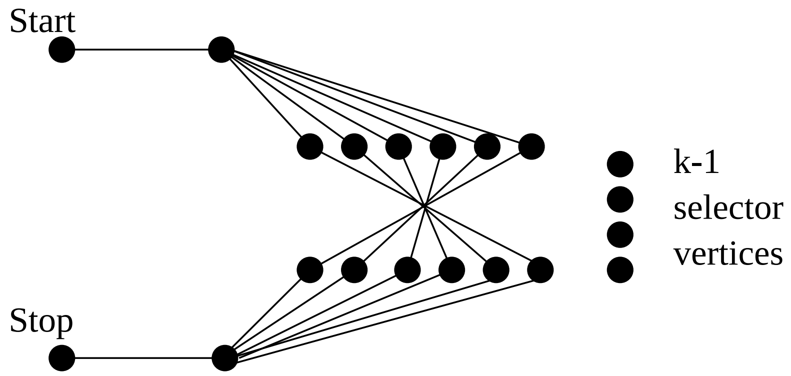
36.5-5 Prove that Hamiltonian Path is NP-complete.

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.

Approximating Vertex Cover

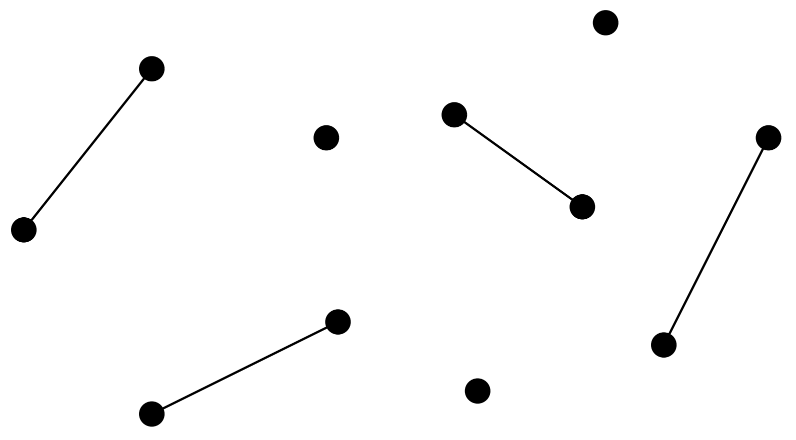
As we have seen, finding the minimum vertex cover is NP-complete. However, a very simple strategy (heuristic) can get us a cover at most twice that of the optimal.

While the graph has edges
    pick an arbitrary edge v, u
    add both u and v to the cover
    delete all edges incident on either u and v

If the graph is represented by an adjacency list this can be implemented in $O(m+n)$ time.

This heuristic must always produce cover, since an edge is only deleted when it is adjacent to a cover vertex.

Further, any cover uses at least half as many vertices as the greedy cover.



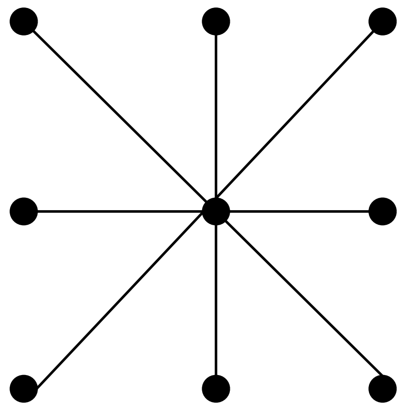
Why? Delete all edges from the graph except the edges we selected.

No two of these edges share a vertex. Therefore, any cover of just these edges must include one vertex per edge, or half the greedy cover!

Things to Notice

- Although the heuristic is simple, it is not stupid. Many other seemingly smarter ones can give a far worse performance in the worst case.

  Example: Pick one of the two vertices instead of both (after all, the middle edge is already covered) The optimal cover is one vertex, the greedy heuristic is two vertices, while the new/bad heuristic can be as bad as $n-1$.



- Proving a lower bound on the optimal solution is the key to getting an approximation result.

- Making a heuristic more complicated does not necessarily make it better. It just makes it more difficult to analyze.

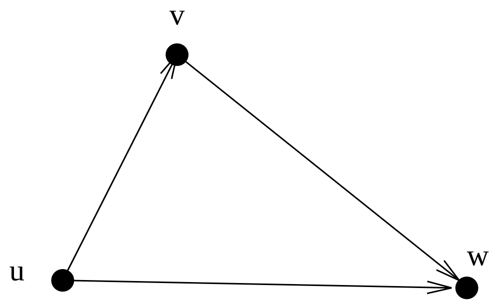
- A post-processing clean-up step (delete any unecessessary vertex) can only improve things in practice, but might not help the bound.

The Euclidean Traveling Salesman

In the traditional version of TSP - a salesman wants to plan a drive to visit all his customers exactly once and get back home.

Euclidean geometry satisfies the triangle inequality, $d(u, w) \leq d(u, v) + d(v, w)$.

TSP remains hard even when the distances are Euclidean distances in the plane.

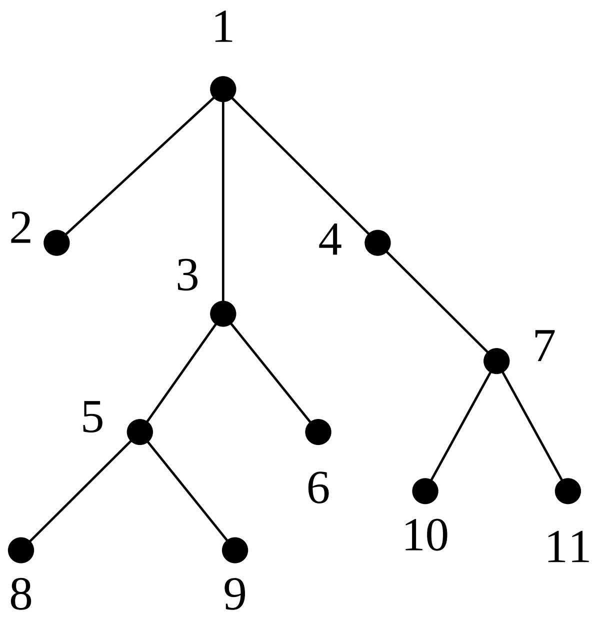


Note that the cost of airfares is an example of a distance function which violates the triangle inequality.

However, we can approximate the optimal Euclidean TSP tour using minimum spanning trees.

Claim: the cost of a MST is a lower bound on the cost of a TSP tour.

Why? Deleting any edge from a TSP tour leaves a path, which is a tree of weight at least that of the MST! If we were allowed to visit cities more than once, doing a depth-first traversal of a MST, and then walking out the tour specified is at most twice the cost of MST. Why? We will be using each edge exactly twice.



Every edge is used exactly twice in the DFS tour: *1*.

However, how can we avoid revisiting cities?

We can take a shortest path to the next unvisited vertex. The improved tour is $1-2-3-5-8-9-6-4-7-10-11-1$. Because we replaced a chain of edges by the edge, the triangle inequality ensures the tour only gets shorter. Thus this is still within twice optimal!

37.1-3 Give an efficient greedy algorithm that finds an optimal vertex cover of a tree in linear time.

In a vertex cover we need to have at least one vertex for each edge.

Every tree has at least two leaves, meaning that there is always an edge which is adjacent to a leaf. Which vertex can we never go wrong picking? The non-leaf, since it is the only one which can also cover other edges!

After trimming off the covered edges, we have a smaller tree. We can repeat the process until the tree as 0 or 1 edges. When the tree consists only of an isolated edge, pick either vertex.

All leaves can be identified and trimmed in $O(n)$ time during a DFS.

Formal Languages and the Theory of NP-completeness

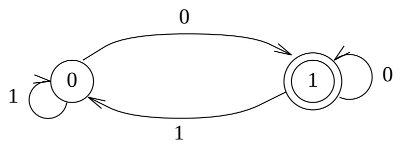
The theory of NP-completeness is based on formal languages and Turing machines, and so we will must work on a more abstract level than usual.

For a given alphabet of symbols Σ =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:

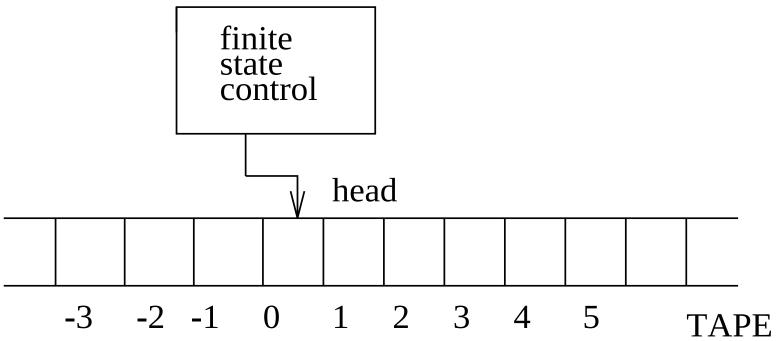


No memory is required, except for the current state.

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)



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!

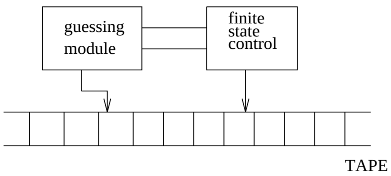
The goal of all this is going to be a formal way to talk about the set of problems which can be solved in polynomial time, and the set that cannot be.

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 Travelling 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?

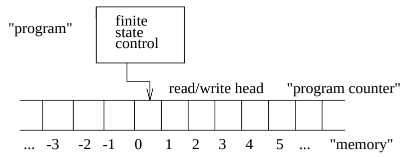
This is the most important question in computer science. Since proving an exponential time lower bound for a problem in NP would make us famous, we assume that we cannot do it.

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}

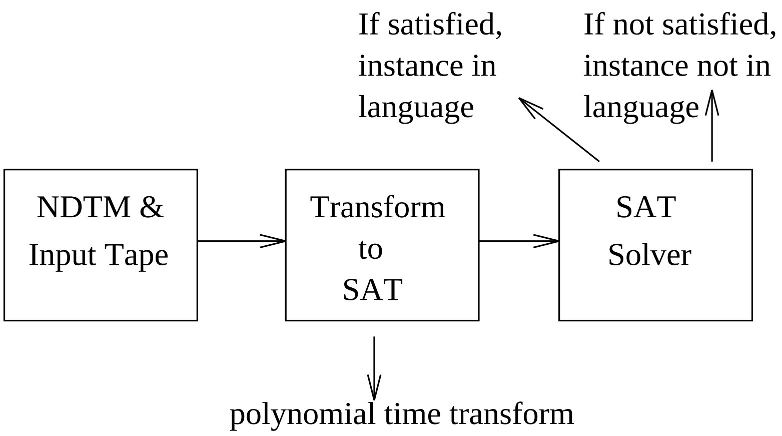
4. A transition function, which takes the current machine state, and current tape symbol and returns the new state, symbol, and head position.

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.

INCLUDE CLAUSE FIGURE DESCRIPTION

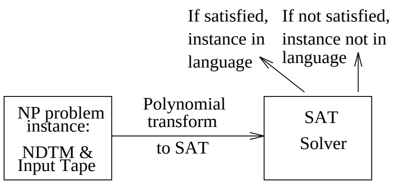
The group 6 clauses enforce the transition function of the machine. If the read-write head is not on tape square j at time i, it doesn't change ....

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.

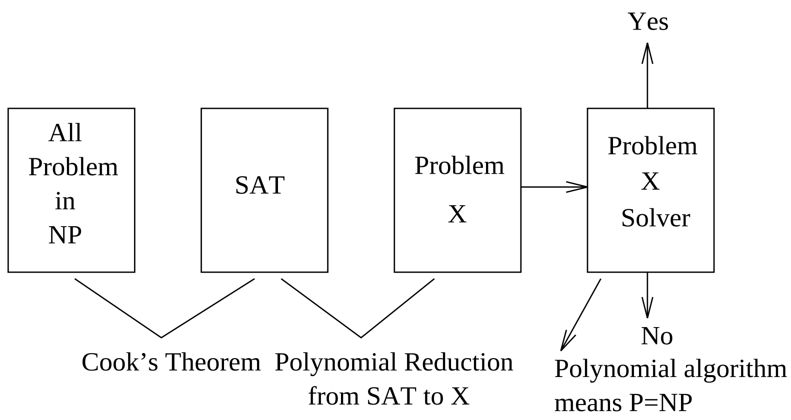
A problem is NP-complete if it is NP-hard and in NP. Cook's theorem proved SATISFIABILITY was NP-hard by using a polynomial time reduction translating each problem in NP into an instance of SAT:



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.

Finding the Optimal Spouse

1. There are up to n possible candidates we will see over our lifetime, one at a time.

2. We seek to maximize our probability of getting the single best possible spouse.

3. Our assessment of each candidate is relative to what we have seen before.

4. We must decided either to marry or reject each candidate as we see them. There is no going back once we reject someone.

5. Each candidate is ranked from 1 to n, and all permutations are equally likely.

For example, if the input permutation is

$$(4, 2, 3, 5, 6, 1)$$

we see $(3, 1, 2)$ after three candidates.

Picking the first or last candidate gives us a probability of $1/n$ of getting the best.

Since we seek maximize our chances of getting the best, it never pays to pick someone who is not the best we have seen.

The optimal strategy is clearly to sample some fraction of the candidates, then pick the first one who is better than the best we have seen.

But what is the fraction?

For a given fraction $1/f$, what is the probability of finding the best?

Suppose $i+1$ is the highest ranked person in the first n/f candidates. We win whenever the best candidate occurs before any number from 2 to i in the last $n(1-1/f)/f$ candidates.

There is a $1/i$ probability of that, so,

$$P = \sum_{i=1}^{\infty} \frac{(\frac{1}{f})(1-\frac{1}{f})^i}{i}$$

In fact, the optimal is obtained by sampling the first n/e candidates.