P/NP Introduction
Chapter 36

Note: Book describes the P/NP problems in terms of Languages; we will skip this method of analysis.

Some problems have algorithms that run in a reasonable amount of time (polynomial time $O(n^c)$ where c is a fixed constant) but some problems are not known to have fast algorithms - maybe they do and maybe they don't. We are going to look at some of these types of problems. So far we have looked at problems that are $O(n^3)$. Although n^3 problems may not seem fast, they are much faster than exponential size problems.

It is important to know when you are trying to write an algorithm to solve a problem that may not have a known fast solution. You then want to look for a different way to formulate the problem.

There are problems that run in time $O(2^{n^c})$ and for which it is provable that this is the BEST algorithm that exists! So the lower bound is exponential! These analysis only apply to serial computation; most models of computation are the same with respect to time (serial random access machine, turing machine, etc.)

- Polynomial time is considered *reasonable* or *tractable*. Algorithms that have a $O(n^c)$ running time.
- Exponential time is considered unreasonable or not tractable. Algorithms with a $\Omega(k^{n^c})$ running time, where k > 1.

Magnitude of problems we can solve:

n	n^2	2^n	
10	100	2^{10}	1024, not too big, 0.001 to compute
170	29000	2^{170}	number of atoms in the planet, centuries to compute
223	49000	2^{223}	number of atoms in galaxy, millennia to compute
1000	1000000	2^{1000}	?, need to measure in astronomical time

Solutions that require exponential time are essentially useless unless n is very small. These are the hard problems!

There are hundreds of interesting, useful problems that are known to require exponential time, in areas such as boolean logic, graphs, arithmetic, network design, number theory, games and puzzles, linguistics, language theory, and more.

Types of problems we will examine:

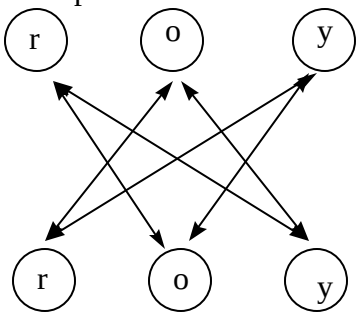
1. Optimization: Find values/configuration to optimize against other constraints. The "answer" is the optimal set of values.
2. Decision: A simple YES/NO answer is given, usually to indicate whether or not some threshold value can be achieved in the problem. We can make a decision problem out of every optimization problem that is almost as hard (still only exponential time solutions known).

Examples of hard problems:

Graph Coloring : A coloring of a graph G=(V,E) is a mapping C:V $\rightarrow$ S where S is a finite set of "colors" such that if (u,v) is an edge in E, then C(v)<>C(u). In other words, adjacent vertices are not assigned the same color. X(G) is the chromatic number of G, or the smallest number of colors needed to color G.

1. Optimization Problem: Given G, determine X(G) and produce an optimal coloring (i.e. one that uses only X(G) colors).
2. Decision Problem: Given G and a positive integer k, is there a coloring of G using at most k colors? If so, G is said to be k-colorable).

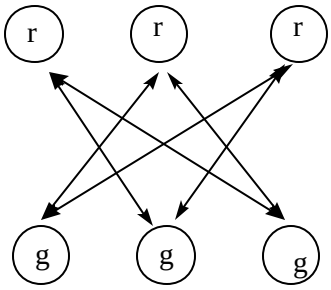
Example:



Basic technique for graph coloring:

```
for i← 1 to n do
        color ← 1
        while there is a vertex adjacent to v[I] that is colored color
                do color=color+1
        color v[I] with color
```

But this is not guaranteed to give optimal. If visit nodes in order top, bottom, top, bottom, … then get a coloring like above. If visit all the top ones first, get below which uses only 2 colors:



The algorithm we just gave is an approximation algorithm – not guaranteed to be optimal, but a solution that may be close to optimal. Finding the optimal is still only known if we use exponential time!

Graph coloring is an abstraction of certain types of scheduling problems.

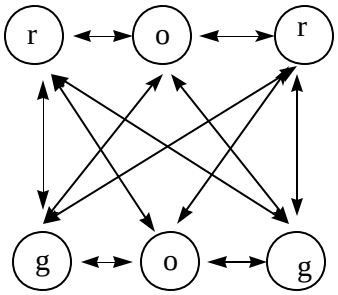
One example:

Final exams at a university, with four exams each day to be held over two days for a total of 8 time slots. Many classes will be having finals in parallel at the same time out of these time slots. Consider that we have an unlimited number of rooms. The exams for some courses must be at different times since many students may be in both classes.

Let V be the set of courses, and let E be the pairs of courses whose exams must not be at the same time; i.e. connect together the courses where a student is enrolled in both courses.

Then the exams can be scheduled in the 8 time slots without conflicts iff the graph G=(V,E) can be colored with 8 colors. Those classes with the same colors can have finals at the same time. The minimum number of colors gives us the minimum number of time slots required to prevent conflicts.

Example with 6 classes:



Can color these with 3 colors. If we had only two time slots then a student is going to miss a final. The red classes can be run at the same time in one time slot, the orange at the same time in another, but if the green is run at any of those two slots there will be a conflict.

Graph coloring a solution to many of these Constraint Satisfaction Problems.

Bin Packing:

Suppose we have an unlimited number of bins each of capacity 1, and n objects with sizes s1, s2, … s_N, where each s_i is between 0 and 1.

Optimization Problem: Determine the smallest number of bins into which the objects can be packed (and find an optimal packing).

Decision Problem: Given, in addition to the inputs described, an integer k, do the objects fit in k bins?

Applications of bin packing include packing data in computer memories (e.g., files on disk tracks, program segments into memory pages, and fields of a few bits each into memory words) and filling orders for a product (e.g. fabric or lumber) to be cut from large, standard-size pieces.

Knapsack Problem:

You are a thief and have broken into a bank. The bank has n objects of size/weight s1, s2, s3, … s_N (such as gold, silver, platinum, etc. bars) and "profits" p1, p2, p3, … p_N where p1 is the profit for object s1. You have with you a knapsack that can carry only a limited size/weight of capacity C.

Optimization Problem: Find the largest total profit of any subset of the objects that fits in the knapsack (and find a subset that achieves the maximum profit).

Decision Problem: Given k, is there a subset of the objects that fits in the knapsack and has a total profit at least k (or equal to k)?

Many different problems fit the knapsack problem, especially in economy or optimizing the use of resources with a limited capacity.

Subset Sum:

This is a simpler version of the knapsack problem. The input is a positive integer C and n objects whose sizes are positive integers s1, s2, …. s_N.

Optimization Problem: Among subsets of the objects with a sum at most C, what is the largest subset sum?

Decision Problem: Is there a subset of the objects whose sizes add up to exactly C? e.g. electoral college problem

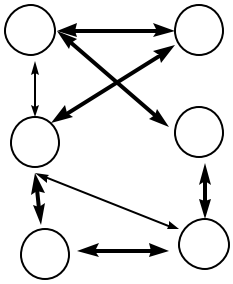
Ex: Given 1,3,5,2,8,4 and C=10

Possible subsets <= 10 are: 1,3,5 3,5,2 2,8 4,5 1,4,5 etc.

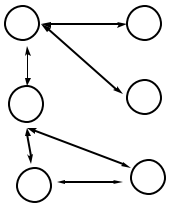
Hamilton Path/Hamilton Circuit

A Hamilton path (Hamilton circuit or cycle) in a graph is a path (cycle) that passes through every vertex exactly once.

Example of hamilton path:



No Hamilton path exists for this graph since we must revisit the same node up top twice:



Decision Problem: Does a given graph have a Hamilton path?

Traveling Salesman Problem

This is harder than the Hamilton circuit problem. Instead of just finding a circuit, we also want to find the circuit that has the lowest cost in a weighted graph. Imagine if we are a traveling salesman, and the vertices are cities to visit while the edges are roads connecting the cities. Weights on the edges indicate time or distance to travel between cities. The salesman wants to visit all cities in the least amount of time/distance. Other applications include routing in networks or trucks for package delivery/pickup.

Optimization Problem: Given a weighted graph, find a minimum weighted Hamilton circuit.

Decision Problem: Given a weighted graph and an integer k, is there a Hamilton circuit with total weight at most k?

P/NP Continued

The problems we have investigated so far are polynomial bound time $O(n^c)$ or tractable. We can formally define these problems to be in a class named "P".

Definition: *P* is the class of decision problems that are polynomial bounded.

Why define a polynomial time bound as a criteria for an entire class? After all, there are many problems in P and can be quite large. However:

1. Non polynomial time problems are intractable. Most researchers believe no polynomial time algorithms exist for many of these well defined non-polynomial algorithms.
2. Polynomials have nice mathematical closure properties
3. P becomes independent of a particular model of computation

To show that an algorithm is in P, we need to be able to come up with a solution to P that runs in polynomial time.

Examples of algorithms in P: sorting, DFS, BFS

Definition: *NP* is the class of decision problems for which a given proposed solution for a given input can be verified in polynomial time to see if it really is a solution. It does not say anything about the time required to find the solution but only to verify if it is correct, hence NP stands for Non-deterministic Polynomial bounded algorithms.

To show that an algorithm is in NP, we need to be able to show that a proposed solution can be verified in polynomial time.

Examples of algorithms in NP: sorting, DFS, BFS, traveling salesman, knapsack problem

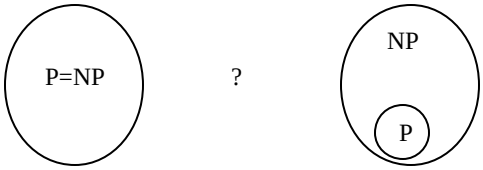
In the knapsack problem, given k, is there a subset of the objects that fits in the knapsack and has a total profit at least k? Given some solution, we can just add the weights together and see if it fits in the knapsack. Similarly, we can add the profits together and see if the value is at least k. So verification of a solution is easy, and takes O(n) time.

Same for traveling salesman problem: Can add up weights on the graph and see if the total weight is at most k, and also ensure that every city is visited only once by just traversing the path and remembering which cities were visited.

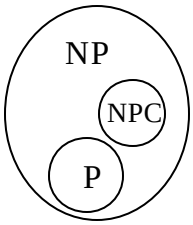
Same for sorting problem: Not only can we find a solution in polynomial time, but verifying that a sequence is sorted takes only linear time to scan through each element and ensure the next is bigger than the previous.

This implies that $P \subseteq NP$.

But does P=NP? We don't think so. No polynomial bound algorithms are known for many problems in NP (including knapsack, traveling salesman, subset sun). But we are not sure since nobody has been able to prove that NP problems have a non-polynomial lower bound.



We can actually get a slighter clearer picture by identifying another class of NP problems, NP-Complete problems:



NP Complete problems: The hardest decision problems of the class. If we can prove that if there were a polynomial bounded algorithm for an NP complete problem, then there would be a polynomial bounded algorithm for each problem in NP.

Before we define, need to *define polynomial reducibility*:

An algorithm A1 is polynomially reducible to A2 if there exists a polynomial time transformation from A1 to A2. We denote this as $A1 \propto A2$. In other words, if we have a problem A1, then in polynomial time we can make a mapping so that the algorithm A2 will solve problem A1; i.e. A2 is "at least as hard" as A1.

Definition: An algorithm A1 is NP-complete (NPC) if:

1. It is in NP and

2. For every other algorithm A2 $\in$ NP, A2 $\propto$ A1. In other words, every other NP problem can be solved with algorithm A1 by first doing a polynomial time mapping.

The process:

Given algorithm A1
Given every problem in NP, call it {A2}, we can do a polynomial time transformation
to make its parameters "fit" our algorithm A1. Call this time P1.
We can run A1 to get a solution
Polynomial time transformation of A1's answer mapping into {A2}, so we now have answers
for {A2}. Call this time P2.

Total runtime for other problems in NP is P1 + T(A1) + P2.
If A1 runs in polynomial time, we can solve all other problems in NP in polynomial time!

If we have a single algorithm A1 known to be NP-Complete, then:

1. For all other Algorithms A2 in NP, A2 $\propto$ A1.
2. This implies that to show a new algorithm A?, if we want to show it is NPC:

→ We have to show that A? is in NP (solution can be verified in P time)
→ We have to show that for some other NPC algorithm A1, A1 $\propto$ A?.
By transitivity, then all other problems in NP are $\propto$ A?
Because {All NP} $\propto$ A1 $\propto$ A?

It is important to show that A1 $\propto$ A? and not A? $\propto$ A1. If our known NPC algorithm A1 can be polynomially transformed into A?, then A? must be at least as hard as A1. However, if we show that A? can polynomially be transformed into A1, this doesn't tell us anything about how hard A? might be. We can solve an easy problem with a hard solution, but this doesn't say anything about all the other problems in NP. But since all the other NP problems can be solved with A1, if A1 could be polynomially transformed into A?, then all other NP problems can be solved in the time it takes to solve A? plus polynomial time.

This leads to the theorem that if any NP-Complete problem is in P, then P=NP.

The big question:

We can show other algorithms to be NP-Complete by showing an existing NPC problem can be polynomially reduced to the new algorithm. But how do we prove the first NPC problem?

Answer: We will not prove, but the first problem proven to be NP-Complete is the circuit satisfiability problem. This is known as Cook's Theorem. Based on Cook's theorem, other theorists were able to prove hundreds of other problems to be NP-complete.

Other NP-Complete Problems: Graph Coloring, Hamilton path, bin packing, subset sum, knapsack, traveling salesman.

Example:

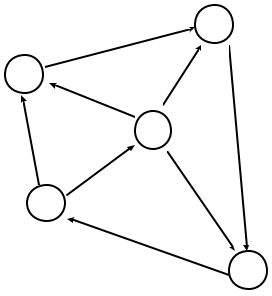
Assume we know that the directed Hamilton circuit problem is NP-Complete (it is). Show that the undirected Hamilton circuit problem is also NP-Complete.

1. Show that the undirected problem is in NP by verifying solution in polynomial time.

   Answer: Given a proposed solution, we can start at any vertex and follow the path, marking each vertex as we go. When we reach the original vertex without having visited any marked vertices, and after having visited every vertex, we are done and can output a YES. O(V) time.

2. Show that the directed problem is polynomial reducible to the undirected problem; i.e. we can turn the directed problem into an undirected graph and use that to solve the directed problem.

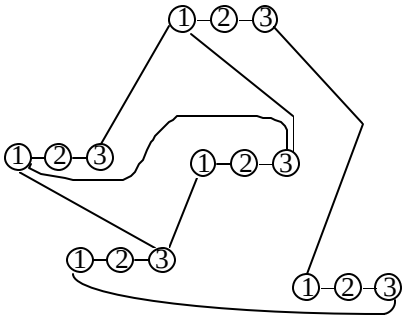
   Consider below:



   How can we turn this into an undirected graph and not lose information? Could just make links bidirectional, but then we can get circuits we couldn't get in the original. Need to preserve the direction.

   Solution: Expand each node into three nodes, where the first node is an input node, the middle a transition node, and the third an output node. The middle node ensures a path

within each node from 1-2-3 or 3-2-1 in sequence, otherwise we could potentially visit "half" a node at a time.



Note that all nodes must be visited in sequence 1-2-3 or 3-2-1, since 3 and 1 are always connected, and 2 is always in the middle. Thus any hamilton circuit discovered on the undirected graph translates back into the directed graph. We can do the transformation both ways in O(V+E) time.

Another Example:

Show that the traveling salesman problem is NPC.

TSP: Given Graph G=(V,E) with edge costs, and target C, find a tour of all the vertices that visits each only once and has total edge cost <=C.

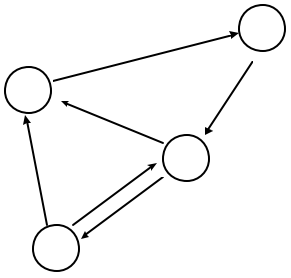
1. TSP can easily be verified to be in NP by traversing path and adding up cost to see if it is <=C.
2. We will show that the Hamiltonian Cycle problem $\propto$ TSP.

Construct an instance of TSP:

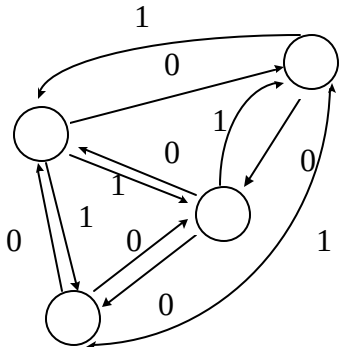
Form the complete graph G'=(V,E') where E'=

$$c(i,j)=\begin{cases}0 \ldots if\ (i,j)\in E\\ 1 \ldots if\ (i,j)\notin E\end{cases}$$

Consider:



We want to find out if there is a hamilton cycle using TSP. The instance of TSP constructed using the above algorithm is:



We can make the transformation in polynomial time. If we run TSP on this problem, then if there is a Hamilton cycle in the original graph G, the TSP must return a path of total cost 0 (not using any other links we added and we formed a cycle) on graph G'.

Last Example:

Show that the knapsack problem is in NPC given that the Subset Sum problem is NPC.

The subset sum problem is given an array A (A1, A2 .. A_n) and integer k. Is there a subset of A that equals k?

The knapsack problem is given an Array A (A1, A2.. A_n), each element with weights W_i and values V_i, is there a subset of A that has total weight <C and total value =k?

1. Given a solution to knapsack, we can easily verify if it is of value = k and weight <=C in polynomial time.
2. Show that SS $\propto$ Knapsack.

Set all of the weights equal to zero, so our knapsack is now unlimited in size. Let k remain the same. The answer returned by the knapsack problem will be the answer to the subset sum problem since the knapsack problem is essentially the subset sum problem with the extra weight condition (now removed).