CS351
Intractable Problems

Intractable problems refer to problems that we cannot solve in a reasonable time on a computer as the size of the problem grows. These are considered to be any problem that grows faster than polynomial time with respect to the input.

Recall that:

- Problems solvable in polynomial time on a computer also take polynomial time on a Turing Machine
- The dividing line between solvable and unsolvable problems is typically exponential vs. polynomial

The Class of Languages P

If a (deterministic) TM M has some polynomial p(n) such that M never makes more than p(n) moves when presented with input of length n, then M is said to be a polynomial-time TM.

P is the set of languages that are accepted by polynomial-time TM's. Equivalently, P is the set of problems that can be solved by a real computer by a polynomial-time algorithm.

- Why? If we can solve the problem in polynomial time on a computer we can build a polynomial time TM that also solves the problem. The solution can be used to verify if a string is in the language or not, all in polynomial time (although perhaps a larger degree of polynomial on the TM than on the computer)..
- Many familiar problems are in P: sorting algorithms, graph reachability, matrix multiplication, etc.

The Class of Languages NP

A nondeterministic TM that never makes more than p(n) moves in any sequence of choices for some polynomial p) is said to be a polynomial-time NTM.

NP is the set of languages that are accepted by polynomial-time NTM's. Equivalently, you can think of NP as the set of problems where a proposed solution can be verified as correct by a real computer using a polynomial-time algorithm.

Why is this? To see if a particular input string belongs to a language described by a NTM, our NTM can fork off multiple NTM’s for all the computations that verify if the string is accepted or not. Say that one of these branches leads to an accepting state. In parallel, we are also examining all the other branches that are not accepting. This is equivalent to verifying the solution is correct (the branch that leads to the accepting state). Other solutions, for example, ones that are not correct, are being examined in parallel along the other branches.

Many problems are in NP but appear not to be in P. A simple example is a jigsaw puzzle. This can be very difficult to solve (presumably not in polynomial time) but it is usually very easy to verify if a proposed solution is correct just by looking at it. Here are some problems in NP but appear to not be in P:

- TSP (is there a tour (where we visit all nodes once) of a graph with total edge weight $\leq$ k?)
- SAT (does this Boolean expression have a satisfying assignment of its variables (i.e. makes the Boolean expression true?)
- CLIQUE (does this graph have a set of k nodes with edges between every pair?).

There are many more…

One of the great mathematical questions of our age: Is there anything in NP that is not in P? Obviously, $P \subseteq NP$ since a deterministic Turing machine is also a nondeterministic Turing machine. Additionally, if we can solve a problem in polynomial time, we can use this algorithm to verify if a proposed solution is correct in polynomial time.

NP Complete Problems

If we can't resolve the P = NP question, we can at least demonstrate that certain problems in NP are "hardest" in the sense that if any one of them were in P, then P = NP.

- Called NP-complete problems
- Intellectual leverage: each NP-complete problem's apparent difficulty re-enforces the belief that they are all hard.

Before we continue, let's revisit **polynomial reducibility**:

A problem P1 is polynomial reducible to P2 if there exists a polynomial time transformation from P1 to P2. We denote this as $P1 \propto P2$. In other words, if we have a problem P1, then in polynomial time we can make a mapping so that a solution to P2 will solve problem P1; i.e. P2 is "at least as hard" as P1.

Definition: A problem P1 is NP-complete (NPC) if:

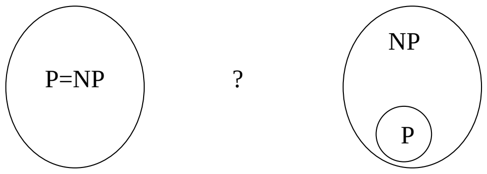
1. It is in NP and
2. For every other problem $P2 \in NP$, $P2 \propto P1$. In other words, every other NP problem can be solved by P1 by doing a polynomial time mapping so that P2 fits the parameters of P1. We may also need to do a polynomial time mapping from the solution of P1 to give us the solution to P2.

   The total time to solve P2 is then: time(P1) + polynomial-mapping-to-P1 +
   Polynomial-mapping-to-p2
   = time(P1) + polynomial-time

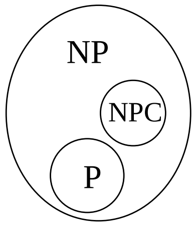
   The total runtime for other problems in NP is then time(P1) + polynomial-time.

If we could show that time(P1) is polynomial, then it means every problem in NP could be solved in polynomial time!

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



or:



The Clay Mathematics Institute has offered a million dollar prize to anyone that can prove that P=NP or that P≠NP. So hopefully you have been paying attention….

Some NP Complete Problems

The SAT, TSP, and CLIQUE problems we saw earlier are all NP Complete.
Here are a few others:

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 $C(v) \neq C(u)$. 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).

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.

1. Optimization Problem: Determine the smallest number of bins into which the objects can be packed (and find an optimal packing).
2. 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.

1. 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).
2. 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.

1. Optimization Problem: Among subsets of the objects with a sum at most C, what is the largest subset sum?
2. Decision Problem: Is there a subset of the objects whose sizes add up to exactly C? e.g. electoral college problem

3SAT: This is a special case of the SAT problem where all formulas are in Conjunctive Normal Form with exactly three literals. CNF is the logical AND of a group of OR terms. For example, the following clause is in CNF where the x's are Boolean variables:

$$(x1 \vee x2 \vee x3) \wedge (x4 \vee x5 \vee \neg x6) \wedge (x4 \vee \neg x9 \vee \neg x3)$$

1. Optimization Problem: What is an assignment to the variables to satisfy the entire clause (i.e. make it true?) This means each individual clause must contain at least one literal that is assigned true. This is harder than it looks. For example, assigning TRUE to x1, x2, and x3 could make the first clause true, but then with x3 true, this could make the last clause FALSE since we have ¬x3.
2. Decision Problem: Does an assignment to the variables exist that satisfies the clause?

Minesweeper Game: The Minesweeper consistency problem is to determine if various states of Minesweeper are valid or not. This problem has shown to be NP-Complete.

There are many other NP Complete problems.

Proving a problem is NP Complete

If we have a single problem P-NPC known to be NP-Complete, then:

1. For all other problems P2 in NP, P2 $\propto$ P-NPC.
2. This implies that to show a new problem P-NEW is NPC:

    → We have to show that P-NEW is in NP (solution can be verified in P time)
    → We have to show that for some other NPC problem such as P-NPC,
    P-NPC $\propto$ P-NEW

    By transitivity, then all other problems in NP are $\propto$ P-NEW
    Because {All NP} $\propto$ P-NPC $\propto$ P-NEW

It is important to show that P-NPC $\propto$ P-NEW and not P-NEW $\propto$ P-NPC. If our known NPC problem P1 can be polynomially transformed into P-NEW, then P-NEW must be at least as hard as P-NPC. However, if we show that P-NEW can polynomially be transformed into P-NPC, this doesn't tell us anything about how hard P-NEW might be. It just says we can use something that is "hard" to solve something that might be really easy.

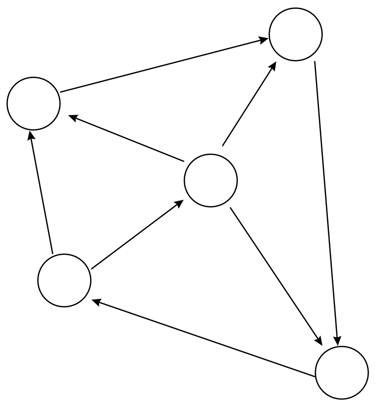
Example: A Hamilton circuit is a path in a graph that visits each node exactly once. Assume we know that the directed Hamilton circuit problem is NP-Complete (it is). Show that the undirected Hamilton circuit problem is also NP-Complete.

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.

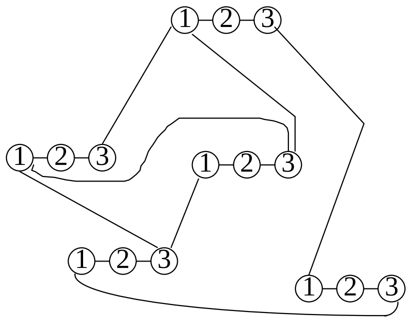
2. Show that the directed problem is polynomial reducible to the undirected problem; i.e. we can turn the directed problem into an undirected graph and use that to solve the directed problem.

    Consider below:



How can we turn this into an undirected graph and not lose information? We could just make the links bidirectional, but then we can get circuits we couldn't get in the original. We 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, where E is the edges and V are the vertices.

Example: Show that the Traveling Salesman Problem (is there a Hamilton Circuit with total edge weight cost ≤ k?) is NPC. Assume that we know the Hamilton Circuit problem is NPC.

This is easy to show, because the Hamilton Circuit problem is a special case of the TSP.

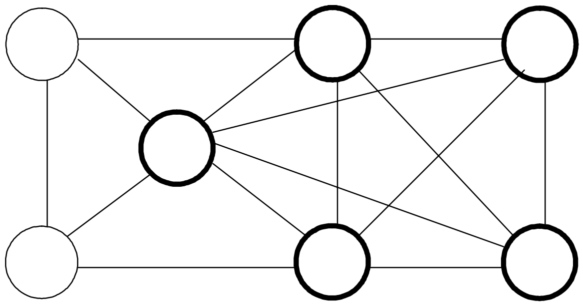
1. First, show that TSP is in NP. This is easy for the decision version of the problem. Given a proposed solution (a tour and the constant k) we simply add up the cost on all the edges, make sure this is a valid tour, and that the total cost is ≤ k. If so, the solution is correct.

2. Show that Hamilton Circuit is reducible to TSP. To do this, we simply construct a special version of the TSP. We make a weight of 1 for every edge in the graph and set k equal to any number ≥ the total number of nodes. Any answer found by the TSP solution must also therefore be a valid Hamilton Circuit.

It is very difficult to prove that the general Hamilton Circuit problem is NP Complete.

Example: Show that the Clique problem is NP Complete. In the Clique problem, you are given an undirected graph. A clique is a subgraph of the larger graph, where every two nodes of the subgraph are connected by an edge.

A k-clique is a clique that contains k nodes. The following is an example of a graph having a 5-clique:



Assume that we know that 3SAT is a NP Complete problem.

1. Show that Clique is in NP. Given a proposed solution consisting of n nodes, systematically loop through each node, and see if it is connected to all of the other n nodes. This requires $O(n^2)$ runtime.

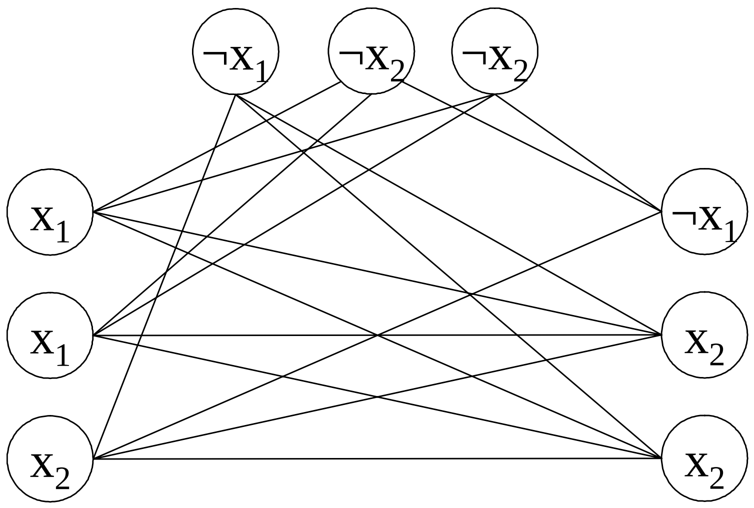
2. Show that 3SAT is polynomial reducible to Clique. To do this, we create a special graph that is designed to mimic the behavior of the variables and clauses in the 3SAT problem.

   Let Φ be a formula with k clauses such as:

   $$\Phi = (a_1 \lor b_1 \lor c_1) \land (a_2 \lor b_2 \lor c_2) \land \ \dots \ (a_k \lor b_k \lor c_k)$$

   The reduction creates the undirected graph G as follows. The nodes in G are organized into k groups of three nodes each called the triples $t_1, t_2, \dots t_k$. Each triple represents one of the clauses in Φ. Edges are present between all pairs of nodes in G, except for nodes in the same triple, and nodes of opposite labels, e.g. x_1 and $\neg x_1$. For example, given:

   $\Phi = (x_1 \lor x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2 \lor \neg x_2) \land \ (\neg x_1 \lor x_2 \lor x_2)$ we construct:



Now, if we can solve the clique problem on this graph, then Φ is satisfied if and only if G has a k-clique.

Suppose a k-clique exists on G. Since there are no edges within a triple, this clique must consist of a single node from each triple. Now, assign a value of "true" to each node in the k-clique. This translates into assigning "true" to the corresponding Boolean value in the 3SAT problem. Since we have no edges connecting contradictory values (e.g., NOT) then we have found a set of variables that makes each individual clause of 3SAT true and therefore the entire clause must be true.

What you should be asking yourself:

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: 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. We will look at a version of the CSAT problem in the next lecture.