Problem Spaces P/NP

P/NP

- Chapter 7 of the book
- We’ll skip around a little bit and pull in some simpler, alternate “proofs”

- Intractable Problems
  - Refer to problems we cannot solve in a reasonable time on the Turing Machine/Computer
    - Dividing line is exponential vs. polynomial, even a big polynomial; e.g. $O(n^{10000})$

Class of Languages P

- If a deterministic Turing Machine 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 accepted by polynomial time Turing Machines

P

- Equivalently, P is the set of problems that can be solved by a real computer with a polynomial time algorithm
  – If we can solve in polynomial time we can build a polynomial time deterministic TM that also solves the problem. The solution can be used to verify if a string presented as input is in the language or not
  – Most familiar problems are in this class P
    - Sorting, graph reachability, matrix multiplication, etc.

Class of Languages NP

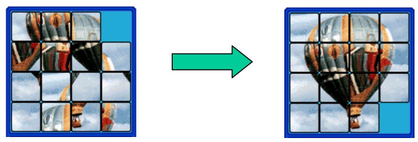
- If a non-deterministic Turing Machine N has some polynomial p(n) such that N never makes more than p(n) moves in any sequence of choices when presented with input of length n, then N is said to be a polynomial time non-deterministic TM
  - These p(n) moves are for one "thread"; N forks off threads in parallel
- NP is the set of languages accepted by polynomial time non-deterministic Turing Machines

NP

- Equivalently, NP is the set of problems where a proposed solution can be verified as correct by a real computer using a polynomial time algorithm
  - To see if a string belongs in a language for NP, our NTM can fork off multiple NTM's for all computations. In parallel we check if one of these paths leads to an accepting state.
  - By verifying if a proposed solution is correct (in polynomial time) we are checking one of these paths to see if it is a solution

NP $\supseteq$ P

- NP is obviously a superset of P
- But many problems appear to be in NP but not in P
  - E.g., consider a "sliding tile" puzzle



Solve in polynomial time? (e.g. function of # of tiles)
But given a proposed solution, easy to verify if it is correct in polynomial time

Other problems

- Here are some well-known problems in NP but appear to not be in P
  - TSP – Traveling Salesman Problem
    - Is there a tour (visit each node once) of a graph with total edge weight $\leq$ k?
  - SAT - Boolean Satisfiability Problem
    - Does a boolean expression have a satisfying assignment of its variables (i.e. make the boolean expression true)?
  - CLIQUE
    - Does a graph have a set of k nodes with edges between every pair?
- Many more…

- 29 Node Traveling Salesperson Problem
- 29! = 8.8 trillion billion billion possible asymmetric routes.
- ASCI White, an IBM supercomputer being used by Lawrence Livermore National Labs to model nuclear explosions, is capable of 12 trillion operations per second (TeraFLOPS) peak throughput
- Assuming symmetric routes, ASCI White would take 11.7 billion years to exhaustively search the solution space

The Big Question

- Is there anything in NP that is not in P?
- We know that P $\subseteq$ NP since a deterministic TM is also a nondeterministic TM
- But it is unknown if P = NP

- The Clay Mathematics Institute has offered a million dollar prize to anyone that can prove that P=NP or that P≠NP

NP Complete Problems

- Most people believe that P ≠ NP due to the existence of problems in NP that are in the class NPC, or NP Complete
- NP Complete
  - The "hardest" problems of the class NP
  - Before we continue to define NPC problems, let's revisit the notion of **reducibility**

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 if we perform the polynomial time mapping
- This means P2 is "at least as hard" as P1
  - E.g. if P1 was impossible to solve, so is P2
  - E.g. if we can prove P1 requires exponential time to solve, then so does P2
- Example:
  - Find-Max-In-Array $\propto$ Sort-Array
    - Mapping
      - Max-Finding to Sorting: do nothing
      - Sorting solution back to Max-Finding: pluck last element of array

Definition of NPC

- 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 via 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

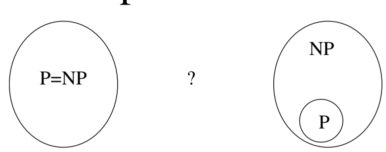
  time(P1) + polynomial-mapping-to-P1 + polynomial-mapping-to-P2

  = time(P1) + polynomial-time

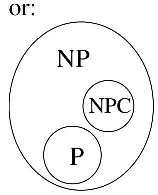
- The total runtime for other problems in NP is then time(P1) + polynomial-time.
  - If we can find such a problem P1 we can solve in polynomial time, then every problem in NP can be solved in polynomial time! i.e. P=NP

P=NP?

- Theorem: If ay NP-Complete problem is in P, then P=NP
- Possible pictures:



or:



Some NP Complete Problems

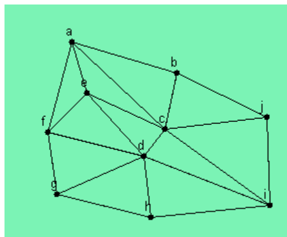
- SAT, TSP, and CLIQUE are all NPC
- Here are some others:
  - Graph Coloring
  - Bin Packing
  - Knapsack
  - Subset Sum
  - 3SAT
  - Minesweeper Constraints
  - Many More

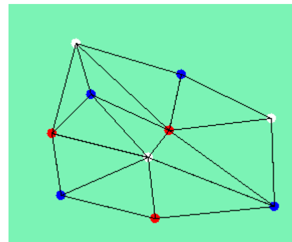
Graph / Map Coloring

- Given a graph with edges and nodes, assign a color to each node so that no neighboring node has the same color. Generally we want the minimum number of colors necessary to color the map (the chromatic number).
- Map coloring example: Nodes could be states on a map, color each state so no neighboring state has the same color and therefore becomes distinguishable from its neighbor

- Sample Graph

Can you determine the minimum number of colors for this graph?





Only known solution guaranteed to be optimal requires exponential time – examining all possible assignments, typically using backtracking after assigning colors

Graph Coloring

- With these problems we can propose them as **decision problems** or as **optimization problems**
- The decision problem is slightly easier, but still is NP Complete
- Decision Problem: Given G and a positive integer k, is there a coloring of G using at most k colors?
- Optimization Problem: Given G, determine X(G), the chromatic number, and produce an optimal coloring that uses only X(G) colors.

Bin Packing

- Suppose we have an unlimited number of bins each of capacity 1, and n objects of sizes s1, s2 … s_n, where each s_i is a number between 0 and 1.
- Optimization: Determine the smallest number of bins into which the objects can be packed (and find the optimal packing)
- Decision: Given, in addition to the inputs described, an integer k, do the objects fit in k bins?

- Lots of applications; packing data in computer files/memory, filling orders from a product, loading trucks

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)?

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

3SAT

- 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:

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

- 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.
- Decision Problem: Does an assignment to the variables exist that satisfies the clause?

Problem 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

Proving NP-C

- Remember to show that P-NPC $\propto$ P-NEW and not vice versa! The reverse tells us that we could only use a NPC problem to solve a potentially easy problem
  - E.g. PairSum Problem:
    - Out of n numbers, is there a pair that equals C?
    - PairSum $\propto$ Subset-Sum
      - Run subset sum to find subsets equal to C, out of those subsets scan through them to find one with only two numbers
    - This is just using a hard problem to solve an easy one; doesn't say anything about PairSum being NP-Complete

Example: NP Complete

- 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.

- Strategy: $\text{Hamilton}_{\text{directed}} \propto \text{Hamilton=}_{\text{undirected}}$

Hamilton Circuit

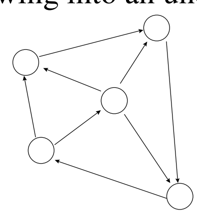
1. Show that the undirected problem is in NP by verifying solution in polynomial time.

   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.

Hamilton Circuit

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.

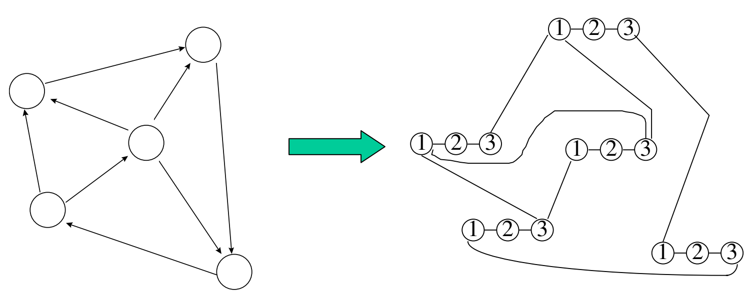
   e.g. turn the following into an undirected graph we could find a HC on:



   could just make links bi-directional, but that would allow circuits that were invalid in the original graph

Hamilton Circuit

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.



Hamilton Circuit

- 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: TSP

- Show that the decision version of the Traveling Salesman Problem (is there a Hamilton Circuit with total edge weight cost $\leq k$?) is NPC. Assume that we know the Hamilton Circuit problem is NPC.

- Strategy: Hamilton $\propto$ TSP

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 $\leq$ 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 $\geq$ the total number of nodes. Any answer found by the TSP solution must also therefore be a valid Hamilton Circuit.

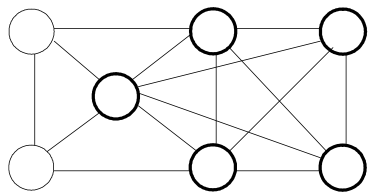
- It is very difficult to prove that the general Hamilton Circuit problem is NP Complete.

Example : Clique

- 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.
- Decision problem: Does a k-clique exist on a graph G?

Clique

- 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.
- Strategy: 3SAT $\propto$ Clique

Clique is NPC

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.

Clique is NPC

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:

$$\Phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ \ldots \ \ (a_k \vee b_k \vee c_k)$$

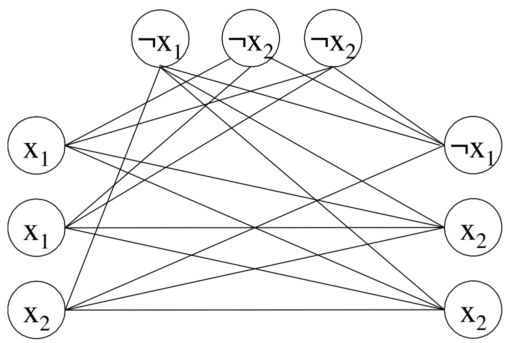
3SAT to Clique

- 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, … 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:

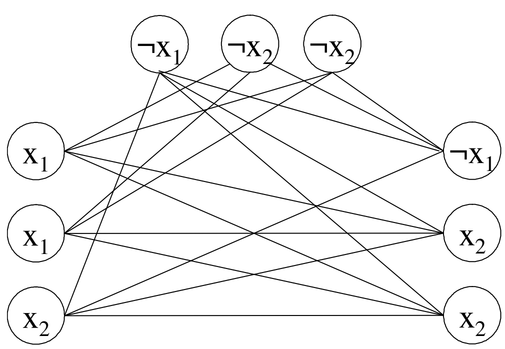
3SAT to Clique

$\Phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Construct:



3SAT to Clique



Suppose a K-Clique exists on G

Since there are no edges **within** a triple, the clique must consist of a single node from **each** triple

Each node in the clique can be a “true” in 3SAT to satisfy the 3SAT expression; no edges connecting opposites

You should be wondering…

- 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.

Cook's Theorem, SAT

- The satisfiability problem was proved by Stephen Cook in the early 70's to be the first NP Complete problem.
- Here we will give the basic argument for a related problem, that of the circuit satisfiability problem.
  - Quite a few more details in Cook's Theorem
  - Recall that to show something is NP Complete means that the problem is in NP, and that all other problems in NP can be reduced to the NP Complete problem in polynomial time.

Certificate

- First, we need the concept of a **certificate**. A certificate is just a proposed solution to a problem. A certificate is used by a verification algorithm, V, where V is used to verify if the certificate is valid or not on input x (i.e. it is used to test if a problem is in NP).
- Algorithm V verifies an input string x if there exists a certificate y such that V(x,y) = true. The language verified by V can be stated as:

$$L = \{ x \mid V(x,y) = 1 \text{ for some string } y\}$$

Certificate

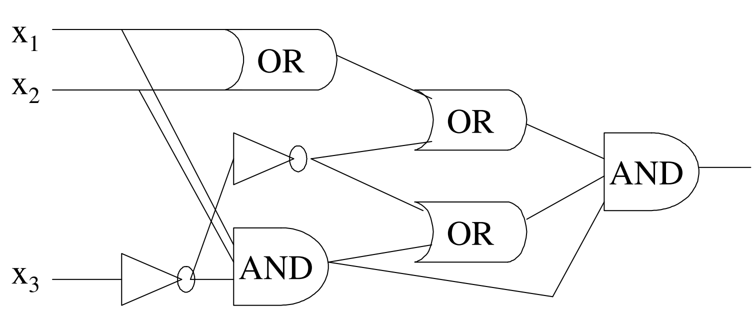
- $L = \{ x \mid V(x,y) = 1 \text{ for some string } y\}$
- In these terms, x is the problem statement (e.g. for Hamilton Cycle, it is the graph and all the edges). Y is the certificate, or the proposed solution (e.g. for Hamilton Cycle, a list of all vertices in the cycle).
- We measure the time of a verifier only in terms of the length of x, so a polynomial time verifier runs in polynomial time in the length of x. If a language has a polynomial time verifier, then by definition it is in NP. For polynomial time verifiers, the length of the certificate must be polynomial bounded to the length of x.

Circuit Satisfiability

- The CSAT problem is very similar to the Satisfiability problem, but uses actual digital circuits instead of Boolean expressions.
- The CSAT problem is:
  - Given a Boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?
- i.e. is there a set of inputs that makes the output 1?

Example

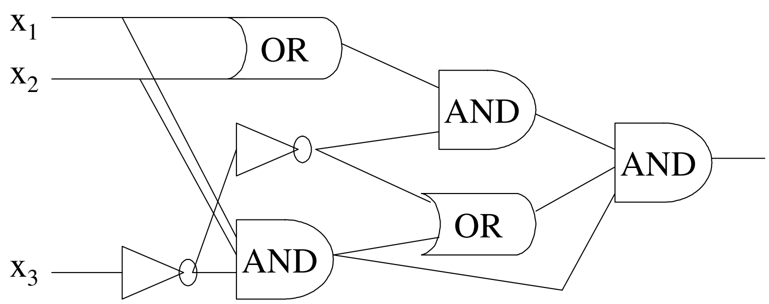
- Set of inputs to make output 1?



In the above circuit, can you find values for X1, X2, and X3 that makes the output 1? Apparently we may need to simply try all the (exponential) possibilities until we find a satisfying assignment. In this case the assignment 1,1,0 makes the output 1.

Not Satisfiable

- Change second or to and:



CSAT

- Our claim is that CSAT is NP-Complete. To make this claim, we must first show that CSAT belongs to the class NP.

- Easy to show: Recall the verification algorithm, V(x,y) where x is the problem specification and y is a certificate.
- Our verification algorithm is to construct the actual circuit out of the input specifications, x. Then, y is a proposed solution. Put the values of y into our constructed circuit and then simulate it. If the output is 1, then the circuit is satisfiable, otherwise 0 is output. This entire operation can be completed in time polynomial to the input, x, thus CSAT is in NP.

CSAT

- Second, we must show that every language in NP is polynomial-time reducible to CSAT. Our proof is based on the workings of an actual computer (and thus is translatable to the workings of a Turing Machine). This is much harder to prove, and here we only give a sketch of the formal proof.

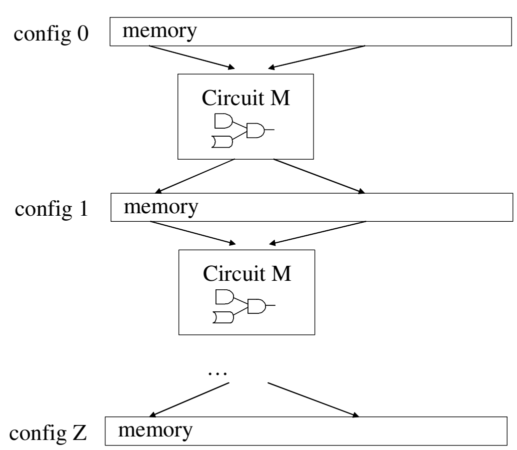
Computer In Operation

- Consider a computer program.
  - The program operates as a sequence of instructions stored in memory. D
  - Data is stored in memory.
  - We have a program counter and other registers to operate on data and store the current instruction.
  - Let's just consider all registers plus RAM to be "memory".
- At any point in the execution of a program, the entire state of the computation is represented in the computer's memory. A particular state of computer memory is a **configuration**. The execution of an instruction may be viewed as mapping one configuration to another. In terms of traversing states, a program is traversing a sequence of configurations.

Configurations

- This mapping from one configuration to another can be accomplished by a combinational circuit (in fact, this is what is done by the computer). Call this combinatorial circuit, M. The execution of a program taking Z+1 steps can then be viewed as the following:

Configuration Mapping



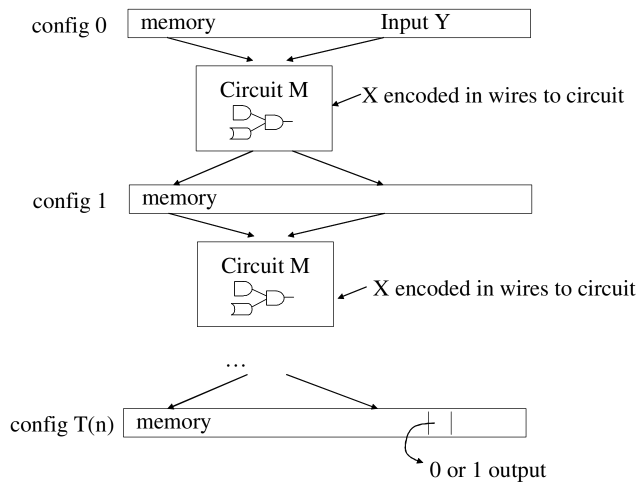
CSAT is NPC

- Let L be any language in NP. By definition, L has a verification algorithm, V(x,y), that runs in polynomial time.
- This means that if the input x is of length n, then there is a constant k such that the runtime of V, T(V), is $O(n^k)$. Similarly, the length of the certificate, y, must also be $O(n^k)$.

Configurations and CSAT

- Since V consists of a polynomial number of steps, then in polynomial time we can construct a single combinational circuit that computes all configurations produced by an initial configuration.
  - The input of x can be hardcoded to circuit M in terms of certain wires (e.g. for Hamilton Circuit, we could have wires encoding the nodes and edges in the graph). This is because x never changes for each step of the configurations.
  - This leaves y as input values to the circuit. In this case, we don't know what y is – we want to find values of y that satisfy the circuit (and therefore solve the original NP problem).

Configurations and CSAT



The last configuration is T(n), the runtime of V. We use some location in memory to get the ultimate output for the satisfiability of the circuit.

We're Done?

- If we take a step back, what have we just created? It is simply a big version of a CSAT problem!
  - The input to the problem is configuration 0, and the output is configuration T(n).
  - In between is a big combinatorial circuit.
- We now have reduced the arbitrary problem in NP to the CSAT problem.
  - In this instance of CSAT, the input is certificate Y and we don't know what the certificate Y is (i.e. we don't know a proposed solution).
  - If we solve this CSAT problem, then we will have determined if an input Y exists or not. If this input Y exists, then it is a solution to the original problem in NP. Thus, if CSAT is satisfiable, then Y solves the NP problem.

Wrapup of CSAT

- In the other direction, suppose that some certificate exists. When we feed this certificate into the circuit, it will produce an output of 1. Thus, if the original problem is solvable, this instance of CSAT is also satisfiable.
- To complete the proof, we must show that the circuit can be constructed in polynomial time – i.e. the reduction is polynomial.
  - Since the verification algorithm runs in polynomial time, there are only a polynomial number of configurations.
  - This means we are hooking together some polynomial number of circuits M. M can be constructed in size polynomial to the length of a configuration making the overall construction time polynomial.
- Based on the two properties of CSAT, we conclude that every language in NP reduces to CSAT in polynomial time and since CSAT is in NP, it is NP Complete.