NP Completeness and Reductions

Connor Lemp and Nikola Kapamadzin

January 22, 2015

Reductions[1]

A problem, L, is said to be polynomial-time reducable to some problem M, if any algorithm that can solve M, can be turned into an algorithm to solve L with at most a polynomial increase in the runtime.[2] This is a very useful concept, because as we will see, many complicated problems can be reduced into simpler ones whose behavior we already know.

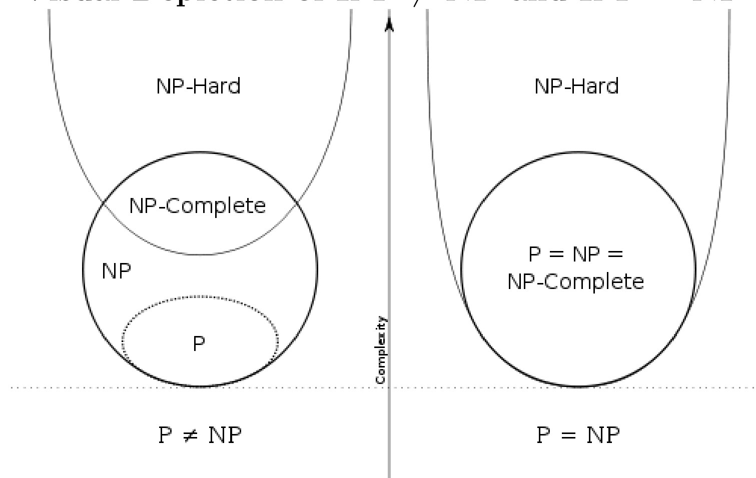
NP Hard

A decision problem, a problem with a yes or no answer, H, is NP-hard when for any problem Q in NP, there is a polynomial-time reduction from Q to H. Problems that are NP Hard can be thought of as being harder than the hardest NP problems.

NP Complete

A problem is NP Complete when it is both in NP and NP Hard. This by definition means that an NP Complete problem can be both checked in polynomial time and any problem in NP can be reduced to it. Most proofs that a problem is NP Complete simply reduce the problem to a simpler problem that we already know and have shown is NP Complete.

Visual Depiction of If P $\neq$ NP and If P = NP[3]



[1] Written by Nikola

[2] Bartlett, Padraic "P and NP"

[3] Esfahbod, Behnam

Fundamental Problem

Let's look at the most fundamental of NP complete problems, SAT. The SAT, or boolean satisfiability problem determines if there exists a set of boolean values to satisfy some boolean formula. The SAT was the first problem proved to be NP Complete, using the Cook-Levin Theorem. The Cook-Levin Theorem uses a strange method of computation, the nondeterministic turing machine, and the proof goes out of our scope. However we will use a different model of computation, circuits, that will allow us to determine that SAT is NP Complete.

Circuits

When discussing algorithms and runtimes, it's important to know the method of computation, because the way the algorithm is structured relies on how you compute basic information. We use a simple, yet incredibly useful, method of computation, the circuit, because not only does it simplify our proofs, but circuits are the method of computation used in our computers today!
Definition: Circuits perform logical operations to boolean inputs through the use of the logic gates *and*, *or*, and *not*, and then output a boolean value.
We can quickly see that this model of computation is the same as SAT, because circuits evalutate boolean expressions, and the SAT problem tells us whether or not a boolean expression is satisfiable, so using circuits simply determines whether or not a a boolean expression is satisfiable.

SAT is NP Complete

We know that by definition a NP Complete problem is both in NP and NP Hard, so let's show that SAT/Circuits are in both NP and NP Hard. This is not an official proof to why SAT is NP Complete, but this is meant to give an idea of why it is NP Complete. We know that NP problems can be checked in polynomial time, so we need to show that if we know all of our input values for the circuit, we can see if it's satisfiable with our boolean expression in polynomial time. When we have all our input values, we have an expression that looks something like the following (this is an example)

$$((T \lor F) \land (T \land F)) \lor (F \lor T)$$

Our cicuit computation method can determine the truth value of our statement quickly, let's call this runtime n. Now if we add one more element for our computer to check anywhere, for example

$$((T \lor T \lor F) \land (T \land F)) \lor (F \lor T)$$

We see that this would take our computer only one more unit of time, as it has to pass one more input through the logic gates to recieve its final output, therefore the

runtime of this is $n+1$, which is polynomial.
Now to show SAT is NP Hard, or that any problem in NP can be reduced to SAT. We know that all problems in NP are decision problems, which means they consist of yes or no answers. We also know that any problem in NP is checkable in polynomial time. If a problem can be checked in polynomial time with given inputs, it can be encoded as a boolean expression because any conditions the problem may have can be stored as either true or false values through the inputs, and the logic operations tell how they interact. This is of the SAT/circuit form which means that SAT/circuits are a generalized form of any NP problem.

Graph 3-coloring Problem[4]

Suppose you have a graph G, and 3 colors to choose from. You want to color each vertex of G with a color, but you don't want any two vertices connected by an edge to be the same color. The question of determining whether or not this is possible for arbitrary graphs is known as "graph 3-coloring".

Graph 3-coloring is in NP

Suppose you have a graph G on n verticies, and an alleged configuration of colors, C, that will properly 3 color G. Can you verify this configuration in polynomial time? Let's consider a fairly inefficient method of determining whether or not C is valid. For every pair of verticies $v, w \in G$, run this process:

1. if v and w are connected by an edge and they share the same color, then exit this process and return that C is not a satisfyign configuration on g.

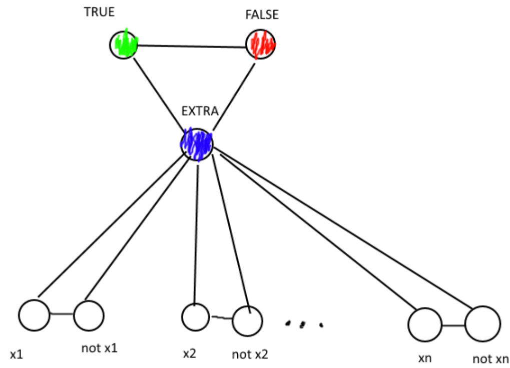
2. go back and pick the next pair of verticies.

We can see that this algorithm, roughly speaking, has a runtime of n^2, as we check each vertex in G against every vertex in G. This is polynomial! Thus, 3 coloring a graph is in NP as a purported solution can be checked in polynomial time.

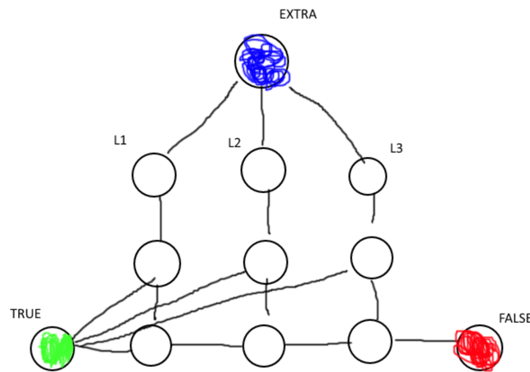
3 SAT reduces to Graph 3 Coloring

The general idea behind this is to take any 3SAT formula, S and use it to construct a graph G. This graph will be 3 colorable iff the 3SAT formula it was derived from is satisfiable.
The first step in constructing G is to "declare" the variables with a portion of a graph. For a graph that is being constructed to solve a 3SAT problem with n variables, the "declaration" portion will look something like this.

[4] Written by Connor

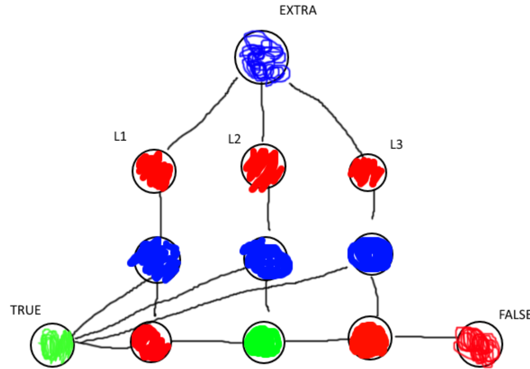


In this portion of the graph, we create verticies for all variables in S, along with vertices labeled TRUE, FALSE and EXTRA, each of which have been colored in arbitrarily. All the variables in s have been connected to EXTRA to ensure that the variables can only have colors corresponding to TRUE or FALSE. Further, variables and their negations have been connected to ensure that they are not assigned the same TRUE/FALSE value. Now that we've declared our variables in S, we need a way of representing clauses and making sure that at least one literal (a variable or its negation) is true in each clause $(L_1 \vee L_2 \vee L_3)$. For this, we will use "gadgets". The gadgets look something like this:

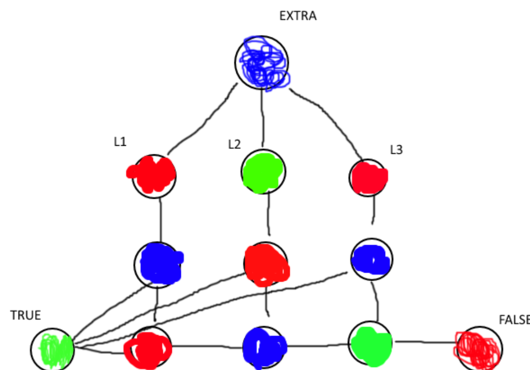


We make a gadget for each clause in S. Each gadget reuses 6 verticies from the declaration portion of the graph (EXTRA, TRUE, FALSE, and the 3 variables/negations

corresponding to (L_1, L_2, L_3)), along with all 13 of the edges that aren't the ones between EXTRA and the literals. This gadget is only 3 colorable if atleast one of (L_1, L_2, L_3) is true. To see this, consider the case where all of (L_1, L_2, L_3) are red (FALSE).



As you can see, when all the literals are red, it forces the vertices in the next row down to be all blue. This means that none of the bottom row can be blue. From here you have to color atleast 2 of the middle 3 vertices on the bottom row either the same color. These same-colored vertices cannot be adjacent, and if they are on opposite side, then one of them will be adjacent to a similarly colored TRUE or FALSE as is shown in the above picture. So when all 3 literals are false, this portion of the graph cannot be 3 colored.



You can see from the above picture, that when atleast one of the literals is green (TRUE), then we can utilize all 3 color on the bottom row, thus making it possible to 3 color.
Because each individual gadget is 3 colorable iff the clause it represents is satisfiable, the entire graph can only be 3 colorable if every gadget within it can be 3 colored, just like how S can only be satisfied when all the clauses evaluate to true. Further, because all clauses that contain a variable x, are dealing with the same vertex the entire graph can only be 3 colorable if all clauses/gadgets can be satisfied/colored simultaneously. Thus, if the graph is colorable, then the 3SAT formula is satisfiable, and if the 3SAT formula is satisfiable, then the graph is colorable. Therefore we can say that 3 SAT reduces to Graph 3 Coloring!

Sources

Source of the 3-coloring the graph problem and solution:
http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/23Reductions.pdf
Source for defining SAT and 3SAT:
http://en.wikipedia.org/wiki/Boolean_satisfiability_problem
Source of picture useful for differentiating P, NP, NP Hard, and NP Complete:
http://en.wikipedia.org/wiki/NP-complete
Source for defining NP Completeness, NP Hard, and useful examples on additional SAT/3SAT terms and examples:
http://math.ucsb.edu/~padraic/mathcamp_2014/np_and_ls/mc2014_np_and_ls_lecture1.pdf