Aaron Lee

CS 4960

Dr. Melanie Martin

P and NP problems

With the advent of computers, it is now possible to do numerous calculations that were once limited to doing by hand, in mere seconds. As technology progressed, hardware was able to perform more rapidly making these calculations appear even faster. Upgrading the hardware, however, is only one of the ways to obtain results quicker. The other method, which may be difficult, is to rewrite the software so that they can be completed in fewer computations. Unfortunately, there are some problems that we are still unable to solve in a reasonable amount of time. Those problems are part of a set known as NP problems.

Before jumping into the world of NP problems, its best to gain some background knowledge about algorithms. When we check the efficiency of an algorithm, we count the amount of calculations that the algorithm is likely to compute in its lifetime before reaching an answer. If a program requires a large amount of calculations, that program is considered slow. The average amount of calculations required to solve the problem is written in Big O notation. It is written as $O(f(x))$, where $f(x)$ is the complexity of the algorithm. The lower the complexity of the algorithm, the higher the likelihood that will run quicker. Of course, each algorithm may have instances where it will outperform an algorithm which is usually faster than it, but in this case, we will only take the average run times into account. From fastest to slowest, the most common complexities are as follows: constant (k), logarithmic (log n), linear (n), polynomial (n^k), exponential (k^n). In these examples, k represents a constant number and n represents a number that will increase without bound.

A large number of the algorithms that we write are of polynomial complexity. Since they can run in polynomial time, they are in the set of P problems. If the amount of time exceeds a Big O notation of a polynomial time, mainly exponential time, then the algorithm is considered inefficient. One example of an algorithm with P complexity are most sorting algorithms. Certainly we are aware that there are different types of sorting algorithms available for us to use, and they all operate at different speeds. Two examples of sorting algorithms that operate at different speeds are bubble sort and heap sort. On average, bubble sort operates at $O(n^2)$ and heap sort operates at $O(n \log n)$. Clearly we see here that both algorithms will reach the same result but will operate at different speeds.

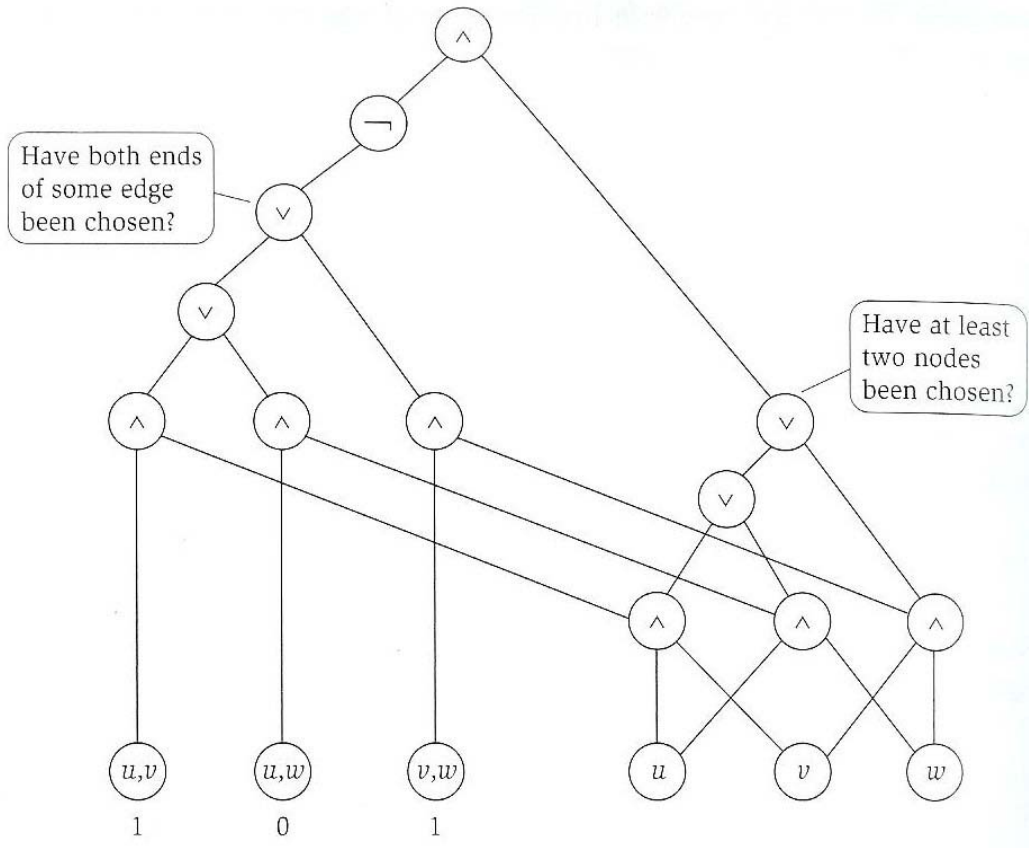
Sometimes, however, we do not have an efficient algorithm to solve problems what we are faced with. To describe these problems, a new set of algorithms was created called NP problems. In this case, NP stands for nondeterministic polynomial-time. Kleinberg and Tardos state that the condition for a problem to be in the set of NP problems is that that answer to that problem must be able to be verified in a polynomial amount of time. In this sense, we find that problems of P complexity are a subset of NP problems, in other words $P \subseteq NP$. But what does it mean when an answer can be verified in a polynomial amount of time? It means that another algorithm, named a certifier, can be used to determine if an answer to a problem is correct. This certifier would be programmed with the relevant knowledge of the problem we were trying to solve. From here, it would take in an input of a possible answer, and determine if this answer was correct. Going back to our sorting example, we could write a certifier that could check to see if the answer we gave it was correct in the context of the problem. Let's say that our sorting problem was to place this set {e, a, c, d, b} in alphabetical order. After developing a certifier, we could pass it an answer. If we passed it the set {a, b, c, d, e}, our certifier would do a quick check to see that this list was indeed in order. The check may look something like this: Is $a<b$? Is $b<c$? Is $c<d$? Is $d<e$? Since the answer for all of these questions was yes, the certifier would then respond that the answer passed in was a correct answer.

Note earlier that I stated that $P \subseteq NP$. It however has not been proven that $NP \subseteq P$ or even that $NP \not\subset P$. Although it has not been proven, since we do not have efficient algorithms for NP problems, we assume that $NP \not\subset P$. The example of a NP problem that Brasard and Bratley use outside of the set of P problems is finding a Hamiltonian cycle. A Hamiltonian cycle is a cycle that goes through an undirected graph and visits every vertex exactly once, then closes on itself. Before looking into the algorithm to come up with the answer, we can see that checking the answer can be done in a polynomial amount of time. The certifier would have to do two things: check to make sure the edges form a cycle and check to make sure all vertices in the graph were visited exactly once. The start/end vertex, however, would be visited twice since it is used to initialize and to end the cycle. There have been numerous attempts to solve this problem efficiently, but there has been no success in developing an algorithm that can solve this problem in a polynomial amount of time. Instead, the solutions that we have developed using a type of programming known as dynamic programming. With dynamic programming, we are able to cut down on the amount of time that a problem requires by also cutting out computations that are repetitive or unneeded. This drastically lowers the amount of computations required to solve the problem.

However, NP problems do not stop at the Hamiltonian cycle. A slightly more complicated problem of the Hamiltonian cycle is known as the traveling salesman problem. In this problem, the objective is to find the minimal Hamiltonian cycle in a graph. In reference to the problem, it would be the shortest path that the salesman could take that would take him to all of the cities he needed to visit and back home. But take into note that we currently do not know if NP problem can or cannot be solved in a polynomial amount of time. As the matter of fact, Clay Mathematics Institute has put out a reward for one million dollars to anyone who would be able to prove either $NP \subseteq P$ or $NP \not\subset P$. Even so, it is difficult to prove that one, let alone all, NP problems can be solved in a polynomial amount of time.

Thanks to problems known as NP-complete problems, that predicament was slightly simplified. Glaber, Pavan, Selman, and Sengupta state that NP-complete problems can be used to represent every other NP problem through a process known as reductions. Reductions are a method of changing one problem into another one where the transition only adds a polynomial amount of time. For example take the NP problem that has been proven to be NP-complete, circuit satisfiability (CIRCUIT-SAT). A CIRCUIT-SAT problem is a circuit composed of AND, OR, and NOT gates whose answer would be the condition in which that circuit is true. An example of a CIRCUIT-SAT problem is as follows: $(a_1 \vee a_2) \wedge (\neg a_1 \vee a_2 \vee \neg a_3) \wedge (\neg a_1 \vee \neg a_2 \vee a_3)$. In this case, an answer could be a_1 = false, a_2 = true, and a_3 = true. To show that CIRCUIT-SAT is NP-complete, we must show that it can be reduced into another NP problem. Ambos-Spies and Bentzien show that CIRCUIT-SAT can be reduced to any other problem from the NP set. First they state that a polynomial certifier does exist. It would only need to plug in the answer that it was given, and see if that answer satisfied the problem. The argument that CIRCUIT-SAT is NP-complete lies within the principle that it is made up of a circuit. Since a computer is a circuit, every algorithm on a computer can be broken down into the three basic gates that a CIRCUIT-SAT is composed of. Following that idea, one can argue that since every NP problem can be programmed, it must also be able to be represented by CIRCUIT-SAT. The individual variables within the CIRCUIT-SAT formula could represent the various bits that go into the problem.

A more concrete example would be constructing a CIRCUIT-SAT from a problem. In this case, Kleinberg and Tardos have come up with an example searching for a two-node independent set. Note that an independent set in a graph is a set of vertices in which no vertices within that set are adjacent. In their example, they use a three node graph consisting of the vertices u, v, and w. There are only two edges in this graph, one from u to v and another from u to w. Their formula is split up into two parts. First to check to make sure that both end of one edge has been chosen, they hard code in the edges and use the following formula: $((u \wedge v) \wedge 1) \vee ((u \wedge w) \wedge 0) \vee ((v \wedge w) \wedge 1)$. Notice that if the user chooses the vertex v and any other vertex, the answer for this will be true. That means that an illegal choice has been made. The second part of their formula checks to make sure at least two nodes have been selected and is as follows: (u ∧ v) ∨ (u ∧ w) ∨ (v ∧ w). Finally these two formulas are combined with an OR together to obtain the CIRCUIT-SAT equivalent of the independent set problem: ¬ ((u ∧ v) ∧ 1) ∨ ((u ∧ w) ∧ 0) ∨ ((v ∧ w) ∧ 1) ∧ ((u ∧ v) ∨ (u ∧ w) ∨ (v ∧ w)). A visual representation is shown below.


Have both ends of some edge been chosen?
Have at least two nodes been chosen?
∧
¬
∨
u,v
u,w
v,w
u
v
w
1
0
1

Algorithm Design p. 470

There are numerous NP problems in society that have not yet been proven to be in the P problem set. One of the biggest uses of the NP problem comes into the use with cryptography. This method is named RSA after the three individuals who first publicly described it and is the main form of the United States encryption and network security. RSA uses the idea that it is hard to find a matching distinctively prime number that is used for decryption. If it were ever the case that we determined that NP $\subseteq$ P, then that would mean that there was an algorithm that could crack RSA in a polynomial amount of time. The quest to solve this problem still continues. In fact, there has been a recent answer set forth by Vinay Deolalikar. Although his answer has not been verified yet, if he is correct, we will now know that NP $\not\subset$ P.

Citations

Kleinberg, J. & Tardos E. (2006) *Algorithm Design*. Boston, MA: Pearson Education, Inc.

Brassard, G. & Bratley, P. (1996) *Fundamentals of Algorithmics*. Upper Saddle River, NJ: Pearson Education.

Ambos-Spies, K. & Bentzien, L. (1997) Separating NP-Completeness Notions under Strong Hypotheses. *Proceedings of the 12th Annual IEEE Conference on Computational Complexity*. Pgs. 121-7. http://www.computer.org/portal/web/csdl/doi/10.1109/CCC.1997.612307

Glaber C., Pavan A., Selman A. L., Sengupta S. (2004) Properties of NP-Complete Sets. *Proceedings of the 19th Annual IEEE Conference on Computational Complexity*. http://www.computer.org/portal/web/csdl/proceedings/cc#5