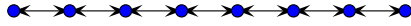


An Introduction to NP-Completeness



Introduction

We know that some problems seem to take a long time to solve while others can be solved rather quickly. Does this have to be so? Perhaps, if we were clever enough, we could come up with a way to solve a problem much more quickly than has been done previously. As you know, even an efficient sort like heapsort or mergesort takes a while on very large data sets. As these sorts are $\Theta(n \log n)$, maybe we can come up with an asymptotically faster sort, say an $\Theta(n \log^* n)$ or even an $\Theta(n)$ sort. Well, as long as we stick with sorts that rearrange values based upon comparisons between the items to be sorted, it turns out that we cannot. It has been shown that any sort that works in this fashion must take $\Theta(n \log n)$ time for arbitrary input and in the worst case. So if we try to find a better comparison sort, asymptotically speaking, we are just wasting our time. The “science” in Computer Science tells us not to bother.

What about some problems that seem to take an exponential amount of time to run? An example of this problem is deciding if one graph is a subgraph of another graph, which is known as the *subgraph isomorphism* problem (**SUB**). No one has come up with an algorithm that runs in less than $\theta(2^n)$ time, where n relates to the size of the graphs, when run on two arbitrary graphs. Like comparison sorts, has anybody proved that you can’t do better? The answer is no. So should we spend all our time trying to come up with a faster version of **SUB**? Probably not. In the first case, some very clever folks have worked on this very idea and haven’t come up with a better algorithm. Secondly, some very, very clever folks have shown that if you could come up with a quick solution for **SUB**, you could use that algorithm to quickly solve a vast number of problems for which no quick solution is known. So not only do you have the experience of the very clever folks who worked on **SUB** telling you not to bother trying to come up with a faster algorithm, you have the experience of a large number of other clever folks who failed finding faster algorithms for all those other problems whose solutions best solutions are as slow as **SUB**’s telling you the same thing as well. My experience is, when a large number of smart people are telling you the same thing, you might want to listen to them.

We need to tighten up our definitions of the words *quick* and *slow*. The words do not refer to how fast one can come up with a solution, but how fast the solution can be generated. Usually, quick means polynomial or better time algorithms for generating solutions and slow means exponential time or worse algorithms. A problem is *intractable* if it has been proven that all algorithms for generating solutions must be slow. For example, enumerating all permutations of a list is intractable, since it is easy to show it can’t be done in polynomial time. In fact, it must take at least $\Theta(n!)$ time. The term *efficient* is sometimes used for quick and the term *inefficient* is sometimes used for slow.

For the rest of this discussion, we will say a problem solution is efficient if it has an algorithm for solving the problem that runs in sub-exponential time. We will say a problem is intractable if it has been proven that no algorithm that runs in sub-exponential time is possible.

The classes P and NP

Let’s explore the idea that an efficient algorithm for **SUB** would necessarily mean an efficient solution for all problems in **NP**. We need to dive into a bit of theory to do this, however. Let’s devise a way to categorize problems in terms of efficient and intractable. Let’s say problems with efficient algorithms belong to class **P**. The letter **P** stands for *Polynomial Time*, meaning that these problems can be solved in $O(n^k)$ time, with k being a constant (n^k is a polynomial, hence the class name). Intractable problems, those problems whose solutions take at least exponential time, are in the class **I**. There is another important set of problems, those whose solutions can be solved on a *non-deterministic* computer in polynomial time. This class of problems is denoted **NP**, which stands for *Non-deterministic Polynomial time*. *Note: beginners have a tendency to say NP means non-polynomial, but that is incorrect.* A non-deterministic computer is far more powerful than a deterministic, or conventional, computer. Problems in **P**, those that can be solved in polynomial time on a conventional computer, surely can be solved in polynomial time on a non-deterministic computer. Therefore, the class **P** belongs to the class **NP**. The main question is: are there problems in **I** that are also in **NP**?

A non-deterministic computer, when given an alternative, always makes the right choice (wish I had one of those!). For example, a non-deterministic computer can solve the **SUB** problem in $O(n^k)$ time, where n relates to the sizes of the graphs involved, and k is a small constant that takes care of looking up vertices and edges in the graph. Assume the non-deterministic computer tries to see if the first graph is a subgraph of the second. Assume further that the first graph is indeed a subgraph of the second. Our computer would take a vertex in the first graph and align it with one of the vertices in the second. Which one? It would guess and it would guess correctly. Then it would attempt to align the second vertex in the first graph and

would guess correctly again. Eventually all n vertices would be aligned. This obviously would take $O(n^k)$ time for the n guesses our computer would have to make.

Alternatively, the class **NP** can be thought of as encompassing those problems for which a solution can be *verified* in polynomial time. It seems to be much easier to verify a solution than to come up with one, so this interpretation of the class **NP** still allows for **P** being a subset of **NP** and the possibility of some problems in **I** also belonging to **NP** as well. Of particular interest is that no known intractable problems can be solved on a non-deterministic computer in polynomial time or verified in polynomial time on a deterministic computer (think about verifying the output of a permutation program). Thus, at the moment, no intractable problems are known to be in **NP**.

Logically, two scenarios exist. Either all problems in **NP** will end up in **P** (someone discovers an efficient algorithm for solving each problem) or the problems in **NP** will be split between those in **P** and those in **I** (someone proves no efficient algorithm for some problems exist). The latter scenario would mean the existence of at least one problem that can be solved with a non-deterministic computer in polynomial time but cannot be solved with a deterministic computer in polynomial time. Let's informally call this subset class **INP**, for "Intractable within **NP**", or, in other words, "in **NP** but known not to have a polynomial time solution on a deterministic computer". Right now, nobody knows if the subset **INP** has any problems in it.

The relationship between **P** and **NP**

What is known about the relationship between problems with efficient algorithms in the class **P** and the remaining problems in the class **NP**? As stated earlier, every problem in **P** is also in **NP**. This can be demonstrated by looking at the formal definition of **NP**. Surely, if a deterministic computer (unfortunately, the only kind I have!) can solve a problem in polynomial time, a non-deterministic computer can solve it in polynomial time as well. Alternatively, if a solution can be devised in polynomial time, it must be the case that the solution can be verified in polynomial time as well since verification is built in to devising a solution. Formally, then, the class **P** is a subset of the class **NP**. Furthermore the class **P** is disjoint from the class **INP**, by definition. Finally, all **NP** problems must be in **P** or **INP**,

Earlier, it was stated that it wasn't really known whether some of the problems in **NP** that appear intractable are indeed intractable. Suppose you figure out that a particular problem in **NP**, which currently has no efficient algorithm, definitely belongs in either **P** or **INP**. In two scenarios, you will become rich and famous (famous among computer scientists, at least). In the third scenario, not so much.

Scenario 1

In this scenario, you find a polynomial time solution for a problem like **SUB** (**SUB** is a problem in the class of problems called **NP-complete** - more on this class of problems in the next section). This means that all **NP** problems will have efficient algorithms for their solutions. In this case, **P** will be an improper subset of **NP**, or more simply, **P** = **NP**, and the class **INP** will be known to be empty. Riches and fame ensue.

Scenario 2

In this second scenario, you prove that a problem in **NP** (it doesn't matter what kind, only that it is in **NP**) can only be solved in super-polynomial time. In this case, it and all the hard problems like **SUB** will immediately become members of class **INP**. In this case, **P** will be a proper subset of **NP**, or more simply **P** \neq **NP**. As in Scenario 1, riches and fame ensue.

Scenario 3

Here, you discover a polynomial-time algorithm for solving a problem in **NP** that currently has no known efficient algorithm. Unfortunately, this problem is not special like **SUB** is special. In such a case, we cannot make any grand claims about whether **P** = **NP** or not (though you may get rich if your polynomial time solution is much better than any previously known algorithm and the problem is of commercial interest).

Whether **P** = **NP** or not, or equivalently, whether **INP** is empty or not, is the greatest unanswered question in theoretical computer science.

The class **NP-complete**

SUB was claimed to be a *special* problem in that a polynomial time solution for it would imply all **NP** problems have efficient algorithms. Alternatively, if it were shown that any particular problem in **NP** could not be solved in polynomial time, **SUB** and all other *special* **NP** problems would immediately move into **INP**. Not all hard problems are known to be special in this way. Two examples are the *graph isomorphism* problem, which answers the question if two graphs are identical to each other, modulo vertex names, and *integer factorization*, which answers the question, does a composite number N have a factor greater than 1 and less than M ? At this point in time, the best (exact) algorithms for these problems are sub-exponential but super-polynomial. If someone found a polynomial time algorithm for graph isomorphism or factoring (<PARANOIA>I believe

the NSA has just such an algorithm</PARANOIA>), all it would mean is that the problem would belong to the class **P**, with no other implications as to whether **P** = **NP**.

Special problems, like **SUB**, are thus closely linked. Once any problem, special or not, becomes **INP**, at a minimum all special problems will become **INP** as well. Alternatively, in the case of a special problem becoming **P**, all **NP** problems, special or not, will belong in **P**. These special problems have their own class, called **NP**-complete. The *complete* comes from the fact that any one of these special problems completely covers **NP**; if a fast solution is found for one, **NP** completely becomes **P**.

To show that a problem, q , is **NP**-complete, one first shows that the problem q belongs to the class **NP**. This is typically rather easy to do since it is usually obvious how to verify a solution in polynomial time. The second step is not quite as simple. One needs to show that every instance of every problem in **NP** can be converted, in polynomial time, to an instance of q .

Now all problems become easy once a fast algorithm for solving q is found. Given an instance of an arbitrary problem, i , one simply converts i into an instance of q and feeds it to the fast algorithm. The answer the algorithm gives is then directly converted to the answer to the original problem instance. This conversion is made rather obvious by restricting the class **NP** to decision problems. That is, the answer to an instance of an **NP** problem will be either *yes* or *no*. For example, the graph isomorphism problem can either be viewed either as a decision problem (is graph G isomorphic to graph H ?) or as a problem that generates much more detail (what is the mapping of vertices in G to H should G be isomorphic to H ?). Intuitively speaking, if a decision problem is hard to answer, surely the non-decision problem is also hard to answer, so the class **NP** can be thought of as representing more than just decision problems. In addition, an algorithm to solve a decision problem can often be used by a non-decision algorithm to generate the details in polynomial time beyond the time used by the decision algorithm.

Still, this second task seems to be all but impossible. How does one show that all problems in **NP** can be converted, even problems that haven't been thought up yet!? Fortunately, a rather clever person named Cook showed that the satisfiability problem (**SAT**), which asks "is there an assignment of truth values to boolean variables such that an arbitrary logical expression of those variables resolves to true?" is **NP**-complete. Since that time, the task of showing some problem q is **NP**-complete has become much simpler. Rather than showing that every instance of every **NP** problem can be converted, in polynomial time, to an instance of q , we just show that every instance of the satisfiability problem (or some other **NP**-complete problem) can be converted, in polynomial time, to an instance of q . Why is this equivalent? Well, we can convert every instance of every hard problem to an instance of **SAT** in polynomial time (because it is **NP**-complete). Then we can convert the resulting instance of **SAT** to an instance of q in polynomial time as well. The time it takes to do the two conversions is additive, so the overall conversion takes polynomial time. So an efficient algorithm for q implies an efficient algorithm for all **NP** problems!

The future of NP-completeness

While it might be nice to dream of a fast algorithm for **SUB**, it is likely to be a fruitless task. There are a vast number of hard problems which a lot of people have spent a lot of time searching for fast algorithms, to no avail. Personally, I used to believe that **P** did indeed equal **NP** and I got my inspiration from the fact that it was believed that soap bubbles always contract and arrange themselves to minimize their surface energy. They do this feat very quickly. So all one would have to do is cast an **NP**-complete problem as a soap bubble system and allow the system to find its minimum energy. That final configuration would reflect the answer to the original problem. Recently however, someone showed that for a certain type of multiple bubble systems, the minimum energy configuration is not reached naturally, which implies that a soap bubble system representing an instance of an **NP**-complete problem might not find the correct answer.