CS 3510 Algorithms 3/28/2023

Lecture 15: Intro to Complexity

Lecturer: Abrahim Ladha *Scribe(s): Joseph Gulian*

Throughout this course we've been talking about algorithms, and most of the time we've been talking about substantially different problems (i.e. shortest path is not very similar to strongly connected components). However many problems have a very similar structure and often an algorithm for one problem can be used for another.

For instance think about the following problems: exploration and shortest paths. By simply running Djikstra's on a path, you could solve a problem requiring exploration of a graph. However the same is not true in reverse: you can not solve the shortest-path problem by running the explore routine. Then we have some idea that certain problems are harder than others and some algorithms for a certain problem can be used to solve other problems with similar structure.

1 Example Problem

Imagine you have some objects which are only ordered (you can only do less than and greater than) and you want a list of indices where the objects would be. A trivial algorithm would be to sort the list and keep track of the indices of the items which gives us a worst case lower bound of $\Omega(n \log n)$. Can we do it faster? Assume we could do it faster, there is some $o(n)$ algorithm. This creates a contradiction because we could use the indices to sort the list (its own sorting algorithm) but the worst case for comparison sorts is $\Omega(n \log n)$. [1] Therefore the complexity must be at least $\Omega(n \log n)$.

In fact this problem is essentially a transformation of one problem into another, you have a list of numbers, you encode their indices somehow, you sort, and then you decode their indices. Sorting here is the blackbox, you could plug in any comparison sort and solve this problem. Similarly, say a breakthrough was made in the comparison sorts, what does this mean for the runtime of this problem? It's lowered to whatever the new lower bound is.

This unit is all about this kind of thinking: you have two problems and you want to turn an instance of one problem into an instance of another problem.

2 Complexity

A decision problem is a problem which can be phrased as a yes or no question. For instance deciding whether or not a number is prime is a decision problem. Another example of a decision problem is $\textsc{Path} = \{\langle G, s, t\rangle \mid$ there exists a path from s to t in $G\}$. [2]

[1] This isn't terribly important but you can refer to CLRS 8.1

[2] What is this notation? This notation has some deeper meaning relating to "languages" in complexity theory, but for this course, none of that is important. To simplify it to the point of being wrong, imagine you had all tuples in the set with the given constraint, you could then decide whether or not some given tuple is present in that set.

A search problem is a type of problem is a problem requiring a solution to be found. Search problems have a corresponding decision problem. In the above case a search problem may ask to find such a path, whereas the decision problem would be does such a path exist.

We'll now define a new class called P, P is the set of all decision problems which can be solved in polynomial time. The set P is closed under many operations, for instance if you wanted to run two efficient algorithms for problems in P, the runtime would also be in P. Alternatively if you wanted to run an efficient algorithm for a problem in P and then run an efficient algorithm on another problem in P using the output from the previous problem.

During this course, you may have developed a notion that all the problems we talk about are in P. This is sort of interesting. Almost every problem has a trivial exponential solution; consider for the PATH problem above, instead of doing something like explore, you could list all possible combinations of paths and check each one. Obviously we don't do this though, we've found some deeper structure in the problem we can use to build our solution. Not all problems have this structure that allows us to build a simple algorithm in P though.

Another class of problems is NP which stands for nondeterministic polynomial. [3] A problem is in the class NP if a solution to the problem can be verified in polynomial time given a witness of the execution. For instance in the path problem above the witness consisting of the nodes in the path could be checked by ensuring there is an edge between every node in the path and the nodes start with s and end with t.

All problems in P are definitely in NP, it's possible that not all problems in P are not in NP. One instance of a problem which may not be in P but is in NP is factoring. As we talked about, factoring a large number is not possible in polynomial time with current algorithms we know. [4] However given a witness of execution consisting of the factors of a number, you can simply multiply them together and check that it is the number that was supposed to be factored. [5]

3 P vs NP

You may have heard of P vs NP from popular culture; are the classes P and NP equal? [6][7] A lot of researchers have tried and failed to prove something about P vs NP and have

[3] Determinism in this context does not imply anything about randomness as it might in some other contexts. Instead (again, I'm simplifying to the point of being wrong), nondeterministic implies having possibly a non-finite number of threads of execution all executing at the same time.

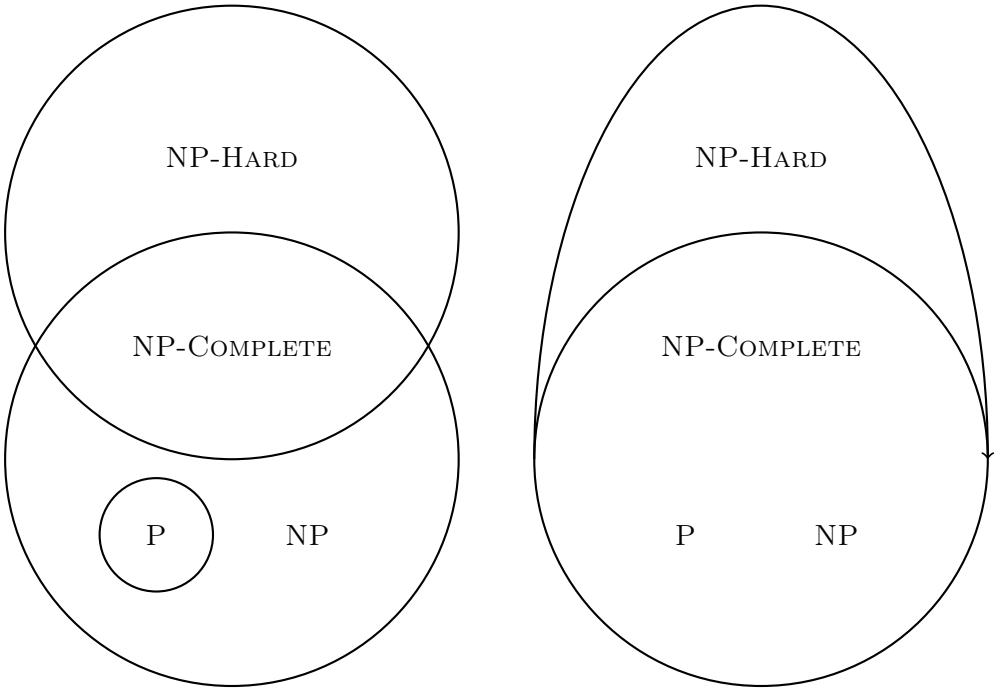
[4] This is the basis for much of the cryptography we discussed earlier.

[5] To tie NP back to nondeterminism, the class NP is about solving problems with nondeterministic machines. As an example consider the factoring problem above. On a nondeterministic machine, in a polynomial number of steps we could develop an exponential number (all) of the witnesses. You could then check every witness and find an answer.

[6] This is the scribe talking, some people I know believe P equals NP and many more people believe P does not equal NP. It's not such much that they know P vs NP though, it's more of a feeling because no one really knows P vs NP... what's the point? Just be careful with your languages and how you think about things when it comes to P vs NP you don't want to accidentally imply P vs NP... if you do have sufficient evidence, don't file a regrade request, write a paper and become the biggest complexity theorist ever. If we (not me) ever imply something about P vs NP, think of it as something coming from the point of view of someone trying to prove it.

[7] This is the instructor talking, I personally believe P vs NP is unprovably unsolvable. Youll learn the proper way to discuss nondeterminism and these ideas when you take 4510

failed. There are many ways people have attempted to prove P vs NP.



if P $\neq$ NP if P = NP

Let Exp be the problems solvable in polynomial time (probably all the problems we've discussed in this class). We know that P $\subsetneq$ Exp, but there are problems in Exp provably not in P. We also know we can brute-force solve problems in NP so we know then that P $\subseteq$ NP $\subseteq$ Exp. Since P $\subsetneq$ Exp, one of the following is true: P $\subsetneq$ NP or NP $\subsetneq$ Exp. Then if you could show something about NP*and*Exp, you could show something about P vs NP.

Let L-Space = SPACE($\log n$) and P-Space = SPACE(n^k). By the space hierarchy theorem, we know L-Space $\subsetneq$ P-Space, but we also know L-Space $\subseteq$ P $\subseteq$ NP $\subseteq$ P-Space. If you could prove L-Space = P and NP = P-Space, then you would prove P-Space $\neq$ NP. Similarly if you could prove P = P-Space, you would prove P = NP. We have no idea how to solve basically any of these open structural questions in complexity since they might accidentally imply something about P = NP

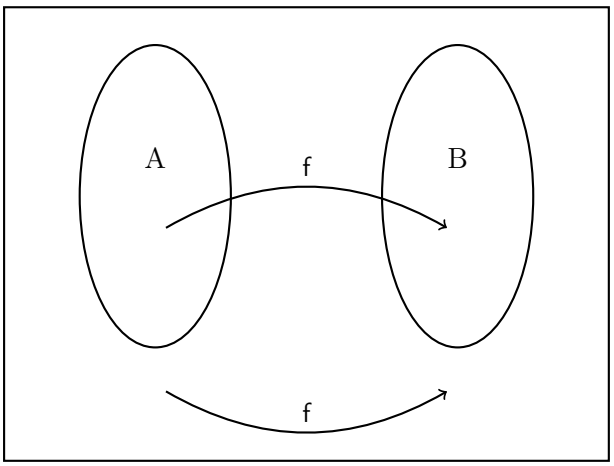
4 NP-Complete

The figure above has two classes we have yet to talk about, these are NP-Hard and NP-Complete. NP-Hard is an interesting class of problems; if a problem is in the class NP-Hard, that means that if a polynomial time algorithm is found to solve it, there is a polynomial time algorithm to solve all the problems in NP. More succinctly, if you can solve an NP-Hard problem in polynomial time, you have shown P = NP. A problem being NP-Complete means it is both NP-Hard and NP. This means that problems in

NP-Complete can be used to solve each other, and are to such a complexity that solving one would give a polynomial time algorithm for all of NP.

Surprisingly, showing a problem is NP-Complete is not as challenging as one might expect. [8] We'll show problems are NP-Complete by showing that one problem can be used to solve another problem; we do this to show that the problem is harder. Then we show that it is in NP by giving a verifier for it.

A reduction is the word we use to describe turning a problem from one instance into a problem in another instance. A reduction from A to B means problem B is at least as hard as A. All reductions you do should be in polynomial time in this course for the simple reason that if you're trying to show that A can be solved in polynomial time if B can be solved in polynomial time, but you're using an exponential time reduction to convert A to B, then you're not actually showing the implication. Lastly the notation $A \leq_{\mathrm{P}} B$ implies that A can be reduced to B. You could think of a reduction as mapping f from A to B.



[8] Well actually it was vary challenging to show the first problem was NP-Complete. We won't cover how or why it was proven we'll cover the first problem soon which you'll use to show other problems are NP-Complete.