

# 1 Introductory Complexity

- So we talked a bit about the fact that the TSP is much harder than the car braking problem. Why? Is it just that we're not clever enough or is there something fundamental?
- It turns out that there are some problems that are just *fundamentally* harder than others. (At least, as far as we can tell.)
- It's useful to have a sense of which category a given problem falls into – gives us a sense of whether we can get an exact solution quickly or whether we need to approximate it.
- Complexity theory tries to quantify this.
- Basic notions are classes of problems that can be solved in polynomial time, polynomial space (but exponential time), exponential time and space, etc.
- Example: most algorithms you've dealt with in intro algs classes are computable in polynomial time: sorting, searching, shortest-paths, etc.
- Call this the class **P**: polynomial time computable.
- Next class up is **NP**. Technically, this stands for nondeterministic polynomial-time computable.
- More informally, it means that if you can *guess* the right answer, then you can *check* to make sure that it's right in polynomial time.
- Note: all problems in **P** fall into **NP**:  $P \subseteq NP$
- A bit more formally, **NP** is defined in terms of *decision problems*: problems with a YES/NO answer
- So for the TSP the **NP** formulation is not “find the shortest path”, but “is there a path with length  $\leq k$ ?”
- This is a subtle difference, but an important one. It may be, for example, that we could answer the YES/NO question *without* actually being able to produce the path.
- Note that if someone *shows* us a tour in the graph, we can verify that the length is  $\leq k$  easily enough. It's finding it in the first place that's hard.
- But how do we know that it's hard to actually *find* the answer in some reasonable time?
- Well, strictly speaking, we *don't* know that. But practically everybody believes it.
- TSP has the nice property that *every* problem in **NP** can be converted into it.
- E.g., consider the *Hamiltonian cycle* problem: given an (unweighted, incomplete) graph, find a cycle that touches each vertex exactly once.
- This problem is *also* in NP:

- If we are given the answer (magically), then clearly we can check to make sure that it really is a HC in only polynomial time.
- Now, HC and TSP look similar – in fact, TSP looks harder b/c you have to think about edge weights as well.
- It turns out that you can use a TSP solver (if you have one) to solve an HC. That is, given any HC problem, convert it into a TSP problem such that the solution to the TSP will give you the solution to the HC.
- Demonstration of reduction here. . . (Class exercise?)
- This process is called *polynomial reduction*: it reduces an instance of HC to an instance of TSP. It turns out that you could reduce *any* problem in **P** to some instance of TSP.
- In fact, HC *also* has this reducibility property
- But many problems in **P** don't – e.g., you really just can't use quicksort to solve TSP (at least, without re-representing the problem in a way that takes exponential time to write down)
- Things that have this reducibility property and are **NP** can, in some senses, be considered to be the “hardest” problems in **NP**.
- That is, since you can use TSP to solve *any* other problem in NP, no other problem is harder than it. Similarly, no problem is harder than HC *either*, so HC and TSP are kind-of “equally hard”.
- The class of things which are “the hardest problems in **NP**” are called “**NP**-complete” and live at the “top” of the class **NP**.
- Since we know that we could use something in NP-complete to solve any other problem in NP, if it turns out that there's some way to solve *any* NP-complete problem in polynomial time, then you could use it to solve *every* problem in NP in polynomial time. Thus, if anything in NP-complete is also in P, then  $P = NP$
- Unfortunately, nobody knows any way to do that – all the best people who have fought with it for the last 50 years haven't been able to find a way to solve any NP-complete problem in polynomial time.
- Equally annoying, however, they also haven't been able to prove that it *can't* be done either. Bleh. So we're left not knowing for sure whether  $P = NP$ . But we *strongly believe* that they're not the same.
- Note that there's a \$1 million prize for proving the question one way or the other.
- But it's useful to know that there's such a thing as NP-complete problems, b/c if you can show that something you're working on is NP-complete, then you know that it's really, really hard. And that you won't be able to get an exact solution, and you'll have to settle for an approximation. (Which is what most of this class is about.)

- So how do you prove that something is NP-complete?
- Won't go through the rigor in this class, but we have all the basic parts already:
  - Show that if you can guess the correct solution to your problem, then you can verify that solution in polynomial time. (This is also called having a polynomially verifiable certificate.)
  - Pick some other problem that's known to be NP-complete and show that your problem is at least as hard as that. That is, show that any instance of the known NP-complete problem can be transformed into some instance of your problem (in polynomial time) and that a solver for your problem could find a solution to this new problem.
- Doing those two things is hard (usually the second is harder than the first), but if you do, then you establish that your problem is as hard as anything in NP.
- Of course, that's not the whole story. It turns out that there's something even harder – **PSPACE**.
- PSPACE is the set of things which can be solved in polynomial *space*.
- Note that something that takes only polynomial space might take exponential time...
- And, correspondingly, there are PSPACE-complete things up at the top of PSPACE. PSPACE-complete things are as hard as anything in PSPACE, and any problem in PSPACE could be turned into one of them.
- Playing *perfect* chess turns out to be PSPACE complete. (Perfect in the sense, can you show that player 1 must always win or must always lose.)