More NP-complete Problems and Dealing with Intractability

Euan Mendoza

2023

1 Intractability and NP-completeness

In the study of the theory of computing there are some fundamental goals we set out to achieve. One of the first goals was to ask are decision problems *decidable*. This let's us know if we can write an algorithm that solves a problem. However, in practice knowing if *there exists* an answer or not is not useful unless we can find the answer *before the heat death of the universe*.

This is the primary motivation for the notion of *intractable* problems. A problem is *thought to be intractable* if it cannot be solved in a *reasonable amount of time*. What constitutes a reasonable amount of time can be seen as the primary motivation for the complexity classes P and NP, as well as the motivation for the complexity class NP-complete.

Before describing P vs NP, it is important to revisit the notion of determinism vs non-determinism. We found out that for finite state automata, non-determinism is equivalently as powerful as determinism. We also found out that deterministic context free grammers are *not* as powerful as non-deterministic context free languages. So it was surprising to find out that all deterministic turing machines are equivalently as powerful as non-deterministic turing machines in terms of computability. However, this is not the complete picture. We also want to know if problems that can be solved *efficiently* on non-deterministic turing machines can also be solved *efficiently* on deterministic turing machines.

Definition 1.1. The complexity class P is the class of decision problems that are *decidable* in *polynomial time* ($O(n^k)$), on a deterministic single-tape turing machine.

Definition 1.2. The complexity class NP is the class of decision problems that are *decidable* in *polynomial time* ($O(n^k)$), on a non-deterministic single-tape turing machine.

We can now see clearly that the question of P vs NP is really a question of determinism vs non-determinism. There is also the equivalent definition of NP as the class of problems that can be *verified* in polynomial time on a deterministic turing machine, which can be found in sipser.

Quite often we say an algorithm is *efficient* if it is in P. One of the biggest open questions in computer science that we are now (almost) equipped to tackle is this problem of P vs NP.

1.1 Aside: Proving Languages are in NP

When I started in computer science one of my absolute favourite concepts was the notion of a non-deterministic turing machine. The notion of non-determinism in turing machines is also fundamental in proving that a language is in NP.

One of the most magical parts of a non-deterministic turing machine is it acts like a normal turing machine augmented with a step usually referred to as *guess* or *find* which can search for a solution to a problem non-deterministically in polynomial time. How this *actually* works can be left as an excercise to the reader. Nevertheless the step can be described as a peak into the future and finding an answer.

As a general strategy for proving a language is in NP we need to think about the notion of *verify*. Usually we follow the general framework of *non-deterministically guess a possible answer* and verify it in polynomial time.

Recall that a *GraphIsomorphism* GraphIso $= \{(G, H) \mid \exists \phi, \phi(G) = H\}$. This is the language that given two graphs, accepts if there is a phi ϕ function that maps all the vertices of G to H and preserves the edge relations. This has a corresponding non-deterministic algorithm.

Lemma 1.1. *GraphIso is in NP.*

Proof. Consider the following algorithm.

Clearly the guess step is polynomial time bounded, the second step corresponds to the definition of the language. If ϕ is an isomorphic function, and $\phi(G) = H$, than by definition of GraphIso, the inputs are in the

Input: G, H are graphs
Output: Accept or Reject

1. guess non-deterministically a isomorphic function ϕ
2. if $\phi(G) = H$, return Accept
3. else return Reject

---

language, while if there is no ϕ, than step 1 would not be able to find the function and the algorithm would reject.

■

This general framework of first showing that you can guess an answer non-deterministically, and than verify that the answer is in the language defined is how we prove problems are in NP.

1.2 Polynomial (Karp) Reductions and the Definition of NP-complete

NP-complete is often cast as a problem of intractability but theoretically (I think) it is really a question of determinism and non-determinism. A open question is P = NP? How would we go about proving that P = NP. We first need to describe reducibility amongst problems.

We have already seen that we can reduce a problem to another problem by finding a computable function that *changes the inputs* of a turing machine from a turing machine that decides one language to a turing machine that decides another. Formally,

Definition 1.3. Let A, and B be decision problems (or languages decided by turing machines). A is *mapping reducible* to B denoted $A \leq_m B$ if and only if there exists a computable function f such that for every

$$w \in A \iff f(w) \in B$$

and the function f is called a *reduction* from A to B.

This has a useful property that if there is a turing machine M that can decide B there is a turing machine N that can decide A given by $N = f \circ M$. Or simply we can find an algorithm that decides problem A by using the algorithm that decides problem B and changing the inputs to the inputs of problem A.

We have to ask out of curiousity, is this really that helpful for proving P = NP? Not really because this doesn't tell us anything about the complexity class NP. We need to first define the notion of a polynomial time (karp) reduction.

Definition 1.4. Let A and B be decision problems (or languages decided by a turing machine). A is *polynomial time reducible* to B denoted $A \leq_P B$ if and only if there exists a polynomial time computable function f such that for every input string w of A

$$w \in A \iff f(w) \in B$$

and the function f is called a *polynomial-time reduction* from A to B.

This seems similar to the definition of mapping reduction, however now we know that if f takes a polynomial amount of steps, than clearly if a problem B is in NP and A is reducible to B than A must also be in NP by chaging the inputs for A to the inputs of B. It is also important to note that a polynomial time reduction can be used to describe the degree of difficulty of a problem. Clearly B is harder than A if A is polynomial time reducible to B since if we can solve B we can also solve A.

It is also important to note that when giving a polynomial time reduction it is not sufficient to only give the algorithm. You *must* also provide a proof that the reduction *is correct*, or that for every input of a problem B it changes the input to a problem B and vice versa. You also must provide a *worst-case analysis* of the running time.

From here we can describe the notion of NP-hard and NP-complete. NP-hard can be seen as atleast as hard as every problem in NP, and NP-complete is the hardest problems in NP.

Definition 1.5. A problem B is said to be *NP-hard* if and only if for every problem A in NP, A is polynomial time reducible to B. A problem is *NP-complete* if it is NP-hard and in NP.

This definition has *two* important consequences.

1. If a problem in NP-complete can be solved in polynomial time, than P = NP.
2. If a A problem is known to be NP-complete and another problem B is in NP and A is polynomial time reducible to B, than B must also be NP-complete.

We now have a framework to both describe how to *come up* with the hardest problems in NP and to prove that P = NP.

2 First NP-complete problems

If we know some *NP-complete* problems, than finding others is easy since we can just reduce a known NP-complete problem to the new problem. So how would we find our first NP-complete problem.

Theorem 1 (Cook-Levin)**.** *SAT is NP-complete*

Cook and Levin proved that SAT can be reduced in polynomial time to every problem in NP. This is a foundational problem in theoretical computer science. It allows us to find other NP-complete problems simply by reduction.

From SAT, the following problems are known to be NP-complete.

Theorem 2. *3-SAT is polynomial time reducible to SAT (as proved last week), so in this week's tutorial we show that the following problems are also NP-complete.*

1. *Independant Set, decide if there exists a size k subset of the vertices of a graph G such that no two vertices are adjacent (Reduction given in lecture).*
2. *Hamiltonian Path, decide if there exists a path in a graph G such that each vertex is visited once. (Reduction given in lecture).*
3. *Subset Sum, decide if there exists a subset of a set of integers such that they sum to some input value N. (Reduction in lecture).*
4. *Vertex Cover, decide if there exists a subset of the vertices of a graph G such that each vertex is connected to each edge in G. (Reduction from last weeks tutorial).*

3 Additional helpful notes

3.1 Space Complexity

3.2 Other Complexity Classes and the Polynomial Time Hierarchy

3.3 Different Types of Reductions

3.4 How to Handle Intractability

1. Talk about the complexity class co-NP.
2. Talk about P-space.
3. Talk about Karp and Turing reductions.
4. Space Complexity.
5. Dealing with intractability (use Luke's slides).