NP-Completeness, Efficient Computability + Reductions

Goals: Formalize ideas about complexity. What does it mean to be $\underbrace{\text{NP-complete, NP-Hard}}_{\text{not the same!}}$?

How do we show a problem is NP-complete?

Decision vs. Optimization Problems

- So far, we've mostly dealt with optimization problems
- Computational complexity most commonly deals with decision problems
- The output of decision problems is "Yes" or "No" – (1 or 0).

- For example, circulation with demands and Max flow are both optimization problems.

The decision version of a problem is no harder (sometimes easier than) the optimization version.

E.g. Max Flow (decision): "Is there a flow value of at least C?"

- If you solved the optimization problem, you could answer the decision problem. For example, if you find the max flow f^*, just return $v(f^*) \geq C$.

Fact: If the decision problem is hard, so is the optimization problem.

Problem Instances and Encodings:

For the purpose of formalizing the notions of complexity, we must choose how to encode instances of a problem. One natural way is to encode each instance as a string (we can consider strings of text, but these are convertible to binary strings).

E.g. An encoding of a Max Flow instance might be

$$u_1, v_1, c_1; u_2, v_2, c_2; \ldots ;; s, t, C$$

All the problems we've considered so far, and all with which we'll be concerned, can have their instances represented as strings.

- They <u>are</u> represented in RAM as a string of bits

A <u>decision problem</u>, X, is actually just a set of strings!

E.g.

Instance	$\in X$
1,10,5; 3,7,20; 10,3,15;; 1,7,5	YES
1,10,5; 3,7,20; 10,3,15;; 1,7,260	NO
⋮	⋮

<u>Definition</u>: A <u>language</u> is a set of strings

So, any "decision problem" is formally equivalent to deciding membership in some language.

That is, for some decision problem P, whose language is L, we say that instance I of P is a YES instance if and only if encoding(I) $\in$ L.

⟹ Note: We will discuss decision problems and languages almost interchangeably.

How can we say that a decision problem is hard?

- Ultimately, we want to say that a computer cannot recognize some language efficiently.

- For our purposes, computer means Turing Machine

• The Church-Turing thesis tells us that everything that is efficiently computable is efficiently computable on a Turing Machine.

Turing Machine

At each time step, the Turing Machine
- reads the symbol at the current position
- Depending on the symbol and current state of the machine, it:

• Writes a new symbol x, • moves left or right, • changes to a new state S

- The set of symbols is <u>finite</u> and non-empty
- The set of states is <u>finite</u> and non-empty

Formally:

$M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$

Q = finite, non-empty set of states
Γ = " " set of tape alphabet symbols
$b \in \Gamma$ = blank symbol
$\Sigma \subseteq \Gamma \setminus \{b\}$ = input symbols (allowed to appear on input tape)

$\delta : (Q \setminus F) \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ = transition function where L, R tell the machine to move Left or Right on the tape.

q_0 = initial state

$F \subseteq Q$ = final or accepting states. The initial tape is accepted by M if it eventually halts in a state from F.

Given all of this machinery, we can define the <u>class</u> of problems P

Def: P is the set of languages whose memberships are decidable by a Turing Machine that makes a polynomial number of steps.

by the CT thesis, this is equivalent to

Def: P is the set of decision problems that can be decided by a computer in polynomial time.

Defining another class, NP, will require some new ideas

Certificates

Recall the independent set problem

Problem (Independent Set). Given a graph G, is there a set S of size $\geqslant k$ such that no two nodes in S are connected by an edge?

- We'll see that <u>finding</u> S appears to be hard

But, if I give you some set S^*, <u>checking</u> whether S^* is an answer is easy: Check that $|S^*| \geqslant k$, and that no 2 nodes in S^* are connected by an edge.

We say that S^* acts as a <u>certificate</u> that $\langle G, k \rangle$ is a "Yes" instance of independent set.

Def: An algorithm B is an <u>efficient certifier</u> for problem X if:

1) B is a polynomial-time algorithm that takes two input strings I (an instance of X) and C (a certificate)

2) B outputs either "Yes" or "No"

3) There exists a polynomial $p(n)$ such that for every string I:

   $I \in X$ if and only if there exists a string C of length $\leq p(|I|)$ such that $B(I, C) =$ "Yes"

Thus, B is an algo that can decide if an instance I is a "Yes" instance given the "help" of a <u>polynomially-long</u> certificate.

Def: NP is the set of languages for which there exists an efficient certifier.

Def: P is the set of languages for which there is an efficient certifier that <u>ignores the certificate</u>.

The main difference:

A problem is in P if we can decide it in polynomial time. It is in NP if we can decide it in polynomial time given the right certificate.

Note: We don't need to be able to <u>find</u> the certificate, we just need to be able to use it.

Theorem: $P \subseteq NP$

Proof: Assume $X \in P$. Then there is a polynomial-time algorithm A for X.

To show $X \in NP$, we need an efficient certifier $B(\cdot,\cdot)$

Let $B(I, C) = A(I)$

Every problem with a polynomial-time algorithm is in P.

The big question: $P = NP$?

Are there some problems in NP that <u>are not</u> in P?

Is checking a solution fundamentally easier than finding one?

This seems natural, but nobody has yet been able to prove it!

- $P = NP$ is a remaining Millenium Problem
- "Proofs" both ways appear on arXiv regularly
- It <u>seems</u> fundamentally new techniques will be required to solve this problem.

How do we prove that a problem is likely hard?

- We'll assume that there exist some problems that have already been proved to be (probably) hard.

We'll use the concept of a reduction to show that some new problem is likely hard.

- Problem X is at least as hard (w.r.t polytime) as problem Y.

To prove such a statement, we'll reduce problem Y to problem X.

If you had an algorithm A that could solve problem X, how could you solve problem Y using a polynomial number of steps plus a polynomial number of calls to A?

Already seen some reductions:

Max Bipartite Matching $\leq_p$ Max Flow
Circulation with demands + lower bounds $\leq_p$ Circulation with demands
Circulation with demands $\leq_p$ Max Flow

If problem Y is polynomially reducible to problem X, we denote this as:

$$Y \leq_p X$$

This implies that **X is at least as hard as Y!**

* Commit this direction of reduction to memory. One of the most common mistakes is to do a reduction in the wrong direction. For example, if I show I can sort an array of numbers by solving Independent Set, this does not prove that sorting is hard, it shows that Independent Set is at least as hard as sorting (... unexciting).

Note: We reduce to the problem we wish to show is at least as hard.

Suppose:

- $Y \leq_p X$
- There is a polynomial-time algo for X

Then, there is a polynomial-time algo. for Y. Why?

- Because polynomials compose
- If we can create a polynomial number of instances of X, each in polynomial time, and each instance can be solved in polynomial time → we have a polynomial-time algorithm for Y.