6.1220 LECTURE 14

BRENDON JIANG

Contents

1. Intractability I

So far, we have not rigorously though about how running time is defined. Here, we give a more rigorous definition

Definition 1.1: Running time

A running time is a function of the size of the input. The size of the input is the number of bits that it contains.

We also define an efficient algorithm to be one that has a polynomial running time, or running time $= O(n^c)$ where c is a fixed constant and n is the size of the input.

Now, let us consider some "innocent variations" of seen problems with efficient solutions.

- Minimum Spanning Tree $\rightarrow$ Maximum Spanning Tree. We can get an efficient algorithm for maximum spanning tree simply by negating the edges of the graph.
- Minimum Cut $\rightarrow$ Maximum cut. There is no known efficient algorithm to get the Maximum cut of a flow network.
- Linear Programming $\rightarrow$ Integer Linear Programming. Still no efficient algorithm.
- Shortest Path $\rightarrow$ Longest Path. One can reduce the problem of getting the longest path to get the Hamiltonian path in a graph, which is known to be a hard problem to solve efficiently.

We also clarify here that instances are a specific configuration of a problem. For example, finding the minimum spanning tree is a problem, where as finding the minimum spanning tree for a specific graph is an instance of the MST problem.

1.1. Characterization of problems

There are several types of problems. We list three of them here as decision, search, and optimization.

Date: October 2024

Problem \ Variant	Decision	Search	Optimization
Shortest Path	Instance (G, s, t, k). Does there exist s-t path in G of length $\leq k$?	Instance (G, s, t, k). Find a s-t path of length $\leq k$	Instance (G, s, t). Find the shortest s-t path in G.
Maximum Flow	Instance (G, k). Is there a flow with value $\geq k$?	Instance (G, k). Find a flow of value $\geq k$	Instance (G). Find a maximum flow on G.

Note that if we can solve optimization problems, we can solve the corresponding search problem. If we can solve search problems, we can solve the corresponding decision. Now, we look at ways to characterize decision problems.

1.2. Complexity Classes

Definition 1.2: Solvability in Polynomial Time

A decision problem π is said to be solvable in polynomial time if and only if there exist a determinsitic algorithm A_π such that on every input/instance x of size n,

(1) $A_\pi(x)$ runs in $O(n^c)$ time for some fixed c.
(2) $A_\pi(x) = \text{YES} \iff \pi(x) = \text{YES}$.

Definition 1.3: Complexity class P

$P = \{\pi : \pi \text{ is solvable in polynomial time}\}$.

The complexity class P is essentially the "easiest" class of problems to solve, because they can all be solved in polynomial time. We look at another class now:

Definition 1.4: Non-deterministic polynomial time

A decision problem π is solvable in nondeterministic polynomial time if and only if there exists a deterministic algorithm ν_π such that on every input/instance x of size n,

(0) The function ν_π takes on inputs x, y, with $|y| \leq n^{c'}$ for some fixed c'.
(1) $\nu_\pi(x, y)$ runs in $O((|x| + |y|)^c)$ time for some fixed c.
(2) $\pi(x) = \text{YES} \iff \exists \nu_\pi(x, y) = \text{YES}$.

Essentially, this says that we are able to verify a solution works given a proof in polynomial time.

Definition 1.5: Complexity class NP

$NP = \{\pi : \pi \text{ is solvable in non-deterministic polynomial time}\}$.

Although this definition is unexpected compared to the natural extension, this formulation of the NP class is equivalent to if we extended the definition for polynomial time to non-deterministic polynomial time.

Now, we can make a few basic claims about the complexity classes.

Claim 1.6

$P \subseteq NP$

Proof

$\forall \pi \in P$, simply take $\nu(x, y) = a(x)$. This is a valid solution validator, therefore any element in P is in NP.

Example 1.7

An open question in complexity is if $P \neq NP$. We do not know if the inequality is true, but theorists belive it is indeed true.

We can extend the definition of NP to coNP, which asks if the negation of the problem can be validated in NP time.

Claim 1.8

$P \subseteq coNP$.

Proof

$\forall \pi \in P$, take $\nu(x, y) = \neg a(x)$.

Corollary 1.9

$p \subseteq NP \cap coNP$.

Reduction

Definition 1.10: Reduction

Let Q, π be decision problems. A reduction from Q to π is a deterministic algorithm R such that

(1) $R(x) = y$, where x is an instance of Q and y an instance of π.
(2) R runs in polynomial time in $|x|$.
(3) $Q(x) = \text{YES} \iff \pi(y) = \text{YES}$.

Definition 1.11: Reducibility

$Q \leq_p \pi \iff$ There exists a reduction from Q to π.

Definition 1.12: NP-hard

π is NP-hard if $\forall Q \in NP, Q \leq_p \pi$.

Definition 1.13

π is NP-complete if it is both NP and NP-hard.

Claim 1.14

$Q \leq_p \pi'$ if $Q \leq_p \pi$ and $\pi \leq_p \pi'$.

Proof

We can run function composition using the reducability idea to get an algorithm for Q in polynomial time of π'.

Claim 1.15

$Q \leq_p \pi \in P \Rightarrow Q \in P$.

Proof

Function composition again.