Overview Of Complexity Theory

Classes Of Problems Based On Their Best Algorithms’ Complexities

Tractable/Intractable Problems

- Informally speaking, a problem is called “tractable”:
    - If there exists an algorithm that solves it (of course).
    - And its worst case running time is $O(n^k)$ for some constant k on inputs of size n.
- Almost all algorithms we learned so far are polynomial time, so those corresponding problems are tractable.
- Not all problems are solvable in polynomial time!
    - Many problems are even unsolvable. E.g., the halting problem.
    - Also, for many problems, there are no known polynomial algorithms.
        - Worst case running times of all known algorithms to such a problem are $\omega(n^k)$ for ***any*** constant k.

Complexity Class P

- Informally, it is the set of all problems each of which can be solved by some algorithm in the worst case running time of $O(n^k)$ for some constant k on inputs of size n.
- This corresponds to the class of “tractable” problems.
- There are many other complexity classes, based on the complexity measure’s characteristics:
  - For the class P, the complexity measure is “the worst case running time of any algorithm that solves the problem” and the characteristics is that its asymptotic notation is $O(n^k)$ for some constant k on inputs of size n.
  - We’ll see other complexity classes based on other complexity measures and/or different characteristics.

Simplifying Different Types Of Problems

- There are many different types of problems.
- Some problem asks if there exists a solution and what it is if any.
    - E.g., the path finding problem given a graph and two vertices
    - We drop the part of finding “what it is”, focusing on the “if there exists a solution” part.
    - The answer to the existence question becomes yes/no, making it a “decision” problem.
- Some problem asks to find an optimal solution (many optimization problems we’ve seen).
    - E.g., the shortest-path finding problem given a graph and two vertices
    - We want to simplify these problems to “decision” problems as well.

Simplifying Optimization Problems To Decision Problems

- Add an additional piece of information to the problem instance and convert it to a decision problem.
- E.g., for the shortest-path finding problem,
  - Add to the problem instance (a graph and two vertices) the maximum length of any path (call it k).
  - And the problem is now to decide whether there exists a path of length at most k in the graph between the two vertices.
- This may seem to simplify the given problem too much.
- However, complexity class-wise, such a simplified decision problem is no easier than the original optimization problem.
- Therefore, we’ll only consider decision problems in discussing complexity classes.

Formal Language Theory

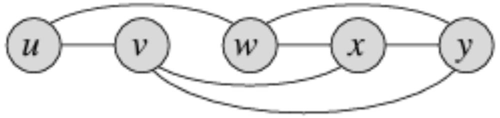
Mathematically Precise Description Of Decision Problems

Brief Intro To Formal Language Theory

- By focusing on decision problems, we can utilize the machinery of formal-language theory. Here's some brief intro of the theory.
- An *alphabet* Σ is a finite set of symbols. E.g., $\Sigma = \{0,1\}$
- A *language* L over Σ is any set of strings made up of symbols from Σ.
  - E.g., $L = \{1,01,10,001,010,010,111, \dots\}$ is the language of all binary strings containing odd number of 1s.
- A few more definitions:
  - The *empty string* is denoted by ε and the empty language by $\emptyset$.
  - The language of all strings over Σ (sometimes called *universe*) is denoted by Σ^*.
    - E.g., if $\Sigma = \{0,1\}$, then $\Sigma^* = \{\varepsilon, 0,1,00,01,10,11,000, \dots\}$, the set of all binary strings.
- Every language L over Σ is a subset of Σ^*.

Encoding Instances of a Decision Problem

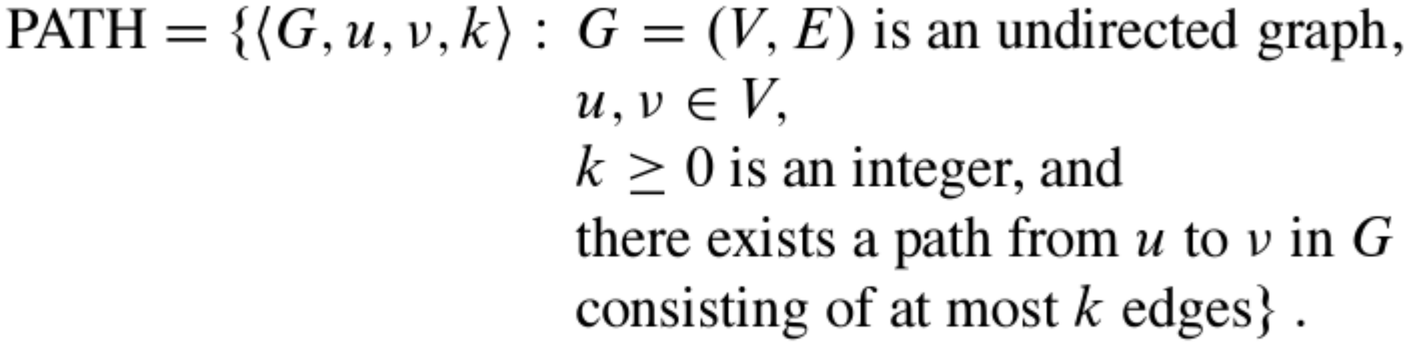
- Formally specifying every instance of a decision problem as a string encoding
- E.g., for the problem of deciding whether there is a path of at most length k between two vertices v_1 and v_2 in a given graph $G = (V, E)$,
  - If G is as shown below, v_1 is u, v_2 is y, and k is 2, then this instance of the problem will be encoded as follows:
    - "$\langle(\{u, v, w, x, y\}, \big\{\{u, v\}, \{u, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{w, y\}, \{x, y\}\big\}), u, y, 2\rangle$"
    - The answer to this specific instance is true, since there's a path of length 2 from u to y.
  - For the same G, but if v_1 is v, v_2 is w, and k is 1, then the encoding will be:
    - "$\langle(\{u, v, w, x, y\}, \big\{\{u, v\}, \{u, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{w, y\}, \{x, y\}\big\}), v, w, 1\rangle$"
    - The answer to this specific instance is false, since there's no path of length 1 from v to w.



Decision Problems and Formal Languages

- Note that the example problem instances given in the previous slide are just *strings* of symbols.
- Using any valid encoding of symbols (e.g., ASCII), those strings are ultimately encoded as binary strings.
- In the reverse direction, any binary string can be checked if it’s a valid encoding of any given problem in the given syntax (parsing).
- We can think of the set of all strings that are valid encodings of all *true* instances of a decision problem.

- In fact, we can view any decision problem Q as a language L_Q over $\Sigma = \{0,1\}$, where
$$L_Q = \{x \in \Sigma^* : Q(x) = 1\}$$
- For example, the problem of deciding whether there exists a path of length at most k for a given graph and two vertices has the corresponding language:



$$\begin{aligned}\text{PATH} = \{\langle G, u, v, k\rangle : \ & G = (V, E) \text{ is an undirected graph,}\\ & u, v \in V,\\ & k \geq 0 \text{ is an integer, and}\\ & \text{there exists a path from } u \text{ to } v \text{ in } G\\ & \text{consisting of at most } k \text{ edges}\}\ .\end{aligned}$$

- For the two examples we've seen earlier,
  - "$\langle(\{u, v, w, x, y\}, \big\{\{u, v\}, \{u, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{w, y\}, \{x, y\}\big\}), u, y, 2\rangle$" ∈ PATH
  - "$\langle(\{u, v, w, x, y\}, \big\{\{u, v\}, \{u, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{w, y\}, \{x, y\}\big\}), v, w, 1\rangle$" ∉ PATH

Languages and Algorithms

- A string (problem instance) and an algorithm
  - An algorithm A *accepts* a string $x \in \Sigma^*$ if, given input x, the algorithm's output $A(x)$ is 1.
  - An algorithm A *rejects* a string $x \in \Sigma^*$ if the algorithm's output $A(x)$ is 0.
- A language (problem) and an algorithm
  - A language (problem) L is *decided* (solved) by an algorithm A if A accepts every string $x \in L$ and A rejects every string $x \notin L$.
  - A language (problem) L is *decided* (solved) *in polynomial time* by an algorithm A if there exists a constant k such that for any length-n string $x \in \Sigma^*$, the algorithm correctly decides (accepts or rejects) whether $x \in L$ or not in time $O(n^k)$.

Complexity Classes
In Formal Language Theory

Back To Complexity Classes Using Formal Language Theory

Complexity Class

- E.g., the language PATH is decided (solved) in polynomial time!
  - Just use BFS to compute a shortest path from u to v, then compare the number of edges on the shortest path obtained with k.
  - BFS algorithm runs in polynomial time w.r.t. # vertices (n).
  - Any valid encoding of PATH is $O(n^2)$. Thus BFS decides PATH in polynomial time!
- *Complexity class*: “A set of languages, membership in which is determined by a complexity measure, such as running time, of an algorithm that determines whether a given string x belongs to language L.”

Complexity Class P

- Language-theoretic definition of the complexity class P:

$$P = \{L \subseteq \{0,1\}^*\text{: there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\}.$$

- In other words, P is the class of all decision problems each of which can be solved (decided) by some polynomial-time algorithm.
- Formally stating,

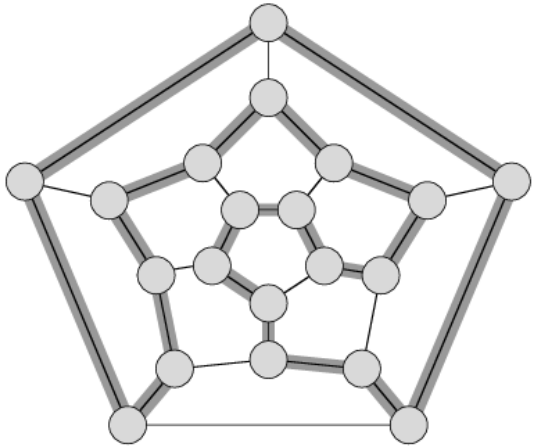
$$P = \{L\text{: } L \text{ is decided by a polynomial-time algorithm}\}.$$

- PATH ∈ P, all the decision problems we've learned so far are in P!

Complexity Class NP

Non-deterministic? Or Polynomial-Time Verifiable?

Some Hard Decision Problems



- For some problems (languages), decision algorithms don’t seem so obvious.
  - E.g., HAM-CYCLE=$\{\langle G\rangle: G$ has a Hamiltonian cycle$\}$
  - A possible decision algorithm would enumerate all permutations of the vertices of G and the check each permutation to see if it’s a Hamiltonian cycle.
  - There are $m!$ possible permutations if $|V(G)| = m = \Omega(\sqrt{n})$. $(n = |\langle G\rangle|)$
  - Checking all permutations will take $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$.
  - This is not $O(n^k)$ for any constant k, thus doesn’t run in polynomial time.
- Turns out there’s no known polynomial-time algorithm for HAM-CYCLE.
- We are characterizing such decision problems as another complexity class called NP.

Polynomial-Time Verification

- Even though it might be difficult to *decide* whether a given problem instance (encoding) is a member (a true instance) of the language (a problem), it might be easier to *verify* if so *with aid of* an evidence (*certificate*).
    - E.g., for HAM-CYCLE, it's not easy to check if for any given G, $\langle G \rangle \in$ HAM-CYCLE.
    - However, if given an ordered list $C = (v_1, v_2, \dots, v_p)$ of vertices of G, it's easy to verify if C is a Hamiltonian cycle of G,
        - which will confirm that $\langle G \rangle \in$ HAM-CYCLE.
        - C is called a *certificate* in that case.

Language Verified By Verification Algorithm

- A verification algorithm A for a decision problem (language)
  - Takes two arguments:
    - A problem instance (candidate member string of the language) x
    - A certificate y
  - Returns true (that is, $A(x, y) = 1$) iff the certificate really proves that the problem instance is a member of the language.
    - By performing algorithmic operations to show that.
- The *language verified* by a verification algorithm A is:

$$L = \{x \in \{0,1\}^*: \text{there exists } y \in \{0,1\}^* \text{ such that } A(x, y) = 1\}.$$

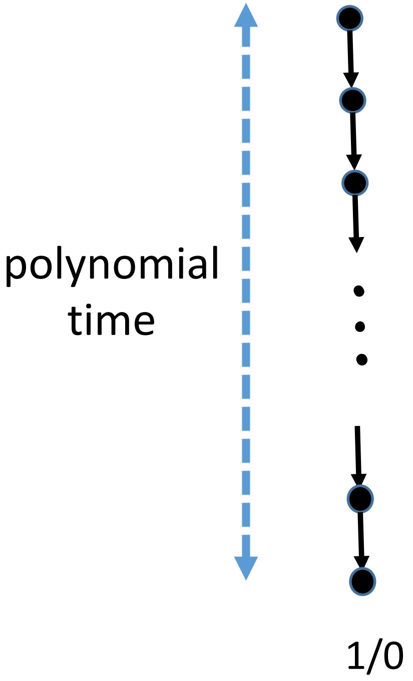
Complexity Class NP

- The class of languages that can be verified by a polynomial-time verification algorithm.
- More precisely, a language L belongs to NP ($L \in NP$) if and only if:
  - There exist a two-input polynomial-time verification algorithm A and a constant c such that:
    - $L = \{x \in \{0,1\}^*$: there exists a certificate y with $|y| = O(|x|^c)$ such that $A(x, y) = 1\}$.
- E.g., HAM-CYCLE $\in NP$.
  - We can build a simple verification algorithm that given $\langle G \rangle$ and $(v_1, v_2, \dots, v_p)$, returns 1 iff $(v_1, v_2, \dots, v_p)$ is a Hamiltonian cycle of G.
    - Just check if every (v_i, v_{i+1}) is an edge of G, if $v_p = v_1$, if all v_i's are distinct, and if $p = |V| + 1$.

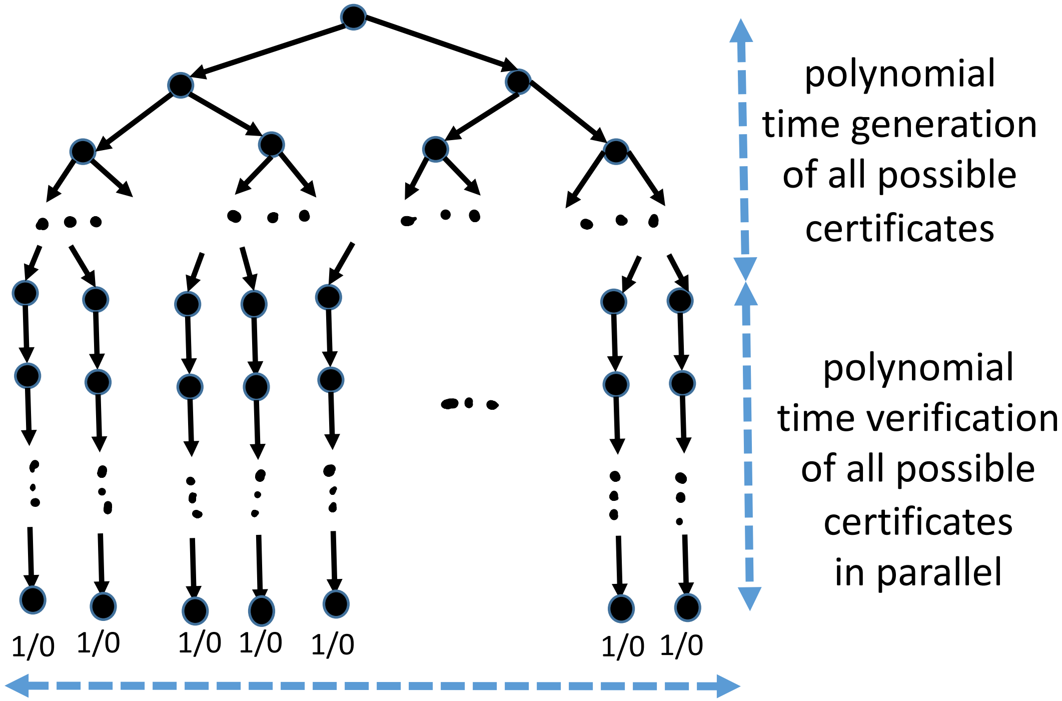
NP: Nondeterministic Polynomial

- Then why named "NP", which means "Nondeterministic Polynomial"?
  - Deterministic: Only one path of computation
  - Non-deterministic: Arbitrarily branching paths of computations
    - Arbitrarily high degree of parallelism allowed (which is impossible in real world)

Deterministic polynomial-time (P)



Non-deterministic polynomial-time (NP)



Possibly exponentially many branches!

P vs NP

- We know for sure that $P \subseteq NP$.
  - That is, if $L \in P$, then $L \in NP$.
  - Why? Just convert the polynomial-time decision algorithm for L to a polynomial-time verification algorithm, which will just do the following:
    - Ignore the certificate input y and run the poly-time decision algorithm with x.
    - And accept (x, y) iff x is accepted by the decision algorithm!
- Then, is $P = NP$ or $P \neq NP$ ($P \subsetneq NP$)?
  - That is, is there a language (problem) L such that $L \in NP$, but $L \notin P$?
    - In other words, is there a problem that can be solved in non-deterministic polynomial time, but not in deterministic polynomial-time?
  - Surprisingly, we don't have a proof of this either way.
    - Even though most people believe that $P \neq NP$ ($P \subsetneq NP$).
    - Proving this (either way) will earn you a million dollar prize. See http://www.claymath.org/millennium-problems/p-vs-np-problem !