CS 121: Introduction to Theoretical Computer Science Fall 2019

Section 6

Prof. Boaz Barak

1 Efficient Algorithms

We have spent much of the semester asking *if* something is computable or not. Now we ask: how long or how many resources does a computation take? One difference we are particular concerned with is the difference between **polynomial** and **exponential** running time, which we will formally define later.

1.1 Min Path/Max Path

Recall that a graph G has a set V of n vertices and a set E of m edges.

- MINPATH: On input a triple (G, s, t), output the number k which is the length of the shortest path between s and t, or -1 if no such path exists. Using BFS, can be computed in $O(n+m)$ time, which is $O(m)$ since $m \geq n-1$. This is **polynomial** time.
- MAXPATH: Given (G, s, t), find the longest non-intersecting path between s and t. The best algorithm takes **exponential** time.

1.2 Min Cut/Max Cut

We define a cut in a graph, $G = (V, E)$ as follows. Given a graph $G = (V, E)$, a *cut* is a subset of S of V such that S is neither empty nor all of V. The edges cut by S are those edges where one of their endpoints is in S and the other is in $\overline{S} = V \setminus S$. If $s, t \in V$, an s, t cut is a cut such that $s \in S, t \in \overline{S}$.

- The MINPATH problem is the problem of finding the minimum s, t cut. More formally: Given (G, s, t), find the minimum number k such that there is an s, t cut cutting k edges (i.e. there exists a set S containing s and not t with exactly k edges that touch S and $\overline{S}$.) There exists an algorithm, the Edmonds-Carp algorithm, that can find the minimum cut between two nodes in $O(n \cdot m^2)$ time: **polynomial** running time.
- The MAXCUT problem however trying to find the maximum k takes **exponential** time, and is the brute-force algorithm that tries all 2^n possibilities for S.

Exercise: How can we use the Edmonds-Carp algorithm as a black-box to find the minimum cut between any nodes in the graph? What is the time complexity?

Solution: There are $\binom{n}{2}$ pairs of nodes in the graph. We could run Edmonds-Carp on all $\frac{n(n-1)}{2}$ pairs, and take the minimum over all pairs. This would have a total time complexity of $O(n^3m^2)$, which is polynomial.

1.3 Existence of a k-clique

Checking for the existence of k-clique is polynomial time for a fixed k but exponential if variable in k. For fixed k, simply check each subgraph with k vertices (there are $O(n^k)$ of these). For each one, check presence of all $O(k^2)$ edges. An exponential problem: find the **maximal clique**.

1.4 2SAT/3SAT

A boolean formula can be written in conjunctive normal form (CNF) as the AND of several ORs of literals x or their negations $\bar{x}$:

$$\Phi = (x_1 \vee x_2) \wedge (x_3 \vee \bar{x_4}) \wedge ...(\bar{x_5} \vee x_6)$$

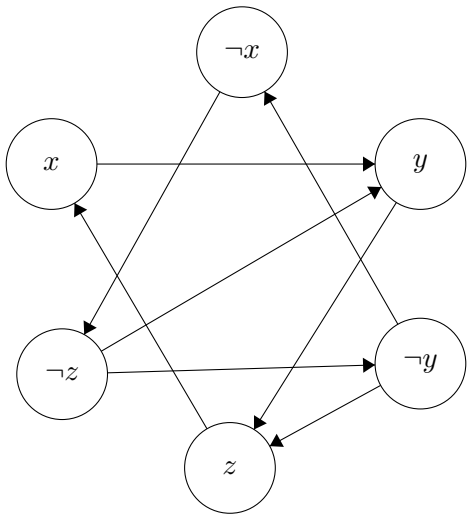
A k-CNF formula is the AND of OR-clauses where each OR contains exactly k literals. The above example is 2-CNF. The 2SAT problem is to find out, given a 2-CNF formula ϕ, whether there is an assignment $x \in \{0,1\}^n$ that *satisfies* ϕ, where by satisfies we means makes ϕ evaluate to 1.

Exercise: Show that 2SAT can be solved in polynomial time by reducing it to a graph connectivity problem for a graph.

*Hint: How would you convert the following formula Φ to an **implication graph** G? Once the graph is constructed, which edges in G should be checked in order to answer whether Φ is satisfiable? Is Φ satisfiable?*

$$\Phi = (\bar{x} \vee y) \wedge (\bar{y} \vee z) \wedge (x \vee \bar{z}) \wedge (z \vee y)$$

Solution: Response to the Hint: Φ is satisfiable. The implication graph for it is below:



Our algorithm is the following:

Given a 2CNF formula ϕ:

1. Create a graph G with $2n$ vertices, two (a true and not true literal) for each variable. For each clause $(a \vee b)$ in ϕ, create a directed edge from $\bar{a}$ to b and from $\bar{b}$ to a.
2. For all $x \in n$ variables:
   (a) Check if there is a path from x to $\bar{x}$ in G.
   (b) Check if there is a path from $\bar{x}$ to x.
   (c) If (a) and (b) are true, return `false`.

   Return `true`.

Analysis of Algorithm Correctness:

1. A clause $(a \vee b)$ will evaluate to 1 if and only if at least one of a, b is true, which is the same as saying if $\bar{a}$ then b, and if $\bar{b}$ then a. This is exactly what our constructed implication graph requires.
2. Claim: A 2CNF formula is unsatisfiable iff there is a variable x_i such that there is a path from $x \to \bar{x}$ and a path from $\bar{x}$ to x. In other words, we want to make sure that the formula does not require that a variable be negated in one clause and not negated in another clause. (This can be shown through proof by contradiction).

Analysis of Time:

1. The conversion process takes polynomial time (more specifically linear time).
2. Checking for existence of a path between vertices s and t (Graph Connectivity) itself is poly. time. Use depth-first search starting from s and mark visited edges. After at most $\binom{n}{2}$ steps, all edges are either visited or will never be visited.

Overall, this process takes polynomial time.

1.5 Relatively Prime: Polynomial Algorithm (Sipser)

Two numbers are **relatively prime** if 1 is the largest integer that evenly divides them both. For example 10 and 21 are relatively prime, but neither are prime on their own. 10 and 22 are not relatively prime because both are divisible by 2.

$$RELPRIME = \{(x, y) | x \text{ and } y \text{ are relatively prime}\}$$

Let E be the following subroutine:

1. on input of two natural numbers x and y:

2. Repeat until $y = 0$:
    - assign $x \leftarrow x$ -mod y
    - Exchange x and y
3. Output x

Now define the polynomial time algorithm R that uses E:

1. Run E on x, y
2. If the result is 1, accept. Otherwise reject.

Exercise: Show that R is polynomial time.

Solution: Clearly, if E runs correctly in poly. time, then so does R. Analyze the running time of E to show that E, thus R, is polynomial. To analyze the time complexity of E we first show that every execution of stage 2 (except possibly the first) cuts the value of x by at least half. After stage 2 is executed $x < y$ because the nature of the mod function. After stage 3, $x > y$ because the two have been exchanged. Thus, when stage 2 is subsequently executed, $x > y$. If $x/2 \geq y$, then x mod $y < y \leq x/2$ and x drops by at least half. If $x/2 < y$, then $xmody = x - y < x/2$ and x drops by at least half. The values of x and y are exchanged every time stage 3 is executed, so each of the original values of x and y reduced by at least half every other time through the loop. Thus the max number of times that stages 2 and 3 are executed is the lesser of $2\log_2 x$ and $2\log_2 y$. These logs are proportional to the lengths of the representations of x and y, giving the number of stages executed as $O(n)$. Each stage of E only uses poly time. so the total running time is poly time.

2 Modeling Running Time

Goals: Formally modeling running time with big O, the classes **P** and **EXP**, and the time hierarchy theorem.

2.1 Definitions of Running Time

Let $T : \mathbb{N} \to \mathbb{N}$ be some function. We say that a function $F : \{0,1\}^* \to \{0,1\}$ is computable in $T(n)$ TM time if there is a Turing Machine P computing F such that for every sufficiently large n and every $x \in \{0,1\}^n$, on input x, P runs for at most $T(n)$ steps.

Similarly, a function F is computable in $T(n)$ NAND-RAM time if there is a NAND-RAM program P computing F such that for sufficiently large n and every $x \in \{0,1\}^n$, P executes as most $T(n)$ lines.

2.2 Choice of Model

We let $TIME_{\mathsf{TM}}(T(n))$ denote the set of Boolean functions that are computable in $T(n)$ TM time.

Note that if you only care about 'coarse enough' resolution (polynomial vs exponential time, as defined below), the choice of computational model (as long as it's 'reasonable') actually does not matter by the extended Church-Turing Thesis. In fact, the lecture notes show a proof that NAND-RAM is efficiently simulated with Turing Machines, by which we mean the overhead simulation of a NAND-RAM program using a Turing Machine is at most polynomial. Similar theorems exist for other models.

Therefore, we can also define $TIME_{\mathsf{RAM}}(T(n))$ as the set of Boolean functions that are computable in $T(n)$ NAND-RAM time.

2.3 Definitions of Time Classes

Polynomial Time A function $F : \{0,1\}^* \to \{0,1\}$ is computable in polynomial time if it is in the class $\mathbf{P} = \bigcup_{c\in\mathbb{N}} TIME(n^c)$

Exponential Time A function $F : \{0,1\}^* \to \{0,1\}$ is computable in exponential time if it is in the class $\mathbf{EXP} = \bigcup_{c\in\mathbb{N}} TIME(2^{n^c})$. Such a function is considered **intractable**.

Exponential time is much larger than polynomial time, so $\mathbf{P} \subset \mathbf{EXP}$. The table below lists some problems that we have been able to show are in $\mathbf{P}$, and others that there we have not been able to show are in $\mathbf{P}$.

P	EXP
Shortest path	Longest Path
Min cut	Max cut
2SAT	3SAT
Linear eqs	Quad. eqs
Zerosum	Nash
Determinant	Permanent
Primality	Factoring

Exercise: Prove that if $F, G : \{0,1\}^* \to \{0,1\}^*$ are in $\mathbf{P}$ then their composition $F \circ G$, which is the function H s.t. $H(x) = F(G(x))$, is also in $\mathbf{P}$.

Solution: Lemma: First, note that the composition of two polynomial functions is polynomial: the composition of two functions f and g with degrees d_1 and d_2 respectively can have a maximum degree of $d_1 * d_2$, which is still polynomial.

For the main proof, note that if F and G are in $\mathbf{P}$, then there must be NAND-RAM programs P_F and P_G that compute F and G respectively that run for $O(n^{k_1})$ and $O(n^{k_2})$ steps of NAND-RAM respectively, where n is the length of the input.

By the sequential composition theorem, we can construct a program P_C that computes $F(G(x))$ in the following way: Start with the program P_G, and compute $G(x)$. Then "paste in" the code for P_F, changing the input to P_F to the output $G(x)$ from P_G. Note that because F is bounded by $O(n^k)$ NAND-RAM steps, it can be written as a NAND program of $O(poly(n^{k_1}))$ lines, which means the size of the output of P_G can be at most $poly(n^{k_2})$. By our lemma, this upper bound on the size of the output is still polynomial, so let the size of the output of the polynomial be $O(n^c)$ for some constant c. This means that the length of the input to the code for P_F is at most $O(n^c)$, so the overall runtime is $O((n^c)^k) = O(n^{ck})$, which is still polynomial.

2.4 Time Hierarchy Theorem

For every nice (as defined in lecture notes) function T, there is a function $F : \{0,1\}^* \to \{0,1\}$ in $TIME(T(n)\log n) \setminus TIME(T(n))$.

The implication of this theorem is that there exists a function that can be computed in time $O(n^2)$ but not $O(n)$, that can be computed in time $O(2^n)$ but not $O(2^{0.9n})$, etc. It also proves that $\mathbf{P} \neq \mathbf{EXP}$.