CSE 3500 Algorithms and Complexity – Fall 2016
Lecture 25: November 29, 2016

Intractable Problems

- There are many problems for which the best known algorithms take a very long time (e.g., exponential in some of the input parameters). No one has been able to prove that these problems are difficult and need this much time. Thus there is a gap in our understanding of these problems. We refer to these problems as *intractable problems*. Two examples are: Satisfiability (SAT) and Clique.
- In the last lecture we showed that SAT can be solved in $O(2^n|F|)$ time on any CNF Boolean forumla F on n variables. We also proved that we can solve the clique problem in $O(\binom{n}{k}k^2)$ time on a graph with n nodes, k being the target clique size.
- These intractable problems have some properties in common. We'll identify them next.

Nondeterminstic Algorithms

- A *decision problem* is one that has only two possible answers, namely, *yes* and *no*. For any problem, we can think of a decision version and an optimization version of it. For example, for the clique problem a decision version will have a graph $G(V, E)$ and an integer k $(1 \leq k \leq |V|)$ as input. The problem is to check if G has a clique of size k. An optimization version will have a graph $G(V, E)$ as the input and the problem is to identify the largest integer k $(1 \leq k \leq |V|)$ such that G has a clique of size k. If we have an efficient algorithm for the decision version of a problem, we might also be able to get an efficient algorithm for the optimization version. In our discussion of intractable problems, we consider only decision problems.
- A nondeterministic algorithm uses the same set of basic operations as a deterministic algorithm except that it has access to another operation called Choice. Choice takes as input a set S and returns an element of S. There is no rule that specifies how this choice has to be made. Whenever there is a set of choices that leads to a successful completion of the algorithm, then one such set of choices is always made.
- There are two phases in any nondeterministic algorithm. In the first phase Choice is used (possibly multiple times) to guess a solution to the problem at hand. The second phase is deterministic and is used to check the correctness of the guesses made in the first phase.
- Here is a nondeterministic algorithm for SAT: (Let F be the input Boolean CNF formula on n variables $x_1, x_2, \ldots, x_n$).

Phase 1

  for $i = 1$ **to** n **do**

    x_i =Choice$\{T, F\}$;

Phase 2

  Check if the assignment guessed in Phase 1 satisfies F;

Run time analysis: Phase 2 takes $O(|F|)$ time. In Phase 1, n calls are made to Choice and each such call takes one unit of time. Thus the total run time of the algorithm is $O(n + |F|)$. This is indeed a linear time algorithm!

- Here is a nondeterministic algorithm for solving the clique problem. Let the input be the graph $G(V, E)$ and the integer k:

Phase 1

  $S = \emptyset$;

  for $i = 1$ **to** k **do**

    x =Choice(V); $S = S \cup \{x\}$;

Phase 2

  Check if the k nodes in S form a clique;

Run time analysis: If we assume that the graph is in adjacency matrix form, then Phase 2 can be completed in $O(k^2)$ time. Phase 1 takes k steps. Thus the total run time is $O(k^2)$.

The complexity classes $\mathcal{P}$ and $\mathcal{NP}$

- By a *polynomial time* we mean a run time of $O(n^c)$ where c is any constant.
- The complexity class $\mathcal{P}$ is the collection of all the problems that can be solved in deterministic polynomial time. Example problems in $\mathcal{P}$: sorting, MST, matrix multiplication, etc.
- The complexity class $\mathcal{NP}$ is the collection of all the problems that can be solved in nondterministic polynomial time. Example problems in $\mathcal{NP}$: SAT, Clique, traveling salesperson problem, etc.
- One of the long-time unanswered questions is: "Is $\mathcal{P} = \mathcal{NP}$?"
- If a problem is in $\mathcal{P}$, then, trivially, it is also in $\mathcal{NP}$. Thus we infer: $\mathcal{P} \subseteq \mathcal{NP}$.

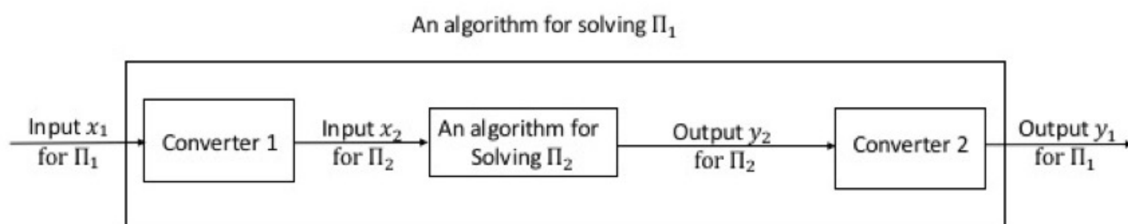
Polynomial Time Reductions

- We say that a problem π_1 reduces to another problem π_2 if we can solve the problem π_1 using an algorithm known for solving π_2.

- For example, if π_1 is selection and π_2 is sorting, then π_1 reduces to π_2.

- If the problem π_1 reduces to the problem π_2, then it should be possible to convert any input x_1 for π_1 into a corresponding input x_2 for π_2. We can then solve π_2 on x_2 to get the answer y_2 (for π_2). y_2 is then converted into a relevant output y_1 for π_1.

- The steps involved in reducing π_1 to π_2 are summarized in the following figure:



Reducibility

- We say that a problem π_1 *polynomially reduces* to the problem π_2 if π_1 reduces to π_2 and Converter 1 and Converter 2 take a (deterministic) polynomial time each. We use the following notation: $\pi_1 \propto \pi_2$.

$\mathcal{NP}$-hard and $\mathcal{NP}$-complete Problems

- A problem π is said to be $\mathcal{NP}$-hard if the problem π' polynomially reduces to π for every $\pi' \in \mathcal{NP}$.

- The problem π is said to be $\mathcal{NP}$-complete is π is $\mathcal{NP}$-hard and $\pi \in \mathcal{NP}$.

- **Cook's Theorem:** Cook showed that SAT is $\mathcal{NP}$-complete. He showed that a Boolean formula in CNF can be constructed in polynomial time such that it simulates each and every step of a non-deterministic machine that runs for a polynomial amount of time. If the machine solves a problem π, then the Boolean formula will have a satisfying assignment if and only if the machine terminates with the answer *yes*.

- **An equivalent definition for $\mathcal{NP}$-complete problems:** A problem π is $\mathcal{NP}$-comlete if $\pi \in \mathcal{NP}$ and $\pi' \propto \pi$, for some problem π' that is known to be $\mathcal{NP}$-complete. This is the definition that is typically used to show $\mathcal{NP}$-completeness of problems.

- **Lemma:** If the problem π is $\mathcal{NP}$-complete and if $\pi \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

  Proof: Let π' be any problem in $\mathcal{NP}$. If π is $\mathcal{NP}$-complete and $\pi \in \mathcal{P}$, then we will show that $\pi' \in \mathcal{P}$ that will imply that $\mathcal{P} = \mathcal{NP}$.

  Since π is $\mathcal{NP}$-complete, it follows that $\pi' \propto \pi$ for any $\pi' \in \mathcal{NP}$. Let x' be any input for π'. It follows that we can convert x' into a corresponding input x for π in deterministic polynomial time. Let this polynomial be $q(.)$. Let y be the answer of π on x. It also follows that y can be converted into a corresponding output y' for π' in deterministic polynomial time. Let this polynomial be $r(.)$ Since $\pi \in \mathcal{P}$, we can solve π on x in a determinstic polynomial time. Let this polynomial be $p(.)$.

  Let the size of the input x' be n. Then we realize that the size of x cannot be more than $q(n)$ and that the size of y cannot be more than $p(q(n))$. As a result, it follows that π' can be solved deterministically in time $q(n) + p(q(n)) + r(p(q(n)))$ which is a constant degree polynomial in n. Specifically, if the degrees of the polynomials $q(.), p(.)$, and $r(.)$ are d_1, d_2, and d_3, respectively, then the above run time is a polynomial of degree $d_1 d_2 d_3$. $\square$

- Along the same lines we can also prove the following Lemma.

  Lemma: If $\pi_1 \propto \pi_2$ and $\pi_2 \propto \pi_3$, then, $\pi_1 \propto \pi_3$.

$\mathcal{NP}$-completeness Proofs

- There are two steps involved in proving that a problem π is $\mathcal{NP}$-complete: 1) show that $\pi \in \mathcal{NP}$ – this is often easy; and 2) Show that $\pi' \propto \pi$ for some known $\mathcal{NP}$-complete problem π'.

- To show that $\pi_1 \propto \pi_2$ there are three steps: 1) Present an algorithm for Converter 1; 2) Show that this algorithm takes (deterministic) polynomial time; and 3) Prove that this reduction is correct. This involves two substeps: a) Prove that if the answer of π_1 on x_1 is *yes* then the answer of π_2 on x_2 is also *yes* and b) Show that if the answer of π_1 on x_1 is *no* then the answer of π_2 on x_2 is also *no*.

Clique is $\mathcal{NP}$-complete

- Now we will show that Clique is $\mathcal{NP}$-complete. We have already shown that Clique is a member of $\mathcal{NP}$.

- We will show that SAT $\propto$ Clique.

- Let $F = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be any Boolean formula in CNF. Here $C_1, C_2, \ldots, C_k$ are clauses, where each clause if a disjunction of literals. (A literal is either a variable or its negation). The input for the Clique problem will be a graph and an integer.

- We generate the following input for Clique: $G(V, E); k$, where there is a node in G for every literal in every clause of F. If x_q is a literal in C_i, then the node corresponding to this literal is denoted as (x_q, i).
- Two nodes (x_q, i) and (x_r, j) will be connected by an edge if and only if $i \neq j$ and $x_q \neq \bar{x}_r$.
- **An Example:** Let $F = (x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3 \vee \bar{x}_1)$. In this case the input for the Clique will be $G(V, E); 2$, where $V = \{(x_1, 1), (\bar{x}_2, 1), (x_2, 2), (x_3, 2), (\bar{x}_1, 2)\}$ and $E = \{((x_1, 1), (x_2, 2)),\ ((x_1, 1), (x_3, 2)),\ ((\bar{x}_2, 1), (x_3, 2)),\ ((\bar{x}_2, 1), (\bar{x}_1, 2))\}$.
- The proof will be completed in the next lecture.