Lecture notes for Jan 18, 2023
Introduction and NP

Chun-Hung Liu

January 18, 2023

1 Introduction

In this course, a *graph* G is a pair $(V(G), E(G))$, where $V(G)$ is a set and $E(G)$ is a multiset of subsets of $V(G)$ of size at most two. Such a graph is also called a "multigraph" in some literature. $V(G)$ is called the *vertex-set* of G, and $E(G)$ is called the *edge-set* of G. A graph G is *simple* if $E(G)$ is a set (but not a multiset) and every member of $E(G)$ has size exactly two.

Sometimes we consider directed graphs. A *directed graph* (or *digraph* for short) D is a pair $(V(D), E(D))$, where $V(D)$ is a set and $E(D)$ is a multiset of an ordered pair (x, y) for some $x, y \in V(G)$. Again $V(D)$ is called the *vertex-set* of D, and $E(D)$ is called the *edge-set* of D. Each element of $E(D)$ is called an *edge* or an *arc* of D. For an arc (x, y), x is called the *tail* and y is called the *head* of this arc. Note that we do not require $x \neq y$.

We mostly consider algorithmic problems on graphs in this course. More precisely, we want to design efficient procedures (i.e. algorithms) to solve the following kinds of problems about graphs or digraphs, or to decide the existence of an efficient algorithm:

- *Decision problem*: Questions for which the answer is "yes" or "no", but not both. For example:
  - Given a graph G, does a graph G have more than 100 vertices?
  - Given a graph G and an integer k, does there exist a set of k edges such that any two edges in this set do not share an end? (Such a set is called a *matching* in G.)

  - Given a graph G, is G connected?

- *Optimization problem*: Questions for which the answer is a number, and this answer is a maximum or a minimum for certain properties. For example:

  - Given a graph G, what is the maximum size of a set of pairwise non-adjacent vertices in G. (Such a set is called a *stable set* in G.)
  - Given a graph G, what is the minimum size of a set S of vertices such that every edge of G has at least one end in S. (Such a set is called a *vertex-cover* in G.)

By "efficient algorithms", we mean an algorithm that solves the problem in polynomial time deterministicly and correctly; by "polynomial time", we mean the running time is polynomial in the size of the input. For most of cases, the input includes a graph G, and the size of G is consider to be $|V(G)| + |E(G)|$. So linear time algorithms run in time $O(|V(G)| + |E(G)|)$. Note that seeing all vertices and edges of G already take linear time, so linear time algorithm is the best that we can expect for deterministic algorithms.

We probably will consider randomized algorithms, for which we only require the algorithm solves the problem correctly with high probability or runs in polynomial time with high probability. We will also consider approximation algorithms, for which we output a "good approximation" for an optimization problem in polynomial time deterministicly or probabilistically. For randomized algorithms and approximation algorithms, sublinear time (or even constant time) algorithms are possible.

1.1 P v.s. NP

For decision problems, our concern is to design a polynomial time algorithm to correctly answer "yes" or "no", or to prove that a deterministic polynomial time algorithm "unlikely" exists for a mathematical reason. For the former, we usually also want to output a "certificate" to support our yes/no answer. For the latter, the main goal is to prove that the problem is NP-hard.

Now we give formal definitions that are enough for this course. For a decision problem, an input is a *positive instance* if the correct answer for this input is "yes"; otherwise it is a *negative instance*. We say that a decision problem is *in* **P** if there exists a deterministic polynomial time algorithm

that correctly answers yes for any positive instance and answers no for any negative instance. (This definition for **P** is slightly different from one defined in complexity theory, in which every decision problem is "encoded" as the set of all its positive instances, and **P** is defined to be the set of all decision problems they can be determined by a deterministic Turing machine in polynomial time. But our definitions are essentially the same, except we do not want to address the encoding and formal language in very detail.)

A decision problem D is in *in* **NP** if every positive instance has a "certificate" with polynomial size so that the positivity can be verified in polynomial time; more formally, there exist polynomials p and q and an algorithm A such that

- the input of A is of the form (x, y), and A runs in time $p(|x| + |y|)$,
- if I is a positive instance of D, then there exists a bit-string C_I with size $q(|I|)$ such that A answers "yes" when the input of A is (I, C_I), and
- if I is a negative instance of D, then for every bit-string s with size $q(|I|)$, A answers "no" when the input of A in (I, s).

(Our definition for **NP** is essentially equivalent to the one in complexity theory, which says that every instance can be correctly decided by a non-deterministic Turing machine. The "N" in **NP** corresponds to "non-deterministic Turning machines".)

Examples: The following problems are in **NP**:

1. The CLIQUE problem is: Given a graph G and a positive integer k, determine whether G has a clique of size at least k.
   - Recall that a *clique* in a graph G is a subset S of $V(G)$ such that any two vertices in S are adjacent in G.
   - Every instance of this problem is a pair (G, k), where G is a graph and k is a positive integer. And (G, k) is a positive instance if and only if G has a clique $S_{G,k}$ of size k. Note that we can encode a clique into a bit string.
   - Consider the algorithm A with input (x, y), where x is an instance of CLIQUE and can be written as (G, k), that decides whether y

is a clique of size k in G by verifying whether y gives a set of k vertices and verifying whether vertices in this set are pairwise adjacent. Clearly A runs in time in time $O(k^2)$.

- So for every positive instance (G, k) of CLIQUE, A answers "yes" when the input of A is $((G, k), S_{G,k})$. For every negative instance (G, k) of CLIQUE, a clique with size k in G does not exist, so A answers "no" when the input of A is $((G, k), s)$ for any string s.
- Hence CLIQUE is in **NP**. And $S_{G,k}$ is a certificate for a positive instance (G, k) of CLIQUE.

2. The CONNECTIVITY problem is: Given a graph G, determine whether G is connected.

   - Recall that a graph G is *connected* if for any two vertices u, v, there exists a path in G between u and v.
   - Note that a graph G is connected if and only if there exists a spanning tree T_G of G. (Recall that a *spanning subgraph* H of a graph G is a subgraph of G with $V(H) = V(G)$. A *spanning tree* of G is a tree and is a spanning subgraph of G.)
   - Let A be the algorithm with input (G, y), where G is a graph and y is a bit-string, such that it decides whether y is a connected spanning subgraph of G. Note that it can be done in time $O(|V(G)| + |E(G)|)$ by using the breadth-first-search or depth-first-search that we will discuss later.
   - So for every positive instance G of CONNECTIVITY, A answers "yes" when the input of A is (G, T_G). And for every negative instance G of CONNECTIVITY, there is no connected spanning subgraph of G, so A answers "no" when the input of A is (G, s) for any string s.
   - Hence CONNECTIVITY is in **NP**. And T_G is a certificate for a positive instance G of CONNECTIVITY.

Remark:

1. Every decision problem D in **P** is also in **NP**, since we can use the algorithm that solves D as the verification algorithm. So $\mathbf{P} \subseteq \mathbf{NP}$.

The problem whether $\mathbf{P} =? \mathbf{NP}$ is one of the major open problems in mathematics and computer science.

2. To prove that a problem D is in **NP**, we only need the existence of the certificate instead of a polynomial time algorithm to find the certificate. For example, we do not know how to find the clique $S_{G,k}$ in the CLIQUE problem in polynomial time.

3. For a problem D in **NP**, we know every positive instance has a certificate, but we do not know whether every negative instance has a certificate so that we can verify the negativity in polynomial time.

4. **co-NP** consists of the decision problems whose each negative instance has a certificate with polynomial size that can be verified in polynomial time. The problem whether $\mathbf{NP} =?$ **co-NP** is another major open problem.

5. Again, every decision problem D in **P** is also in **co-NP** since we can use the algorithm that solves D as the verification algorithm. So $\mathbf{P} \subseteq \mathbf{NP} \cap$ **co-NP**. The problem whether $\mathbf{P} =? \mathbf{NP} \cap$ **co-NP** is another major open problem.

1.2 NP-completeness

As we mentioned above, $\mathbf{P} =? \mathbf{NP}$ is one of the major open problems. How can we solve it? If we can solve the "hardest" problem in **NP** in polynomial time, then $\mathbf{P} = \mathbf{NP}$; if we can prove that the "hardest" problem in **NP** cannot be solved in polynomial time, then $\mathbf{P} \neq \mathbf{NP}$. So it is sufficient to study such a "hardest" problem.

A decision problem D is **NP**-*hard* if for every problem D' in **NP**, there exists a polynomial time algorithm $A_{D'}$ that given an instance I' of D', produces an instance $f(I')$ of D such that I' is a positive instance for D' if and only if $f(I')$ is a positive instance for D. This implies that every **NP**-hard problem D is (not necessarily strictly) harder than all problems in **NP**, because if we have a polynomial time algorithm A to solve D, then for any problem D' in **NP**, we can combine the algorithm $A_{D'}$ and A to obtain a polynomial time algorithm to solve D'. Such an algorithm $A_{D'}$ is called a *polynomial time reduction* from D' to D.

In fact, we can define **NP**-hardness for non-decision problems in a similar way. An algorithmic problem D is **NP**-*hard* if for every problem D' in **NP**, there exists a polynomial time algorithm $A_{D'}$ that given an instance I' of D', produces an instance $f(I')$ of D such that the answer for $f(I')$ gives the answer for I in polynomial time.

A decision problem is **NP**-*complete* if it is in **NP** and is **NP**-hard. That is, every **NP**-complete problem is a (not necessarily strictly) hardest problem in **NP**.

Now we describe an NP-complete problem. We need some definitions in logic. A *formula* (with variables $x_1, x_2, ..., x_n$) is a function $\phi(x_1, x_2, ..., x_n) : \{\text{“True”}, \text{“False”}\}^n \to \{\text{“True”}, \text{“False”}\}$, and it consists of the variables and the following three logical operations: “negation” $\neg$, “or” $\vee$ and “and” $\wedge$. For example, $(x_1 \wedge x_2) \vee \neg x_1$ is a formula with variables x_1, x_2. A formula with variables $x_1, ..., x_n$ is in *conjunctive normal form* if it can be written as $c_1 \wedge c_2 \wedge ... \wedge c_m$ for some positive integer m, where each c_i is called a *clause* and is of the form $(a_{i,1} \vee a_{i,2} \vee ... \vee a_{i,k_i})$ for some positive integer k_i, and each $a_{i,j}$ is called a *literal* and is either x_k or $\neg x_k$ for some $1 \leq k \leq n$. For example, $(x_1 \wedge x_2) \vee \neg x_1$ is not in conjunctive normal form, but its logically equivalent formula $(x_1 \vee \neg x_1) \wedge (x_2 \vee \neg x_1)$ is. A *truth assignment* for a formula $\phi(x_1, ..., x_n)$ is a choice of each x_i so that $\phi(x_1, ..., x_n)$ is True. For example, choosing $x_1 =$ False and $x_2 =$ True is a truth assignment for the formula $(x_1 \vee \neg x_1) \wedge (x_2 \vee \neg x_1)$. We say that a formula is *satisfiable* if it has a truth assignment.

SAT is the following decision problem:

SAT
Input: A formula ϕ.
Output: Determine whether ϕ is satisfiable.

It is easy to see that SAT is in NP since a truth assignment of a satisfiable formula has polynomial size and can be verified in polynomial time. In fact, SAT is the “hardest” problem in NP.

Theorem 1 (Cook) *SAT is* **NP**-*complete.*