21120491 Advanced Data Structure and Algorithm Analysis **Spring-Summer 2023**

Lecture 10: NP-Completeness

Lecturer: Deshi Ye *Scribe: D. Ye*

10.1 Overview

When we face a new problem, we would like to know how hard it is to solve it. At the extreme are undecidable problems that cannot be solved by a computer in any finite amount of time (not even in exponential time, not even in doubly-exponential time, and so on). One famous example of an undecidable problem is the halting problem: Given a program, determine whether it goes into an infinite loop or eventually halts.

In this lecture, we only focus on the problem that can be computed. A computational problem is a problem that can be solved step-by-step with a computer in a finite amount of time. These problems usually have well-defined input, constraints, and conditions that the output must be satisfied. Usually, there are four types of computational problems:

- Decision problem. Output "yes" if there is a feasible solution and "no" otherwise.
- Search problem. Output a feasible solution if one exists, and "no solution" otherwise.
- Optimization problem. Output a feasible solution with the best-possible objective function value (or "no solution," if none exists).
- Counting problem. Output the number of solutions to a search problem.

Given a computational problem X, and a machine model M, how long does it take to solve X using a machine from M?

Fact: The decision version of a problem is easier than (or the same as) the optimization version. If you could solve the optimization version and got a solution of value M, then you could just check to see if M is "yes" or "no". Thus, if you can solve the optimization problem, you can solve the decision problem. On the other hand, if the decision problem is hard, then so is the optimization version.

Fact: For each optimization problem X, there is a decision version X' of the problem by setting a bound.

When we define the P and NP, we only consider decision problems.

What is a "fast algorithm"? A "fast algorithm" is an algorithm whose worst-case running time grows slowly with the input size. And what do we mean by "grows slowly"?

Easy and Hard Problems Given a computational problem, is it easy or hard to compute?

- Easy problem: it can be solved with a polynomial-time algorithm.

- Hard problem: it requires exponential time in the worst case.

The input of a problem will be **encoded** as a binary string. Example, the input for sorting is $(3, 6, 100, 9, 60)$, which will be encoded as the binary $(11, 110, 1100100, 1001, 111100)$. The ASCII code can also be encoded as a binary string, for example, the encoding of A is 1000001. We can encode a compound object as a binary string by combining the representations of its constituent parts. Polygons, graphs, functions, ordered pairs, programs—all can be encoded as binary strings.

Definition 10.1 (Input Size): *The size of an input is the length of the encoded string s for the input, denoted as $|s|$.*

To encode an integer x, we need at most of $\lceil \log x \rceil + 1$ bits.

Definition 10.2 *A polynomial-time algorithm is an algorithm with a worst-case running time $O(n^k)$, where n denotes the input size and k is a constant (independent of n).*

Definition 10.3 (Polynomial-Time Solvable Problems): *A computational problem is polynomial-time solvable if there is a polynomial-time algorithm that solves it correctly for every input.*

10.2 Machine: Model of Computation

Many models of computation have been proposed, while the fundamental one is the Turing machine.

10.2.1 Turing Machine

A **Turing machine** (TM for short) consists of one or more tapes, which we can think of as storing an infinite (in both directions) array of symbols from some alphabet $\sum$, one or more heads, which point to particular locations on the tape(s), and a **finite-state** control that controls the movement of the heads on the tapes and that may direct each head to rewrite the symbol in the cell it currently points to, based on the current symbols under the heads and its state, an element of its state space Q.

In its simplest form, a Turing machine has exactly one tape that is used for input, computation, and output, and has only one head on this tape. This is often too restrictive to program easily, so we will typically assume at least three tapes (with corresponding heads): one each for input, work, and output. This does not add any significant power to the model, and indeed not only is it possible for a one-tape Turing machine to simulate a k-tape Turing machine for any fixed k, but it can also do so with only polynomial slowdown. Similarly, even though in principle we can limit our alphabet to just $\{0, 1\}$, we will in general assume whatever (finite) alphabet is convenient for the tape cells.

Fig. 10.1 show a deterministic 3-tapes Turing machine.

10.2.2 Random Access Machine

Definition 10.4 (Random Access Machine (RAM)) :

- *Has a controller with registers and a memory*

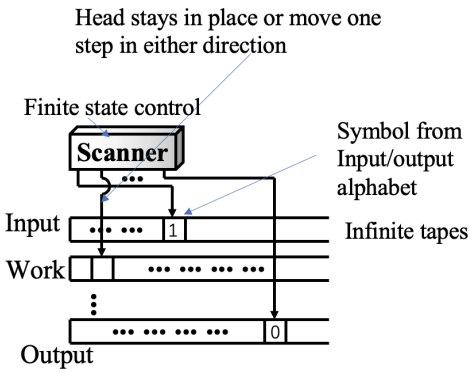

Figure 10.1: A deterministic 3-tapes Turing machine

- *Each register and memory location holds an integer*
- *Do arithmetic on registers, compare registers, and load or store a value from a memory location addressed by a particular register all as a single step (i.e., 1 time step).*
- *Loops and subroutines are not considered simple operations. Instead, they are the composition of many single-step operations.*

The running time of an algorithm (time complexity) is based on RAM.

Theorem 10.5 [The extended Church-Turing thesis]*: Any language that can be decided by a physically realizable computing device M in time $T(n)$ can be decided by a Turing machine in time $O(T(n)^k)$ for some fixed k that depends only on M.*

What is a physically realizable computing device? Today's digital computers can be regarded as implementations of this model. The extended Church-Turing thesis says that this simulation between computational models involves only polynomial slowdown. So things like $O(n)$ or $O(n^2)$ time may depend on our choice of the computational model. But if we just talk about polynomial time, this is robust against changes in the model.

Given any computer algorithm, a Turing machine capable of simulating that algorithm's logic can be constructed.

10.2.3 Language (equivalently decision problem)

We denote by $\{0,1\}^*$ the set of all strings composed of symbols from the set $\{0,1\}$.

A language L over $\sum$ is any set of strings made up of symbols from $\sum$ ($\sum = \{0,1\}$ for example).

A language is the formal realization of a problem.

Decide If we have a Turing machine M that halts in an accepting state on any input $x \in L$ and halts in a rejecting state on any input $x \notin L$, we say that M decides language L.

Algorithm A **accepts** a string $x \in 0, 1^*$ if $A(x) = 1$. Algorithm A **rejects** a string x if $A(x) = 0$.

A language L is decided by an algorithm A if every binary string in L is accepted by A and every binary string **not in L** is rejected by A.

Decision problem or language X:

- Problem X is a set of strings.
- Instance s is one string.
- Algorithm A decides problem X: $A(s) =$ yes iff $s \in X$.

To accept a language L, an algorithm need only worry about strings in L, but to decide a language, it must correctly accept or reject every string in $\{0, 1\}^*$.

10.2.4 Nondeterministic Turing Machine

Historically, the only difference between an NDTM and a standard TM is that an NDTM has two transition functions δ_0 and δ_1, and the NDTM can choose between them. We give it the magical ability to split into two copies, each with a different bit telling it what to do next. These copies can then split further into exponentially many copies or branches. If any of the branches accept, then we say the machine as a whole accepts; if none do, the machine rejects. A Nondeterministic Turing Machine is free to choose its next step from a finite set. And if one of these steps leads to a solution, it will always choose the correct one.

Nondeterminism is now typically represented by giving a machine an extra input, the **certificate** or **witness**. The witness provides the sequence of choices that lead to the accepting branch (if there is one).

A language L is decided by a **nondeterministic** machine M if, for every $x \in L$, there exists a witness w such that $M(x, w)$ accepts, and for every $x \notin L$, there does not exist a witness w such that $M(x, w)$ accepts.

A verification algorithm $A(x, y)$ is a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a certificate.

A two-argument algorithm A verifies an input string x if there exists a certificate y such that $A(x, y) = 1$.

The language L verified by a verification algorithm A is $L = \{x \in \{0, 1\}^* : \exists y \in \{0, 1\}^* \textit{such that } A(x, y) = 1\}$.

10.3 Complexity class

A problem belongs to a complexity class. In the following, informally, we define three complexity classes.

- **P** is the set of decision problems that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved quickly.
- **NP** is the set of decision problems with the following property: If the answer is "Yes", then there is a proof of this fact that can be checked in polynomial time. Intuitively, NP is the set of decision problems where we can verify a "Yes" answer quickly if we have the solution in front of us.
- **co-NP** is essentially the opposite of NP. If the answer to a problem in co-NP is "No", then there is a proof of this fact that can be checked in polynomial time.

For a problem X, its complement $\bar{X}$ is the problem such that $s \in \bar{X}$ if and only if $s \notin X$.

Co-NP is the set of decision problems X such that $\bar{X} \in NP$. Examples of co-NP are UnSAT, No-Hamiltonian-Cycle, No-3-Colorable, and Prime.

Definition 10.6 (The class P) *$P = \{L \subseteq \{0,1\}^*$: there exists an algorithm A that decides L in polynomial time $\}$.*

The class NP has two equivalent definitions.

Definition 10.7 (The class NP) *Language L is in NP if there is a non-deterministic polynomial time algorithm A (Turing Machine) that decides L.*

- *For $x \in L$, A has* **some non-deterministic choice** *of moves that will make A* **accept** *x.*
- *For $x \notin L$,* **no choice** *of moves will make A accept x.*

Definition 10.8 (The class NP) *Language L has an efficient certifier $A(x, y)$, and the algorithm A is a two-input polynomial-time algorithm. The algorithm A verifies language L in polynomial time:*

- *For $x \in L$, there is a string y (certificate) of length polynomial in $|x|$, such that $A(x, y) = 1$.*
- *For $x \notin L$, no string y will make $A(x, y) = 1$.*

Theorem 10.9 *Decision problem X is in P if and only if $\bar{X}$ is in P.*

Proof: If X is in P, let A be a polynomial-time algorithm for X. Construct polynomial time algorithm A' for $\bar{X}$ as follows: given input x, A' runs A on x and if A accepts x, A' rejects x and if A rejects x then A' accepts x. "Only if" direction is essentially the same argument. ■

Problems that require polynomial time to solve are said to be **tractable**. Problems that require exponential time to solve are said to be **intractable**.

In computational complexity, problems that are in the complexity class NP but are neither in the class P nor NP-complete. Such problems are called NP-intermediate. Ladner's theorem, shown in 1975 by Richard E. Ladner [2], if $P \neq NP$, NP-intermediate is not empty. Some problems that are considered good candidates (or believed) for being NP-intermediate are the graph isomorphism problem, factoring, and computing the discrete logarithm.

10.4 Reductions and NP-complete problems

In this section, we would like to explore relations between different problems. Usually, when we meet a new problem A, we would like to ask whether this problem A is at least as hard as some known problem B. The useful technique to answer this question is **reduction**.

Definition 10.10 (Reductions) *A problem A reduces to another problem B if an algorithm that solves B allows us to solve A. We write $A \leq_T B$.*

Definition 10.11 (Polynomial-Time Reducations) *If the problem A reduces to the problem B, then A can be solved using a polynomial (in the input size) number of calls to a subroutine for B, plus a polynomial amount of additional work. We write $A \leq_P B$.*

The Definition 10.10 is called Turing reductions, and also oracle reductions. The general idea of Turing reduction is that we have a TM that computes the solution to one problem, and we use that to compute the solution to another problem. The Definition 10.11 is called polynomial-time reductions, and also called Cook reductions.

Typically, to prove one problem is NP-complete, we use the technique of **polynomial-time many-one reductions**, which are also called Karp reductions.

Definition 10.12 (Karp reductions) *A language L_1 is polynomial-time reducible to a language L_2 ($L_1 \leq_P L_2$) if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$, $x \in L_1$ iff $f(x) \in L_2$.*

By Karp reduction 10.12, proving a reduction $A \leq_P B$ generally involves 3 steps:

1. We have to find the mapping function f and show that it runs in polynomial time.
2. We have to show that if $x \in A$, then $f(x) \in B$. That is the mapping function f will convert any instance of A to the instance of B.
3. We have to show that if $f(x) \in B$, then $x \in A$. Equivalently, we can show that if $x \notin A$ then $f(x) \notin B$.

Definition 10.13 (NP-Hard) *A language L is NP-hard if $L' \leq_P L$ for any language L' in NP.*

Definition 10.14 (NP-Complete) *A language L is NP-Complete if $L' \leq_P L$ for any language L' in NP, and $L \in NP$.*

By Definition 10.14, if we can solve any NP-complete (or NP-Hard) problem in polynomial time, then we will be able to solve, in polynomial time, all the problems in NP.

How to prove a problem is NP-Complete To prove a problem Y is NP-Complete:

1. Show that Y is in NP.
2. Choose an NP-complete problem X, and show that $X \leq_P Y$. (Cook reduction Def. 10.11 and Karp reduction Def. 10.12 are both fine, but usually it is easy to use Karp reduction)

Satisfiability (SAT problem) A Boolean variable x_i or its negation $\bar{x}_i$ is call a literal. A clause is a disjunction of literals, e.g., $C_i = x_1 \vee \bar{x}_2 \vee x_3$. Conjunctive normal form (CNF) Φ is a collection of clauses that are AND'ed together, e.g., $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge \ldots$.

The Satisfiability problem is given a CNF formula Φ, does it have a satisfying truth assignment. The k-SAT problem, in which each clause has exactly k literals.

A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The size of a clique is the number

of vertices it contains. The clique problem is the optimization problem of finding a clique of maximum size in a graph.

As a decision problem of the clique problem, we ask simply whether a clique of a given size k exists in the graph. The formal definition is: CLIQUE $= \{\langle G, k\rangle : G \text{ is a graph with a clique of size } k\}$.

Theorem 10.15 *The clique problem is NP-complete.*

Proof: We prove it by reducing 3-SAT to CLIQUE and showing that CLIQUE is in NP.

Firstly, we show CLIQUE is in NP as below. For a given graph $G = (V, E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. Checking whether V' is a clique can be accomplished in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

Next, we show that $3 - SAT \leq_P$ CLIQUE. Let $\psi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a boolean formula in 3-SAT with k clauses. We shall construct a graph G such that ψ is satisfiable if and only if G has a clique of size k. The algorithm to construct such a graph G is the mapping function f. We shall prove that f runs in polynomial time.

For each clause $C_r = (r_1 \vee r_2 \vee r_3)$ in ψ, we place a triple of vertices r_1, r_2, r_3 in V. Thus, we create a total of $3k$ vertices.

We put an edge between two vertices r_i, s_j if both of the following hold: r_i and s_j are in different triples, that is, $r \neq s$, and their corresponding literals are consistent, that is r_i is not the negation of s_j. In other words, there is an edge between r_i and s_j if r_i and s_j are in different clauses in ψ and $r_i \neq \neg s_j$.

Given any instance ψ of the CLIQUE problem, this graph G can easily be computed in polynomial time.

For example, $\psi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$, the graph G is given in Fig 10.2. Given the ψ, we create vertices $x_1, \neg x_2, \neg x_3$ as a triple corresponding to the clause C_1, vertices $\neg x_1, x_2, x_3$ as a triple corresponding to the clause C_2, and vertices x_1, x_2, x_3 corresponding to the clause C_3, respectively.

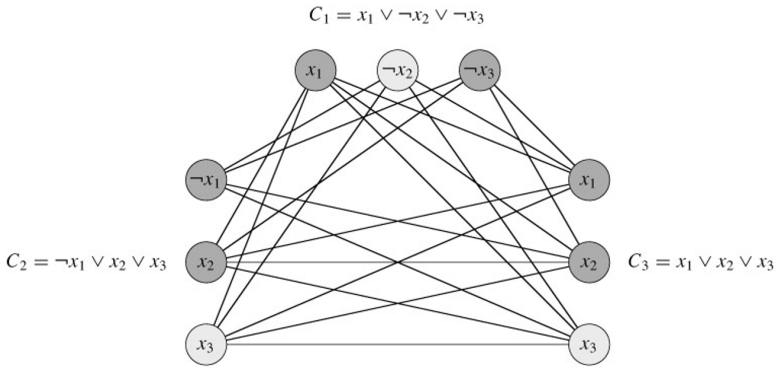

Figure 10.2: The graph constructed from ψ in the example [1]

Note that there is no edge between vertices in the same triple. There is an edge between vertices in different triples if they are not the negation of each other. If x_1 in the first triple does not have an edge between $\neg x_1$ in C_2, because their literals are the negation of each other.

First, suppose that ψ has a satisfying assignment. That means ψ is a "Yes" instance, and it is in the language L, where L is the 3-SAT problem. Then each clause C_r contains at least one literal r_j that is assigned 1. And each such literal corresponds to a vertex r_j. Picking one such "true" literal from each clause yields a set V' of

k vertices. We claim that V' is a clique. For any two vertices r_i, s_j where $r \neq s$, both corresponding literals r_i and s_j are mapped to 1 by the given satisfying assignment, and thus the literals cannot be complements, and thus there exists an edge between r_i and s_j. Thus, V' is a clique and its size is k. Here $V' = f(\psi)$, where f is the mapping function that construct the graph. Now we have V' is a "Yes" instance of the language CLIQUE.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple, and so V' contains exactly one vertex per triple. We can assign 1 to each literal r_i such that the vertex $r_i \in V'$, then assigning 1 to both a literal and its complement can not happen, since G contains no edges between inconsistent literals. Each clause is satisfied, and so ψ is satisfied. Any variables that do not correspond to a vertex in the clique may be set arbitrarily. ■

For the first NP-Complete problem and further examples, we refer to the book [1].

10.5 Self-reduction

Recall that decision problems are those with yes/no answers, while search problems require an explicit solution for a yes instance. An efficient algorithm for search problem implies an efficient algorithm for decision problem. Can an efficient algorithm for decision imply an efficient algorithm for search?

Definition 10.16 (Self-reduction) *A problem is said to be self-reducible if the search problem reduces (by Cook reduction) in polynomial time to the decision problem.*

Theorem 10.17 *SAT is self-reducible.*

Proof: The input of SAT is a formular ψ with n variables $x_1, x_2, \ldots, x_n$. Firstly, set $x_1 = 0$, we get a new formular ψ_1. Check if ψ_1 is satisfiable using decision algorithm. If ψ_1 is satisfiable, then we find the assignment for $x_1 = 0$, we can recursively find assignments to the remaining variables. If ψ_1 is unsatisfiable, then set $x_1 = 1$, we get a new formular ψ_2. Check if ψ_2 is satisfiable using decision algorithm. If ψ_2 is satisfiable, we find the assignment for x_1, that is $x_1 = 1$, we can recursively find assignment to the remaining variables. If ψ_2 is unsatisfiable, we now get both ψ_1 and ψ_2 are unsatisfiable, then ψ is not satisfiable.

It is a polynomial-time reduction because there are at most $2n$ calls to the decision algorithm. ■

Theorem 10.5 states there is a polynomial-time algorithm to find the satisfying assignment if one can periodically check if some formula is satisfiable.

Theorem 10.18 *Every NP-Complete problem/language L is self-reducible.*

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[2] R. E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM (JACM)*, 22(1):155–171, 1975.