Lecture XXX
NP-COMPLETENESS

We have studied many computational problems in the course of this book. Despite the common theme of complexity in our studies, there is so far no coherent framework encompassing these problems. This final chapter introduces some elements of complexity theory to unify a large portion of our investigations.

We have mostly looked at algorithms for computational problems – these provide upper bounds on computational complexity. We have almost exclusively focused on problems that are solvable in polynomial-time. In complexity theory, we are also interested in "inherent complexity". Another way of saying this is we also want to prove lower bounds. This is a much harder quest: for instance, to show that a problem cannot be solved in n^2 time, we must prove something about all conceivable algorithsm for solving the problem! How can one do this? The first thing step is the characterize what are "all conceivable algorithms". This leads to the notion of computational model.

Once this is settled, we need to take another less obvious step: we want to classify problems into those that are "tractable" and those that are not. This step has precedent in the computability theory, where a fundamental dichotomy of problems are the computable ones versus the uncomputable ones. The meta-principle here says that "solvable using a polynomial amount of resources" is equated with **tractability**. This is a meta-principle because we still have to choose the computational resource, machine model, etc. For simplicity, we will assume that the computational resource of interest is time.

As in computability theory, this step turns out to be extremely fruitful, both theoretically as well as in practice. Intractable as well as suspected-intractable problems actually arise very frequently in applications. This forces us to develop new techniques for attacking such problems. While these techniques may be still fundamentally non-polynomial, they allow non-trivial instances to be solved. For instance, improving an algorithm from 2^n time to $2^{\sqrt{n}}$ time can have significant practical impact. Often, in the worst case, we know no better than using a "brute-force search" which typically means an exponential time search for solutions. To circumvent this, we can introduce more powerful computational models (e.g., randomization, approximation) or more refined complexity models (introduction of output-sensitivity in classifying algorithms).

The study of suspected-intractable problems has a discouraging side: all attempts to prove that they are actually intractable has failed miserably. Indeed, we could not even prove that these problems require at least cubic time, say. But the bright side is that researchers discovered a remarkable phenomenon. There is a large class of suspected-intractable problems that are equivalent in the sense that any problem in this equivalence class is tractable if and only if all of them are tractable. This is the theory of NP-completeness which we will study in this chapter. Along the way, we introduce some basic elements of complexity theory.

§1. Some Hard Problems

Consider the following computational problems.

- **Bin Packing**. Recall the linear bin packing problem introduced to illustrate the greedy method: given numbers $(M; w_1, \ldots, w_n)$ we want to pack the weights w_i into the minimum number of bins where each bin has capacity M. The problem is "linear" because the order of packing the weights w_i into bins are specified. In the general bin packing problem, you can rearrange the weights in any way you want. We showed that the general problem can be reduced to linear bin packing to achieve a complexity of $O(n^{n-(1/2)})$.

- **Longest Path Problem**. Given a bigraph $G = (V, E; s)$, we want to compute a "longest path" from s, namely a path $p = (s, v_1, v_2, \ldots, v_k)$ such that k is maximized. The notion of longest path here need to be clarified, because if s can reach any cycle then we can have paths that are arbitrarily long, but no single path is the longest. Since we do not want to exclude cycles from G, we will insist that the "longest path" must by simple (i.e., no vertex is visited twice). This is deceptively similar to the shortest path problem which we can solve using BFS. But we shall see that this is very far from the truth.

- **Travelling Salesman Problem** (TSP). Given a $n \times n$ matrix M whose (i, j)-th entry $(M)_{ij}$ represents the distance from city i to city j, Let π be a permutation of $\{1, \ldots, n\}$, *i.e.*, a bijection $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$. We view π as a **tour** or itinerary of a salesman who begins in city $\pi(1)$, and visits cities $\pi(2), \pi(3), \ldots, \pi(n)$ and finally returning to city $\pi(1)$ again. The cost $C(\pi)$ of this tour is the sum of all the intercity distances travelled. The problem is compute a tour π of minimum cost.

  This problem has many important applications. For instance, in integrated circuit fabrication we may have a very complex circuitry with thousands of points that need soldering by a robot arm. What we want is a minimum cost tour for a robot arm to visit all these point ("cities"). If we can improve on a tour by 10%, this might suggest that the soldering process can be sped up 10%, a real competitive advantage in manufacturing!

- **Knapsack Problem**. Suppose you are packing for your vacation and you have the n items to pack: shoes, clothes, books, toiletry, scuba diving gear, etc. Let the ith item have size $s_i > 0$ and an utility $u_i > 0$. But you have one knapsack with capacity $C > 0$. A subset $I \subseteq \{1, \ldots, n\}$ is called **feasible** if

$$\sum_{i \in I} s_i \leq C.$$

  You are to select a feasible set I such that the utility $u(I) = \sum_{i \in I} u_i$ is maximized.

- **Chromatic Number of a Graph**. Given a bigraph $G = (V, E)$, we want to compute the chromatic number $\chi(G)$ of G. This is defined to be the minimum k such that G has a k-coloring. A k**-coloring** of G is an assignment of the "colors" $1, 2, \ldots, k$ to the vertices of G such that no two adjacent vertices have the same color.

The above problems can be said to be **optimization problems** because there are some minimality or maximality criteria. Typically, any optimization problem can be simplified into **decision problems**, in which the required output is binary-valued (YES/NO). Let us illustrate this remark:

- **Travelling Salesman Decision Problem** (TSD). Given the matrix M as before, and a real number B, does there exist a tour π such that $C(\pi) \leq B$?

- **Knapsack Decision Problem** Given C, $s_1, \ldots, s_n$ and $u_1, \ldots, u_n$ as before, and a real number B, does there exist a feasible set I such that $\sum_{i \in I} u_i \geq B$?

- **Chromatic Number Decision Problem**. Given a bigraph G and an integer $k > 0$, is $\chi(G) \leq k$?

When we discuss complexity of problems, we need a notion of input size. For simplicity, we say that the "size" of each of the above problems is n. In the case of TSP and Knapsack, we need to bound the numbers M_{ij}, s_i, u_i in terms of n. For simplicity, we assume that each number in the input is a binary number with at most n bits.

We currently do know if any of these problems are tractable: that is, whether there exist algorithms with running time $O(n^k)$ for any fixed k. This is true for the optimization problem as well as for their simpler decision counterpart.

It will turn out that as far as tractability is concerned, the original problem is tractable iff the corresponding decision problem is tractable. This may at first appear surprising because it is clear that the decision problem is simpler than the corresponding optimization problem. This can be make rigorous using the notion of reduction which we will introduce below. In view of this tractability-equivalence, the theory we are about to develop will mostly deal with decision problems.

The above list is just a small sampling of a host of problems not known to be tractable. What is more remarkable is that they all share this characteristic: if any one of these problems is shown to be tractable, then all of them would be tractable. Such problems are "*NP*-Complete", a concept we will shortly introduce. The book [1] contains a list of over 300 problems from all areas of the computational literature with the same property. Of course, the list has grown considerably since the writing of the book. The existence of this *NP*-completeness phenomenon has important implications for the study of algorithms.

- First, it tells us that there is overwhelming evidence for the inherent difficulty of these problems. In fact, most experts believe that these problems are intractable.
- Second, instead of attempting to show efficient algorithms for a problem, especially if we suspect that it is not possible, we can also attempt to show it to be *NP*-complete. This would bring relative closure to our investigation, as a kind of negative result.
- Third, it has led to the investigation of new computational techniques (especially randomized ones) for attacking such problems.

In short, the overall impact of this theory on the computational literature is far-ranging.

Exercises

Exercise 1.1: Give some good algorithmic solution for the following problems: (a) TSP, (b) Chromatic Number and (c) Knapsack. Note that while your solution will be non-polynomial, you should try to make it as efficient as you can. ♢

Exercise 1.2: For each case of the previous question, estimate the largest size n of the problem that your algorithm can solve in one day of (current) computer time. Make explicit any assumptions you need (speed of your computer, etc). ♢

End Exercises

§2. Model of Computation

In order to bring the various problems under one framework, we need to have a "universal computational model". Many general models of computation have been proposed. Relative to goal of classifying computable and noncomputable problems, all these models turn out to be equivalent. But in terms of complexity, the issue is considerably more subtle (this is related to the concept of "computational modes" [2]). In any case, the canonical choice here is the **Turing machine model**. Again, there are many variants of Turing machines. For our present purpose, we use the **Simple Turing Machine** (STM) model.

REWRITE THE FOLLOWING INFORMATION (Use Theory Lecture Notes)

We start with the initial idea of a **finite state machine**. This machine M can be represented by a directed graph whose vertex set Q is finite and where each edge is labeled by a symbol from a set Σ called the **alphabet** of M. The vertices in Q are called **states** and edges called **transitions**. If a transition $(u, v) \in Q^2$ is labeled by a symbol $a \in \Sigma$, we may denote it by $(u \xrightarrow{a} v)$. There are distinguished states, a **start state** $q_0 \in Q$ and an **accept state** $q_a \in Q$.

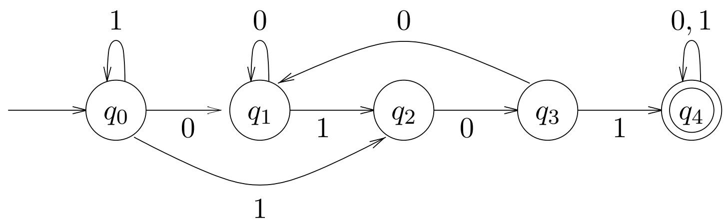

Figure 1: Finite State Machine

EXAMPLE. In Figure 1, we show a finite state machine illustrating several conventions. The alphabet here is $\Sigma = \{0, 1\}$ and state set is $Q = \{q_0, q_1, \ldots, q_4\}$. The start state is q_0 (this is indicated by the arrow from nowhere, and its accept state q_a is q_4 (indicated by the concentric circles around q_4). When two or more transitions (edges) share the same start and end vertices, we will just draw one arrow, labeled by two or more the symbols. Thus, the transitions $(q_4 \xrightarrow{0} q_4)$ and transitions $(q_4 \xrightarrow{1} q_4)$ in this figure is represented by just one edge, labeled by the symbols 0 and 1.

The operation of a finite state machine M is as follows: its input is some string $w \in \Sigma^*$. It executes a sequence of transitions in the following sense: at any moment, M has a current state and a current **head position** $h \in \{1, 2, \ldots, n, n+1\}$ where $|w| = n$. Initially, M is in state q_0 and head position is $h = 1$. In head position h, we say that M is **scanning** the hth symbol $w[h] \in \Sigma$ in w. When in state q and scanning symbol $a \in \Sigma$, our machine M can **execute** any transition of the form $(q \xrightarrow{a} q')$, and thereby move into state q'. Its head position is incremented, $h \leftarrow h + 1$. Note two possibilities:

- If there is more than one transition from state q that are labeled by symbol a, then M can execute any one of them. In this case, we say M made a **nondeterministic move**.
- M could be **stuck** in the sense that there are no executable transitions from state q.

EXAMPLE (contd). Suppose $w = 000100$. Then the machine in Figure 1 would enter the following sequence of states as the head moves from position 1 to 6 is $(q_0, q_1, q_1, q_1, q_2, q_3, q_1)$. In this case, no non-deterministic moves were made. Suppose $w = 111$. In this case, the very first move must make a non-deterministic (M can go to state q_2 or remain in state q_0). One possible sequence of states might be $(q_0, q_2, *)$ where $*$ indicates that the machine is now stuck at head position 2. Alternatively, the state sequence can be (q_0, q_0, q_0, q_2).

A **configuration** of M is a pair $c = (q, h)$ where q is a state and h is a position. A **computation** of M is a sequence of configurations $C = (c_1, \ldots, c_m)$ for some $m \leq n$ and such that $c_i \to c_{i+1}$ is legal for all $i = 1, \ldots, m-1$. We say C is an **accepting computation** if the state in c_m is the accept state. We say M **accepts** w if there is an accepting computation of M on input w. Note that M does not accept q_a, there for every computation path C are two possibilities: it could be stuck before reaching position $n+1$, or it could reach position $n+1$ in a state different than q_a. A machine M is said to be **nondeterministic** it has at least one pair of transitions $(u \xrightarrow{a} v)$ $(u' \xrightarrow{a'} v')$ where $u = u'$ and $a = a'$. Otherwise we say M is **deterministic**.

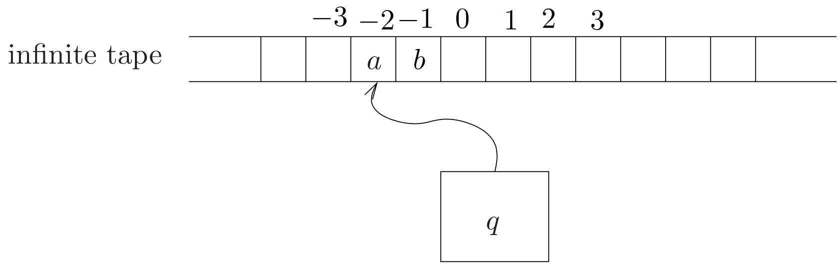

Figure 2: Turing machine in state $q \in Q$

A **simple Turing machine** is just an extension of a finite state machine in which each transition is now labeled by a triple $(a, a', D) \in \Sigma \times \Sigma \times \{0, \pm 1\}$. The idea is that the machine can now change the symbol at its current position from a to a', and its head position is changed by the amount D, from h to $h + D$. If $D = 0$, its head position is unchanged, and $D = -1$ means that its head position can move left. In the finite state machine, D is always $+1$ implicitly. Also, the computation is not required to stop after $|w|$ transitions – it can even continue forever. The computation halts when it reaches the accept state q_a or is stuck.

Another new feature is that we view the machine as computing on a doubly-infinite **tape** where the tape is made up of individual **cells** (tape squares) which are indexed by the integers. Each cell can store a symbol from Σ or a special **blank symbol** $\sqcup$ (assumed not in Σ). The transition rules can refer to symbols in $\Sigma \cup \{\sqcup\}$. Initially, the input w is placed on cells $1, 2, \ldots, |w|$, and all the other cells are initially blank (i.e., contains $\sqcup$). The tape head is scanning cell 0 (thus sees a blank).

REMARK: we call the above model the "simple Turing machine" (STM) because there are many variants of Turing machines, and these are invariably more elaborate than our version. But for our purposes, the STM model suffices.

What does a Turing machine compute? Our main use of the simple Turing is to **accept languages**. That is, we say M **accepts** $w \in \Sigma^*$ if there exists a computational path on input w that leads to the accept state. Let

$$L(M) \subseteq \Sigma^*$$

denote the language accepted by M.

We sometimes need simple Turing machines to compute functions of the form $f : \Sigma^* \to \Sigma^*$. The function f may be partial. For this purpose, we need some conventions. First of all, it is easiest to assume M is deterministic. On input w, if M does not halt, then $f(w) \uparrow$. Otherwise, when it halts, let the tape head be scanning cell i for some $i \in \mathbb{Z}$. If cell i is blank, then $f(w) = \epsilon$ (the empty string). Otherwise, there is a maximal contiguous block of cells that contains non-blank symbols and that includes cell i. The output $f(w)$ is the word contained in the block of cells.

Exercises

Exercise 2.1: Construct a Turing machine M to check if a bigraph is connected. Assume (be explicit) some reasonable encoding of bigraphs. Please describe the actions of M in words, *not* by writing down its set of instructions! $\diamondsuit$

Exercise 2.2: Show that if TSD can be solved in polynomial time, then we can solve TSP in polynomial time. ♢

End Exercises

§3. Computational Problems

The above computational model apparently computes on input strings. But computational problems arise in mathematical domains such as integers, sets, graphs, matrices, etc. In order to solve these problems, we must therefore assume some encoding of these objects as strings. The following will be assumed unless otherwise indicated:

- Integers: these are represented in binary notation. We can generalize this to rational numbers, represented by a pair of integers.
- Matrices: assuming a representation of the matrix entries (say binary numbers) then the entire matrix can be represented by a row-major order lising of entries. Of course, we must also explicitly encode the size of the matrix.
- Sets: again, relative to some encoding of the elements of the set, we encode a set by an arbitrary listing of its elements. The encoding of a set is not unique (there are $n!$ possible ways to list its elements).

If g is an object, we may write $\#(g)$ for the encoded version of g. But often, we do not even make this distinction, and identify g with $\#(g)$.

The above encoding of matrices includes vectors or tuples as special cases. In these encodings, it is simplest if we introduce new symbols (e.g., commas and parenthesis symbols) to separate items in a list or set.

EXAMPLE: encoding of digraphs. Three main methods are: (1) listing of the edge set, (2) adjacency lists and (3) adjacency matrix. Assuming that the nodes have some given encoding already (say, as integers) and edges are just pairs of nodes, then method (1) amounts to a representation of a set, and method (2) amounts to a list of lists of nodes. Method (3) can be viewed as a boolean matrix.

Efficiency of Encoding. The choice of encoding is usually not important, but there are exceptions. The most important example is the encoding of integers: we can use k-ary encoding of integers for any $k > 2$, instead of the default binary encoding ($k = 2$). On the other hand, we must not use unary encoding ($k = 1$). The reason is that this is exponentially less efficient than k-ary encoding for $k > 1$. This will have drastic consequence on the complexity of the problem: an exponential time problem may become polynomial time just by this encoding artifact. This shows that it is important to have "compact encodings". On the other hand, we should not insist on the most compact encoding, as this would involve difficult computational problems to find the most compact one!

Satisfiability Problem. A **Boolean formula** is an expression over the infinite supply of Boolean variables

$$x_0, x_1, x_2, \ldots$$

and defined recursively: any Boolean variable is a Boolean formula. If F_1, F_2 are Boolean formulas, then so are

$$(\neg F_1), \quad (F_1 \vee F_2), \quad (F_1 \wedge F_2).$$

As a stylistic variant, we can also write these formulas in the "arithmetical style", namely,

$$(-F_1), \quad (F_1 + F_2), \quad (F_1 * F_2).$$

The reason for the arithmetical style is that we usually like to simplify our writing of formulas by dropping parethesis and using implicit operators. In particular, we want to exploit (1) associativity of $\vee$ and $\wedge$, (2) introduce rules of precedence for operators, and (3) replace $\wedge$ by juxtaposition of variables (i.e., write $x_0 x_1$ instead of $x_0 \wedge x_1$. Most people are familiar with similar rules in arithmetical expressions, and can parse such formulas easily. Hence the motivation the arithmetical style. For example: we intend to write $x \vee y \vee z$, instead of the formally correct $((x \vee y) \vee z)$ (exploiting associativity). If we assume $\wedge$ has higher precedence than $\vee$, we can also write $x \vee y \wedge z$ instead of $(x \vee (y \wedge z))$. But if we write this in arithmetic style, we get the even more familiar and compact expressions, $x + y * z$ or $x + yz$. Instead of `-x` we also write $\overline{\mathtt{x}}$.

Satisfaction. We define satisfiability of a Boolean formula. An **assignment** for F is a function $I : V \to \{0, 1\}$. where V contains all the variables that occurs in the formula F (V may contain more than the variables that occurs in F). We say I **satisfies** F as follows: (BASIS) If F is a variable x, then I satisfies F iff $I(x) = 1$. (INDUCTION) If $F = (-F_1)$, then I satisfies F iff I does not satisfy F_1. If $F = (F_1 + F_2)$, then I satisfies F iff I satisfies F_1 or F_2. If $F = (F_1 * F_2)$, then I satisfies F iff I satisfies F_1 and F_2.

We say F is **satisfiable** if there exist some I that satisfies F. Let SAT denote the set of all satisfiable Boolean formulas.

EXAMPLE: the formula

$$F = (x + y + z)(x + \overline{y})(y + \overline{z})(z + \overline{x})(\overline{x} + \overline{y} + \overline{z}) \tag{1}$$

is not satisfiable, as the reader may verify.

Let us discuss the encoding of Boolean formulas. For any formula F, let $\#(F)$ denote its encoding. We will use the the alphabet $\Sigma = \{\mathtt{x,0,1,+,*,(,)}\}$ and so $\#(F) \in \Sigma^*$. A string $w \in \Sigma^*$ is said to be **well-formed** if it is equal to $\#(F)$ for some F; otherwise it is **ill-formed**.

For variable x_i let $\#(x_i)$ be the string that begins with `x` followed by the binary representation of i. E.g., $\#(x_5) = \mathtt{x101}$.

Lemma 1 *A simple Turing machine can decide if a string $w \in \Sigma$ is a well-formed Boolean formula or not in polynomial time.*

Discussion: In general, when we introduce encodings, we are faced with the problem of well-formedness. Let $\mathcal{D}$ be some mathematical domain, and

$$\# : \mathcal{D} \to \Sigma^* \tag{2}$$

be any **encoding** (i.e., a $1-1$) map. The inverse of this encoding

$$\rho : \Sigma^* \to \mathcal{D} \tag{3}$$

is called a **representation** of $\mathcal{D}$. Note that this inverse is a partial $1-1$ function: $\rho(w)$ can be undefined. This corresponds to the case where w is ill-defined. In general, a representation of $\mathcal{D}$ is any function (3) that is a partial $1-1$ function.

We have two computational problems associated to any ρ: (1) The **parsing problem** is to determine if a string w is well-formed. (2) The **equality problem** is to determine if two well-formed strings w, w' represents the same object in $\mathcal{D}$.

Example: let $\mathcal{D} = \mathbb{N}$. The usual representation of $\mathbb{N}$ is the binary representation with $\Sigma = \{0, 1\}$. Parsing is trivial because every binary string is well-formed. The equality problem is also easy because two binary strings represents the same number if, after omitting any leading 0's, they are the same string.

Let f is any operation $f : \mathcal{D}^n \to \mathcal{D}$. Relative to the representation (3), an algorithm F **implements** f if, for every well-formed $w_1, \ldots, w_n$, the algorithm computes $F(w_1, \ldots, w_n)$ such that $\rho(F(w_1, \ldots, w_n)) = f(\rho(w_1), \ldots, \rho(w_n))$. Thus, the usual high school algorithm implements the multiplication operation on $\mathbb{N}$ relative to the standard binary representation.

Usually both the parsing and equality problems are polynomial time. But they become an issue for mathematical domains that are "abstract", whose their objects might be defined by some non-trivial equivalence relation over more concrete ones. For instance, in graph theory, we normally identify two graphs up to isomorphism, meaning a renaming of their vertices so that the have the same set of edges. Let $\mathcal{G}$ be the set of these abstract graphs. The equality problem for any encoding of $\mathcal{G}$ is the **graph isomorphism problem**. It is not known if there exists a representation $\rho : \mathcal{G} \to \Sigma^*$ such that both the parsing and equality problems are polynomial time.

Exercises

Exercise 3.1: Give a representation of the mathematical domain $\mathbb{N}$ such that the parsing problem is easy and the operation of multiplication can be implemented in linear time. How efficiently can you implement the operation of addition in this representation? $\diamondsuit$

Exercise 3.2: (i) Give an representation of $\mathcal{G}$ for which the parsing problem can be decided in polynomial time.
(ii) Give the best algorithm you can for deciding if two well-formed strings represent the same graph of $\mathcal{G}$. HINT: do not expect to find a polynomial time algorithm. $\diamondsuit$

End Exercises

§4. Complexity Classes

We now introduce concepts of complexity. By a **complexity function** we mean a partial function

$$f : \mathbb{R} \to \mathbb{R} \cup \{\infty\}$$

that is defined on the natural numbers. We are usually interested in **families** of complexity functions. The following are the main families:

$$\mathcal{O}(\log n), \quad \mathcal{O}(n), \quad n^{\mathcal{O}(1)}, \quad \mathcal{O}(1)^n, \quad 2^{n^{\mathcal{O}(1)}}.$$

We introduce **(computational) resources**: time and space will be our most important examples of resources. Define the **time** of the computation path π to be one less then the number of configurations in the sequence (which could be infinite). The **space** of π is the total number of cells that are scanned by some work tape in some configuration in π. Note that the cells in the input tape are not counted.

For any complexity function f and TM M, we define what it means for M to **accept in time** f: this means that for all inputs w of length n, if M accepts w then there is an accepting computation path using time at most $f(n)$. Note that if M does not accept w then we impose no requirement. Also, f is just an upper bound on the computation length. We similarly define what it means for M to **accept in space** f.

Finally, a **complexity class** K is **characterized** by a choice of mode μ, family F of complexity functions and a computational resource ρ. We write

$$K = \chi(\mu, \rho, F)$$

to denote the class of languages L such that there exists $f \in F$ and a μ-TM that accepts L in ρ $f(n)$. A more standard way to represent these classes is to associate symbols with each of these parameters: D for deterministic, N for nondeterministic, $TIME$ for time and $SPACE$ for space. Then $\chi(deterministic, time, F)$ is usually denoted $DTIME(F)$. If $F = \{f\}$ then we write $DTIME(f)$ instead of $DTIME(\{f\})$. Similarly, the notationa $NTIME(F), DSPACE(F)$ and $NSPACE(F)$ are self-explanatory.

The Classes P and NP. Using the above conventions, the class

$$DTIME(n^{\mathcal{O}}(1))$$

comprises the languages accepted by deterministic TM running in polynomial time. This class is usually denoted P. Again, $NTIME(n^{\mathcal{O}}(1))$ is similar to P except the mode is non-deterministic and this class is usually denoted NP. Another important class is $PSPACE := DSPACE(n^{\mathcal{O}(1)})$. The following inclusions are straightforward to show:

$$P \subseteq NP \subseteq PSPACE.$$

These classes are usually called **Deterministic Polynomial Time**, **Nondeterministic Polynomial Time** and **Polynomial Space**, respectively. These are extremely important classes for several reasons: most problems that we can solve in practice falls under these classes. Of course, if we agree that "tractable" means deterministic polynomial time, then P is just the class of tractable problems.

Satisfiablity. We now verify the membership of some important prblems in the class NP.

Lemma 2 *$SAT \in NP$.*

Variation: A 3-**conjunctive normal Form** (3CNF) formula is a Boolean formula that is a conjunction of disjuncts, where each disjunct has at most 3 literals. Our above example is a $3CNF$ formula. The $3SAT$ problem is the restriction of SAT to inputs that are in $3CNF$.

Hamiltonian Path Problem. A **Hamiltonian circuit** of a bigraph G is a simple closed path that visits every vertex in G. Let HAM denote the set of (encodings) of G that has Hamiltonian circuits.

Lemma 3 *$HAM \in NP$.*

Exercises

Exercise 4.1: Show that everything computed by a deterministic TM can be computed by a non-deterministic TM in the same time and space ♢

Exercise 4.2: Another approach to *NP* is as follows: A **verification machine** M is a deterministic Turing machine with *two* input tapes. An input is a pair (w, v) with w on the first input tape and v on the second input tape. We say M **verifies** a word $w \in \Sigma^*$ if there exists a word $v \in \Sigma^*$ such that on input (w, v), M eventually enters the accept state q_a and halts. Say M **verifies in time** $t(n)$ if for all inputs w, if M verifies w then there exists a v such that M on (w, v) will halt within $t(|w|)$ steps. Let $V(M)$ be the set of words that is verified by M. Show that L is in *NP* iff L is verified by a polynomial-time verification machine. ♢

Exercise 4.3: Show that $NP \subseteq PSPACE$. ♢

End Exercises

§5. Reductions

Let T be a deterministic Turing machine acting as a transducer and computing the **transformation** $f : \Sigma^* \to \Sigma^*$.

We say (L, Σ) is **Karp-reducible** (or, simply, **reducible**) to (L', Σ') if there exists a polynomial-time computable transformation f such that for all $x \in \Sigma^*$,

$$x \in L \quad \text{iff} \quad f(x) \in L'.$$

We also write

$$L \leq_m^P L'.$$

Lemma 4 **(i) Transitivity** *If $L \leq_m^P L'$ and $L' \leq_m^P L''$ then $L \leq_m^P L''$.*

(ii) Closure of P *If $L \leq_m^P L'$ and $L' \in P$ then $L \in P$.*

Lemma 5

$$HAM \leq_m^P SAT$$

Proof. Given G we construct a 3CNF formula $f(G)$ that is satisfiable iff $G \in HAM$. Assume nodes of G are $\{1, \ldots, n\}$. A **tour** of G is a path $T = (u_1, \ldots, u_n)$ such that (u_i, u_{i+1}) is an edge of G for $i = 1, \ldots, n$ (where we assume $u_{n+1} = u_1$). Hence a tour represents a cycle of G. Introduce a variable x_{ij} where i range over the nodes in G and j ranges from 1 to n. We want x_{ij} to stand for the proposition about some unknown tour T of G:

Node i is the jth node in tour T.

With the help of these elementary propositions x_{ij}, we write down the following propositions that must be true of T:
(1) For each j, there is a unique i such that x_{ij} is true.
(2) For each i, there is a unique j such that x_{ij} is true.
(3) For each $i \neq i'$, if x_{ij} and $x_{i',j+1}$ are true then (i, i') is an edge of G.

This is quite easy, so we just illustrate the proposition (1):

$$\left(\bigvee_{i=1}^{n} x_{ij}\right) \wedge \left(\bigwedge_{1\le i<i'\le n} (\overline{x}_{ij} \vee \overline{x}_{i'j})\right).$$

It is clear that if G has a tour, then (1), (2) and (3) must be satisfiable. Conversely, if (1), (2) and (3) are satisfiable, we can construct a tour of G. **Q.E.D.**

Exercises

Exercise 5.1: Prove the transitivity and closure properties of Karp-reducibility. ◊

Exercise 5.2: A bigraph $G = (V, E)$ is said to be **triangular** if $|V| = 3n$ for some n and V can be partitioned into n disjoint subsets

$$V_1 \uplus V_2 \uplus \cdots \uplus V_n$$

where each V_i has three vertices that form a triangle, *i.e.*, if $V_i = \{u, v, w\}$ then $\{(u, v), (v, w), (w, u)\} \subseteq E$. Let L be the set of encodings of triangular bigraphs. We want to show by a direct argument that L is Karp-reducible to SAT. We will guide you through a sequence of subproblems to solve this: To show that L is Karp-reducible to SAT, you need to construct a Boolean formula $\phi(G)$ such that G is triangular iff $\phi(G) \in SAT$. Moreover, this construction must be polynomial-time.
(i) If $G = (V, E)$ and $|V|$ is not divisible by 3 then there is no solution. What would you output as $\phi(G)$ in this case?
(ii) Suppose $|V| = 3m$. So our goal is to form m disjoint triangles from the vertices of G. Introduce the Boolean variable x_{ij} which corresponds to the proposition "Node i is in the jth Triangle". Here, $i \in V$ and $j = 1, \ldots, m$. Using these variables, you construct a Boolean formula $F_1(i)$ that is satisfiable iff i is in at least one of the m triangles?
(iii) Similarly, construct $F_2(i)$ that is satisfiable iff i is in at most one triangle.
(iii) Construct a formula $F_3(j)$ that is satisfiable iff the jth triangle has at least three nodes.
(iv) Construct a formula $F_4(j)$ that is satisfiable iff the jth triangle has at most three nodes.
(v) Construct a formula $F_5(j)$ that is satisfiable iff each pair of vertices in the jth triangle has an edge in the graph G. [NOTE: this is the first time you are actually using specific information about the edges of G. You know G since it is in the input.]
(vi) Using the above formulas, describe the formula $\phi(G)$ that is satisfiable iff G is triangular. You must prove this claimed property about $\phi(G)$.
(vii) Conclude that L is Karp-reducible to SAT. ◊

Exercise 5.3: We continue to consider the problem L of recognizing triangular graphs from the previous exercise.
(i) Show by a direct argument that L is in NP.
(ii) Conclude that L is K-reducible to SAT.

Remark: In other words, we could short cut the explicit "reduction" of the previous exercise to come to the same conclusion!. ◊

Exercise 5.4: Suppose instead of polynomial time, we restrict the transducer to run in logarithmic space and linear time. Prove the transitivity and closure properties of such reducibility. $\diamondsuit$

End Exercises

§6. Fundamental Questions and Completeness

The most important open questions of complexity theory are all of the form: is $K \subseteq K'$ where K, K' are complexity classes. The most famous of such questions is the $NP \subseteq P$ question. A fundamental tool to study such **inclusion questions** is the theory of **completeness**.

Let K be a class of languages. A language L is K-**hard** if for all $L' \in K$, $L' \leq_m^P L$. We say L is K-**complete** if L is K-hard and $L \in K$. Here we prove some simple lemmas for the case $K = NP$.

Lemma 6 *Let L_0 be NP-complete. If $L \in P$ then $P = NP$.*

Thus, we transform inclusion questions about a class into questions about a single language in the class! But are there any NP-complete languages?

Theorem 7 (Cook's Theorem) *SAT is NP-complete.*

Once we get one complete language, we can show more by the following technique:

Lemma 8 *If $L \in NP$ and $L' \leq_m^P L$ then L' is NP-complete implies L is NP-complete.*

Lemma 9 *$3SAT$ in NP-complete.*

Proof. By the previous lemma, we only have to reduce SAT to $3SAT$. **Q.E.D.**

Lemma 10 *HAM is NP-complete.*

Proof. We will reduce $3SAT$ to HAM. Let F be a $3CNF$ formula. We will construct a graph $G = G_F$ such that F is satisfiable iff G_F has a Hamiltonian circuit. We need two types of "gadgets":

Figure 3(a) shows the choice gadget and figure 3(b) shows the XOR (exclusive-or) gadget. These gadgets have **entry nodes** (indicated by large black circles and labeled "*in*" or "*out*", respectively). We will put several of these gadgets together to form G_F. There will be additional edges added in G_F but these edges will only connect to each gadgets via the entry nodes. Let us note some properties of these gadgets.

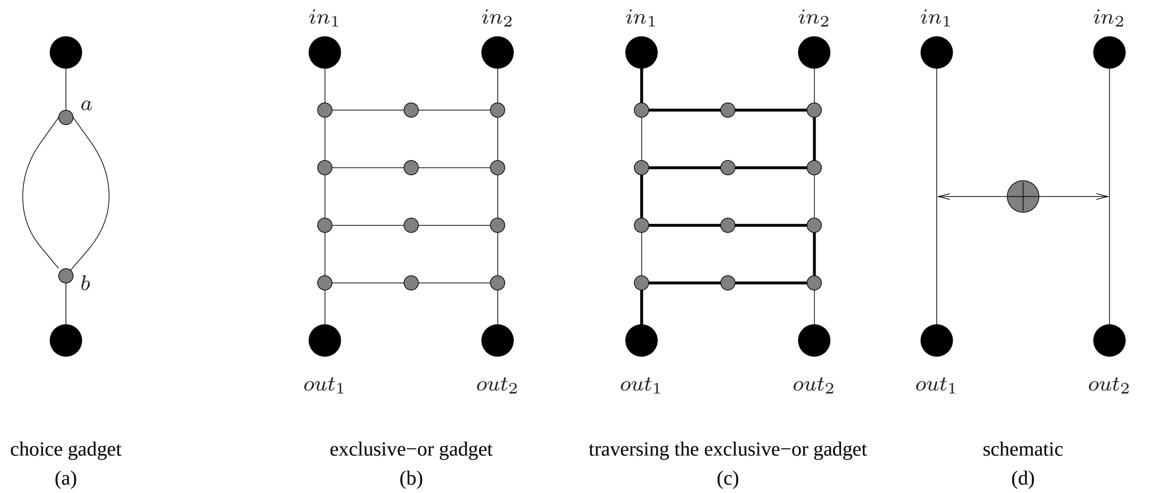

Figure 3: Gadgets for reducing SAT to HAM

- The choice gadget is strictly speaking not a graph — it is a multigraph because it has two parallel edges (*i.e.*, edges sharing the same pair of endpoints). But this will not be a problem because in the course of putting together these gadgets, we will be inserting vertices into one of the parallel edge. Let us call the two parallel edges the **choice paths** (in a Hamiltonian cycle of the constructed graph, we will have to choose one of these two paths). Also, the two non-entry vertices (a, b in figure 3(a)) of the choice gadget are called **choice vertices**.
- The XOR gadget has 4 vertices of degree 2 each. These vertices can only be visited in a Hamiltonian cycle that enters through one of these entry nodes. But it is not hard to see that if the Hamiltonian cycle enters the gadget through the entry node labeled in_1 then it must exit via the node out_1, as illustrated in figure 3(c). Otherwise, the some vertex of degree 2 will not be visited. We call this a **traversal** of the XOR gadget. Of course, the symmetrical traversal holds with respect to the entry nodes in_2, out_2. These two traversals are the only ways to visit all the 4 vertices of degree 2 in a Hamiltonian circuit. In figure 3(d), we have a schematic representation of the XOR gadget: intuitively, this schematic suggests that in_1 and out_1 are connected by an "edge", and so are in_2 and out_2. Moreover, only one of these two "edges" can be traversed (hence they are linked by an exclusive-or $\oplus$ symbol).

It is best to show how we form G_F by an example. Let F be the formula

$$(x + y + z)(x + \overline{y} + z)(\overline{x} + \overline{y} + \overline{z}). \tag{4}$$

To form G, we use one choice gadget to "simulate" each variable in F and three XOR gadgets to "simulate" each clause of F. For the choice gadget that simulates a variable x_i ($i = 1, 2, 3$), its two choice paths are labeled x_i and $\overline{x_i}$, respectively. The choice gadgets are linked together sequentially in an arbitrary linear order as shown in figure 4(a). Call this the "choice chain". Let s_0, t_0 be the first and last node in the choice chain.

Consider the clause $x + \overline{y} + z$. The three XOR gadgets for simulating this clause corresponds to the literals $x, \overline{y}, z$. The six in_1 or out_1 entry nodes in these gadgets are identified in pairs so that they form a "triangle" of nodes – see figure 4(b). The in_2, out_2 entry nodes of XOR gadget are "spliced into" the choice paths that is labeled by the corresponding literal in the choice chain, as in figure 4(c). More precisely, each XOR gadget has a path of length 5 connecting in_2 and out_2: this path is now made a subpath of the corresponding choice path. We do this for each clause. In our example, the literal $\overline{y}$ occurs in two clauses. Hence two paths of length 5 will be spliced into the choice path labeled $\overline{y}$ so that this choice path has length 13 in the final graph G. See figure 4(d).

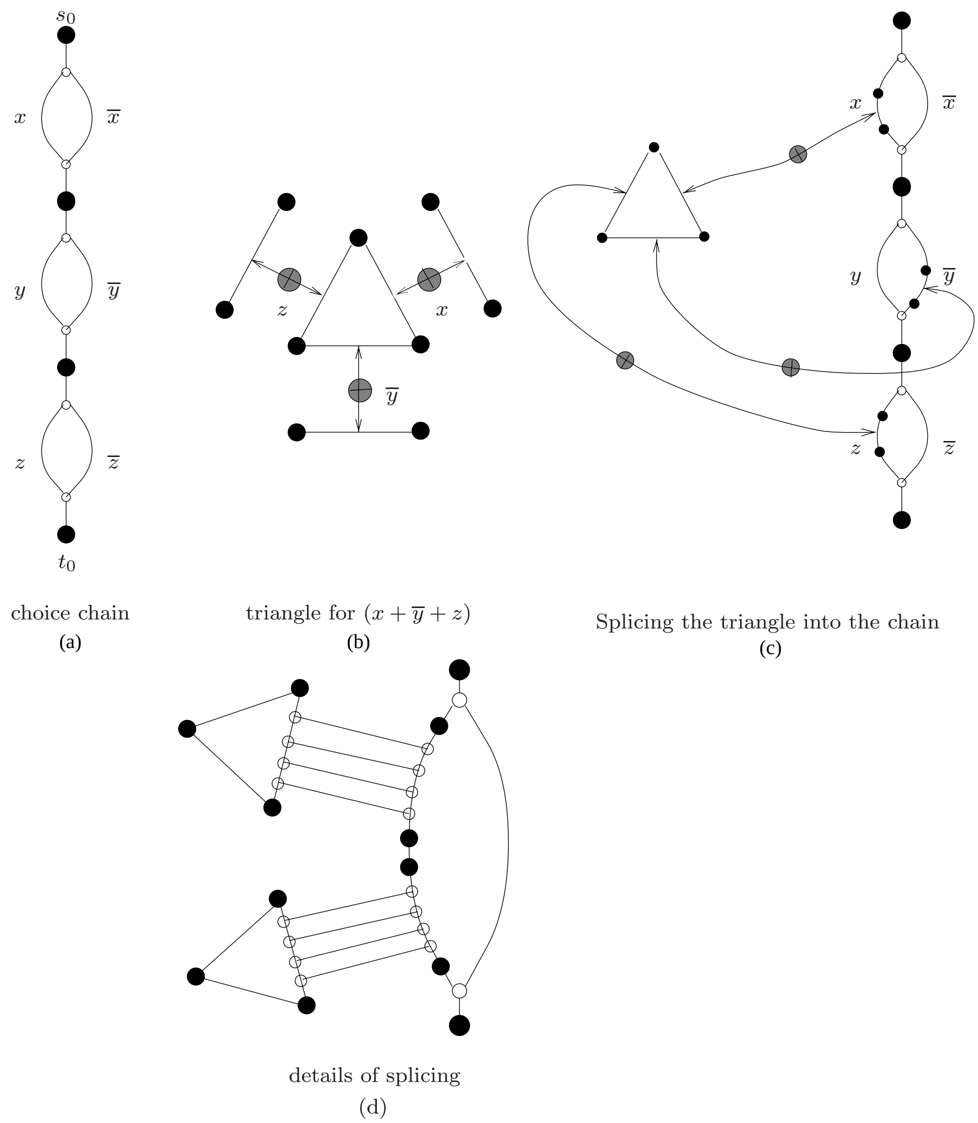

Figure 4: Graph corresponding to F

Finally, we add the edges of the complete graph K defined on the following set of vertices: (i) entry nodes in triangles (there are three such nodes per triangle), and (ii) the first and last entry node in each choice path (there are four such nodes per choice gadget). This completes our description of the graph G_F.

F is satisfiable implies $G_F \in HAM$: Suppose F is satisfiable by an assignment I to the variables. We show how to construct a Hamiltonian cycle: starting from s_0, we traverse each choice gadget such that for each variable x_i, if $I(x_i) = 1$ then we take the choice path labeled x_i, and otherwise we take the choice path labeled $\overline{x_i}$. Now, as we traverse a choice path, we are obliged to traverse each XOR gadget that is spliced into that path, in the canonical way illustrated in figure 3(c). Since I satisfies F, this means that in every triangle, at least one of the three XOR gadgets is traversed. This proceeds until we reach node t_0. At this point, two kinds of entry nodes are still not yet visited:

(I) Entry nodes in choice paths that are not taken,

(II) Entry nodes that forms the corners of triangles (such entry nodes have subscript 1).

We now use the edges of the complete graph K: from t_0, we start to visit entry nodes of type (I). When this is done, we start to visit the entry nodes of type (II). But now, we also take the opportunity to traverse any XOR gadget that is not yet traversed. Note that since I is a satisfying assignment, there are at most two XOR gadgets in a triangle that is not yet traversed. It is easy to see how to traverse the $0, 1$ or 2 XOR gadgets in each triangle, in addition to visiting the 3 entry nodes per triangle. At the end of this process, we use an edge of K to take us back to the starting vertex s_0. This completes our description of a Hamiltonion circuit.

$G_F \in HAM$ implies F is satisfiable: Suppose H is a Hamiltonion cycle. First, we claim that H must traverse exactly one of choice paths for each choice gadget: if it traverse neither of the choice paths, then there is no way the two choice vertices of the gadget could be visited by H. If it traverse both choice paths, then some entry node common to two choice gadgets will not be visited. From this claim, we conclude that H defines an assignment $I = I_H$ corresponding to the choice paths that it traverses. We next claim that I_H must be a satisfying assignment. This means that each triangle must have at least one XOR gadget traversed from the choice paths. If not, we could not traverse the three XOR gadgets using the entry nodes in each triangle. This concludes our proof.

Q.E.D.

Exercises

Exercise 6.1: Complete the reduction of SAT to HAM. Show in particular: if F is satisfiable, then the graph $f(F)$ has a Hamiltonion circuit, and conversely. $\diamondsuit$

End Exercises

§7. Postscript

The significance of P, NP is that P can be identified with the "tractable problems" and NP contains many important problems of interest for which we do not know how to solve in polynomial time. Almost all of these problems have been shown to be NP-complete. Hence if any of these is in P then all of them are.

The list has grown to hundreds of problems in all areas of computational literature. Thus it serves to unify diverse areas.

It also serves as a guide to what problems can be put into P. If your problem of interest looks similar to an NP-complete problem, you should be careful.

This forces us to consider other "computational modes" such as randomization, parallelization, or even quantum modes. Another approach is to relax the optimization problem to epsilon-approximation problems. Another direction is distinguish among the input complexity parameters of problem, and to improve on the critical exponential parameter. For instance, in many problems, there are two input parameters say k and n and the exponential behavior is in k alone. An example is the problem of deciding if a graph has chromatic number at most k. If the graph has n vertices, then the algorithm is exponential in k but polynomial in k, e.g., $O(2^k n^2)$. If we can improve the algorithm to $O(2^{\alpha k} n^{O(1)})$ for some $\alpha < 1$, then asymptotically, we have a faster algorithm.

References

[1] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, 1979.

[2] C. K. Yap. Introduction to the theory of complexity classes, 1987. Book Manuscript. Preliminary version (on ftp since 1990),
URL `ftp://cs.nyu.edu/pub/local/yap/complexity-bk`.