CS663 Theory of Computation

1 Introduction

1.1 What is Theory of Computation?

Theory of Computation is to study the fundamental capabilities and limitations of computers. It is all about bounds. It contains three areas.

- Automata theory: Models of computation. Seeking a precise but concise definition of a computer. FA$\rightarrow$PDA$\rightarrow$LBA$\rightarrow$TM.
- Computability theory: What can and cannot a computer do? Computationally unsolvable versus computationally solvable problems. Determining whether a mathematical statement is true or false is unsolvable by any computer algorithms.
- Complexity theory: What can a computer do efficiently? Computationally hard versus computationally easy problems. For example, factorization and sorting. Cryptography needs hard problems such as factorization to ensure security.

1.2 A Brief History of Theory of Computation

1.2.1 The very beginning

- 1900: An eventful year (Sigmund Freud, Max Planck, and David Hilbert).

  The Hilbert's tenth problem (proposed on the 2nd International Congress of Mathematicians held in 1900 and was among a list of 23 open problems for the new century): Devise a process with a finite number of operations that tests whether a diophantine equation (polynomial equation with integer coefficients and n variables) has an integer solution.

  Remark 1: Formulating Hilbert's problem with today's terminology: Is there an algorithm to test whether a diophantine equation has an integer solution? (If yes, give the algorithm.) Or equivalently, is there a Turing machine to decide language $D = \{< d > | d \text{ is a diophantine equation with integer solution}\}$?

  Remark 2: In 1970, it was proved by Matijasevic (at the age of 22) that the problem is unsolvable even for $n = 13$. In 1978, it was proved to be **NP**-complete for quadratic equation $ax^2 + by = c$.

1.2.2 The 1930s and 1940s

- Turing (in 1936 at the age of 24, but died at 42, 1912-1954) and Church (1903-1995) independently introduced computational models, Turing machine and lambda calculus, respectively, showed that some well-stated computational problems have no solution, e.g., the halting problem, and demonstrated the existence of universal computing machines, i.e., machines capable of simulating every other machine of their type.
- Other similar work: Herbrand's functions (dies at 23 in a mountain climbing accident, 1908-1931), Post's systems (lost one arm as a child, suffered bipolar disorder, died of heart attack after electric shock treatment in mental hospital, 1897-1954), Godel's functions (at the age of 24 in U. Vienna, (1906-1978), and Kleen's functions (1909-1994).
- The Church-Turing Thesis in today's terminology: Any reasonable attempt to model mathematically computer algorithms and their time performance is bound to end up with a model of computation and associated time cost that is equivalent to Turing machines within a polynomial.
- The earlier version of the Halting Problem (proposed by Hilbert): Is there a "finitary way" to decide the truth or falsity of any mathematical statement? The unsolvability of the problem was proved independently by Turing and Church.
- Development inspired by and in parallel with the work by Turing and Church:
  - The birth of the computers: Z3 in 1941, ENIAC in 1946, and Von Neumann's machine in 1947.
  - The development of languages: FORTRAN, COBOL, LISP in the 1950s.

1.2.3 The 1950s

- The formalization of the finite automata, pushdown automata, and linear bounded automata occurred in parallel with the development of languages, followed by the study of the capabilities and limitations of these models.
- Starting in the late 1950s and continuing in the 1960s was an explosion of research on formal languages. The Chomsky language hierarchy, consisting of the regular, context-free, context-sensitive, and recursively enumerable languages, was established, as was the correspondence of these languages and the machines recognizing them, namely the finite automaton, the pushdown automaton, the linear-bounded automaton, and the Turing machine.

1.2.4 The 1960s

- The birth of Complexity Theory: Are some problems much harder than others? If so, what are they? How much harder are they? If they are harder, what can we do?
- The work by Hartmanis, Lewis, and Stearns, and later by Rabin and Blum on classification of problems by the resources used in solving them set the foundation of computational complexity, leading to many important discoveries in the early 1970s.
- Problem tractability/intractability was established by Edmonds and independently Cobham. They observed that a number of problems are apparently harder to solve, yet have solutions that are easy to verify. Their work eventually led to the introduction of the class **NP**. (Indeed, the class **NP** was mentioned much earlier by Godel in a letter to von Neumann!)
- Opposite to intractable problems, a problem is called tractable if it has a polynomial-time solution. Why polynomial? Three reasons: closure properties of polynomials; polynomially-related models of computation; and in general an exponential-time algorithm is only feasible for small inputs, with exceptions such as the simplex algorithm for LP.

1.2.5 The 1970s

- Cook and independently Levin established the notion of **NP**-complete languages, the languages associated with many hard combinatorial optimization problems, and proved the first **NP**-complete problem SATISFIABILITY (SAT).
- Karp was instrumental in demonstrating the importance of NP-complete languages with a list of 21 **NP**-complete problems. The question whether any NP-complete language can be recognized in polynomial time in their length resulted in the most famous open problem in computer theory, i.e., is P equal to NP?
- The connection between computation time on Turing machines and circuit complexity was established, opening up a new direction for prove the P=NP? problem.
- Garey and Johnson's book and Johnson's NP-completeness column on Journal of Algorithms.
- The study of programming language semantics started its very productive period, with models and denotations for language development and formal methods for program correctness proofs.
- Algorithms (models for programs) and data structures experienced a flowering, pioneered by Knuth, Aho, Hopcroft, and Ullman. New algorithms were invented for sorting, data storage and retrieval, problems on graphs, polynomial evaluation, solving linear systems of equations, and computational geometry.

1.2.6 The 1980s and 1990s

- New topics for theoretical computer science emerged, such as models and complexity classes for parallel computing, quantum computing, and DNA computing.
- The development of networks of computers brought to light some hard logical problems that led to a theory of distributed computing, including efficiently coordinating activities of agents, handling processor failure, and reaching consensus in the presence of adversaries.
- Cryptography became a serious and active research area in the 90s, with the emergence of e-commerce.
- A new proof model is emerged, where proofs are provided by one individual/process and checked by another, i.e., a proof is a communication tool designed to convince someone else of the correctness of a statement. A big shift from the traditional proof methodology. The most striking finding is that a large class of polynomially long proofs can be checked with high probability of success by reading only a fixed number of characters selected randomly from the proof. Most of the proof is unread, yet the verifier can assert with high probability that it is correct!

2 Preliminaries

2.1 Algorithms and complexity

Most courses on algorithms end with a brief discussion on computability and NP-completeness. So it is fitting to start this course by recounting some basic facts about algorithms.

- An algorithm is a well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output.
- Asymptotic notation:
  - $f(n) = O(g(n))$ iff $\exists c$ and n_0 s.t. $f(n) \le cg(n)\ \forall n \ge n_0$.
  - $f(n) = \Omega(g(n))$ iff $g(n) = O(f(n))$.
  - $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
  - Important facts: $(\log n)^k = O(n^d)$ and $n^d = O(c^n)$ $(c > 1)$.
- Polynomial paradigm
  - $p(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0 = \Theta(n^d)$: Polynomial/tractable/efficient.
  - $q(n) \ne O(n^d), \forall d$: Nonpolynomial/exponential/intractable.
  - Why polynomial?
    — $n^d = O(c^n)$ $(c > 1)$. (Controversial: n^{80} versus $2^{\frac{n}{100}}$)
    — A stable class: Closed under +/-/*.
    — TM, RAM: Polynomially related.
- To achieve time efficiency in algorithms, algorithm design techniques, such as divide-and-conquer, greedy method, dynamic programming, may be used in combination with advanced data structures.

2.2 Languages

- Alphabet Σ: A finite and nonempty set of symbols. For example, $\Sigma = \{0,1\}$ and $\Sigma = \{a, b, \ldots, z\}$.
- String (or word): A finite sequence of symbols chosen from some alphabet. The empty string ε is a string with zero occurrences of symbols. For string w, the length of the string, $|w|$, is the number of positions for symbols in the string. $|\varepsilon| = 0$. If $w_1 = a_1 \cdots a_n$ and $w_2 = b_1 \cdots b_m$, then the concatenation $w_1 w_2 = a_1 \cdots a_n b_1 \cdots b_m$. If $w = a_1 \cdots a_n$, then the reverse $w^R = a_n \cdots a_1$. A string x is a substring of w if x appears consecutively within w. The empty string ε is a substring of any string.
- Language: A language is a set of strings. Some special languages include $\{\varepsilon\}$ and $\emptyset$. Other examples of languages are $A = \{w | w \text{ has an equal number of 0s and 1s}\}$ and $B = \{0^n 1^n | n \ge 1\}$. An alphabet Σ can be treated as a language.
- Common operators: Let A and B be two languages.
  - Union: $A \cup B = \{x | x \in A \text{ or } x \in B\}$.
  - Concatenation: $AB = \{xy | x \in A \text{ and } y \in B\}$.
  - Star: $A^* = \{x_1 x_2 \cdots x_k |\text{ all } k \ge 0 \text{ and each } x_i \in A\}$. A^* is infinite unless $A = \emptyset$ or $A = \{\varepsilon\}$.
  - Other operators: Intersection, complement, difference, and reversal.
- Power of a language: Let A be a language.
  - $A^k = \{x_1 x_2 \cdots x_k | x_i \in A \text{ for } i = 1, 2, \cdots, k\}$
  - $A^0 = \{\varepsilon\}$
  - $A^* = A^0 \cup A^1 \cup A^2 \cup \cdots$
  - $A^+ = A^1 \cup A^2 \cup \cdots$
  - $A^* = A^+ \cup \{\varepsilon\}$
- Why languages, not problems:
  - Meta claim: Every computational problem has a decision version of roughly equal computational difficulty.
  - Decision problems: Given an input x, does x satisfy property P, or is $x \in \{y | y \text{ satisfies } P\}$?
  - Membership in a language: Given a language A and a string $w \in \Sigma^*$, is w a member of A, or is $w \in A$?

2.3 Proof techniques essential in this class

- Direct proof: To prove "if p then q", assume p is true and show that q is true.
- By counterexample: Used to prove that a statement is not true.

  Example: There is no integers a and b such that $a \bmod b = b \bmod a$. (Choose $a = b$.)
- Proof by cases: When we are unable to prove a theorem with a single argument that hold for all possible cases, we may need to construct our proof for each of the possible cases, one by one. The rule of inference behind this method lies in the equivalence of $(p_1 \vee p_2 \vee \cdots \vee p_n) \to q$ and $(p_1 \to q) \wedge (p_2 \to q) \wedge \cdots \wedge (p_n \to q)$.
- Existence proof: To prove there exists an object of a certain type, we can simply construct one. This is the constructive existence proof. It is also possible to give a existence proof that is nonconstructive or not totally constructive. Sometimes a nonconstructive existence proof uses proof by contradiction.

  Example: Prove that there is a positive integer that can be written as the sum of cubes of positive integers in two different ways.

  Proof: $1729 = 10^3 + 9^3 = 12^3 + 1^3$.

  Example: Prove that there exist irrational numbers x and y such that x^y is rational.

  Proof: Consider the irrational number $\sqrt{2}$ and make a number $z = \sqrt{2}^{\sqrt{2}}$. If z is a rational number, then we have found x and y. However, if z is irrational, then let $x = z$ and $y = \sqrt{2}$ so that $x^y = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$, a rational!
- By contradiction: There are three variations based on the following logic identities.
  - $(p \to q) \leftrightarrow (\neg q \to \neg p)$ (This is also called proof by contraposition.)
  - $(p \to q) \leftrightarrow (\neg q \wedge p \to r \wedge \neg r)$ for some r
  - $p \leftrightarrow (\neg p \to r \wedge \neg r)$ for some r

  Example: The number of primes is infinite.

  Example (Sisper p. 22): $\sqrt{2}$ is irrational.
- Proof by induction: To prove that predicate $P(n)$ is true for all $n \geq 1$, there are two variations of induction:
  - Prove that (1) $P(1)$ is true and (2) $P(n-1) \to P(n)$ is true. (Simple integer induction)
  - Prove that (1) $P(1)$ is true and (2) $P(1) \wedge \cdots \wedge P(n-1) \to P(n)$ is true. (General integer induction)

  Example: Prove that $1 + 2 + \cdots + n = \frac{1}{2}n(n+1)$ for all $n \geq 1$.

  Example: Prove that $T(n) = \frac{1}{2}n^{\frac{1}{3}}((\log_8 n)^2 + \log_8 n + 2)$ if $n = 8^k$ and

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\frac{n}{8}) + n^{\frac{1}{3}} \log_8 n & \text{if } n > 1 \end{cases}$$

- By reduction: A key technique in both computability and complexity theory and a central concept in this course.
- By diagonalization: Unique in computability theory.

  (1) Diagonalization and uncountable sets: Prove by contradiction. Assume the set is countable and therefore its members can be exhaustively listed. The contradiction is achieved by producing a member of the set that cannot occur anywhere in the list.

  Example: Some countable sets: N (natural numbers), E (even numbers), and Q (rational numbers).

  Example (Sispser p. 205): R, the set of real numbers, is uncountable.

  Example: The set $F = \{f : \mathbf{N} \to \mathbf{N}\}$ is uncountable.

  (2) Diagonalization and self-reference: The method is used to demonstrate that certain properties are inherently contradictory, in particular in nonexistence proofs that no object satisfies such a property. Diagonalization proofs of nonexistence frequently depend on contradictions that arise from self-reference: an object analyzing its own actions, properties, or characteristics. Russell's paradox, the undecidability of the Halting Problem, and Godel's proof of the undecidability of number theory are all based on contradictions associated with self-reference.

Example: (The barber's paradox) Prove that the following statement is false: "In a town, every man who is able to shave himself does so and the barber of the town (a man himself) shaves all and only those who cannot shave themselves."

Proof: Let $M = \{p_1, p_2, \ldots\}$ be the set of all males in the town. Assume that the given statement is true. Then a table T can be constructed, where $T[i, j] = 1$ if p_i shaves p_j and $T[i, j] = 0$ is p_i does not shave p_j. Assume that the barber is p_k. Consider $T[k,k]$. If $T[k,k] = 1$, then the barber shaves himself. This is in contradiction to that the barber only shaves those who cannot shave themselves. If $T[k,k] = 0$, then the barber does not shave himself. This is in contradiction to that any man who is able to shave himself does so.

Example: (The Russell's paradox) Prove that Cantor's statement is false: Any property defines a set which contains all objects that satisfy the property.

Proof: Consider the property that a set is not a member of itself. If Cantor's statement is true, then this property defines a set $S = \{X | X \notin X\}$, which contains all sets that are not members of themselves. Clearly, $S \neq \emptyset$ since $\{a\} \in S$. Since S is a set itself, we may ask if S is not a member of S. If S is a member of itself, then $S \in S$, so $S \notin S$ by definition. If S is not a member of itself, then $S \notin S$, so $S \in S$ by definition. A contraction in each case.

Where is diagonalization in the above proof? Consider the membership relation between any two sets S_i and S_j. A table M can be used to define the relation, where $M[S_i, S_j] = 1$ if S_i is a member of S_j and $M[S_i, S_j] = 0$ otherwise. Therefore the complement of the diagonal of the table defines S, i.e., $S_i \in S$ if and only if $M[S_i, S_i] = 0$. To get the contradiction, ask $M[S, S] = ?$.

Example: Another diagonalization proof given by Cantor that a nonempty set S cannot be placed into a one-to-one correspondence with its power set 2^S.

Proof: Assume the opposite, i.e., there is a bijection function f such that for any $x \in S$, we can uniquely associate it with $f(x) \subseteq S$ (or equivalently $f(x) \in 2^S$). Consider the relation between x and $f(x)$, either $x \in f(x)$ or $x \notin f(x)$. Define $A \subseteq S$ such that $A = \{x | x \notin f(x)\}$. There must be y such that $f(y) = A$. Is $y \in f(y)$? If $y \in f(y)$ then $y \in A$ since $f(y) = A$. Thus, $y \notin f(y)$ by the definition of A. If $y \notin f(y)$ then $y \in A$ by the definition of A. Thus $y \in f(y)$ since $f(y) = A$. So $y \in f(y)$ iff $y \notin f(y)$. A contradiction in each case.

Where is diagonalization in the above proof?

3 Turing Machines

The question of what computers can do/solve, or equivalently, what languages can be defined/recognized by any computational device whatsoever.

3.1 Problems that computers cannot solve

- Two types of problems: "Solve this" and "decide this".
- Decision problems (the "decide this" type) have a yes/no solution. They are usually just as hard as their "solve this" version in the sense of dealing with important questions in computability/complexity theory.
- A problem is said to be unsolvable/undecidable if it cannot be solved/decided using a computer.
- Recall that a decision problem is really about membership of strings in a language. For example, the problem of primality testing is actually the language of all prime numbers in binary representation.
- The number of problems/languages over an alphabet with more than one symbol is uncountably infinite. However, the number of programs that a computer may use to solve problems is countably infinite. Therefore, there are more problems than there are programs. Thus, there must be some undecidable problems. We will prove this statement rigorously later.
- An undecidable problem (the halting problem):
  - Input: Any program P and any input I to the program;
  - Output: "Yes" if P terminates on I and "No" otherwise.

3.2 Basics

- A Turing machine includes a control unit, a read-write head, and a one-way infinite tape (i.e., unlimited and unrestricted memory).
- TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where
  - Q: The finite set of states for the control unit.
  - Σ: An alphabet of input symbols, not containing the "blank symbol" B.
  - Γ: The complete set of tape symbols. $\Sigma \cup \{B\} \subset \Gamma$.
  - δ: The transition function from $Q \times \Gamma$ to $Q \times \Gamma \times D$, where $D = \{L, R\}$.
  - q_0: The start state.
  - q_{accept}: The accept state.
  - q_{reject}: The reject state.

  Remarks: (1) δ may also be defined as $Q - \{q_{accept}, q_{reject}\} \times \Gamma \to Q \times \Gamma \times D$. (2) If a TM M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move. (3) Initially the tape contains only th input string and is blank (B) everywhere else.
- Configuration: $X_1 \cdots X_{i-1} q X_i \cdots X_n$; is a configuration (snapshot) of the TM in which q is the current state, the tape content is $X_1 \cdots X_n$, and the head is scanning X_i.
  - If $\delta(q, X_i) = (p, Y, L)$, then $X_1 \cdots X_{i-1} q X_i \cdots X_n \vdash X_1 \cdots X_{i-2} p X_{i-1} Y X_{i+1} \cdots X_n$.
  - If $\delta(q, X_i) = (p, Y, R)$, then $X_1 \cdots X_{i-1} q X_i \cdots X_n \vdash X_1 \cdots X_{i-1} Y p X_{i+1} \cdots X_n$.
- The start configuration $q_0 w$, accepting configuration $u q_{accept} v$, and rejecting configuration $u q_{reject} v$, where the latter two are called the halting configurations.
- Language of a Turing machine M (or language recognized/accepted by M) is $L(M) = \{w \in \Sigma^* | q_0 w \overset{*}{\vdash} \alpha q_{accept} \beta \text{ for } \alpha, \beta \in \Gamma^*\}$.
- For any given input, a TM has three possible outcomes: accept, reject, and loop. Accept and reject mean that the TM halts on the given input, but loop means that the TM does not halt on the input.

- A language A is Turing-recognizable (or recursively enumerable) if there is a TM M such that $A = L(M)$, or equivalently, A is the language of some TM M. In other words, $\forall w \in A$, M accepts w by entering q_{accept}. However, $\forall w \notin A$, M may reject or loop.
- A language A is Turing-decidable (or decidable, or recursive) if there is a TM M such that $A = L(M)$ and M halts on all inputs. In other words, $\forall w \in A$, M accepts w and $\forall w \notin A$, M rejects w. Such TM's are a good model for algorithms.
- Definition of the output of M on w, denoted by $M(w)$: $M(w) =$"accept", "reject", the string when q_{accept} is entered, or "undefined" if M never halts.
- There are several ways to describe a TM:

  (1) Formal description: List all moves in δ (using state diagram or transition table). The lowest and most detailed level of description.

  Example: Give a TM M with $L(M) = \{0^n 1^n | n \geq 1\}$. (Or give a TM M that to accepts/recognizes $\{0^n 1^n | n \geq 1\}$.)

  (2) Implementation description: Use English prose to describe TM operations. Similar to pseudocode, easier to follow, but hides details.

  Example (Sipser p. 171): Give a TM M that decides $A = \{0^{2^n} | n \geq 0\}$.

  (3) High-level description: Use English prose to describe an algorithm, ignoring the implementation details.

  Example (Sipser p. 174): Give a TM M that decides $C = \{a^i b^j c^k | i \times j = k \text{ and } i, j, k \geq 1\}$.

  Example (Sipser p. 182): Revisit Hilbert's tenth problem.

3.3 Properties of TDLs and TRLs

Theorem: A Turing-decidable language is also Turing-recognizable, but not vice versa.
Theorem: A and $\overline{A}$:

- If A is Turing-decidable, so is $\overline{A}$. (That is, TDLs are closed under complementation.)
- If A and $\overline{A}$ are both Turing-recognizable, then A is Turing-decidable.
- For any A and $\overline{A}$. we have one of the following possibilities: (1) Both are Turing-decidable; (2) Neither is Turing-recognizable; (3) One is Turing-recognizable but not decidable, the other is not Turing-recognizable.

Additional properties: TRLs and TDLs both are closed under union, intersection, concatenation, and star. See Sipser p. 189, problems 3.15 and 3.16.

3.4 Time and space bounds of TMs

We define a variation of TM, called TM with k-strings (and naturally k-heads), where $\delta : Q \times \Gamma^k \to Q \times (\Gamma \times D)^k$. It can be proved that this is an equivalent model to Turing machines.
Time bound:

- Use TM with k-strings as the model (quadratic speed-up compared to basic TM) and count the number of steps.
- TM M has time bound $f(n)$ if for any input x, with $|x| = n$, M requires at most $f(n)$ steps to decide (accept/reject) x.
- Complexity class $\textbf{TIME}(f(n))$ is a set of all languages decidable by TMs within time bound $f(n)$.
- Constants and low-order terms don't count: Use big-O, e.g., $\textbf{TIME}(2n^2) = \textbf{TIME}(n^2 + 3n + 1) = \textbf{TIME}(n^2)$.
- $\textbf{P} = \cup_{k \geq 0} \textbf{TIME}(n^k)$.

Space bound:

- Use TM with k-strings as the model and compute $\max_j \sum_{i=1}^{k} |w_i^j|$ (overcharge) or alternatively use TM with k-string with input and output (The first string is read-only and the last string is write-only.) and compute $\max_j \sum_{i=2}^{k-1} |w_i^j|$. Note that w_i^j is the ith string on the tape in the jth step of the computation of M.
- TM M has space bound $f(n)$ if for any input x, with $|x| = n$, M requires at most $f(n)$ tape squares to decide x.
- Complexity class $\textbf{SPACE}(f(n))$ is a set of all languages decidable by TMs within space bound $f(n)$.
- Constants and low-order terms don't count: Use big-O.
- $\textbf{PSPACE} = \cup_{k \geq 0} \textbf{SPACE}(n^k)$.

3.5 Nondeterministic Turing machine (NTM)

- An unrealistic (unreasonable) model of computation which can be simulated by other models with an exponential loss of efficiency. It is a useful concept that has had great impact on the theory of computation.
- NTM $N = (Q,\Sigma,\Gamma,\delta,q_0,q_{accept},q_{reject})$, where $\delta : Q \times \Gamma \to 2^P$ for $P = Q \times \Gamma \times \{L,R\}$.
- $\delta(q,X)$ is a set a moves. Which one to choose? This is nondeterminism. The computation can be illustrated by a tree, with each node representing a configuration.
- Nondeterminism can be viewed as a kind of parallel computation wherein multiple independent processes or threads can be running concurrently. When a nondeterministic machine splits into several choices, that corresponds to a process forking into several children, each then proceeding separately. If at least one process accepts, then the entire computation accepts.
- N accepts L if $\forall x \in L$, there is a sequence of moves that eventually leads to q_{accept}, i.e., there is a path from the root to a leaf corresponding to an accepting configuration in the tree. A rejecting state at a node only terminates the expanding of the node (thus making the node a leaf) but does not terminate the computation of the tree.
- N decides L if N accepts L and further $\forall x \in \Sigma^*$, all paths in the computation tree halt.
- N decides L in time $f(n)$ if N decides L, and $\forall x \in \Sigma^*$ with $|x| = n$, N does not have computation paths longer than $f(n)$, i.e., the height of the computation tree for x is no longer than $f(n)$.
- Complexity class **NTIME**$(f(n))$ is a set of languages decidable by NTM within time bound $f(n)$.
- **NP**$= \cup_{k\geq 0}$**NTIME**(n^k).
- **P**$\subseteq$**NP** since determinism is a special case of nondeterminism.
- Any NTM with time bound $f(n)$ can be simulated by a DTM with time bound $O(c^{f(n)})$ for some $c > 1$. This is consistent with the Church-Turing Thesis. However, we don't know whether **NP**$\subseteq$**P**.
- Example: TSP (DEC) is in **NP**.

  INSTANCE: An edge-weighted, undirected, and complete graph $G(V,E,w)$ and a bound $B \geq 0$. (The weight on an edge is allowed to be $+\infty$.)

  QUESTION: Is there a tour (a cycle that passes through each node exactly once) in G with total weight no more than B?

  A NTM N that decides TSP in polynomial time can be defined as follows: Assume that a coded instance of TSP is shown on the tape as input. N first generates an arbitrary permutation of the n nodes and then checks whether the permutation represents a tour with total weight no larger than the bound B. If so, N accepts the input. If not. N rejects the input. Note that the permutation generated can be any of the $n!$ possible permutations. The nondeterminism of the machine guarantees that if the instance is a yes-instance, the right permutation will always be selected.

  Note: The acceptance of an input by a nondeterministic machine is determined by whether there is an accepting computation among all, possibly exponentially many, computations. In the above proof, if there is a solution, it will always be generated by the Turing machine. This is like that a nondeterministic machine has a guessing power. A tour can only be found by a deterministic machine in exponential time, however, it can be found by a nondeterministic machine in just linear steps of the tour length. So any nondeterministic proof should always contain two stages: Guessing and verifying.
- Nondeterministic space bound: Assume that in NTM N all computation paths halt on all inputs (i.e., NTM as a decider). Then its space bound is defined, as a function of n, to be the maximum number of tape squares that N scans on any path of its computation for any input of length n.
- Complexity class **NSPACE**$(f(n))$ is a set of languages decidable by NTM within space bound $f(n)$.
- **NPSPACE**$= \cup_{k\geq 0}$**NSPACE**(n^k).
- NTM is not a true model of computation. However, nondeterminism is a central concept in complexity theory because of its affinity not so much with computation itself, but with the applications of computation, most notably logic and combinatorial optimization. For example, a sentence in logic may try all proofs, but it succeeds in becoming a theorem if any of them works. Also, a typical problem in combinatorial optimization seeks one solution among exponentially many that has the lowest cost. A nondeterministic machine can search an exponentially large space quite effortlessly.

3.6 Variants of TM's

- TM with multi-tapes (and thus multi-heads) ($\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L,R\}^k$).

  Theorem (Sipser p. 177): Every multi-tape Turing machine has an equivalent single-tape Turing machine.

  Corollary (Sipser p. 178): A language is Turing-recognizable (Turing-decidable) if and only if some multi-tape Turing machine recognizes (decides) it.

- Nondeterministic TM ($\delta : Q \times \Gamma \to 2^{Q\times\Gamma\times D}$).

  Theorem (Sipser p.178): Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

  Corollary (Sipser p. 180): A language is Turing-recognizable (Turing-decidable) if and only if some nondeterministic Turing machine recognizes (decides) it.

- TM with multi-strings (and thus multi-heads).
- TM with multi-heads.
- TM with multi-tracks.
- TM with two-way infinite tape.
- TM with multi-dimensional tape.

Theorem: (The equivalent computing power of the above TM's)
For any language L, if $L = L(M_1)$ for some TM M_1 with multi-tapes, multi-strings, multi-heads, multi-tracks, two-way infinite tape, multi-dimensional tape, or nondeterminism, then $L = L(M_2)$ for some basic TM M_2.
Theorem: (The equivalent computing speed of the above TM's except for nondeterministic TM's)
For any language L, if $L = L(M_1)$ for some TM M_1 with multi-tapes, multi-strings, multi-cursors, multi-tracks, two-way infinite tape, or multi-dimensional tape in a polynomial number of steps, then $L = L(M_2)$ for some basic TM M_2 in a polynomial number of steps (with a higher degree). Or in other words, all reasonable models (except NTM) of computation can simulate each other with only a polynomial loss of efficiency.
Note: The speed-up of a nondeterministic TM vs. a basic TM is exponential.
The Church-Turing Thesis: Any reasonable attempt to model mathematically algorithms and their time performance is bound to end up with a model of computation and associated time cost that is equivalent to Turing machines within a polynomial. Or, any computation that can be carries out by mechanical means can be performed by some Turing machine.

4 Undecidability

4.1 Encoding schemes

- Turing machine (as a decider) $\Leftrightarrow$ algorithm; string $\Leftrightarrow$ instance.
- Reasonable encoding schemes: Binary or k-ary ($k \geq 3$) representation without unnecessary information or symbols "padded". (Why not unary? For number n, its unary representation is n-bit long while its binary representation is $\log n$-bit long. An exponential difference.)
- Any two reasonable encoding schemes are polynomially related. Let i be any instance and e_1 and e_2 be two reasonable encoding schemes. If $|e_1(i)| = O(n)$, then $|e_2(i)| = O(p(n))$, where $p(n)$ is a polynomial.
- An algorithm having polynomial time complexity under one reasonable encoding scheme is also polynomial under another reasonable encoding scheme.
- A binary encoding scheme for TM:

  $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ with q_1 to be the start state, q_2 to be the accept state, and q_3 to be the reject state.

  $\Gamma = \{X_1, X_2, \ldots, X_{|\Gamma|}\}$.

  $D = \{D_1, D_2\}$ with D_1 to be L and D_2 to be R.

  A transition $\delta(q_i, X_j) = (q_k, X_l, D_m)$ is coded as $0^i 1 0^j 1 0^k 1 0^l 1 0^m$. A TM is coded as $C_1 11 C_2 11 \cdots 11 C_n$, where each C is the code for a transition.

  TM M with input w is represented by $< M, w >$ and coded as $M111w$.
- With encoding schemes, any instance (e.g., a graph, an array of numbers, a grammar, a finite automaton, a Turing machine) can be encoded into a string and fed as input to a Turing machine.

4.2 Existence of non-Turing-recognizable languages

- **Theorem** (Sipser p. 206): Some languages are not Turing-recognizable.

  Proof: Let T be the set of all TMs and L be the set of all languages. For simplicity, we fix the alphabet to be Σ.

  We first observe that the set of all strings is countable since the strings can be listed alphabetically (or in lexicographic order). Consider the set T of all TMs. Since each TM M has a string encoding of $< M >$, we can omit those strings that are not legal encodings of TMs and obtain a list of all TMs. Thus set T is countable.

  We next observe that the set B of all infinite binary sequences is uncountable (HW 2/Problem 1). We now show that set L is uncountable by giving a correspondence with B. Let $\Sigma^* = \{s_1, s_2, \ldots\}$. Each language A in L has a unique sequence in B, where the ith bit in that sequence is a 1 if $s_i \in A$ and is a 0 if $s_i \notin A$. This correspondence between a language in L and an infinite binary sequence in B is one-to-one and onto. Thus set L and set B have the same size. So L is uncountable.

  Since T is countable and L is uncountable, there are more languages than there are TMs. Then there must be some languages that are not Turing-recognizable.
- A language that is not Turing-recognizable:
  - Enumerating binary strings: $\varepsilon, 0, 1, 00, 01, 10, 11, \cdots$. Let w_i denote the ith string in the above lexicographic ordering.
  - Let the ith TM, M_i, be the TM whose code is w_i, the ith binary string. If w_i is not a valid TM code, then let M_i be the TM that immediately rejects any input, i.e., $L(M_i) = \emptyset$.
  - Define the diagonalization language $A_D = \{w_i | w_i \notin L(M_i)\}$. Construct a boolean table, where the (i, j) entry indicates whether TM M_i accepts string w_j. Then language A_D is made by complementing the diagonal.
  - A_D is not Turing-recognizable.

    Proof: Suppose, by contradiction, there is a TM M such that $A_D = L(M)$. Then $M = M_d$ with code w_d for some d. $w_d \in A_D$ iff $w_d \notin L(M_d)$ by definition of A_D. $w_d \in A_D$ iff $w_d \in L(M_d)$ by $A_D = L(M_d)$. A contradiction.

4.3 The Halting problem

- The universal Turing machine:
  - Each TM (among those discussed) can only solve a single problem (or accept a single language), however, a computer can run arbitrary algorithms. Can we design a general-purposed TM that can solve a wide variety of problems?
  - Theorem: There is a universal TM U which simulates an arbitrary TM M with input w and produces the same output as M.

    $U =$ On input $< M, w >$, where M is a TM and w is a string:

    1. Simulate M on input w.

    2. If M enters q_{accept}, accept; if M enters q_{reject}, reject.

    Remark: Step 2 implies that if M loops, U also loops. But this case cannot be defined explicitly.
  - TM U is an abstract model for computers just as TM M is a formal notion for algorithms.

- The universal language: Turing-recognizable but not Turing decidable

  Let $A_{TM} = \{< M, w > | M$ is a TM and M accepts string $w\}$

  A_{TM} is Turing-recognizable since it can be recognized by TM U. A_{TM} is called the universal language.

  A_{TM} is not Turing-decidable.

  Proof: Assume that A_{TM} is decided by TM T, Then on input $< M, w >$, T accepts if $< M, w > \in A_{TM}$ and T rejects if $< M, w > \notin A_{TM}$. That is, T accepts if M accepts w and T rejects if M does not accept w. Or simply, T accepts iff M accepts w.

  Define TM D, which on input $< M >$, runs T on input $< M, < M >>$ and accepts iff T rejects $< M, < M >>$.

  Feed $< D >$ to D. We see that D accepts $< D >$ iff T rejects $< D, < D >>$ by the definition of D. On the other hand, T rejects $< D, < D >>$ iff D does not accept $< D >$ by the definition of T. Combining the two statements, we get D accepts $< D >$ iff D does not accept $< D >$. A contradiction.

  Diagonalization is used in this proof. Why?

- The halting language: Turing-recognizable but not Turing decidable

  Let $HALT_{TM} = \{< M, w > | M$ is a TM and M halts on string $w\}$.

  $HALT_{TM}$ is Turing-recognizable since it can be recognized by TM U. $HALT_{TM}$ is called the halting language.

  $HALT_{TM}$ is not Turing-decidable.

  Proof: Suppose there is a TM H that decides $HALT_{TM}$. Then on input $< M, w >$, H accepts iff M halts on w.

  Define TM D, which on input $< M >$, runs H on $< M, < M >>$ and accepts if H rejects and loops if H accepts.

  Feed $< D >$ to D. We see that D halts on $< D >$ iff H rejects $< D, < D >>$ iff D does not halt on $< D >$. A contradiction.

4.4 Reducibility

- Reduction is a way of converting one problem to another in such a way that a solution to the second problem can be used to solve the first. We say that problem A reduces (or is reducible) to problem B, if we can use a solution to B to solve A (i.e., if B is decidable/solvable, so is A.).

  We may use reducibility to prove undecidability as follows: Assume that we wish to prove B (a language or a decision problem) to be undecidable and we know that A has already been proved undecidable. We use contradiction. Assume B is decidable. Then there must exist a TM M_B that decides B. If we can use M_B as a sub-routine to construct a TM M_A that decides A, then we have a contradiction. The construction of TM M_A using TM M_B establishes that A is reducible to B. The key in this method is to choose a known undecidable A and figure out the conversion from the input of A/M_A to the input of B/M_B and the conversion from the output of B/M_B to the output of A/M_A.

- A variation of proving reducibility:

  Assume that A is the decision problem (language) we know to be undecidable and B is the decision problem (language) we wish to prove to be undecidable. We prove that A reduces, or is reducible, to B. To do so, we just show that for any instance (string) I_A of A, we can construct an instance (string) I_B of B such that I_A is a yes-instance (string I_A is in language A) iff I_B is a yes-instance (string I_B is in language B).

- Another proof that A_{TM} is not decidable:

  Recall $A_{TM} = \{<M,w> | w \in L(M)\}$. Recall $A_D = \{w_i | w_i \notin L(M_i)\}$.

  Assume that A_{TM} is decidable. Let M_{TM} be the TM that decides A_{TM}. We will construct a TM M_D that would decide A_D, an undecidable language. M_D works as follows: For input w_i (the ith binary string in the lexicographic sequence of all binary strings), it first makes a string $w_i 111 w_i$ and then feedd it to M_{TM}. We notice that what is fed to M_{TM} is actually $<M_i, w_i>$ (recall that M_i is the TM with code w_i) and that $<M_i, w_i> \in L(M_{TM})$ iff $w_i \in L(M_i)$ iff $w_i \notin L(M_D)$. So M_{TM} accepts $<M_i, w_i>$ iff M_D rejects w_i.

  We just proved that A_D is reducible to A_{TM}.

- Another proof that $HALT_{TM}$ is not decidable:

  We will reduce A_{TM} to $HALT_{TM}$. Assume TM R decides $HALT_{TM}$. We construct TM S that decides A_{TM} as follows: On input $<M,w>$ where M is a TM and w is a string, S first run TM R on $<M,w>$, if R rejects, rejects. If R accepts, simulate M on w until it halts. If M accepts, accept; if M rejects, reject.

- It is undecidable whether a TM M (input) halts on all inputs. (Equivalently, language $ALL_{TM} = \{<M>: TMM$ halts on all inputs$\}$ is not recursive.)

  Proof 1. Assume that it is decidable. Then there is a TM P which takes $<M>$ as input and decides whether M halts on all inputs. We will show that we can use P to define a new TM Q which solves the halting problem. Here is how Q works: Given $<M,w>$, Q first constructs TM M', which on input string x, runs M on w if $x = w$ and halts immediately if $x \neq w$. Then Q feeds $<M'>$ to P. Obviously, M' halts on all inputs iff M halts on x. So Q accepts iff P accepts.

  Proof 2. We reduce the halting problem to this problem. Given any instance M and w of the halting problem, we define an instance M' (whose definition is given in proof 1) of our problem. Then we show that M halts on w iff M' halts on all inputs.

- More examples:

  - (Sipser p. 217) $E_{TM} = \{<M> | M$ is a TM and $L(M) = \emptyset\}$ is undecidable.
  - (Sipser p. 220) $EQ_{TM} = \{<M_1, M_2> | M_1$ and M_2 are TMs and $L(M_1) = L(M_2)\}$ is undecidable.

- Rice's Theorem: Let $\mathcal{C}$ be a proper, non-empty subset of the set of all Turing-recognizable languages. Then the following problem is undecidable: Given a TM M, is $L(M) \in \mathcal{C}$, where $L(M)$ is the language accepted by M?

  The above theorem implies that any nontrivial property of Turing machines is undecidable.

4.5 Post's correspondence problem (PCP)

- We formulate the Post's Correspondence Problem as a puzzle.

  Post's Correspondence Problem (PCP)

  INSTANCE: $P = \{\frac{t_1}{b_1}, \frac{t_2}{b_2}, \ldots, \frac{t_k}{b_k}\}$, where $t_1, t_2, \ldots, t_k$ and $b_1, b_2, \ldots, b_k$ are strings over alphabet Σ. (P can be regarded as a collection of dominos, each containing two strings, with one stacked on top of the other.)

  QUESTION: Does P contain a match, i.e., $i_1, i_2, \ldots, i_l \in \{1, 2, \ldots, k\}$ with $l \geq 1$ such that $t_{i_1} t_{i_2} \cdots t_{i_l} = b_{i_1} b_{i_2} \cdots b_{i_l}$?

  Equivalently, defined as a language, we have $L_{PCP} = \{<P> | P$ is an instance of PCP with a match$\}$.

- For example, for $P_1 = \{\frac{b}{ca}, \frac{a}{ab}, \frac{ca}{a}, \frac{abc}{c}\}$, sequence $2,1,3,2,4$ indicates a match. For $P_2 = \{\frac{abc}{ab}, \frac{ca}{a}, \frac{acc}{ba}\}$, there is no match.

- Modified PCP (MPCP).

  The same as PCP except that the match sequence requires to always start with the first domino.

  Modified Post's Correspondence Problem (MPCP)

  INSTANCE: $P' = \{\frac{t_1}{b_1}, \frac{t_2}{b_2}, \ldots, \frac{t_k}{b_k}\}$, where $t_1, t_2, \ldots, t_k$ and $b_1, b_2, \ldots, b_k$ are strings over the alphabet Σ.

  QUESTION: Does P' contain a match, i.e., $1, i_1, i_2, \ldots, i_l \in \{1, 2, \ldots, k\}$ with $l \geq 0$ such that $t_1 t_{i_1} \cdots t_{i_l} = b_1 b_{i_1} \cdots b_{i_l}$?

  Lemma. If PCP is decidable, then MPCP is also decidable.

  Proof. Let $u = u_1 u_2 \ldots u_n$ be any string of length n. We define

$$*u = *u_1 * u_2 * \ldots * u_n,$$

$$u* = u_1 * u_2 * \ldots * u_n *,$$

$$*u* = *u_1 * u_2 * \ldots * u_n * .$$

To prove the lemma, we only need to prove MPCP reduces to PCP. Assume that symbols $*, \# \notin \Sigma$. Let $P' = \{\frac{t_1}{b_1}, \frac{t_2}{b_2}, \ldots, \frac{t_k}{b_k}\}$ be any instance for MPCP. We define an instance for PCP as $P = \{\frac{*t_1}{*b_1*}, \frac{*t_1}{b_1*}, \frac{*t_2}{b_2*}, \ldots, \frac{*t_k}{b_k*}, \frac{*\#}{\#}\}$.

We observe that any match for P will have to start with the first domino $\frac{*t_1}{*b_1*}$ since it is the only one that both the top and the bottom start with the same symbol, namely $*$. Also, any match for P will have to end with the last domino $\frac{*\#}{\#}$.

The above observation makes it easy to show that P' has a match for MPCP iff P has a match for PCP.

The reduction proof implies that MPCP is no harder than PCP. So if PCP is decidable, so is MPCP, or equivalently, if MPCP is undecidable, so is PCP.

- Theorem. MPCP is undecidable.

  Proof. Recall the universal language $A_{TM} = \{< M, w > | M \text{ accepts } w\}$. Since A_{TM} is undecidable, equivalently, the decision problem whether M accepts w is undecidable. To prove the theorem, we will reduce this decision problem to MPCP. In other words, for any TM M and string w, we will design an instance P' for MPCP to simulate M on w. The intention is to have the concatenation of the strings in the match for P' to represent a sequence of configurations to accept string w.

  Given any TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\}$ and string $w = w_1 w_2 \cdots w_n$, we construct any instance P' for MPCP in following steps.

  Step 1. Put $\frac{\#}{\# q_0 w_1 w_2 \cdots w_n \#}$ into P' as the first domino.

  Step 2. For every $\delta(q,a) = (r,b,R)$, put $\frac{qa}{br}$ into P'.

  Step 3. For every $\delta(q,a) = (r,b,L)$, put $\frac{cqa}{rcb}$ into P' for all $c \in \Gamma$.

  Step 4. For all $a \in \Gamma$, put $\frac{a}{a}$ into P'.

  Step 5. Put $\frac{\#}{\#}$ and $\frac{\#}{\sqcup\#}$ into P'. (The second domino is to add a blank symbol at the end of the configuration to simulate the infinitely many blank symbols to the right that are suppressed when we write the configuration.)

  Step 6. For all $a \in \Gamma$, put $\frac{aq_{accept}}{q_{accept}}$ and $\frac{q_{accept}a}{q_{accept}}$ into P'.

  Step 7. Put $\frac{q_{accept}\#\#}{\#}$ into P'.

  Consider an example to show the simulation.

  Let TM M be a machine that accepts all strings on alphabet $\{a, b\}$ with at least one "b". So the transition function contains moves $\delta(q_0, a) = (q_0, a, R)$, $\delta(q_0, b) = (q_1, b, R)$, and $\delta(q_0, \sqcup) = (q_2, \sqcup, L)$, where q_0 is the start state, q_1 is the accepting state, and q_2 is the rejecting state. Let the input be $w = aab$. Now define the instance for MPCP as follows.

$$\{\frac{\#}{\#q_0aab\#}, \frac{q_0a}{aq_0}, \frac{q_0b}{bq_1}, \frac{aq_0\sqcup}{q_2a\sqcup}, \frac{bq_0\sqcup}{q_2b\sqcup}, \frac{\sqcup q_0\sqcup}{q_2\sqcup\sqcup}, \frac{a}{a}, \frac{b}{b}, \frac{\sqcup}{\sqcup}, \frac{\#}{\#}, \frac{\#}{\sqcup\#}, \frac{aq_1}{q_1}, \frac{bq_1}{q_1}, \frac{\sqcup q_1}{q_1}, \frac{q_1a}{q_1}, \frac{q_1b}{q_1}, \frac{q_1\sqcup}{q_1}, \frac{q_1\#\#}{\#}\}.$$

  For $w = aab$, there is a sequence of configurations to accept w is

$$\#q_0aab\#aq_0ab\#aaq_0b\#aabq_1\#.$$

  A solution for MPCP is given below.

$$\#|q_0a|a|b|\#|a|q_0a|b|\#|a|a|q_0b|\#|a|a|bq_1|\#|a|aq_1|\#|aq_1|\#|q_1\#\#|$$

$$\#q_0aab\#|aq_0|a|b|\#|a|aq_0|b|\#|a|a|bq_1|\#|a|a|q_1|\#|a|q_1|\#|q_1|\#|\#|$$

  In general there is an accepting computation on w in M iff there is a match in the corresponding MPCP. So the universal problem reduces to MPCP. Since the universal problem is undecidable, MPCP is also undecidable. So PCP is undecidable.

- Theorem (Corollary). PCP is undecidable.

- Application of PCP.

  CFG $G = (V, \Sigma, S, R)$, where $R : V \to (V \cup \Sigma)^*$.

  Leftmost derivation. For example, $S \to AA$ and $A \to AAA|bA|Ab|a$. A leftmost derivation: $S \Rightarrow AA \Rightarrow aA \Rightarrow abA \Rightarrow aba$.

  Ambiguity: $\exists w \in L(G)$ with two leftmost derivations. For example, $S \Rightarrow AA \Rightarrow AbA \Rightarrow abA \Rightarrow aba$.

  Theorem. It is undecidable whether any given CFG G is ambiguous. (AMB)

  Proof. Wish to prove PCP reduces to AMB. Assume PCP has an instance $P = \{\frac{t_1}{b_1}, \frac{t_2}{b_2}, \ldots, \frac{t_k}{b_k}\}$. Further assume that $a_1, a_2, \ldots, a_k \notin \Sigma$. Now construct a CFG G which contains the following rules: $S \to S_A|S_B$, $S_A \to t_i S_A a_i | t_i a_i$ for $i = 1, 2, \ldots, k$ and $S_B \to b_i S_B a_i | b_i a_i$ for $i = 1, 2, \ldots, k$. Obviously, $L(G) = L_A \cup L_B$, where $L_A = \{t_{i_1} t_{i_2} \ldots t_{i_l} a_{i_l} \ldots a_{i_2} a_{i_1} | l \geq 1\}$ and $L_B = \{b_{i_1} b_{i_2} \ldots b_{i_l} a_{i_l} \ldots a_{i_2} a_{i_1} | l \geq 1\}$.

  We next prove that the PCP has a match iff G is ambiguous. If PCP has a match, say $i_1, i_2, \ldots, i_l$, $t_{i_1} t_{i_2} \ldots t_{i_l} = b_{i_1} b_{i_2} \ldots b_{i_l}$. Let $u = t_{i_1} t_{i_2} \ldots t_{i_l} a_{i_l} \ldots a_{i_2} a_{i_1} = b_{i_1} b_{i_2} \ldots b_{i_l} a_{i_l} \ldots a_{i_2} a_{i_1}$. Obviously, $u \in L(G)$. In fact, $u \in L_A$ and $u \in L_B$. So u has two leftmost derivations. Therefore G is ambiguous.

  If G is ambiguous, there is $u \in L(G)$ with two leftmost derivations. Any word that can only be derived from S_A or S_B has just one derivation. So u must be derived from both S_A and S_B. Then $u = t_{i_1} t_{i_2} \ldots t_{i_l} a_{i_l} \ldots a_{i_2} a_{i_1} = b_{i_1} b_{i_2} \ldots b_{i_l} a_{i_l} \ldots a_{i_2} a_{i_1}$. So PCP has a match $i_1, i_2, \ldots, i_l$.

  Why are a_i's necessary?

4.6 Chomsky language hierarchy

- Regular languages⇔regular grammars⇔finite automaton.
- Context-free languages⇔context-free grammars⇔push-down automaton.
- Context-sensitive languages⇔context-sensitive grammars⇔linear-bounded automaton.
- Turing-decidable languages⇔Turing machines as a decider (recursive).
- Turing-recognizable languages⇔grammars⇔Turing machines as a recognizer(recursively enumerable).
- Turing-nonrecognizable languages (non-recursively enumerable)

5 The Complexity of Solvable/Decidable Problems

5.1 The Satisfiability problem

- Why logic?
  - Computation of Turing machines can be expressed in logic statements, by certain expressions in number theory.
  - Many logic problems are undecidable or **NP**-complete.
- Boolean variables: x_i (**true, false**). Literals: x_i, $\neg x_i$. Operations: $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\rightarrow$ (implies), $\leftrightarrow$ (equivalent).
- Boolean expression/function ϕ: Literals connected by operations. Can also be defined recursively.
- Truth assignment $T : \{x_1, x_2, \ldots\} \rightarrow \{\textbf{true, false}\}$.
- T satisfies ϕ: $T \models \phi$ if ϕ is **true** under the truth assignment T.
- ϕ is satisfiable if $(\exists T)(T \models \phi)$. ϕ is valid if $(\forall T)(T \models \phi)$. ϕ is unsatisfiable if $(\forall T)(T \not\models \phi)$ or equivalently, $\neg\phi$ is valid.
- Conjunction normal form (CNF): $\phi = \wedge_i C_i$, where C_i, a clause, is the disjunction of one or more literals. Any Boolean expression can be written as one in CNF.
- Disjunction normal form (DNF): $\phi = \vee_i D_i$, where D_i is the conjunction of one or more literals. Any Boolean expression can be written as one in DNF.
- SATISFIABILITY (SAT):

  INSTANCE: A Boolean expression ϕ in CNF with variables $x_1, \ldots, x_n$.

  QUESTION: Is ϕ satisfiable?

  Remark: 3SAT (2SAT): Requires that each clause has exactly three (two) literals.

  Remark: SAT$\in$**NP**, 3SAT$\in$**NP**, 2SAT$\in$**P**.
- Boolean circuits: A graphical representation of Boolean expressions/functions. For example, consider the circuit for $\phi = ((x_1 \vee (x_1 \wedge x_2)) \vee ((x_1 \wedge x_2) \wedge \neg(x_2 \vee x_3)))$. Note that ϕ can be simplified to $x_1 \vee (x_1 \wedge x_2)$, and then x_1.
- Circuit complexity: Number of gates (nodes). Let C_n be any circuit with $n \geq 2$ inputs, $g(C_n)$ be its complexity, and $g(n)$ be $\max_{\forall C_n}\{g(C_n)\}$. Then $g(n) > 2^n/(2n)$. (Or equivalently, there is a $f(x_1, \ldots, x_n)$ for which no Boolean circuit with $\leq 2^n/2n$ gates can compute it.)

  Proof: Wish to prove that the number of functions is more than the number of circuits. First, there are 2^{2^n} different n-ary Boolean functions. (Why?) Next, the number of circuits with $m \leq 2^n/(2n)$ gates is no more than $((n+5)m^2)^m$. (Why? Each of the m gates can be one of the n variables or one of the three operations or one of the two values (T/F). So there are $n+5$ choices for the gate type. Also, each gate will receive up to two inputs, each of which can be from a gate. So there are $(n+5)m^2$ choices to define a gate.) Since $m \leq 2^n/(2n)$ then $2^{2^n} > ((n+5)m^2)^m$. (Why?) So there is at least one n-ary Boolean function that cannot be computed by any Boolean circuit with $m \leq 2^n/(2n)$ gates.
- CIRCUIT SAT:

  INSTANCE: A Boolean circuit C.

  QUESTION: Is C satisfiable?

  Remark: CIRCUIT SAT$\in$**NP**.

5.2 Decision (DEC) and optimization (OPT) problems

- Decision problems: Ask a question that can be answered with yes or no. Conveniently unifying and simplifying.
- Optimization problems: Ask to find a feasible solution S which maximizes/minimizes an objective function $f(S)$. (What is a feasible solution?)

  Example: BIPARTITE MATCHING:

  INSTANCE: A bipartite graph $B = (U, V, E)$ with $|U| + |V| = n$ and $|E| = m$.

GOAL: Find a maximum (size) matching. (A matching is $M \subseteq E$ s.t. no two edges in M share an endpoint.)

Example: MAX FLOW:

INSTANCE: A network $N = (V, E, s, t, c)$, where $G = (V, E)$ is a directed graph, s (source), t (sink) $\in V$, and $c : E \to \mathbf{N}$ maps an edge (i, j) to an integer $c((i, j))$, called the flow capacity.

GOAL: Define $f : E \to \mathbf{N}$ with $f((i, j)) \leq c((i, j))$ and for each $j \in V - \{s,t\}$, $\sum_{\forall i} f((i, j)) = \sum_{\forall k} f((j,k))$ such that the total flow from s to t is maximized.

Remark: Algorithms of MAX FLOW: $O(n^3 \max_{\forall(i,j)\in E} c((i, j))) \to O(n^5) \to O(m^2 n) \to O(mn^2) \to O(n^3)$.

Theorem: If there is an $O(T(m,n))$-time algorithm for MAX FLOW, there is an $O(T(m,n))$-time algorithm for BIPARTITE MATCHING. (BIPARTITE MATCHING reduces to MAX FLOW.)

Proof. Let A be the $O(T(m,n))$-time algorithm for MAX FLOW. Define an algorithm for BIPARTITE MATCHING which has two steps: (1) Take an instance $B = (U,V,E)$ and construct an instance for the MAX FLOW. Let $N = (U \cup V \cup \{s,t\}, E \cup \{(s,u) : \forall u \in U\} \cup \{(v,t) : \forall v \in V\}, s, t, c)$, where c is a unit-capacity function; (2) Apply A to the instance constructed and convert the maximum flow to the maximum matching it represents. The correctness of this algorithm lies in the easy-to-prove fact that the maximum matching for $B = (U,V,E)$ corresponds to the maximum flow for N constructed and vice versa. The time complexity is $O(T(m,n))$ plus the time to do the instance construction and the time to convert the maximum flow found to the maximum matching. Obviously, it is $O(T(m,n)+n+m) = O(T(m,n))$.

- Maximization (Minimization) $\to$ Decision: Add a bound $B > 0$ in the INSTANCE, and ask in QUESTION if there is a feasible solution S such that $f(S) \geq B$ ($f(S) \leq B$).

- Theorem: If there is an $O(T(n))$-time algorithm for the OPT, there is an $O(T(n))$-time algorithm for the corresponding DEC. (DEC is no harder than its OPT, or DEC reduces to OPT.) (Does OPT also reduce to DEC?)

  Proof. Let A_{OPT} be the $O(T(n))$-time algorithm for the OPT. Define an algorithm A_{DEC} for the corresponding DEC as follows: (1) Run A_{OPT} and let S^* be the optimal solution for the OPT; (2) Compare $f(S^*)$ with the bound B and decide whether the answer is "yes" or "no". Obviously A_{DEC} solves the DEC and its complexity is $O(T(n))$.

5.3 Defining Complexity Classes

- A complexity class is specified by several parameters.

  — Model of computation. (e.g., k-string TM.)

  — Mode of computation: How a machine accepts its input. (e.g., deterministic/nondeterministic modes.)

  — Resource to measure. (e.g., time and space.)

  — Bound on resource. (e.g., polynomial time and exponential space.)

- Some complexity class forms: **TIME**$(f(n))$, **SPACE**$(f(n))$, **NTIME**$(f(n))$, **NSPACE**$(f(n))$.

  (**TIME**$(f(n)) \subseteq$**NTIME**$(f(n))$, **SPACE**$(f(n)) \subseteq$**NSPACE**$(f(n))$)

  (**NTIME**$(f(n)) \subseteq$**SPACE**$(f(n))$, **NSPACE**$(f(n)) \subseteq$**TIME**$(k^{\log n + f(n)})$)

- Some complexity classes:

  — **P**$= \cup_{k\geq 0}$**TIME**(n^k).

  — **NP**$= \cup_{k\geq 0}$**NTIME**(n^k). (**P**$\subseteq$**NP**)

  — **PSPACE**$= \cup_{k\geq 0}$**SPACE**(n^k). (**NP**$\subseteq$**PSPACE**)

  — **NPSPACE**$= \cup_{k\geq 0}$**NSPACE**(n^k). (**PSPACE**=**NPSPACE**)

  — **EXP**$= \cup_{k\geq 0}$**TIME**(2^{n^k}). (**P**$\subset$**EXP**, **NP**$\subseteq$**EXP**, **NPSPACE**$\subseteq$**EXP**)

  — **L**=**SPACE**$(\log n)$.

  — **NL**=**NSPACE**$(\log n)$. (**L**$\subseteq$**NL**$\subseteq$**P**)

  — **coP**$= \{L : \overline{L} \in$**P**$\}$. (**P**=**coP**)

  — **coNP**$= \{L : \overline{L} \in$**NP**$\}$. (**NP**=**coNP**???)

  — **coPSPACE**$= \{L : \overline{L} \in$**PSPACE**$\}$. (**PSPACE**=**coPSPACE**)

  — **coNPSPACE**$= \{L : \overline{L} \in$**NPSPACE**$\}$. (**NPSPACE**=**coNPSPACE**)

Remark: For any complexity class $\mathcal{C}$, its complement **co**$\mathcal{C} = \{L : \overline{L} \in \mathcal{C}\}$. If $\mathcal{C}$ is deterministic, then $\mathcal{C} =$**co**$\mathcal{C}$. However, if $\mathcal{C}$ is nondeterministic, it is sometimes not known whether $\mathcal{C} =$**co**$\mathcal{C}$.

5.4 NP revisited

- Time complexity of nondeterministic TMs (NTMs): Let N be an NTM that is a decider (where all computation paths halt in the tree for any input). The time complexity of N, $f(n)$, is the maximum number of steps that N uses on any computation path for any input of length n. In other words, $f(n)$ is the maximum height of all computation trees for all input of length n.

- An unreasonable model of computation:

  Theorem; Every $T(n)$-time multi-tape TM has an equivalent $O(T^2(n))$-time single-tape TM.

  Theorem: Every $T(n)$-time single-tape NTM has an equivalent $O(2^{O(T(n))})$-time single-tape DTM.

- Definition of the **NP** class: The set of all languages (problems) that can be decided (solved) in polynomial time by nondeterministic TMs (algorithms).

- Polynomial verifiability: Verifying a solution is usually much easier than finding a solution. For example, the current fastest algorithm for finding a Hamiltonian path between two nodes in a graph takes exponential time. But if someone presents to you a Hamiltonian path, it is very easy (in just linear time) for you to verify its correctness.

  Definition: A verifier for a language L is an algorithm that accepts $< w, c >$ for some string c, called a certificate, and any $w \in L$. (Here, w is any string in L but c is generated nondeterministically based on w.)

  Example: Consider the language associated with the Hamiltonian Path problem (HP), $L_{HP} = \{< G > | G$ is an undirected graph with a Hamiltonian path that visits each node once$\}$. The input to the verifier is G and a candidate path P that serves as a certificate. The verifier can check whether P is indeed a Hamiltonian path in G in polynomial time.

  Theorem: A language is decided by an NTM in polynomial time iff it has a polynomial-time verifier.

- Another definition of the **NP** class: The set of languages (problems) that have polynomial-time verifiers.

  Example: A clique in an undirected graph is a subgraph which is also complete.

  - MAX-CLIQUE: (OPT)
    INSTANCE: An undirected graph G.
    GOAL: Find a clique in G with the maximum size (number of nodes).
  - CLIQUE: (DEC)
    INSTANCE: An undirected graph G and an integer bound k.
    QUESTION: Is there a clique in G with size greater than or equal to k?
  - $L_{CLIQUE} = \{< G, k > | G$ has a clique with size greater than or equal to $k\}$.

  CLIQUE (or L_{CLIQUE}) is in **NP** since a polynomial-time verifier can be designed as follows: Given input G and k, and a candidate clique C (the certificate), test if C is indeed a clique for G with at least k nodes, which can be done in polynomial time. Or, here is a more often used form of proof that a problem (such as CLIQUE) is in **NP**. We design an algorithm that first nondeterministically selects (or guesses) a clique, and then tests (or verifies) if the guessed clique is indeed a clique with at least k nodes.

5.5 Reduction

- For problems in **NP**, some seem to be more difficult than others. For example, both TSP and REACHABILITY are in **NP**, but the fastest deterministic algorithm available for TSP takes exponential time while REACHABILITY can be solved in polynomial time.

- Polynomial reduction: $\Pi_1 \propto_p \Pi_2$ if there is a function $R : \{I_1\} \to \{I_2\}$ such that R can be computed in polynomial time and for any instance I_1 for Π_1, I_1 has a yes-solution if and only if $R(I_1)$ (an instance for Π_2) has a yes-solution.

- The polynomial-time computability requirement for reduction is not needed for undecidability proofs. Why?

- $\Pi_1 \propto_p \Pi_2$ implies that Π_2 is at least as hard as Π_1, i.e., if there is a polynomial-time solution to Π_2 then Π_1 can also be solved in polynomial time, or equivalently, if Π_1 requires exponential time, Π_2 takes at least exponential time.

- Transitivity: If $\Pi_1 \propto_p \Pi_2$ and $\Pi_2 \propto_p \Pi_3$, then $\Pi_1 \propto_p \Pi_3$

- HAMILTONIAN PATH

  INSTANCE: A graph $G = (V, E)$.

  QUESTION: Is there a path that visits each node exactly once?

  HAMILTONIAN PATH$\propto_p$SAT.

  Proof. For any graph G with n nodes, $1, 2, \ldots, n$, define n^2 Boolean variables x_{ij} for $1 \le i, j \le n$. $x_{ij} = 1$ implies that node j is the ith node in the Hamiltonian path. We then include the following clauses in the CNF:

  $(x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj})$ for each j. Each node has to be included in the Hamiltonian path.

  $(\neg x_{ij} \vee \neg x_{kj})$ for each j and any $i \neq k$. Each node can not appear more than once in the Hamiltonian path.

  $(x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$ for each i. Some node must be the ith in the path.

  $(\neg x_{ij} \vee \neg x_{ik})$ for each i and any $j \neq k$. No two nodes should be ith in the path.

  $(\neg x_{ki} \vee \neg x_{(k+1)j})$ for each k and any edge $(i, j) \notin E$. For any edge $(i, j) \notin E$, node j should not come right after i in the path.

  Obviously, this construction R can be done in polynomial time. Next we must prove that there is a Hamiltonian path in G if and only if the Boolean expression $R(G)$ is satisfiable. The proof is omitted.

- SAT$\propto_p$3SAT.

  Proof. Given any instance of SAT, $f(x_1, \ldots, x_n) = c_1 \wedge \cdots \wedge c_m$, where c_i is a disjunction of literals. To construct an instance for 3SAT, we need to convert any c_i to an equivalent c_i', a conjunction of clauses with exactly 3 literals.

  Case 1. If $c_i = z_1$ (one literal), define y_i^1 and y_i^2. Let $c_i' = (z_1 \vee y_i^1 \vee y_i^2) \wedge (z_1 \vee y_i^1 \vee \neg y_i^2) \wedge (z_1 \vee \neg y_i^1 \vee y_i^2) \wedge (z_1 \vee \neg y_i^1 \vee \neg y_i^2)$.
  Case 2. If $c_i = z_1 \vee z_2$ (two literals), define y_i^1. Let $c_i' = (z_1 \vee z_2 \vee y_i^1) \wedge (z_1 \vee z_2 \vee \neg y_i^1)$.

  Case 3. If $c_i = z_1 \vee z_2 \vee z_3$ (three literals), let $c_i' = c_i$.

  Case 4. If $c_i = z_1 \vee z_2 \vee \cdots \vee z_k$ ($k > 3$), define $y_i^1, y_i^2, \ldots, y_i^{k-3}$. Let $c_i' = (z_1 \vee z_2 \vee y_i^1) \wedge (\neg y_i^1 \vee z_3 \vee y_i^2) \wedge (\neg y_i^2 \vee z_4 \vee y_i^3) \wedge \cdots \wedge (\neg y_i^{k-3} \vee z_{k-1} \vee z_k)$.

  If c_i is satisfiable, then there is a literal $z_l = T$ in c_i. If $l = 1, 2$, let $y_i^1, \ldots, y_i^{k-3} = F$. If $l = k-1, k$, let $y_i^1, \ldots, y_i^{k-3} = T$. If $3 \le l \le k-2$, let $y_i^1, \ldots, y_i^{l-2} = T$ and $y_i^{l-1}, \ldots, y_i^{k-3} = F$. So c_i' is satisfiable.

  If c_i' is satisfiable, assume $z_l = F$ for all $l = 1, \ldots, k$. Then $y_i^1, \ldots, y_i^{k-3} = T$. So the last clause $(\neg y_i^{k-3} \vee z_{k-1} \vee z_k) = F$. Therefore, c_i' is not satisfiable. Contradiction.

  The instance of 3SAT is therefore $f'(x_1, \ldots, x_n, \ldots) = c_1' \wedge \cdots \wedge c_m'$, and f is satisfiable if and only if f' is satisfiable.

5.6 Completeness

- Reducibility defines a partial order among problems. What are the *maximal elements* (more than one) in this partial order?

- Let $\mathcal{C}$ be a complexity class, and let L be a language in $\mathcal{C}$. We say that L is $\mathcal{C}$-complete if any language L' in $\mathcal{C}$ can be reduced to L, i.e., $L' \propto_p L, \forall L' \in \mathcal{C}$.

- **P**-complete: $L \in$**P** is **P**-complete if $L' \propto_p L, \forall L' \in$**P**. (Hardest in **P** and therefore captures the essence and difficulty of **P**.)

- **NP**-complete: $L \in$**NP** is **NP**-complete if $L' \propto_p L$, $\forall L' \in$**NP**. (Hardest in **NP** and therefore captures the essence and difficulty of **NP**.)

- If a **P**-complete problem is in **L**, then **P=L**. If a **P**-complete problem is in **NL**, then **P=NL**. If an **NP**-complete problem is in **P**, then **P=NP**.

- $\mathcal{C}$ is closed under reductions if $L \propto_p L'$ and $L' \in \mathcal{C}$ implies $L \in \mathcal{C}$. **P**, **NP**, **coNP**, **L**, **NL**, **PSPACE**, and **EXP** are all closed under reductions.

- If two classes $\mathcal{C}$ and $\mathcal{C}'$ are both closed under reductions, and there is a language L which is complete for both $\mathcal{C}$ and $\mathcal{C}'$, then $\mathcal{C} = \mathcal{C}'$.

5.7 P-completeness

- How to prove a problem Π is **P**-complete:

  Show that (1) $\Pi \in$**P** and (2) for any $\Pi' \in$**P**, $\Pi' \propto_p \Pi$.

  Show that (1) $\Pi \in$**P** and (2) for some **P**-complete Π', $\Pi' \propto_p \Pi$.

- The first method is more difficult than the second. But we have to use the first method to prove the first **P**-complete problem.

- CIRCUIT VALUE

  INSTANCE: A Boolean circuit C without variables.

  QUESTION: Is C satisfiable?

  CIRCUIT VALUE is **P**-complete.

5.8 NP-completeness

- How to prove a problem Π is **NP**-complete:

  Show that (1) $\Pi \in$**NP** and (2) for any $\Pi' \in$**NP**, $\Pi' \propto_p \Pi$.

  Show that (1) $\Pi \in$**NP** and (2) for some **NP**-complete Π', $\Pi' \propto_p \Pi$.

- The first method is more difficult than the second. But we have to use the first method to prove the first **NP**-complete problem.

- Cook's Theorem: SAT is **NP**-complete.

  Proof. SAT is clearly in **NP** since a NTM exists that guesses a truth assignment and verifies its correctness in polynomial time. Now we wish to prove $\forall \Pi \in$**NP**, $\Pi \propto_p$SAT, or equivalently, for any polynomial-time NTM M, $L(M) \propto_p L_{SAT}$.

  For any NTM M, assume $Q = \{q_0, q_1(\text{accept}), q_2(\text{reject}), \ldots, q_r\}$ and $\Gamma = \{s_0, s_1, s_2, \ldots, s_v\}$. Also assume that the time is bounded by $p(n)$, where n is the length of the input. We wish to prove that there is a function $f_M : \Sigma^* \to \{\text{instances of SAT}\}$ such that $\forall x \in \Sigma^*$, $x \in L(M)$ iff $f_M(x)$ is satisfiable. In other words, we wish to use a Boolean expression $f_M(x)$ to describe the computation of M on x.

  Variables in $f_M(x)$:

  — State: $Q[i,k]$. M is in q_k after the ith step of computation (at time i).

  — Head: $H[i,j]$. Head points to tape square j at time i.

  — Symbol: $S[i,j,l]$. Tape square j contains s_l at time i. (Assume the tape is one-way infinite and the leftmost square is labeled with 0.)

  For example, initially $i = 0$. Assume the configuration is q_0abba. Let $s_0 = B$, $s_1 = a$, and $s_2 = b$. Therefore, we set the following Boolean variables to be true: $Q[0,0]$, $H[0,0]$, $S[0,0,1]$, $S[0,1,2]$, $S[0,2,2]$, $S[0,3,1]$ and $S[0,j,0]$ for $j = 4,5,\ldots$. A configuration defines a truth assignment, but not vice versa.

  Clauses in $f_M(x)$:

  — At any time i, M is in exactly one state.

  $Q[i,0] \vee \cdots \vee Q[i,r]$ for $0 \le i \le p(n)$.

  $\neg Q[i,k] \vee \neg Q[i,k']$ for $0 \le i \le p(n)$ and $0 \le k < k' \le r$.

  — At any time i, head is scanning exactly one square.

  $H[i,0] \vee \cdots \vee H[i,p(n)]$ for $0 \le i \le p(n)$.

  $\neg H[i,j] \vee \neg H[i,j']$ for $0 \le i \le p(n)$ and $0 \le j < j' \le p(n)$.

  — At any time i, each square contains exactly one symbol.

  $S[i,j,0] \vee \cdots \vee S[i,j,v]$ for $0 \le i \le p(n)$ and $0 \le j \le p(n)$.

  $\neg S[i,j,l] \vee \neg S[i,j,l']$ for $0 \le i \le p(n)$, $0 \le j \le p(n)$ and $0 \le l < l' \le v$.

  — At time 0, M is in its initial configuration. Assume $x = s_{l_1} \cdots s_{l_n}$.

  $Q[0,0]$.

$H[0,0]$.

$S[0,0,l_1],\ldots,S[0,n-1,l_n]$.

$S[0,j,0]$ for $n \leq j \leq p(n)$.

— By time $p(n)$, M has entered q_1 (accept)). (If M halts in less than $p(n)$ steps, additional moves can be included in the transition function.)

$Q[p(n),1]$.

— Configuration at time $i \rightarrow$ configuration at time $i+1$. Assume $\delta(q_k,s_l) = (q_{k'},s_{l'},D)$, where $D = -1,1$.

If the head does not point to square j, symbol on j is not changed from time i to time $i+1$.

$H[i,j] \vee \neg S[i,j,l] \vee S[i+1,j,l]$ for $0 \leq i \leq p(n)$, $0 \leq j \leq p(n)$, and $0 \leq l \leq v$.

If the current state is q_k, the head points to square j which contains symbol s_l, then changes are made accordingly.

$\neg H[i,j] \vee \neg Q[i,k] \vee \neg S[i,j,l] \vee H[i+1,j+D]$,

$\neg H[i,j] \vee \neg Q[i,k] \vee \neg S[i,j,l] \vee Q[i+1,k']$, and

$\neg H[i,j] \vee \neg Q[i,k] \vee \neg S[i,j,l] \vee S[i+1,j,l']$, for $0 \leq i \leq p(n)$, $0 \leq j \leq p(n)$, $0 \leq k \leq r$, and $0 \leq l \leq v$.

Let $f_M(x)$ be the conjunction of all the clauses defined above. Then $x \in L(M)$ iff there is an accepting computation of M on x iff $f_M(x)$ is satisfiable. f_M can be computed in polynomial time since $|f_M(x)| \leq$ (number of clauses) $*$(number of variables)$= O(p(n)^2) * O(p(n)^2) = O(p(n)^4)$.

So there is a polynomial reduction from any language in **NP** to SAT. So SAT is **NP**-complete.

6 NP-Complete Problems

6.1 How to prove Π is NP-complete

- Step 1. Show that $\Pi \in$**NP**.
- Step 2. Show that Π is **NP**-hard.

  — Choose a known **NP**-complete Π' such as SAT;

  — Construct a polynomial reduction f from any instance of Π' to some instance of Π. (Polynomial-time computability and the if and only if relation)

6.2 NP-complete problems

- SAT is **NP**-complete. (Cook's Theorem)
- 3SAT is **NP**-complete. (3SAT$\in$**NP** and SAT$\propto_p$3SAT)
- CLIQUE is **NP**-complete.

  INSTANCE: $G = (V,E)$ and $k \leq |V|$.

  QUESTION: Does G contain a clique (a complete subgraph) of size k or more?

  Proof. CLIQUE is obviously in **NP**. Now we show that SAT$\propto_p$CLIQUE. For any Boolean expression $f(x_1,\ldots,x_n) = c_1 \wedge \cdots \wedge c_m$, we define a graph G, where $V = \{< y,i > | y \text{ is a literal in } c_i\}$ and $E = \{< y,i >< z,j > | i \neq j \text{ and } y \neq \neg z\}$. Finally, let $k = m$. Since $|V| = O(mn)$ and $|E| = O(m^2n^2)$, the graph can be constructed in polynomial time. Next we show that f is satisfiable iff there is a clique in G with size m.

  Assume there is a truth assignment for f such that at least one literal, say l_i, in c_i $(i = 1,\ldots,m)$ is true. Consider a subgraph of G consisting of m nodes $< l_i, i >$. For any $i \neq j$, we have $l_i \neq \neg l_j$. So edge $< l_i,i >< l_j, j >$ is in G. Therefore the subgraph is a clique of size m.

  Assume G contains a clique of size m or more. Let V' be the set of any m nodes in the clique. Since no edge in G connects $< y,i >$ and $< z, j >$ when $i = j$, and nodes in V' are fully connected, then V' contains $< l_1, 1 >,\ldots,< l_m,m >$, where l_i is a literal in clause c_i. Further $l_i \neq \neg l_j$ for any $i \neq j$. Then there is a truth assignment such that $l_i = true$ for $i = 1,\ldots,m$. Obviously, $f(x_1,\ldots,x_n) = c_1 \wedge \cdots \wedge c_m$ is satisfiable under the truth assignment.

- VERTEX COVER is **NP**-complete.

  INSTANCE: $G = (V,E)$ and $k \leq |V|$.

  QUESTION: Is there $V' \subseteq V$ with k or less nodes such that $\forall(u,v) \in E$, either $u \in V'$ or $v \in V'$?

  Proof. VERTEX COVER is clearly in **NP**. Now we wish to show 3SAT$\propto_p$VERTEX COVER. Let $f(x_1,\ldots,x_n) = c_1 \wedge \ldots \wedge c_m$ be any instance for 3SAT. We construct a graph $G = (V,E)$ as follows. For each Boolean variable x_i, define $V_i = \{x_i, \neg x_i\}$ and $E_i = \{(x_i, \neg x_i)\}$. For each clause $c_j = l_1[j] \vee l_2[j] \vee l_3[j]$, define $V'_j = \{a_1[j], a_2[j], a_3[j]\}$, $E'_j = \{(a_1[j],a_2[j]),(a_2[j],a_3[j]),(a_3[j],a_1[j])\}$, and $E''_j = \{(a_1[j],l_1[j]),(a_2[j],l_2[j]),(a_3[j],l_3[j])\}$. Let $G = (V,E)$, where $V = (\cup_{i=1}^n V_i) \cup (\cup_{j=1}^m V'_j)$ and $E = (\cup_{i=1}^n E_i) \cup (\cup_{j=1}^m E'_j) \cup (\cup_{j=1}^m E''_j)$. Let $k = n+2m$. It is easy to see that the construction can be accomplished in polynomial time.

  Next we prove that f is satisfiable if and only if G has a vertex cover of size k or less. Assume f is satisfiable. There is a truth assignment $A : \{x_1,\ldots,x_n\} \to \{T,F\}$ for f. The corresponding vertex cover includes one vertex from each V_i and two vertices from each V'_j. The vertex from V_i is x_i if $A(x_i) = T$ and is $\neg x_i$ if $A(x_i) = F$. This ensures that at least one of the three edges in E''_j is covered because A satisfies each clause c_j. Therefore we need only include in the vertex cover the endpoints from V'_j of the other two edges in E''_j. This gives the desired vertex cover of size $n+2m$.

  Assume there is a vertex cover with size k or less for G. It must contain at least one vertex from each V_i and at least two vertices from each V'_j. Since this gives a total of at least $n+2m = k$, the vertex cover contains exactly one vertex from each V_i and two vertices from each V'_j. Define a truth assignment $A : \{x_1,\ldots,x_n\} \to \{T,F\}$ such that $A(x_i) = T$ if x_i is in the vertex cover and $A(x_i) = F$ if $\neg x_i$ is in the vertex cover. For the three edges in each E''_j, two will be covered by the two vertices in V'_j and the third will be covered by the vertex in V_i. Therefore, clause c_j will be satisfied by the truth assignment. So f is satisfiable.

  Remark 1. VERTEX COVER$\propto_p$CLIQUE

  For any instance of VERTEX COVER: $G = (V,E)$ and k, define an instance for CLIQUE: $\overline{G} = (V,\overline{E})$ and $l = |V| - k$.

Remark 2. CLIQUE$\propto_p$VERTEX COVER

For any instance of CLIQUE: $G = (V,E)$ and k, define an instance for VERTEX COVER: $\overline{G} = (V, \overline{E})$ and $l = |V| - k$.

Remark 3. INDEPENDENT SET is equivalent to VERTEX COVER

INSTANCE: An undirected graph $G = (V,E)$ and an integer $k > 0$.

QUESTION: Is there an independent set I with $|I| = k$? ($I \subseteq V$ is an independent set if whenever $i, j \in I$ then $(i, j) \notin E$.)

INDEPENDENT SET$\propto_p$VERTEX COVER and VERTEX COVER$\propto_p$INDEPENDENT SET. This is because that I is an independent set iff $V - I$ is a vertex cover of the same graph.

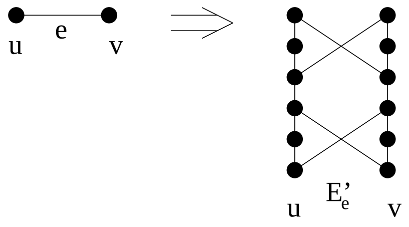
- HAMILTONIAN CYCLE is **NP**-complete.

  INSTANCE: $G = (V,E)$.

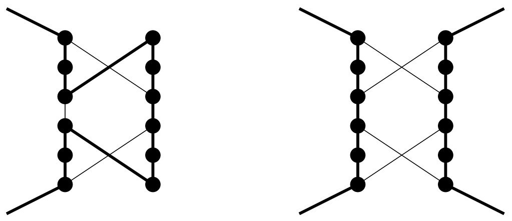
  QUESTION: Is there a cycle that visits each node exactly once?

  Proof. HAMILTONIAN CYCLE is clearly in **NP**. Now we wish to show that VERTEX COVER$\propto_p$HAMILTONIAN CYCLE. Take any instance of VERTEX COVER: $G = (V,E)$ and $k \leq |V|$. We define as follows an instance for HAMILTONIAN CYCLE: $G' = (V', E')$.

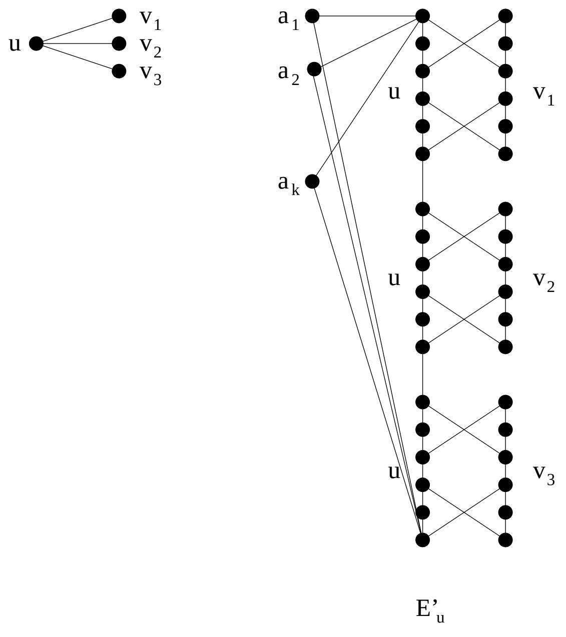
  For each $e = (u, v) \in E$ in G, construct an edge-component E'_e in G' as follows, with 12 nodes and 14 edges.



Only four vertices at the corners can be connected to vertices in other components. The Hamiltonian cycle in G', if exists, has the following two possible ways to pass through a component.

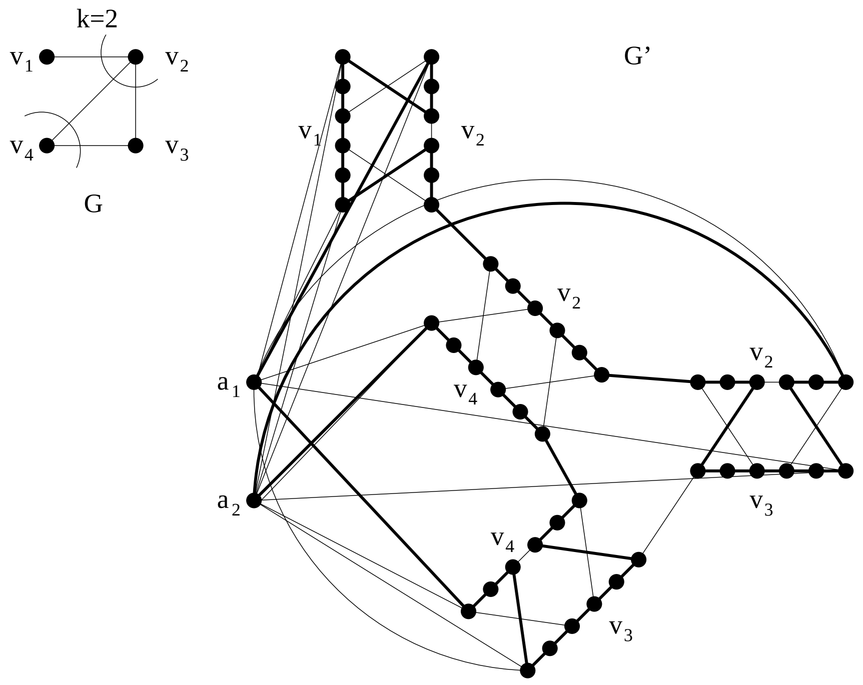


Add $a_1, \ldots, a_k$ to V' such that for each $u \in V$ in G,



Note that in the above figure, connect different edge components with edges on that sides that correspond to a shared vertice.

Now the following figure shows an example.

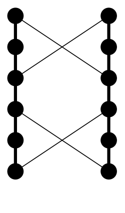


Clearly, the reduction can be constructed in polynomial time since $|V'| = k + 12|E| \le |V| + 12|E|$ and $|E'| = 14|E| + (2|E| - |V|) + 2k|V| \le 2|V|^2 + 16|E| - |V|$.

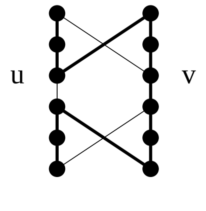
Now we prove that G has a vertex cover of size k or less iff G' has a Hamiltonian cycle.

Let $V_1 = \{v_1, \dots, v_l\} \in V$ be a vertex cover of G. Let $V_1^* = \{v_1, \dots, v_l, \dots, v_k\}$ (if $l < k$). V_1^* is still a vertex cover. We construct a Hamiltonian cycle as follows:

1. Portions in the edge components. Let $e = (u, v) \in E$. In the edge component E'_e, if $u, v \in V_1^*$,



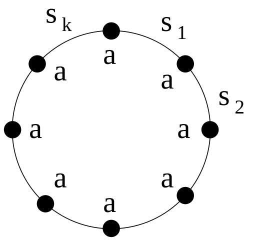
and if $u \in V_1^*$ and $v \notin V_1^*$,



2. Connect the edge components naturally. (Only one way)

3. Add $a_1, \dots, a_k$ into the paths to form a Hamiltonian cycle: $a_1 \dots a_2 \dots a_k$. (May have several ways)

Assume G' has a Hamiltonian cycle. Then there must be k paths $s_1, \dots, s_k$ in the Hamiltonian cycle, each separated by the a-type nodes.



Let s_i pass through a few edge components. Then the edges they represent must share one common vertex v_i in G. Define $V_1 = \{v_1, \ldots, v_k\}$. For each edge in G, it is represented by some edge component which is passed through by some s_i. Therefore all edges in G are covered by vertices in V_1. So V_1 is a vertex cover of size k.

Remark 1. HAMILTONIAN CYCLE, HAMILTONIAN PATH, and HAMILTONIAN PATH BETWEEN u AND v are **NP**-complete. DIRECTED HAMILTONIAN CYCLE, DIRECTED HAMILTONIAN PATH, and DIRECTED HAMILTONIAN PATH BETWEEN u AND v are **NP**-complete.

Remark 2. TRAVELING SALESMAN PROBLEM (TSP) is **NP**-complete.

INSTANCE: An edge-weighted graph $G = (V,E)$, and $B \geq 0$.

QUESTION: Is there a Hamiltonian cycle with total weight no greater than B?

Proof. TSP is obviously in **NP**. Now we show that HAMILTONIAN CYCLE$\propto_p$TSP. For any instance of HAMILTONIAN CYCLE: $G = (V,E)$, define an instance of TSP: $G' = G$, $w(e) = 1, \forall e \in E$, $B = |V|$. The iff proof is straightforward.

- 3D MATCHING (3DM) is **NP**-complete.

  INSTANCE: $M \subseteq X \times Y \times Z$, where X,Y,Z are disjoint sets and $|X| = |Y| = |Z| = q$.

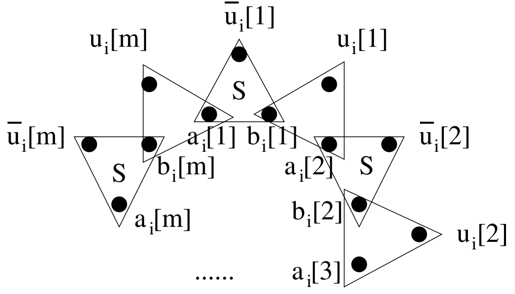
  QUESTION: Does M contain a matching M', i.e., $M' \subseteq M$ such that $|M'| = q$ and no two elements in M' agree in any coordinate?

  Remark. 2D MATCHING (and the marriage problem) is in **P**.

  Proof. 3DM is obviously in **NP**. We will prove that 3SAT$\propto_p$3DM. Consider any instance for 3SAT with variables $u_1, \ldots, u_n$ and clauses $c_1, \ldots, c_m$.

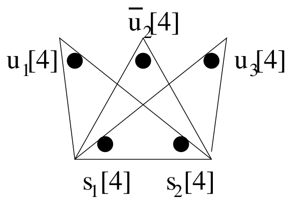
  The truth setting components:

  For each variable u_i, define $u_i[1], \overline{u}_i[1], \ldots, u_i[m], \overline{u}_i[m] \in X$, $a_i[1], \ldots, a_i[m] \in Y$, and $b_i[1], \ldots, b_i[m] \in Z$. Then add the following triples in M: $(\overline{u}_i[j], a_i[j], b_i[j])$ for $j = 1, \ldots, m$, $(u_i[j], a_i[j+1], b_i[j])$ for $j = 1, \ldots, m-1$, and finally $(u_i[m], a_i[1], b_i[m])$. Note that $a_i[j]$ and $b_i[j]$ will not appear in any other triples in M. Later in the iff proof, $u_i = T$ under a truth assignment iff the shaded matching (marked by S in the figure) is chosen iff $u_i[1], \ldots, u_i[m]$ are not covered, and $u_i = F$ under a truth assignment iff the unshaded matching is chosen iff $\overline{u}_i[1], \ldots, \overline{u}_i[m]$ are not covered,



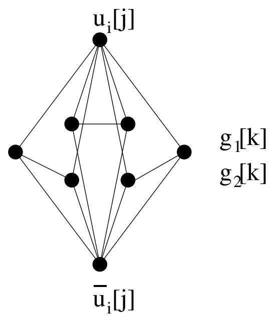
  The satisfaction testing components:

  For any clause c_j, define $s_1[j] \in Y$ and $s_2[j] \in Z$. Add to M $(u_i[j], s_1[j], s_2[j])$ if $u_i \in c_j$ and $(\overline{u}_i[j], s_1[j], s_2[j])$ if $\overline{u}_i \in c_j$. Note that $s_1[j]$ and $s_2[j]$ will no appear in any other triples in M. In the iff proof, one of the three literals in a clause is true iff the corresponding node is covered iff the corresponding triple is in the matching M'. For example, if $c_4 = u_1 \vee \overline{u}_2 \vee u_3$, then



The garbage collection components:

So far in the instance for 3DM we have $|X| = 2mn$, $|Y| = mn+m$, and $|Z| = mn+m$. Also $|M| = 2mn+3m$. Next we will add $mn-m$ nodes to Y and Z respectively so that $|X| = |Y| = |Z| = 2mn$. Subsequently, we need to add some more triples to M. Define $g_1[k]$ in Y and $g_2[k]$ in Z, for $k = 1,\ldots,mn-m$. Add to M triples $(u_i[j], g_1[k], g_2[k])$ and $(\bar{u}_i[j], g_1[k], g_2[k])$ for all $i = 1,\ldots,n$, $j = 1,\ldots,m$, and $k = 1,\ldots,mn-m$. The final size of M is then $2mn+3m+2(mn(mn-m))$.



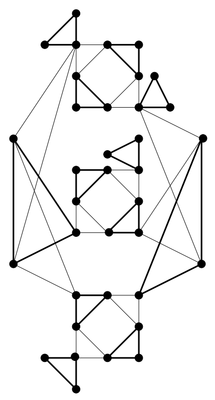
Since the iff proof is rather tedious, we will verify the reduction by an example. Let $f(u_1,u_2,u_3)$ be the Boolean expression for the 3SAT and let $c_1 = u_1 \vee \bar{u}_2 \vee u_3$ and $c_2 = u_1 \vee u_2 \vee \bar{u}_3$. Then the instance for 3DM is defined as follows:

$X = \{u_1[1], \bar{u}_1[1], u_1[2], \bar{u}_1[2], u_2[1], \bar{u}_2[1], u_2[2], \bar{u}_2[2], u_3[1], \bar{u}_3[1], u_3[2], \bar{u}_3[2]\}$

$Y = \{a_1[1], a_1[2], a_2[1], a_2[2], a_3[1], a_3[2], s_1[1], s_1[2], g_1[1], g_1[2], g_1[3], g_1[4]\}$

$Z = \{b_1[1], b_1[2], b_2[1], b_2[2], b_3[1], b_3[2], s_2[1], s_2[2], g_2[1], g_2[2], g_2[3], g_2[4]\}$

M contains the following triples (not all are shown in the figure). The matching M' is represented by boldface triangles. It corresponds to the satisfying truth assignment $u_1 = T, u_2 = F, u_3 = F$.



Remark. EXACT COVER BY 3-SETS (X3C) is **NP**-complete.

INSTANCE: Set U with $|U| = 3q$ and $C = \{c_1,\ldots,c_m\}$, where $c_i \subseteq U$ and $|c_i| = 3$.

QUESTION: Is there $C' \subseteq C$ such that each element of U occurs in exactly one member of C'?

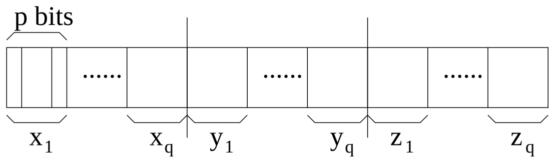
Proof. Consider the following reduction from 3DM. For any $M \subseteq X \times Y \times Z$ with $|X| = |Y| = |Z| = q$, define $U = X \cup Y \cup Z$ and $C = M$.

- PARTITION is **NP**-complete.

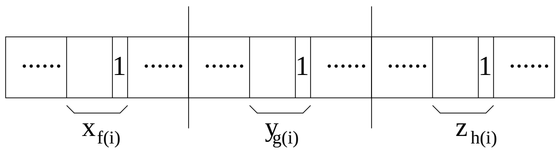
  INSTANCE: Set A of n positive integers $a_1,\ldots,a_n$.

QUESTION: Is there $A' \subseteq A$ such that $\sum_{a_i \in A'} a_i = \sum_{a_i \in A-A'} a_i$?

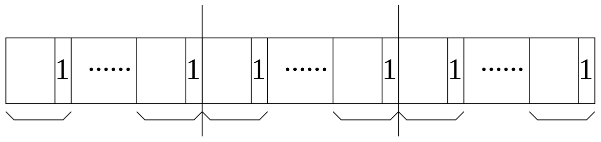
Proof. PARTITION is clearly in **NP**. Next we wish to prove that 3DM$\propto_p$PARTITION. Consider an arbitrary instance for 3DM: $M \subseteq X \times Y \times Z$ with $|X| = |Y| = |Z| = q$. Assume $M = \{m_1, \ldots, m_k\}$. Now we define $A = \{a_1, \ldots, a_k, b_1, b_2\}$. First, the binary representation of each number in A will be of the following format, where $p = \lceil \log_2 k \rceil$:



Then for any triple $m_i = (x_{f(i)}, y_{g(i)}, z_{h(i)}) \in M$, define $a_i \in A$ as follows:



Note that there are only three 1's in each a_i. Next define the bound B as follows:



Note that there are $3q$ 1's in the binary representation of B. Obviously, for any $A' \subseteq \{a_1, \ldots, a_k\}$, $\sum_{a_i \in A'} a_i = B$ iff $M' = \{m_i | a_i \in A'\}$ is a matching. Finally, let us define $b_1 = 2\sum_{i=1}^k a_i - B$ and $b_2 = \sum_{i=1}^k a_i + B$.

The time needed to construct the above instance for PARTITION is $O(k \cdot (3pq) + 3pq + k + k) = O(kpq) = O(kq \log k)$. Next we prove that M has a matching M' iff A has a partition.

Assume that there is a partition in A, i.e., $A' \subseteq A$ such that $\sum_{a_i \in A'} a_i = \sum_{a_i \in A-A'} a_i = 2\sum_{i=1}^k a_i$. So b_1 and b_2 must be in different subsets of the partition. WLOG, assume that $b_1 \in A'$ (then $b_2 \in A - A'$). The remaining numbers in A' form a subset of $\{a_1, \ldots, a_k\}$ with sum B. Therefore, M has a matching M'.

Assume that M has a matching $M' \subseteq M$. Let $A' = \{b_1\} \cup \{a_i | m_i \in M'\}$. Therefore, $\sum_{a_i \in A'} a_i = b_1 + B = 2\sum_{i=1}^k a_i = \sum_{a_i \in A-A'} a_i$. So there is a partition in A.

6.3 A summary of proving techniques ($\Pi' \propto_p \Pi$)

- Restriction: Π contains a known **NP**-complete Π' as a special case. For example, 3DM$\propto_p$X3C, HC$\propto_p$TSP.

  HITTING SET

  INSTANCE: Collection C of subsets of a set S, positive integer K.

  QUESTION: Does S contain a hitting set for C of size K or less, that is, a subset $S' \subseteq S$ with $|S'| \leq K$ and such that S' contains at least one element from each subset in C?

  Proof. Restrict to VC by allowing only instances having $|c| = 2$ for all $c \in C$.

  KNAPSACK

  INSTANCE: A finite set U, a size $s(u) \in Z^+$ and a value $v(u) \in Z^+$ for each $u \in U$, a size constraint $S \in Z^+$, and a value goal $V \in Z^+$.

  QUESTION: Is there a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq S$ and $\sum_{u \in U'} v(u) \geq V$?

  Proof. Restrict to PARTITION by allowing only instances in which $s(u) = v(u)$ for all $u \in U$ and $S = V = \frac{1}{2}\sum_{u \in U} s(u)$.

- Local replacement:

  Replace in a uniform way some basic units in the instance for Π' with some structures which make up the instance for Π. For example, SAT$\propto_p$3SAT, 3DM$\propto_p$PARTITION.

  SEQUENCING WITHIN INTERVALS

  INSTANCE: A finite set T of tasks and, for each $t \in T$, release time $r(t)$, a deadline $d(t)$, and a length $l(t)$.

  QUESTION: Is there a feasible schedule for T, that is, a function $\sigma : T \to Z^+$ such that, for each $t \in T$, $\sigma(t) \geq r(t)$, $\sigma(t) + l(t) \leq d(t)$, and for any $t' \in T - \{t\}$, either $\sigma(t') + l(t') \leq \sigma(t)$ or $\sigma(t') \geq \sigma(t) + l(t)$?

  Proof. Reduce from PARTITION. The basic units of the PARTITION instance are the individual elements $a \in A$. The local replacement for each $a \in A$ is a single task t_a with $r(t_a) = 0$, $d(t_a) = B + 1$, and $l(t_a) = a$. Here, $B = \sum_A a$. The enforcer is a single task $\bar{t}$ with $r(\bar{t}) = B/2$, $d(\bar{t}) = B/2 + 1$, and $l(\bar{t}) = 1$.

- Component design:

  Components are designed and combined to construct the instance for Π. For example, 3SAT$\propto_p$VC, VC$\propto_p$HC, 3SAT$\propto_p$3DM.

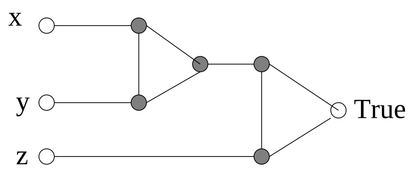
  GRAPH 3-COLORABILITY

  INSTANCE: $G = (V, E)$.

  QUESTION: Is G 3-colorable, that is, does there exist a function $f : V \to \{1, 2, 3\}$ such that $f(u) \neq f(v)$ whenever $(u, v) \in E$?

  Proof. Use 3SAT. Define the graph G as follows:

  - For each variable x_i, create two nodes, one for itself and one for its negation;
  - Create three special nodes: Red, $True$, and $False$;
  - Create five nodes for each clause;
  - Add literal edges, which make triangles for $x_i, \neg x_i$, and Red, for $i = 1, \ldots, n$, and a triangle for $Red, True$, and $False$;
  - Add clause edges as follows: For each clause $(x \vee y \vee z)$, construct the following widget.



Prove that if each of x, y, and z is colored T or F, then the widget is 3-colorable if and only if at least one of x, y, or z is colored T.

7 Solving NP-complete Problems

7.1 Methods

- Special cases: For example, 3SAT is **NP**-complete, but 2SAT$\in$**P**
- Clever exhaustive search: Dynamic programming and branch-and-bound.
- Probabilistic/randomized algorithms: Design algorithms which are fast/optimal most of the time with high probability.
- Approximation algorithms: Find near-optimal solutions fast.

7.2 Special cases

- 3DM is **NP**-complete, but 2DM is in **P**.
- Maximum bipartite matching: The optimization version of 2DM.

  Given a bipartite graph $G = (L \cup R, E)$. Find the maximum size matching $M \subseteq E$ such that no two edges in M share endpoints.
- A polynomial-time algorithm

  G (a bipartite graph) $\Rightarrow G'$ (a flow network with arcs, unit-capacity, source and sink).

  There is a maximum matching with size k iff there is a maximum flow with amount k.

  Use Ford-Fulkerson's algorithm for the maximum flow problem.

7.3 Clever exhaustive search

- Dynamic programming.

  TSP: Given an edge-weighted $G = (V, E, w)$. Find a tour with the minimum total length.

  Naive exhaustive search: Check all $n!$ possible tours. $O(n!n)$.

  Dynamic programming: Assume $S \subseteq \{2, \ldots, n\}$ and $i \notin S$. Define $C(S,i)$ to be the minimum length of all simple paths $1 \to S \to i$. Obviously, $C(\emptyset, i) = w(1,i)$ and $C(S,i) = \min_{k \in S}\{C(S - \{k\}, k) + w(k,i)\}$. The solution to TSP is then $C(\{2,3,\ldots,n\},1)$. In the table for $C(S,i)$, there are about $O(2^n)$ rows for all possible subsets of $\{2,\ldots,n\}$, and n columns for all possible values for i. The time to compute each entry is about $O(n)$. So the time complexity of the algorithm is $O(n^2 2^n)$.
- Branch-and-bound.

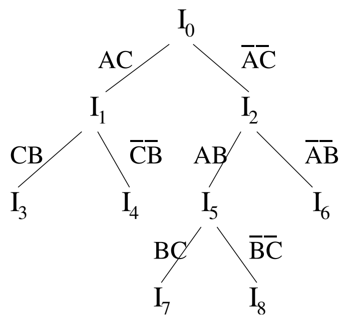
  DTSP: Given an edge-weighted directed $G = (V, E, w)$. Find a tour with the minimum total length.

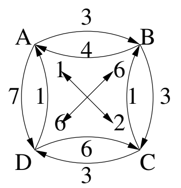
  Branch-and-bound: Define a lower bounding function lb: $\frac{1}{2}\sum_{v_i}$(shortest arc entering v_i + shortest arc leaving v_i). At any time in constructing the search tree, always choose a node with the smallest bound to expand, since that is the most promising node. Also, there is no need to expand a node if its bound is larger than the length of the current shortest tour.

  I_0 (the initial graph): $lb(I_0) = \frac{1}{2}(3+4+3+4) = 7$.

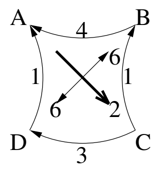
  I_1 (use arc AC in the tour, and remove all useless arcs): $lb(I_1) = \frac{1}{2}(3+5+3+4) = 7.5$.

  I_2 (not use arc AC in the tour, and remove AC): $lb(I_2) = \frac{1}{2}(4+4+4+4) = 8$.

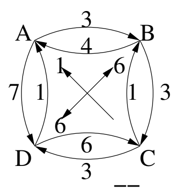




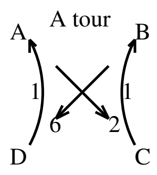
I_0 (7)



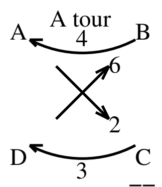
I_1 (7.5): AC



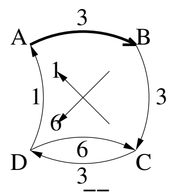
I_2 (8): $\overline{AC}$



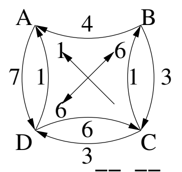
I_3 (10): AC CB



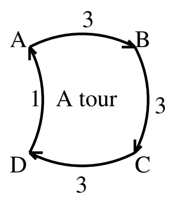
I_4 (15): AC $\overline{CB}$



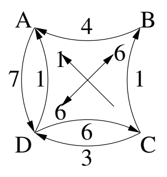
I_5 (9): $\overline{AC}$ AB



I_6 (10): $\overline{AC}$ $\overline{AB}$



I_7 (10): $\overline{AC}$ AB BC



I_8 (12): $\overline{AC}$ AB $\overline{BC}$

7.4 Randomization

- Power of randomization:

  Treasure hunter at C. Treasure in one of two hiding places A and B. Five days to travel from one place to another. Four days to decipher the map using a computer at C.

  Dragon takes away y from the treasure every day. An elf offers to decipher the map for a price of $3y$.

  Let x be the value of the treasure today.

  Option 1: Spend four days to solve the mystery and five days to travel to the hiding place. $x - 9y$.

  Option 2: Accept elf's offer. $x - 3y - 5y = x - 8y$.

  Option 3. Flip a coin to choose from A and B randomly. With probability 0.5, the treasure hunter gets $x - 5y$, and probability 0.5, he gets $x - 10y$. But the average is $0.5(x - 5y) + 0.5(x - 10y) = x - 7.5y$.

- Randomized linear search:

  Assume that L is an unsorted list. Wish to determine that if a given x is a member of L. In linear search, we start with $L[1]$. In the worst case when $x = L[n]$, we will have to search the entire list. However, the worst case becomes the best case if the search is from the right to the left.

  In randomized linear search, we first flip a coin. If we get the "Head", we search from left to right. If we get the "Tail", we search from right to left. There is a 50% chance to avoid having to search the entire list.

- Primality testing:

  Prime vs. composite.

  Finding the factors of n is much harder than verifying if $n = a \times b$ for some a and b.

  In 1903, Frank Cole presented a paper without saying a word:

$$2^{67} - 1 = 147573952589676412927 = 193707721 \times 761838257287.$$

  It only takes minutes to verify the factorization. However, it took Cole three years of Sundays to find the factorization.

A conventional algorithm: If n is composite, then there exist a and b such that $n = a \times b$. Let $a = \min\{a, b\}$. Then $a \le \sqrt{n}$. Based on this fact, we can design a primality testing algorithm with time complexity $O(\sqrt{n}) = O(2^{d/2})$, which is exponential in d, the number of digits in the binary representation of n.

Fermat's Little Theorem: If n is prime, then for any a between 1 and $n-1$, we have $a^{n-1} \bmod n = 1$.

Equivalently, if there exists a between 1 and $n-1$ such that $a^{n-1} \bmod n \neq 1$, then n is composite.

Probabilistic primality testing:

```
Do $k$ times
  Choose $a$ between 1 and $n-1$ randomly;
  If $a^{n-1}$ mod $n \neq 1$
    $n$ is composite and return;
$n$ is prime and return;
```

The algorithm occasionally makes mistakes. It is a Monte Carlo algorithm.

How does one compute a^{n-1} mod n efficiently?

```
$expomod(a,n,m)$ //Computes $a^n$ mod $m$
  $i \leftarrow n$; $r \leftarrow 1$; $x \leftarrow a$ mod $m$;
  While $i > 0$
    If $i$ is odd $r \leftarrow rx$ mod $m$;
    $x \leftarrow x^2$ mod $m$;
    $i \leftarrow i/2$;
  Return $r$;
```

Clearly, the time complexity of the randomized primality testing algorithm is $O(k \log n) = O(kd)$.

- Randomized algorithms vs. probabilistic algorithms:

  Randomized algorithms: Usually fast; Never lies; Always stops; Usually unpredictable; Uses random numbers.

  Probabilistic algorithms: Always fast; Usually tells the truth; Usually stops; May be unpredictable; May use random numbers.

- Three types of probabilistic algorithms:

  - Numerical probabilistic algorithm: Always gives an approximate solution. For example, with probability 90%, the correct answer is 59 plus or minus 3. The more time you allow such numerical algorithms, the more precise the answer is.

    For example, given a needle of length l and a floor of planks of width $w = 2l$. Throw the needle on the floor. It can be proved that the probability that the needle will fall across a crack (always of width 0) is $1/\pi$. One can use this result to estimate π: (1) Throw the needle n times; (2) Count the number of times, k, that it falls across a crack; and (3) Compute n/k. When $n \to \infty$, $k/n \to 1/\pi$, thus $n/k \to \pi$. The larger n is, the more precise the estimation is.

  - Monte Carlo: Always gives an answer efficiently, but may err with a small probability.

    For example, given three $n \times n$ matrices A, B, and C, one wishes to verify if $A \times B = C$ without computing $A \times B$. (Note: The best known matrix multiplication algorithm takes time $\Theta(n^{2.37})$.)

```
function $VerifyMult(A,B,C,n)$        //$\Theta(n^2)$
    for $j \leftarrow 1$ to $n$ $X_j \leftarrow random(0,1)$
    if $(XA)B = XC$ return 1
    else return 0
```

    Analysis: For each instance, $VerifyMult$ returns a correct answer with probability at least $1/2$. So $VerifyMult$ is $1/2$-correct. (The proof is omitted.) An easier algorithm can achieve the same error rate by flipping a coin without even looking at the three matrices in question! What makes $VerifyMult$ worthy is that if it returns 0, this answer is always correct.

```
function $RepeatVerifyMult(A,B,C,n,k)$        //$\Theta(kn^2)$
    for $i \leftarrow 1$ to $k$
        if $VerifyMult(A,B,C,n) = 0$ return 0
        return 1
```

Analysis: For $RepeatVerifyMult$, if $AB = C$, the error probability is 0. On the other hand, if $AB \neq C$, the probability that any call of $VerifyMult$ returns an incorrect 1 is at most $1/2$. So the probability that k successive calls each return an incorrect 1 is at most $1/2^k$. Since this is the only way for $RepeatVerifyMult$ to return an erroneous answer, therefore $RepeatVerifyMult$ is $(1 - 1/2^k)$-correct.

```
function $VerifyMultEpsilon(A,B,C,n,\varepsilon)$        //$\Theta(n^2 \log 1/\varepsilon)$
    $k \leftarrow \lceil \log 1/\varepsilon \rceil$
    return $RepeatVerifyMult(A,B,C,n,k)$
```

Analysis: In $VerifyMultEpsilon$, k is $\log 1/\varepsilon$. Therefore, the algorithm is $(1-\varepsilon)$-correct, or equivalently, the error probability of the algorithm is ε.

- Las Vegas: Always gives a correct answer once it terminates, but may never terminate for some inputs.

  For example, when did Christopher Columbus discover America?

  An algorithm is run 5 times and gives the following answers: 1492, 1492, —, 1492, 1492, —. This algorithm is a Las Vegas algorithm.

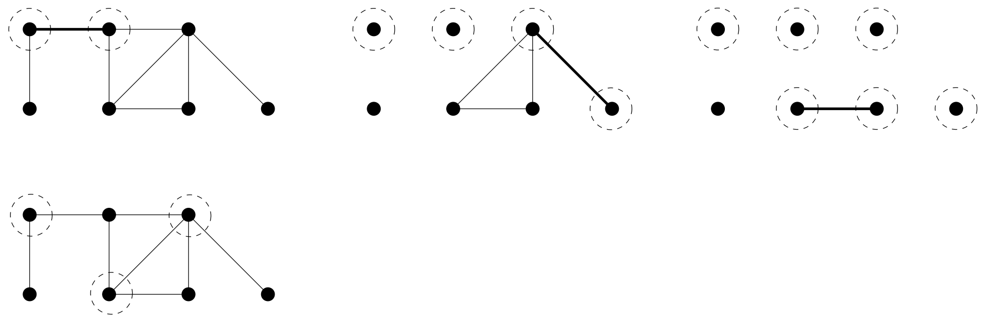
7.5 Approximation algorithms

- Two steps to design an efficient (both in time and performance) approximation algorithm for an NP-complete optimization problem:
  - Unravel the algorithmically relevant combinatorial structure of the problem; and
  - Find algorithmic techniques that can exploit this structure.
- Notation:
  - Π: An **NP**-complete optimization problem.
  - I: Any instance of the problem.
  - OPT: The exponential-time optimal algorithm. (You don't have to know what it is.)
  - A: A polynomial-time approximation algorithm.
  - $OPT(I)$: Optimal solution (value) for I obtained by applying OPT.
  - $A(I)$: Approximation solution (value) for I when A is used.
- Analysis of the performance of approximation algorithms, using minimization problems as example:
  - Performance ratio: $R_A = \max_{\forall I}\{A(I)/OPT(I)\}$.
  - Or almost equivalently, determine the smallest possible factor r such that for any I, $A(I) \leq r \cdot OPT(I) + c$ for some fixed constant c. Here, we say that the algorithm is a factor r approximation algorithm.
  - Lower bounding $OPT(I)$: For an NP-complete problem, not only is it intractable to find the optimal solution for any instance of the problem, but it is also hard to compute the value of the objective function for the optimal solution. So there is a need to determine a nontrivial lower bound to $OPT(I)$ for the purpose of analyzing an approximation algorithm.
- VERTEX COVER

  Define an approximation algorithm A as follows:

```
$C \leftarrow \emptyset$; $E' \leftarrow E$;
While $E' \neq \emptyset$ do
    $C \leftarrow C \cup \{u,v\}$ for some $(u,v) \in E'$;        (*)
    Remove from $E'$ every edge incident to either $u$ or $v$;
Return $C$;
```

For example, $A(G) = |C| = 6$ and $OPT(G) = 3$.



Analysis: The time complexity of A is $O(|E|)$.

Theorem: $R_A^n \leq 2$.

Proof. Let S be the set of edges that were picked in step (*). $|C| = 2|S|$. Let C^* be the minimal vertex cover for the same instance. $|C^*| \geq |S|$ since the edges in S don't share any endpoints. So $R_A^n = \max\{|C|/|C^*|\} \leq 2$. This bound is tight since there exists an instance (with n nodes and just one edge) that achieves the ratio 2.

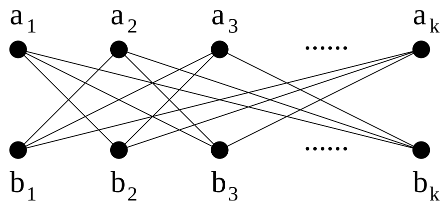
- GRAPH COLORING

Define an approximation algorithm A as follows:

```
For $i \leftarrow 1$ to $n$ do
    $c \leftarrow 1$;
    While there is a vertex adjacent to $v_i$ with color $c$
        $c \leftarrow c+1$;
    color $v_i$ with $c$;
```

For example in the figure below, if $V = \{a_1, \ldots, a_k, b_1, \ldots, b_k\}$, then two colors are needed to color the nodes (optimal), and if $V = \{a_1, b_1, \ldots, a_k, b_k\}$, then k colors (bad).

Analysis: Time complexity $O(n^2)$. Performance ratio $R_A^n \geq \frac{k}{2} = \frac{n}{4}$. Asymptotic performance ratio $R_A^\infty = \lim_{n\to\infty} R_A^n = \infty$.



Approximate graph coloring is hard. In fact, no polynomial-time algorithm exists for which R_A^n is bounded by a constant.

Theorem: GRAPH 3-COLORABILITY$\in$**P** if there is a polynomial-time graph coloring algorithm A such that $R_A^n < \frac{4}{3}$.

Proof. Apply A to an instance G. If G is 3-colorable, then $OPT(G) \leq 3$ and $A(G) < \frac{4}{3}OPT(G) \leq 4$. So $A(G) \leq 3$. If G is not 3-colorable, then $A(G) \geq OPT(G) > 3$. Therefore, G is 3-colorable if and only if $A(G) \leq 3$. So we can use A to solve GRAPH 3-COLORABILITY in polynomial time.

- SET COVERING

INSTANCE: A finite set X and a family F of subsets of X such that $X = \cup_{S\in F} S$.

GOAL: Determine $C \subseteq F$ with the minimum size such that $X = \cup_{S\in C} S$, i.e., members of C cover all of X.

The decision version of SET COVERING is **NP**-complete. (Reduce from VERTEX COVER.)

An example of the SET COVERING problem: $X = \{1,2,3,4,5,6,7,8,9,10,11,12\}$ and $F = \{S_1, S_2, S_3, S_4, S_5, S_6\}$, where $S_1 = \{1,2,5,6,9,10\}$, $S_2 = \{6,7,10,11\}$, $S_3 = \{1,2,3,4\}$, $S_4 = \{3,5,6,7,8\}$, $S_5 = \{9,10,11,12\}$, and $S_6 = \{4,8\}$. The optimal solution is $C = \{S_3, S_4, S_5\}$.

An approximation algorithm based on the greedy strategy:

```
GREEDY-SET-COVER($X, F$)
    $U \leftarrow X$
```

```
$C \leftarrow \emptyset$
while $U \neq \emptyset$ do
    select an $S \in F$ that maximizes $|S \cap U|$
    $U \leftarrow U - S$
    $C \leftarrow C \cup \{S\}$
return $C$
```

Applying GREEDY-SET-COVER to the example, we have a cover C of size 4 containing S_1, S_4, S_5, S_3.

Theorem: For any instance, GREEDY-SET-COVER finds a cover C of size no larger than $H(\max\{|S| : S \in F\})$ times the optimal size. (Note: $H(d) = \sum_{i=1}^{d} 1/i$ is the dth harmonic number.)

Proof. Let S_i be the ith subset selected by GREEDY-SET-COVER. We assume that the algorithm incurs a cost of 1 when it adds S_i to C. We spread this cost of selecting S_i evenly among the elements covered for the first time by S_i. Let c_x denote the cost allocated to element $x \in X$. Each element is assigned a cost only once, when it is covered for the first time. If x is covered for the first time by S_i, then

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$

The algorithm finds a solution C of total cost $|C|$, and this cost has been spread out over the elements of X. Therefore, since the optimal cover C^* also covers X, we have

$$|C| = \sum_{x \in X} c_x \leq \sum_{S \in C^*} \sum_{x \in S} c_x.$$

The remainder of the proof rests on the following key inequality. For any $S \in F$,

$$\sum_{x \in S} c_x \leq H(|S|).$$

Suppose the above is true. We have

$$|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x \leq \sum_{S \in C^*} H(|S|) \leq |C^*| \cdot H(\max\{|S| : S \in F\}).$$

Now let us focus on the proof of the inequality $\sum_{x \in S} c_x \leq H(|S|)$. For any $S \in F$ and $i = 1, \ldots, |C|$, let

$$u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|$$

be the number of elements in S remaining uncovered after $S_1, \ldots, S_i$ have been selected by the algorithm. We define $u_0 = |S|$ to be the number of elements in S, which are initially uncovered. Let k be the least index such that $u_k = 0$, so that every element in S is covered by at least one of the sets $S_1, \ldots, S_k$. Then $u_{i-1} \geq u_i$ and $u_{i-1} - u_i$ elements of S are covered for the first time by S_i for $i = 1, \ldots, k$. Thus

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \frac{1}{|S_i - (S_1 \cup \cdots \cup S_{i-1})|}.$$

Observe that

$$|S_i - (S_1 \cup \cdots \cup S_{i-1})| \geq |S - (S_1 \cup \cdots \cup S_{i-1})| = u_{i-1},$$

because the greedy choice of S_i guarantees that S cannot cover more new elements than S_i does. So we obtain

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \frac{1}{u_{i-1}}.$$

For integer a and b with $a < b$,

$$H(b) - H(a) = \sum_{i=a+1}^{b} \frac{1}{i} \geq (b-a)\frac{1}{b}.$$

Using this inequality,

$$\sum_{x\in S} c_x \le \sum_{i=1}^{k}(H(u_{i-1})-H(u_i)) = H(u_0)-H(u_k) = H(u_0) = H(|S|).$$

Corollary: GREEDY-SET-COVER has a performance ratio of $\ln|X|+1$.

- Parallel Job Scheduling with Overhead
  - Motivation: In a parallel system, a job can be assigned to an arbitrary number of processors to execute. The more processors assigned to the job, the less the computation time. This is the linear speedup in computation. However, having multiple processors working on a job will incur an overhead cost such as communication and synchronization. The more processors, the more the overhead cost. This is the linear slowdown in overhead.
  - A mathematical model: Assume job J_j has processing requirement of p_j. This is given. If in scheduling J_j, k_j processors are assigned, then the execution time $t_j = p_j/k_j + (k_j-1)c$, where the term p_j/k_j is the computation time and the term $(k_j-1)c$ is the overhead with c being the constant overhead per processor. Note that if $k_j = 1$, $t_j = p_j$. This is the scheduling of sequential jobs.
  - Problem formulation:

    Input: A parallel system of m identical processors and a sequence of n independent jobs $J_1,\dots,J_n$ where p_j is the processing requirement of job J_j.

    Output: A schedule with the minimum makespan.
  - An online scheduling algorithm processes the jobs in the order of $J_1,\dots,J_n$ and for each job J_j, it chooses k_j, the number of processors to execute J_j simultaneously, and s_j, the start time of J_j on some k_j processors.
  - SET: Shortest Execution Time

    For job J_j, define function $t_j(k) = p_j/k + (k-1)c$. $t_j(k)$ is minimized at $k = \sqrt{p_j/c}$. Since k_j has to be an integer in $[1,m]$, let $k_j = \min\{m, \lfloor\sqrt{p_j/c}\rfloor\}$ if $t_j(\lfloor\sqrt{p_j/c}\rfloor) \le t_j(\lceil\sqrt{p_j/c}\rceil)$ and $k_j = \min\{m, \lceil\sqrt{p_j/c}\rceil\}$ otherwise. Once the k_j that minimizes t_j is computed, the job is assigned to the k_j processors that give the job the earliest start time.

    An example: $m = 3$, $n = 6$, and $c = 1$. (In the final schedule, $C = 14$.)

j	1	2	3	4	5	6
p_j	4	1	4	1	9	4
k_j	2	1	2	1	3	2
t_j	3	1	3	1	5	3
s_j	0	0	3	1	6	11

    A lower bound to the competitive ratio for SET:

    Case 1. Even m. Consider an instance with $n = 2\alpha m$ jobs for some positive integer α. For odd index j, let $p_j = (m/2)(m/2-1)c+\varepsilon$ and for even index j, let $p_j = (m/2)(m/2+1)c+\varepsilon$. According to SET, for odd j, $k_j = m/2$ and for even j, $k_j = m/2+1$. The makespan of the SET schedule $C = \sum_j(p_j/k_j+(k_j-1)c) = n(m-1)c$. The minimum makespan $C^* \le (1/m)\sum_j p_j = nmc/4$. Thus the competitive ratio is at least $n(m-1)c/(nmc/4) = 4(m-1)/m = 4-4/m$.

    Case 2. Odd m. Consider an instance with $n = \alpha m$ jobs where $p_j = (m+1)/2\cdot(m-1)/2\cdot c+\varepsilon$. According to SET, $k_j = (m+1)/2$. So $C = \sum_j(p_j/k_j+(k_j-1)c) = n(m-1)c$ and $C^* \le (1/m)\sum_j p_j = n(m^2-1)c/(4m)$. So the competitive ratio is at least $n(m-1)c/(n(m^2-1)c/(4m)) = 4m/(m+1) = 4-4/(m+1)$.

    The tight competitive ratio of SET is proved to be 4.
  - ECT: Earliest Completion Time

    For job J_j, choose k_j processors and the start time s_j such that the completion time $C_j = s_j+t_j = s_j+p_j/k_j+(k_j-1)c$ is the earliest. For the above example, $k_1 = 2$, $s_1 = 0$, $k_2 = 1$, $s_2 = 0$, $k_3 = 1$, $s_3 = 1$, $k_4 = 1$, $s_4 = 3$, $k_5 = 2$, $s_5 = 4$, $k_6 = 1$, $s_6 = 5$, with $C = 9.5$.

    The competitive ratio for ECT is conjectured to be $30/13$.

8 Additional Topic 1: DNA Computing

8.1 The Hamiltonian path problem (HPP)

- Given a directed graph with two node specified as the source and the destination. The Hamiltonian path is one that starts at the source node and ends at the destination node such that each node in the graph appears once and only once on the path.
- The Hamiltonian path problem is to determine whether there is a Hamiltonian path for a directed graph and maybe furthermore, find a Hamiltonian path if one exists.
- An example of a graph of seven nodes with node 0 as the source and node 6 as the destination, which was used in Adleman's DNA experiment.
- HPP is NP-complete. Using exhaustive search, one has to generate all $n!$ possible paths (for a graph with n nodes) and check for each path if it is a Hamiltonian path. Even for a graph of only 10 nodes, the number of possible paths is $10! = 3,628,800$.
- An idea for fast algorithms: Use parallel computation to test all possible paths at the same time. The power of a parallel computer (a few thousands simultaneous computations) is not enough.

8.2 A five-step algorithm used in Adleman's experiment

- Step 1. Generate a large number of paths through the graph.
- Step 2. Keep only those that start with the source and end with the destination.
- Step 3. Keep only those of length n.
- Step 4. Keep only those that pass through each node once.
- Step 5. If there are any paths left, any of them will be a Hamiltonian path.

The algorithm is not deterministic since in Step 1 not all paths of length 7 (or n in general) are to be generated. Adleman estimated that his DNA computation generated about 10^{14} paths, so the overwhelming likelihood was that any given path of length 7 would be produced many times over.

8.3 DNA 101

- DNA is the storage medium for genetic information.
- Four kinds of *bases* that form our alphabet: adenine (A), thymine (T), guanine (G) and cytosine (C).
- A single DNA strand with b bases is a string of length b over the alphabet $\{A,T,G,C\}$. The two ends of a strand are marked with 3' and 5', respectively. So a DNA strand is considered oriented: 5' to 3' or 3' to 5'.
- Two complimentary pairs: A-T and G-C, caused by mutual attraction (hydrogen bonding) between A and T and between G and C.
- The Watson-Crick complementary (dual or mirror image): 5'GCTATT3' and 3'CGATAA5'.
- A DNA molecule consists of two intertwined complementary strands made up with four bases, A, T, G, C. This is called double helix structure, discovered by Watson and Crick.
- Heat (90° C) separates the double strands and cooling bonds them back.
- A typical human DNA molecule is about three billion bases long. But synthetic DNA (called oligonucleotide or oligo for short) may be ten to a hundred bases long.

8.4 Encoding the graph into DNA strands

- Nodes: For each node $i = 0, 1, \ldots, 6$, Adleman chose a random 20-base strand of DNA, O_i, to represent the node. For example,

$$O_2 = 5'\text{TATCGGATCGGTATATCCGA}3'$$

$$O_3 = 5'\text{GCTATTCGAGCTTAAAGCTA}3'$$

$$O_4 = 5'\text{GGCTAGGTACCAGCATGCTT}3'$$

- Edges: For each edge $i \to j$, where $i \neq 0$ (i.e., i is not the source) and $j \neq 6$ (i.e., j is not the destination), Adleman created a 20-base strand $O_{i\to j}$ that consisted of the last ten bases of O_i and the first ten bases of O_j. For example,

$$O_{2\to 3} = 5'\text{GTATATCCGAGCTATTCGAG}3'$$

$$O_{3\to 4} = 5'\text{CTTAAAGCTAGGCTAGGTAC}3'$$

  In the case of $i = 0$ (the source), $O_{i\to j}$ was all of O_i followed by the first ten bases of O_j. And in the case of $j = 6$ (the destination), $O_{i\to j}$ was the last ten bases of O_i followed by all of O_j. (Why?)

- Paths: To join edges to form a path, DNA strands need to be bonded together.

  Ligation reaction: First the strands to be joined were held together temporarily by a "molecule splint". Then they were bonded together permanently by the action of an enzyme that occurs in living cells called ligase. The second step is really a case of letting nature take its course: provided the two strands are held together for a sufficient length of time, the permanent bond will form.

  The molecule splint to ligate $O_{i\to j}$ and $O_{j\to k}$ is the Watson-Crick complementary of O_j, denoted by $\overline{O}_j$.

  What would the Hamiltonian path $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$ look like in the form of a DNA molecules?

8.5 The experiment step by step

- Step 1. Generate paths

  For each node i, get 50 pmol of $\overline{O}_i$. For each edge $i \to j$ in the graph, get 50 pmol of $O_{i\to j}$. Mix all together ($\frac{1}{50}$ tsp.) in a single ligation reaction. DNA molecules were formed that encoded a large number of random paths in the graph.

- Step 2. From node 0 to node 6

  This step is achieved by making many copies of the DNA molecules that encode the paths from node 0 to node 6 (instead of throwing away those paths that do not satisfy the requirement). The process is called amplification by *polymerase chain reaction* (PCR).

  In general, to amplify a DNA with a strand xyz and another strand $\overline{xyz}$, heat is used to separate strands. Then primers x and $\overline{z}$ are added to the solution to start PCR. First x is bonded with $\overline{xyz}$ and $\overline{z}$ is bonded with xyz. Then the DNA polymerase, an enzyme in living cells, facilitates the growth of missing parts. At the end, two copies of the DNA molecules are formed.

  To amplify (multiply) DNA molecules that encode paths from node 0 to node 6, two primers O_0 and $\overline{O}_6$ are used in the PCR process.

- Step 3. Length of 7

  A standard technique to separate DNA of different lengths known as *gel electrophoresis* is used. First the DNA mixture is put on one end of a sheet of sugar gel. A uniform electric charge is then applied to the two ends of the gel, negative at the end containing the DNA. The electric charge causes the DNA molecules to migrate toward the positive end. The shorter the DNA molecule, the faster it moves. After charging for a while, the DNA is separated into a spectrum ranging from the longest at the negative end and the shortest at the positive end.

  How can the DNA with 140 bases be identified? Do a separate, yet simultaneous gel run using a DNA mixture of only 140-base molecules, and then use the new position of this 140-base only mixture as a measuring stick to identify the 140-base molecules in the original mixture.

  Finally, the part of the gel containing the DNA with 140 bases (corresponding to paths of length 7) is cut off and the DNA is extracted from the sugar gel and purified.

- Step 4. Each node at least once

  Since each path already contains node 0 and node 6, only five nodes, $1,2,3,4,5$, need to be checked. The following method, known as *affinity purification*, identifies those paths containing node i and should be applied five times for $i = 1,2,3,4,5$.

  A $\overline{O_i}$ probe is a system that attaches the oligo $\overline{O_i}$ to a magnetic bead. First place a magnet alongside the test tube to draw and hold all the beads. Then pour in separated DNA strands (heated after Step 3). Only those strands with O_i in the sequences are drawn to the probes thus held on the magnet while the remaining contents of the tube are poured away. Next use heat to separated the DNA strands (with O_i) from the beads. Finally pour out the solution. It contains all strands with O_i in the sequences.

- Step 5. Read the result

  Amplify the product of Step 4 by PCR and run on a gel. The visible presence of molecules (in the form of a band) on the gel indicates that the graph does have a Hamiltonian path. Had Adleman started with a graph having no Hamiltonian path, the result of the final gel run would not have produced any band in the gel.

8.6 Discussion

- Limitations of Adleman's method
  - Problem-specific: Tailored only to solve HPP
  - Seven days of experiments to solve an instance that can be solved by any human in seconds
  - Not suitable for numerical computation
  - Results not as certain as with an electronic computer.
- Advantages of Adleman's method
  - Massive parallelism to run a large number of trials simultaneously
  - Tremendous storage capacity (1 bit: one cubic nanometer of DNA versus one trillion cubic nanometer on a magnetic medium)
  - HPP is an NP-complete problem. The computational equivalence among NP-complete problems indicates that similar DNA computation may be used to solve other NP-complete problem (Lipton's work)
- A striking comparison
  - 1948, Tom Kilburn, Manchester Mark I (the first programmable computer), to find the largest factor of an integer, 52 minutes, manual labor
  - 1994, Len Adleman, DNA computing, to solve the HPP, 7 days, manual labor

9 Additional Topic 2: Quantum Computing

9.1 Church's Thesis

- Church's Thesis: Any physical (reasonable) computing device can be simulated by a Turing machine in a number of steps polynomial in the resources used by the computing device.
- Quantum computers seem to be a counterexample to the above statement since there are evidences that suggest that quantum computers are more powerful than Turing machines.

9.2 The State of a Quantum System

- Consider a quantum system with n components (electrons), where each component has two states, 0 and 1 (spin-up and spin-down). In the corresponding Hilbert space, there are 2^n orthogonal basis vectors (eigenstates), each representing a binary number of n bits. The true state of the quantum system is one of the eigenstates. However, since measuring the state will change the state, it is never possible to find out the true state of a system, but rather the state of the quantum system before the measurement. Therefore, the state of the quantum system is represented by the superposition of the basis states: $\sum_{i=0}^{2^n-1} a_i|S_i\rangle$, where a_i is a complex amplitude such that $\sum_i |a_i|^2 = 1$ and each $|S_i\rangle$ is a basis vector in the Hilbert space. For any i, $|a_i|^2$ is the probability that the quantum system is in the basis state $|S_i\rangle$. Note that if $a_i = x+iy$ then $|a_i|^2 = (x+iy)(x-iy) = x^2+y^2$.
- For example, when $n = 1$, the state of the quantum system is $a_0|0\rangle + a_1|1\rangle$. This is in fact a quantum bit that is used to form a quantum memory register.

9.3 Quantum Memory Register

- Recall that each quantum bit is a 2-state system. When two such systems (bits) are put together, a new composite quantum system is formed with the basis states to be $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. A general state of this 2-bit memory register is therefore $a_0|00\rangle + a_1|01\rangle + a_2|10\rangle + a_3|11\rangle$. Similarly, an n-bit quantum memory register has 2^n basis states.
- How does a quantum memory register store a number? Each basis state represents a binary number. To store two numbers 1011001 and 0100110, the quantum register stores the state $\frac{1}{\sqrt{2}}(|1011001\rangle + |0100110\rangle)$. (For other basis states, their amplitudes are zero.)
- A quantum register is able to store an exponential amount of classical information in only a polynomial number of quantum bits.

9.4 Quantum Circuit Model

- In 1993, A. C.-C. Yao proved that the quantum circuit model is computationally equivalent to the quantum Turing machine model. This justifies the use of the simpler quantum circuit model in the study of quantum computing.
- In the quantum circuit model, computation is done through a series of quantum gate operations, similarly to the classical circuit model.
- An example of a quantum gate:

$$\begin{aligned}
|00\rangle &\rightarrow |00\rangle \\
|01\rangle &\rightarrow |01\rangle \\
|10\rangle &\rightarrow \frac{1}{\sqrt{2}}(|10\rangle + |11\rangle) \\
|11\rangle &\rightarrow \frac{1}{\sqrt{2}}(|10\rangle - |11\rangle).
\end{aligned}$$

  Suppose our machine is in the superposition of states $\frac{1}{\sqrt{2}}|10\rangle - \frac{1}{\sqrt{2}}|11\rangle$ and we apply the transformation of the gate, the machine will go to the superposition of state $|11\rangle$, since $\frac{1}{\sqrt{2}}|10\rangle - \frac{1}{\sqrt{2}}|11\rangle = \frac{1}{2}(|10\rangle + |11\rangle) - \frac{1}{2}(|10\rangle - |11\rangle) = |11\rangle$.

9.5 The Number Theory Behind Shor's Factorization Algorithm

- Construct a function $f_{n,x}(a) = x^a \bmod n$, where n is the odd integer to be factored and x is any random number that is relatively prime to n, i.e., $gcd(x,n) = 1$.
- Function $f_{n,x}(a)$ is periodic. As a increases, the values of the function eventually fall into a repeating pattern. Let r be the smallest positive integer such that $f_{n,x}(r) = 1$ (i.e., $x^r \bmod n = 1$). r is called the period of the function and the order of $x \bmod n$. Try $n = 9$ and $x = 4$. Then $f_{n,x}(1,2,3,4,5,6,\ldots) = 4,7,1,4,7,1,\ldots$. Thus $r = 3$.
- If r can be found (it always exists for x and n with $gcd(x,n) = 1$), then n can be factored with a high probability of success. Since $x^r \equiv 1 \bmod n$, then $x^r - 1 \equiv 0 \bmod n$. In the case that r is even, $(x^{r/2} - 1)(x^{r/2} + 1) \equiv 0 \bmod n$. Unless $x^{r/2} \equiv \pm 1 \bmod n$, at least one of $x^{r/2} - 1$ and $x^{r/2} + 1$ has a nontrivial factor in common with n. Thus we have a good chance of finding a factor of n by computing $gcd(x^{r/2} - 1, n)$ and $gcd(x^{r/2} + 1, n)$.
- It was shown that the above procedure, when applied to a random x, yields a nontrivial factor of n with probability at least $1 - 1/2^{k-1}$, where k is the number of distinct odd prime factors of n. The probability $1/2^{k-1}$ is for the cases that the above method does not work: r is odd or $x^{r/2} \equiv \pm 1 \bmod n$.
- There is no known way of calculating the required period/order efficiently on any classical computer. However, Shor discovered an efficient algorithm to do so using a quantum computer.

9.6 Some Basics and Facts You Must Know to Understand Shor's Algorithm

- Measuring a quantum system: The result is also one of the basis state. Which state depends on the probability, or $|a_i|^2$. Moreover, the measurement of a subset of the quantum bits in the register projects out the state of the whole register into a subset of eigenstates consistent with the answers obtained for the measured part. For example, if the state of a two bit quantum system is $\frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$, after measuring the second bit with a result of 1, the state becomes $\frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|11\rangle$.
- Modular exponentiation: There exists a quantum circuit (gate array) that takes only $O(l)$ space and $O(l^3)$ time to compute $x^a \bmod n$ from a, where a, x, and n are l-bit numbers and x and n are relatively prime. For details, see Shor's paper on prime factorization.
- Quantum Fourier transform: There exists a quantum circuit (gate array) that performs a Fourier transform. That is, given a state $|a\rangle$, the transformation takes it to the state $\frac{1}{\sqrt{q}}\sum_{c=0}^{q-1} exp(2\pi iac/q)|c\rangle$ for some q. ($exp(i\theta) = \cos(\theta) + i\sin(\theta)$, $|exp(i\theta)|^2 = 1$.)

  Briefly, Fourier transforms map functions in the time (spatial) domain to functions in the frequency domain, i.e., any function in the time domain can be represented as the sum of some periodic functions in the frequency domain. The frequency is the inverse of the period. When the function is periodic, its Fourier transform will be peaked at multiples of the inverse period $1/r$. When Fourier transform is done in a quantum system, the state corresponding to integer multiples of $1/r$ appears with great amplitudes. Thus if you measure the state, you would be highly likely to get a result close to some multiple of $1/r$.

9.7 Shor's Algorithm for Factoring n: The Big Picture

- Step 1: Pick a number q to be a power of 2.
- Step 2: Pick a random x that is relatively prime to n.
- Step 3: Create a quantum memory register and partition its quantum bits into two sets, register 1 and register 2, with its composite state represented by $|reg1, reg2\rangle$.
- Step 4: Load register 1 with all integers from 0 to $q-1$ and load register 2 with all zeroes. The state of the complete register is

$$|\Phi\rangle = \frac{1}{\sqrt{q}}\sum_{a=0}^{q-1}|a,0\rangle.$$

- Step 5: Apply, in quantum parallel, the transformation $x^a \bmod n$ to each number in register 1 and place the result in register 2. The state of the complete register is

$$|\Phi\rangle = \frac{1}{\sqrt{q}} \sum_{a=0}^{q-1} |a, x^a \bmod n\rangle.$$

- Step 6: Measure the state of register 2, obtaining some integer k. This has the effect of projecting out the state of register 1 to be a superposition of just those values of a such that $x^a \bmod n = k$. Let A be the set of all such a's. The state of the complete register is

$$|\Phi\rangle = \frac{1}{\sqrt{|A|}} \sum_{a \in A} |a, k\rangle.$$

- Step 7: Apply Fourier transform to the state in register 1. The state of the complete register is

$$|\Phi\rangle = \frac{1}{\sqrt{|A|}} \sum_{a \in A} \frac{1}{\sqrt{q}} \sum_{c=0}^{q-1} exp(2\pi iac/q) |c, k\rangle.$$

- Step 8: Measure the state of register 1. This returns some $c = \lambda/r$.
- Step 9: Repeat Step 3 - Step 8 several times until multiple samples (say k) of $c = \lambda/r$ are obtained. With k equations, $c_1 = \lambda_1/r$, ..., $c_k = \lambda_k/r$, and $k+1$ integer unknowns, $\lambda_1, \ldots, \lambda_k$ and r, we have a good chance to figure out r.
- Step 10: Once r is known, factors of n can be obtained from $gcd(x^{r/2} - 1, n)$ and $gcd(x^{r/2} + 1, n)$.

10 Final Words

10.1 Problem classification

- Undecidable/unsolvable: HALTING, PCP, and Hilbert's Tenth Problem.
- Decidable:
  - Non-**NP**: Requires exponential time even by NTM. Some problems in Automata Theory were proved to be Non-**NP**.
  - **NP**:
    - **NP**-complete: Hardest in **NP**. SAT, CLIQUE, and 3DM.
    - **NP**-intermediate: Exists if **P**$\neq$**NP**.

      If **P**$\neq$**NP**, then **P**$\cup$**NPC**$\subset$**NP** (Ladner, 1975). Define **NPI**$=$**NP**$-$(**P**$\cup$**NPC**).

      Potential members in **NPI**:

      GRAPH ISOMORPHISM:
      INSTANCE: $G=(V,E)$ and $G'=(V,E')$.
      QUESTION: Is there a one-to-one function $f:V\to V$ such that $(u,v)\in E$ iff $(f(u),f(v))\in E'$?

      COMPOSITE NUMBER:
      INSTANCE: Positive integer k.
      QUESTION: Are there integers $m,n>1$ such that $k=mn$?
    - **P**: Easiest in **NP**. SORTING and MST.

The following figure is based on the conjecture that **P**$\neq$**NP**.

10.2 NP-intermediate

- Theorem: If B is recursive and $B\notin$**P**, then $\exists D\in$**P** such that $A=D\cap B\notin$**P**, $A\propto_p B$ and $B\not\propto_p A$.

  Proof. By Ladner(1975).
- Theorem: **NPI**$\neq\emptyset$ iff **P**$\neq$**NP**.

  Proof. "If": Let B be a recursive and **NP**-complete language. Since **P**$\neq$**NP**, then $B\notin$**P**. By Ladner's theorem, $\exists D\in$**P** such that $A=D\cap B\notin$**P**, $A\propto_p B$ and $B\not\propto_p A$. So $A\in$**NP**. Since $B\not\propto_p A$, then $A\notin$**NPC**. So $A\in$**NPI**.

  "Only if": Assume **P**$=$**NP**. Then **NP**$=$**P**$\cup$**NPC**. So **NPI**$=$**NP**$-$(**P**$\cup$**NPC**)$=\emptyset$. Contradiction!

10.3 Randomization revisited

- PTM (Probabilistic TM):

  A NTM which accepts an input if at least $\frac{2}{3}$ of its possible computation paths on the input end in the accepting state.
- **PP**: Class of languages accepted in polynomial time by a PTM.
- **BPP**: Class of languages, each of which has a polynomial-time PTM such that w is in the language if at least $\frac{2}{3}$ of the computations accept w and that w is not in the language if at least $\frac{2}{3}$ of the computations reject w.
- **RP**: A subset of **BPP** whose languages are accepted in polynomial time by some PTM such that the machines may reject strings in the languages, but only at most $\frac{1}{3}$ of the time.
- An analogy: There are two types of court room errors. Type I: Convict the innocent. Type II: Acquit the guilty. Assume that accepting a string $\Leftrightarrow$ convicting a person (accepting a person in a prison) and that rejecting a string $\Leftrightarrow$ acquitting a person.

  PP: To avoid type I.

  BPP: To avoid type I and type II.

  RP: To avoid type I and type II.