Introduction to Computational Complexity

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Subsection 1

Approximation Algorithms

Optimization Problems and Approximation Algorithms

- In **optimization problems** we seek the best solution among a collection of possible solutions.
- Example: We may want to find:
  - A largest clique in a graph;
  - A smallest vertex cover;
  - A shortest path connecting two nodes.
- When an optimization problem is NP-hard, as is the case with the first two problems, no polynomial time algorithm exists that finds the best solution unless $P = NP$.
- In practice, instead of finding the absolute best or **optimal solution**, a solution that is nearly optimal may be good enough and may be much easier to find.
- An **approximation algorithm** is designed to find such approximately optimal solutions.

Approximation Algorithm for MinVertexCover

- The vertex cover problem was introduced as the language VertexCover representing a decision problem, one that has a yes/no answer.
- In the optimization version, called MinVertexCover, we aim to produce one of the smallest vertex covers among all possible vertex covers in the input graph.
- The following polynomial time algorithm approximately solves this optimization problem. It produces a vertex cover that is never more than twice the size of one of the smallest vertex covers.

  A: On input $\langle G \rangle$, where G is an undirected graph:

  1. Repeat the following until all edges in G touch a marked edge:
     2. Find an edge in G untouched by any marked edge.
     3. Mark that edge.
  4. Output all nodes that are endpoints of marked edges.

Proving Correctness of the Algorithm A

Theorem

A is a polynomial time algorithm that produces a vertex cover of G that is no more than twice as large as a smallest vertex cover.

- A obviously runs in polynomial time.
- Let X be the set of nodes that it outputs. Let H be the set of edges that it marks. H contains or touches every edge in G. Thus, X touches all edges in G. Hence, X is a vertex cover.
- To prove that X is at most twice as large as a smallest vertex cover Y, we establish two facts:
  - X is twice as large as H: Every edge in H contributes two nodes to X, so X is twice as large as H.
  - H is not larger than Y: Y is a vertex cover, so every edge in H is touched by some node in Y. No such node touches two edges in H because the edges in H do not touch each other. So, vertex cover Y is at least as large as H, since Y contains a different node that touches every edge in H.

k-Optimal Approximation Algorithms

- In a **minimization problem** we aim to find the smallest among the collection of possible solutions.
- In a **maximization problem** we seek the largest solution.
- An approximation algorithm for a minimization problem is k-**optimal** if it always finds a solution that is not more than k times optimal.

  Example: The preceding algorithm is 2-optimal for the vertex cover problem.
- For a maximization problem a k-**optimal** approximation algorithm always finds a solution that is at least $\frac{1}{k}$ times the size of the optimal.

An Approximation Algorithm for MaxCut

- A **cut** in an undirected graph is a separation of the vertices V into two disjoint subsets S and T. A **cut edge** is one that goes between a node in S and a node in T. An **uncut edge** is an edge that is not a cut edge. The **size** of a cut is the number of cut edges.
- The MaxCut problem asks for a largest cut in a graph G.
  The MaxCut problem is NP-complete.
- The following algorithm approximates MaxCut within a factor of 2.
  B: On input $\langle G \rangle$, where G is an undirected graph with nodes V:
  1. Let $S = \emptyset$ and $T = V$.
  2. If moving a single node, either from S to T or from T to S, increases the size of the cut, make that move and repeat this stage.
  3. If no such node exists, output the current cut and halt.
- This algorithm starts with a (presumably) bad cut and makes local improvements until no further local improvement is possible.
- Although it will not give an optimal cut in general, we show that it does give one that is at least half the size of the optimal one.

Proving Correctness of the Algorithm B

Theorem

B is a polynomial time, 2-optimal approximation algorithm for MaxCut.

- B runs in polynomial time because every execution of Stage 2 increases the size of the cut to a maximum of the total number of edges in G.
- We show that B's cut is at least half optimal:
  Actually, we show B's cut contains at least half of all edges in G:
  Observe that, at every node of G, the number of cut edges is at least as large as the number of uncut edges; Otherwise, B would have shifted that node to the other side. We add up the numbers of cut edges at every node: That sum is twice the total number of cut edges. By the preceding observation, that sum must be at least the corresponding sum of the numbers of uncut edges at every node. Thus, G has at least as many cut edges as uncut edges, and, therefore, the cut contains at least half of all edges.

Subsection 2

Probabilistic Algorithms

Probabilistic Algorithms

- A **probabilistic algorithm** is an algorithm designed to use the outcome of a random process. Typically, it would contain an instruction "flip a coin" and the outcome would influence the algorithm's subsequent execution and output.
- Certain types of problems seem to be more easily solvable by probabilistic algorithms than by deterministic algorithms.
- Making a decision by flipping a coin can sometimes be better than actually calculating, or even estimating, the best choice:
  - Calculating the best choice may require excessive time.
  - Estimating the best option may introduce a bias that invalidates the result.

  Example: Statisticians use random sampling to determine information about the individuals in a large population, such as their tastes or political preferences:

  > Querying all the individuals might take too long, and querying a non-randomly selected subset might tend to give erroneous results.

Probabilistic Turing Machines

- We define the model of a probabilistic Turing machine and give a complexity class associated with efficient probabilistic computation:

Definition (Probabilistic Turing Machine)

A **probabilistic Turing machine** M is a type of nondeterministic Turing machine in which each nondeterministic step is called a **coin-flip step** and has two legal next moves. We assign a probability to each branch b of M's computation on input w as follows: Define the **probability of branch** b to be $\Pr[b] = 2^{-k}$, where k is the number of coin-flip steps that occur on branch b. Define the **probability that** M **accepts** w to be

$$\Pr[M \text{ accepts } w] = \sum_{b \text{ is an accepting branch}} \Pr[b].$$

- The probability that M accepts w is the probability that we would reach an accepting configuration if we simulated M on w by flipping a coin to determine which move to follow at each coin-flip step.
- We let $\Pr[M \text{ rejects } w] = 1 - \Pr[M \text{ accepts } w]$.

Recognition with Error

- When a probabilistic Turing machine recognizes a language, it must accept all strings in the language and reject all strings out of the language as usual, except that now we allow the machine a small probability of error.
- For $0 \leq \epsilon < \frac{1}{2}$, we say that M **recognizes language** A **with error probability** ϵ if
  1. $w \in A$ implies $\Pr[M \text{ accepts } w] \geq 1 - \epsilon$;
  2. $w \notin A$ implies $\Pr[M \text{ rejects } w] \geq 1 - \epsilon$.

  I.e., the probability that we would obtain the wrong answer by simulating M is at most ϵ.
- We also consider error probability bounds that depend on the input length n. For example, error probability $\epsilon = 2^n$ indicates an exponentially small probability of error.

The Class BPP

- We are interested in probabilistic algorithms that run efficiently in time and/or space.
- We measure the time and space complexity of a probabilistic Turing machine in the same way we do for a nondeterministic Turing machine, by using the worst case computation branch on each input:

Definition (The Class BPP)

BPP is the class of languages that are recognized by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$.

- We defined this class with an error probability of $\frac{1}{3}$, but any constant error probability would yield an equivalent definition as long as it is strictly between 0 and $\frac{1}{2}$: This is ensured by the following **amplification lemma**, which provides a simple way of making the error probability exponentially small.

The Amplification Lemma

Lemma (Amplification Lemma)

Let ϵ be a fixed constant strictly between 0 and $\frac{1}{2}$. For any polynomial poly(n), a probabilistic polynomial time Turing machine M_1 that operates with error probability ϵ has an equivalent probabilistic polynomial time Turing machine M_2 that operates with an error probability of $2^{-\text{poly}(n)}$.

- M_2 simulates M_1 by running it a polynomial number of times and taking the majority vote of the outcomes. The probability of error decreases exponentially with the number of runs of M_1 made.
- Consider the case where $\epsilon = \frac{1}{3}$. It corresponds to a box that contains many green and red balls, $\frac{2}{3}$ of one color and the remaining $\frac{1}{3}$ of the other, but unknown predominant color. We can test for that color by sampling and testing which color comes up most frequently.

  If green is accepting and red rejecting computation, M_2 samples the color by running M_1. M_2 errs with exponentially small probability if it runs M_1 a polynomial number of times and picks majority outcome.

Proof of the Amplification Lemma

- Given TM M_1 recognizing a language with an error probability of $\epsilon < \frac{1}{2}$ and a polynomial poly(n), we construct a TM M_2 that recognizes the same language with an error probability of $2^{-\text{poly}(n)}$:
  M_2: On input w:
  1. Calculate k (as detailed below).
  2. Run $2k$ independent simulations of M_1 on input w.
  3. If most runs of M_1 accept, then accept; otherwise, reject.
- We bound the probability that M_2 gives the wrong answer on an input w: Stage 2 yields a sequence of $2k$ results from simulating M_1, each result either correct or wrong. If most of these results are correct, M_2 gives the correct answer. We bound the probability that at least half of these results are wrong.
- Let S be any sequence of results that M_2 might obtain in Stage 2. Let p_S be the probability M_2 obtains S. Say that S has c correct results and w wrong results, so $c + w = 2k$. If $c < w$ and M_2 obtains S, then M_2 outputs incorrectly. We call such an S a *bad sequence*.

Bounding the Probability of Obtaining a Bad Sequence

- If S is any bad sequence then $p_S \leq \epsilon^w(1-\epsilon)^c$, which in turn is at most $\epsilon^k(1-\epsilon)^k$ because $k < w$ and $\epsilon < 1-\epsilon$. Summing p_S for all bad sequences S gives the probability that M_2 outputs incorrectly. We have at most 2^{2k} bad sequences because 2^{2k} is the number of all sequences. Hence $\Pr[M_2 \text{ outputs incorrectly on input } w] = \sum_{\text{bad } S} p_S \leq 2^{2k}\epsilon^k(1-\epsilon)^k = (4\epsilon(1-\epsilon))^k$. We have assumed $\epsilon < \frac{1}{2}$, so $4\epsilon(1-\epsilon) < 1$ and, therefore, the above probability decreases exponentially in k and so does M_2's error probability.
- To calculate a specific value of k that allows us to bound M_2's error probability by 2^{-t} for any $t \geq 1$, we let $\alpha = -\log_2(4\epsilon(1-\epsilon))$ and choose $k \geq \frac{t}{\alpha}$. Then we obtain an error probability of $2^{-\text{poly}(n)}$ within polynomial time.

Primality

- A **prime number** is an integer greater than 1 that is not divisible by positive integers other than 1 and itself.
- A nonprime number greater than 1 is called **composite**.
- A polynomial time algorithm for the problem of testing whether an integer is prime or composite is now known.
- We only describe a much simpler probabilistic polynomial time algorithm for primality testing.
- One way to determine whether a number is prime is to try all possible integers less than that number and see whether any are divisors, also called **factors**. This algorithm has exponential time complexity because the magnitude of a number is exponential in its length.
- The probabilistic primality testing algorithm does not search for factors.

Number Theoretic Notation

- All numbers considered here are integers.
- For any $p > 1$, we say that two numbers are **equivalent modulo** p if they differ by a multiple of p. If numbers x and y are equivalent modulo p, we write $x \equiv y \pmod{p}$.
- We let $x \mod p$ be the smallest nonnegative y where $x \equiv y \pmod{p}$.
- Every number is equivalent modulo p to some member of the set $\mathbb{Z}_p = \{0, \ldots, p-1\}$. For convenience we let $\mathbb{Z}_p^+ = \{1, \ldots, p-1\}$. We may refer to the elements of these sets by other numbers that are equivalent modulo p, as when we refer to $p-1$ by -1.

Theorem (Fermat's Little Theorem)

If p is prime and $a \in \mathbb{Z}_p^+$, then $a^{p-1} \equiv 1 \pmod{p}$.

The Fermat Test for Primality

- If $p = 7$ and $a = 2$, the theorem says that $2^{(7-1)} \mod 7$ should be 1 because 7 is prime.

  The simple calculation $2^{(7-1)} = 2^6 = 64$ and $64 \mod 7 = 1$ confirms this result.

  Suppose that we try $p = 6$ instead. Then $2^{(6-1)} = 2^5 = 32$ and $32 \mod 6 = 2$ gives a result different from 1, implying, by the theorem, that 6 is not prime. This method demonstrates that 6 is composite without finding its factors.
- Fermat's Last Theorem provides a type of "test" for primality, called a **Fermat test**. When we say that p **passes the Fermat test at** a, we mean that $a^{p-1} \equiv 1 \pmod{p}$.

  The theorem states that primes pass all Fermat tests for $a \in \mathbb{Z}_p^+$.
- Since 6 fails some Fermat test, 6 is not prime.

Primes, Pseudoprimes and the Carmichael Numbers

- We want to use, if possible, the Fermat Test to give an algorithm for determining primality.
- Call a number **pseudoprime** if it passes the Fermat tests at all smaller a relatively prime to it.
- With the exception of the infrequent **Carmichael numbers**, which are composite yet pass all Fermat tests, the pseudoprime numbers are identical to the prime numbers.
- We first give a very simple probabilistic polynomial time algorithm that distinguishes primes from composites except for the Carmichael numbers. Then, we present and analyze the complete probabilistic primality testing algorithm.
- A pseudoprimality algorithm that goes through all Fermat tests would require exponential time. The key to the probabilistic polynomial time algorithm is that, if a number is not pseudoprime, it fails at least half of all tests.

Distinguishing Pseudoprimes

- The algorithm works by trying several tests chosen at random. If any fail, the number must be composite. It contains a parameter k that determines the error probability.

  PseudoPrime: On input p:

  1. Select $a_1, \ldots, a_k$ randomly in $\mathbb{Z}_p^+$.
  2. Compute $a_i^{p-1} \mod p$, for each i.
  3. If all computed values are 1, accept; otherwise, reject.

- Correctness:
  - If p is prime, it passes all tests and the algorithm accepts with certainty.
  - If p is not pseudoprime, it passes at most half of all tests. In that case it passes each randomly selected test with probability at most $\frac{1}{2}$. The probability that it passes all k randomly selected tests is at most 2^{-k}.
- Time complexity: The algorithm operates in polynomial time because modular exponentiation is computable in polynomial time.

Square Root Test for Primality

- To convert the algorithm to a primality algorithm, we need a test that avoids the problem with the Carmichael numbers.
  - The underlying principle is that the number 1 has exactly two square roots, 1 and -1, modulo any prime p.
  - For many composite numbers, including all the Carmichael numbers, 1 has four or more square roots.
- Example: ± 1 and ± 8 are the four square roots of 1, modulo 21.
- If a number passes the Fermat test at a, the algorithm finds one of its square roots of 1 at random and determines whether that square root is 1 or -1. If it is not, we know that the number is not prime.
- If p passes the Fermat test at a, $a^{p-1} \mod p = 1$. Thus, we can obtain square roots of 1, since $a^{\frac{p-1}{2}} \mod p$ is a square root of 1. If that value is still 1, we may repeatedly divide the exponent by 2, so long as the resulting exponent remains an integer, and see whether the first number that is different from 1 is -1 or some other number.

The Probabilistic Algorithm for Primality

- Select $k \geq 1$ as a parameter that determines the maximum error probability to be 2^{-k}:

  PRIME: On input p:

  1. If p is even, accept if $p = 2$; otherwise, reject.
  2. Select $a_1, \ldots, a_k$ randomly in $\mathbb{Z}_p^+$.
  3. For each i from 1 to k:
     4. Compute $a_i^{p-1} \mod p$ and reject if different from 1.
     5. Let $p - 1 = st$, where s is odd and $t = 2^h$ is a power of 2.
     6. Compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, \ldots, a_i^{s \cdot 2^h}$ modulo p.
     7. If some element of this sequence is not 1, find the last element that is not 1 and reject if that element is not -1.
  8. All tests have passed at this point, so accept.

Primes are Accepted with Probability 1

- The following two lemmas show that PRIME works correctly.
  - We only need to consider the case when p is odd.
- Say that a_i is a **(compositeness) witness** if the algorithm rejects at either Stage 4 or Stage 7, using a_i.

Lemma

If p is an odd prime number, $\Pr[\text{PRIME accepts } p] = 1$.

- If p is prime, no witness exists, whence no branch of PRIME rejects:
  - If a were a Stage 4 witness, $a^{p-1} \mod p \neq 1$ and Fermat's Little Theorem implies that p is composite.
  - If a were a Stage 7 witness, some b exists in $\mathbb{Z}_p^+$, where $b \not\equiv \pm 1 \pmod{p}$ and $b^2 \equiv 1 \pmod{p}$. Therefore, $b^2 - 1 \equiv 0 \pmod{p}$. Factoring $b^2 - 1$ yields $(b-1)(b+1) \equiv 0 \pmod{p}$, which implies that $(b-1)(b+1) = cp$ for some positive integer c. Because $b \not\equiv \pm 1 \pmod{p}$, both $b-1$ and $b+1$ are strictly between 0 and p. Therefore, p is composite (a multiple of a prime number cannot be expressed as a product of numbers that are smaller than it is).

Composites Accepted with Low Probability I

- Two numbers are **relatively prime** if they have no common divisor other than 1.
- The **Chinese Remainder Theorem** says that, if p and q are relatively prime, a one-to-one correspondence exists between $\mathbb{Z}_{pq}$ and $\mathbb{Z}_p \times \mathbb{Z}_q$. Each number $r \in \mathbb{Z}_{pq}$ corresponds to a pair (a, b), where $a \in \mathbb{Z}_p$ and $b \in \mathbb{Z}_q$, such that

$$r \equiv a \pmod{p}, \quad \text{and} \quad r \equiv b \pmod{q}.$$

Lemma

If p is an odd composite number, $\Pr[\textsc{Prime} \text{ accepts } p] \leq 2^{-k}$.

- We show that, if p is an odd composite number and a is selected randomly in $\mathbb{Z}_p^+$, $\Pr[a \text{ is a witness}] \geq \frac{1}{2}$ by demonstrating that at least as many witnesses as non-witnesses exist in $\mathbb{Z}_p^+$. We do so by finding a unique witness for each non-witness.

Composites Accepted with Low Probability II

- At every non-witness, the sequence computed in Stage 6 is either all 1s or contains -1 at some position, followed by 1s. E.g., 1 itself is a non-witness of the first kind, and -1 is a non-witness of the second kind because s is odd and $(-1)^{s \cdot 2^0} \equiv -1$ and $(-1)^{s \cdot 2^1} \equiv 1$.

  Among all non-witnesses of the second kind, find a non-witness for which the -1 appears in the largest position in the sequence. Let h be that non-witness and let j be the position of -1 in its sequence, where the sequence positions are numbered starting at 0. Hence, $h^{s \cdot 2^j} \equiv -1 \pmod{p}$.

  Because p is composite, either

Case 1: p is the power of a prime or

Case 2: p is the product of two numbers q and r that are relatively prime.

Case 2: p Product of Relatively Primes q, r

- The Chinese remainder theorem implies that some number t exists in $\mathbb{Z}_p$, such that $t \equiv h \pmod{q}$ and $t \equiv 1 \pmod{r}$. Therefore, $t^{s \cdot 2^j} \equiv -1 \pmod{q}$ and $t^{s \cdot 2^j} \equiv 1 \pmod{r}$. Hence t is a witness because $t^{s \cdot 2^j} \not\equiv \pm 1 \pmod{p}$, but $t^{s \cdot 2^{j+1}} \equiv 1 \pmod{p}$. Now that we have one witness, we can get many more. We prove that $dt \mod p$ is a unique witness for each non-witness d by making two observations:
  - $d^{s \cdot 2^j} \equiv \pm 1 \pmod{p}$ and $d^{s \cdot 2^{j+1}} \equiv 1 \pmod{p}$ owing to the way j was chosen. Therefore, $dt \mod p$ is a witness because $(dt)^{s \cdot 2^j} \not\equiv \pm 1$ and $(dt)^{s \cdot 2^{j+1}} \equiv 1 \pmod{p}$.
  - Second, if d_1 and d_2 are distinct non-witnesses, $d_1 t \mod p \neq d_2 t \mod p$. The reason is that $t^{s \cdot 2^{j+1}} \mod p = 1$. Hence, $t \cdot t^{s \cdot 2^{j+1} - 1} \mod p = 1$. Therefore, if $td_1 \mod p = td_2 \mod p$, then $d_1 = t \cdot t^{s \cdot 2^{j+1} - 1} d_1 \mod p = t \cdot t^{s \cdot 2^{j+1} - 1} d_2 \mod p = d_2$.

Thus, the number of witnesses must be as large as the number of nonwitnesses.

Case 1: p is a Power of a Prime

- We have $p = q^e$, where q is prime and $e > 1$. Let $t = 1 + q^{e-1}$. Expanding t^p using the binomial theorem, we obtain $t^p = (1 + q^{e-1})^p = 1 + p \cdot q^{e-1} +$ multiples of higher powers of q^{e-1}, which is equivalent to $1 \mod p$. Hence t is a Stage 4 witness because, if $t^{p-1} \equiv 1 \pmod{p}$, then $t^p \equiv t \not\equiv 1 \pmod{p}$. As in the previous case, we use this one witness to get many others.
  - If d is a non-witness, we have $d^{p-1} \equiv 1 \pmod{p}$, whence $dt \mod p$ is a witness.
  - Moreover, if $d_1 \neq d_2$ are non-witnesses, then $d_1 t \mod p \neq d_2 t \mod p$; Otherwise

$$d_1 = d_1 \cdot t \cdot t^{p-1} \mod p = d_2 \cdot t \cdot t^{p-1} \mod p = d_2.$$

Thus, the number of witnesses must be as large as the number of non-witnesses, and the proof is complete.

PRIMES and One-Sided Error

- Define

$$\text{PRIMES} = \{n : n \text{ is a prime number in binary}\}.$$

- The preceding algorithm and its analysis prove the following:

Theorem

PRIMES $\in$ BPP.

- Note that the probabilistic primality algorithm has **one-sided error**.
  - When the algorithm outputs reject, we know that the input must be composite.
  - When the output is accept, we know only that the input could be prime or composite.

Thus, an incorrect answer can only occur when the input is a composite number.

The Class RP

- The one-sided error feature is common to many probabilistic algorithms, so deserves a special complexity class RP:

Definition (The Class RP)

RP is the class of languages that are recognized by probabilistic polynomial time Turing machines where:

- Inputs in the language are accepted with a probability of at least $\frac{1}{2}$;
- Inputs not in the language are rejected with a probability of 1.

- We can make the error probability exponentially small, while maintaining a polynomial running time, by using a probability amplification technique.
- Our earlier algorithm shows that $\textsc{Composites} \in \text{RP}$.

Branching Programs and their Equivalence

- A branching program is a model of computation:
  It represents a decision process that queries the values of input variables and makes decisions about the way to proceed based on the answers to those queries.
- We represent this decision process as a graph whose nodes correspond to the variables queried at particular points in the process.
- We focus on the complexity of testing whether two branching programs are equivalent.
- In general, the problem is coNP-complete.
- If we restrict the class of branching programs, we can give a probabilistic polynomial time algorithm for testing equivalence.
- The algorithm is especially interesting because:
  - No polynomial time algorithm is known for this problem;
  - It introduces the technique of assigning non-Boolean values to normally Boolean variables in order to analyze the behavior of some Boolean function of those variables.

Branching Programs

Definition (Branching Program)

A **branching program** is a directed acyclic graph where all nodes are labeled by variables, except for two **output nodes** labeled 0 or 1. The nodes that are labeled by variables are called **query nodes**.

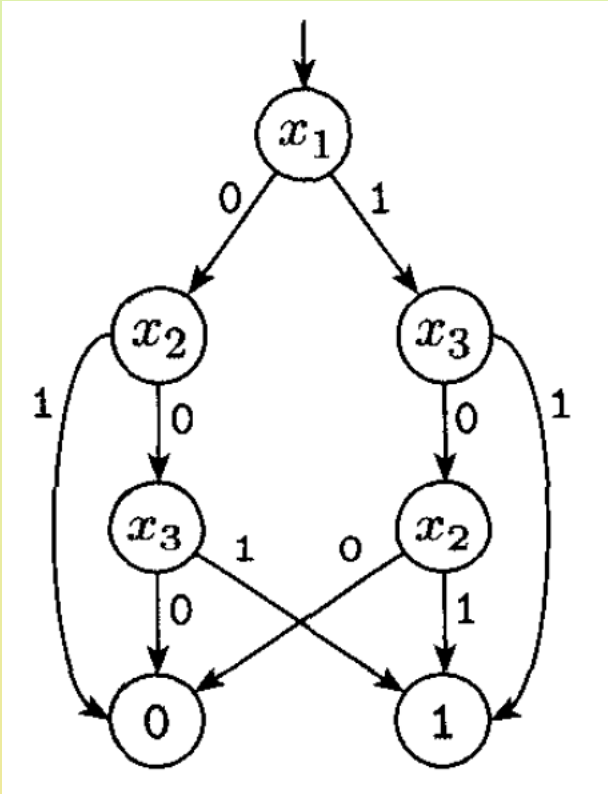
- Every query node has two outgoing edges, one labeled 0 and the other labeled 1.
- Both output nodes have no outgoing edges.

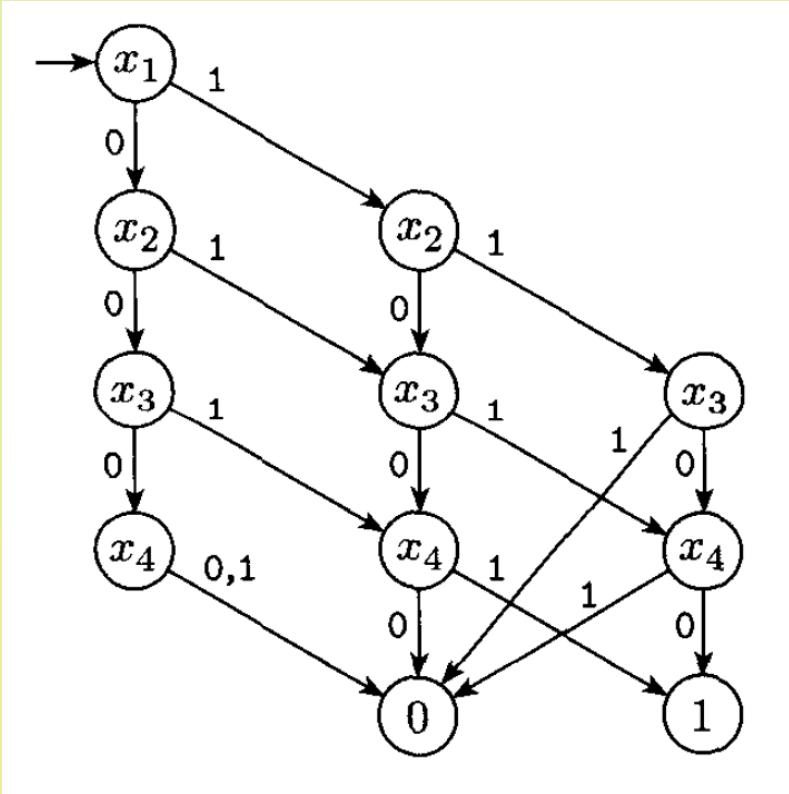
One of the nodes in a branching program is designated the **start node**.

- A branching program determines a Boolean function as follows:
  - Take any assignment to the variables appearing on the query nodes.
  - Begin at the start node, follow the path determined by taking the outgoing edge from each query node according to the value assigned to the indicated variable, until one of the output nodes is reached.
  - The value is the label of that output node.

Examples of Branching Programs

- The following are examples of branching programs:





- A branching program with polynomially many nodes can test membership in any language over $\{0, 1\}$ that is in L (log space).

Equivalence of Read-Once Branching Programs

- Two branching programs are **equivalent** if they determine equal functions.
- The problem of testing equivalence is coNP-complete.
- A **read-once branching program** is one that can query each variable at most once on every directed path from the start to an output node.
- Let

$$\mathrm{EQ}_{\mathrm{ROBP}} = \{\langle B_1, B_2\rangle : B_1 \text{ and } B_2 \text{ are equivalent read-once branching programs}\}.$$

Theorem

$\mathrm{EQ}_{\mathrm{ROBP}}$ is in BPP.

Idea for the Proof

- First we try assigning random values to the variables x_1 through x_m that appear in B_1 and B_2, and evaluate these branching programs on that setting.
  - We accept if B_1 and B_2 agree on the assignment;
  - We reject otherwise.
- This strategy does not work because two inequivalent read-once branching programs may disagree only on a single assignment out of the 2^n possible Boolean assignments to the variables. The probability that we would select that assignment is exponentially small. Hence we would accept with high probability even when B_1 and B_2 are not equivalent, and that is unsatisfactory.
- We modify this strategy by randomly selecting a non-Boolean assignment to the variables and evaluate B_1 and B_2 in a suitably defined manner. We can then show that, if B_1 and B_2 are not equivalent, the random evaluations will likely be unequal.

The Probabilistic Algorithm for $\mathrm{EQ}_{\mathrm{ROBP}}$

- Assign polynomials over $x_1, \ldots, x_m$ to the nodes and to the edges of a read-once branching program B as follows:
  - The constant function 1 is assigned to the start node.
  - If a node labeled x has been assigned polynomial p, assign the polynomial xp to its outgoing 1-edge, and assign the polynomial $(1 - x)p$ to its outgoing 0-edge.
  - If the edges incoming to some node have been assigned polynomials, assign the sum of those polynomials to that node.
  - Finally, the polynomial that has been assigned to the output node labeled 1 is also assigned to the branching program itself.
- The probabilistic polynomial time algorithm for $\mathrm{EQ}_{\mathrm{ROBP}}$: Denote by $\mathbb{F}$ a finite field with at least $3m$ elements.

  D: On input $\langle B_1, B_2 \rangle$, two read-once branching programs:
  1. Select elements a_1 through a_m at random from $\mathbb{F}$.
  2. Evaluate the assigned polynomials p_1 and p_2 at a_1 through a_m.
  3. If $p_1(a_1, \ldots, a_m) = p_2(a_1, \ldots, a_m)$, accept; otherwise, reject.

Polynomiality and Correctness of the Algorithm

- The algorithm runs in polynomial time: We can evaluate the polynomial without actually constructing the polynomial.
- For any Boolean assignment to B's variables, all polynomials assigned to its nodes evaluate to either 0 or 1. The polynomials that evaluate to 1 are those on the computation path for that assignment. Hence B and p agree when the variables take on Boolean values.
- Similarly, because B is read-once, we may write p as a sum of product terms $y_1 y_2 \cdots y_m$, where each y_i is x_i, $(1 - x_i)$, or 1, and where each product term corresponds to a path in B from the start node to the output node labeled 1. The case of $y_i = 1$ occurs when a path does not contain variable x_i. Take each such product term of p containing a y_i that is 1 and split it into the sum of two product terms, one where $y_i = x_i$ and the other where $y_i = (1 - x_i)$. Continue splitting product terms until each y_i is either x_i or $(1 - x_i)$. The end result is an equivalent polynomial q that contains a product term for each assignment on which B evaluates to 1.

Correctness of the Algorithm

- If B_1 and B_2 are equivalent, D always accepts: If the branching programs are equivalent, they evaluate to 1 on exactly the same assignments. Consequently, the polynomials q_1 and q_2 are equal because they contain identical product terms. Therefore p_1 and p_2 are equal on every assignment.
- If B_1 and B_2 are not equivalent, D rejects with a probability of at least $\frac{1}{2}$: We show this using two lemmas.

First Lemma

Lemma

For every $d > 0$, a degree-d polynomial p on a single variable x either has at most d roots, or is everywhere equal to 0.

- We use induction on d.

  Basis: For $d = 0$, a polynomial of degree 0 is constant. If that constant is not 0, the polynomial clearly has no roots.
- Induction Step: Assume true for $d - 1$ and prove true for d. If p is a nonzero polynomial of degree d with a root at a, the polynomial $x - a$ divides p evenly. Then $\frac{p}{x-a}$ is a nonzero polynomial of degree $d - 1$. Thus, it has at most $d - 1$ roots by virtue of the induction hypothesis.

Second Lemma I

Lemma

Let $\mathbb{F}$ be a finite field with f elements and let p be a nonzero polynomial on the variables x_1 through x_m, where each variable has degree at most d. If a_1 through a_m are selected randomly in $\mathbb{F}$, then

$$\Pr[p(a_1, \ldots, a_m) = 0] \leq \frac{md}{f}.$$

- We use induction on m.
  Basis: For $m = 1$, by the preceding lemma, p has at most d roots. So the probability that a_1 is one of them is at most $\frac{d}{f}$.
  Induction Step: Assume true for $m - 1$ and prove true for m. Let x_1 be one of p's variables. For each $i \leq d$, let p_i be the polynomial comprising the terms of p containing x_1^i, but where x_1^i has been factored out. Then
  $$p = p_0 + x_1 p_1 + x_1^2 p_2 + \cdots + x_1^d p_d.$$

Second Lemma II

- We wrote $p = p_0 + x_1 p_1 + x_1^2 p_2 + \cdots + x_1^d p_d$. If $p(a_1, \ldots, a_m) = 0$, one of two cases arises: Either all p_i evaluate to 0 or some p_i does not evaluate to 0 and a_1 is a root of the single variable polynomial obtained by evaluating p_0 through p_d on a_2 through a_m.
  - To bound the probability that the first case occurs, observe that one of the p_j must be nonzero because p is nonzero. Then the probability that all p_i evaluate to 0 is at most the probability that p_j evaluates to 0. By the induction hypothesis, that is at most $\frac{(m-1)d}{f}$ because p_j has at most $m-1$ variables.
  - To bound the probability that the second case occurs, observe that if some p_i does not evaluate to 0, then on the assignment of a_2 through a_m, p reduces to a nonzero polynomial in the single variable x_1. The basis already shows that a_1 is a root of such a polynomial with a probability of at most $\frac{d}{f}$.

  Therefore the probability that a_1 through a_m is a root of the polynomial is at most $\frac{(m-1)d}{f} + \frac{d}{f} = \frac{md}{f}$.

Subsection 3

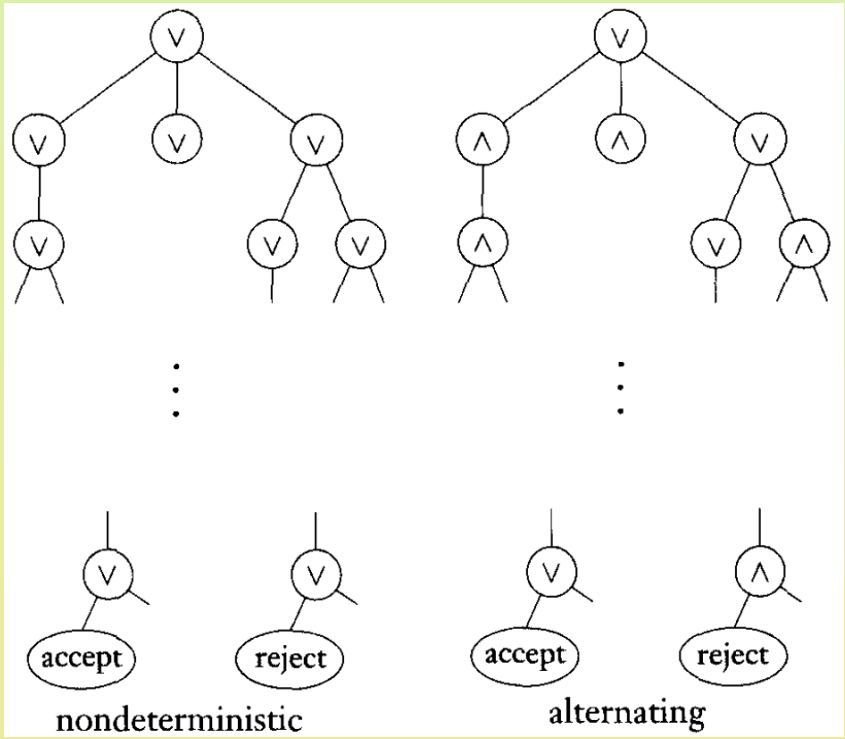
Alternation

Idea Behind Alternation

- Alternation is a generalization of nondeterminism, useful in
  - understanding relationships among complexity classes;
  - classifying specific problems according to their complexity;
  - exhibiting a surprising connection between the time and space complexity measures.
- An alternating algorithm may contain instructions to branch a process into multiple child processes, just as in a nondeterministic algorithm.
- The difference between the two lies in the mode of determining acceptance.
  - A nondeterministic computation accepts if any one of the initiated processes accepts.
  - When an alternating computation divides into multiple processes, two possibilities arise:
    - The algorithm can designate that the current process accepts if at least one of the children accept.
    - The algorithm can designate that the current process accepts if all of the children accept.

Nondeterminism Versus Alternation

- The difference between alternating and nondeterministic computation:



- In a nondeterministic computation, each node computes the OR operation of its children. It corresponds to the usual nondeterministic acceptance mode whereby a process is accepting if any of its children are accepting.

- In an alternating computation, the nodes may compute the AND or OR operations as determined by the algorithm. It corresponds to the alternating acceptance mode whereby a process is accepting if all or any of its children accept.

Alternating Turing Machines

Definition (Alternating Turing Machine)

An **alternating Turing machine** is a nondeterministic Turing machine with an additional feature.

- Its states, except for q_{accept} and q_{reject} are divided into **universal states** and **existential states**.
- When we run an alternating Turing machine on an input string, we label each node of its nondeterministic computation tree with $\wedge$ or $\vee$, depending on whether the corresponding configuration contains a universal or existential state.
- We determine **acceptance** by designating a node to be **accepting** if:
  - It is labeled with $\wedge$ and all of its children are accepting;
  - It it is labeled with $\vee$ and at least one of its children is accepting.

Alternating Time and Space

Definition (Alternating Time and Space)

$$\begin{aligned} \mathrm{ATIME}(t(n)) &= \{L : L \text{ is decided by an } \mathrm{O}(t(n)) \text{ time alternating Turing machine}\}; \\ \mathrm{ASPACE}(f(n)) &= \{L : L \text{ is decided by an } \mathrm{O}(f(n)) \text{ space alternating Turing machine}\}. \end{aligned}$$

- We define AP, APSPACE and AL to be the classes of languages that are decided by:
  - alternating polynomial time,
  - alternating polynomial space, and
  - alternating logarithmic space

  Turing machines, respectively.

Example: Tautologies

- A **tautology** is a Boolean formula that evaluates to 1 on every assignment to its variables:

$$\textsc{Taut} = \{\langle\phi\rangle : \phi \text{ is a tautology}\}.$$

- The following alternating algorithm shows that $\textsc{Taut}$ is in AP:
  On input $\langle\phi\rangle$:
  1. Universally select all assignments to the variables of ϕ.
  2. For a particular assignment, evaluate ϕ.
  3. If ϕ evaluates to 1, accept; otherwise, reject.
- Stage 1 nondeterministically selects every assignment to ϕ's variables with universal branching. This requires all branches to accept in order for the entire computation to accept.
  Stages 2 and 3 deterministically check whether the assignment that was selected on a particular computation branch satisfies the formula.
  Hence, the algorithm accepts if all assignments are satisfying.
- Observe that $\textsc{Taut}$ is a member of coNP.
- Similarly, any problem in coNP can be shown to be in AP.

A language in AP Not Known to Be in NP or in coNP

- Let ϕ and ψ be two Boolean formulas. ϕ and ψ are **equivalent** if they evaluate to the same value on all assignments to their variables.
- A **minimal formula** is one that has no shorter equivalent, where the **length** of a formula is the number of symbols that it contains:

$$\textsc{MinFormula} = \{\langle\phi\rangle : \phi \text{ is a minimal Boolean formula}\}.$$

- The following algorithm shows that $\textsc{MinFormula}$ is in AP:
  On input $\langle\phi\rangle$:
  1. Universally select all formulas ψ that are shorter than ϕ.
  2. Existentially select an assignment to the variables of ϕ.
  3. Evaluate both ϕ and ψ on this assignment.
  4. Accept if the formulas evaluate to different values. Reject if they evaluate to the same value.
- This algorithm starts with universal branching to select all shorter formulas and switches to existential branching to pick an assignment.
- The term alternation stems from the ability to alternate, or switch, between universal and existential branching.

Alternating and Deterministic Time and Space

- The following theorem demonstrates:
  - an equivalence between alternating time and deterministic space for polynomially related bounds;
  - another equivalence between alternating space and deterministic time when the time bound is exponentially more than the space bound.

Theorem

- For $f(n) \geq n$, we have $\mathrm{ATIME}(f(n)) \subseteq \mathrm{SPACE}(f(n)) \subseteq \mathrm{ATIME}(f^2(n))$.
- For $f(n) \geq \log n$, we have $\mathrm{ASPACE}(f(n)) = \mathrm{TIME}(2^{O(f(n))})$.

- Consequently,
  - $\mathrm{AL} = \mathrm{P}$;
  - $\mathrm{AP} = \mathrm{PSPACE}$;
  - $\mathrm{APSPACE} = \mathrm{EXPTIME}$.
- The proof of the theorem is given in four lemmas.

Alternating Time and Deterministic Space

Lemma

For $f(n) \geq n$, we have $\mathrm{ATIME}(f(n)) \subseteq \mathrm{SPACE}(f(n))$.

- We convert an alternating time $\mathrm{O}(f(n))$ machine M to a deterministic space $\mathrm{O}(f(n))$ machine S that simulates M:
  - On input w, the simulator S performs a depth-first search of M's computation tree to determine which nodes in the tree are accepting.
  - Then S accepts if it determines that the root of the tree, corresponding to M's starting configuration, is accepting.
- Machine S requires space for storing the recursion stack.
  - Each recursion level stores one configuration using $\mathrm{O}(f(n))$ space.
  - The recursion depth is M's time complexity, which is $\mathrm{O}(f(n))$.

  Hence S uses $\mathrm{O}\left(f^2(n)\right)$ space.
- To improve space complexity, instead of the entire configuration, we record only the nondeterministic choice that M made to reach that configuration from its parent. Then S can recover this configuration by replaying the computation from the start. Now the space used is constant at each level and the total is, thus, $\mathrm{O}(f(n))$.

Deterministic Space and Alternating Time I

Lemma

For $f(n) \geq n$, we have $\text{SPACE}(f(n)) \subseteq \text{ATIME}(f^2(n))$.

- We start with a deterministic space $\text{O}(f(n))$ machine M and construct an alternating machine S that uses time $\text{O}\left(f^2(n)\right)$ to simulate it.

  The approach is that followed in the proof of Savitch's Theorem, where we constructed a general procedure for the yieldability problem: Given configurations c_1 and c_2 of M and a number t, we must test whether M can get from c_1 to c_2 within t steps.
  - An alternating procedure for this problem first branches existentially to guess a configuration c_m midway between c_1 and c_2.
  - Then it branches universally into two processes, one that recursively tests whether c_1 can get to c_m within $\frac{t}{2}$ steps and the other whether c_m can get to c_2 within $\frac{t}{2}$ steps.

Deterministic Space and Alternating Time II

- Machine S uses this recursive alternating procedure to test whether the start configuration can reach an accepting configuration within $2^{df(n)}$ steps. Here, d is selected so that M has no more than $2^{df(n)}$ configurations within its space bound.
- The maximum time used on any branch of this alternating procedure is $O(f(n))$ to write a configuration at each level of the recursion, times the depth of the recursion, which is $\log(2^{df(n)}) = O(f(n))$. Hence this algorithm runs in alternating time $O\left(f^2(n)\right)$.

Alternating Space and Deterministic Time I

Lemma

For $f(n) \geq \log n$, we have $\mathrm{ASPACE}(f(n)) \subseteq \mathrm{TIME}(2^{O(f(n))})$.

- We construct a deterministic time $2^{O(f(n))}$ machine S to simulate an alternating space $O(f(n))$ machine M.
  - On input w, the simulator S constructs the following graph of the computation of M on w:
    - The nodes are the configurations of M on w that use at most $df(n)$ space, where d is the appropriate constant factor for M.
    - Edges go from a configuration to those configurations it can yield in a single move of M.
  - After constructing the graph, S repeatedly scans it and marks certain configurations as accepting.
    - Initially, only the accepting configurations of M are marked this way.
    - A configuration that performs universal branching is marked accepting if all of its children are so marked.
    - An existential configuration is marked if any of its children are marked.

Alternating Space and Deterministic Time II

- Continuing with the operation of S:
  - Machine S continues scanning and marking until no additional nodes are marked on a scan.
  - Finally, S accepts if the start configuration of M on w is marked.
- The number of configurations of M on w is $2^{O(f(n))}$ because $f(n) \geq \log n$. Thus, the size of the configuration graph is $2^{O(f(n))}$.
  - Constructing the graph may be done in $2^{O(f(n))}$ time.
  - Scanning the graph once takes roughly the same time.
  - The total number of scans is at most the number of nodes in the graph, because each scan except for the final one marks at least one additional node.

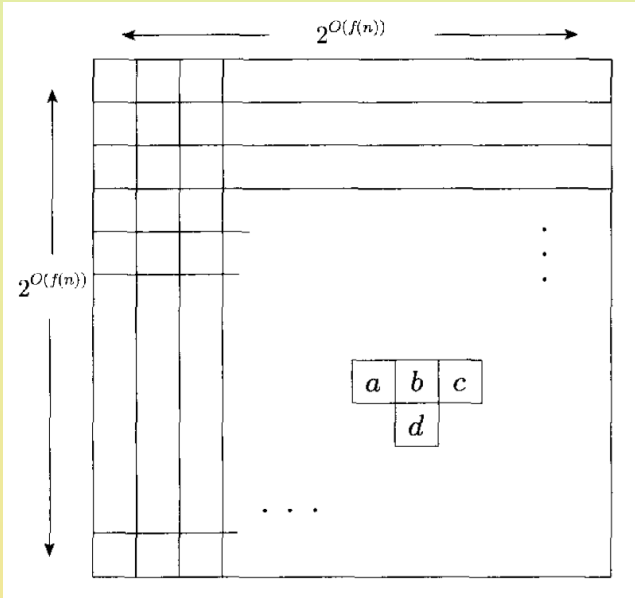
Hence, the total time used is $2^{O(f(n))}$.

Deterministic Time and Alternating Space I

Lemma

For $f(n) \geq \log n$, we have $\mathrm{ASPACE}(f(n)) \supseteq \mathrm{TIME}(2^{O(f(n))})$.

- We show how to simulate a deterministic time $2^{O(f(n))}$ machine M by an alternating Turing machine S that uses space $O(f(n))$. This simulation is tricky because the space available to S is so much less than the size of M's computation.
  In this case S has only enough space to store pointers into a tableau for M on w: We represent configurations in a way that a single symbol may represent both the state of the machine and the contents of the tape cell under the head. The contents of cell d are then determined by the contents of its parents a, b and c.



Deterministic Time and Alternating Space II

- Simulator S operates recursively to guess and then verify the contents of the individual cells of the tableau. To verify the contents of a cell d outside the first row, simulator S:
  - existentially guesses the contents of the parents,
  - checks whether their contents would yield d's contents according to M's transition function,
  - then universally branches to verify these guesses recursively.

  If d were in the first row, S verifies the answer directly because it knows M's starting configuration.
- We assume that M moves its head to the left-hand end of the tape on acceptance, so S can determine whether M accepts w by checking the contents of the lower leftmost cell of the tableau.
- Hence, S never needs to store more than a single pointer to a cell in the tableau, so it uses space $\log\left(2^{\mathrm{O}(f(n))}\right) = \mathrm{O}\left(f(n)\right)$.

Σ- and Π-Alternating Turing Machines

- Alternating machines provide a way to define a natural hierarchy of problems within the class PSPACE.

Definition (Σ- and Π-Alternating Turing Machines)

Let i be a natural number.

- A Σ_i-**alternating Turing machine** is an alternating Turing machine that contains at most i runs of universal or existential steps, starting with existential steps.
- A Π_i-**alternating Turing machine** is similar except that it starts with universal steps.

- Define $\Sigma_i\text{TIME}(f(n))$ to be the class of languages that a Σ_i-alternating Turing machine can decide in $O(f(n))$ time.
- Similarly, define $\Pi_i\text{TIME}(f(n))$ for Π_i-alternating Turing machines.
- Define the classes $\Sigma_i\text{SPACE}(f(n))$ and $\Pi_i\text{SPACE}(f(n))$ for space bounded alternating Turing machines.

The Polynomial Time Hierarchy

- We define the **polynomial time hierarchy** to be the collection of classes

$$\Sigma_i \mathrm{P} = \bigcup_k \Sigma_i \mathrm{TIME}(n^k) \quad \text{and} \quad \Pi_i \mathrm{P} = \bigcup_k \Pi_i \mathrm{TIME}(n^k).$$

- Define $\mathrm{PH} = \bigcup_i \Sigma_i \mathrm{P} = \bigcup_i \Pi_i \mathrm{P}$.
- Clearly, $\mathrm{NP} = \Sigma_1 \mathrm{P}$ and $\mathrm{coNP} = \Pi_1 \mathrm{P}$.
- Additionally, $\textsc{MinFormula} \in \Pi_2 \mathrm{P}$.

Subsection 4

Interactive Proof Systems

Prover and Verifier in NP

- We saw that probabilistic polynomial time algorithms provide a probabilistic analog to P.
- Interactive proof systems provide a way to define a probabilistic analog of the class NP.
- Recall that the languages in NP are those whose members all have short certificates of membership that can be easily checked. One may think of having:
  - a Prover that finds the proofs of membership;
  - a Verifier that checks them.

  The Verifier is required to be a polynomial time bounded machine, since, otherwise, it could figure out the answer itself.

  No computational bound on the Prover is imposed, because finding the proof may be time consuming.
- Example: In $\textsc{Sat}$, a Prover can convince a polynomial time Verifier that a formula ϕ is satisfiable by supplying the satisfying assignment.

Prover and Verifier in Interactive Proof Systems

- Can a Prover convince a computationally limited Verifier that a formula is not satisfiable?
- The complement of SAT is not known to be in NP so we cannot rely on the certificate idea.
- The answer, surprisingly, is yes, provided we give the Prover and Verifier two additional features:
  - First, they are permitted to engage in a two-way dialog.
  - Second, the Verifier may be a probabilistic polynomial time machine that reaches the correct answer with a high degree of, but not absolute, certainty.

Such a Prover and Verifier constitute an interactive proof system.

Graph Isomorphism and Graph Non-Isomorphism

- Graphs G and H are **isomorphic** if the nodes of G may be reordered so that it is identical to H:

$$\textsc{Iso} = \{\langle G, H\rangle : G \text{ and } H \text{ are isomorphic graphs}\}.$$

  $\textsc{Iso}$ is obviously in NP, but extensive research has so far failed to discover either a polynomial time algorithm or a proof that it is NP-complete.

- Consider the language

$$\textsc{NonIso} = \{\langle G, H\rangle : G \text{ and } H \text{ are not isomorphic graphs}\}.$$

  $\textsc{NonIso}$ is not known to be in NP because we do not know how to provide short certificates that graphs are not isomorphic.

Interactive Proof System for Graph Non-Isomorphism

- When two graphs G_1 and G_2 are not isomorphic, a Prover can convince a Verifier of this fact:
  - The Verifier randomly selects either G_1 or G_2 and, then, randomly reorders its nodes to obtain a graph H. The Verifier sends H to the Prover.
  - The Prover must respond by declaring whether G_1 or G_2 was the source of H.
- If G_1 and G_2 were indeed non-isomorphic, the Prover could always carry out the protocol because the Prover could identify whether H came from G_1 or G_2.

  If the graphs were isomorphic, H might have come from either G_1 or G_2, so even with unlimited computational power, the Prover would have no better than a 50-50 chance of getting the correct answer.
- Thus, if the Prover is able to answer correctly consistently (say in 100 repetitions of the protocol) the Verifier has convincing evidence that the graphs are actually nonisomorphic.

The Verifier

- The **Verifier** is a function V that computes its next transmission to the Prover from the message history sent so far. V has three inputs:
  1. **Input string**: The objective is to determine whether this string is a member of some language.
  2. **Random input**: For convenience, we provide the Verifier with a randomly chosen input string instead of the equivalent capability to make probabilistic moves during its computation.
  3. **Partial message history**: A script of the dialog up to the present point, represented by a string containing the messages sent thus far. $m_1 \# m_2 \# \cdots \# m_i$ represents the exchange of messages m_1 through m_i.

  The Verifier's **output** is either the next message m_{i+1} in the sequence or accept or reject.
- V has the functional form $V : \Sigma^* \times \Sigma^* \times \Sigma^* \to \Sigma^* \cup \{\text{accept}, \text{reject}\}$ and $V(w, r, m_1 \# \cdots \# m_i) = m_{i+1}$ means that the input string is w, the random input is r, the current message history is m_1 through m_i, and the Verifier's next message to the Prover is m_{i+1}.

The Prover

- The **Prover** is a party with unlimited computational ability. We define it to be a function P with two inputs:
  1. **Input string**.
  2. **Partial message history**.

  The Prover's output is the next message to the Verifier.
- Formally, P has the form $P : \Sigma^* \times \Sigma^* \to \Sigma^*$ and

$$P(w, m_1 \# \cdots \# m_i) = m_{i+1}$$

means that the Prover sends m_{i+1} to the Verifier after having exchanged messages m_1 through m_i so far.

The Interaction Between Prover and Verifier

- For particular strings w and r, we write

$$(V \leftrightarrow P)(w, r) = \text{accept}$$

if a message sequence m_1 through m_k exists for some k, such that

1. For $0 \leq i < k$, where i is even, $V(w, r, m_1 \# \cdots \# m_i) = m_{i+1}$;
2. For $0 < i < k$, where i is odd, $P(w, m_1 \# \cdots \# m_i) = m_{i+1}$;
3. The final message m_k in the message history is accept.

- To simplify the definition of the class IP we assume that the lengths of the Verifier's random input and each of the messages exchanged between the Verifier and the Prover are $p(n)$ for some polynomial p that depends only on the Verifier.
- Furthermore we assume that the total number of messages exchanged is at most $p(n)$.

The Class IP

- For any string w of length n, we define

$$\Pr[V \leftrightarrow P \text{ accepts } w] = \Pr[(V \leftrightarrow P)(w, r) = \text{accept}],$$

where r is a randomly selected string of length $p(n)$.

Definition (The Class IP)

Say that language A **is in** IP if there exist a polynomial time function V and arbitrary function P, such that, for every function $\widetilde{P}$ and string w:

1. $w \in A$ implies $\Pr[V \leftrightarrow P \text{ accepts } w] \geq \frac{2}{3}$;
2. $w \notin A$ implies $\Pr[V \leftrightarrow \widetilde{P} \text{ accepts } w] \leq \frac{1}{3}$.

- We may amplify the success probability of an interactive proof system by repetition to make the error probability exponentially small.
- IP contains both of the classes NP and BPP. Moreover, it contains the language NonIso, which is not known to be in either NP or BPP.

Polynomial Interactive Proofs and Polynomial Space

- A remarkable theorem in complexity is the equality of IP and PSPACE: For any language in PSPACE, a Prover can convince a probabilistic polynomial time Verifier about the membership of a string in the language, even though a conventional proof of membership might be exponentially long.

Theorem

IP = PSPACE.

- The proof consists of establishing inclusions in each direction.
  - The left-to-right containment involves a standard simulation of an interactive proof system by a polynomial space machine.
  - The right-to-left containment involves **arithmetization**, in which a polynomial $p(x_1, \ldots, x_m)$ is associated to a CNF-formula ϕ, with variables x_1 through x_m, that mimics ϕ by simulating the Boolean $\wedge$, $\vee$ and $\neg$ operations via the arithmetic operations $+$, $\times$.

  We omit this rather long and tedious proof.

Subsection 5

Parallel Computation

Parallel Computers

- A **parallel computer** is one that can perform multiple operations simultaneously.
- Parallel computers may solve certain problems much faster than **sequential computers**, able to do a single operation at a time.
- In practice, the distinction between the two is slightly blurred because most real computers (including "sequential" ones) are designed to use some parallelism as they execute individual instructions.
- We focus on *massive parallelism* whereby a huge number of processing elements are actively participating in a single computation.
- We introduce the theory of parallel computation:
  - We describe one model of a parallel computer;
  - Use it to give examples of certain problems that lend themselves well to parallelization;
  - Explore the possibility that parallelism may not be suitable for certain other problems.

Boolean Circuits as Models of Parallel Computers

- One model in theoretical work on parallel algorithms is called the **Parallel Random Access Machine** or **PRAM**:

  In the PRAM model, idealized processors with a simple instruction set patterned on actual computers interact via a shared memory.
- We use an alternative model of parallel computer, Boolean circuits:
  - The model is simple to describe, which makes proofs easier.
  - Circuits also bear an obvious resemblance to actual hardware designs and in that sense the model is realistic.
  - Circuits are awkward to "program" because the individual processors are so weak.
  - Furthermore, we disallow cycles in our definition of Boolean circuits, in contrast to circuits that we can actually build.
- We take each gate to be an individual processor, so we define the **processor complexity** of a Boolean circuit to be its size.
- Each processor computes its function in a single time step, so the **parallel time complexity** is the depth, or the longest distance from an input variable to the output gate.

Uniform Circuit Families

- Any particular circuit has a fixed number of input variables, so we use circuit families for recognizing languages.
- To correspond to parallel computation models, such as PRAMs, where a single machine is capable of handling all input lengths, Boolean circuit families must be refined:

Definition (Uniform Family of Circuits)

A family of circuits $(C_1, C_2, \ldots)$ is **uniform** if some log space transducer T outputs $\langle C_n \rangle$ when T's input is 1^n.

- The size and depth complexity of languages were defined in terms of families of circuits of minimal size and depth.
- A language has **simultaneous size-depth circuit complexity** at most $(f(n), g(n))$ if a uniform circuit family exists for that language with size complexity $f(n)$ and depth complexity $g(n)$.

Strings with Odd Number of 1s

- Let A be the language over $\{0,1\}$ consisting of all strings with an odd number of 1s.
- We can test membership in A by computing the parity function. We can implement the two input parity gate $x \oplus y$ with the standard AND, OR, and NOT operations as

$$x \oplus y = (x \wedge \neg y) \vee (\neg x \wedge y).$$

  Let the inputs to the circuit be $x_1, \ldots, x_n$. One way to get a circuit for the parity function is to construct gates g_i whereby $g_1 = x_1$ and $g_i = x_i \oplus g_{i-1}$, for $i \leq n$. The construction uses $\mathrm{O}(n)$ size and depth.
- We have described another circuit for the parity function with $\mathrm{O}(n)$ size and $\mathrm{O}(\log n)$ depth by constructing a binary tree of $\oplus$ gates. This is a significant improvement because it uses exponentially less parallel time than does the preceding construction. Thus, the size-depth complexity of A is $(\mathrm{O}(n), \mathrm{O}(\log n))$.

Boolean Matrix Multiplication

- Consider the Boolean matrix multiplication function:
  - The input has $2m^2 = n$ variables representing two $m \times m$ matrices $A = \{a_{ik}\}$ and $B = \{b_{ik}\}$.
  - The output is m^2 values representing the $m \times m$ matrix $C = \{c_{ik}\}$, where

$$c_{ik} = \bigvee_j (a_{ij} \wedge b_{jk}).$$

The circuit for this function contains:

- Gates g_{ijk} that compute $a_{ij} \wedge b_{jk}$ for each i, j and k;
- For each i and k, a binary tree of $\vee$ gates to compute $\bigvee_j g_{ijk}$. Each such tree contains $m - 1$ OR gates and has $\log m$ depth.

Consequently, these circuits for Boolean matrix multiplication have size $\mathrm{O}\left(m^3\right) = \mathrm{O}\left(n^{3/2}\right)$ and depth $\mathrm{O}\left(\log n\right)$.

The Transitive Closure of a Matrix

- If $A = \{a_{ij}\}$ is an $m \times m$ matrix, we let the **transitive closure** of A be the matrix $A \vee A^2 \vee \cdots \vee A^m$, where A^i is the matrix product of A with itself i times and $\vee$ is the bitwise OR of the matrix elements.
- The transitive closure is related to PATH and, hence, to the class NL:
  - If A is the adjacency matrix of a directed graph G, A^i is the adjacency matrix of the graph, with the same nodes, in which an edge indicates the presence of a path of length i in G.
  - The transitive closure of A is the adjacency matrix of the graph in which an edge indicates the presence of a path of any length in G.
- We can represent the computation of A^i with a binary tree of size i and depth $\log i$ wherein a node computes the product of the two matrices below it:
  - Each node needs a circuit of $O\left(n^{3/2}\right)$ size and logarithmic depth.
  - Hence the circuit computing A^m has size $O\left(n^2\right)$ and depth $O\left(\log^2 n\right)$.

  We make circuits for each A^i:
  - This adds a factor of m to the size and an extra $O(\log n)$ depth.

  Hence, the size-depth complexity is $\left(O\left(n^{5/2}\right), O\left(\log^3 n\right)\right)$.

NC-Computable Languages

- Many problems have size-depth complexity $\left(O\left(n^k\right), O\left(\log^k n\right)\right)$ for some constant k. Such problems may be considered to be highly parallelizable with a moderate number of processors:

Definition (NC-Computable Languages)

For $i \geq 1$, let NC^i be the class of languages that can be decided by a uniform family of circuits with polynomial size and $O\left(\log^i n\right)$ depth. Let NC be the class of languages that are in NC^i, for some i. Functions that are computed by such circuit families are called NC^i-**computable** or NC-**computable**.

- We will see that:
  - Problems that are solvable in logarithmic depth are also solvable in logarithmic space.
  - Conversely, problems that are solvable in logarithmic space, even nondeterministically, are solvable in logarithmic squared depth.

Logarithmic Depth and Logarithmic Space

Theorem

$NC^1 \subseteq L$.

- We sketch a log space algorithm to decide a language A in NC^1.
  - On input w of length n, the algorithm can construct the description as needed of the n-th circuit in the uniform circuit family for A.
  - Then the algorithm can evaluate the circuit by using a depth-first search from the output gate. The only memory that is necessary to keep track of the progress of the search is to record the path to the current gate that is being explored and to record any partial results that have been obtained along that path.
- The circuit has logarithmic depth. Hence, only logarithmic space is required by the simulation.

Nondeterministic Log Space and Log Squared Depth

Theorem

$\mathrm{NL} \subseteq \mathrm{NC}^2$.

- We compute the transitive closure of the graph of configurations of an NL-machine and output the position corresponding to the presence of a path from the start configuration to the accept configuration.
- Let A be a language that is accepted by an NL machine M, where A has been encoded into the $\{0,1\}$ alphabet. We construct a uniform circuit family $(C_0, C_1, \ldots)$ for A. To get C_i we construct a graph G that is similar to the computation graph for M on an input w of length n. We do not know w - only its length n. The inputs to the circuit are variables $w_1, \ldots, w_n$ corresponding to input positions.
  - A configuration of M on w describes the state, the contents of the work tape, and the positions of both the input and the work tape heads, but does not include w itself. So the collection of configurations does not depend on w - only on w's length n. These polynomially many configurations form the nodes of G.

Proof (Cont'd)

- We continue with the edges:
  - The edges of G are labeled with the input variables w_i.
    - If c_1 and c_2 are two nodes of G and c_1 indicates input head position i, we put edge (c_1, c_2) in G with label w_i (or $\overline{w_i}$) if c_1 can yield c_2 in a single step when the input head is reading a 1 (or 0), according to M's transition function.
    - If c_1 can yield c_2 in a single step, whatever the input head is reading, we put that edge in G unlabeled.

If we set the edges of G according to a string w of length n, a path exists from the start configuration to the accepting configuration if and only if M accepts w. Hence, a circuit that computes the transitive closure of G and outputs the position indicating the presence of such a path accepts exactly those strings in A of length n. That circuit has polynomial size and $O\left(\log^2 n\right)$ depth. Finally, a log space transducer is capable of constructing G and, thus, also C_n, on input 1^n.

NC and Polynomial Time

- All NC-problems are solvable in polynomial time:

Theorem

$NC \subseteq P$.

- A polynomial time algorithm can run the log space transducer to generate circuit C_n and simulate it on an input of length n.
- Are all problems in P also in NC, i.e., are the two classes equal?
  - Equality would be surprising because it would imply that all polynomial time solvable problems are highly parallelizable.
  - We introduce the phenomenon of P-completeness as a means to provide evidence that some problems in P are inherently sequential.

Definition (P-Complete Language)

A language B is P-complete if:

1. $B \in P$;
2. Every A in P is log space reducible to B.

Polynomial Time Completeness of Circuit Value

- Recall the proof that, if $A \leq_{\mathrm{L}} B$ and $B \in \mathrm{L}$, then $A \in \mathrm{L}$.
- Since NL and NC machines can compute log space reductions:

Theorem

If $A \leq_{\mathrm{L}} B$ and B is in NC, then A is in NC.

- For a circuit C and input x, $C(x)$ denotes the value of C on x:
  $\textsc{CircuitValue} = \{\langle C, x\rangle : C \text{ is a Boolean circuit and } C(x) = 1\}$.

Theorem

$\textsc{CircuitValue}$ is P-complete.

- The tableau construction, used to encode a DTM into a family of circuits, can reduce any language A in P to $\textsc{CircuitValue}$. On input w, the reduction produces a circuit that simulates the polynomial time Turing machine for A. The input to the circuit is w.
- The reduction can be carried out in log space because the circuit it produces has a simple and repetitive structure.