Introduction to Computational Complexity

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Subsection 1

Measuring Complexity

A Turing Machine for $\{0^k1^k : k \geq 0\}$

- Consider the language $A = \{0^k1^k : k \geq 0\}$.
- A is a decidable language.
- How much time does a single-tape Turing machine need to decide A?
- The following is a single-tape TM M_1 for A at a low level description, including the actual head motion on the tape, so that we can count the number of steps that M_1 uses when it runs:

  M_1: On input string w:

  1. Scan across the tape and reject if a 0 is found to the right of a 1.
  2. Repeat if both 0s and 1s remain on the tape:
     3. Scan across the tape, crossing off a single 0 and a single 1.
  4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, reject.
     Otherwise, if neither 0s nor 1s remain on the tape, accept.
- We analyze this algorithm for TM M_1 that decides A to determine how much time it uses.

Parameters and Worst- vs. Average-Case Analysis

- The number of steps that an algorithm uses on a particular input may depend on several parameters.

  Example: If the input is a graph, the number of steps may depend on the number of nodes, the number of edges, and the maximum degree of the graph, or some combination of these and/or other factors.
- For simplicity we compute the running time of an algorithm purely as a function of the length of the string representing the input and do not consider any other parameters.
- In **worst-case analysis**, we consider the longest running time of all inputs of a particular length.

  In **average-case analysis**, we consider the average of all the running times of inputs of a particular length.

Running Time or Time Complexity

Definition (Running Time or Time Complexity)

Let M be a deterministic Turing machine that halts on all inputs. The **running time** or **time complexity** of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.

If $f(n)$ is the running time of M, we say that M **runs in time** $f(n)$ and that M is an $f(n)$-**time Turing machine**.

The symbol n usually represents the length of the input.

Asymptotic Analysis

- Because the exact running time of an algorithm often is a complex expression, we usually just estimate it.
- In one convenient form of estimation, called **asymptotic analysis**, we seek to understand the running time of the algorithm when it is run on large inputs.
- This is done by considering only the highest order term of the expression for the running time of the algorithm, disregarding both the coefficient of that term and any lower order terms, because the highest order term dominates the other terms on large inputs.
- Example: The function $f(n) = 6n^3 + 2n^2 + 20n + 45$ has four terms, and the highest order term is $6n^3$. Disregarding the coefficient 6, we say that f is **asymptotically at most** n^3. The **asymptotic notation** or **big-O notation** for describing this relationship is $f(n) = \mathrm{O}\left(n^3\right)$.

Asymptotic Upper Bounds

Definition (Asymptotic Upper Bound)

Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = \mathrm{O}(g(n))$ if there exist positive integers c and n_0, such that, for every integer $n \geq n_0$,

$$f(n) \leq cg(n).$$

When $f(n) = \mathrm{O}(g(n))$ we say that $g(n)$ is an **upper bound** for $f(n)$, or more precisely, that $g(n)$ is an **asymptotic upper bound** for $f(n)$, to emphasize that we are suppressing constant factors.

- Example: Let $f_1(n)$ be the function $5n^3 + 2n^2 + 22n + 6$. Then, selecting the highest order term $5n^3$ and disregarding its coefficient 5 gives $f_1(n) = \mathrm{O}(n^3)$. To see that this result satisfies the formal definition, let $c = 6$ and $n_0 = 10$. Then, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$, for every $n \geq 10$. In addition, $f_1(n) = \mathrm{O}(n^4)$ because n^4 is larger than n^3. However, $f_1(n)$ is not $\mathrm{O}(n^2)$.

Big-O and Logarithmic Functions

- The big-O interacts with logarithms in a particular way:

  When we use logarithms we must specify the base, as in $x = \log_2 n$. This equality is equivalent to the equality $2^x = n$.
  Changing the value of the base b changes the value of $\log_b n$ by a constant factor, owing to the identity $\log_b n = \dfrac{\log_2 n}{\log_2 b}$.

  Thus, when we write $f(n) = \mathrm{O}(\log n)$, specifying the base is no longer necessary because we are suppressing constant factors anyway.
- Example: Let $f_2(n)$ be the function $3n \log_2 n + 5n \log_2 \log_2 n + 2$. In this case we have $f_2(n) = \mathrm{O}(n \log n)$ because $\log n$ dominates $\log \log n$.

Big-O in Arithmetic Expressions

- Big-O notation appears in arithmetic expressions, such as $f(n) = O(n^2) + O(n)$.
  In that case each occurrence of the O symbol represents a different suppressed constant. $O(n^2)$ dominates $O(n)$, so $f(n) = O(n^2)$.
- When the O symbol occurs in an exponent, as in $f(n) = 2^{O(n)}$, the same idea applies. This expression represents an upper bound of 2^{cn} for some constant c.
- The expression $f(n) = 2^{O(\log n)}$ occurs in some analyses. Since $n = 2^{\log_2 n}$, we have $n^c = 2^{c \log_2 n}$, so $2^{O(\log n)}$ is an upper bound of n^c, for some c.
  The expression $n^{O(1)}$ represents the same bound, since $O(1)$ represents a value that is never more than a fixed constant.
- Frequently we derive bounds of the form n^c for c greater than 0. Such bounds are called **polynomial bounds**. Bounds of the form 2^{n^δ} are called **exponential bounds** when δ is a real number greater than 0.

Small-o Notation

- Big-O notation has a companion called **small-o notation**.
  - Big-O says that one function is asymptotically no more than another.
  - Small-o says that one function is asymptotically less than another.

  The difference is analogous to the difference between $\leq$ and $<$.

Definition (Small-o Notation)

Let f and g be functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. I.e., $f(n) = o(g(n))$ means that, for any real number $c > 0$, a number n_0 exists, such that $f(n) < cg(n)$, for all $n \geq n_0$.

- Examples: The following are easy to check:
  1. $\sqrt{n} = o(n)$.
  2. $n = o(n \log\log n)$.
  3. $n \log\log n = o(n \log n)$.
  4. $n \log n = o(n^2)$.
  5. $n^2 = o(n^3)$.
  6. But $f(n)$ is never $o(f(n))$.

Analysis of the Turing Machine for $\{0^k1^k : k \geq 0\}$

- We analyze the TM algorithm for the language $A = \{0^k1^k : k \geq 0\}$.
- We consider each of its four stages separately:
  - In Stage 1, the machine scans across the tape to verify that the input is of the form 0^*1^*. Performing this scan uses n steps. Repositioning the head at the left-hand end of the tape uses another n steps. So the total used in this stage is $2n$ steps. In big-O notation we say that this stage uses $O(n)$ steps.
  - In Stages 2 and 3, the machine repeatedly scans the tape and crosses off a 0 and 1 on each scan. Each scan uses $O(n)$ steps. Because each scan crosses off two symbols, at most $\frac{n}{2}$ scans can occur. So the total time taken by Stages 2 and 3 is $\frac{n}{2}O(n) = O(n^2)$ steps.
  - In Stage 4, the machine makes a single scan to decide whether to accept or reject. The time taken in this stage is at most $O(n)$.

Thus, the total time of M_1 on an input of length n is $O(n) + O(n^2) + O(n)$, or $O(n^2)$.

Time Complexity Classes

- We set up some notation for classifying languages according to their time requirements:

Definition (Time Complexity Classes)

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the **time complexity class** $\mathrm{TIME}(t(n))$ to be the collection of all languages that are decidable by an $\mathrm{O}(t(n))$ time Turing machine.

- Example: We looked at the language $A = \{0^k 1^k : k \geq 0\}$. We showed that the machine M_1 decides A in time $\mathrm{O}(n^2)$. The class $\mathrm{TIME}(n^2)$ contains all languages that can be decided in $\mathrm{O}(n^2)$ time. We conclude that $A \in \mathrm{TIME}(n^2)$.

Machine M_2 Deciding A

- Is there a machine that decides $A = \{0^k 1^k : k \geq 0\}$ asymptotically more quickly than $O(n^2)$?
- We can improve the running time by crossing off two 0s and two 1s on every scan (instead of just one) because doing so cuts the number of scans by half. But that improves the running time only by a factor of 2 and does not affect the asymptotic running time.
- The following machine M_2 uses a different method to decide A asymptotically faster in time $n \log n$.

  M_2: On input string w:

  1. Scan across the tape and reject if a 0 is found to the right of a 1.
  2. Repeat as long as some 0s and some 1s remain on the tape:
     3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, reject.
     4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
  5. If no 0s and no 1s remain on the tape, accept. Otherwise, reject.

Correctness of M_2

- We first verify that M_2 actually decides A.
  - On every scan performed in Stage 4, the total number of 0s remaining is cut in half and any remainder is discarded. This stage has the same effect on the number of 1s.
  - Starting with, say 13 0s and 13 1s, the first execution of stage 3 finds an odd number of 0s and an odd number of 1s. On subsequent executions an even number (6) occurs, then an odd number (3), and an odd number (1). For the sequence of parities found (odd, even, odd, odd) if we replace the evens with 0s and the odds with 1s and then reverse the sequence, we obtain 1101, the binary representation of 13, or the number of 0s and 1s at the beginning. The sequence of parities always gives the reverse of the binary representation.
  - When Stage 3 checks to determine that the total number of 0s and 1s remaining is even, it actually is checking on the agreement of the parity of the 0s with the parity of the 1s. If all parities agree, the binary representations of the numbers of 0s and of 1s agree, and so the two numbers are equal.

Time Analysis of M_2

- For the running time of M_2, note that every stage takes $O(n)$ time.
  - Stages 1 and 5 are executed once, taking a total of $O(n)$ time.
  - Stage 4 crosses off at least half the 0s and 1s each time it is executed, so at most $1 + \log_2 n$ iterations of the repeat loop occur before all get crossed off. Thus the total time of stages 2, 3, and 4 is $(1 + \log_2 n)\, O(n)$, or $O(n \log n)$.

  The running time of M_2 is $O(n) + O(n \log n) = O(n \log n)$.
- We showed that $A \in \text{TIME}(n^2)$, but now we have the better bound $A \in \text{TIME}(n \log n)$.
- This result cannot be further improved on single tape Turing machines.
- In fact, it can be shown that any language that can be decided in $o(n \log n)$ time on a single-tape Turing machine is regular.

A Two-Tape Machine Deciding A in Linear Time

- The language A can be decided in $O(n)$ time (also called **linear time**) if the Turing machine has a second tape.
- The following two-tape TM M_3 decides A in linear time: It copies the 0s to its second tape and then matches them against the 1s.

  M_3: On input string w:

  1. Scan across the tape and reject if a 0 is found to the right of a 1.
  2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
  3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, reject.
  4. If all the 0s have now been crossed off, accept. If any 0s remain, reject.
- Each of the four stages uses $O(n)$ steps, so the total running time is $O(n)$ and thus is linear. This running time is the best possible because n steps are necessary just to read the input.

Complexity and Computability

- We have produced a single-tape TM M_1 that decides A in $O(n^2)$ time and a faster single tape TM M_2 that decides A in $O(n \log n)$ time.
- It can be shown that no single-tape TM can decide A more quickly.
- We also exhibited a two-tape TM M_3 that decides A in $O(n)$ time.
- The complexity of A depends on the model of computation selected.
- Key difference between complexity theory and computability theory:
  - In computability theory, the Church-Turing thesis implies that all reasonable models of computation are equivalent;
  - In complexity theory, the choice of model affects the time complexity of languages.
- Since in complexity theory, we classify computational problems according to their time complexity and this depends on the model of computation, how do we decide which model to use?
  - It turns out that time requirements do not differ greatly for typical deterministic models.
  - If our classification system is not very sensitive to small differences in complexity, the choice of deterministic model is not crucial.

Multi-Tape TMs and Single-Tape TMs

- We examine how the choice of computational model can affect the time complexity of languages. We consider three models:
  - The single-tape Turing machine;
  - The multi-tape Turing machine;
  - The nondeterministic Turing machine.

Theorem

Let $t(n)$ be a function, where $t(n) \geq n$. Every $t(n)$ time multi-tape Turing machine has an equivalent $O\left(t^2(n)\right)$ time single-tape Turing machine.

- We have seen how to convert any multi-tape TM into a single-tape TM that simulates it.
  We need to analyze that simulation to determine how much additional time it requires.
  - We show that simulating each step of the multi-tape machine uses at most $O\left(t(n)\right)$ steps on the single-tape machine.
  - Hence, the total time used is $O\left(t^2(n)\right)$ steps.

Proof of the Multi-Tape to Single-Tape Theorem I

- Let M be a k-tape TM that runs in $t(n)$ time. We construct a single-tape TM S that runs in $O\left(t^2(n)\right)$ time.
  - S uses its single tape to represent the contents on all k of M's tapes. The tapes are stored consecutively, with the positions of M's heads marked on the appropriate squares.
  - Initially, S puts its tape into the format that represents all the tapes of M and then simulates M's steps.
  - To simulate one step, S scans all the information stored on its tape to determine the symbols under M's tape heads. Then S makes another pass over its tape to update the tape contents and head positions. If one of M's heads moves rightward onto the previously unread portion of its tape, S must increase the amount of space allocated to this tape. It does so by shifting a portion of its own tape one cell to the right.

Proof of the Multi-Tape to Single-Tape Theorem II

- For each step of M, machine S makes two passes over the active portion of its tape.
  - The first obtains the information necessary to determine the next move;
  - The second carries it out.
- The length of the active portion of S's tape determines how long S takes to scan it. Thus, we must obtain an upper bound on this length.
  - We take the sum of the lengths of the active portions of M's k tapes. Each of these active portions has length at most $t(n)$ because M uses $t(n)$ tape cells in $t(n)$ steps if the head moves rightward at every step and even fewer if a head ever moves leftward. Thus, a scan of the active portion of S's tape uses $O(t(n))$ steps.

Proof of the Multi-Tape to Single-Tape Theorem III

- To simulate each of M's steps, S performs two scans and possibly up to k rightward shifts, each of which uses $\mathrm{O}(t(n))$ time. So the total time for S to simulate one of M's steps is $\mathrm{O}(t(n))$.
- Now we bound the total time used by the simulation.
  - The initial stage, where S puts its tape into the proper format, uses $\mathrm{O}(n)$ steps.
  - Afterward, S simulates each of the $t(n)$ steps of M, using $\mathrm{O}(t(n))$ steps, so this part of the simulation uses $t(n) \times \mathrm{O}(t(n)) = \mathrm{O}\left(t^2(n)\right)$ steps.

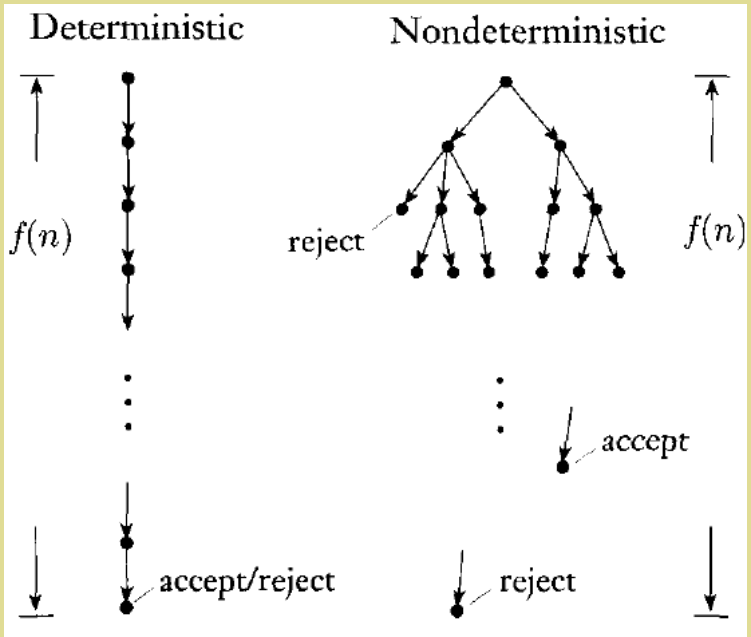
  Therefore the entire simulation of M uses $\mathrm{O}(n) + \mathrm{O}\left(t^2(n)\right)$ steps.
- We have assumed that $t(n) \geq n$ (a reasonable assumption because M could not even read the entire input in less time), whence the running time of S is $\mathrm{O}\left(t^2(n)\right)$.

Running Time of a Non-Deterministic Turing Machine

- We now show that any language that is decidable on a single-tape non-deterministic Turing machine is also decidable on a deterministic single-tape Turing machine that requires significantly more time.
- Recall that a nondeterministic Turing machine is a **decider** if all its computation branches halt on all inputs.

Definition (Running Time of a NDTM)

Let N be a nondeterministic Turing machine that is a decider. The **running time** of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.



Non-Deterministic to Deterministic Turing Machines

Theorem

Let $t(n)$ be a function, where $t(n) \geq n$. Every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

- Let N be a nondeterministic TM running in $t(n)$ time. The deterministic TM D simulates N by searching breadth-first N's nondeterministic computation tree.
- On an input of length n, every branch of N's nondeterministic computation tree has a length of at most $t(n)$. Every node in the tree can have at most b children, where b is the maximum number of legal choices given by N's transition function. Thus, the total number of leaves in the tree is at most $b^{t(n)}$. The simulation proceeds by exploring this tree breadth first, i.e., it visits all nodes at depth d before going on to any of the nodes at depth $d + 1$.

Non-Deterministic to Deterministic TMs (Cont'd)

- The corresponding algorithm starts at the root and travels down to a node whenever it visits that node.

  The total number of nodes in the tree is less than twice the maximum number of leaves. So we bound it by $O\left(b^{t(n)}\right)$. The time for starting from the root and traveling down to a node is $O\left(t(n)\right)$. Therefore the running time of D is $O\left(t(n)b^{t(n)}\right) = 2^{O(t(n))}$.

Subsection 2

The Class P

Polynomial Time vs. Exponential Time

- For complexity purposes, polynomial differences in running time are considered small; exponential differences are considered large.
- There are several reasons for making this distinction between polynomials and exponentials rather than between other classes:
  - The growth rates of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n are dramatically different.
  - Polynomial time algorithms are fast enough for many purposes, but exponential time algorithms rarely are useful.
  - Exponential time algorithms typically arise when we solve problems by exhaustive search of a solution space, called **brute-force search**. Sometimes, brute-force search may be avoided through a deeper understanding of a problem, which may reveal a polynomial time algorithm of greater utility.
  - All reasonable deterministic computational models are polynomially equivalent, i.e., any one of them can simulate another with only a polynomial increase in running time.
    E.g., the deterministic single-tape and multi-tape Turing machine models are polynomially equivalent.

The Class P

- We focus on aspects of time complexity theory that are unaffected by polynomial differences in running time.
- By considering polynomial differences insignificant and ignoring them, we develop the theory in a way that does not depend on the selection of a particular model of computation.

Definition (The Class P)

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine, i.e., $\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k)$.

- The class P is important because:
  1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine;
  2. P roughly corresponds to the class of problems that are realistically solvable on a computer.
- Item 1 indicates that P is a mathematically robust class.
- Item 2 indicates that P is relevant in practice.

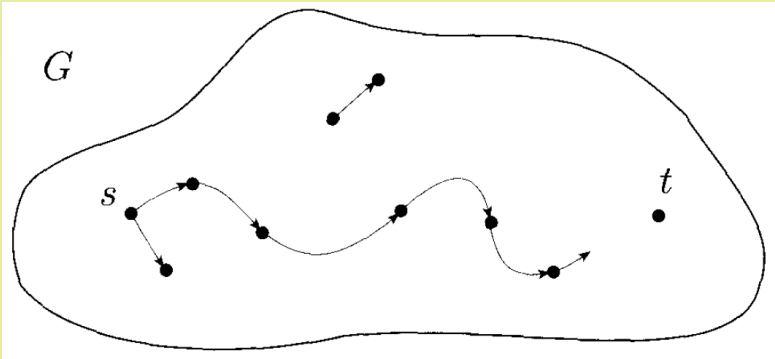
High Level Description of Algorithms

- Polynomial time algorithms are provided via high-level descriptions, without reference to features of a particular computational model.
- These involve describing algorithms with numbered stages. Analysis to show polynomial time involves:
  - Giving a polynomial upper bound on the number of stages that the algorithm uses when it runs on an input of length n.
  - Making sure that each individual stage can be implemented in polynomial time on a reasonable deterministic model.

  The algorithm, then, runs in polynomial time because it runs for a polynomial number of stages, each of which takes polynomial time, and the composition of polynomials is a polynomial.
- A sensitive point is the encoding method: The notation $\langle \bullet \rangle$ indicates a reasonable encoding of one or more objects into a string: A reasonable method should allow for polynomial time encoding and decoding into natural representations or other reasonable encodings.
- Unary notation for encoding numbers is not reasonable!

Paths in Directed Graphs

- For graphs, one reasonable encoding consists of a list of its nodes and edges.
- Another is the adjacency matrix, where the (i, j)-th entry is 1 if there is an edge from node i to node j and 0 if there is no edge.
- In analysis of algorithms on graphs, the running time may be computed in terms of the number of nodes instead of the size of the graph representation: In a reasonable representation, the size of the representation is a polynomial in the number of nodes.
- A directed graph G contains nodes s and t. The PATH problem is to determine whether a directed path exists from s to t.



$$\text{PATH} = \{(G, s, t) : G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$$

PATH is in P: Idea of Proof

Theorem

PATH $\in$ P.

- We prove this theorem by presenting a polynomial time algorithm that decides PATH.
  - A brute-force algorithm is not fast enough. Examining all potential paths in G and determining whether any is a directed path from s to t would take exponential time:
    A potential path is a sequence of nodes in G having a length of at most m, where m is the number of nodes in G. The number of such potential paths is roughly m^m, which is exponential in the number of nodes in G.
  - To get a polynomial time algorithm for PATH, we must do something that avoids brute force. One way is to use a graph-searching method such as breadth-first search. We successively mark all nodes in G that are reachable from s by directed paths of length 1, then 2, then 3, through m. Bounding the running time by a polynomial is easy.

PATH is in P: Formal Proof

Theorem

PATH $\in$ P.

- A polynomial time algorithm M for PATH operates as follows.
  M: On input $\langle G, s, t \rangle$, G is a directed graph with nodes s and t:
  1. Place a mark on node s.
  2. Repeat the following until no additional nodes are marked:
     3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
  4. If t is marked, accept. Otherwise, reject.
- Stages 1 and 4 are executed only once. Stage 3 runs at most m times, since each time except the last it marks an additional node in G. The total number is at most $1 + 1 + m$, a polynomial in the size of G.
- Stages 1 and 4 of M are easily implemented in polynomial time on any reasonable deterministic model. Stage 3 involves a scan of the input and a test of whether certain nodes are marked, which is, also, easily implemented in polynomial time.

RelPrime is in P: Idea of Proof

- Two numbers x, y are **relatively prime** if $\gcd(x, y) = 1$.
- Example: 10 and 21 are relatively prime; 10 and 22 are not relatively prime because both are divisible by 2.
- Let RelPrime be the problem of testing whether two numbers are relatively prime:

$$\text{RelPrime} = \{\langle x, y \rangle : x \text{ and } y \text{ are relatively prime}\}.$$

Theorem

RelPrime $\in$ P.

- Searching through all possible divisors of both numbers and accepting if none are greater than 1 is not efficient enough: The magnitude of a number represented in binary is exponential in the length of its representation.
- The **Euclidean algorithm** for computing the greatest common divisor of natural numbers x and y does the job in polynomial time. Recall that x mod y is the **remainder** after the integer division of x by y.

RelPrime is in P: Formal Proof

Theorem

RelPrime $\in$ P.

- The **Euclidean algorithm** E is as follows:

  E: On input $\langle x, y \rangle$, where x and y are natural numbers in binary:

  1. Repeat until $y = 0$:
     2. Assign $x \leftarrow x \bmod y$.
     3. Exchange x and y.
  4. Output x.

- Algorithm R solves RelPrime, using E as a subroutine.

  R: On input $\langle x, y \rangle$, where x and y are natural numbers in binary:

  1. Run E on $\langle x, y \rangle$.
  2. If the result is 1, accept. Otherwise, reject.

- Clearly, if E runs correctly in polynomial time, so does R. Hence, we only need to analyze E for time and correctness.

Analysis of the Euclidean Algorithm E

- Since the correctness is well known, it will not be discussed.
- We analyze the time complexity of E:
  - Every execution of Stage 2 (except possibly the first), cuts the value of x by at least half:
    After Stage 2 is executed, $x < y$ because of the nature of the mod function. After stage 3, $x > y$ because the two have been exchanged. Thus, when Stage 2 is subsequently executed, $x > y$.
    - If $\frac{x}{2} > y$, then $x \bmod y < y < \frac{x}{2}$ and x drops by at least half.
    - If $\frac{x}{2} < y$, then $x \bmod y = x - y < \frac{x}{2}$ and x drops by at least half.
  - The values of x and y are exchanged every time Stage 3 is executed, so each of the original values of x and y are reduced by at least half every other time through the loop.

Thus the maximum number of times that Stages 2 and 3 are executed is the lesser of $2\log_2 x$ and $2\log_2 y$. These are proportional to the lengths of the representations, so number of stages is $O(r)$. Each stage of E uses only polynomial time. So the total running time is polynomial.

Context-Free Languages

Theorem

Every context-free language is a member of P.

- Let L be a CFL generated by CFG G that is in Chomsky normal form. Any derivation of a string w has $2n-1$ steps, where n is the length of w because G is in Chomsky normal form. The decider for L works by trying all possible derivations with $2n-1$ steps when its input is a string of length n.
  - If any of these is a derivation of w, the decider accepts;
  - if not, it rejects.

  This algorithm does not run in polynomial time. The number of derivations with k steps may be exponential in k.
- To get a polynomial time algorithm we introduce a technique called **dynamic programming**. This technique uses the accumulation of information about smaller subproblems to solve larger problems. We record the solution to any subproblem so that it is solved only once.

Dynamic Programming for CFLs

- Dynamic programming uses the accumulation of information about smaller subproblems to solve larger problems.
- In the case of context-free languages, we consider the subproblems of determining whether each variable in G generates each substring of w.
- The algorithm enters the solution in an $n \times n$ table:
  - For $i \leq j$ the (i, j)-th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$.
  - For $i > j$ the table entries are unused.
- The algorithm fills in the table entries for each substring of w.
  - First it fills in the entries for the substrings of length 1,
  - then those of length 2, and so on.

  It uses the entries for the shorter lengths to assist in determining the entries for the longer lengths.
- Example: To determine whether a variable A generates a particular substring of length $k + 1$, the algorithm splits that substring into two nonempty pieces in the k possible ways. It then examines each rule $A \to BC$ to determine whether B, C generate the pieces, using table entries previously computed.

The Dynamic Programming Algorithm

Let G be a CFG in Chomsky normal form generating the CFL L, with S the start variable.

D: On input $w = w_1 \cdots w_n$:

1. If $w = \varepsilon$ and $S \to \varepsilon$ is a rule, accept.
2. For $i = 1$ to n:
    3. For each variable A:
        4. Test whether $A \to b$ is a rule, where $b = w_i$.
        5. If so, place A in table(i, i).
6. For $\ell = 2$ to n:
    7. For $i = 1$ to $n - \ell + 1$:
        8. Let $j = i + \ell - 1$,
        9. For $k = i$ to $j - 1$:
        10. For each rule $A \to BC$:
        11. If table(i, k) contains B and table$(k + 1, j)$ contains C, put A in table(i, j).
12. If S is in table$(1, n)$, accept. Otherwise, reject.

Analysis of the Dynamic Programming Algorithm

- Each stage is easily implemented to run in polynomial time.
  - Stages 4 and 5 run at most nv times, where v is the number of variables in G, a fixed constant independent of n.
    Hence these stages run $O(n)$ times.
  - Stage 6 runs at most n times.
    Each time Stage 6 runs, Stage 7 runs at most n times.
    Each time Stage 7 runs, Stages 8 and 9 run at most n times.
    Each time Stage 9 runs, Stage 10 runs r times, where r is the number of rules of G and is another fixed constant.
    Thus, Stage 11, the inner loop of the algorithm, runs $O\left(n^3\right)$ times.

Summing the total shows that D executes $O\left(n^3\right)$ stages.

Subsection 3

The Class NP

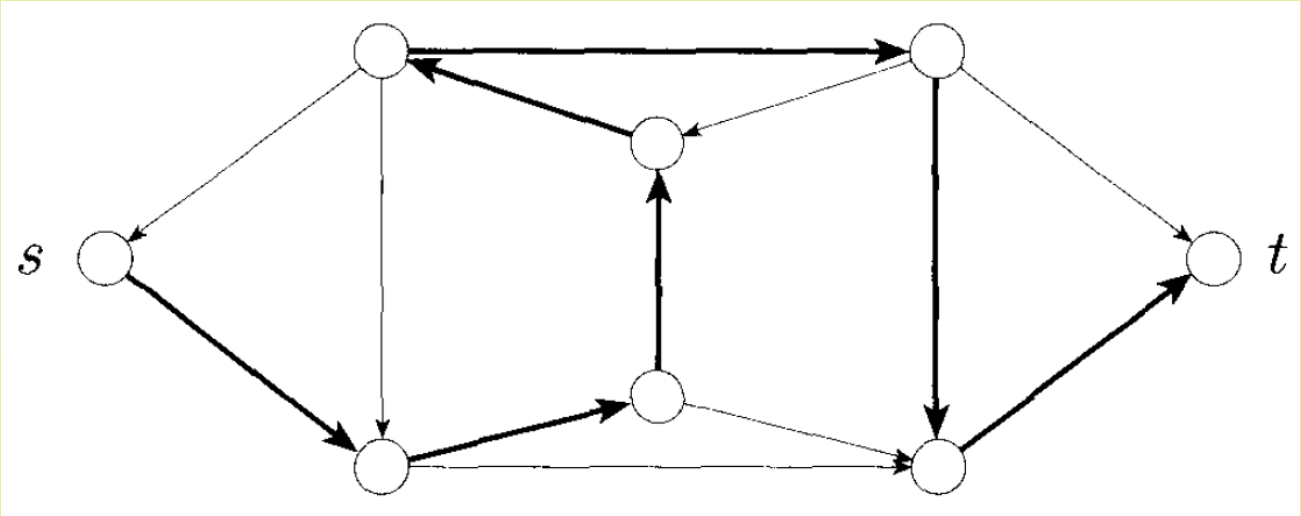
Inability to Discover Polynomial Algorithms

- In some instances, it is possible to avoid brute-force search and obtain polynomial time solutions.
- However, attempts to avoid brute force in certain other problems, including many interesting and useful ones, have not been successful, and polynomial time algorithms that solve them are not known to exist.
- The reason why success is elusive in finding polynomial time algorithms for these problems is not known.
  - Perhaps these problems have, as yet undiscovered, polynomial time algorithms that rest on unknown principles.
  - Possibly some of these problems simply cannot be solved in polynomial time. They may be intrinsically difficult.
- One remarkable discovery concerning this question is that the complexities of many problems are linked:

  A polynomial time algorithm for one such problem can be used to solve an entire class of problems.

The Hamiltonian Path Problem

- A **Hamiltonian path** in a directed graph G is a directed path that goes through each node exactly once.
- We consider the problem of testing whether a directed graph contains a Hamiltonian path connecting two specified nodes:



- Let

$$\text{HAMPATH} = \{\langle G, s, t\rangle : G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}.$$

Polynomial Verifiability

- We can easily obtain an exponential time algorithm for the HAMPATH problem by modifying the brute-force algorithm for PATH.

  We need only add a check to verify that the potential path is Hamiltonian.
- No one knows whether HAMPATH is solvable in polynomial time.
- The HAMPATH problem does have a feature called **polynomial verifiability** that is important for understanding its complexity.
- Even though we do not know of a fast (i.e., polynomial time) way to determine whether a graph contains a Hamiltonian path, if such a path were discovered somehow (perhaps using the exponential time algorithm), we could easily convince someone else of its existence, simply by presenting it.
- In other words, verifying the existence of a Hamiltonian path may be much easier than determining the existence.

The Compositeness Problem

- Another polynomially verifiable problem is compositeness:
  A natural number is **composite** if it is the product of two integers greater than 1 (i.e., a composite number is one that is not prime).
  Let
  $$\textsc{Composites} = \{x : x = pq, \text{ for integers } p, q > 1\}.$$
- We can easily verify that a number is composite - all that is needed is a divisor of that number.
- Recently, a polynomial time algorithm for testing whether a number is prime or composite was discovered, but it is considerably more complicated than the preceding method for verifying compositeness.
- Some problems may not be polynomially verifiable.
  Example: Consider $\overline{\textsc{HamPath}}$, the complement of the $\textsc{HamPath}$ problem. Even if we could determine (somehow) that a graph did not have a Hamiltonian path, we do not know of a way for someone else to verify its nonexistence without using the same exponential time algorithm for making the determination in the first place.

Verifiers

Definition (Verifier)

A **verifier** for a language A is an algorithm V, where

$$A = \{w : V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

A **polynomial time verifier** runs in polynomial time in the length of w. A language A is **polynomially verifiable** if it has a polynomial time verifier.

- A verifier uses additional information, represented by c, to verify that $w \in A$. c is called a **certificate**, or **proof**, of membership in A.
- For polynomial verifiers, the certificate has polynomial length in $|w|$ because that is all that can be accessed within the time bound.
- Example: For the HAMPATH problem, a certificate for a string $\langle G, s, t \rangle \in$ HAMPATH is the Hamiltonian path from s to t. For the COMPOSITES problem, a certificate for the composite x is one of its divisors. In both cases the verifier can check in polynomial time that the input is in the language when it is given the certificate.

The Class NP

Definition (The Class NP)

NP is the class of languages that have polynomial time verifiers.

- The class NP is important because it contains many problems of practical interest.
- Example: From the previous slide HAMPATH, COMPOSITES $\in$ NP. As we mentioned, COMPOSITES is also a member of P which is a subset of NP, but proving this stronger result is much more difficult.
- The term NP comes from **nondeterministic polynomial time** and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines.
- Problems in NP are sometimes called NP-**problems**.

Nondeterministic TM Deciding HamPath

- The following is a nondeterministic Turing machine (NTM) that decides the HamPath problem in nondeterministic polynomial time.
- Recall that we defined the time of a nondeterministic machine to be the time used by the longest computation branch.

N_1: On input $\langle G, s, t \rangle$, G is a directed graph with nodes s and t:

1. Write a list of m numbers, $p_1, \ldots, p_m$, where m is the number of nodes in G. Each number is nondeterministically selected between 1 and m.
2. Check for repetitions in the list. If any are found, reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
4. For each $i = 1, \ldots, m-1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, reject. Otherwise, all tests have been passed, so accept.

- To analyze this algorithm, we examine each of its stages:
  - In Stage 1, the nondeterministic selection runs in polynomial time.
  - In Stages 2 and 3, each part is a simple check, so together they run in polynomial time.
  - Stage 4 also clearly runs in polynomial time.

Thus, this algorithm runs in nondeterministic polynomial time.

NTM Characterization of NP

Theorem

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

- For the forward direction, let $A \in$ NP and show that A is decided by a polynomial time NTM N. Let V be the polynomial time verifier for A. Assume that V is a TM that runs in time n^k. Construct N as follows:
  N: On input w of length n:
  1. Nondeterministically select string c of length at most n^k.
  2. Run V on input $\langle w, c \rangle$.
  3. If V accepts, accept; otherwise, reject.
- For the other direction, assume that A is decided by a polynomial time NTM N. Construct a polynomial time verifier V as follows:
  V: On input $\langle w, c \rangle$, where w and c are strings:
  1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to be made at each step.
  2. If this branch of N's computation accepts, accept; otherwise, reject.

The Classes NTIME($t(n)$)

- We define the nondeterministic time complexity class NTIME($t(n)$):

Definition (The Class NTIME($t(n)$))

$$\text{NTIME}(t(n)) = \{L : L \text{ is a language decided by a } \mathrm{O}(t(n)) \text{ time nondeterministic Turing machine}\}.$$

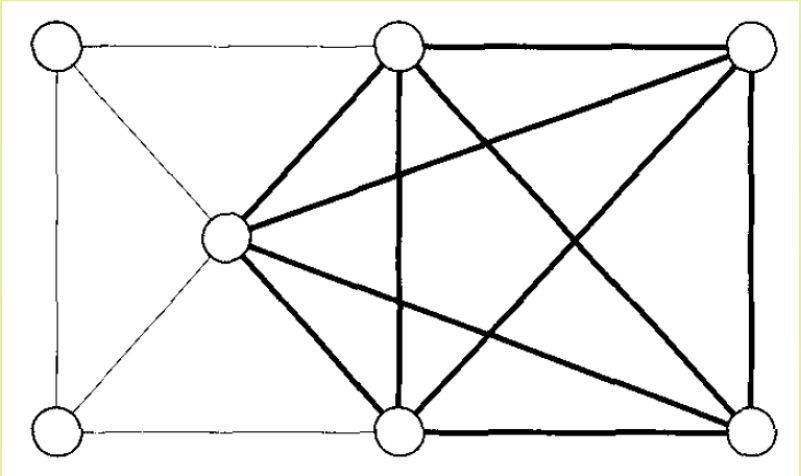
Corollary

$\text{NP} = \bigcup_k \text{NTIME}(n^k)$.

- NP is not sensitive to the choice of reasonable nondeterministic (ND) computational model because all such models are polynomially equivalent.
- When describing and analyzing nondeterministic polynomial time algorithms, we show that:
  - Every branch uses at most polynomially many stages.
  - Each stage has an obvious implementation in ND polynomial time on a reasonable ND computational model.

The Clique Problem in Undirected Graphs

- A **clique** in an undirected graph is a subgraph, wherein every two nodes are connected by an edge. A k-**clique** is a clique that contains k nodes.
- Example: The following is a graph having a 5-clique:



- The clique problem is to determine whether a graph contains a clique of a specified size:

$$\textsc{Clique} = \{\langle G, k\rangle : G \text{ is an undirected graph with a } k\text{-clique}\}.$$

CLIQUE is in NP

Theorem

CLIQUE is in NP.

- The clique is the certificate. More precisely, the following is a verifier V for CLIQUE:

  V: On input $\langle\langle G, k\rangle, c\rangle$:

  1. Test whether c is a set of k nodes in G.
  2. Test whether G contains all edges connecting nodes in c.
  3. If both pass, accept; otherwise, reject.

- An alternative proof involves providing a nondeterministic polynomial time Turing machine that decides CLIQUE (observe the similarity):

  N: On input $\langle G, k\rangle$, where G is a graph:

  1. Nondeterministically select a subset c of k nodes of G.
  2. Test whether G contains all edges connecting nodes in c.
  3. If yes, accept; otherwise, reject.

The Subset Sum Problem

- We have a collection of numbers $x_1, \ldots, x_k$ and a target number t. Determine whether the collection contains a subcollection that adds up to t.

$$\textsc{SubsetSum} = \{\langle S, t\rangle : S = \{x_1, \ldots, x_k\} \text{ and, for some } \{y_1, \ldots, y_\ell\} \subseteq \{x_1, \ldots, x_k\}, \sum y_i = t\}.$$

- Example: $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in \textsc{SubsetSum}$ because $4 + 21 = 25$.
- Here $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_\ell\}$ are considered to be multisets and, thus, repetition of elements is allowed.

The Subset Sum Problem is in NP

Theorem

SUBSETSUM $\in$ NP.

- The subset is the certificate. The following is a verifier V for SUBSETSUM:

  V: On input $\langle\langle S, t\rangle, c\rangle$:

  1. Test whether c is a collection of numbers that sum to t.
  2. Test whether S contains all the numbers in c.
  3. If both pass, accept; otherwise, reject.

- For a nondeterministic polynomial time Turing machine for SUBSETSUM:

  N: On input $\langle S, t\rangle$:

  1. Nondeterministically select a subset c of the numbers in S.
  2. Test whether c is a collection of numbers that sum to t.
  3. If the test passes, accept; otherwise, reject.

The Class coNP

- The complements of CLIQUE and SUBSETSUM are not obviously members of NP.
- Verifying that something is not present seems to be more difficult than verifying that it is present.
- We make a separate complexity class, called coNP, which contains the languages that are complements of languages in NP.
- We do not know whether coNP is different from NP.

P Versus NP

- Loosely referring to polynomial time solvable as solvable "quickly":

  P = the class of languages for which membership can be decided quickly.
  NP = the class of languages for which membership can be verified quickly.

- We have presented examples of languages, such as HAMPATH and CLIQUE, that are members of NP but that are not known to be in P.
- The power of polynomial verifiability seems to be much greater than that of polynomial decidability. But P and NP could be equal. The question of whether P = NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics.
- If these classes were equal, any polynomially verifiable problem would be polynomially decidable.
- The best method known for solving languages in NP deterministically uses exponential time, i.e., $NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k})$,
- We do not know whether NP is contained in a smaller deterministic time complexity class.

Subsection 4

NP-completeness

Introducing NP-Completeness

- In the early 1970s Stephen Cook and Leonid Levin discovered certain problems in NP whose individual complexity is related to that of the entire class.
- If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable.
- These problems are called NP-**complete**.
- NP-completeness is key for both theoretical and practical reasons:
  - On the theoretical side, a researcher trying to show that P is unequal to NP may focus on an NP-complete problem.
    - If any problem in NP requires more than polynomial time, an NP-complete one does.
    - Attempting to prove that P equals NP only requires finding a polynomial time algorithm for an NP-complete problem.
  - On the practical side, the phenomenon of NP-completeness may prevent wasting time searching for a nonexistent polynomial time algorithm to solve a particular problem.
    - The belief that P is unequal to NP implies that proving a problem is NP-complete is strong evidence of its non-polynomiality.

Boolean Formulas

- Our first NP-complete problem is called the **satisfiability problem**.
- Variables that can take on the values TRUE and FALSE are called **Boolean variables**.
- Usually, we represent TRUE by 1 and FALSE by 0.
- The Boolean operations AND, OR, and NOT, represented by the symbols $\wedge$, $\vee$, and $\neg$, respectively, are described below, where the overbar is used as a shorthand for $\neg$, i.e., $\overline{x}$ means $\neg x$:

AND	OR	NOT
$0 \wedge 0 = 0$	$0 \vee 0 = 0$	$\overline{0} = 1$
$0 \wedge 1 = 0$	$0 \vee 1 = 1$	$\overline{1} = 0$
$1 \wedge 0 = 0$	$1 \vee 0 = 1$	
$1 \wedge 1 = 1$	$1 \vee 1 = 1$	

- A **Boolean formula** is an expression involving Boolean variables and operations.
- Example: $\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$ is a Boolean formula.

Boolean Formula Satisfiability

- A Boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.
- Example: The formula $\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$ is satisfiable because the assignment $x = 0$, $y = 1$, and $z = 0$ makes ϕ evaluate to 1. We say the assignment **satisfies** ϕ.
- The **satisfiability problem** is to test whether a Boolean formula is satisfiable:

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

- The Cook-Levin theorem links the complexity of the $\textsc{Sat}$ problem to the complexities of all problems in NP:

The Cook-Levin Theorem

$\textsc{Sat} \in \mathrm{P}$ iff $\mathrm{P} = \mathrm{NP}$.

- We now develop the method that is central to the proof of the Cook-Levin theorem.

Polynomial Time Reducibility

- Recall the concept of reducing one problem to another: If problem A reduces to problem B, a solution to B can be used to solve A.
- Efficient reducibility of problem A to a problem B should imply that an efficient solution to B can be used to solve A efficiently:

Definition (Polynomial Time Computable Function)

A function $f : \Sigma^* \to \Sigma^*$ is a **polynomial time computable function** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

Definition (Polynomial Time Reducibility)

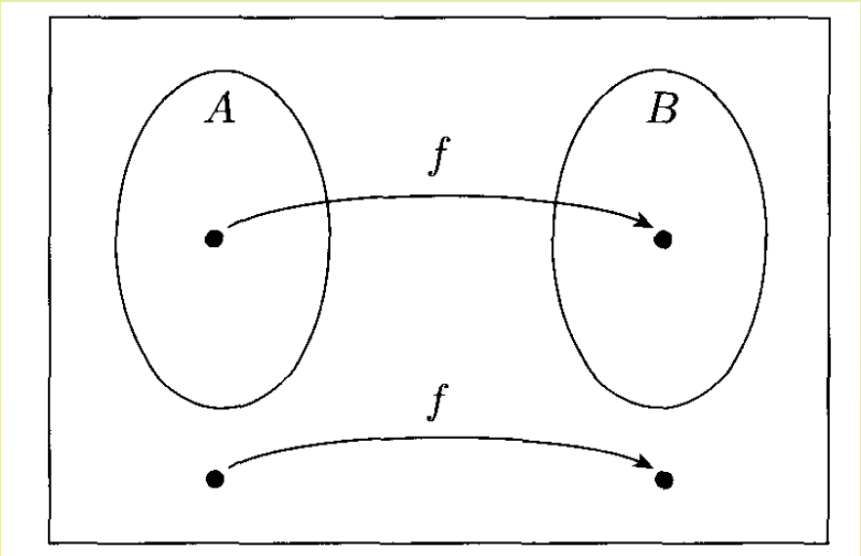
Language A is **polynomial time mapping reducible**, or simply **polynomial time reducible**, to language B, written $A \leq_{\mathrm{P}} B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists, such that, for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the **polynomial time reduction** of A to B.

Polynomial Reducibility and Language Membership

- As with an ordinary mapping reduction, a polynomial time reduction of A to B provides a way to convert membership testing in A to membership testing in B, with the conversion done efficiently.



- To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$.

Polynomial Reducibility and Polynomial Decidability

Theorem

If $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.

- Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. A polynomial time algorithm N deciding A works as follows:

  N: On input w:

  1. Compute $f(w)$.
  2. Run M on input $f(w)$ and output whatever M outputs.

  - We have $w \in A$ if and only if $f(w) \in B$ because f is a reduction from A to B. Thus M accepts $f(w)$ if and only if $w \in A$.
  - Moreover, N runs in polynomial time because each of its two stages runs in polynomial time. Stage 2 runs in polynomial time because the composition of two polynomials is a polynomial.

The 3CNF-Formula Satisfiability Problem

- The problem 3SAT is a special case of the satisfiability problem in which all formulas are in a special form.
- A **literal** is a Boolean variable or a negated Boolean variable, as in x or $\overline{x}$.
- A **clause** is several literals connected with $\vee$s, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$.
- A Boolean formula is in **conjunctive normal form**, called a **CNF-formula**, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6}).$$

- It is a **3CNF-formula** if all the clauses have three literals, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

- Let

$$3\text{SAT} = \{\langle\phi\rangle : \phi \text{ is a satisfiable 3CNF-formula}\}.$$

- In a satisfiable CNF-formula, each clause must contain at least one literal that is assigned 1.

Reduction of 3SAT to CLIQUE

Theorem

3SAT is polynomial time reducible to CLIQUE.

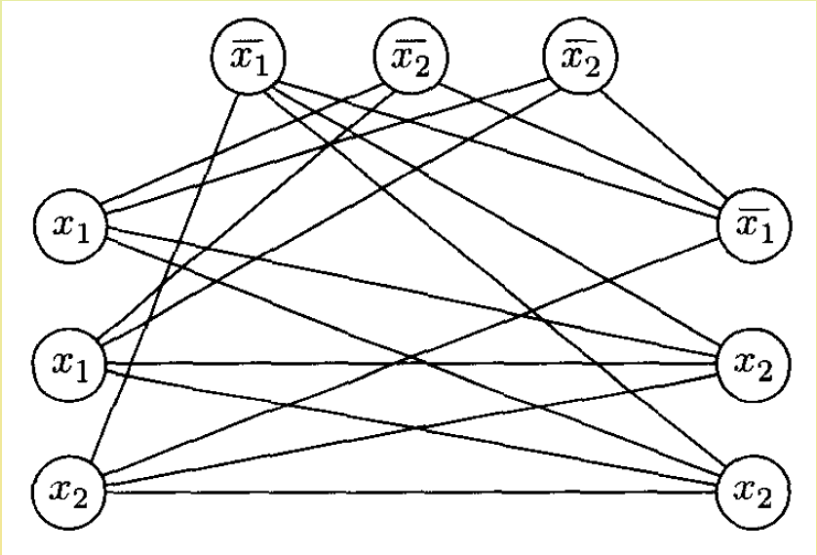
- The polynomial time reduction f from 3SAT to CLIQUE converts formulas to graphs. In the constructed graphs, cliques of a specified size correspond to satisfying assignments of the formula. Structures within the graph are designed to mimic the behavior of the variables and clauses.
- Let ϕ be a formula with k clauses such as

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k).$$

  The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows:
  - The nodes in G are organized into k groups of three nodes each called the **triples**, $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in ϕ, and each node in a triple corresponds to a literal in the associated clause. Label each node of G with its corresponding literal in ϕ.

Reduction of 3SAT to CLIQUE (Cont'd)

- f generates the string $\langle G, k \rangle$. We continue the description of the undirected graph G:
  - Recall each node of G is labeled with the literal in ϕ to which it corresponds.
  - The edges of G connect all but two types of pairs of nodes in G:
    - No edge is present between nodes in the same triple;
    - No edge is present between two nodes with contradictory labels, as in x_2 and $\overline{x_2}$.
- Example: Consider $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.



Correctness of the Reduction of 3SAT to CLIQUE

- We show that ϕ is satisfiable iff G has a k-clique.
  - Suppose that ϕ has a satisfying assignment. In that satisfying assignment, at least one literal is true in every clause. In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. The nodes just selected form a k-clique:
    - The number of nodes selected is k, because we chose one for each of the k triples.
    - Each pair of selected nodes is joined by an edge because no pair fits one of the exceptions described previously: They could not be from the same triple because we selected only one node per triple. They could not have contradictory labels because the associated literals were both true in the satisfying assignment.
  - Suppose that G has a k-clique. No two of the clique's nodes occur in the same triple, so each of the k triples contains exactly one of the k clique nodes. Each literal in ϕ labeling a clique node is made true. Doing so is always possible because two contradicting nodes are not connected by an edge and, hence, cannot both be in the clique. This assignment satisfies ϕ because each triple contains a clique node and, hence, each clause contains a literal that is assigned TRUE.

NP-Complete Languages

- Thus, if CLIQUE is solvable in polynomial time, so is 3SAT.
- At first glance, this connection between these two problems appears quite remarkable because, superficially, they are rather different.
- Polynomial time reducibility links their complexities.

Definition (NP-Complete Language)

A language B is NP-**complete** if it satisfies two conditions:

1. B is in NP;
2. Every A in NP is polynomial time reducible to B.

Theorem

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

- We know $\mathrm{P} \subseteq \mathrm{NP}$. If $A \in \mathrm{NP}$, then, since B is NP-complete, $A \leq_{\mathrm{P}} B$. But $B \in \mathrm{P}$, whence $A \in \mathrm{P}$. This shows that $\mathrm{NP} \subseteq \mathrm{P}$.

NP-Completeness and Polynomial Reducibility

Theorem

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

- We are given that C is in NP.
- It remains to show that every A in NP is polynomial time reducible to C. Because B is NP-complete, every language in NP is polynomial time reducible to B. B in turn is polynomial time reducible to C. Polynomial time reductions compose:

  If A is polynomial time reducible to B and B is polynomial time reducible to C, then A is polynomial time reducible to C.

  Thus, every language in NP is polynomial time reducible to C.

The First NP-Complete Problem

- Once we have one NP-complete problem, we may obtain others by polynomial time reduction from it.
- Our first NP-complete problem is SAT.

The Cook-Levin Theorem (Restated)

SAT is NP-complete.

- Showing that SAT is in NP is easy. The hard part of the proof is showing that any language in NP is polynomial time reducible to SAT. To do so we construct a polynomial time reduction for each language A in NP to SAT.
  The reduction for A takes a string w and produces a Boolean formula ϕ that simulates the NP machine for A on input w.
  - If the machine accepts, ϕ has a satisfying assignment that corresponds to the accepting computation.
  - If the machine does not accept, no assignment satisfies ϕ.

  Therefore w is in A if and only if ϕ is satisfiable.

SAT is in NP

- First, SAT is in NP: A nondeterministic polynomial time machine can guess an assignment to a given formula ϕ and accept if the assignment satisfies ϕ.
- Next, we take any language A in NP and show that A is polynomial time reducible to SAT.

  Let N be a nondeterministic Turing machine that decides A in n^k time for some constant k.

  For technical convenience, we actually assume that N runs in time $n^k - 3$.

  A **tableau** for N on w is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of N on input w:

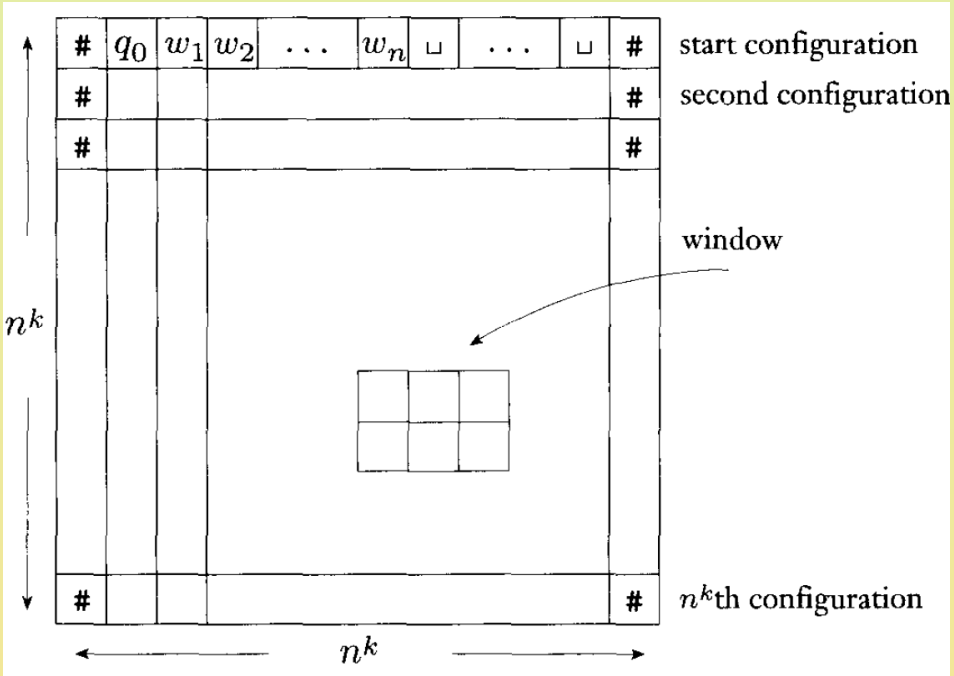


Tableau Functionality

- For convenience later we assume that each configuration starts and ends with a # symbol, so the first and last columns of a tableau are all #s.
  - The first row of the tableau is the starting configuration of N on w;
  - Each row follows the previous one according to N's transition function.
  - A tableau is **accepting** if any row of the tableau is an accepting configuration.

  Every accepting tableau for N on w corresponds to an accepting computation branch of N on w.

  Thus, the problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tableau for N on w exists.

The Polynomial Time Reduction

- Now we get to the description of the polynomial time reduction f from A to SAT: On input w, the reduction produces a formula ϕ.

  We begin by describing the variables of ϕ: Say that Q and Γ are the state set and tape alphabet of N. Let $C = Q \cup \Gamma \cup \{\#\}$. For each i and j between 1 and n^k and for each s in C we have a variable, $x_{i,j,s}$. Each of the $(n^k)^2$ entries of a tableau is called a **cell**. The cell in row i and column j is called cell$[i,j]$ and contains a symbol from C. We represent the contents of the cells with the variables of ϕ. If $x_{i,j,s}$ takes on the value 1, it means that cell$[i,j]$ contains an s.
- Now we design ϕ so that a satisfying assignment to the variables does correspond to an accepting tableau for N on w. The formula ϕ is the AND of four parts

$$\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}},$$

  which are described next.

The Formula ϕ_{cell}

- The first thing we must guarantee in order to obtain a correspondence between an assignment and a tableau is that the assignment turns on exactly one variable for each cell. Formula ϕ_{cell} ensures this requirement by expressing it in terms of Boolean operations:

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

ϕ_{cell} is actually a large expression that contains a fragment for each cell in the tableau because i and j range from 1 to n^k.

  - The first part of each fragment says that at least one variable is turned on in the corresponding cell.
  - The second part of each fragment says that no more than one variable is turned on (literally, it says that for each pair of variables, at least one is turned off) in the corresponding cell.

The Formula ϕ_{start}

- Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}
\phi_{\text{start}} = \quad & x_{1,1,\#} \wedge && \text{(margin)} \\
& x_{1,2,q_0} \wedge && \text{(start state)} \\
& x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \cdots \wedge x_{1,n+2,w_n} \wedge && \text{(input)} \\
& x_{1,n+3,\sqcup} \wedge \cdots \wedge x_{1,n^k-1,\sqcup} \wedge && \text{(blanks)} \\
& x_{1,n^k,\#}. && \text{(margin)}
\end{aligned}$$

The Formula ϕ_{accept}

- Formula ϕ_{accept} guarantees that an accepting configuration occurs in the tableau. It ensures that q_{accept}, the symbol for the accept state, appears in one of the cells of the tableau, by stipulating that one of the corresponding variables is on:

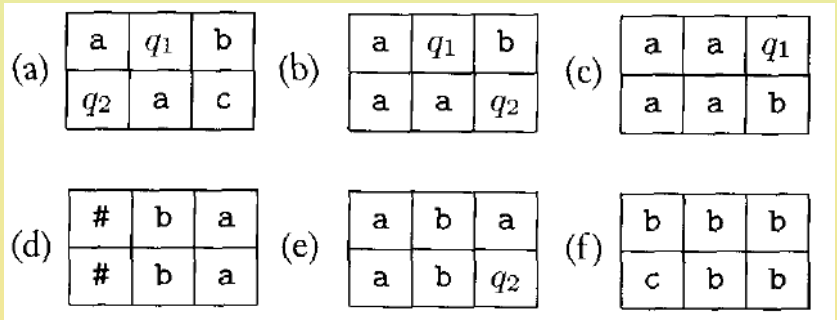
$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}.$$

Preparation for the Formula ϕ_{move}

- Finally, formula ϕ_{move} guarantees that each row of the table corresponds to a configuration that legally follows the preceding row's configuration according to N's rules. It does so by ensuring that each 2×3 window of cells is legal.

  We say that a 2×3 window is **legal** if that window does not violate the actions specified by N's transition function.
- Example: Say that a, b and c are members of the tape alphabet and q_1 and q_2 are states of N. Assume that:
  - When in state q, with the head reading an a, N writes a b, stays in state q_1 and moves right, formally $\delta(q_1, a) = \{(q_1, b, R)\}$;
  - When in state q_1, with the head reading a b, N nondeterministically:

    1. writes a c, enters q_2 and moves to the left, or
    2. writes an a, enters q_2 and moves to the right.

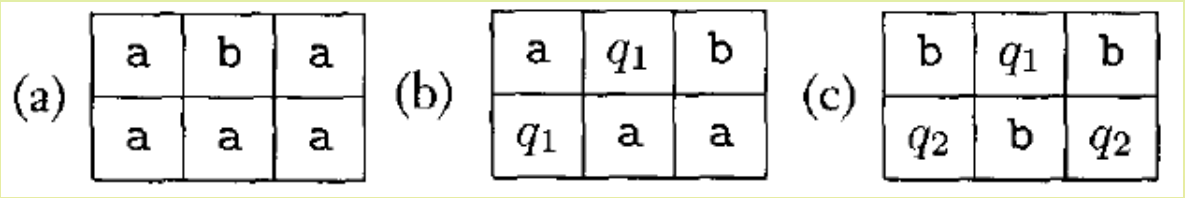


Examples of legal windows for this machine:

Example (Cont'd)

- We are looking at a nondeterministic machine N with $\delta(q_1, a) = \{(q_1, b, R)\}$ and $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$.

  The following windows are not legal for machine N:



  - In window (a) the central symbol in the top row cannot change because a state is not adjacent to it.
  - Window (b) is not legal because the transition function specifies that the b gets changed to a c but not to an a.
  - Window (c) is not legal because two states appear in the bottom row.

A Claim Concerning ϕ_{move}

- Claim: If the top row of the table is the start configuration and every window in the table is legal, each row of the table is a configuration that legally follows the preceding one.

  Consider any two adjacent configurations in the table, called the **upper configuration** and the **lower configuration**.
  - In the upper configuration, every cell that is not adjacent to a state symbol and that does not contain the boundary symbol #, is the center top cell in a window whose top row contains no states. Therefore, that symbol must appear unchanged in the center bottom of the window. Hence, it appears in the same position in the bottom configuration.
  - The window containing the state symbol in the center top cell guarantees that the corresponding three positions are updated consistently with the transition function.

  Therefore, if the upper configuration is a legal configuration, so is the lower configuration, and the lower one follows the upper one according to N's rules.

The Formula ϕ_{move}

- The formula ϕ_{move} stipulates that all the windows in the tableau are legal.
- Each window contains six cells, which may be set in a fixed number of ways to yield a legal window.
- The formula says that the settings of those six cells must be one of these ways:

$$\phi_{\text{move}} = \bigwedge_{\substack{1<i\leq n^k \\ 1<j<n^k}} (\text{the } (i,j) \text{ window is legal}).$$

- Writing the contents of six cells of a window as $a_1, a_2, \ldots, a_6$, we replace the text "the (i,j) window is legal" with the formula:

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}).$$

Complexity of the Reduction

- To analyze the complexity of the reduction and show that it operates in polynomial time, we examine the size of ϕ.
  - First, we estimate the number of variables it has. The tableau is an $n^k \times n^k$ table, so it contains n^{2k} cells. Each cell has ℓ variables associated with it, where ℓ is the number of symbols in C. Because ℓ depends only on the TM N and not on the length of the input n, the total number of variables is $O\left(n^{2k}\right)$.
  - We estimate, next, the size of each of the parts of ϕ.
    - Formula ϕ_{cell} contains a fixed size fragment of the formula for each cell of the tableau. So its size is $O\left(n^{2k}\right)$.
    - ϕ_{start} has a fragment for each cell in the top row, i.e., size $O\left(n^k\right)$.
    - Formulas ϕ_{move} and ϕ_{accept} each contain a fixed-size fragment of the formula for each cell of the tableau. So their size is $O\left(n^{2k}\right)$.

    Thus ϕ's total size is $O\left(n^{2k}\right)$. The bound shows that the the size of ϕ is polynomial in n.

ϕ can be produced in polynomial time from the input w, because each component is composed of many nearly identical fragments, varying only with the indices in a simple way.

NP-Completeness of 3SAT

Corollary

3SAT is NP-complete.

- It is clear that 3SAT is in NP. Thus, it suffices to show that all languages in NP reduce to 3SAT in polynomial time. One way is by showing that SAT polynomial time reduces to 3SAT. Instead, we modify the proof of the preceding theorem so that it directly produces a formula in conjunctive normal form with three literals per clause.
  It produces a formula that is almost in conjunctive normal form:
  - Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus, ϕ_{cell} is an AND of clauses and, so, is already in CNF.
  - Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in CNF.
  - Formula ϕ_{accept} is a big OR of variables and is thus a single clause.
  - Formula ϕ_{move} is the only one that is not already in CNF.

Converting ϕ_{move} into 3CNF

- We must convert ϕ_{move} into CNF.
  - Recall that ϕ_{move} is a big AND of subformulas, each of which is an OR of ANDs that describes all possible legal windows. The distributive laws of Boolean logic state that we can replace an OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each subformula, but it can only increase the total size of ϕ_{move} by a constant factor because the size of each subformula depends only on N. The result is a formula that is in conjunctive normal form.
- The formula is now written in CNF.
- We convert it to one with three literals per clause:
  - In each clause that currently has one or two literals, we replicate one of the literals until the total number is three.
  - In each clause that has more than three literals, we split it into several clauses and add additional variables to preserve the satisfiability or non-satisfiability of the original.

Illustrating the Conversion from CNF to 3CNF

- Suppose we want to convert the clause $(a_1 \vee a_2 \vee a_3 \vee a_4)$, wherein each a_i is a literal, into 3CNF.

  We replace it by the two-clause expression
  $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$, wherein z is a new variable.
  If some setting of the a_i's satisfies the original clause, we can find some setting of z so that the two new clauses are satisfied.
- In general, if the clause contains ℓ literals, $(a_1 \vee a_2 \vee \cdots \vee a_\ell)$, we can replace it with the $\ell - 2$ clauses
  $(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{\ell-3}} \vee a_{\ell-1} \vee a_\ell)$.
  We may easily verify that the new formula is satisfiable iff the original formula was.

Subsection 5

Additional NP-complete Problems

Reductions and Gadgets

- We now present additional theorems showing that various languages are NP-complete. They provide examples of relevant techniques.
- The general strategy is to exhibit a polynomial time reduction from 3SAT to the language in question. But, sometimes, reduction from other NP-complete languages may be more convenient.
- When constructing a polynomial time reduction from 3SAT to a language, we look for structures, called **gadgets**, in that language that can simulate the variables and clauses in Boolean formulas.
- Example: In the reduction from 3SAT to CLIQUE, individual nodes simulate variables and triples of nodes simulate clauses. An individual node may or may not be a member of the clique, which corresponds to a variable that may or may not be true in a satisfying assignment.

Corollary

CLIQUE is NP-complete.

The Vertex Cover Problem

- If G is an undirected graph, a **vertex cover** of G is a subset of the nodes, such that every edge of G touches at least one of them.
- The vertex cover problem asks whether a graph contains a vertex cover of a specified size:

$$\textsc{VertexCover} = \{\langle G, k\rangle : G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}.$$

Theorem

$\textsc{VertexCover}$ is NP-complete.

- We must show that $\textsc{VertexCover}$ is in NP and that all NP-problems are polynomial time reducible to it.
  - The first part is easy: a certificate is simply a vertex cover of size k.
  - To prove the second part we show that $3\textsc{Sat}$ is polynomial time reducible to $\textsc{VertexCover}$. The reduction converts a 3CNF-formula ϕ into a graph G and a number k, so that ϕ is satisfiable whenever G has a vertex cover with k nodes. The conversion is done without knowing whether ϕ is satisfiable and, in effect, G simulates ϕ by using gadgets that mimic the variables and clauses of the formula.

Informal Description of the Gadgets

- Designing the gadgets requires a bit of ingenuity.
  - For the variable gadget, we look for a structure in G that can participate in the vertex cover in either of two possible ways, corresponding to the two possible truth assignments to the variable. Two nodes connected by an edge is a structure that works, because one of these nodes must appear in the vertex cover. We arbitrarily associate TRUE and FALSE to these two nodes.
  - For the clause gadget, we look for a structure that induces the vertex cover to include nodes in the variable gadgets corresponding to at least one true literal in the clause. The gadget contains three nodes and additional edges so that any vertex cover must include at least two of the nodes, or possibly all three.
    - Only two nodes would be required if one of the vertex gadget nodes helps by covering an edge, as would happen if the associated literal satisfies that clause.
    - Otherwise three nodes would be required.
- Finally, we chose k so that the sought-after vertex cover has one node per variable gadget and two nodes per clause gadget.

The Formal Construction

- The reduction from 3SAT to VERTEXCOVER maps a Boolean formula ϕ to a graph G and a value k.
  - For each variable x in ϕ, we produce an edge connecting two nodes. We label the two nodes in this gadget x and $\overline{x}$. Setting x to be TRUE corresponds to selecting the left node for the vertex cover, whereas FALSE corresponds to the right node.
  - Each clause gadget is a triple of three nodes that are labeled with the three literals of the clause. These three nodes are connected to each other and to the nodes in the variable gadgets that have the identical labels.

  Thus, the total number of nodes that appear in G is $2m + 3\ell$, where ϕ has m variables and ℓ clauses.

  Finally, let $k = m + 2\ell$.

Illustrating the Construction

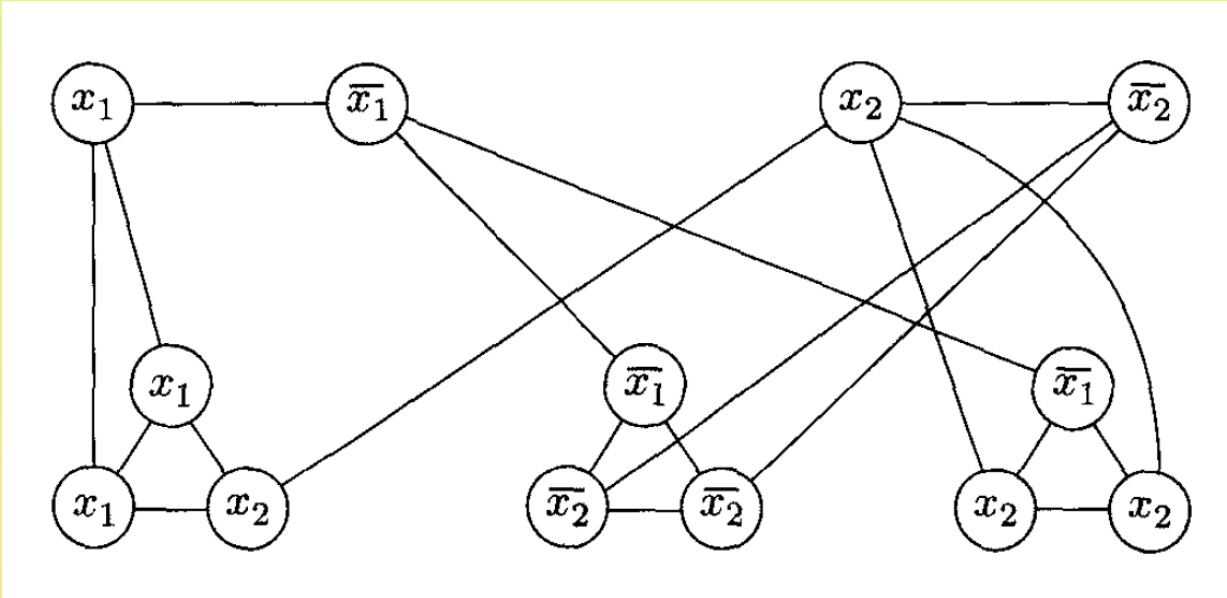
- For example, if

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2),$$

we have $m = 2$ and $\ell = 3$ and the reduction produces $\langle G, k \rangle$ from ϕ, where

  - G has $2 \cdot 2 + 3 \cdot 3 = 13$ nodes;
  - $k = 2 + 2 \cdot 3 = 8$.

G is:



Proof of the Correctness

- Suppose ϕ has a satisfying assignment. We put the nodes of the variable gadgets that correspond to the true literals in the assignment into the vertex cover. Then, we select one true literal in every clause and put the remaining two nodes from every clause gadget into the vertex cover. Now, we have a total of $k = m + 2\ell$ nodes. They cover all edges because:
  - every variable gadget edge is clearly covered,
  - all three edges within every clause gadget are covered, and
  - all edges between variable and clause gadgets are covered.

  Hence G has a vertex cover with k nodes.
- If G has a vertex cover with k nodes, it must contain:
  - one node in each variable gadget, in order to cover its edge, and
  - two in every clause gadget, in order to cover its three edges.

  We take the nodes of the variable gadgets that are in the vertex cover and assign the corresponding literals TRUE. That assignment satisfies ϕ because each of the three edges connecting the variable gadgets with each clause gadget is covered and only two nodes of the clause gadget are in the vertex cover. Therefore, one of the edges must be covered by a node from a variable gadget. So that assignment satisfies the corresponding clause.

The Hamiltonian Path Problem

- The Hamiltonian path problem asks whether the input graph contains a path from s to t that goes through every node exactly once.

Theorem

HAMPATH is NP-complete.

- We have already seen that HAMPATH is in NP.
- To show that every NP-problem is polynomial time reducible to HAMPATH, we show that 3SAT is polynomial time reducible to HAMPATH. We give a way to convert 3CNF-formulas to graphs in which Hamiltonian paths correspond to satisfying assignments of the formula. The graphs contain gadgets that mimic variables and clauses:
  - The variable gadget is a diamond structure that can be traversed in either of two ways, corresponding to the two truth settings.
  - The clause gadget is a node. Ensuring that the path goes through each clause gadget corresponds to ensuring that each clause is satisfied in the satisfying assignment.

The 3CNF Formula

- Since we know that HamPath is in NP, we only need to show $3\text{Sat} \leq_{\text{P}} \text{HamPath}$.

  For each 3CNF-formula ϕ, we show how to construct a directed graph G with two nodes, s and t, so that a Hamiltonian path exists between s and t iff ϕ is satisfiable. Let

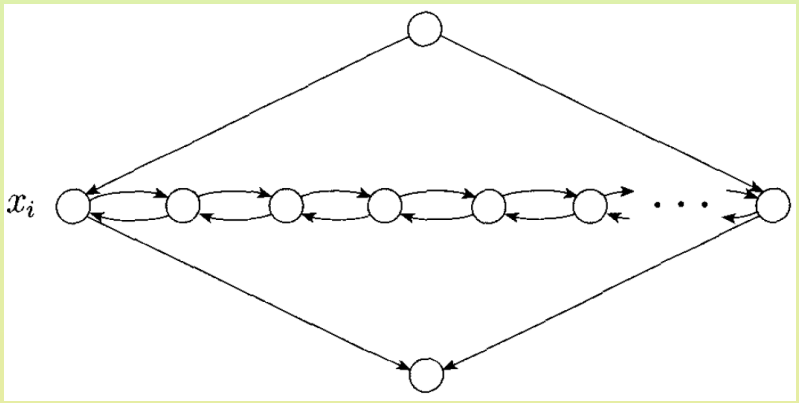
  $$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k),$$

  where each a, b and c is a literal x_i or $\overline{x_i}$. Let $x_1, \ldots, x_\ell$ be the ℓ variables of ϕ.

  Now we show how to convert ϕ to a graph G. The graph G has various parts to represent the variables and clauses that appear in ϕ.

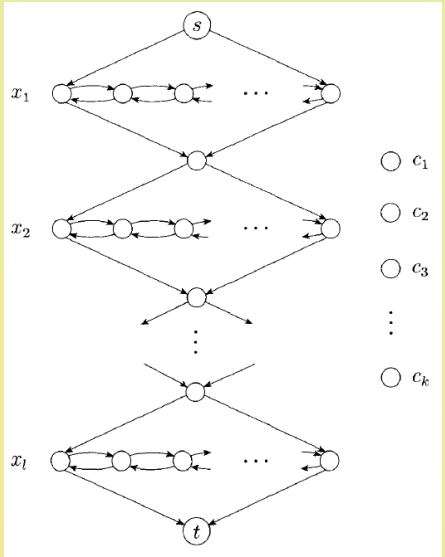
The Variable and the Clause Gadgets

- We represent each variable x_i with a diamond-shaped structure that contains a horizontal row of nodes:



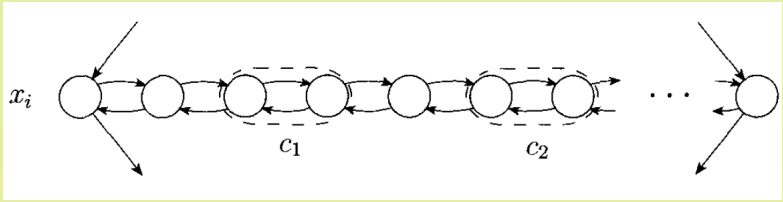
The number of nodes in the horizontal row will be specified later.

- We represent each clause as a single node.
- The global structure of G is shown here: It shows all the elements of G and their relationships, except the edges that represent the relationship of the variables to the clauses that contain them.

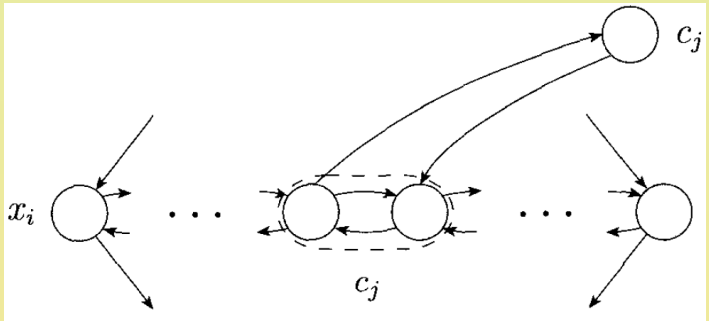


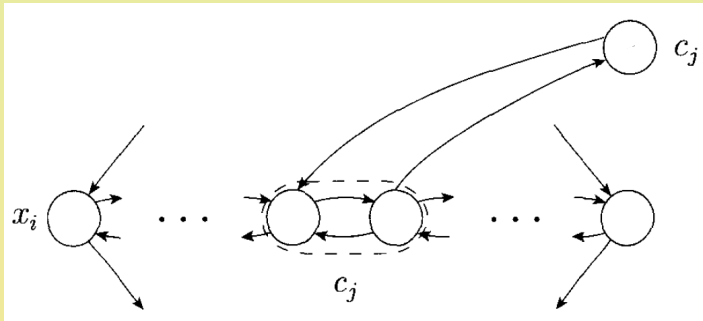
Connecting the Diamonds with the Nodes

- Each diamond structure contains a horizontal row of nodes connected by edges running in both directions. The horizontal row contains $3k + 1$ nodes in addition to the two nodes on the ends belonging to the diamond. These nodes are grouped into adjacent pairs, one for each clause, with extra separator nodes next to the pairs:



- If variable x_i appears in clause c_j, we add the following two edges from the j-th pair in the i-th diamond to the j-th clause node:





If $\overline{x_i}$ appears in clause c_j, we add two edges from the j-th pair in the i-th diamond to the j-th clause node, as shown on the right:

From Satisfiability to Hamiltonicity

- Suppose that ϕ is satisfiable. The Hamiltonian path begins at s, goes through each diamond in turn, and ends up at t. To hit the horizontal nodes in a diamond, the path either zig-zags from left to right or zag-zigs from right to left:
  - If x_i is assigned TRUE, the path zig-zags through the diamond.
  - If x_i is assigned FALSE, the path zag-zigs.

  To cover the clause nodes, we add detours at the horizontal nodes. In each clause, we select a literal assigned TRUE by the assignment:
  - If we selected x_i in clause c_j, we can detour at the j-th pair in the i-th diamond. This is possible because x_i must be TRUE, so the path zig-zags from left to right through the corresponding diamond. Hence the edges to the c_j node are in order that allows a detour and return.
  - Similarly, if we selected $\overline{x_i}$ in clause c_j, we can detour at the j-th pair in the i-th diamond because x_i must be FALSE, so the path zag-zigs from right to left through the corresponding diamond. Hence the edges to the c_j node again are in the correct order to allow a detour and return.

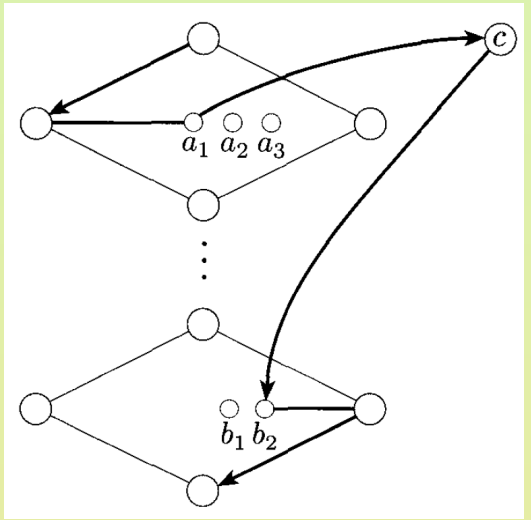
  Thus, we have constructed the desired Hamiltonian path.

From Hamiltonicity to Satisfiability

- For the reverse direction, if G has a Hamiltonian path from s to t, we demonstrate a satisfying assignment for ϕ.
  - If the Hamiltonian path is *normal*, i.e., it goes through the diamonds in order from the top one to the bottom one, except for the detours to the clause nodes, we can easily obtain the satisfying assignment.
    - If the path zig-zags through the diamond, we assign the corresponding variable TRUE;
    - If it zag-zigs, we assign FALSE.

    Because each clause node appears on the path, by observing how the detour is taken, we may determine which of the literals in the corresponding clause is TRUE.
  - Normality may fail only if the path enters a clause from one diamond but returns to another. In the next slide, we show that this is not possible. Hence a Hamiltonian path must be normal.
- Since the reduction obviously operates in polynomial time, the proof is complete.

Normality is Necessary in a Hamiltonian Path



- Normality may fail only if the path enters a clause from one diamond but returns to another. The path goes from node a_1 to c, but instead of returning to a_2 in the same diamond, it returns to b_2 in a different diamond. If that occurs, either a_2 or a_3 must be a separator node.
  - If a_2 were a separator node, the only edges entering a_2 would be from a_1 and a_3.
  - If a_3 were a separator node, a_1 and a_2 would be in the same clause pair, and hence the only edges entering a_2 would be from a_1, a_3 and c.

  In either case, the path could not contain node a_2:
  - The path cannot enter a_2 from c or a_1 because the path goes elsewhere from these nodes.
  - The path cannot enter a_2 from a_3, because a_3 is the only available node that a_2 points at, so the path must exit a_2 via a_3.

Undirected Hamiltonian Path Problem

- To show an undirected version UHAMPATH of the Hamiltonian path problem is NP-complete we give a polynomial time reduction from HAMPATH.

Theorem

UHAMPATH is NP-complete.

- The reduction takes a directed graph G with nodes s and t and constructs an undirected graph G' with nodes s' and t'. G has a Hamiltonian path from s to t iff G' has a Hamilton path from s' to t'.
- G' is constructed as follows:
  - Each node u of G, except for s and t, is replaced by a triple of nodes u^{in}, u^{mid} and u^{out} in G'. Nodes s and t in G are replaced by nodes s^{out} and t^{in} in G'.
  - Edges of two types appear in G'.
    - Edges connect u^{mid} with u^{in} and u^{out}.
    - An edge connects u^{out} with v^{in} if an edge goes from u to v in G.

  This completes the construction of G'.

Correctness of the Reduction

- We show that G has a Hamiltonian path from s to t iff G' has a Hamiltonian path from s^{out} to t^{in}.
  - To show one direction, we observe that a Hamiltonian path P in G, $s, u_1, u_2, \ldots, u_k, t$ has a corresponding Hamiltonian path P' in G', $s^{\text{out}}, u_1^{\text{in}}, u_1^{\text{mid}}, u_1^{\text{out}}, u_2^{\text{in}}, u_2^{\text{mid}}, u_2^{\text{out}}, \ldots, t^{\text{in}}$.
  - For the other direction, we must show that any Hamiltonian path in G' from s^{out} to t^{in} must go from a triple of nodes to a triple of nodes, except for the start and finish, as does the path P' we just described. This would complete the proof because any such path has a corresponding Hamiltonian path in G.
    Start at node s^{out}. Observe that the next node in the path must be u_i^{in} for some i because only those nodes are connected to s^{out}. The next node must be u_i^{mid}, because no other way is available to include u_i^{mid} in the Hamiltonian path. After u_i^{mid} comes u_i^{out} because that is the only other one to which u_i^{mid} is connected. The next node must be u_j^{in} for some j because no other available node is connected to u_i^{out}. The argument then repeats until t^{in} is reached.

The Subset Sum Problem

- In the SubsetSum problem, we are given a collection of numbers $x_1, \ldots, x_k$ together with a target number t, and want to determine whether the collection contains a subcollection that adds up to t.

Theorem

SubsetSum is NP-complete.

- We have already seen that SubsetSum is in NP.
- We prove that all languages in NP are polynomial time reducible to SubsetSum by reducing the NP-complete language 3Sat to it. Given a 3CNF-formula ϕ we construct an instance of the SubsetSum problem that contains a subcollection summing to the target t if and only if ϕ is satisfiable. Call this subcollection T. To achieve this reduction we find structures of the SubsetSum problem that represent variables and clauses.
  - We represent variables by pairs of numbers;
  - Clauses are represented by certain positions in the decimal representations of the numbers.

The Idea of the Reduction

- The idea involves representing a variable x_i in ϕ by two numbers y_i and z_i.

  By proving that either y_i or z_i must be in T for each i, we establish the encoding for the truth value of x_i in the satisfying assignment.
- Each clause position contains a certain value in the target t, which imposes a requirement on the subset T.

  We prove that this requirement is the same as the one in the corresponding clause, i.e., that one of the literals in that clause is assigned TRUE.

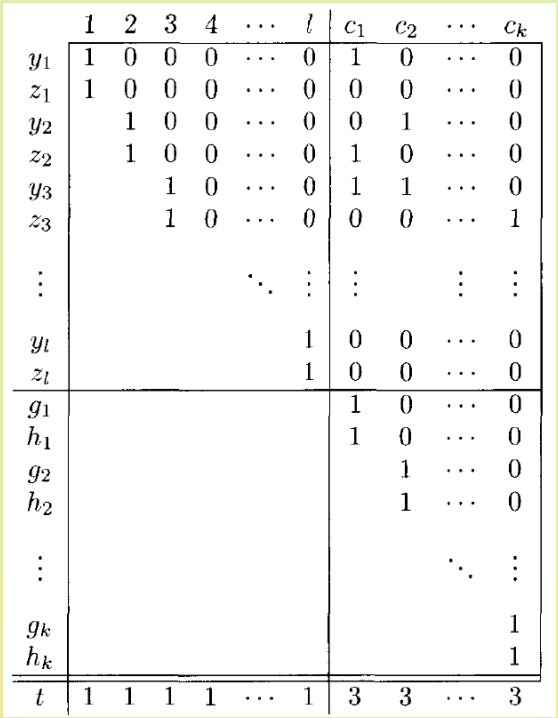
The Reduction I

- Since we know that SubsetSum $\in$ NP, we only show that 3Sat $\leq_{\mathrm{P}}$ SubsetSum.
- Let ϕ be a Boolean formula with variables $x_1, \ldots, x_\ell$ and clauses $c_1, \ldots, c_k$. The reduction converts ϕ to an instance of the SubsetSum problem $\langle S, t \rangle$, wherein the elements of S and the number t are the rows in:

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

The rows above the double line are labeled y_1, z_1, y_2, z_2, …, y_ℓ, z_ℓ and g_1, h_1, g_2, h_2, …, g_k, h_k and comprise the elements of S. The row below the double line is t. Thus, S contains one pair of numbers, y_i, z_i, for each variable x_i in ϕ.

The Reduction II

- 

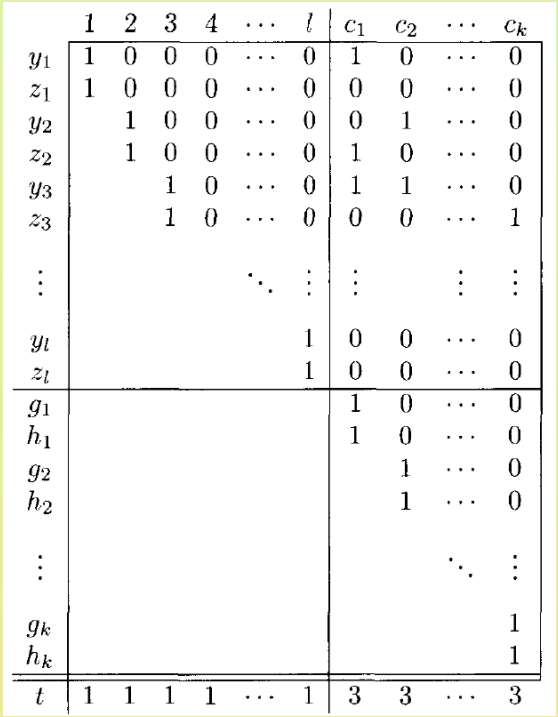
	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

The left part of the decimal representation comprises a 1 followed by $\ell - i$ 0s. The right part contains one digit for each clause, where the j-th digit of y_i is 1 if clause c_j contains literal x_i and the j-th digit of z_i is 1 if clause c_j contains literal $\overline{x_i}$. Digits not specified to be 1 are 0. Additionally, S contains one pair of numbers, g_j, h_j, for each clause c_j, which are equal and consist of a 1 followed by $k - j$ 0s.

Finally, the target number t, the bottom row of the table, consists of ℓ 1s, followed by k 3s.

We prove that ϕ is satisfiable iff some subset of S sums to t.

From Satisfiability to Summability



	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

Suppose that ϕ is satisfiable. We construct a subset of S as follows:

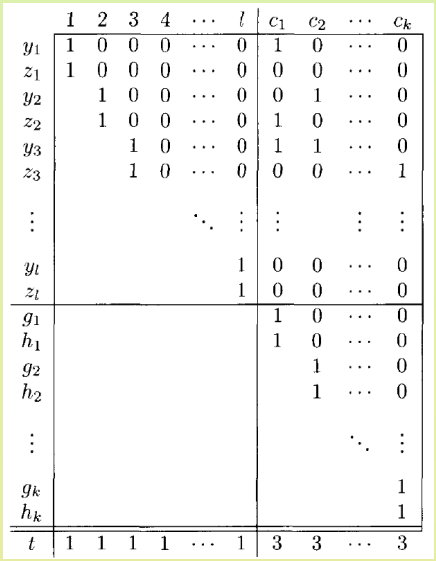
- We select y_i, if x_i is assigned TRUE in the satisfying assignment;
- We select z_i, if x_i is assigned FALSE.

If we add up what we have selected so far, we obtain a 1 in each of the first ℓ digits because we have selected either y_i or z_i for each i.

Furthermore, each of the last k digits is a number between 1 and 3 because each clause is satisfied and so contains between 1 and 3 true literals. Now, we further select enough of the g and h numbers to bring each of the last k digits up to 3, thus hitting the target.

From Summability to Satisfiability

- Suppose that a subset of S sums to t. Note the following:



	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

  - All digits in members of S are 0 or 1.
  - Each column in the table describing S contains at most five 1s. Hence a "carry" into the next column never occurs when a subset of S is added.
  - To get a 1 in each of the first ℓ columns the subset must have either y_i or z_i for each i, but not both.

  For the assignment:

  If the subset contains y_i, we assign x_i TRUE, and, otherwise, FALSE. This assignment must satisfy ϕ because in each of the final k columns the sum is always 3. In column c_j, at most 2 can come from g_j and h_j, so at least 1 must come from some y_i or z_i in the subset.
  - If it is y_i, then x_i appears in c_j and is assigned TRUE, so c_j is satisfied.
  - If it is z_i, then $\overline{x_i}$ is in c_j and x_i is assigned FALSE, so c_j is satisfied.
- The table has size of roughly $(k+\ell)^2$, and each entry can be easily calculated for any ϕ. Total time is $O\left(n^2\right)$ easy stages.