Theory of Computation
Time Complexity

Dimitris Diochnos
School of Computer Science
University of Oklahoma

Outline

Table of Contents

Measuring Complexity

Definition 1

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.
Customarily we use n to represent the length of the input.

Big-O and Small-O Notation

Definition 2

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

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

When $f(n) = O(g(n))$, 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.

Big-O and Small-O Notation

Example 3

Let $f_1(n) = 5n^3 + 2n^2 + 22n + 6$.
Then, selecting the highest order term $5n^3$ and disregarding its coefficient 5 gives $f_1(n) = O(n^3)$.

- We can verify this is indeed the case (i.e., we satisfy Definition 2) by letting $c = 6$ and $n_0 = 10$.
  Then, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$, for every $n \geq 10$.
- In addition, $f_1(n) = O(n^4)$ because n^4 is larger than n^3 and so n^4 is still an asymptotic upper bound on f_1.
- However, $f_1(n)$ is not $O(n^2)$. Regardless of the values we assign to c and n_0, Definition 2 can not be satisfied in this case.

Big-O and Small-O Notation

Remarks on Big-O:

- For logarithms, we do not care about the base. $f(n) = O(\log n)$
  Recall: $\log_b n = \log_2 n / \log_2 b$
- $f(n) = 2^{O(n)}$ implies $f(n) \leq 2^{c \cdot n}$ for some constant c.
- $f(n) = 2^{O(\log n)}$ implies $f(n) \leq 2^{c \cdot \log n}$,
  or in other words $f(n) \leq n^c$ for some constant c.
- $f(n) = n^{O(1)}$ implies $f(n) \leq n^c$ for some constant c.
  So same bound as above; just written differently
- **Polynomial bounds**: Bounds of the form n^c for $c > 0$.
- **Exponential bounds**: Bounds of the form $2^{(n^\delta)}$ for some $\delta > 0$.

Small-O

Definition 4

Let f and g be functions such that $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = o(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

In other words, $f(n) = o(g(n))$ means that, for any real number $c > 0$, a number n_0 exists, where $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

Small-O

Example 5

It is easy to check the following:

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. $n^2 = o(n^3)$.

However, $f(n)$ is never $o(f(n))$.

Analyzing Algorithms

Definition 6

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

Example 7

Consider the language $A = \{0^k 1^k \mid k \geq 0\}$.

One solution:
Scan repeatedly the input string and cross off a 1 (one) for every 0 (zero) read. $\Rightarrow$
we need $\frac{n}{2}$ such scans (where n is the length of the input) and in each scan we move the head above n cells of the tape.
So, roughly we need to move the tape $O(n^2)$ times $\Rightarrow A \in \text{TIME}(n^2)$.

Analyzing Algorithms

Example 7 (cont.)

Another solution:
Scan repeatedly the input string and cross off half of the zeros and half of the ones that are still "alive" $\Rightarrow$
Each scan eliminates about half of the "alive" 0s and 1s from the previous scan $\Rightarrow$
we need roughly $\log_2 n$ such scans to eliminate all the 0s and 1s, and each scan takes time about n for moving the head from the leftmost cell to the first blank (end of input) $\Rightarrow$
So, roughly we need to move the tape $O(n \log n)$ times $\Rightarrow A \in \text{TIME}(n \log n)$.

Third solution:
Copy the 0s to a second tape and then move both heads (for 0s and 1s) simultaneously so that we can decide the input string $\Rightarrow$
Need only one pass on the input $\Rightarrow A$ decided in $O(n)$ time.

Analyzing Algorithms

Interesting observation

Any language decided in $o(n \log n)$ time on a **single-tape** Turing machine, is **regular**.

Complexity depends of model of computation.

Complexity Relationships Among Models

Theorem 8

Let $t(n)$ be a function, where $t(n) \geq n$.
Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

Proof (to be cont.)

Let M be a k-tape TM that runs in $t(n)$ time.
Simulation: Store all k tapes consecutively in a TM.

- one pass determines the symbols for each head in M,
- a second pass updates the contents and moves the simulated heads (we may need to shift everything one cell to the right).

Complexity Relationships Among Models

Proof (cont.)

The input length of each tape is at most $t(n)$.
$\Rightarrow$ For k tapes $\leq k \cdot t(n)$
$\Rightarrow$ One pass in S may read up to $O(t(n))$ cells
$\Rightarrow$ Two passes + potentially up to k shifts of the contents one cell to the right
$\Rightarrow O(t(n)) + O(t(n)) + O(t(n)) = O(t(n))$.

So overall for S:

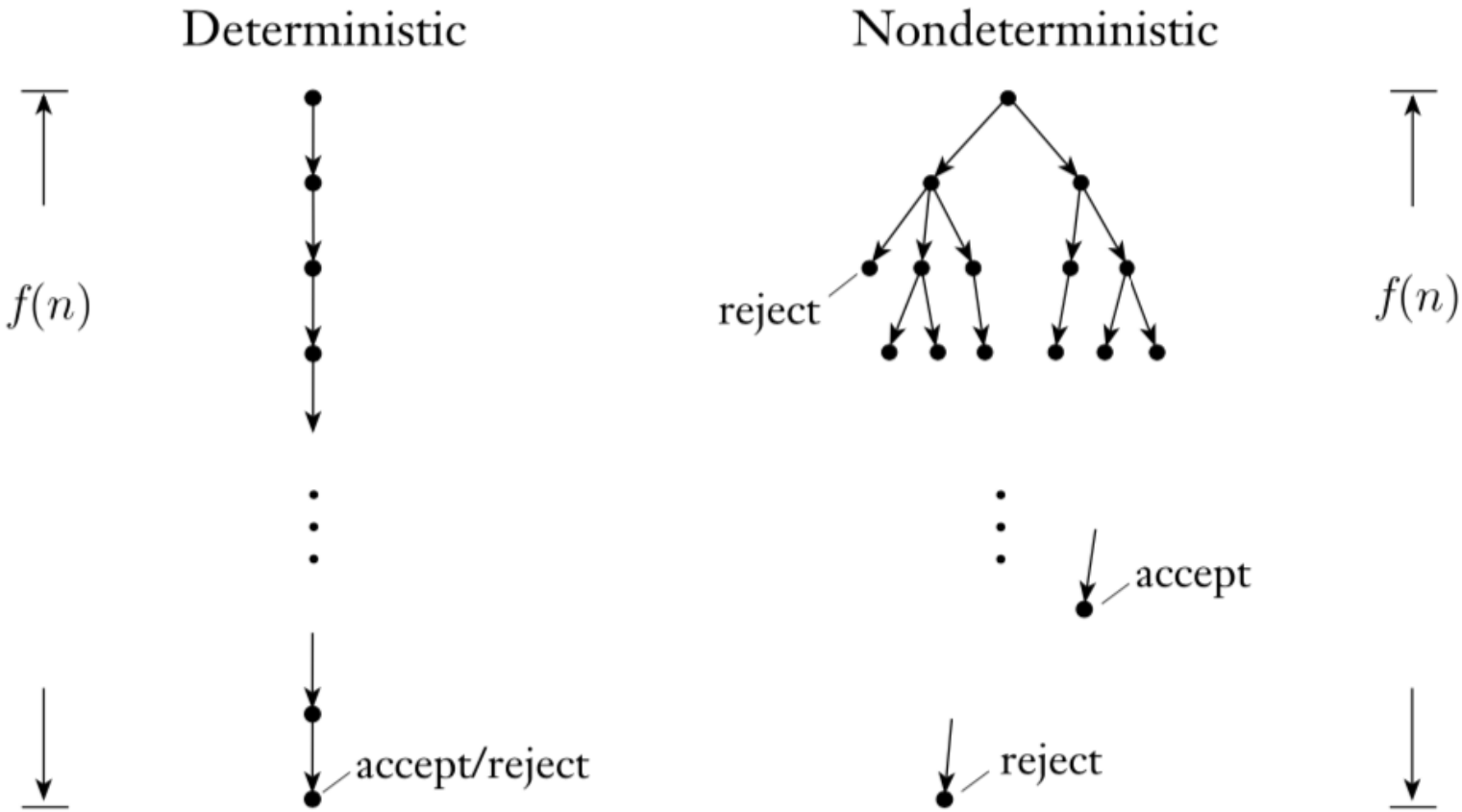
- Initialization: $O(t(n))$
- Simulation: Up to $t(n)$ steps and need $O(t(n))$ time for each step simulated
  $\Rightarrow O(t^2(n))$ time.

Nondeterministic time

Definition 9

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.

Deterministic vs Nondeterministic



Measuring deterministic and nondeterministic time

Deterministic vs Nondeterministic

Theorem 10

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

Proof.

Construct a deterministic 3-tape TM D that simulates N (as in the proof of Theorem 3.16).
On an input of length n, every branch of N's nondeterministic computation tree has length at most $t(n)$.
Since every node can have at most b children $\Rightarrow \leq b^{t(n)}$ leaves.
$\Rightarrow$ Total number of tree nodes $\leq 2 \cdot$ maximum number of leaves $\leq 2 \cdot b^{t(n)}$.
$\Rightarrow$ Traveling down from root to node: time $\leq O(t(n))$.
$\Rightarrow$ Running time for D is $O(t(n) \cdot b^{t(n)})$ which is $O(2^{t(n)} \cdot 2^{\log_2 b \cdot t(n)})$.

(Also true for single-tape TM in the end...) □

Table of Contents

The Class P

Definition 11

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$P = \cup_k TIME(n^k).$$

P is important because:

1. All reasonable models of computation (models that try to approximate the behavior of actual computers) are polynomially equivalent to a deterministic single-tape TM. So P is invariant for all such models.
2. P roughly corresponds to problems that are realistically solvable on a computer.

The Class P

Key Observations:

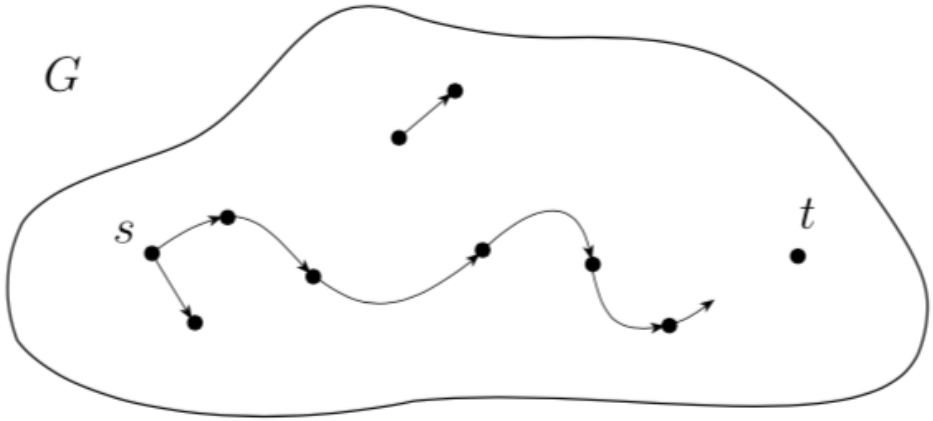
- Composition of polynomials is a polynomial
  (polynomially many stages of an algorithm, each of which can be done in polynomial time)
- Assume "reasonable" representations, e.g.,
  - numbers such as 9 are not written down in unary as 111111111, but in some base ≥ 2, e.g., in binary 1001 (which takes exponentially less space)
  - graphs: adjacency matrix, or list of nodes and edges $\Rightarrow$ ok to compute running time in terms of number of nodes n, since number of edges are at most $O(n^2)$.
- In any case, for "reasonable" inputs of length n, when working in P we want algorithms that decide the instances of the problem in time $O(n^k)$.

Examples of Problems in P

Theorem 12

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$$

$$PATH \in P$$



The PATH problem: Is there a path from s to t?

Examples of Problems in P

Proof (to be cont.)

- Note that enumerating all paths is inefficient
  e.g., if G has m nodes, the paths of length $m-1$ (w/o repeating nodes) are: $s \times \underbrace{m-1 \times m-2 \ldots \times 2 \times 1}_{(m-1)! \text{ ways}}$
  Note that $(m-1)! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot \lfloor \frac{m-1}{2} \rfloor \cdot \lceil \frac{m-1}{2} \rceil \cdot \ldots \cdot (m-1)$
  ($\lfloor \frac{m-1}{2} \rfloor$ and $\lceil \frac{m-1}{2} \rceil$ are different when m is even), so:
  $(m-1)! \geq (m-1) \cdot (m-2) \cdot \ldots \cdot (\lfloor \frac{m-1}{2} \rfloor) \cdot 1 \geq (m-1) \cdot (m-2) \cdot \ldots \cdot (\frac{m-1}{2})$
  $\Rightarrow (m-1)! \geq (\frac{m-1}{2})^{(\frac{m-1}{2})}$, so exponential in m.
  (On another note, for $m \geq 2$: $m! \geq \underbrace{2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2}_{m \text{ times}} = 2^m$)
- Another approach: Propagate a "label" iteratively, indicating which nodes can be reached from s. So, in particular do a BFS if you want.

Examples of Problems in P

Proof (cont.)

- Another idea is:
  $M =$ "On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:
  1. Place a label on node s.
  2. Repeat until no additional nodes are labeled:
  3. Scan all the edges of G. If an edge (a, b) is found going from a labeled node a to an unlabeled node b, label node b.
  4. If t is labeled, **accept**. Otherwise, **reject**."

  Time Complexity: Stages 1 and 4 are executed only once.
  Stage 3 runs at most m times since it introduces one node each time (except the last one).
  $\Rightarrow$ total number of stages is at most $1 + 1 + m = \text{poly}(\langle G, s, t \rangle)$.
  Each stage can also be implemented in polynomial time
  $\Rightarrow$ Overall, PATH $\in$ TIME$(\text{poly}(\langle G, s, t \rangle))$.

Examples of Problems in P

Theorem 13

$$RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\} \in P.$$

Proof (to be cont.)

"Reasonable" encoding of integers x and y (say, binary)

$$\Rightarrow \text{We need} \begin{cases} \approx \log x \text{ bits to represent } x \\ \approx \log y \text{ bits to represent } y \end{cases}$$

- The idea of checking each integer in the range $\{2, 3, \ldots, \min\{2, 3\}\}$ and see if it divides both x and y and stopping the first time it works, in the worst case may need to check about $y = \min\{x, y\}$ numbers. But this means we need to check **exponentially many** integers compared to the size of the input (about $\log x$ bits)
  $\Rightarrow$ We need a different approach.

Examples of Problems in P

Proof (cont.)

- Use the Euclidean algorithm.
  $E =$ "On input $\langle x, y \rangle$, where x and y are natural numbers in binary:
  1. Repeat until $y = 0$:
  2. Assign $x \leftarrow x \mod y$.
  3. Exchange x and y.
  4. Output x."

  Solve the problem with the following algorithm:
  $R =$ "On input $\langle x, y \rangle$, where $x, y \in \mathbb{N}$:
  1. Run E on $\langle x, y \rangle$.
  2. If the result is 1, **accept**. Otherwise, **reject**."

  Time complexity: Stages 2 and 3 in E are executed $\approx \log_2 x$ or $\log_2 y$ times (whichever is smaller).
  $\Rightarrow$ That's poly-proportional to the size of input $\Rightarrow \in P$ □

Examples of Problems in P

Key observation:

- $y \leq \frac{x}{2} \Rightarrow$
  $\Rightarrow x \mod y \in \{0, 1, 2, \ldots, y-1\}$
  $\Rightarrow x \mod y < y \leq \frac{x}{2}$
  $\Rightarrow x \leftarrow x \mod y$
  drops x by at least half
- $\frac{x}{2} < y < x \Rightarrow$
  $\Rightarrow x \mod y = x - y$
  $\Rightarrow x \mod y < \frac{x}{2}$
  $\Rightarrow x \leftarrow x \mod y$
  drops x by at least half

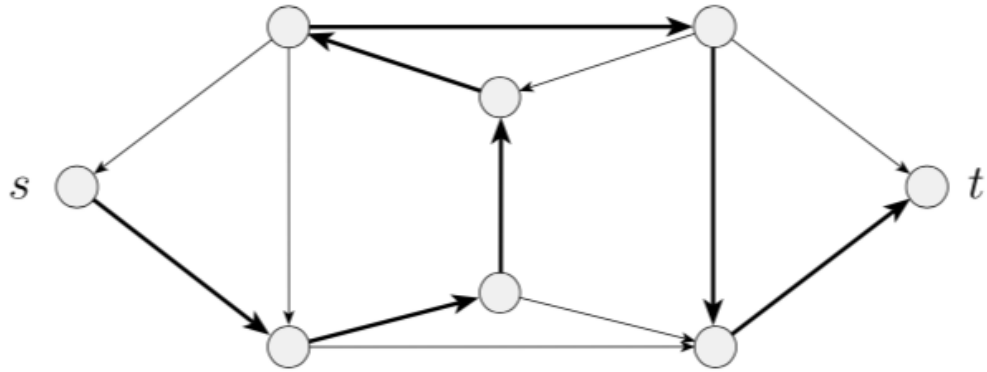
Table of Contents

Hamiltonian path

Hamiltonian path

A Hamiltonian path in a directed graph G is a directed path that goes through each node exactly once.

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



A Hamiltonian path goes through every node exactly once

Hamiltonian path

- easy for an exponential time solution
  $\Rightarrow$ Use PATH algorithm with the additional constraint to check if the path found is Hamiltonian.
- No one knows whether HAMPATH is solvable in polynomial time.
- **However**, HAMPATH has **polynomial verifiability**
  (We can easily, i.e., in polynomial time, convince someone else when HAMPATH has a solution, by simply providing such a solution.)
- Some problems may not be polynomially verifiable; e.g., $\overline{\text{HAMPATH}}$.
  We do not know of a way to convince someone that a graph does not have a Hamiltonian path, other than using the same exponential time algorithm in the first place.

Class NP

Definition 14

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

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

We measure the time of a verifier only in terms of the length of w, so 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.

Class NP

Definition 15

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

NP stands for **nondeterministic polynomial time** and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines.

Example

Solve HAMPATH with an NTM

N_1 = "On input $\langle G, s, t \rangle$, where 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 to be 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 between 1 and $m - 1$, 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**."

Class NP

Theorem 16

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

Proof (to be cont'd).

($\Rightarrow$)

$A \in NP \Rightarrow A$ has a polynomial time verifier V; a TM that runs in time n^k.

$N =$ "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**."

Class NP

Proof (cont.)

($\Leftarrow$)

A decided by some NTM N. Then we can construct a polynomial time verifier V:

$V =$ "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 make at each step.
2. If this branch of N's computation accepts, **accept**; otherwise, **reject**."

Class NP

Definition 17

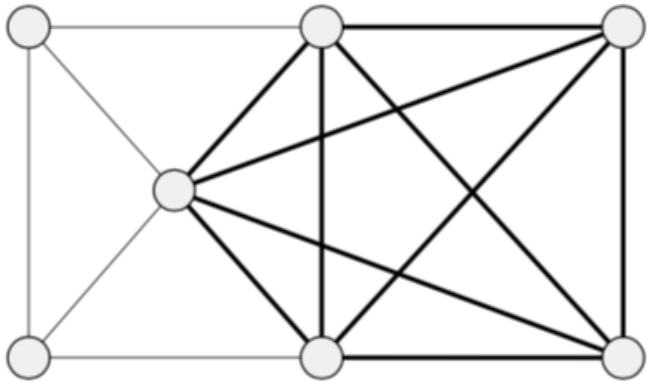
$$NTIME(t(n)) = \{L | L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

Corollary 18

$$NP = \cup_k NTIME(n^k).$$

Examples of problems in NP: CLIQUE

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.



A graph with a 5-clique

Examples of problems in NP: CLIQUE

Theorem 19

CLIQUE is in NP.

Proof.

Construct 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**."

□

Examples of problems in NP: CLIQUE

Theorem 20

CLIQUE is in NP.

Alternative proof.

Construct a NTM solving the problem.
$N =$ "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**."

Examples of problems in NP: SUBSET-SUM

Define:

$$SUBSET - SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \sum y_i = t\}.$$

For example, $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ SUBSET-SUM because $4 + 21 = 25$.

Note that $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_l\}$ are considered to be **multisets** and so allow repetition of elements.

Examples of problems in NP: SUBSET-SUM

Theorem 21

SUBSET-SUM is in NP.

Proof.

Here is a verifier:

$V =$ "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**."

□

Examples of problems in NP: SUBSET-SUM

Theorem 22

SUBSET-SUM is in NP.

Alternative proof.

We construct a NTM:
$N =$ "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 P versus NP Question

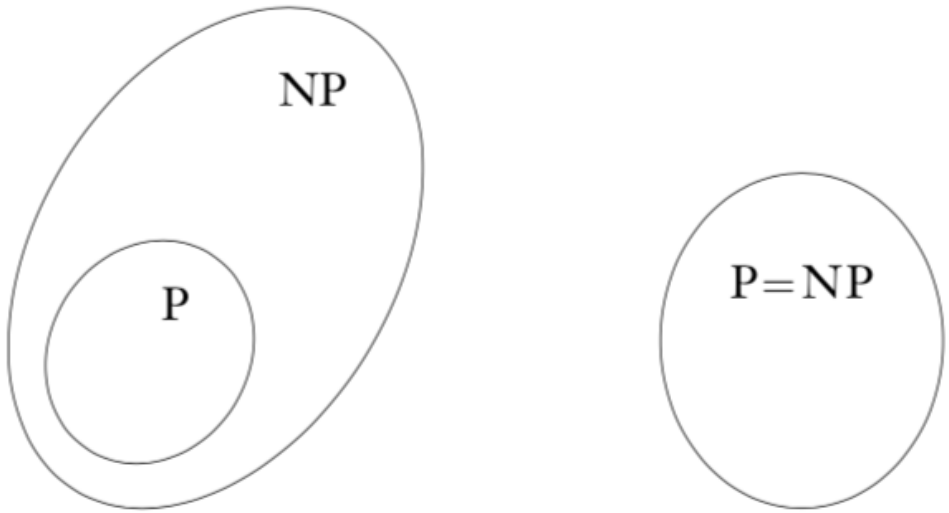
P = the class of languages for which membership can be **decided** quickly.
NP = the class of languages for which membership can be **verified** quickly.

"quickly": in polynomial time

The *P* versus *NP* Question

It appears that polynomial time verifiability is more powerful compared to polynomial time decidability.

However, we are unable to prove the existence of a single language in *NP* that is not in *P*.



One of these two possibilities is correct

The P versus NP Question

The best method currently known for solving languages in NP deterministically uses exponential time.
So we can prove that:

$$NP \subseteq EXPTIME = \cup_k TIME(2^{n^k})$$

but we don't know whether *NP* is contained in a smaller deterministic time complexity class.

Table of Contents

NP-completeness

1970s: Stephen Cook and Leonid Levin

- 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.

Importance of NP-completeness

Theory:

- Want to prove $P \neq NP$?
  Show that an NP-complete problem requires more than polynomial time.
- Want to prove $P = NP$?
  Show that an NP-complete problem has a polynomial time algorithm.

Practice:
Proving that a problem is NP-complete is strong evidence that the problem should not be solvable (in the general case) in polynomial time.

The Satisfiability problem

The **Satisfiability problem**:

$$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}.$$

Theorem 23 (Cook-Levin Theoorem)

$SAT \in P$ *iff* $P = NP$.

Polynomial Time Reducibility

Definition 24

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.

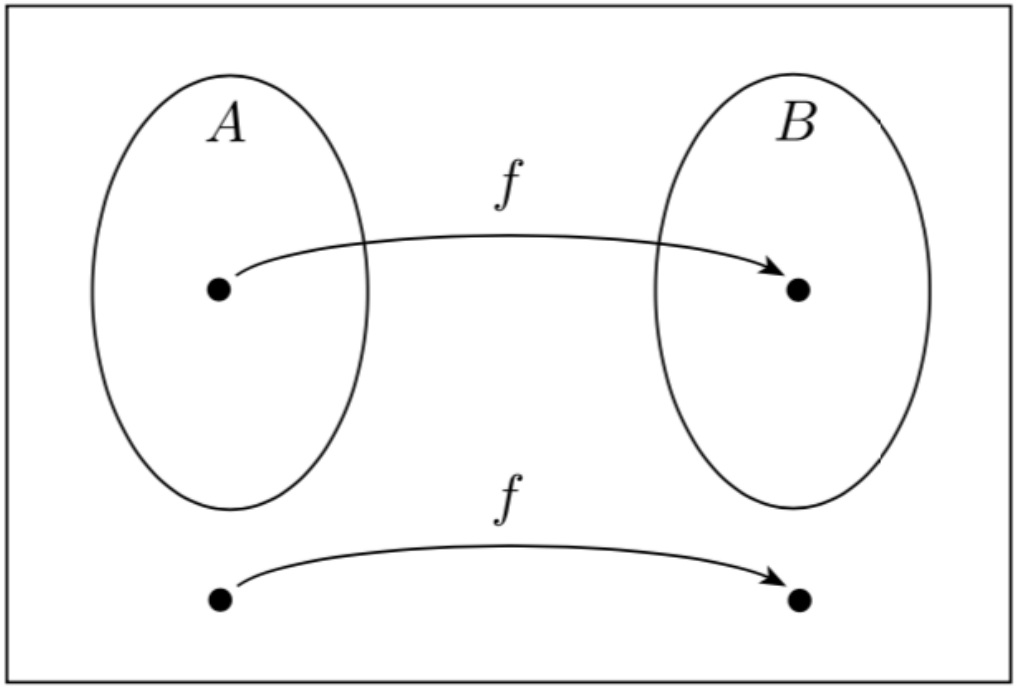
Definition 25

Language A is **polynomial time mapping reducible**, or simply **polynomial time reducible**, to language B, written $A \leq_P B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists, where for every w,

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

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

Polynomial Time Reducibility



Polynomial time function f reducing A to B

(Similar to mapping reductions, but now the conversion needs to be done efficiently.)

Polynomial Time Reducibility

Theorem 26

If $A \leq_P B$ and $B \in P$, then $A \in P$.

Proof (to be cont.)

Let M be a polynomial time algorithm deciding B.
Let f be a polynomial time reduction from A to B.
Then, a polynomial time algorithm N deciding A is the following.

$N =$ "On input w:

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

Polynomial Time Reducibility

Proof (cont'd).

We have $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B.
Thus, M accepts $f(w)$ whenever $w \in A$.
Moreover, N runs in polynomial time, since each stage runs in polynomial time.
Note that stage 2 runs in polynomial time, because the composition of two polynomials is a polynomial. □

Notation

- A **literal** is a Boolean variable or a negated Boolean variable; e.g., x or $\overline{x}$.
- A **clause** is several literals connected with $\vee$s, e.g., $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$.
- 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 3 literals, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$. (Repetition of literals is allowed in a clause.)
- $3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$.

3SAT

Theorem 27

3SAT is polynomial time reducible to CLIQUE.

Proof idea.

- The polynomial time reduction f will convert 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.

□

3SAT

Theorem 28

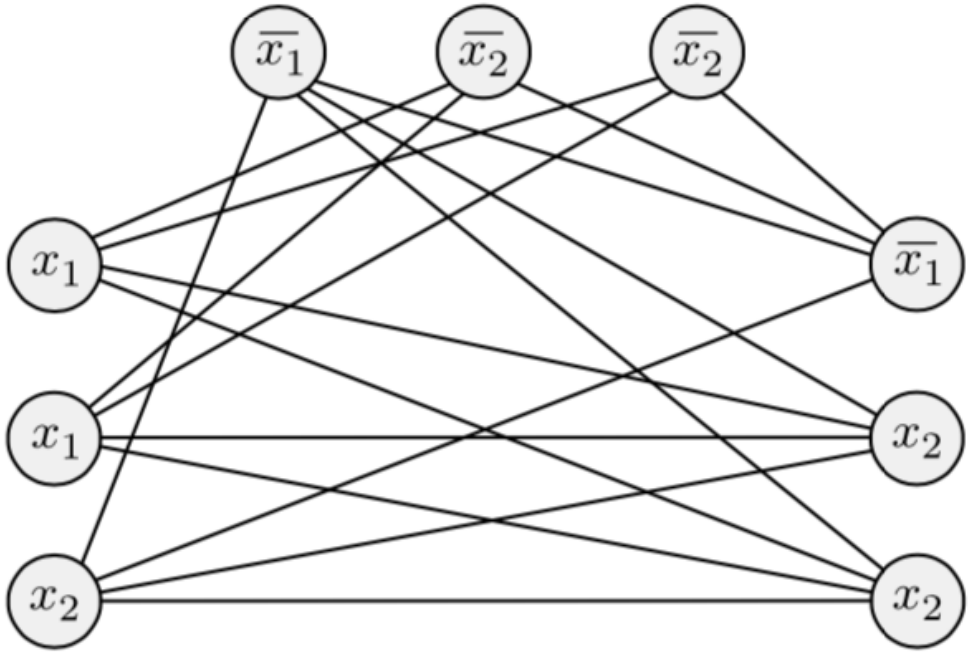
3SAT is polynomial time reducible to CLIQUE.

Proof (to be cont.)

Let ϕ be a formula with k clauses such as:
$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$.
The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows.

- Organize the nodes in G into k groups of three nodes each, called **triples**, $t_1, \ldots, t_k$. Each triple corresponds to a clause in ϕ.
  Each node in a triple corresponds to a literal in the associated clause.
  Label each node with its corresponding literal in ϕ.
- The edges of G connect all but two types of pairs of nodes in G:
  - nodes in the same triple
  - nodes with contradictory labels (e.g., x_2 and $\overline{x_2}$)

3SAT



The graph that the reduction produces from

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

Why the construction works?

Proof (to be cont.)

- Let ϕ have a satisfying assignment, say $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$.
  In each triple of G, we select one node corresponding to a true literal in the satisfying assignment.
  If more than one literal is true in a particular clause, we choose one of them literal arbitrarily.
  These nodes now form a k-clique.
  - We selected k nodes.
  - No pair of these nodes should not be joined by an edge.

  $\Rightarrow$ G contains a k-clique.

Why the construction works?

Proof (cont').

- Let G have a k-clique.
  Then no two of the clique's nodes occur in the same triple.
  Therefore, each of the k triples contains exactly one of the k clique nodes.
  Now assign truth values to the variables of ϕ so that each literal labeling a clique node is made true.
  Doing so is always possible, because two nodes labeled in a contradictory way are not connected by an edge (so both can not be simultaneously in the clique).
  $\Rightarrow$ This assignment satisfies ϕ because each triple contains a clique node $\Rightarrow$ ϕ is satisfiable.

□

In other words, due to Theorems 26 and 28, if CLIQUE is solvable in polynomial time, so is 3SAT.

NP-completeness

Definition 29

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

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

Theorem 30

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

Proof.

Immediate from the definition of polynomial time reducibility. □

NP-completeness

Theorem 31

If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

Proof.

We know that $C \in NP$. So we need to show that every $A \in NP$ is polynomial time reducible to C.
Since B is NP-complete, every problem in NP is polynomial time reducible to B. Now B is polynomial time reducible to C.
Polynomial time reductions compose $\Rightarrow$ if A is polynomial time reducible to B and B is polynomial time reducible to C, then A is polynomial time reducible to C.
So every language in NP is polynomial time reducible to C. □

Theorem 32 (Restate Theorem 23 by Cook and Levin)

SAT is NP-complete.