Complexity of problems

BEKKOUCHE Mohammed

National Higher School of Computer Science, Sidi Bel Abbès

2nd Year Second Cycle

Specialization: Artificial Intelligence and Data Science (IASD)

m.bekkouche@esi-sba.dz

September 15, 2025

Introduction

- Some combinatorial optimization problems have polynomial algorithms, while others still do not have any[1]

Is there truly a class of combinatorial problems for which polynomial algorithms will never be found, or do difficult problems indeed have such algorithms that have not yet been discovered?

- For a long time, there has been a conjecture about the existence of a class of inherently difficult problems.
- Because several difficult problems (such as the traveling salesman problem) have resisted research efforts for over 50 years: despite these efforts, no polynomial algorithm has been discovered.

[1] The complexity of a problem represents the complexity of the best program to solve it.

1. A combinatorial problem consists in, given a finite collection of objects and a set of constraints, finding an object of the collection that satisfies all constraints (and possibly that optimizes some objective function).
2. Solving a combinatorial optimization problem involves finding the optimum of a function among a finite number of often very large choices. It generally involves maximizing (maximization problem) or minimizing (minimization problem) an objective function under certain constraints.
3. A combinatorial problem is one whose solution involves a vast number of combinations to explore.
4. An example of a combinatorial optimization problem is the traveling salesman problem, which involves determining, given a list of cities and the distances between all pairs of cities, the shortest route that visits each city exactly once.

Introduction

- Complexity theory was developed in the 1970s to address this question.
- The main result is that all difficult problems are interconnected.
  $\Longrightarrow$ The discovery of a polynomial algorithm for a single difficult problem would lead to deducing polynomial algorithms for all others.
- Complexity theory deals only with existence problems[2], with yes/no answers.
- This is not problematic for problems involving computing solutions or optimization.
  - An efficient algorithm for an existence problem can be used to efficiently solve its computing solution problem.
  - Similarly, an efficient algorithm for an existence problem can be used to efficiently solve its associated optimization problem through a simple dichotomy on the objective function values.

[2]These are also called decision problems

1. A decision problem is a mathematical question to which the answer is either "yes" or "no".
2. Computing solution problems: if a formula is satisfiable, then compute a satisfying valuation. If a graph is 3-colorable, then compute a 3-coloring.
3. An optimization problem is one that aims to find the best solution among all possible solutions.
4. Optimization problems can be divided into two categories depending on whether the variables are continuous or discrete. An optimization problem with discrete variables is known as a combinatorial optimization problem.
5. For each combinatorial optimization problem, there is a corresponding decision problem that asks whether there is a possible solution for certain measurements.
6. For each computing solution problem, there is a corresponding decision problem.

Introduction

- Let's consider the SAT problem and assume it has a polynomial algorithm A to solve it. Then, we can construct the following algorithm, which calculates a satisfying valuation σ (if one exists) for a given formula φ:

Algorithm 1: SolSAT(φ)

Data: A formula $\varphi(x_1, ..., x_n)$
Result: A satisfying valuation for φ (if one exists)

if $A(\varphi)$ returns "No" **then**
  return "Unsatisfiable"
end
for $i \leftarrow 1$ **to** n **do**
  if $A(\varphi(b_1, ..., b_{i-1}, 1, x_{i+1}, ..., x_n))$ returns "Yes" **then**
    $b_i \leftarrow 1$;
  else
    $b_i \leftarrow 0$;
  end
end
return $(b_1, ..., b_n)$;

1. For example, to transform "finding the smallest non-trivial divisor of N", we would add an input k to get the question "is there a non-trivial divisor of N that is less than k?".
   If we can answer this question, then we can quickly find the smallest non-trivial divisor of N: we just need to perform a binary search by testing if there is a divisor $\leq N/2$, then if yes, if there is a divisor $\leq N/4$, or if not, if there is a divisor $\leq 3N/4$, and so on.
2. Solving the problem of : finding the smallest non-trivial divisor of 25. The decision corresponding problem instance is : Is there a non-trivial divisor of 25 that is less than k ?

Easy Problems

Graph Exploration:

Data: A directed graph $G = (V, E)$, two vertices s and t in V.
Question: Is there a path from s to t?

Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms with $O(|V| + |E|)$ complexity

Shortest Path:

Data: $G = (V, E, \nu)$ a valued directed graph with an application $\nu : E \to \mathbb{R}$, two vertices s and t in V
Task: Find a shortest path from s to t

Bellman's algorithm with $O(|V||E|)$ complexity. Dijkstra's algorithm with $O(|E| + |V| \log |V|)$ complexity

Max Flow:

Data: $G = (V, E, \nu)$ a valued directed graph with a capacity function $\nu : V \times V \to \mathbb{N}$ such that $\nu(x, y) = 0$ if $(x, y) \notin E$, two vertices s and t in V
Task: Maximize the flow rate that can go through the network between s and t.

Ford-Fulkerson algorithm with $O(|V||E|^2)$ complexity

1. A problem consists of two elements: an input (or instance) and a question or task to perform. Here are two examples that can be reformulated as follows:
   PRIMALITY:
   Data: An integer N.
   Question: Is N prime?
   SORTING:
   Data: A list of integers l.
   Task: Sort l in ascending order.
2. **Note:** Easy means that polynomial algorithms are known to solve these problems, i.e., the complexity of these algorithms is of the form $O(n^k)$ with $k \in \mathbb{N}$.
   An algorithm with $O(n^{100})$ complexity is not efficient. However, the discovered polynomial algorithms rarely go beyond degree 6.
3. A function $f : E \to F$ is an application if $Dom(f) = E$.

Easy Problems

Minimum Weight Spanning Tree:

Data: $G = (V, E, \nu)$ a valued simple undirected graph with an application $\nu : E \to \mathbb{R}$

Task: Find a minimum weight spanning tree.

Prim's algorithm with $O(|V|^2)$ complexity. Kruskal's algorithm with $O(|E| \log |E|)$ complexity. If G is directed, an arborescence could be computed in $O(|V||E|)$ using Edmonds' algorithm

Matching:

Data: A simple undirected graph $G = (V, E)$

Task: Find a maximum cardinality matching.

Edmonds' algorithm with $O(|V|^4)$ complexity

Eulerian Cycles:

Data: An undirected graph $G = (V, E)$

Question: Does an Eulerian cycle exist in G?

The existence problem is solvable in $O(|V|^2)$

1. A "chain" in a graph is a succession of edges that make it possible to link two vertices in the graph.
2. A chain that touches each vertex one time at most is called "elementary."
3. A chain that uses each edge one time at most is called "simple."
4. A simple graph is a graph without loops or multiple edges. There are only edges between distinct vertices, and between two vertices there is at most one edge.
5. A matching of G is a subset of edges such that no two of them have a common vertex.
6. An Eulerian cycle is a path that visits each edge of the graph exactly once. A Chinese postman tour visits each edge at least once.

Easy Problems

Planarity Testing:

Data: A simple graph $G = (V, E)$

Question: Is G planar, i.e., can it be drawn in the plane without edge crossings?

Algorithm with $O(|E|)$ complexity by Hopcroft and Tarjan

Bipartiteness Testing:

Data: A simple graph $G = (V, E)$

Question: Is G bipartite?

It can be proven that a graph is bipartite if and only if it contains no odd cycle.
Algorithm with $O(|E|)$ complexity

Search in n elements:

This is the search for an element in an array of n elements

Sorting N numbers:

Solvable using heap sort algorithm in $O(n \log n)$

1. A graph is bipartite if its vertex set can be partitioned into two disjoint subsets U and U' such that each edge has one endpoint in U and the other in U'.
2. A cycle containing an odd (respectively even) number of edges is called an odd (respectively even) cycle.

Hard Problems

Maximum Stable Set:

Data: A simple graph $G = (V, E)$

Task: Find a maximum cardinality stable set, i.e., a largest possible subset of graph vertices where no two elements are adjacent.

Minimum Vertex Cover:

Data: A simple graph $G = (V, E)$

Task: Find a minimum set of vertices to cover all edges.

Maximum Clique:

Data: A simple graph $G = (V, E)$

Task: Find a clique (a subset of vertices of graph G that are all pairwise adjacent, also known as a complete subgraph) with the maximum number of vertices.

Minimum Coloring:

Data: A simple graph $G = (V, E)$

Task: Determine the chromatic number of G.

1. A subset of vertices S is a stable set - also known as an independent set - if there is no edge between any two vertices in S.
2. A subset of vertices T is a vertex cover (or transversal) if every edge of G has at least one endpoint in T.
3. A subset of vertices Q is a clique if every pair of vertices in Q is connected by an edge. Q thus induces a complete graph.
4. G is k-colorable if its vertices can be colored with k distinct colors, such that no two adjacent vertices have the same color. The smallest value of k for which G is k-colorable is the chromatic number of G.

Hard Problems

Hamiltonian Problem:

Data: A simple graph $G = (V, E)$
A Hamiltonian cycle is a Hamiltonian path that forms a cycle. A Hamiltonian path in an undirected or directed graph is a path that passes through all vertices exactly once.

Question: Does G have a Hamiltonian cycle?

Traveling Salesman Problem (TSP):

Data: $G = (V, E, \nu)$ a complete directed weighted graph with an application $\nu : E \to \mathbb{R}$

Task: Find a Hamiltonian cycle, with minimal cost.

Boolean Satisfiability Problem (SAT):

Data: A propositional formula

Question: Determine whether there exists an assignment of propositional variables that makes all clauses true.

1. A cycle is a sequence of consecutive edges, a simple path (not containing the same edge multiple times), with identical endpoints.

Hard Problems

Integer Knapsack Problem:

Data: A set of items, each with a weight and a value.

Task: Determine the quantity of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is maximized.

0-1 Knapsack Problem:

$$\begin{cases} Maximize \sum_{i=1}^{n} v_i x_i \\ \sum_{i=1}^{n} w_i x_i \leq W \\ x_i \in \{0,1\} \end{cases}$$

Bounded Knapsack Problem:

$$\begin{cases} Maximize \sum_{i=1}^{n} v_i x_i \\ \sum_{i=1}^{n} w_i x_i \leq W \\ x_i \in \{0,1,2,...,c\} \end{cases}$$

Unbounded Knapsack Problem:

$$\begin{cases} Maximize \sum_{i=1}^{n} v_i x_i \\ \sum_{i=1}^{n} w_i x_i \leq W \\ x_i \in \mathbb{N} \end{cases}$$

x_i represents the number of instances of the item to include in the knapsack.

Bin Packing:

Data: n items with weights a_i and an unlimited number of bins with capacity b

Task: Distribute the items into a minimal number of bins.

1. **The Knapsack Problem** is to maximize the sum of the values of the items in the knapsack such that the sum of the weights is less than or equal to the knapsack's capacity.

2. Example (Bin Packing Problem)

   Input: Weights $= \{4, 8, 1, 4, 2, 1\}$
   Bin capacity $b = 10$.

   Output: 2.
   We need at least 2 bins to accommodate all the items.
   The first bin contains $\{4, 4, 2\}$ and the second bin $\{8, 1, 1\}$.

   Input: Weights $= \{9, 8, 2, 2, 5, 4\}$
   Bin capacity $b = 10$.

   Output: 4.
   We need at least 4 bins to accommodate all the items.

Undecidable Problems

An undecidable problem is a decision problem for which it has been proven impossible to construct an algorithm that always leads to a correct answer of yes or no.

Halting Problem:

Data: A computer program and input data for that program.
Question: Determine whether the program will eventually stop or continue running forever.

Hilbert's Tenth Problem:

Data: A diophantine equation with any number of unknowns and rational integer coefficients.
Question: Determine if the equation is solvable (has roots) in rational integers.

Post's Correspondence Problem (1946):

Data: n pairs of words $(u_1, v_1), ..., (u_n, v_n)$ over an alphabet Σ.
Question: Is there a finite sequence $i_1, ..., i_k$ $(k > 0)$ such that
$u_{i_1} ... u_{i_k} = v_{i_1} ... v_{i_k}$
Note: The index sequences are the same.

1. Alan Turing proved in 1936 that there is no algorithm to solve the halting problem, making it undecidable.
2. A diophantine equation in mathematics is a polynomial equation with one or more unknowns that are sought to be found among the integers or rational numbers, with coefficients also being integers or rational numbers.
3. **Post's Correspondence Problem (1946):** The pairs (u_i, v_i) can be seen as dominos.
   Example:

a	aa	ba	bab
ab	a	aa	abba

   A solution: a.bab.ba.aa.aa = ab.abba.aa.a.a
   Index sequence: 1, 4, 3, 2, 2.

Classes P and NP

- Given a function $f : \mathbb{N} \longrightarrow \mathbb{N}$, a problem is said to belong to the class $DTIME(f)$ if there exists a deterministic machine that, on any input of length n, solves the problem in $O(f(n))$ computational steps.

The set of decision problems that have polynomial-time algorithms forms the class P. We can define $P = \bigcup_{k \geq 0} DTIME(n \longrightarrow n^k)$.

The class NP[a] consists of decision problems for which a proposed solution of "yes" can be verified in polynomial time. We also define the class $NP = \bigcup_{k \geq 0} NTIME(n \longrightarrow n^k)$, where N stands for non-deterministic[b].

[a] NP does NOT mean Non-Polynomial; it stands for Non-Deterministic Polynomial!

[b] The non-deterministic computation model is enriched by an "choice" instruction, where there exists an immediate way to lead to the solution.

- Problems not in NP exist, but for the most part, they are of theoretical interest only. NP includes P.

1. A deterministic machine is the formal model of a machine as we know them (our computers are deterministic machines).
2. A non-deterministic machine is a purely theoretical variant of deterministic machines: we can't build one. At each step of its computation, this machine can make a non-deterministic choice: it has a choice between several actions, and it takes one. If one of the choices leads to accepting the input, it's considered that it made the right choice. In some sense, it always guesses correctly.
3. **NP Class:** Quickly Verifiable.
   P Class: Quickly Solvable.

Classes P and NP

Class NP

- Let A be a problem. An verifier for A is an algorithm V that takes pairs $\langle x, y \rangle$ as input, where x is an instance of A, and verifies that y is indeed a proof (or certificate) showing that x is a positive instance.
- Problem A has a polynomial verifier V if:
  1. There exists a polynomial $p(.)$ such that for every instance x of A: x is a positive instance if and only if there exists a certificate y of size at most $p(\text{size}(x))$ and such that V accepts $\langle x, y \rangle$
  2. The algorithm V is polynomial in size(x)

Example

- A verifier for SAT takes as input a CNF formula and a valuation of its variables, and decides if the valuation makes the formula true.
- A verifier for 3-coloring takes as input a graph and a vertex coloring with $1, 2, 3$, and verifies that adjacent vertices have different colors.

1. An instance of problem A is the question posed on a particular input/data of A. Example:
   - Problem: Determine if a non-directed graph is connected.
   - Instance: $V = 1, 2, 3, 4, 5$, $E = (1, 2), (2, 3), (3, 1), (4, 5)$
   - Algorithm: Depth-first search
2. A positive instance of a decision problem is an instance (data) for which the problem's output is "True" (Yes).
3. **Note:** We can always restrict ourselves to certificates of polynomial length in the length of x, since in polynomial time in the length of x, the algorithm V cannot read more than a polynomial number of symbols from the certificate.

Classes P and NP

Class NP

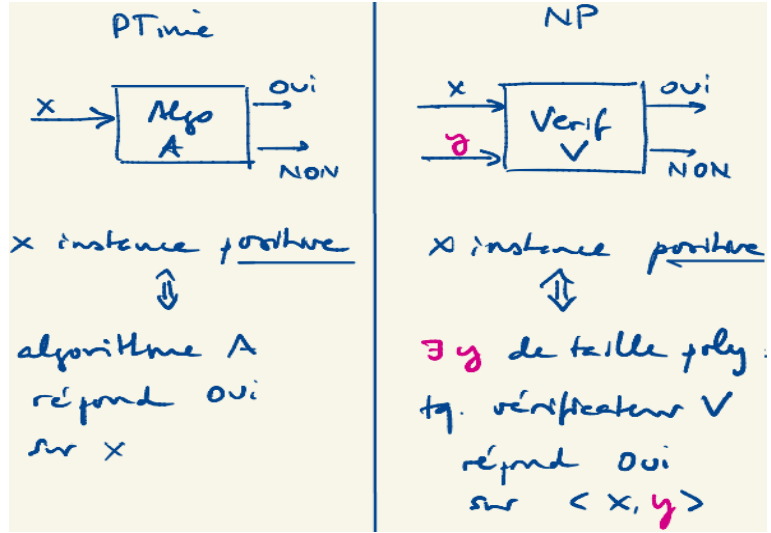
- A verifier for Hamiltonian path takes as input a graph $G = (V, E)$ and a permutation $v_{i_1}, v_{i_2}, ..., v_{i_n}$ of vertices in V, and verifies that it forms a path in G: $(v_{i_j}, v_{i_{j+1}}) \in E$ for all j

How to Prove Membership in NP

The class NP is the class of problems that have polynomial verifiers.

Example

Consider the following problem: given a set S of n integers and an integer b, does there exist a subset T of S whose sum of elements equals b?
This problem is in NP because verifying that the sum of a subset T (certificate), which is a subset of a set S (problem instance) of size n, is equal to b can be done in $O(n^2)$. In this verification, it is necessary to ensure that all elements of T are in S, which will require at most n^2 tests.



NP-Complete Problems

- These are the most difficult problems in NP, often referred to as the "hard core".
- The concept of NP-complete problems is based on that of polynomial transformation (reduction) of one problem to another.

Polynomial Reduction

A problem of existence P_1 is polynomially reducible to another problem P_2 if there exists a polynomial algorithm A that transforms any instance of P_1 into an instance of P_2, while preserving the Yes and No answers.

- We denote this as $P_1 \leq_p P_2$.

Example

A stable set in a simple graph G is a clique in the complement graph G_c.

- An NP-complete problem is an NP problem to which any other NP problem can be polynomially reduced.
- The class of NP-complete problems is denoted as NPC.

1. The existence problem of a stable set in a simple graph G corresponds to the existence problem of a clique in the complement graph G_c. The polynomial transformation here involves computing the complement graph G_c.
2. In the literature, you may encounter the term "NP-hard" for optimization problems: a combinatorial optimization problem is NP-hard if its associated existence problem is NP-complete.

NP-Complete Problems

- The strong practical consequence of the NPC class:
  - If we were to find a polynomial algorithm A for a single NP-complete problem X
    $\Longrightarrow$ We could derive one for any other difficult problem Y in NP
  - It's sufficient to polynomially transform instances of Y into instances for X, and then execute the algorithm for X
- The logician Cook (and Levin) showed in 1970 that the SAT problem is NP-complete

SAT Problem

Data Given a set of variables $x_1, ..., x_n$ and a logical formula φ in conjunctive normal form $\varphi = c_1 \wedge c_2 \wedge ... \wedge c_l$, where each clause $c_i = (y_{i,1} \vee y_{i,2} \vee ... \vee y_{i,k_i})$, and each $y_{i,j}$ is a literal.

Question Is there a truth value assignment to the variables that makes φ true?

1. SAT is the first problem proven to be NP-complete. The proof was established by Cook and Levin.

NP-Complete Problems

How to Prove a Problem is NP-Complete?

- Given a problem P, we consider its associated existence problem
  - If P is an optimization problem, we consider the associated existence problem by replacing the search for an optimal solution with a search for a solution with cost at most k, where k is an integer added to the input.
- The technique used to prove that an existence problem X in NP is NP-complete involves demonstrating that a known NP-complete problem Y can be polynomially reduced to X (and not the other way around, a common mistake!)

Reduction Technique

To prove the NP-completeness of a problem P, it is sufficient to prove:

1. it has a polynomial verifier (ensuring that $P \in NP$)
2. $P' \leq_p P$ with a known NP-complete problem P' (ensuring $C \leq_p P$ for any problem $C \in NP$)

Optimization Problems: The goal of an optimization problem is to find a solution that maximizes (or minimizes) a given objective function. For each optimization problem, we can associate a decision problem that determines whether there exists a solution for which the objective function is greater (or less) than or equal to a given value. The complexity of an optimization problem is related to that of the associated decision problem. In particular, if the decision problem is NP-complete, then the optimization problem is called NP-hard.

3-SAT Problem

Definition

Data: A set of variables $x_1, ..., x_n$ and a formula $\varphi = c_1 \wedge c_2 \wedge ... \wedge c_p$ where $c_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$ and for all i, j, $y_{i,j}$ is a literal, i.e., $y_{i,j}$ is either x_k or $\neg x_k$ for one of the x_k.

Question: Is there an assignment of truth values to the variables that makes φ true?

Theorem

The 3-SAT problem is NP-complete.

To prove this theorem, it's necessary to demonstrate:

1. 3-SAT $\in NP$
2. $P' \leq_p$ 3-SAT with a known NP-complete problem P' (we'll take $P' = SAT$)

The 3-SAT problem is a restriction of the SAT problem to formulas that are in conjunctive normal form with 3 literals.

3-SAT Problem

Proof of 3-SAT Problem's NP-Completeness

1. Is 3-SAT in NP?
   A verifier for 3-SAT takes as input a 3-CNF formula and a variable valuation (certificate), and checks if the valuation makes the formula true. This verification can be done in $O(p)$ time (p is the number of clauses). Therefore, the verifier is polynomial.
   $\Longrightarrow$ Consequently, 3-SAT is in NP.
2. Is SAT $\leq_p$ 3-SAT?
   For any instance φ of SAT, we will associate an instance $\widetilde{\varphi}$ of 3-SAT such that:

$$\varphi \text{ is satisfiable } \Longleftrightarrow \widetilde{\varphi} \text{ is satisfiable}$$

Remark

We will construct $\widetilde{\varphi}$ from φ in polynomial time (polynomial reduction) with respect to the size $|\varphi|$ of φ.

Polynomial reduction ensures that we can use an algorithm P for 3-SAT to solve SAT: for any CNF formula φ, we first construct $\widetilde{\varphi}$ and then run P on $\widetilde{\varphi}$. Let P' be the algorithm constructed in this way.
Important: If P is a polynomial algorithm for 3-SAT, then P' is also polynomial.

Proof of the NP-completeness of the 3-SAT problem

Reduction from SAT to 3-SAT

If $\varphi = c_1 \wedge ... \wedge c_k$ where each c_i is a clause, we construct $\widetilde{\varphi} = \varphi_1 \wedge ... \wedge \varphi_k$, where:

- Each φ_i is a conjunction of 3-clauses
- φ_i uses the variables from c_i, + possibly new variables
- If an assignment of variables makes c_i true, we can complete it to make φ_i true
- Conversely, if an assignment of variables makes φ_i true, its restriction to the variables in c_i makes c_i true

Proof of the NP-completeness of the 3-SAT problem

Reduction from SAT to 3-SAT: Construction of φ_i

- If $c_i = \ell_1$ (a literal), we add 2 new variables y_i, z_i, and

$$\varphi_i = (\ell_1 \vee y_i \vee z_i) \wedge (\ell_1 \vee \neg y_i \vee z_i) \wedge (\ell_1 \vee y_i \vee \neg z_i) \wedge (\ell_1 \vee \neg y_i \vee \neg z_i)$$

- If $c_i = \ell_1 \vee \ell_2$ (2 literals), we add 1 new variable y_i, and

$$\varphi_i = (y_i \vee \ell_1 \vee \ell_2) \wedge (\neg y_i \vee \ell_1 \vee \ell_2)$$

- If c_i is a 3-clause: $\varphi_i = c_i$
- If $c_i = \ell_1 \vee ... \vee \ell_k$ with $k \geq 4$, we add $k-3$ new variables $t_{i,1}, ..., t_{i,k-3}$ and

$$\varphi_i = (t_{i,1} \vee \ell_1 \vee \ell_2) \wedge (\neg t_{i,1} \vee \ell_3 \vee t_{i,2}) \wedge (\neg t_{i,2} \vee \ell_4 \vee t_{i,3}) \wedge ... \wedge (\neg t_{i,k-3} \vee \ell_{k-1} \vee \ell_k)$$

Proof of the NP-completeness of the 3-SAT problem

Reduction from SAT to 3-SAT: Construction of φ_i

Example

$$\varphi = (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee x_5) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

Proof of the NP-completeness of the 3-SAT problem

Reduction from SAT to 3-SAT: Construction of φ_i

Example

$$\varphi = (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee x_5) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

Then the construction yields:

$$\begin{aligned}\widetilde{\varphi} = \quad & (t_{1,1} \vee x_1 \vee \neg x_2) \wedge (\neg t_{1,1} \vee x_3 \vee t_{1,2}) \wedge (\neg t_{1,2} \vee \neg x_4 \vee x_5) \\ & \wedge (y_2 \vee x_1 \vee \neg x_2) \wedge ((\neg y_2 \vee x_1 \vee \neg x_2)) \\ & \wedge (\neg x_1 \vee x_2 \vee x_4)\end{aligned}$$

Proof of the NP-completeness of the 3-SAT problem

Reduction from SAT to 3-SAT

We verify that with the previous construction:

- If an assignment of the variables makes each c_i true, it can be easily extended to make each φ_i true.
- Conversely, if an assignment of the variables makes each φ_i true, restricting this assignment to the variables of c_i makes c_i true.
- Therefore,

$$c_i \text{ is satisfiable}$$

$$\Updownarrow$$

$$\varphi_i \text{ is satisfiable with the same values for the variables of } c_i$$

- Since the variables added in φ_i only appear in φ_i:

$$\varphi \text{ is satisfiable} \iff \widetilde{\varphi} \text{ is satisfiable}$$

Proof of the NP-completeness of the 3-SAT problem

Reduction from SAT to 3-SAT

- The computational time is reduced to writing the clauses of φ_i, whose length is polynomial in relation to the size of φ.
- The entire reduction process is therefore carried out in polynomial time.

We have thus established:

$$\text{SAT} \leq_p \text{3-SAT}$$

Conclusion

$$\begin{cases} \text{3-SAT} \in NP \\ \text{SAT} \leq_p \text{3-SAT} \end{cases} \Longrightarrow \text{3-SAT is NP-complete}$$

STABLE Problem (Independent Set)

Definition

Data: An undirected graph $G = (V, E)$ and an integer k

Question: Does there exist a subset $V' \subset V$ with $|V'| = k$, such that $u, v \in V' \Rightarrow (u, v) \notin E$?

Theorem

The STABLE problem is NP-complete

To prove this theorem, we establish that:

1. STABLE $\in NP$
2. 3-SAT $\leq_p$ STABLE (It is already known that 3-SAT is an NP-complete problem)

The STABLE problem is the decision problem which, given a graph G and an integer k, aims to determine whether there exists in G a set of k vertices pairwise non-adjacent. Such a set is called a stable set of k vertices.

STABLE Problem (Independent Set)

Proof of the NP-completeness of the STABLE Problem

1. STABLE $\in NP$?
   A verifier for STABLE takes as input a graph and a set of vertices V' (certificate), and checks whether V' is a set of pairwise non-adjacent vertices of size k. This algorithm runs in $O(|V|^2)$ in the worst case, making the verifier polynomial.
   $\Longrightarrow$ Consequently, STABLE $\in NP$
2. 3-SAT $\leq_p$ STABLE ?
   For any instance φ of 3-SAT, we associate an instance G_φ, k_φ of STABLE such that

$$\varphi \text{ is satisfiable } \Longleftrightarrow G_\varphi \text{ has an independent set of size } k_\varphi$$

Remark

We construct G_φ, k_φ from φ in polynomial time (polynomial reduction) with respect to the size $|\varphi|$ of φ

Proof of the NP-completeness of the STABLE Problem

Reduction 3-SAT to STABLE

- Let $\varphi = \bigwedge_{1 \leq j \leq k} (x_{1j} \vee x_{2j} \vee x_{3j})$. We construct a graph G_φ with $3k$ vertices, one for each occurrence of a literal in a clause
  - Constraints related to literals: G_φ has an edge between each vertex associated with a literal x_i and each vertex associated with a literal $\neg x_i$ (thus, an independent set of G_φ corresponds to a truth value assignment to a subset of the variables)
  - Constraints associated with clauses: for example, for a clause in φ of the form $c = (x_1 \vee \neg x_2 \vee x_3)$, G_φ has edges $(x_1, \neg x_2)$, $(\neg x_2, x_3)$, (x_3, x_1) forming a triangle (thus, an independent set of G_φ contains at most one of the three vertices associated with the clause c)
- We choose the integer k_φ to be equal to k (the number of clauses in φ)

Proof of the NP-completeness of the STABLE Problem

Reduction 3-SAT to STABLE: Construction of G_φ

Example

$$\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$$

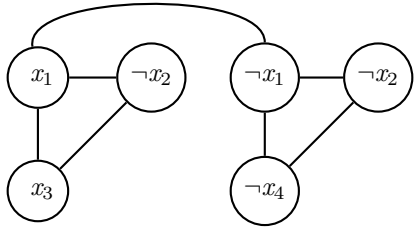
Proof of the NP-completeness of the STABLE Problem

Reduction 3-SAT to STABLE: Construction of G_φ

Example

$$\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$$

Then the construction yields the following graph G_φ:



Proof of NP-completeness of the STABLE Problem

Reduction from 3-SAT to STABLE

We now show that φ is satisfiable if and only if G_φ has a stable of size k_φ:

- If φ is satisfiable, we consider an assignment of variables satisfying φ. For each clause c in φ, we choose y_c as a literal in c made true by the assignment. This defines k vertices forming a stable in G_φ.
- Conversely, if G_φ has a stable of size k_φ, then it must have a vertex in each triangle. This vertex corresponds to a literal that makes the associated clause true, forming a consistent variable assignment due to the construction of edges.
- Therefore,

$$\varphi \text{ is satisfiable } \iff G_\varphi \text{ has a stable of size } k_\varphi$$

Proof of NP-completeness of the STABLE Problem

Reduction from 3-SAT to STABLE

- The computation time reduces to constructing the graph G_φ from the formula φ
- It is polynomial with respect to the size of φ

Therefore, we have:

$$\text{3-SAT} \leq_p \text{STABLE}$$

Conclusion

$$\begin{cases} \text{STABLE} \in NP \\ \text{3-SAT} \leq_p \text{STABLE} \end{cases} \Longrightarrow \text{STABLE is NP-complete}$$

CLIQUE Problem

Definition

Data: An undirected graph $G = (V, E)$ and an integer k

Question: Does there exist a set $V' \subset V$, with $|V'| = k$, such that $u, v \in V' \Rightarrow (u, v) \in E$?

Theorem

The CLIQUE problem is NP-complete

To prove this theorem, we demonstrate that:

1. CLIQUE $\in NP$
2. STABLE $\leq_p$ CLIQUE (STABLE is an NP-complete problem)

The CLIQUE problem is to determine if a graph contains a clique (a subset of graph vertices all adjacent to each other, also known as a complete subgraph) of size k.

CLIQUE Problem

Demonstration of CLIQUE Problem NP-completeness

1. Is CLIQUE $\in NP$?
   1. **Certificate:** Let the certificate be a set S consisting of nodes in the clique, and S is a subgraph of G.
   2. **Verification:** Verifying if the number of nodes in S equals k takes $O(1)$ time. Verifying if each vertex has an external degree of $(k-1)$ takes $O(k^2) = O(|V|^2)$ time (since $k \leq |V|$). Thus, the decision problem for clique has a polynomial verifier.
      $\Longrightarrow$ Therefore, CLIQUE $\in NP$
2. STABLE $\leq_p$ CLIQUE?
   For any instance G, k of STABLE, we associate an instance G', k' of CLIQUE such that

$$G \text{ has a stable of size } k \Longleftrightarrow G' \text{ has a clique of size } k'$$

Remark

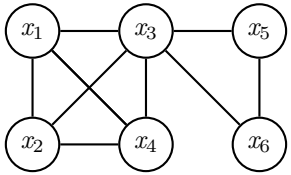
We construct G', k' from G, k in polynomial time relative to the size of G.

Demonstration of CLIQUE Problem NP-completeness

Reduction STABLE to CLIQUE

- We will construct the graph $G' = (V', E')$ from the graph $G = (V, E)$ by the following modifications:
  - $V' = V$, meaning that all vertices of the graph G are part of the graph G'
  - $E' =$ complement of the edges E, that is, the edges not present in the original graph G
- The graph G' is the complementary graph of G

Example of constructing a complementary graph

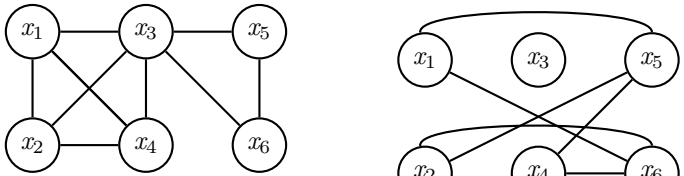


Demonstration of CLIQUE Problem NP-completeness

Reduction STABLE to CLIQUE

- We will construct the graph $G' = (V', E')$ from the graph $G = (V, E)$ by the following modifications:
  - $V' = V$, meaning that all vertices of the graph G are part of the graph G'
  - $E' =$ complement of the edges E, that is, the edges not present in the original graph G
- The graph G' is the complementary graph of G

Example of constructing a complementary graph



Demonstration of CLIQUE Problem NP-completeness

Reduction STABLE to CLIQUE

We now demonstrate that G has a stable of size k if and only if G' has a clique of size k':

- Suppose the graph G contains a clique of size k. This implies that there are k vertices in G, where each of these vertices is connected by an edge to the remaining vertices. Furthermore, since these edges are contained in G, they cannot be present in G'. Therefore, these k vertices are not adjacent to each other in G' and thus form a stable of size k.
- Conversely, assume the complementary graph G' has a stable of vertices of size k'. None of these vertices share an edge with other vertices. When we complete the graph to obtain G, these k' vertices will share an edge and thus become adjacent to each other. Consequently, the graph G will have a clique of size k'.
- Hence,

$$G \text{ has a stable of size } k \iff G' \text{ has a clique of size } k$$

Demonstration of CLIQUE Problem NP-completeness

Reduction STABLE to CLIQUE

- The computation time reduces to calculating the complementary graph G', which can be done in the worst case in $O(|V|^2)$ time.
- The reduction process is thus carried out in polynomial time.

We therefore have:

$$\text{STABLE} \leq_p \text{CLIQUE}$$

Conclusion

$$\begin{cases} \text{CLIQUE} \in NP \\ \text{STABLE} \leq_p \text{CLIQUE} \end{cases} \Longrightarrow \text{CLIQUE is NP-complete}$$

3-COLORABILITY Problem

Definition

Data: An undirected graph $G = (V, E)$.

Question: Does there exist a coloring of the graph using at most 3 colors, such that no two adjacent vertices receive the same color?

Theorem

The 3-COLORABILITY problem is NP-complete.

To prove this theorem, we show that:

1. 3-COLORABILITY $\in NP$
2. 3-SAT $\leq_p$ 3-COLORABILITY

A k-COLORABILITY problem for undirected graphs aims to determine if there is a vertex-coloring of the graph such that no two adjacent vertices share the same color, and at most k colors are used to complete the graph coloring.

3-COLORABILITY Problem

Proof of NP-Completeness of the 3-COLORABILITY Problem

1. Is 3-COLORABILITY in NP?
   1. **Certificate:** A coloring assignment $\{c_1, c_2, c_3\}$ where each vertex is assigned one of these three colors.
   2. **Verification:** Verify that for each edge (u, v), the color of vertex u is different from that of vertex v. The number of edges in a graph in the worst case is $\frac{|V|(|V|-1)}{2}$ ($|V|$ is the number of vertices in the graph). Thus, the verification takes time in $O(|V|^2)$
      Therefore, the 3-COLORABILITY problem has a polynomial verifier
      $\Longrightarrow$ Hence, 3-COLORABILITY $\in NP$
2. Is 3-SAT $\leq_p$ 3-COLORABILITY?
   For every instance φ of 3-SAT, we associate an instance G_φ of the 3-COLORABILITY problem such that

$$\varphi \text{ is satisfiable} \Longleftrightarrow G_\varphi \text{ has a 3-coloring}$$

Remark

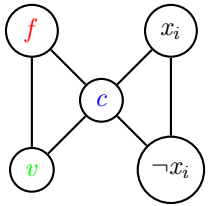
We will construct G_φ from φ in polynomial time relative to the size of φ

Proof of NP-Completeness of the 3-COLORABILITY Problem

Reduction from 3-SAT to 3-COLORABILITY

- Let $\varphi = c_1 \wedge ... \wedge c_m$. We construct a graph G_φ as follows:
  1. We create three vertices v, f, c (for true, false, control). These three vertices are connected pairwise in a triangle, so they must all be of different colors. We associate a vertex with each variable and its complement. To ensure that a variable takes either the true or false value, for each variable x_i we construct a triangle with vertices x_i, $\neg x_i$, and c
     $\Longrightarrow$ This results in a total of $2n + 3$ vertices and $3n + 3$ edges

Illustration

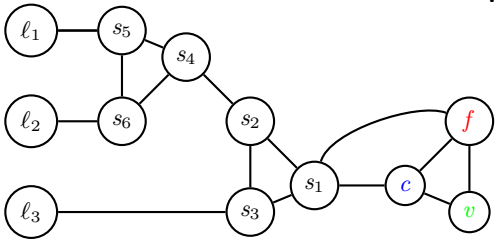


This enforces that either color(x_i) = true and color($\neg x_i$) = false, or color(x_i) = false and color($\neg x_i$) = true

Proof of NP-Completeness of the 3-COLORABILITY Problem

Reduction from 3-SAT to 3-COLORABILITY

2. We now need to encode the evaluation rules of a clause. To do this, we introduce the following subgraph, which corresponds to a clause $\ell_1 \vee \ell_2 \vee \ell_3$



Property:

- If ℓ_1, ℓ_2, ℓ_3 are colored false in a 3-coloring, then the output node s_1 will be colored false (which is impossible as s_1 must be true)
- If any of ℓ_1, ℓ_2, ℓ_3 is colored true, then the constructed subgraph can be 3-colorable in a way that the output node s_1 is colored true

Similarly, we create subgraphs associated with the other clauses of φ

$\Longrightarrow$ This results in $6m$ new vertices and $12m$ new edges

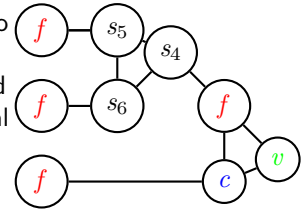
In total, the graph has $2n + 6m + 3$ vertices and $3n + 12m + 3$ edges

Proof of NP-Completeness of the 3-COLORABILITY Problem

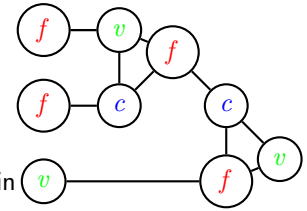
Reduction from 3-SAT to 3-COLORABILITY

Proof of the Property:

1. If ℓ_1, ℓ_2, ℓ_3 are all colored false, then we are forced to use the coloring shown in the figure. But then s_4, s_5, s_6 must all be colored with different colors, and none of them can be colored false, so there is no legal coloring.



2. The rest of the proof considers the 7 possible color combinations of ℓ_1, ℓ_2, ℓ_3 such that at least one is colored true and the others are colored false, and shows that a 3-coloring exists in each case. For example, if ℓ_3 is colored true but ℓ_1 and ℓ_2 are colored false, we have the legal 3-coloring illustrated in the figure. The other cases are similar.



Proof of NP-Completeness of the 3-COLORABILITY Problem

Reduction from 3-SAT to 3-COLORABILITY

Example

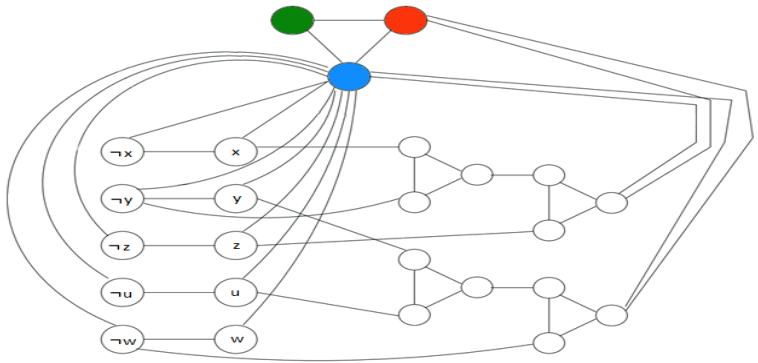
$\varphi = (x \vee \neg y \vee z) \wedge (y \vee u \vee \neg w)$

Proof of NP-Completeness of the 3-COLORABILITY Problem

Reduction from 3-SAT to 3-COLORABILITY

Example

$\varphi = (x \vee \neg y \vee z) \wedge (y \vee u \vee \neg w)$, the graph G_φ constructed from φ is as follows:



Proof of NP-Completeness of the 3-COLORABILITY Problem

Reduction from 3-SAT to 3-COLORABILITY

It remains to show that φ is satisfiable if and only if the constructed graph G_φ is 3-colorable:

- **φ is satisfiable implies G_φ is 3-colorable**
  - If φ is satisfiable, then in each clause, at least one of the literals x_i must be true. As a result, the vertex corresponding to x_i in G_φ can be assigned a true color and $\neg x_i$ can be assigned a false color. Now, by extending this, for each clause, the corresponding subgraph in G_φ can be 3-colorable. Therefore, the graph can be 3-colorable (by the property above).
- **G_φ is 3-colorable implies φ is satisfiable**
  - If φ is not satisfiable, there exists at least one clause $c_j = (\ell_1 \vee \ell_2 \vee \ell_3)$ where all three literals ℓ_1, ℓ_2, ℓ_3 are false. In this case, the subgraph corresponding to c_j cannot be 3-colorable. The output node of the subgraph is colored false, which is impossible because it must be true (by the property above). Therefore, the graph G_φ is not 3-colorable [3].

[3] We are using a proof by contrapositive.

1. A proof by contrapositive shows an implication $P \Longrightarrow Q$ by proving its contrapositive $\neg Q \Longrightarrow \neg P$

Proof of NP-Completeness of the 3-COLORABILITY Problem

Reduction from 3-SAT to 3-COLORABILITY

- The computation time is reduced to creating the vertices and edges of G_φ, the number of which ($2n + 6m + 3$ vertices and $3n + 12m + 3$ edges) is polynomial in relation to the size of φ.
- The reduction is thus polynomial.

Therefore, we have:

$$\text{3-SAT} \leq_p \text{3-COLORABILITY}$$

Conclusion

$$\begin{cases} \text{3-COLORABILITY} \in NP \\ \text{3-SAT} \leq_p \text{3-COLORABILITY} \end{cases} \Longrightarrow \text{3-COLORABILITY is NP-complete}$$

Conjecture $P \stackrel{?}{=} NP$

- The crucial question in complexity theory is to determine:
  - Whether NP-complete problems can be solved polynomially $\Longrightarrow$ in which case $P = NP$
  - Or if polynomial algorithms for them will never be found $\Longrightarrow$ in which case $P \neq NP$
- This question is still unsettled.
- As hundreds of NP-complete problems resist the assault of research efforts, most theoretical computer scientists today conjecture that $P \neq NP$.
- Problems with undetermined status, which are in NP, are among the hard problems to solve (the best available algorithms have exponential complexity), but a proof of their membership in NPC is not yet available.

An example of an undetermined status problem is the graph isomorphism problem:

Two graphs G and H are isomorphic if one can be transformed into the other by renumbering the vertices.

The graph isomorphism problem is the decision problem that involves determining whether two undirected graphs are isomorphic, meaning they are the same up to vertex renaming.
The graph isomorphism problem doesn't have a known polynomial algorithm, but nobody has been able to show that it's NP-complete.

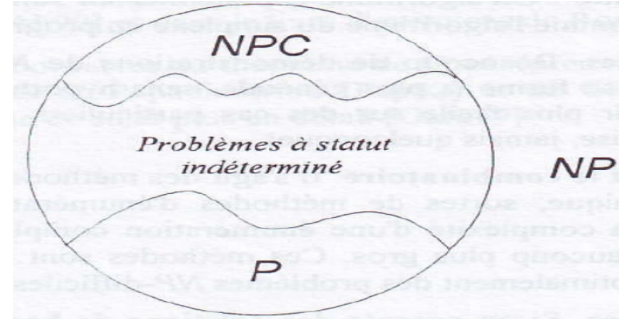


Figure: Illustration of the classes P, NP, and NPC.

Impact on the Practical Approach to a Real Problem

- There are combinatorial problems that you shouldn't insist on solving.
- If you have to solve a combinatorial problem, it's a good idea to consult a directory of NP-complete problems to see if the associated existence problem is listed there[4].
- If your problem is NP-complete, it's better to give up hope of finding a polynomial algorithm.
- The problem of solving arises for any problem where polynomial algorithms are not available (primarily NP-complete problems). In practice, problems are solved by leveraging:
  - **Data size:**
    Complete enumeration can be feasible for small cases.
  - **Average complexity:**
    An exponential algorithm in its worst-case scenario can be quite fast on average (like the simplex algorithm).

[4] Over 3000 problems listed in 2019!

1. A problem is classified as combinatorial when its solution involves exploring a huge number of combinations.
2. The simplex algorithm is an algorithm for solving linear programming problems. It's probably the first algorithm that allows minimizing a function over a set defined by inequalities.

Impact on the Practical Approach to a Real Problem

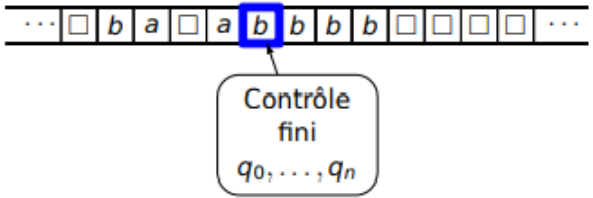
- **Based on the Nature of Data:**
  NP-completeness proofs consider a problem in its most general form, without assumptions about the data. The problem might become easier for specific cases or for specific data of a company, never for arbitrary cases.
- **Using Methods to Reduce Combinatorial Complexity:**
  These include tree-based methods, dynamic programming, and constraint programming – intelligent enumeration methods.
- **Applying Approximation Methods:**
  If accepting solutions of good quality without optimality guarantees, fast algorithms for local search can be found. In a business context, a solution is valuable if it enables cost savings compared to existing solutions or manual methods, even if it's not optimal.

1. An optimal solution (optimization problem) is a feasible solution where the objective function achieves its maximum (or minimum) value, i.e., the highest profit or the least cost.
2. Local search involves moving from one solution to a neighboring solution in the space of candidate solutions (the search space) until a solution considered optimal is found or the allotted time is exceeded.

Turing Machines

An Turing machine consists of:

- An infinite tape, with each cell holding an element from an alphabet Γ that includes a "blank" symbol ($\square$).
- A read/write head, which is in a state chosen from a finite set Q. There's an initial state and final states.
- A transition function, which, based on the state of the read/write head and the content of the cell being read, determines the new state of the head, the new content of the cell, and moves the head one cell to the right or left.



Turing Machines: Formalization

An Turing machine (TM) with one tape is given by $M = (Q, q_0, F, \Sigma, \Gamma, \delta)$:

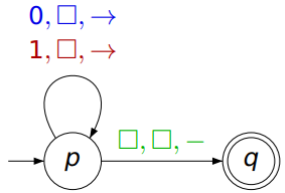
- Q: finite set of states.
- q_0: initial state.
- $F \subseteq Q$: set of final (or accepting) states.
- Γ: finite tape alphabet, including $\square$.
- Σ: input alphabet, with $\Sigma \subseteq \Gamma \backslash \{\square\}$.
- δ: set of transitions. A transition is of the form (p, a, q, b, d), denoted $p \xrightarrow{a,b,d} q$, with:
  - $p, q \in Q$
  - $a, b \in \Gamma$
  - $d \in \{\leftarrow, -, \rightarrow\}$
- We assume that no transition originates from a state in F.

Turing Machines: Graphical Representation

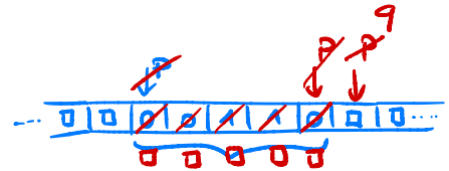
- Turing machines are often represented like automata.
- Only the transition labels change.

Example

With $\Gamma = \{0, 1, \square\}$ and $\Sigma = \{0, 1\}$:



$$\delta = \{(p, 0, \square, \rightarrow, p), (p, 1, \square, \rightarrow, p), (p, \square, \square, -, q)\}$$



Operation of a Turing Machine

- Initially, a word w is written on the tape surrounded by $\square$ symbols.
- A computation of a Turing machine on w is a sequence of computation steps.
- This sequence can be finite or infinite.
- The computation starts:
  - with the read-write head on the first letter of w, and
  - in the initial state q_0.
- Each computation step involves applying a transition, if possible.
- The computation only stops if no transition is applicable.
- Each step consists of applying a transition.
- A transition of the form $p \xrightarrow{a,b,d} q$ is possible only if:
  1. the machine is in state p, and
  2. the symbol under the read-write head is a.
- In this case, applying the transition involves:
  - changing the control state to q,
  - replacing the symbol under the read-write head with b,
  - moving the head one position to the left if $d = \leftarrow$, or
  - moving the head one position to the right if $d = \rightarrow$, or
  - not moving the head if $d = -$.

Configurations and Computations

- A configuration represents an instantaneous snapshot of the computation.
- The configuration uqv signifies that:
  - The control state is q.
  - The word written on the tape is uv, surrounded by $\square$ symbols.
  - The read-write head is on the first letter of v.
- The initial configuration on w is therefore $q_0 w$.
- For 2 configurations C, C', we write $C \vdash C'$ when we obtain C' by applying a transition from C.

A computation of a Turing machine is a sequence of configurations:

$$C_0 \vdash C_1 \vdash C_2 \vdash \cdots$$

Three cases are possible:

- The computation is infinite.
- The computation halts on a final state (in F).
- The computation halts on a non-final state (not in F).

A configuration = generalized state.

Accepted Languages

One can use a Turing machine to accept words.

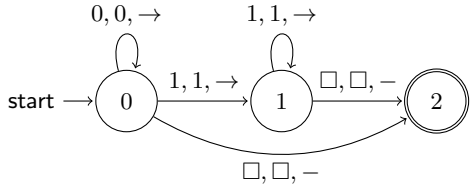
- The language $L(M) \subseteq \Sigma^*$ of words accepted by a TM M is the set of words w for which there exists a finite computation

$$C_0 \vdash C_1 \vdash C_2 \vdash \cdots \vdash C_n$$

  with $C_0 = q_0 w$ (w is the input word) and $C_n \in \Gamma^* F \Gamma^*$
- Three exclusive cases: a computation can:
  - either halt on an accepting state,
  - or halt on a non-accepting state,
  - or not halt at all.
- A machine is said to be deterministic if, for every $(p, a) \in Q \times \Gamma$, there exists at most one transition of the form $p \xrightarrow{a,b,d} q$.
- If M is deterministic, it has only one computation per input.

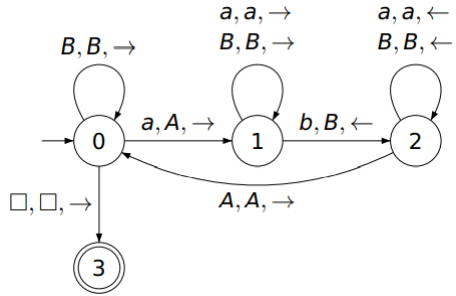
TM Accepting the Language 0^*1^*

- The set of states is $Q = \{0, 1, 2\}$,
- The input alphabet is $\Sigma = \{0, 1\}$,
- The tape alphabet is $\Gamma = \{0, 1, \square\}$,
- The initial state is $q_0 = 0$,
- The only accepting state is state 2,
- The set of transitions is described by the figure on the right.



TM Accepting the Language $X = \{a^n b^n | n \geq 0\}^*$

- The set of states is $Q = \{0, 1, 2, 3\}$,
- The input alphabet is $\Sigma = \{a, b\}$,
- The tape alphabet is $\Gamma = \{a, b, A, B, \square\}$,
- The initial state is $q_0 = 0$,
- The only accepting state is state 3,
- The set of transitions is described by the figure on the right.



1. Idea: Mark the first a and the first b, then repeat.
2. Calculation generated for the word: ab
   $0ab \vdash A1b \vdash 2AB \vdash A0B \vdash AB0 \vdash AB\square 3$
   The word is accepted (3 is an accepting state).
3. Calculation generated for the word: $abab$
   $0abab \vdash ... \vdash AB0ab \vdash ... \vdash ABAB\square 3$
   The word will be accepted.
4. Calculation generated for the word: $aabb$
   $0aabb \vdash A1abb \vdash Aa1bb \vdash A2aBb \vdash 2AaBb \vdash A0aBb \vdash AA1Bb \vdash AAB1b \vdash AA2BB \vdash A2ABB \vdash AA0BB \vdash AAB0B \vdash AABB0AABB\square 3$
   The word is accepted (3 is an accepting state).
5. Calculation generated for the word: aba
   $0aba \vdash A1ba \vdash 2ABa \vdash A0Ba \vdash AB0a \vdash ABA1\square$
   The word is not accepted (calculation stopped at state 1 which is not accepting).

Problems and Languages

- Decision problems versus computation problems: YES/NO answer for the former, output for the latter. The first category is just a special case of the second.
- Instance of a problem A: an input for A. A positive instance of a decision problem is an instance for which the answer is YES.
- Abstractly, a decision problem A can be interpreted as a language problem:
  - We choose a coding f for instances of A over an alphabet Σ.
  - The set of codings for positive instances of A defines a language $L_A \subseteq \Sigma^*$.
  - Asking whether an instance I of A is positive is equivalent to asking if $f(I) \in L_A$.
- The informal notion of an algorithm to solve a decision problem can then be translated into the formal notion of a Turing machine program accepting a word $w \in \Sigma^*$, where w results from the coding of an instance I.

1. This distinction separates two types of problems: those that involve answering a given question with either yes or no (for example, determining if an integer is prime), called decision problems; and those that involve producing a new object (for example, producing a sorted list), called evaluation (or computation) problems. The latter involve evaluating a function: in the example, it's about evaluating the function that associates a sorted list with a list of integers.
2. A deterministic Turing machine accepting a language L computes the characteristic function of L defined by:

$$f : \quad \Sigma^* \to \{0,1\}$$
$$w \to \begin{cases} 0 & \text{if } w \notin L, \\ 1 & \text{if } w \in L. \end{cases}$$

Problems and Languages

Definition (Time Complexity for a Deterministic Turing Machine)

The time complexity of a program of a deterministic Turing machine M is the function of n (size of a coded word from the language):

$$T_M(n) = \max_{x \in \Sigma^*, |x|=n} \{k | k \text{ is the length of computation on input } x\}$$

Definition (Time Complexity for a Non-deterministic Turing Machine)

The time complexity of a program of a non-deterministic Turing machine M is the function of n (size of a coded word from the language):

$$T_M(n) = \max_{x \in \Sigma^*, |x|=n} \{\min\{k | k \text{ is the length of computation on input } x\}\}$$

- The definition of time complexity for a non-deterministic Turing machine program is more intricate.
  - We take the maximum over the set of words of size n, but for each word, we consider following the shortest path leading to an acceptance in the resolution tree of the non-deterministic Turing machine.
  - This is due to the non-deterministic nature of a computation where multiple execution paths are possible.

1. The time complexity of a deterministic Turing machine is a function that depends on an integer n, which will be the size of a word. And for example, for $n = 10$, we look at all words of size 10, and we find the maximum number (the cost) of transitions used to read a word of size 10 in the Turing machine (to determine if it's accepted or rejected).
   For example, the cost of the word "aba" on the Turing machine defined in slide 54 is 5 ($cost_M(aba) = 5$). This definition is close to the complexity of an algorithm where, for an input of size n, we look at the worst-case complexity, i.e., counting the maximum number of elementary operations performed by the algorithm.

Further Notes on Problem Classification

- $DTIME(f(n))$ is the set of languages that can be recognized in time $O(f(n))$ by a deterministic Turing machine.
- $NTIME(f(n))$ is the set of languages that can be recognized in time $O(f(n))$ by a non-deterministic Turing machine.
- $DSPACE(f(n))$ is the set of languages that can be recognized using memory consumption $O(f(n))$ by a deterministic Turing machine.
- $NSPACE(f(n))$ is the set of languages that can be recognized using memory consumption $O(f(n))$ by a non-deterministic Turing machine.

$$
\begin{aligned}
P &= \bigcup_{k\geq 0} DTIME(n^k) \\
NP &= \bigcup_{k\geq 0} NTIME(n^k) \\
PSPACE &= \bigcup_{k\geq 0} DSPACE(n^k) \\
NPSPACE &= \bigcup_{k\geq 0} NSPACE(n^k) \\
EXPTIME &= \bigcup_{k\geq 0} DTIME(2^{n^k})
\end{aligned}
$$

1. If M is a deterministic Turing machine and x is an input, the space used by M on x is the number of different tape cells visited by M during its computation.
2. The class P of decision problems decidable by a deterministic Turing machine in polynomial time with respect to the input size.
3. The class NP of decision problems decidable by a non-deterministic Turing machine in polynomial time with respect to the input size.
4. $PSPACE$ is the complexity class of decision problems decided by a deterministic Turing machine with polynomial space.
5. The complexity class **NPSPACE** is the set of decision problems that can be solved using polynomial space by a non-deterministic Turing machine.
6. The class $EXPTIME$ of decision problems decidable by a deterministic Turing machine in exponential time.
   An example of an EXPTIME problem is generalized chess: given a $2n \times 2n$ chessboard and a given piece arrangement, we want to determine if this position is winning for the white pieces.

Further Notes on Problem Classification

- Given a class of languages C, we define $co - C$ as the set of languages whose complements are in C. Thus, we have the classes $co - P$, $co - NP$, $co - PSPACE$, and $co - NPSPACE$.
- Savitch's theorem states that any problem decidable by a non-deterministic machine in polynomial space is also decidable by a deterministic machine in polynomial space ($PSPACE = NPSPACE$).

More generally, we have: $P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$

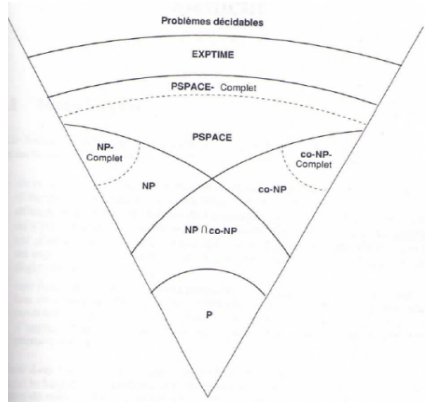


Figure: Complexity Classes

1. For every problem in NP, we can construct a "dual" problem in co-NP.

Problem in NP	**Complementary Problem in co-NP**
The SAT problem	Given a boolean formula, is it false for all assignments of its variables?
The Hamiltonian path problem	The problem of non-existence of a Hamiltonian path.
The clique problem	The non-clique problem: given a graph G and an integer k, is it true that G does not have a clique of size k?

2. The question $co - NP = NP$ is an open question, but it's conjectured that these classes are distinct. If $P = NP$, then $NP = co - NP$, but the converse is not known.
3. We know that $P \subseteq NP \cap co - NP$, but equality is an open question. The primality testing problem is known to be in both NP and $co - NP$, but it was generally believed not to be in P until the proof of the AKS theorem in 2002.