



TCS3-IN Introduction to computational complexity

Christophe Garion, Stéphanie Roussel and others...

ISAE-SUPAERO/DISC

These slides have been written using previous lectures material from several authors. The following people should be accredited as co-authors of the present slides:

- Hélène Fargier (CNRS)
- Gérard Verfaillie (ONERA)
- Cédric Pralet (ONERA)
- Stéphanie Roussel (ONERA)

Many thanks to Pierre-Antoine Morin for his implication and the exercise on resource-constrained project scheduling.

Many thanks to Nicolas Barnier for his careful re-reading of the slides.



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license (CC BY-NC-SA 3.0)

You are free to Share (copy, distribute and transmit) and to Remix (adapt) this work under the following conditions:



Attribution – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).



Noncommercial – You may not use this work for commercial purposes.



Share Alike – If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

See <http://creativecommons.org/licenses/by-nc-sa/3.0/>.

Learning outcomes

At the end of this session you should

- have understood that some problems are (really) more difficult than others
- be able to show that a problem is in complexity class P , either directly or by reduction
- be able to show that a problem is in complexity class NP
- be able to show that a problem is in complexity class NPC by reduction
- know some classic problems: SAT, TSP, graph coloring etc.



All solutions to exercises will be given before the exam.



Credits: © 2008 Jorge Cham, PhD Comics, under fair use
<http://phdcomics.com/comics/archive.php?comicid=4>

1 Introduction to combinatorial optimization

- Some examples
- Framework

2 Complexity theory

3 The P complexity class

4 The NP and NPC classes

1 Introduction to combinatorial optimization

- Some examples
- Framework

2 Complexity theory

3 The P complexity class

4 The NP and NPC classes

1 Introduction to combinatorial optimization

- Some examples
- Framework

2 Complexity theory

3 The P complexity class

4 The NP and NPC classes



Input

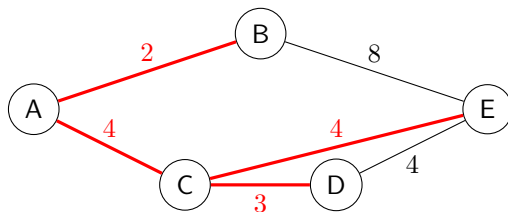
- some **equipments**
- possible **connections** between equipments
- **costs** associated to connections

Objective

Connect all equipments by **minimizing the total connections cost**.



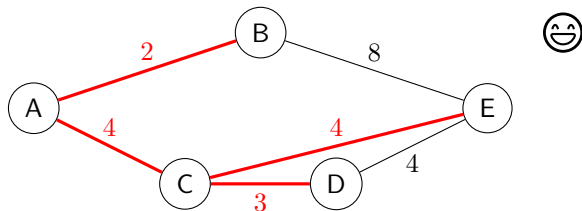
A possible model: **weighted graphs**.



Exercise

Is it easy or hard to compute?

A possible model: **weighted graphs**.



Exercise

Is it easy or hard to compute?

Easy, this is a classical problem: computing the **Minimum Spanning Tree** of a graph.



Input

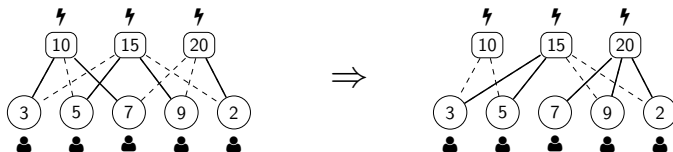
- a set of **power stations** with capacities
- a set of **users** with fixed demands
- for each users, a **subset of the set of the power stations** to which the user can be attached
- two solutions S and S' :
 - each user is attached to a **single power station**
 - the **sum of users demands** attached to a station does not exceed the station's capacity

Objective

Can we find a **sequence** of solutions $[S, S_1, \dots, S_n, S']$ s.t. each S_i is also a **feasible solution** and the only difference between S_i and S_{i+1} is that a **single user** has changed its power station.



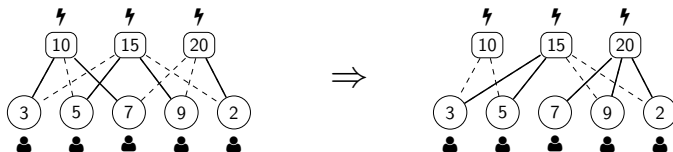
The problem can be modeled by a graph:



Exercise

Is it easy or hard to compute?

The problem can be modeled by a graph:



Hard to compute, the power supply reconfiguration problem is a **difficult problem**.



Takehiro Ito et al.

“On the complexity of reconfiguration problems”.

In: **Theoretical Computer Science** 412.12 (2011), pp. 1054–1065.

ISSN: 0304-3975.

DOI: <https://doi.org/10.1016/j.tcs.2010.12.005>.



Input

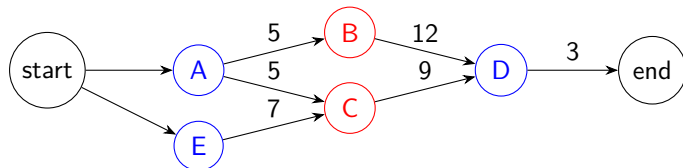
- tasks
- tasks **durations**
- **precedence** constraints between tasks
- **resources**

Objective

Organize tasks in time in order to **minimize** the project total length.



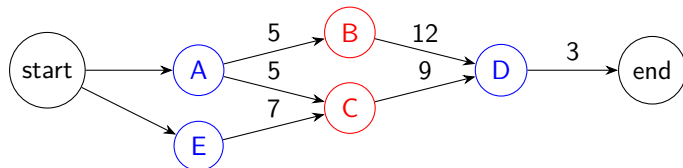
A possible model: **weighted graphs** again!



Exercise

Is it easy or hard to compute?

A possible model: **weighted graphs** again!



Exercise

Is it easy or hard to compute?

Easy, if you do not consider resources: longest path in a weighted acyclic graph.

Hard otherwise!

1 Introduction to combinatorial optimization

- Some examples
- Framework

2 Complexity theory

3 The P complexity class

4 The NP and NPC classes

Some common features (and vocabulary)

- a **search space** $\mathcal{S}$: the possible alternatives
- the search space is represented by a set of **variables**
 $V = \{v_i : i \in \{1, \dots, n\}\}$, each variable v_i having a domain D_i
 - an alternative is an instantiation of each v_i
 - the search space is thus $\prod_{i \in \{1, \dots, n\}} D_i$
- a set Co of **constraints** to be satisfied
 - constraints are **assertions**
 - they may represent physical (real-world) limitations or user prerequisites
 - a **solution** is an alternative that satisfies all constraints in Co
- a set Cr of **criteria** to be satisfied as best
 - a criterion is a function from $\mathcal{S}$ to a totally ordered set ($\mathbb{R}^+$ for instance). This function has to be **minimized or maximized**
 - they represent **user preferences**

The knapsack problem

Input

- a set O of **objects** to put in the knapsack
- a **dimension** D to be considered (weight for instance)
- maximal **capacity** C w.r.t. the dimension for the knapsack
- for each object o a **consumption** W_o (weight for example)
- for each object o , its **value** V_o reflects its importance

Objective

Decide which objects to put in the knapsack in order to maximize the sum of values of the chosen objects while respecting the capacity of the knapsack. You can put an object only one time in the knapsack.



Exercise

Model the knapsack problem!



Exercise

Model the knapsack problem!

- variables: $\forall o \in O \ p_o \in \{0, 1\}$
- constraint: $\sum_{o \in O} p_o \times W_o \leq C$
- criterion: maximize $\sum_{o \in O} p_o \times V_o$

Challenges

In the general case, there is **no analytical solution** to the problem.

Even if there is no analytical solution, **computational power** can be used to compute optimal or approached solutions.

Challenges

In the general case, there is **no analytical solution** to the problem.

Even if there is no analytical solution, **computational power** can be used to compute optimal or approached solutions.

But:

- even if it is finite, the search space can be **huge**
- alternatives **enumeration** (to verify them, compare them or find the best) can be impossible in practice
- sometimes, even **finding a solution** can be difficult!

Example: the Traveling Salesman Problem (TSP)

Input

- a set of towns to be visited by a salesman
- the distances between the towns

Objective

Find a **circuit** with **minimal length** visiting each town exactly one time (**Hamiltonian circuit**).

Example: the Traveling Salesman Problem (TSP)

Input

- a set of towns to be visited by a salesman
- the distances between the towns

Objective

Find a **circuit** with **minimal length** visiting each town exactly one time (**Hamiltonian circuit**).

If you have n towns, the size of the space search is $(n - 1)!$.

Are TSP solution enumerable?

Optimistic hypothesis: 10^{-12} s to compute and evaluate a path.

# of towns	Computation time (s)
10	$3.63 \cdot 10^{-6}$
15	1.31
20	2432902
30	265252859812191058636

Some intuitions...

- $2432902 \text{ s} \approx 0.77 \text{ year}$
 - $265252859812191058636 \text{ s} \approx 84111130077 \text{ millenia} = \text{more than } \mathbf{500 \text{ times the age of the Universe!}}$
- ➡ combinatorial explosion

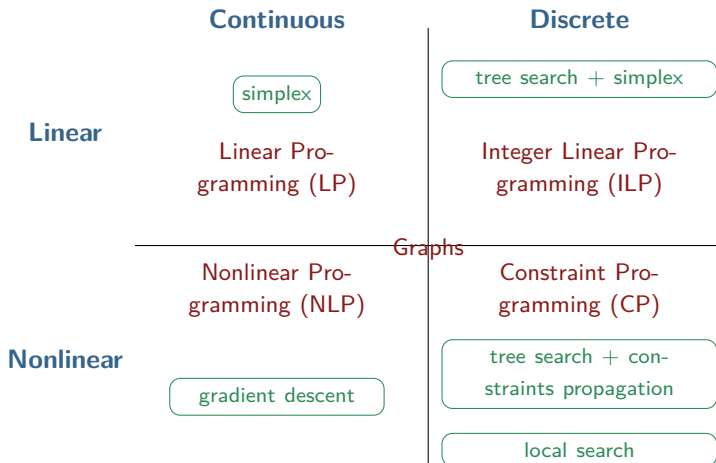
How to deal with combinatorial explosion?

Challenge

Produce optimal or approached solution **without exploring the complete search space**.

- dedicated algorithms: efficient, but costly
- use/adapt existing algorithms and tools
- use a **generic modelling framework**, with **generic resolution algorithms**

Some modelling framework and resolution methods



- 1 Introduction to combinatorial optimization
- 2 Complexity theory**
- 3 The P complexity class
- 4 The NP and NPC classes

Motivation

Observation from previous examples:

- some **algorithms** seem to be **more efficient** than others
- for a given problem, some **instances** seem to be **more difficult to solve** than others
- some **problems** seem to be more difficult than others

Motivation

Observation from previous examples:

- some **algorithms** seem to be **more efficient** than others
- for a given problem, some **instances** seem to be **more difficult to solve** than others
- some **problems** seem to be more difficult than others

Objective

- build a **theory** that **explains** and **predicts** such phenomena
- the theory must be **independent** from the **languages**, **compilers** or **machines** that are used

This is **complexity theory**.

Algorithms and problems

Do not mix algorithms, instances and problems!

Problem

A set of **instances** with the **same structure** on which you ask a question.

Instance

A problem with data.

Algorithm

A **procedure** taking an instance as **input** and answering the question of the problem as output.



Exercise

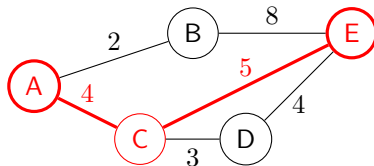
Can you “instantiate” the previous notions on the shortest path problem?



Exercise

Can you “instantiate” the previous notions on the shortest path problem?

- **problem:** find the shortest path between 2 vertices in a graph
- **instance:** find the shortest path between (A) and (E) on the following graph



- **(several) algorithms:** paths enumeration, Dijkstra’s algorithm, Bellman-Ford algorithm, Floyd-Warshall algorithm, Johnson’s algorithm etc.

Another (important) example: SAT



The **SAT** (for **satisfiability**) problem is an important problem for complexity theory.

Definition (clause)

Let $\{x_1, \dots, x_n\}$ be a set of boolean variables (i.e. $D_{x_i} = \{True, False\}$).

- **literal**: a variable x_i or the negation of a variable ($\neg x_i$)
 - **clause**: a disjunction of literals, e.g. $x_1 \vee \neg x_4 \vee x_5$
-
- **problem**: given a **finite set of clauses**, is the set **satisfiable** (i.e. can you assign variables to make all the clauses true)?
 - **instance**:
 - **algorithms**:



The **SAT** (for **satisfiability**) problem is an important problem for complexity theory.

Definition (clause)

Let $\{x_1, \dots, x_n\}$ be a set of boolean variables (i.e. $D_{x_i} = \{True, False\}$).

- **literal**: a variable x_i or the negation of a variable ($\neg x_i$)
 - **clause**: a disjunction of literals, e.g. $x_1 \vee \neg x_4 \vee x_5$
-
- **problem**: given a **finite set of clauses**, is the set **satisfiable** (i.e. can you assign variables to make all the clauses true)?
 - **instance**: is $\{x_1 \vee \neg x_4 \vee x_5, \neg x_1 \vee \neg x_5\}$ satisfiable?
 - **algorithms**: models enumeration, resolution algorithm, DPLL procedure etc.

Previously in TCS1-IN “Algorithms and programming”...

For a given algorithm $\mathcal{A}$:

- the **time complexity** of $\mathcal{A}$ is the time taken by $\mathcal{A}$ to answer
- the **space complexity** of $\mathcal{A}$ is the memory space taken by $\mathcal{A}$ to answer
- in practice, for time complexity:
 - the **worst-case** complexity is often used. The **maximal number of executed operations** for an input with a **given size** is computed
 - a **cost model** is chosen (the number of assignments, of array accesses etc.)
 - **asymptotic analysis** is used (mainly O notation)



Sorting an array with n elements

- selection sort:
- mergesort:

Shortest path in a graph with n vertices and m edges

- path enumerations:
- Dijkstra's algorithm (with simple implementation):

SAT with n variables with models enumeration

- satisfiable instance: sometimes immediate, sometimes 2^n
 - unsatisfiable instance: always 2^n
- ➡ time complexity:



Sorting an array with n elements

- selection sort: $O(n^2)$
- mergesort: $O(n \cdot \log n)$

Shortest path in a graph with n vertices and m edges

- path enumerations: $O(n!)$
- Dijkstra's algorithm (with simple implementation): $O(m + n^2)$

SAT with n variables with models enumeration

- satisfiable instance: sometimes immediate, sometimes 2^n
 - unsatisfiable instance: always 2^n
- ➡ time complexity: $O(2^n)$

Time complexity: some numbers

Complexity	Size of the biggest problem solvable with 10^6 instructions	Size of the biggest problem solvable with 10^{12} instructions
n	1 000 000	1 000 000 000 000
$n \cdot \log n$	64 000	32 000 000 000
n^2	1 000	1 000 000
n^3	100	10 000
2^n	20	40

Time complexity of a problem

Definition

The **time complexity** of a **problem** is the time complexity of the **best algorithm** solving the problem.

For instance, the time complexity of the array sorting problem is $O(n \cdot \log n)$ with n being the size of the array.

Time complexity of a problem

Definition

The **time complexity** of a **problem** is the time complexity of the **best algorithm** solving the problem.

For instance, the time complexity of the array sorting problem is $O(n \cdot \log n)$ with n being the size of the array.

- **polynomial problem**: easy, **tractable** problem for which there is a polynomial algorithm (**beware of degree and constants of the polynomial!**)
- **exponential problem**: hard, **intractable** problem

Some polynomial problems

- **sorting** problem
- **shortest path** in a weighted graph
- computing the **inverse** of an invertible matrix
- **linear programming**
- **primality** of an integer
- **minimum spanning tree** in a weighted graph
- **maximum flow** in a weighted graph
- **assignment problem**: maximum matching in a bipartite graph
- ...

Some exponential problems

Some problems for which the best known algorithm is exponential:

- **SAT**
- **Hamiltonian path**
- **Traveling Salesman** problem
- **graph coloring**
- **knapsack** problem
- **integer linear programming**
- ...

Some exponential problems

Some problems for which the best known algorithm is exponential:

- **SAT**
- **Hamiltonian path**
- **Traveling Salesman** problem
- **graph coloring**
- **knapsack** problem
- **integer linear programming**
- ...

But are these problems **really exponential**?

- ➡ **partial** response: **computational complexity theory** (do not confuse with algorithm complexity!)
- ➡ **idea**: **complexity classes** for problems

Which problems?

We will focus on **decision problems**, i.e. problems for which the answer is either **yes** or **no**.

For instance:

- existence of an Hamiltonian path in a graph
- SAT

Which problems?

We will focus on **decision problems**, i.e. problems for which the answer is either **yes** or **no**.

For instance:

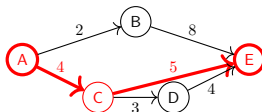
- existence of an Hamiltonian path in a graph
- SAT

And **optimization problems**, i.e. problems for which you want to optimize (minimize) a criterion?

- ➡ you can always associate a decision problem to the optimization problem: is there a solution for which the **criterion is less than a value k** ?
- ➡ **if the criterion is finite**, you can solve the optimization problem by **solving a finite number** of associated decision problems

Optimization and decision: example

- **problem:** shortest path between two vertices of a graph
- **associated decision problem:** is there a path with length less than k between the two vertices ($k \leq \sum_{e \in \text{edges}} d_e$)?



- **use the decision problem to solve the optimization one:**
 - let $k = 27$. Is there a path with length strictly less than 27?
Yes (ABE)
 - let $k = 26$...
 - ...
 - let $k = 9$. Is there a path with length strictly less than 9?
No: the shortest path has a length of 9.

Outline

- 1 Introduction to combinatorial optimization
- 2 Complexity theory
- 3 The P complexity class**
- 4 The NP and NPC classes

Definition

The P class (**Polynomial class**) is the class of decision problems that are **polynomial**.

Remember: this means that **there is at least one algorithm with polynomial-time complexity that solves the problem.**

Definition

The P class (**Polynomial class**) is the class of decision problems that are **polynomial**.

Remember: this means that **there is at least one algorithm with polynomial-time complexity that solves the problem**.

How to show that a problem is in P?

➡ an obvious approach: find a polynomial algorithm...



Exercise

Show that the following problem is in P: considering a graph with n vertices and m edges, is there a path between two vertices?



Exercise

Show that the following problem is in P: considering a graph with n vertices and m edges, is there a path between two vertices?

Easy: you can use

- depth-first search
- breadth-first search
- even Dijkstra's algorithm!

A more complicated example

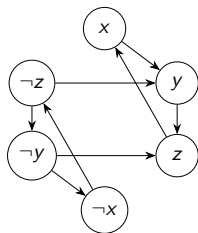
Theorem

*The **2-SAT** problem (the SAT problem in which all clauses have at most 2 literals) is in P.*

A more complicated example

Finding a polynomial algorithm is not so easy... You may represent the problem with a graph.

$$\{\neg x \vee y, \neg y \vee z, x \vee \neg z, z \vee y\}$$



- if there is a x s.t. there are a path from x to $\neg x$ and a path from $\neg x$ to x , φ is **UNSAT**
- otherwise, φ is **SAT**

You can of course reuse the results on the previous problem about vertex reachability!

Polynomial reduction

Finding a polynomial algorithm for a problem may be difficult. There is also another approach: **polynomial reduction**.

Definition

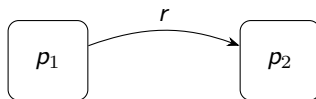
A **polynomial reduction** r of a problem p_1 to a problem p_2 is an **algorithm** with **polynomial-time complexity** that associates to each instance i_1 of p_1 an instance i_2 of p_2 such that the answers to i_1 and i_2 are **identical**.

If there is a polynomial reduction from p_1 to p_2 , then $p_1 \leq_P p_2$ (p_2 is at least as difficult as p_1).

Theorem

Let p_1 and p_2 be problem s.t. $p_1 \leq_P p_2$. If $p_2 \in P$, then $p_1 \in P$.

Polynomial reduction



Remember:

- p_2 is **at least as difficult as** p_1
- therefore, if p_2 is “easy to solve”, then is also p_1



Exercise

Let consider the 2-coloring problem: is it possible to color the vertices of a graph with only 2 colors s.t. two adjacent vertices do not have the same color? Show that this problem is in P.



Exercise

Let consider the 2-coloring problem: is it possible to color the vertices of a graph with only 2 colors s.t. two adjacent vertices do not have the same color? Show that this problem is in P.

We can exhibit a polynomial reduction from 2-coloring to **2-SAT**:

- transform 2-coloring into 2-SAT:
 - for each vertex v_i of the graph, create one boolean variable x_i
 - for each edge (v_i, v_j) of the graph, add two clauses $x_i \vee x_j$ and $\neg x_i \vee \neg x_j$
- how to go from a solution of 2-SAT to a solution of 2-coloring:
 - if x_i is true, then v_i has the first color
 - if x_i is false, then v_i has the second color
- we can show that answering “is the obtained 2-SAT problem satisfiable?” is equivalent to answering “is the graph 2-colorable?”
- this reduction is **polynomial**, therefore **2-coloring** $\leq_P$ **2-SAT**

Therefore **2-coloring** $\in$ **P**.

- 1 Introduction to combinatorial optimization
- 2 Complexity theory
- 3 The P complexity class
- 4 The NP and NPC classes**

What does “NP” stand for?

How many among you think that NP means “Non Polynomial”?
Be honest 😊.

The NP class: Turing again!

Definition

A problem p is in the NP class if it can be solved by a **nondeterministic Turing machine** in **polynomial time** (hence the name NP).

Hu? We have not talked about Turing machines!

➡ they are in fact foundations for complexity theory

The NP class: a more intuitive view

Definition (more intuitive)

A problem p is in the NP class if **verifying** if an alternative is a solution can be done in **polynomial time** (existence of a **polynomial verifier**).

If p is a problem in NP, there is a verifier V , such that for every instance i_p of p :

- if the answer to i_p is YES, there is a solution (also called a **certificate** or a **witness**) W such that V returns YES in polynomial time for the input (i_p, W)
 - ➡ V can verify in polynomial time that W is a correct solution to the problem
- if the answer to i_p is NO, then for all possible certificates W , V returns NO in polynomial time for the input (i_p, W)

P and NP: intuition

An intuitive view:

- P: the class of problems for which **finding a solution is easy**
- NP: the class of problems for which **verifying a solution is easy**

P and NP: intuition

An intuitive view:

- P: the class of problems for which **finding a solution is easy**
- NP: the class of problems for which **verifying a solution is easy**

Exercise

Is $P \subseteq NP$?

P and NP: intuition

An intuitive view:

- P: the class of problems for which **finding a solution is easy**
- NP: the class of problems for which **verifying a solution is easy**

Exercise

Is $P \subseteq NP$?

Yes, because if a problem p is in P, there is a polynomial algorithm $\mathcal{A}$ that solves p .

The verifier for p uses simply $\mathcal{A}$ (ignore the certificate and answer the question on p).

Some problems in NP

- **SAT**: the certificate is the boolean valuation of variables and the verifier is provided by boolean computation
- **Traveling Salesman Problem**: is there a cycle going through all vertices of a weighted graph with a total cost less than k ?

Certificate:

Verifier:

- **k-coloring**: same as 2-coloring but with k colors

Certificate:

Verifier:

Some problems in NP

- **SAT**: the certificate is the boolean valuation of variables and the verifier is provided by boolean computation
- **Traveling Salesman Problem**: is there a cycle going through all vertices of a weighted graph with a total cost less than k ?
Certificate: the list of towns to travel in a specified order
Verifier: simply verify first that all towns are in the certificate and then compute the total cost of the certificate
- **k-coloring**: same as 2-coloring but with k colors
Certificate: the graph in which each node has a color
Verifier: simply consider each edge of the graph in the certificate and verify that its two ends do not have the same color

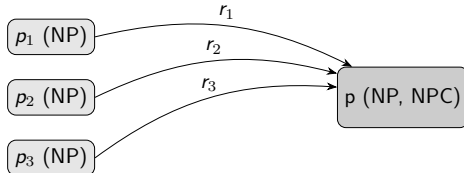
NP-completeness

Are there in NP problems that are more difficult than the others?

Definition

A problem p is NP-complete iff:

- $p \in \text{NP}$
- for **every** problem p' in NP, there is a **polynomial** reduction from p' to p .



p is at least as difficult as any problem in NP.

NP-completeness: why polynomial reductions?

If p is NP-complete, the reduction r from a problem p' in NP to p must be **polynomial in time**.

Otherwise, you may “hide” resolution steps of p' in r !

Notice that the size of the instance of p obtained with r should also be polynomial w.r.t. the size of the instance of p' .

How to prove that your problem is in NPC?

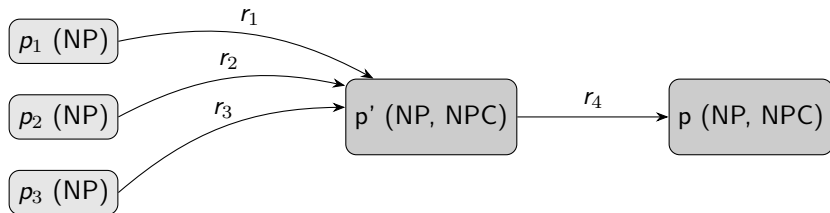
The hard way: direct proof of the hypothesis to verify to be in NPC.

Theorem (Cook, 1971)

SAT is NP-complete.

How to prove that your problem is in NPC (easier)?

To prove that p is in NPC, use a reduction!



It is sufficient to prove that:

- $p \in \text{NP}$
- there is a problem p' in NPC s.t. $p' \leq_P p$

Beware: you have to reduce p' to p , not p to p' !

Reduction: example of ILP

ILP (Integer Linear Programming)

Linear optimization problem with $z \in M_n(\mathbb{Z})$, $A \in M_{p,n}(\mathbb{Z})$, $b \in M_p(\mathbb{Z})$ and $c \in M_n(\mathbb{Z})$:

linear constraints $Az \leq b$

$$z \geq 0$$

linear function to optimize $\min c^T z$

Reduction: example of ILP

ILP (Integer Linear Programming)

Linear optimization problem with $z \in M_n(\mathbb{Z})$, $A \in M_{p,n}(\mathbb{Z})$, $b \in M_p(\mathbb{Z})$ and $c \in M_n(\mathbb{Z})$:

linear constraints $Az \leq b$

$$z \geq 0$$

linear function to optimize $\min c^T z$

k-ILP

Associated **decision problem**: for a given $k \in \mathbb{Z}$, is there a solution for $c^T z \leq k$?

Reduction: is ILP NP-complete?

Question: is k-ILP NP-complete?

- k-ILP $\in$ NP (for a given z , verifying that $c^T z \leq k$ and that z respects the constraints can be done in polynomial time)

Reduction: is ILP NP-complete?

Question: is k-ILP NP-complete?

- k-ILP $\in$ NP (for a given z , verifying that $c^T z \leq k$ and that z respects the constraints can be done in polynomial time)
- to show that k-ILP is NP-complete, we will exhibit a reduction $r_{SAT \rightarrow k-ILP}$ from **SAT** to **k-ILP**:

Reduction: is ILP NP-complete?

Question: is k-ILP NP-complete?

- k-ILP $\in$ NP (for a given z , verifying that $c^T z \leq k$ and that z respects the constraints can be done in polynomial time)
- to show that k-ILP is NP-complete, we will exhibit a reduction $r_{SAT \rightarrow k-ILP}$ from **SAT** to **k-ILP**:
 - for each var. x_i of the SAT instance, create a integer var. z_i with domain $\{0, 1\}$

Reduction: is ILP NP-complete?

Question: is k-ILP NP-complete?

- k-ILP $\in$ NP (for a given z , verifying that $c^T z \leq k$ and that z respects the constraints can be done in polynomial time)
- to show that k-ILP is NP-complete, we will exhibit a reduction $r_{SAT \rightarrow k-ILP}$ from **SAT** to **k-ILP**:
 - for each var. x_i of the SAT instance, create a integer var. z_i with domain $\{0, 1\}$
 - let f be the function s.t. $f(x_i) = z_i$ and $f(\neg x_i) = 1 - z_i$

Reduction: is ILP NP-complete?

Question: is k-ILP NP-complete?

- k-ILP $\in$ NP (for a given z , verifying that $c^T z \leq k$ and that z respects the constraints can be done in polynomial time)
- to show that k-ILP is NP-complete, we will exhibit a reduction $r_{SAT \rightarrow k-ILP}$ from **SAT** to **k-ILP**:
 - for each var. x_i of the SAT instance, create a integer var. z_i with domain $\{0, 1\}$
 - let f be the function s.t. $f(x_i) = z_i$ and $f(\neg x_i) = 1 - z_i$
 - for each clause $C_j = l_{j,1} \vee l_{j,2} \vee \dots \vee l_{j,k}$ of the SAT instance, we will consider a constraint $\sum_{i=1}^k f(l_{j,i}) \geq 1$ representing the fact that C_j is satisfiable.

Reduction: is ILP NP-complete?

Question: is k-ILP NP-complete?

- k-ILP $\in$ NP (for a given z , verifying that $c^T z \leq k$ and that z respects the constraints can be done in polynomial time)
- to show that k-ILP is NP-complete, we will exhibit a reduction $r_{SAT \rightarrow k-ILP}$ from **SAT** to **k-ILP**:
 - for each var. x_i of the SAT instance, create a integer var. z_i with domain $\{0, 1\}$
 - let f be the function s.t. $f(x_i) = z_i$ and $f(\neg x_i) = 1 - z_i$
 - for each clause $C_j = l_{j,1} \vee l_{j,2} \vee \dots \vee l_{j,k}$ of the SAT instance, we will consider a constraint $\sum_{i=1}^k f(l_{j,i}) \geq 1$ representing the fact that C_j is satisfiable.
 - each line A_j of A and component b_j of b are filled using the previous constraint $-\sum_{i=1}^k f(l_{j,i}) \leq -1$

Reduction: is ILP NP-complete?

Question: is k-ILP NP-complete?

- $k\text{-ILP} \in \text{NP}$ (for a given z , verifying that $c^T z \leq k$ and that z respects the constraints can be done in polynomial time)
- to show that $k\text{-ILP}$ is NP-complete, we will exhibit a reduction $r_{\text{SAT} \rightarrow k\text{-ILP}}$ from **SAT** to **k-ILP**:
 - for each var. x_i of the SAT instance, create a integer var. z_i with domain $\{0, 1\}$
 - let f be the function s.t. $f(x_i) = z_i$ and $f(\neg x_i) = 1 - z_i$
 - for each clause $C_j = l_{j,1} \vee l_{j,2} \vee \dots \vee l_{j,k}$ of the SAT instance, we will consider a constraint $\sum_{i=1}^k f(l_{j,i}) \geq 1$ representing the fact that C_j is satisfiable.
 - each line A_j of A and component b_j of b are filled using the previous constraint $-\sum_{i=1}^k f(l_{j,i}) \leq -1$
 - the SAT instance is satisfiable iff there is a z such that $Az \leq b$

Reduction: is ILP NP-complete?

Question: is k-ILP NP-complete?

- k-ILP $\in$ NP (for a given z , verifying that $c^T z \leq k$ and that z respects the constraints can be done in polynomial time)
- to show that k-ILP is NP-complete, we will exhibit a reduction $r_{SAT \rightarrow k-ILP}$ from **SAT** to **k-ILP**:
 - for each var. x_i of the SAT instance, create a integer var. z_i with domain $\{0, 1\}$
 - let f be the function s.t. $f(x_i) = z_i$ and $f(\neg x_i) = 1 - z_i$
 - for each clause $C_j = l_{j,1} \vee l_{j,2} \vee \dots \vee l_{j,k}$ of the SAT instance, we will consider a constraint $\sum_{i=1}^k f(l_{j,i}) \geq 1$ representing the fact that C_j is satisfiable.
 - each line A_j of A and component b_j of b are filled using the previous constraint $-\sum_{i=1}^k f(l_{j,i}) \leq -1$
 - the SAT instance is satisfiable iff there is a z such that $Az \leq b$
 - notice that $c^T z \leq k$ does not participate in the proof, we just have to choose c and k s.t. the constraint is trivially true (e.g. $c = \vec{0}$ and $k = 0$)
 - ➡ this is normal, SAT is not an optimization problem

Reduction from SAT to ILP: example

Here is an example of the previous reduction. Consider the following SAT problem:

$$\{x_1 \vee \neg x_2 \vee x_4, x_3 \vee x_4\}$$

Reduction from SAT to ILP: example

Here is an example of the previous reduction. Consider the following SAT problem:

$$\{x_1 \vee \neg x_2 \vee x_4, x_3 \vee x_4\}$$

The constraints associated to the two clauses are the following:

$$\begin{aligned} z_1 + (1 - z_2) + z_4 &\geq 1 \\ z_3 + z_4 &\geq 1 \end{aligned}$$

Reduction from SAT to ILP: example

Here is an example of the previous reduction. Consider the following SAT problem:

$$\{x_1 \vee \neg x_2 \vee x_4, x_3 \vee x_4\}$$

The constraints associated to the two clauses are the following:

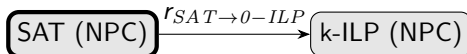
$$\begin{aligned} z_1 + (1 - z_2) + z_4 &\geq 1 \\ z_3 + z_4 &\geq 1 \end{aligned}$$

Finally, the matrices A and b are:

$$A = \begin{bmatrix} -1 & 1 & 0 & -1 \\ 0 & 0 & -1 & -1 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

Reduction: k-ILP is NP-complete

- k-ILP is in NP
- the SAT problem can be reduced to 0-ILP and SAT is NP-complete
- therefore the decision problem associated to ILP is NP-complete





Exercise

You are now an engineer responsible of a geostationary satellite that should take images of ground objects. Maneuvering to change the target of your satellite from another target costs some fuel. You start from an initial position and should finish your maneuvers by returning to this initial position.

You want to minimize the global amount of fuel used to take all images.

- How can you model this problem with a problem presented previously?
- What decision problem should you use to solve this problem?
- What is its complexity class? How can you prove it?
You may use the fact that the Hamiltonian circuit decision problem is NP-complete



The previous problem can be viewed as an instance of the **TSP** problem.

Recall that in the TSP problem:

- we consider n towns, each pair of towns being associated with a distance
- we want to find a circuit (i.e. a path starting from a town and returning to this town) going through all towns exactly once
- we want to minimize the global distance of the circuit



The previous problem can be viewed as an instance of the **TSP** problem.

Recall that in the TSP problem:

- we consider n towns, each pair of towns being associated with a distance
- we want to find a circuit (i.e. a path starting from a town and returning to this town) going through all towns exactly once
- we want to minimize the global distance of the circuit

We can consider here:

- the targets of the satellite as the towns to be visited
- the maneuvers costs between two targets as the distance between the corresponding towns



The previous problem can be viewed as an instance of the **TSP** problem.

Recall that in the TSP problem:

- we consider n towns, each pair of towns being associated with a distance
- we want to find a circuit (i.e. a path starting from a town and returning to this town) going through all towns exactly once
- we want to minimize the global distance of the circuit

We can consider here:

- the targets of the satellite as the towns to be visited
- the maneuvers costs between two targets as the distance between the corresponding towns

Solving such an instance of the TSP solves our problem.



The TSP problem is an **optimization problem** (we want to minimize the global cost of the tour).

The associated decision problem is therefore the following: “given an instance of the TSP problem with n towns and the associated distances between each pair of towns and a given k , is there a tour visiting each towns exactly once with a total cost less than k ?”.



In order to prove that the TSP decision problem is NP-complete, we should:

- ❶ prove that TSP is in NP (should not be too difficult)
- ❷ find a NP-complete problem P and a **polynomial** reduction $r_{P \rightarrow \text{TSP}}$ from P to TSP s.t. given an instance p of P :
 - if p has a solution, then $r_{P \rightarrow \text{TSP}}(p)$ has also a solution
 - if $r_{P \rightarrow \text{TSP}}(p)$ has a solution, then p has also a solution

Again, the fact that the reduction is **polynomial** both in time and space is **crucial**, otherwise you may “hide” resolution steps in the reduction!



Hint: use the fact that the Hamiltonian circuit decision problem is NP-complete.

Hamiltonian circuit: given a **not necessary complete** graph G_H with no weights, is there a circuit visiting each vertex of the graph exactly once?



Hint: use the fact that the Hamiltonian circuit decision problem is NP-complete.

Hamiltonian circuit: given a **not necessary complete** graph G_H with no weights, is there a circuit visiting each vertex of the graph exactly once?

The TSP problem can be modeled by a graph G_{TSP} :

- each town of the TSP is a vertex of the graph
- for each pair of towns with distance d , there is an edge G_{TSP} between the corresponding vertices with weight d



Hint: use the fact that the Hamiltonian circuit decision problem is NP-complete.

Hamiltonian circuit: given a **not necessary complete** graph G_H with no weights, is there a circuit visiting each vertex of the graph exactly once?

The TSP problem can be modeled by a graph G_{TSP} :

- each town of the TSP is a vertex of the graph
- for each pair of towns with distance d , there is an edge G_{TSP} between the corresponding vertices with weight d

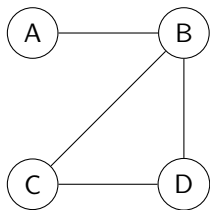
Question: is there a polynomial reduction $r_{H \rightarrow \text{TSP}}$?



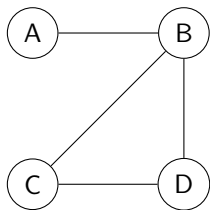
Problem: the graph in the Hamiltonian problem may not be complete, whereas the graph representation of the TSP is complete.

Given an instance of the Hamiltonian problem with graph G_H composed of n vertices, let us consider the following reduction $r_{H \rightarrow TSP}$ building an instance of TSP modeled with graph G_{TSP} :

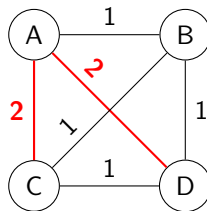
- the vertices of G_{TSP} are the ones of G_H
- for an edge e in G_H , there is a corresponding edge in G_{TSP} with a weight of 1
- an edge e in G_{TSP} that does not appear in G_H has a weight of 2
- the upper bound in the decision version of the TSP problem must be n



Hamiltonian circuit instance



Hamiltonian circuit instance



TSP instance with upper bound for length equals to 4

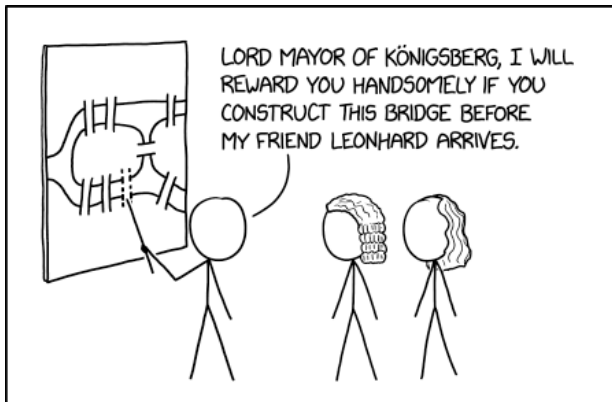


The TSP decision problem is **NP-complete**:

- you can verify that an alternative is a solution in polynomial time
- $r_{H \rightarrow \text{TSP}}$ is a correct reduction. Consider an instance h of the Hamiltonian circuit problem:
 - if h has a solution, then $r_{H \rightarrow \text{TSP}}(h)$ has also a solution: consider the edges in $r_{H \rightarrow \text{TSP}}(h)$ corresponding to the solution of h . They clearly define a tour in the TSP with length $\leq n$.
 - if $r_{H \rightarrow \text{TSP}}(h)$ has a solution, then h has also a solution: the edges in the solution of $r_{H \rightarrow \text{TSP}}(h)$ must have an associated weight of 1, otherwise the global cost would be $> n$ (there are exactly n edges in the tour). As edges in $G_{r_{H \rightarrow \text{TSP}}(h)}$ with weight 1 have a counterpart in G_H , we can build a solution for h .
 - $r_{H \rightarrow \text{TSP}}$ is polynomial: transforming a given instance of the Hamiltonian problem into the corresponding TSP problem can be done in polynomial time.

Notice that the size of the graph associated to the TSP problem is also polynomial w.r.t. the size of the graph of the Hamiltonian problem.

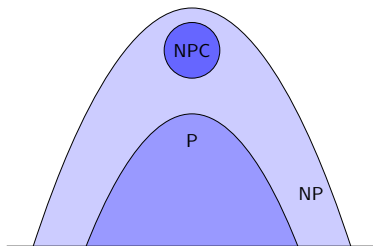
A simpler solution



I TRIED TO USE A TIME MACHINE TO CHEAT ON MY ALGORITHMS
FINAL BY PREVENTING GRAPH THEORY FROM BEING INVENTED.

Relation between P, NP and NPC

Complexity hierarchy:



- for some problems in NP, we do not know if they are in P
- if **one** problem in P is also in NPC, then **$P = NP$**
- there is a \$1 000 000 prize for this question...
- **conjecture:** $P \neq NP$

Beyond P and NP...

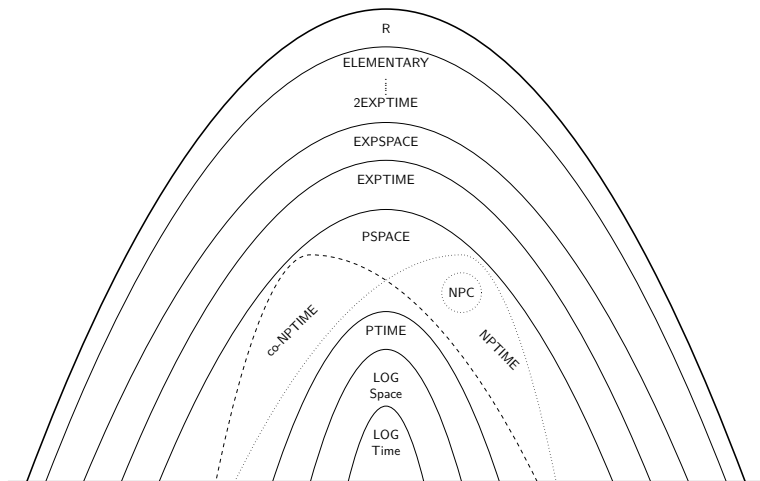


Image by [S. Sardina](#) based on [4].

Conclusion

What you should remember:

- algorithms complexity $\neq$ problems complexity
- if your problem is in P:
 - it means that your problem can be **solved in polynomial time**
 - it is a rather **good news**, but **beware of coefficients and degree**
- if your problem is in NPC, it is **hard** to solve, do not expect a polynomial algorithm, but:
 - the complexity is a **worst case** complexity
 - the **size** of the instances you deal with may be **small**
 - a **subproblem** of your problem may be in P
 - you may use **incomplete algorithms** to solve your problem
- if your problem is in NP, but not in P nor NPC, this is an uncertain situation...

Back to combinatorial optimization

Remember that:

- Linear Programming is a problem in P
 - ➡ (relatively) easy to solve
- Integer Linear Programming is a NP-complete problem
 - ➡ some instances are not so difficult to solve
 - ➡ other ones may be really difficult to solve

In the remaining of the course, we will be interested in

- finding efficient methods to solve ILP problems
- solving optimization problems using ILP

References



Scott Aaronson et al.

The Complexity Zoo.

2016.

URL: https://complexityzoo.uwaterloo.ca/Complexity_Zoo.



Michael R. Garey and David S. Johnson.

Computers and Intractability; A Guide to the Theory of NP-Completeness.

New York, NY, USA: W. H. Freeman & Co., 1990.

ISBN: 0716710455.



Takehiro Ito et al.

“On the complexity of reconfiguration problems”.

In: **Theoretical Computer Science** 412.12 (2011), pp. 1054–1065.

ISSN: 0304-3975.

DOI: <https://doi.org/10.1016/j.tcs.2010.12.005>.



C.H. Papadimitriou.

Computational Complexity.

Theoretical computer science.

Addison-Wesley, 1994.