REDP 2021 - Session 4 (Spanning trees & complexity)

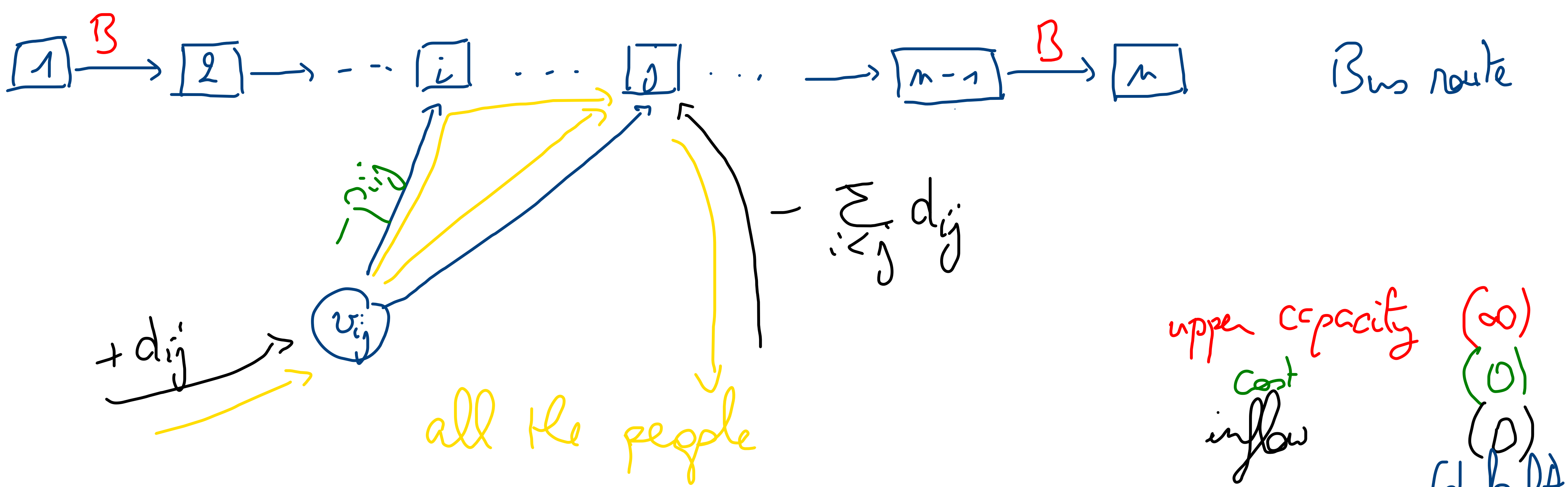
I/ Homework

A) Ex 6.10 (Bus)

Bus with B seats visiting cities 1 through n

$\forall\, i<j$, d_{ij} passengers want to go $i \to j$, with price p_{ij}

Minimum cost flow problem (max revenue) $\quad \sum_{v\in V} b_v \overset{!}{=} 0$ balance



One edge (v_{ij}, i) for people who take the bus

(v_{ij}, j) for disappointed people who take something else than bus

Last step : prove that the 2 pbs are equivalent

⇒ Given a solution to the real life problem, we can construct a solution to the flow problem with the same value

⇒ ∼∼∼ flow problem, ∼∼∼ real life problem ∼∼∼

Read the full answer in the notes to have an example

B) Ex 6.13 (Taxi fleet)

We build a flow graph where taxis will "flow"

lower capacities (0) ≤ ... ≤ upper capacities

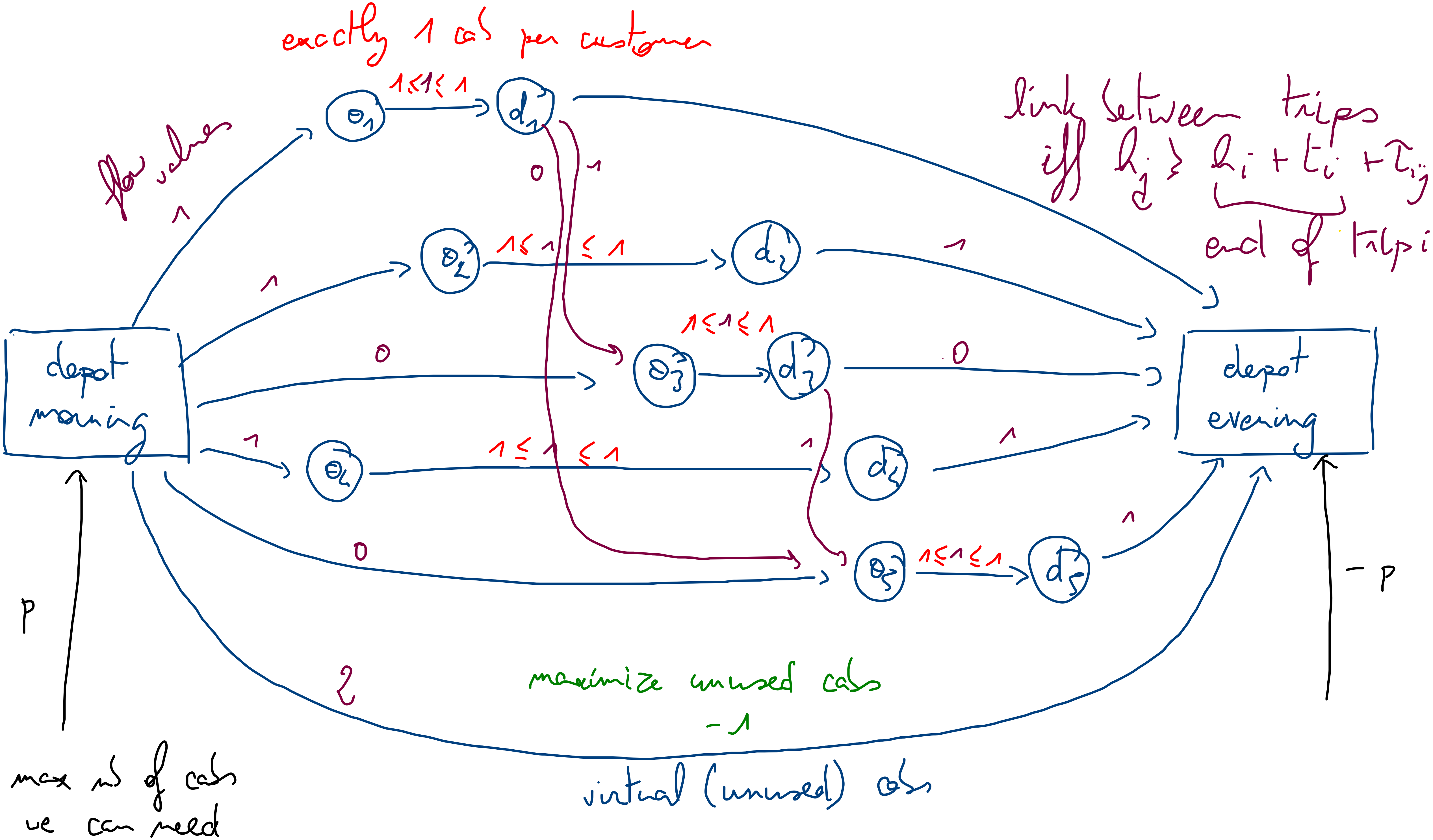
costs

inflows

The flow graph is defined after

⇒ Given a real car assignment, it is easy to deduce a feasible flow with the same cost

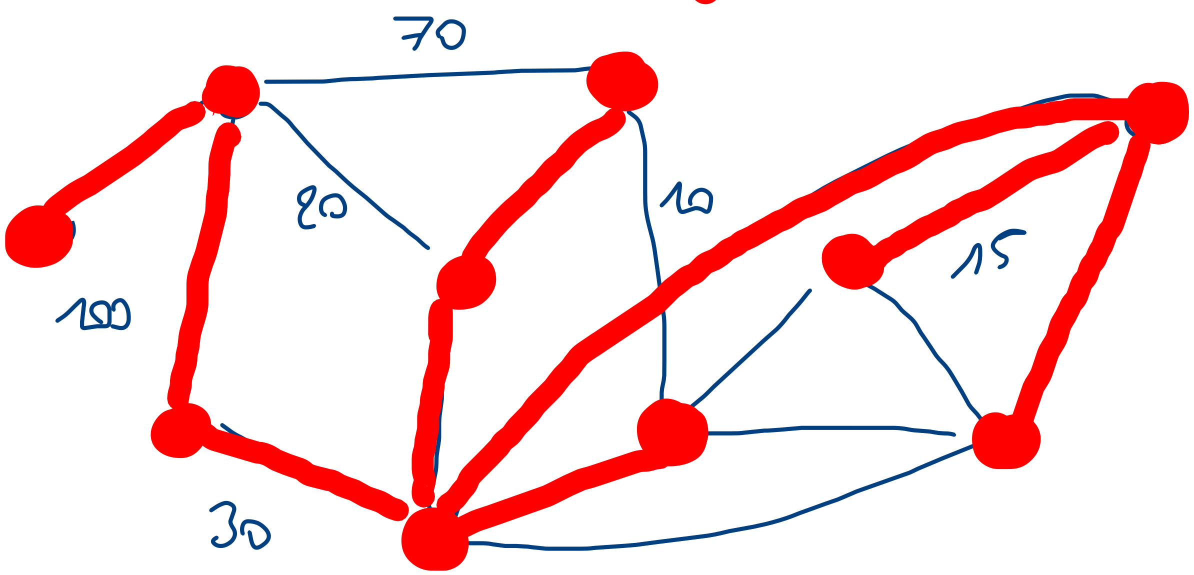
⇒ Given an <u>integer</u> flow, can we always deduce a car assignment?



Reconstructing the assignment relies on the fact that a (integer) flow can always be decomposed as a (integer) positive sum of flows along paths + flows along cycles

II/ Minimum spanning trees (MST)

A) Definitions



min cost electrical network
- must touch every vertex
- must be connected
- if we remove one edge, it shouldn't work anymore

→ Minimum Spanning Tree

A spanning subgraph of $G = (V, E)$ is a subgraph $H = (V', E')$ such that:
- $V' = V$
- E' is incident to all vertices in V

We will often confuse a subgraph H with its edge set E' in this lecture because we only consider those for which $V' = V$

If a spanning subgraph is a tree (connected, ~~no~~ forest cycles), we talk about a spanning tree

MST problem
- Input: an undirected connected graph G with edge weights $c(e)$
- Output: a spanning tree $T = (V, T)$ with min weight $\sum_{e \in T} c(e)$

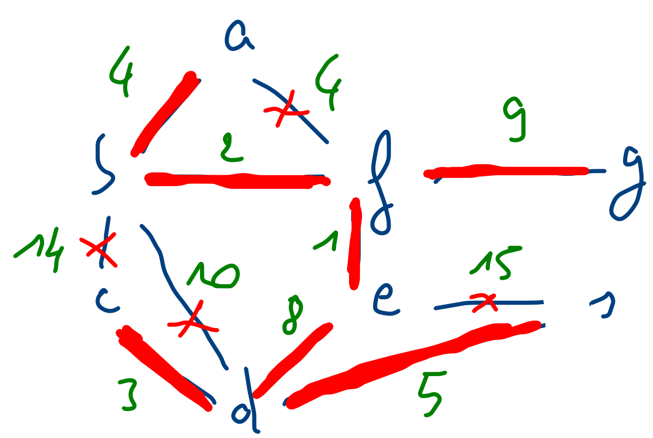
Naive alg: enumerate & compare trees $\longrightarrow$ there are too many

B) Kruskal's algorithm

1. Sort edge set E by increasing weight: $E = \{e_1, \dots, e_m\}$
   with $c(e_i) < c(e_j)$ if $i < j$
2. Start with $F_0 = \emptyset$
3. For every $i \in \{1, \dots, m\}$:
   - if $F_{i-1} \cup \{e_i\}$ has no cycles, add e_i: $F_i = F_{i-1} \cup \{e_i\}$
   - otherwise, $F_i = F_{i-1}$
4. Return $\mathcal{T} = (V, F_m)$

Greedy algorithm: make the best possible decision at each step regardless of the future ... but it works!

Ex 4.2:



$1 + 2 + 3 + 4 + 5 + 8 + 9 = 32$

Why does the algorithm work? Because, at each iteration, there is a minimum spanning tree T_i containing the forest F_i we are building

→ See my notes to do the proof as an exercise

c) Spanning tree polytope

The MST problem can be formulated as an Integer Linear Program:

Decision variable $x_e = \begin{cases} 1 & \text{if we select edge } e \text{ for the tree} \\ 0 & \text{otherwise} \end{cases}$

$$\min \sum_{e \in E} c(e)\, x_e \quad \text{s.t.}$$

(MST-ILP)

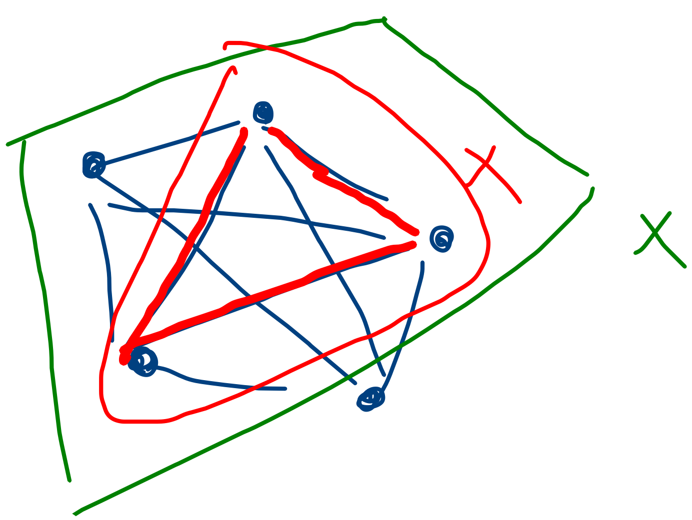
$$x_e \in \{0,1\}$$

$$\sum_{e \in E} x_e = |V| - 1$$

exponential nb of constraints

$$\sum_{e \in E[X]} x_e \leq |X| - 1 \qquad \forall X \subset V,\ X \neq \emptyset,\ X \neq V$$

↳ edges within the set of vertices X



MST - ILP is hard to solve because

1) Integer variables ⟶ Theorem 4.4 says we can replace $x_e \in \{0,1\}$ with $x_e \geq 0$ & still be sure that the optimal solutions returned by the simplex are integer-valued

2) $2^n - 2$ subsets X to consider in the constraints ⟶ Exercise 4.4 HOMEWORK

Remark: For the standard MST problem no need to use linear programming (⟶ Kruskal). Useful for more general versions (additional constraints)

III/ Complexity theory

See Session 1 for intuitive definitions of "problem" & "algorithm"

There are problems which computers cannot solve: undecidable

example: the halting problem (Turing) - decide if a program will terminate on a given input

In this class we focus on decidable problems & whether we can solve them efficiently

A) Formal definitions

Anything we can give to a computer can be described as a word x from a language X. A decision problem is a couple (X, Y) where

- X is a language called the input
- $x \in X$ is called an instance (a word : binary data for ex.)
- $Y \subset X$ contains all instances for which the answer is YES

A solution algorithm is a function $f : X \longrightarrow \{YES, NO\}$
s.t. $\forall x \in Y$, $f(x) = YES$
$\forall x \in X \setminus Y$, $f(x) = NO$

Ex:
Hamiltonian path problem : X = set of graphs (with a given encoding)
x = a particular graph
Y = set of graphs that have a ham. path

The alg. f runs in polynomial time if there is a polynomial P such that $\forall x$, $\text{runtime}(f, x) \leq P(\text{size}(x))$ ⟶ using binary encoding

Ex: Integer factorization / primality testing

To see whether n is prime, try to divide it by all integers from 2 to $\sqrt{n}$. In binary, $size(n) = \log_2(n)$

There are $\sqrt{n} = \sqrt{2^{\log_2 n}}$ iterations in this algorithm

$= 2^{s/2}$ (s is the size) $\longrightarrow$ exponential complexity

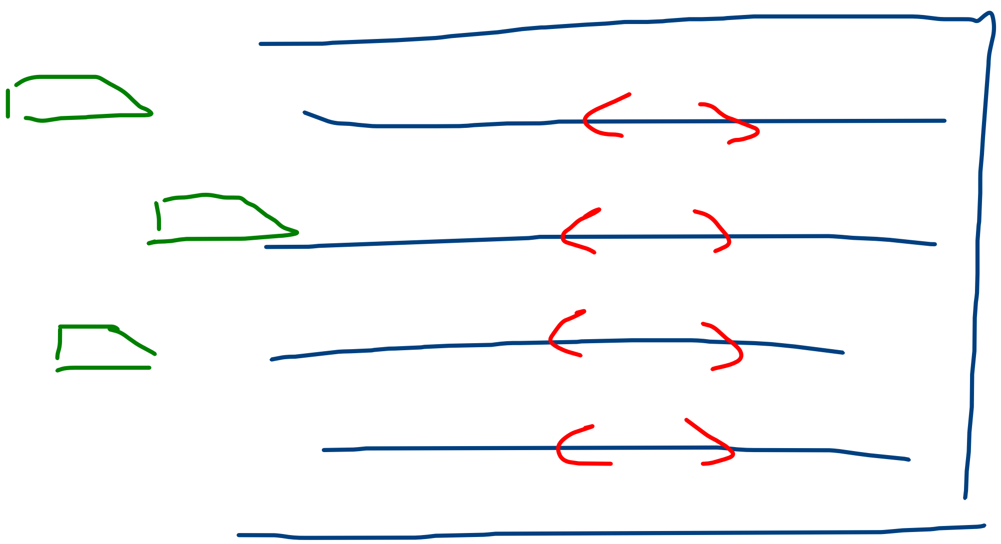
B) Complexity classes

Important to | separate "easy" & "hard" problems
| know which methods we can apply

$\longrightarrow$ exact algorithms

$\longrightarrow$ approximations / heuristics

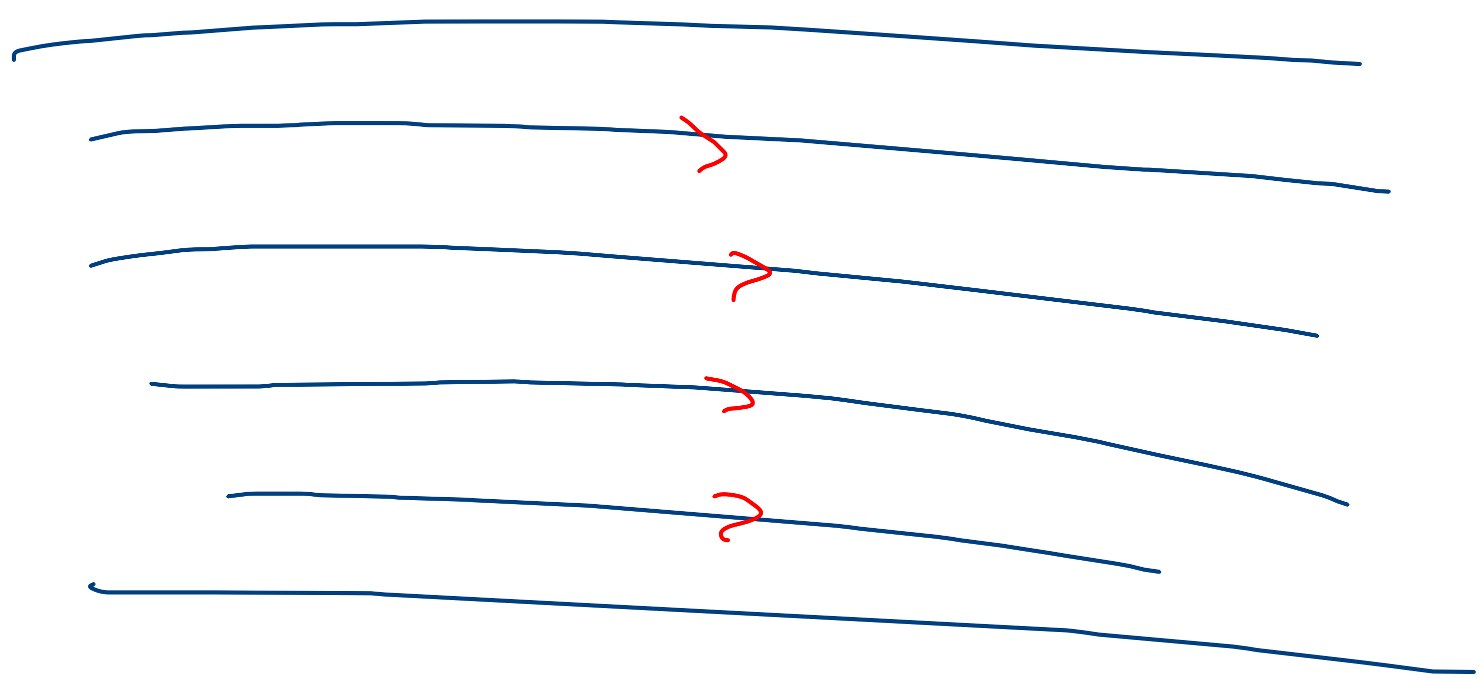
Easy	Hard
Shortest simple path, $w > 0$	Shortest simple path, $w < 0$
Eulerian cycle	Hamiltonian cycle
Minimum spanning tree	Minimum Steiner tree (spans a subset $S \subset V$)
Train shunting (3 var)	Train shunting (1 var)



α

α night: arrivals
then departures

α day: both arrivals
& departures
mixed



α

The class **P** contains all decision problems that have a polynomial solution algorithm

The class **NP** contains all decision problems for which there is a polynomial algorithm that verifies a solution, given a so-called certificate. We don't need to be able to find a certificate in polynomial time, only to check it with respect to the instance & the constraints.

Nondeterministic Polynomial

$P \subseteq NP$? True because, given an empty certificate, we can verify the solution (answer YES or NO) by generating it ourselves in polynomial time

Remarks:
- Polynomial time: independent of the machine
- Why no mention of memory?
  In Turing machines, memory is linked to time

C) Reductions

A reduction of problem $P = (X, Y)$ to $P' = (X', Y')$
is a function $red : X \longrightarrow X'$
such that $x \in Y \iff red(x) = x' \in Y'$

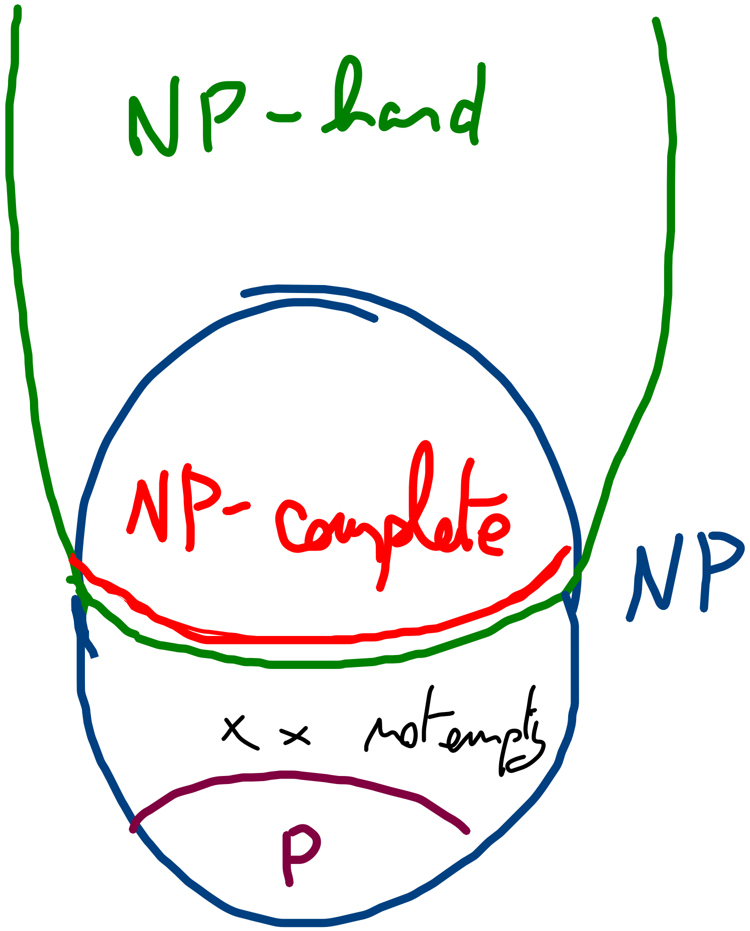
It is called polynomial reduction if red is a polynomial function

$$\begin{tikzcd}
X & x \arrow[r, "red"] & x' \arrow[d, "f'"] & X' \\
P & \{\underline{YES}, NO\} & \{\underline{YES}, NO\} \arrow[l, bend left] & P'
\end{tikzcd}$$

If $\exists$ a poly. reduction from X to X', it means X' is at least as hard as X

A decision pb P is **NP-hard** if every problem $Q \in$ **NP** reduces to P

———— **NP-complete** if 1) it is **NP-hard**
2) it belongs to **NP**



likely scenario

We don't know whether $\boxed{P \stackrel{?}{=} NP}$

It is likely that $P \not\subseteq NP$

Common task in OR:
prove that a problem is NP-hard

(NP-hard also applies to optimization problems, not just decision problems)

a/ Find a known NP-complete pb / NP-hard

b/ Reduce it polynomially to the one you study

To prove an optimization pb is NP-hard, you can study its decision version

TSP: find the shortest traveling salesman tour → harder

TSP-decision: is there a tour of length $\leq L$?

IV Homework

Ex 4.4.

Exercise: Prove that the shortest simple path pb with $w<0$ is NP-hard, using the fact that Hamiltonian path is NP-hard

Exercise: Prove that CLIQUE is NP-complete using the fact that 3-SAT is NP-complete

→ see description of 3-SAT in my lecture notes (end of week)

→ first proven NP-complete pb (Cook thm)

→ use the trick in the Wooclap slide