Advanced Algorithmics

Sophie Pinchinat
sophie.pinchinat@irisa.fr

IRISA, Université de Rennes 1

2024

Course outline

1. Recall on NP-completeness theory (in French)
2. Optimization and Decision problems
3. Bibliography
4. Backtracking
5. Branch-and-Bound
6. Approximation algorithms
7. Local search

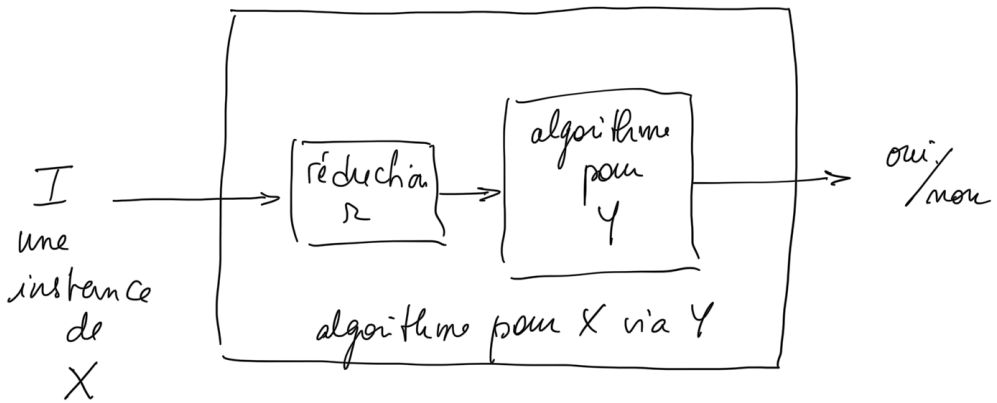
Réduction polynomiales

Definition (Réduction "à la Karp")

Le problème X se réduit au problème Y, noté $X \leq_P Y$, s'il existe une fonction effective $r(.)$ qui transforme toute instance I de X en une instance $r(I)$ de Y en temps polynomial de sorte que

$$I \in X \text{ si, et seulement si, } r(I) \in Y$$

On remarque que forcément $|r(I)| \leq p(|I|)$ pour un certain polynôme $p(.)$.



Réductions polynomiales

La notion de réduction polynomial permet de classifier les problèmes en fonction de leur difficulté relative.

Theorem (Conception d'algorithmes)

Si $X \leq_P Y$ et Y peut être résolu en temps polynomial, alors X peut être résolu en temps polynomial.

Theorem (Établir l'intractabilité)

Si $X \leq_P Y$ et X ne peut pas être résolu en temps polynomial, alors Y ne peut pas être résolu en temps polynomial.

Definition

On note $X \equiv_P Y$ lorsque $X \leq_P Y$ et $Y \leq_P X$

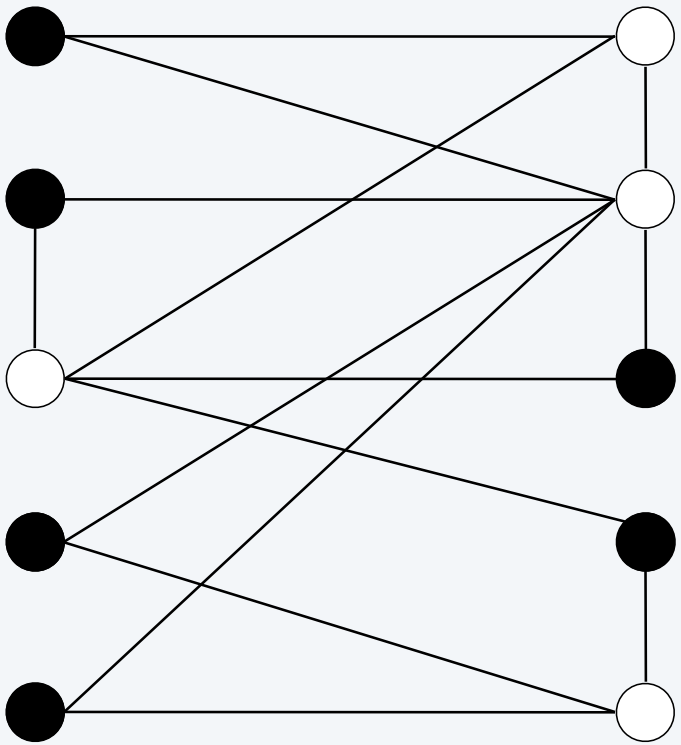
Theorem

Si $X \equiv_P Y$, alors
X peut être résolu en temps polynomial ssi Y peut être résolu en temps polynomial.

Packing and covering problems

Independent Set

INDEPENDENT-SET Étant donné un graphe $G = (V, E)$ et un entier k, existe-t-il un sous-ensemble de sommets $I \subseteq V$ tel que $|I| \geq k$ et qu'aucune paire de sommets dans I n'est reliée par un arc (Autrement dit, pour tous $u, v \in I, (u, v) \notin E$) ?

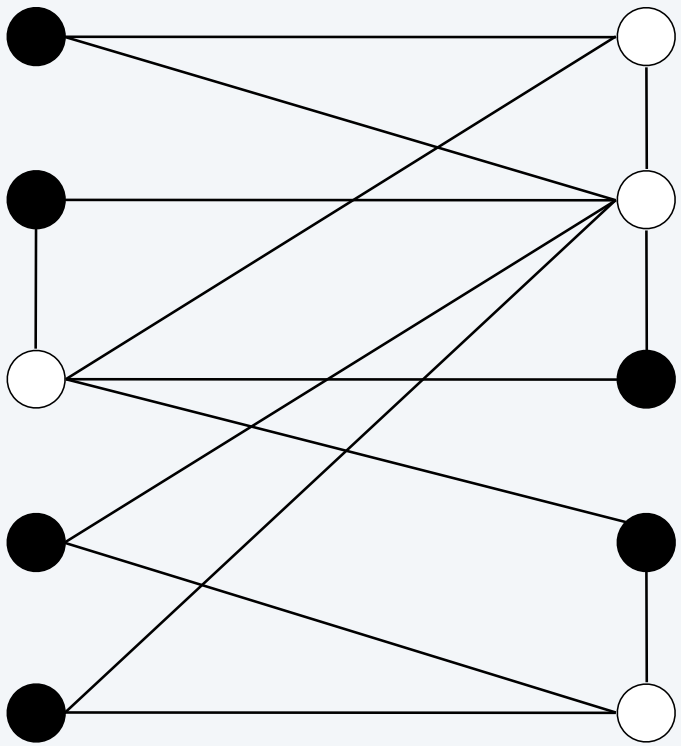


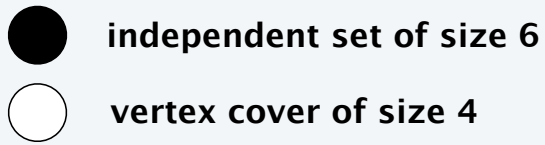
independent set of size 6

11

Vertex Cover (couverture de sommets)

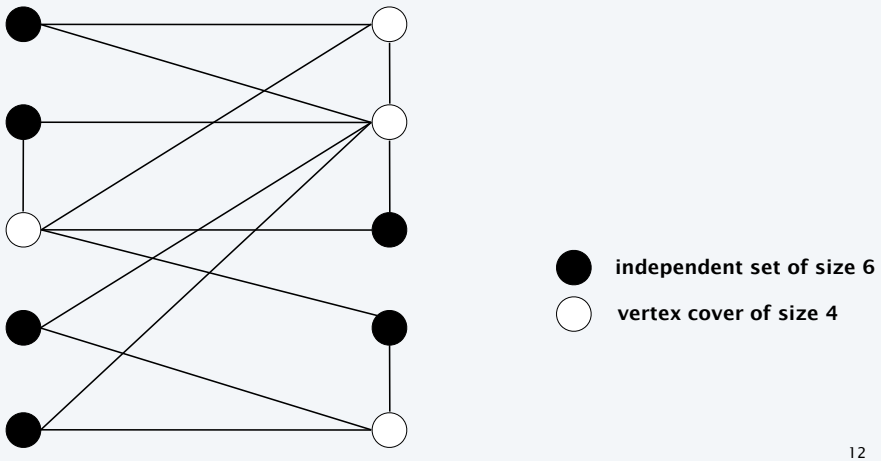
VERTEX-COVER Étant donné un graphe $G = (V, E)$ et un entier k, existe-t-il un sous-ensemble de sommets $S \subseteq V$ tel que $|S| \leq k$ et pour chaque arc, au moins une des deux extrémités est dans S (Autrement dit, pour tous $(u, v) \in E$, $u \in S$ ou $v \in S$) ?





12

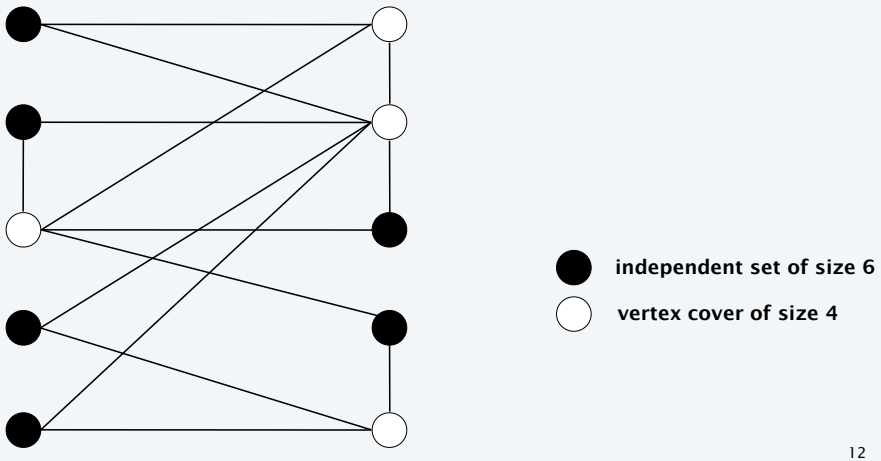
Vertex Cover et Independent Set se réduisent l'un à l'autre



Theorem

VERTEX-COVER $\equiv_P$ INDEPENDENT-SET

Vertex Cover et Independent Set se réduisent l'un à l'autre



Theorem

VERTEX-COVER $\equiv_P$ INDEPENDENT-SET

Proof.

Elle repose sur le fait suivant.

S est un Vertex Cover de cardinal k ssi $V \setminus S$ est un Independent Set de cardinal $n - k$.

□

Your turn!

Terminer la preuve du Théorème 6.

Vertex Cover $\leq_P$ Programmation Linéaire en Nombres Entiers (PLNE)

Étant donné un graphe $G = (V, E)$ et un entier k, on introduit l'ensemble de variables $X = \{x_v \mid v \in V\}$, avec l'interprétation que si $x_v = 1$, alors le sommet v est choisi dans le Vertex Cover, sinon $x_v = 0$.
On considère alors le systèmes $\mathcal{E}_G$ d'inéquations suivant.

$$\mathcal{E}_G \begin{cases} \sum_{v \in V} x_v \leq k & \\ x_u + x_v \geq 1 & (u, v) \in E \\ x_v \in \{0, 1\} & v \in V \end{cases}$$

Theorem

G a un Vertex Cover S de taille k ssi $\mathcal{E}_G$ a une solution telle que $\sum_{v \in V} x_v = k$.

Your turn!

Établir le Théorème 7.

Set Cover

SET-COVER Étant donné un ensemble $U = \{1, 2, \ldots, n\}$ d'éléments, une collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ de sous-ensembles de U, et un entier k, existe-t-il au plus k ensembles de $\mathcal{S}$ dont l'union est égale à U ?

Exemples d'application

- m modules logiciels.
- U un ensemble de n fonctionnalités que vous aimeriez que votre système ait.
- Le $i$ème module fournit l'ensemble $S_i \subseteq U$ de fonctionnalités.
- But : réaliser les n fonctionnalités en utilisant le moins de modules possible.

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_1 = \{ 3, 7 \}$ $S_4 = \{ 2, 4 \}$

$S_2 = \{ 3, 4, 5, 6 \}$ $S_5 = \{ 5 \}$

$S_3 = \{ 1 \}$ $S_6 = \{ 1, 2, 6, 7 \}$

$k = 2$

a set cover instance

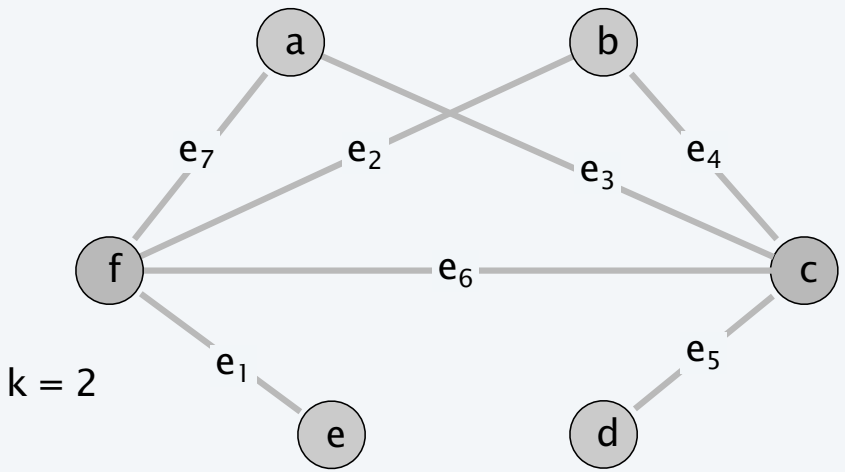
Vertex Cover se réduit à Set Cover

Theorem

VERTEX-COVER $\leq_P$ SET-COVER

Proof.

Étant donnée une instance $G = (V, E)$ (et k) de VERTEX-COVER, on construit l'instance $(U, \mathcal{S}, k)$ de SET-COVER de taille polynomial en la taille de G.



vertex cover instance (k = 2)

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 3, 7 \}$ $S_b = \{ 2, 4 \}$

$S_c = \{ 3, 4, 5, 6 \}$ $S_d = \{ 5 \}$

$S_e = \{ 1 \}$ $S_f = \{ 1, 2, 6, 7 \}$

set cover instance (k = 2)

Vertex Cover se réduit à Set Cover

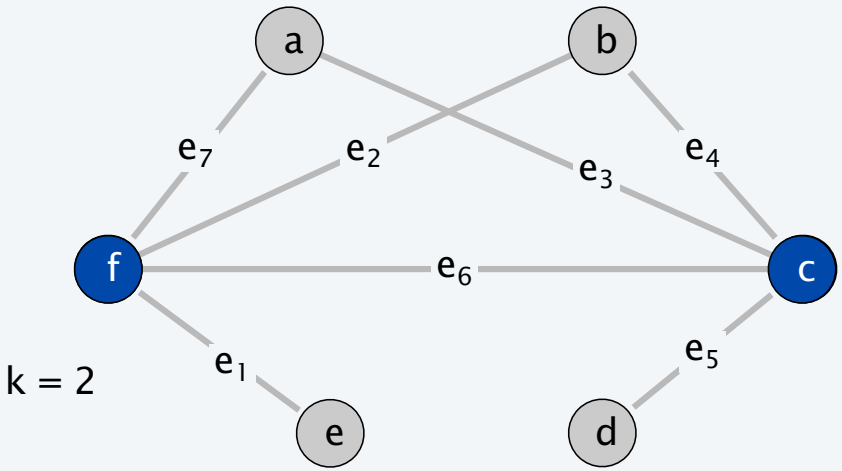
Theorem

VERTEX-COVER $\leq_P$ SET-COVER

Proof.
Étant donnée une instance $G = (V, E)$ (et k) de VERTEX-COVER, on construit l'instance $(U, \mathcal{S}, k)$ de SET-COVER de taille polynomial en la taille de G.

Lemma

G a un Vertex Cover de taille k ssi $(U, \mathcal{S})$ a un Set Cover de taille k.



vertex cover instance
(k = 2)

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 3, 7 \}$ $S_b = \{ 2, 4 \}$

$S_c = \{ 3, 4, 5, 6 \}$ $S_d = \{ 5 \}$

$S_e = \{ 1 \}$ $S_f = \{ 1, 2, 6, 7 \}$

set cover instance
(k = 2)

Set Cover $\leq_P$ Programmation Linéaire en Nombres Entiers (PLNE)

Étant donnée une instance de Set Cover, $(U, \mathcal{S})$ et k, on introduit l'ensemble de variables $X = \{x_j \mid S_j \in \mathcal{S}\}$, avec l'interprétation que si $x_j = 1$, alors l'ensemble S_j est choisi dans le Set Cover, sinon $x_j = 0$.
On considère alors le systèmes $\mathcal{E}_{(U,\mathcal{S})}$ d'inéquations suivant.

$$\mathcal{E}_{(U,\mathcal{S})} \begin{cases} \sum_{j=1}^{m} x_j \leq k \\ \{\sum_{j \mid i \in S_j} x_j \geq 1\}_{i \in U} \\ \{x_j \in \{0,1\}\}_{j \in \{1,\dots m\}} \end{cases}$$

Theorem

$(U, \mathcal{S})$ et k est une instance positive de Set Cover ssi $\mathcal{E}_{(U,\mathcal{S})}$ a une solution telle que $\sum_{j=1}^{m} x_j = k$.

Your turn!

Établir la preuve du Théorème 10.

Problèmes de recherche d'une solution

Problèmes de décision Existe-t-il un vertex cover de taille $\leq k$?
(Vertex-Cover)

Problèmes de recherche d'une solution Trouver un vertex cover de taille $\leq k$
(Find-Vertex-Cover).

Exemple Pour trouver un vertex cover de taille $\leq k$:

1. Déterminer s'il existe un vertex cover de taille $\leq k$.
2. Si oui, trouver un sommet v tel que $G \setminus \{v\}$ a un vertex cover de taille $\leq k-1$ (récursivement).
   (tout sommet v d'un Vertex Cover de taille k est tel que $G \setminus \{v\}$ a un Vertex Cover de taille $\leq k-1$).
   Inclure v dans le Vertex Cover de taille $k-1$ de $G \setminus \{v\}$.
   Sinon répondre qu'il n'y a pas de solution.

Bilan Vertex-Cover et Find-Vertex-Cover ont la même complexité

Problèmes de recherche d'une solution

Problèmes de décision Existe-t-il un vertex cover de taille $\leq k$? (Vertex-Cover)

Problèmes de recherche d'une solution Trouver un vertex cover de taille $\leq k$ (Find-Vertex-Cover).

Problèmes d'optimisation Trouver un vertex cover de taille minimum (Minimum-Vertex-Cover).

Exemple Pour trouver un vertex cover de taille minimum :

- Recherche (dichotomique) pour la taille k^* du minimum vertex cover
- Résoudre le problème de recherche associé.

Bilan Vertex-Cover et Find-Vertex-Cover ont la même complexité, et qui est la même que celle de $\equiv_P$ Minimum-Vertex-Cover

Cycle Hamiltonien se réduit à TSP

TSP Étant donné un ensemble de n villes et une distance $d(u, v)$ entre chaque ville, , et un entier k, existe-t-il une tournée de longueur $\leq k$?

HAM-CYCLE Étant donné un graphe non-dirigé $G = (V, E)$, existe-t-il un cycle élémentaire dont les sommets forment V ?

Theorem

HAM-CYCLE $\leq_P$ TSP

Proof.
Étant donné $G = (V, E)$ une instance de HAM-CYCLE, on crée n villes avec la fonction de distance

$$d(u, v) = \begin{cases} 0 & \text{if } (u, v) \in E \\ 1 & \text{if } (u, v) \notin E \end{cases}$$

Lemma

Cette instance de TSP *admet une tournée de longueur* 0 *ssi* G *a un cyle Hamiltonien.*

□

Définition de la classe de complexité NPTIME

Intuition des algorithmes de certification. On considère un problème de décision X.

- Un certifieur pour X est un algorithme $C(I, c)$ où I est une instance de X et qui utilise un certificat (ou encore témoin) c pour répondre à la quesiton "$I \in X$?".
- Un certifieur $C(I, c)$ ne détermine pas si $I \in X$ en soi, mais il utilise efficacement (c-à-d. en temps polynomial) le certificat c pour justifier que $I \in X$.

Definition

NPTIME *est l'ensemble des problèmes pour lesquels il existe un certifieur* $C(I, c)$ *qui calcule en temps polynimial où le certificat* c *est de taille polynomial :* $|c| \leq p(|I|)$, *où* $p(.)$ *est un polynôme.*

Cela signifie qu'in peut résoudre le problème X par l'algorithme non-déterministe polynomial suivant :

1. On calcule de façon non-déterministe un certificat c (cela prend un temps polynomial puisque $|c| \leq p(|I|)$);
2. On exécute l'algorithme $C(I, c)$ (ce qui prend un temps polynomial):
   - Si $C(I, c)$ retourne "oui" alors on **accepte** l'instance I
   - sinon on **rejette** l'instance I.

NPTIME se lit "non-deterministic polynomial time"

Définition de NPTIME

NPTIME. Les problèmes de décision pour lesquels il existe un certifieur polynomial.

Problem	Description	Algorithm	yes	no
L-SOLVE	Is there a vector x that satisfies $Ax = b$?	Gauss-Edmonds elimination	$\begin{bmatrix} 0 & 1 & 1 \\ 2 & 4 & -2 \\ 0 & 3 & 15 \end{bmatrix}, \begin{bmatrix} 4 \\ 2 \\ 36 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$
COMPOSITES	Is x composite ?	AKS (2002)	51	53
FACTOR	Does x have a nontrivial factor less than y ?	?	(56159, 50)	(55687, 50)
SAT	Is there a truth assignment that satisfies the formula ?	?	$\neg x_1 \vee x_2$ $x_1 \vee x_2$	$\neg x_2$ $\neg x_1 \vee x_2$ $x_1 \vee x_2$
3-COLOR	Can the nodes of a graph G be colored with 3 colors?	?		
HAM-PATH	Is there a simple path between s and t that visits every node?	?		

Theorem

$\mathrm{PTIME} \subseteq \mathrm{NPTIME}$

Proof.

Soit $X \in \mathrm{PTIME}$.

Par définition, il existe une algorithme polynomial-time $A(.)$ qui résout X.

On définit le certifieur $C(I, c) = A(I)$ qui ignore le certificat. □

Theorem

NPTIME ⊆ EXPTIME

Proof.

Soit un problème $X \in \text{NPTIME}$. Par définition, il existe une certifieur polynomial-time $C(I, c)$ pour X.
L'algorithme $A(I)$ exponentiel pour résoudre le problème X est le suivant :

1. pour chaque certificat c (où $|c| \leq p(|I|)$) // il y en a un nombre exponentiel
   Exécuter $C(I, c)$;
   Retourner $I \in X$ si $C(I, c)$ retourne "oui".
2. Retourner $I \notin X$.

□

La grande question : $\mathrm{PTIME} = \mathrm{NPTIME}$?

Q. Comment résout-on une instance de 3-SAT avec n variables ?

A. Par une recherche exhaustive : on essaie tous les 2^n valuations.

Q. Ne peut-on pas faire quelque chose de plus astucieux ?

Conjecture. *Il n'existe pas d'algorithme polynomial* pour 3-SAT.

"intractable"

La grande question PTIME = NPTIME ?

A-t-on PTIME = NPTIME ? [Cook 1971]
Les problèmes de décision sont-il aussi faciles que les problèmes de certification ?

Si oui. Alors nous avons des algorithmes efficaces pour 3-SAT, TSP, 3-Color, etc.
Si non. Il n'existe pas d'algorithmes efficaces pour 3-SAT, TSP, 3-Color, etc.

Un consensus. Sans doute que la réponse est "non", c'est à dire PTIME ≠ NPTIME.

NPTIME-complétude

Polynomial transformation

Definition (Réduction "à la Karp", rappel)

Le problème X se réduit au problème Y, noté $X \leq_P Y$, si il existe un polynôme $p(.)$ tel que pour toute instance arbitraire I du problème X, on peut construire une instance $r(I)$ du problème Y avec $|r(I)| \leq p(|I|)$ telle que

$$I \in X \text{ ssi } r(I) \in Y$$

Definition (NPTIME-complétude)

Un problème Y est NPTIME-complet si
(1) $Y \in$ NPTIME, et
(2) pour tout problème $X \in$ NPTIME, on a $X \leq_P Y$.

Polynomial transformation

Definition (NPTIME-complétude)

Un problème Y est NPTIME-*complet si*
(1) $Y \in$ NPTIME, et
(2) pour tout problème $X \in$ NPTIME, on a $X \leq_P Y$.

Theorem

Soit Y un problème NPTIME-*complet. Alors, $Y \in$ PTIME iff* PTIME $=$ NPTIME.

Proof.
$\Leftarrow$ Si PTIME $=$ NPTIME, comme $Y \in$ NPTIME alors $Y \in$ PTIME.
$\Rightarrow$ Supposons $Y \in$ PTIME.

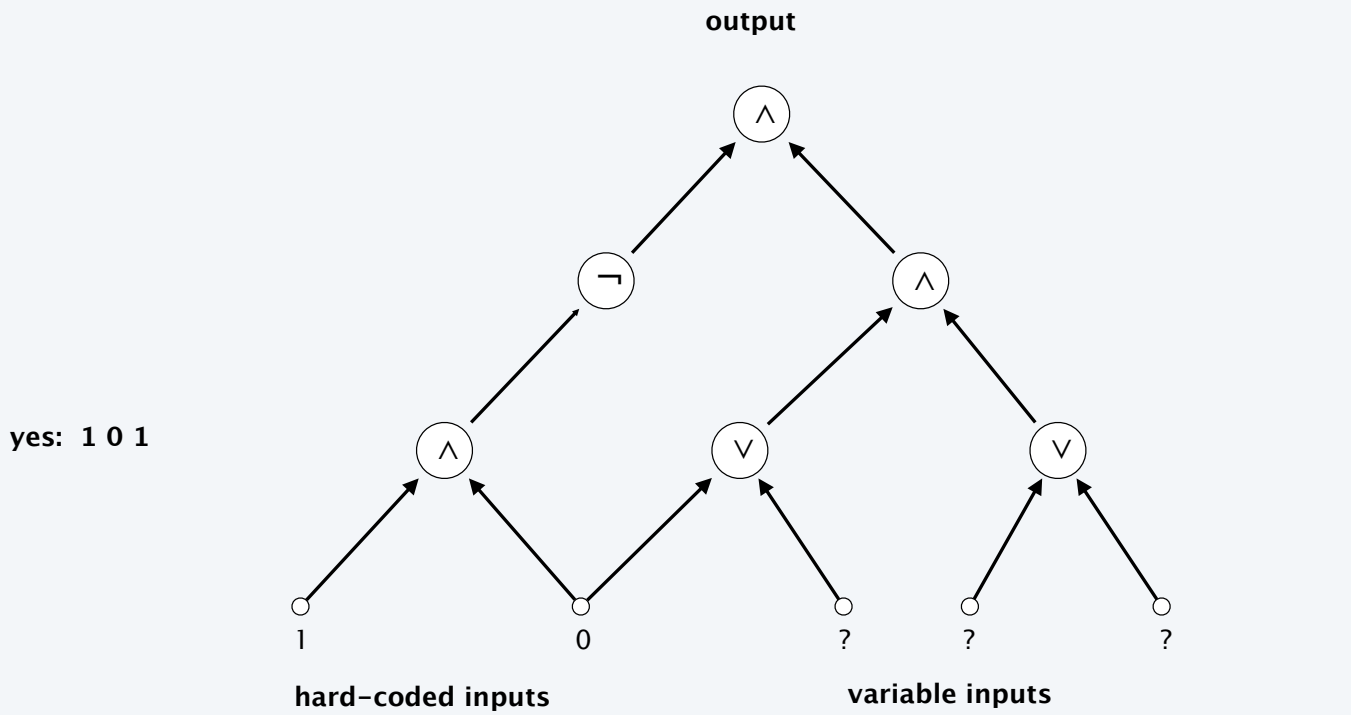
- Soit $X \in$ NPTIME. Puisque $X \leq_P Y$, on a $X \in$ PTIME
- Donc NPTIME $\subseteq$ PTIME.
- Or on sait déjà que PTIME $\subseteq$ NPTIME.

□

Question fondamentale. Existe-t-il des problèmes naturels NPTIME-complets ?

Satisfaisabilité d'un circuit

CIRCUIT-SAT Étant donné un circuit combinatoire construit à partir de portes AND, OR et NOT, existe-t-il une valeur pour les entrées du circuit de sorte qu'il en resorte la valeur 1 ?



Theorem (Cook 1971, Levin 1973)

CIRCUIT-SAT $\in$ NPTIME-*complet.*

Admis.

Établir la NPTIME-complétude d'un problème

Une fois qu'on a établi qu'un problème est NPTIME-complet, on peut en récupérer plein d'autres.

Pour montrer que $Y \in$ NPTIME-complet.

1. On établit que $Y \in$ NPTIME.
2. On choisit un problème NPTIME-complet X.
3. On établit que $X \leq_P Y$.

Theorem

Si $X \in$ NPTIME-complet, $Y \in$ NPTIME, et $X \leq_P Y$, alors $Y \in$ NPTIME-complet.

Proof.

Soit un problème $Z \in$ NPTIME. Alors on a $Z \leq_P X$, et aussi $X \leq_P Y$.

- Par transitivité $Z \leq_P Y$
- Donc $Y \in$ NPTIME-complet.

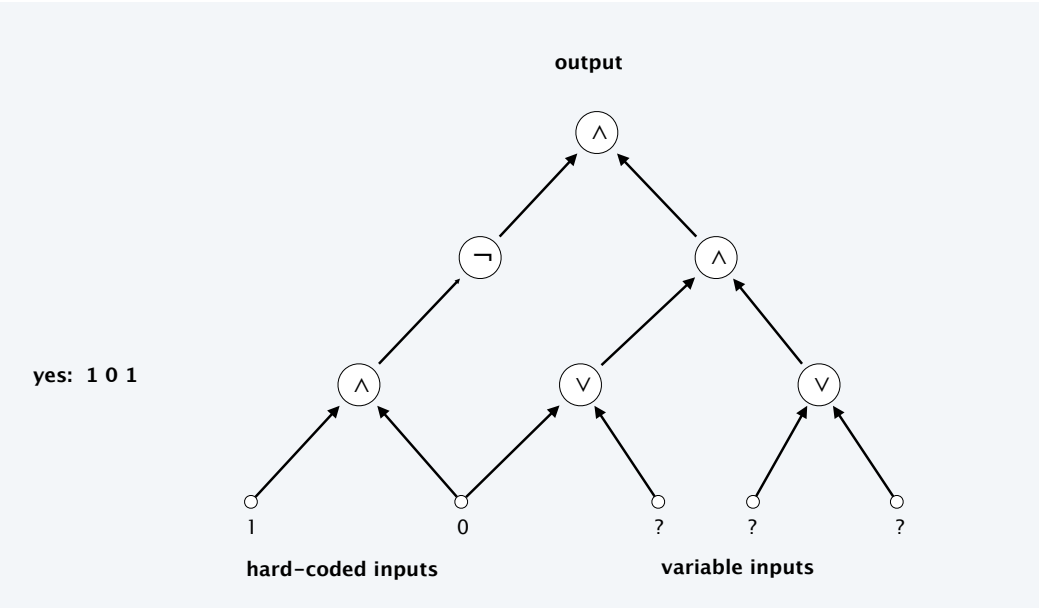
□

Theorem

3-SAT $\in$ NPTIME-*complet.*

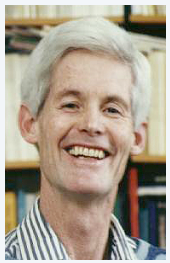
Proof.

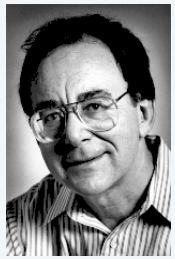
Elle repose sur CIRCUIT-SAT $\leq_P$ 3-SAT.



que l'on réduit à la satisfaisabilité de $(\neg(\texttt{true} \wedge \texttt{false})) \wedge ((\texttt{false} \vee p_1) \wedge (p_2 \vee p_3))$. □

Implications de Karp et Cook-Levin





Quelques problèmes NPTIME-complets

- Packing + covering problems: Set-Cover, Vertex-Cover,Independent-Set.
- Constraint satisfaction problems: Circuit-Sat, Sat, 3-sat.
- Sequencing problems: Ham-Cycle, TSP.
- Partitioning problems: 3d-Matching, 3-Color.
- Numerical problems: Subset-Sum, Partition.

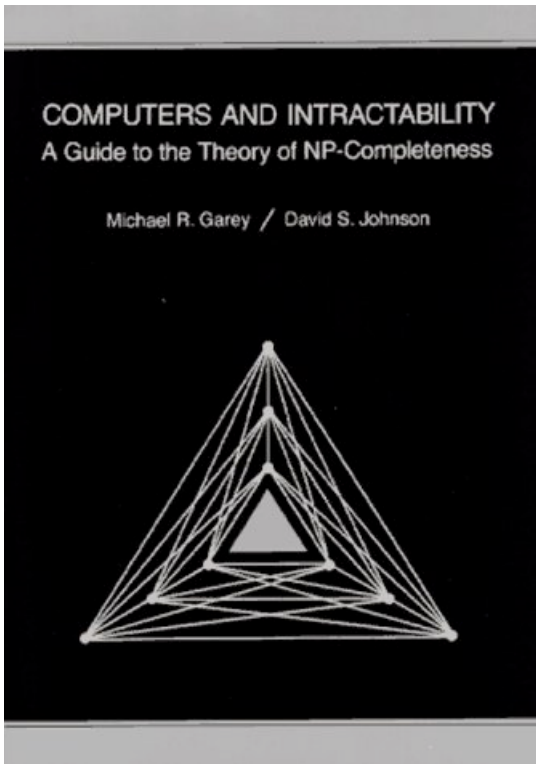
En pratique.
La plupart des problèmes NPTIME sont connus pour être soit dans PTIME, soit NPTIME-complets.

Exceptions notables. Factor, Graph-Isomorphism, Nash Equilibrium.

Plus de problèmes NPTIME-complets

Garey and Johnson

- Avec une annexe de plus de 300 problèmes NPTIME-complets.
- La référence en Informatique la plus citée.



Plus de problèmes NPTIME-complets

Aerospace engineering. Optimal mesh partitioning for finite elements.

Biology. Phylogeny reconstruction.

Chemical engineering. Heat exchanger network synthesis.

Chemistry. Protein folding.

Civil engineering. Equilibrium of urban traffic flow.

Economics. Computation of arbitrage in financial markets with friction.

Electrical engineering. VLSI layout.

Environmental engineering. Optimal placement of contaminant sensors.

Financial engineering. Minimum risk portfolio of given return.

Game theory. Nash equilibrium that maximizes social welfare.

Mathematics. Given integer a_1, ..., a_n, compute $\int_0^{2\pi} \cos(a_1\theta) \times \cos(a_2\theta) \times \cdots \times \cos(a_n\theta)\, d\theta$

Mechanical engineering. Structure of turbulence in sheared flows.

Medicine. Reconstructing 3d shape from biplane angiocardiogram.

Operations research. Traveling salesperson problem.

Physics. Partition function of 3d Ising model.

Politics. Shapley-Shubik voting power.

Recreation. Versions of Sudoko, Checkers, Minesweeper, Tetris.

Statistics. Optimal experimental design.

Optimization problems

Definition

An optimization problem $\mathcal{P}$ can be represented by a 4-tuple $(I_\mathcal{P}, F_\mathcal{P}, f_\mathcal{P}, goal_\mathcal{P})$ where

- *$I_\mathcal{P}$ is the collection of all instances of $\mathcal{P}$*
- *$F_\mathcal{P}$ with domain $I_\mathcal{P}$ provides candidate solutions: for $X \in I_\mathcal{P}$, $F_\mathcal{P}(X)$ is the family of feasible solutions of the instance X.*
- *$f_\mathcal{P}$ is the measure function which is used to compare two candidate solutions.*
  Let $P = \{(X, S) \mid X \in I_\mathcal{P} \text{ and } S \in F_\mathcal{P}(X)\}$, then $f_\mathcal{P} : P \to \mathbb{Z}$.
- *$goal_\mathcal{P}$ indicates whether the optimization problem is a minimization problem or a maximization problem.*

We are primarily concerned with minimization problems.
We simply consider $(I_\mathcal{P}, F_\mathcal{P}, f_\mathcal{P})$ where $goal_\mathcal{P}$ is implicit.
Also we may drop the subscribe $\mathcal{P}$ when the context is obvious.
Given an instance $X \in I$, $S^* \in F(X)$ is an optimal solution of X if

$$f(X, S^*) \leq f(X, S) \text{ for all } S \in F(X)$$

Optimization problems

A classic optimization problem:

Example (Symmetric TSP)

Let $G = (V, E)$ be a complete graph with a prescribed cost d_e for each edge $e \in E$. Then the symmetric traveling salesman problem (TSP) is to find a least cost Hamiltonian cycle in G. This problem is characterized by the triplet $(I_{\mathrm{TSP}}, F_{\mathrm{TSP}}, f_{\mathrm{TSP}}, goal_{\mathrm{TSP}})$ where

- $$I_{\mathrm{TSP}} = \{G = (V, E) \mid G \text{ is a complete graph with prescribed cost } w_e \text{ for each } e \in E\}$$
- $F_{\mathrm{TSP}}(G) = \{H \mid H \text{ is a hamiltonian cycle in } G\}$;
- $f_{\mathrm{TSP}}(G, H) = \sum_{e \in H} w_e$, *for any* $H \in F_{\mathrm{TSP}}(G)$.
- $goal_{\mathrm{TSP}}$ *is to minimize* f_{TSP}.

Decision problems

A decision problem is a 'yes/no' question.

Example

HAMILTONIAN CYCLE (HC)

Input : *A graph* $G = (V, E)$.
Output : *Is there a cycle that visits each vertex exactely once?*

Let $\mathcal{P}$ be a decision problem and $I_{\mathcal{P}}$ be the collection of all instances of $\mathcal{P}$.
Then $I_{\mathcal{P}}$ can be partitioned into two classes $Y_{\mathcal{P}}$ and $N_{\mathcal{P}}$, where

- $Y_{\mathcal{P}}$ contains all instances with answer yes, and
- $N_{\mathcal{P}}$ contains all instances with answer no.

Thus given an instance $X \in I_{\mathcal{P}}$, the decision problem is to determine if $X \in Y_{\mathcal{P}}$.

The natural decision problem associated to $\mathcal{P} = (I_{\mathcal{P}}, F_{\mathcal{P}}, f_{\mathcal{P}})$ is:
$\mathcal{P}_d$

Input : An instance $X \in I_{\mathcal{P}}$ and an integer $k \in \mathbb{Z}$.
Output : Does there exist $S \in F_{\mathcal{P}}(X)$ such that $f_{\mathcal{P}}(X, S) \leq k$?

Constraints problems, SAT

Satisfiability (SAT)

Input : A set $C_1, \ldots, C_n$ of n clauses.
Output : Is there an assignment of the variables that satisfies all the clauses?

For you: What would it be useful for?

3Satisfiability (3SAT)

Input : A set $C_1, \ldots, C_n$ of n clauses with 3 literals in each clause.
Output : Is there an assignment of the variables that satisfies all the clauses?

Maximum Satisfiability (MAXSAT)

Input : A set $C_1, \ldots, C_n$ of n clauses and an integer g.
Output : Is there an assignment of the variables that satisfies at least g clauses.

For you: What would it be useful for?

Have a look at

Practical Applications of Boolean Satisfiability, by Joao Marques-Silva
`http://eprints.soton.ac.uk/265340/1/jpms-wodes08.pdf`

Problems on graphs: Rudrata, also known as *Hamiltonian cycle*

A Kashmiri poet named Rudrata had asked this question: Can one visit all the squares of the chessboard, without repeating any square, in one long walk that ends at the starting square and at each step makes a legal knight move?
`https://en.wikipedia.org/wiki/Rudrata`
`https://en.wikipedia.org/wiki/Knight's_tour`

RUDRATA (Rudrata)

Input : A graph $G = (V, E)$
Output : A cycle that passes through every vertex exactly once.

For you: What would it be useful for?

Problems on graphs: Independent set, vertex cover, and clique

Independent Set (IndepSet)

Input : A graph $G = (V, E)$ and a bound b
Output : Does there exist an independent set S in G of cardinal at least b?

For you: What would it be useful for?

Vertex Cover (VerCov)

Input : An undirected graph $G = (V, E)$ and $k > 0$.
Output : Does there exist a subset of vertices $S \subseteq V$ with $|S| \leq k$ that touches every edge.

For you: What would it be useful for?

Clique (CLIQUE)

Input : A graph $G = (V, E)$ and an integer g.
Output : Is there a subset of g vertices such that all possible edges between them are present?

For you: What would it be useful for?

The Traveling Salesman Problem

Travelling Salesman (TSP)

Input : A graph $G = (V, E)$, for each $e \in E$, an integer $w_e > 0$.
Output : A cycle that passes through every vertex exactly once with minimal cost.

Symmetric Travelling Salesman (SymTSP)

Input : A undirected graph $G = (V, E)$, for each $e \in E$, an integer $d_e > 0$.
Output : A cycle that passes through every vertex exactly once with minimal cost.

Numerical problems: Subset Sum and Knapsack problems

Subset Sum (SubsetSum)

Input : Natural numbers w_1, ..., w_n and an integer W
Output : Is there a subset that adds up to exactly W?

Knapsack (KP)

Input : n items, for each item i two values w_i (≥ 0 its weight) et p_i (its profit), a constant $K \geq 0$.
Output : An assignement of the variables $x_1, x_2, \ldots, x_n$ in $\{0, 1\}$ such that $\Sigma_{i=1}^{n} w_i.x_i \leq K$ and which maximizes the total profit $\Sigma_{i=1}^{n} p_i.x_i$.

From decision problems to optimization problems

Each solution has a cost that we want to minimize/maximize.

Maximum Independent Set problem

Input : A graph $G = (V, E)$
Output : A largest possible independent set S of G.

Minimum Vertex Cover Problem

Input : An undirected graph $G = (V, E)$.
Output : The minimum vertex cover of G.

Maximum Satisfiability Problem

Input : A set $C_1, \ldots, C_n$ of n clause.
Output : A assignment of the variables that satisfies as many of them as possible.

Complexity classes

Definition

An algorithm A for a computational problem $\mathcal{P}$ (optimization or decision version) is a polynomial time (space) algorithm if any $X \in I_{\mathcal{P}}$ can be solved in time (memory space) $O(n^d)$ for some constant d, where n is the input size of X.

Definition

The collection of decision problems that can be solved by a polynomial time algorithm constitutes the class PTIME *(Its elements are* PTIME*-easy problems).*

Definition

A super-class of PTIME*, called* NPTIME*, is defined as the class of decision problems $\mathcal{P}$ where every $X \in Y_{\mathcal{P}}$ has a concise certificate the validity of which can be verified in polynomial time (Its elements are* NPTIME*-easy problems).*

For a decision problem $\mathcal{P}$ we may write $\mathcal{P} \in \mathrm{PTIME}$, $\mathcal{P} \in \mathrm{NPTIME}$, etc.

An example for NPTIME-easiness

Travelling Salesman (TSP)

Input : A graph $G = (V, E)$ with for each $e \in E$, an integer $w_e > 0$, and an integer $k \in \mathbb{Z}$.
Output : A cycle that passes through every vertex exactly once with cost $\leq k$.

Proposition

TSP $\in$ NPTIME.

Proof.

For an 'yes' instance of this problem a concise certificate would be a collection of edges that form a Hamiltonian cycle in G with cost less than or equal to k. Clearly, whether this collection of edges forms a Hamiltonian cycle in G and its cost is less than or equal to K can be verified in $O(n)$ time, where $n = |V|$. Thus the decision version of TSP $\in$ NPTIME. □

Your turn!

Recall
Satisfiability (SAT)

Input : *A set $C_1, \ldots, C_n$ of n clauses.*
Output : *Is there an assignment of the variables that satisfies all the clauses?*

Show that SAT *is* NPTIME*-easy.*

An example likely not in NPTIME

The following problem

Input : A graph $G = (V, E)$ and $k \in \mathbb{N}^+$.
Output : Do there exist $\lceil \frac{|V|}{k} \rceil !$ Hamiltonian cycles in G?

is not known to be in NPTIME since we do not know a concise certificate for a 'yes' instance of this problem, the validity of which can be verified in polynomial time.

- One possible certificate is a list of $\lceil \frac{|V|}{k} \rceil !$ Hamiltonian cycles.
- Even though verifying each member of this list is indeed a Hamiltonian cycle can be done in polynomial time, counting that there are $\lceil \frac{|V|}{k} \rceil !$ of them takes time exponential in $|V|$ and hence the certificate is not concise.

Reductions

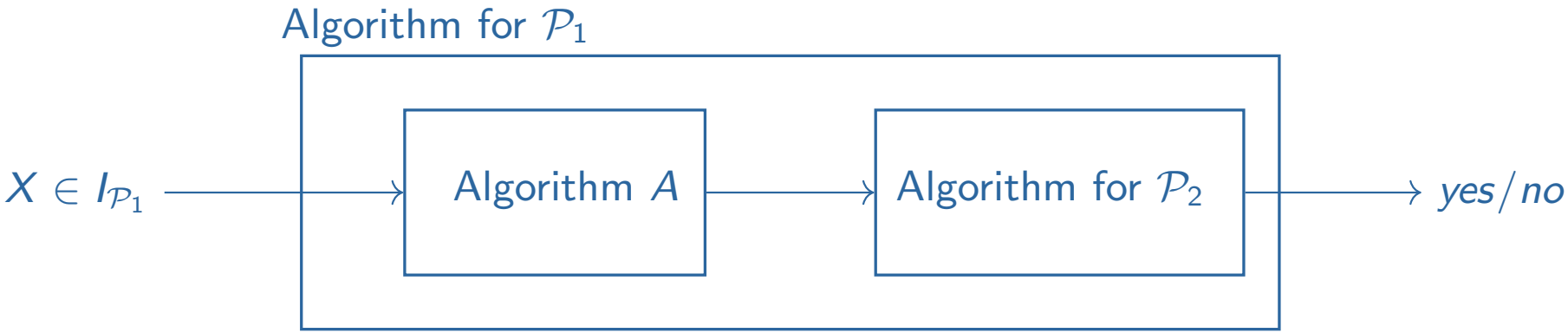
Definition

A decision problem $\mathcal{P}_1$ is reducible to a decision problem $\mathcal{P}_2$ if there exists an algorithm A whose input is any instance $X \in I_{\mathcal{P}_1}$, and returns $A(X) \in I_{\mathcal{P}_2}$

$$X \in I_{\mathcal{P}_1} \longrightarrow \boxed{\text{Algorithm } A} \longrightarrow A(X) \in I_{\mathcal{P}_2}$$

such that $X \in Y_{\mathcal{P}_1}$ if, and only if, $A(X) \in Y_{\mathcal{P}_2}$.

Algorithm A is a reduction from $\mathcal{P}_1$ to $\mathcal{P}_2$.
If moreover A is polynomial time, we say that $\mathcal{P}_1$ is polynomially reducible to $\mathcal{P}_2$ and denote this by $\mathcal{P}_1 \leq_{\text{PTIME}} \mathcal{P}_2$.



NPTIME-complete problems

Definition

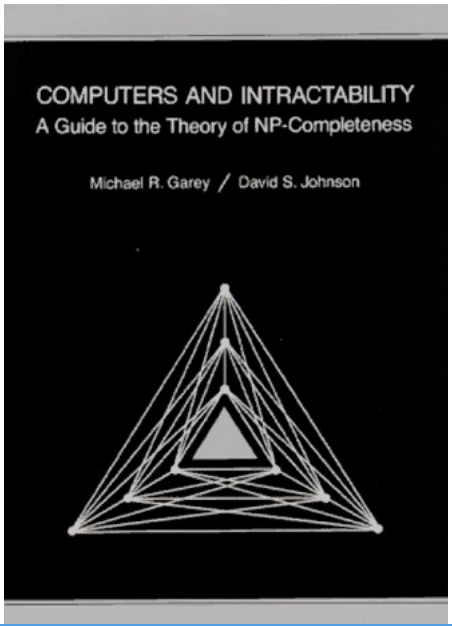
A decision problem $\mathcal{P}$ is said to be NPTIME*-hard if*

$$\text{for any } \mathcal{P}' \in \text{NPTIME}, \mathcal{P}' \leq_{\text{PTIME}} \mathcal{P}.$$

Definition

The class of NPTIME*-complete problems is the class of problems that are both* NPTIME*-easy and* NPTIME*-hard.*

Do have a look at **Computers and Intractability**, by M.R. Garey and D.S. Johnson.



Theorem (Cook-Levin 1978)

SAT *is an* NPTIME*-complete problem.*

Example

The problems SAT, 3SAT, CLIQUE, HC *(*Rudrata*)*, IndepSet *and* VerCov *are* NPTIME*-complete problems.*

Bibliography

- **Conception d'algorithmes Principes et 150 exercices non corrigés.**, by Bosc, P., Guyomard, M., and Miclet, L. EYROLLES 2016.
- **Combinatorial Optimization: Algorithms and Complexity**, by Christos H. Papadimitriou, Kenneth Steiglitz. Dover publications, 1998.
- **Computers and Intractability**, by M.R. Garey and D.S. Johnson. W.H. Freeman, San Francisco, CA, 1979.
- **Computational Complexity** by C.H. Papadimitriou. Addison-Wesley, New York, 1994.
- **Algorithms**, by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani. McGraw-Hill Education (India) Pvt Limited, 2006.
- **Algorithm Design**, by J. Kleinberg and E. Tardos. Addison Wesley, 2005.
- **Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization**, E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys, editors. Wiley. Chichester, 1985.
- **The Traveling Salesman Problem and Its Variations**, G. Gutin and A. Punnen. Kluwer Academic Publishers, 2002.

Principles of intelligent search:

Backtracking / Essais Successifs

Branch-and-Bound / Évaluation et Séparation Progressive

The backtracking paradigm: The zero-sum subset problem

Consider the following problem.

ZERO-SUM SUBSET PROBLEM

Input : A set E with n (≥ 0) elements in $\mathbb{Z}$.
Output : The list of all the subsets of E that sum up to zero.

The set C of candidates is the powerset $\mathcal{P}(E)$, we have to go through all the 2^n candidates.

The *selection predicate* is "the sum of all elements equals zero".

Bibliography "Conception d'algorithmes: Principes et 150 exercices corrigés" by Guyomard, Bosc et Miclet.
Bibliography Chapter 6 of "The Algorithm Design Manual" by Steven S. Skiena.

The zero-sum subset problem

We assume that the set E is represented by an array $A[1..n]$ of integer.

Example (where $n = 6$)

$E = \{10, 0, -3, -7, 5, -5\}$ *represented by* $A = [10, 0, -3, -7, 5, -5]$

Each candidate is represented by its *characteristic vector*.

Example

Vector $[1, 0, 0, 0, 0, 0]$ *represents* $\{10\}$, *and vector* $[1, 1, 1, 1, 1, 1]$ *represents* E.

We alternatively may identify the characteristic vectors with functions from from $\{1, \ldots, n\}$ to $\{0, 1\}$.

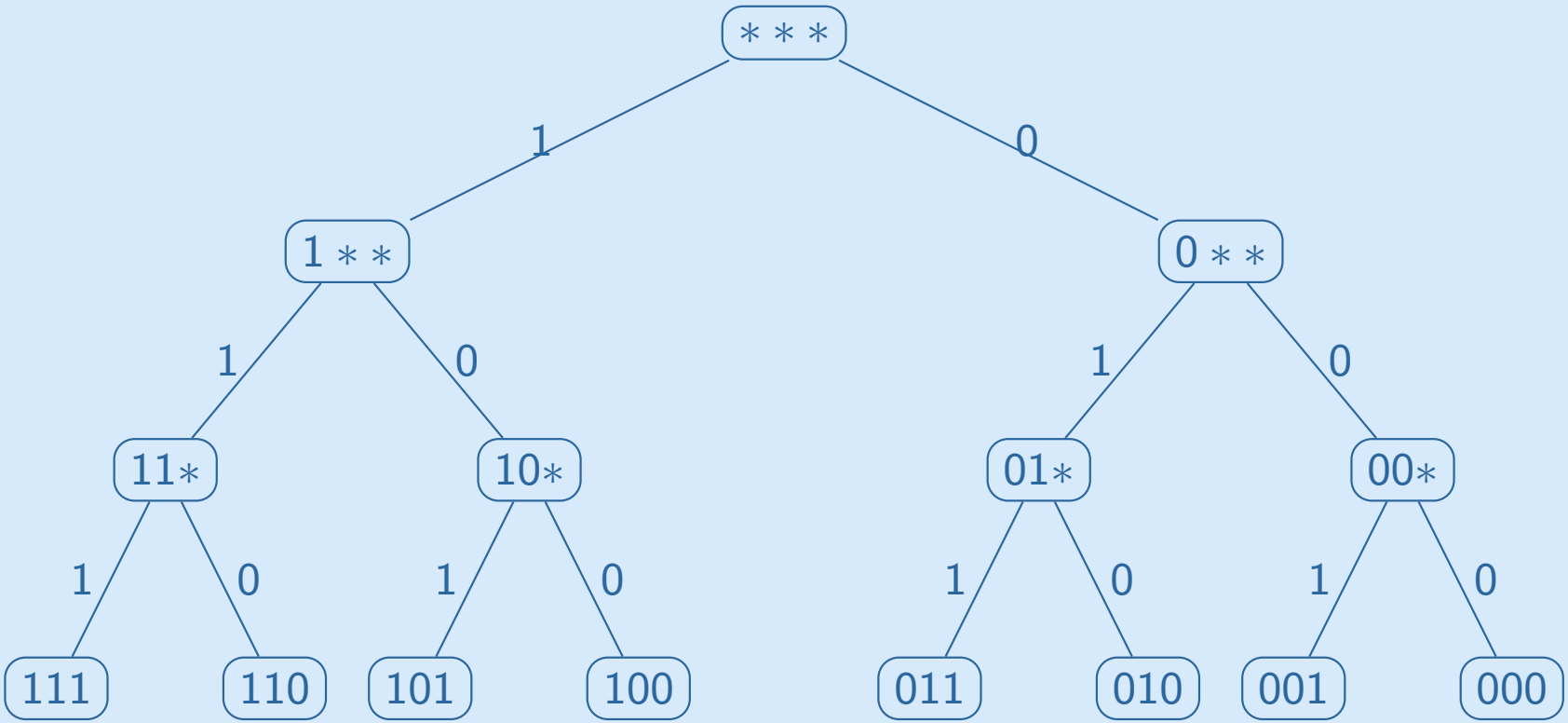
Given a characteristic vector X, the sum of the elements of the subset it denotes is given by:

$$\sum_{i=1,\ldots,n} X[i] * A[i]$$

Enumerate characteristic vectors

First we need to find a way to enumerate the 2^n candidates.
We use a *recursion tree* whose leaves are characterictic vectors.

Example (where $n = 3$)



The skeleton algorithm to enumerate characteristic vectors

We perform a traversal of the recursion tree.

Algorithm $Subsets1(i)$

Require: $i \in 1..n$

```
1: for j in 1..0 do
2:     X[i] ← j
3:     if i = n then
4:         Print(X)
5:     else
6:         Subsets1(i + 1)
7:     end if
8: end for
```

Initial call:

```
1: Constants n ∈ N and n = ... and A ∈ [1..n] → Z
2: Variables X ∈ [1..n] → [0..1]
3: begin
4: Subsets1(1)
5: end
```

Correction of Algorithm *Subsets*1(i)

Algorithm *Subsets*1(i)

Require: $i \in 1..n$

```
1: for j in 1..0 do
2:     X[i] ← j
3:     if i = n then
4:         Print(X)
5:     else
6:         Subsets1(i + 1)
7:     end if
8: end for
```

Initial call:

```
1: Constants n ∈ N and n = ... and A ∈ [1..n] → ℤ
2: Variables X ∈ [1..n] → [0..1]
3: begin
4: Subsets1(1)
5: end
```

Your turn!

Show that Subsets1 is correct.

Selection of the zero-sum subsets

Algorithm *Subsets*2 is like *Subsets*1 but filters according to the zero sum predicate.

Algorithm *Subsets*2(i)

Require: $i \in 1..n$

```
1: for j in 1..0 do
2:     X[i] ← j
3:     if i = n and ZeroSum then
4:         Print(X)
5:     else
6:         Subsets2(i + 1)
7:     end if
8: end for
```

Initial call:

```
1: Constants n ∈ N and n = ... and A ∈ [1..n] → ℤ and ZeroSum ∈ Bool
2: Variables X ∈ 1..n → 0..1
3: begin
4: ZeroSum ← false ; Subsets2(1)
5: end
```

Optimization of $Subsets2(i)$: $Subsets3(i)$

Compute the sum on-the-fly:

Algorithm $Subsets3(i)$

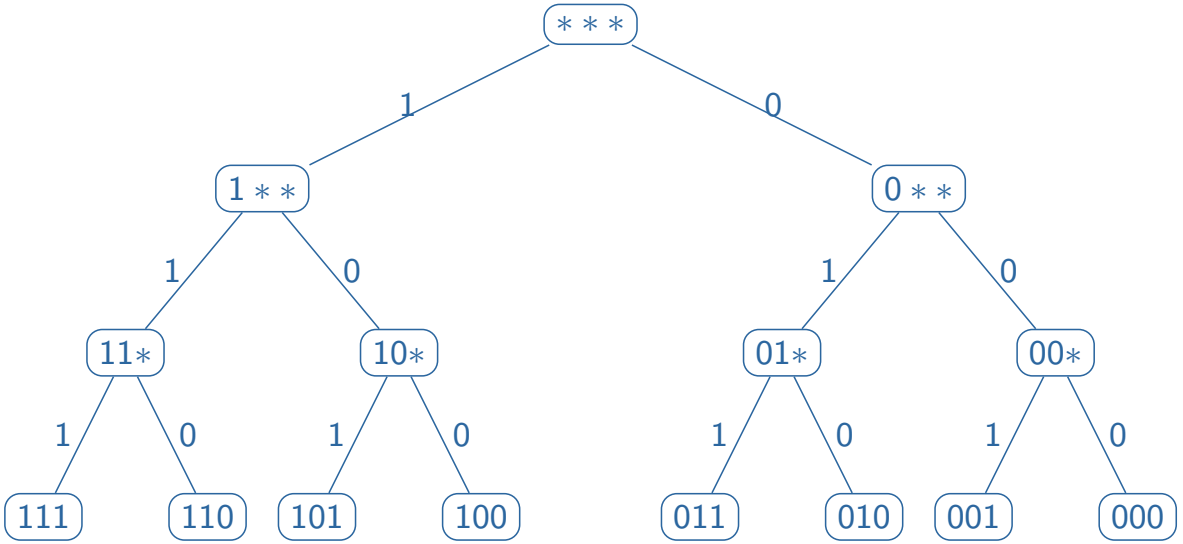
Require: $i \in 1..n$

```
1:  for j in 1..0 do
2:      X[i] ← j ; [CurrentSum ← CurrentSum + j * A[i]]
3:      if i = n [and CurrentSum = 0] then
4:          Print(X)
5:      else
6:          if i ≠ n then
7:              Subsets3(i + 1)
8:          end if
9:      end if
10:     [CurrentSum ← CurrentSum − j * A[i]]
11: end for
```

Initial call:

```
1: Variables X ∈ 1..n → 0..1 et CurrentSum ∈ ℤ
2: begin
3: [CurrentSum ← 0] ; Subsets3(1)
4: end
```

Complexity



The number of recursive calls corresponds to the number of internal nodes of the recursion tree, namely $2^n - 1$.

Can we avoid this *brute force* approach?

More generally, we will see that it is not always necessary to enumerate all the candidates: some of them may be discarded at an early stage.

This principle is called the *pruning*.

The pruning criterion for the zero-sum subsets problem

We may remark the following: if at a given time of the computation, the absolute value $|CurrentSum|$ is strictly greater than absolute value of the extremum of A (written $AbsExtA$) multiplied by the number of remaining values in A, then this node need not be developed further.

Your turn!

Give an example.

Your turn!

Find another criterion that is smarter.

Improvement of $Subsets3(i)$ by pruning: $Subsets4(i)$

Algorithm $Subsets4(i)$

Require: $i \in 1..n$

```
1:  for j in 1..0 do
2:      CurrentSum ← CurrentSum + j * A[i]
3:      if |CurrentSum| ≤ (n − i) * AbsExtA then
4:          X[i] ← j
5:          if i = n and CurrentSum = 0 then
6:              Print(X)
7:          else
8:              if i ≠ n then
9:                  Subsets4(i + 1)
10:             end if
11:         end if
12:     end if
13:     CurrentSum ← CurrentSum − j * A[i]
14: end for
```

Initial call: $AbsExtA \leftarrow max\{|max(A)|, |min(A)|\}$; $Subsets4(1)$

Functions and skeleton

We consider a more general/abstract situation where we seek for functions from $\mathtt{I} = 1..n$ to $\mathtt{F} = 1..m$.

We will consider:

- the total functions;
- the surjective total functions;
- the injective total functions;

Cases of partial functions will be considered in supervised exercises (TD).

Enumeration of all total functions

from interval $\mathtt{I} = 1..n$ to interval $\mathtt{F} = 1..m$.

Algorithm $T(i)$

Require: $i \in 1..n$

1: **for** j in $1..m$ **do**
2: $X[i] \leftarrow j$
3: **if** $i = n$ **then**
4: Print(X)
5: **else**
6: $T(i+1)$
7: **end if**
8: **end for**

Initial call:

1: Constants $n \in \mathbb{N}$ and $m \in \mathbb{N}$.
2: Variables $X \in 1..n \rightarrow 1..m$
3: **begin**
4: $T(1)$
5: **end**

Complexity of Algorithm $T(1)$

Algorithm $T(i)$

Require: $i \in 1..n$

```
1: for j in 1..m do
2:     X[i] ← j
3:     if i = n then
4:         Print(X)
5:     else
6:         T(i + 1)
7:     end if
8: end for
```

Initial call:

```
1: Constants n ∈ N and m ∈ N.
2: Variables X ∈ 1..n → 1..m
3: begin
4: T(1)
5: end
```

There are $\sum_{i=1}^{i=n} m^i$ recursive calls, namely $O((m^{n+1} - 1)/(m - 1))$.

Because $(m^{n+1} - 1))/(m - 1)$ is exponenetial in n, this prevents us from using this algorithm in practice.

Total functions over a cartesian product

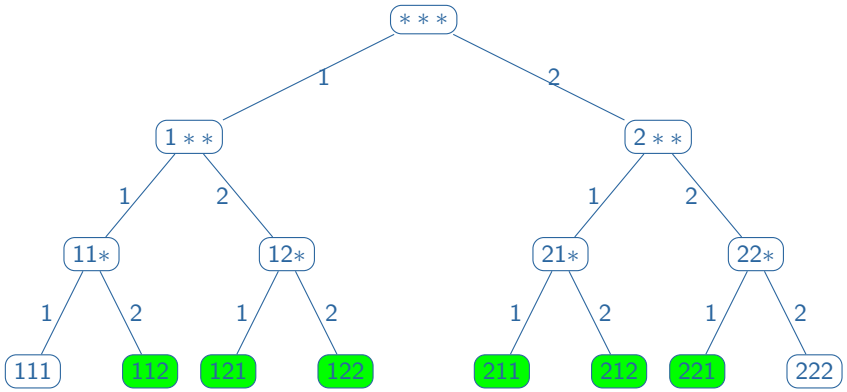
How can we adapt the previous approach if $\mathtt{I}$ is 2-dimensional, say $\mathtt{I} = [1..n] \times [1..q]$?
Now, vector X is a matrix, and we get the following algorithm.

Algorithm $T2D(\ell, c)$

Require: $\ell \in 1..n$ et $c \in 1..q$

```
1:  for j in 1..m do
2:      X[ℓ, c] ← j
3:      if ℓ = n and c = q then
4:          Print(X)
5:      else
6:          if c = q then
7:              T2D(ℓ + 1, 1)
8:          else
9:              T2D(ℓ, c + 1)
10:         end if
11:     end if
12: end for
```

Recursion tree for surjective total functions from 1..3 to 1..2.



Surjective total functions correspond to green leaves.

Algorithm $TS(i)$

Require: $i \in 1..n$

1: **for** j in $1..m$ **do**
2: $X[i] \leftarrow j$; $\boxed{previousB \leftarrow B[j] \ ; \ B[j] \leftarrow vrai}$
3: **if** $i = n$ $\boxed{\text{and for all } k \in 1..m,\ B[k]}$ **then**
4: Print(X)
5: **else**
6: $TS(i+1)$
7: **end if**
8: $\boxed{B[j] \leftarrow previousB}$
9: **end for**

Surjective total functions

Enumerate all surjective total functions between intervals `I` et `F`.

Algorithm $TS(i)$

Require: $i \in 1..n$

1: **for** j in $1..m$ **do**
2: $X[i] \leftarrow j$; $previousB \leftarrow B[j]$; $B[j] \leftarrow vrai$
3: **if** $i = n$ and for all $k \in 1..m$, $B[k]$ **then**
4: Print(X)
5: **else**
6: $TS(i+1)$
7: **end if**
8: $B[j] \leftarrow previousB$
9: **end for**

Initial call:

1: Variables $X \in [1..n] \rightarrow k[1..m]$ and $previousB \in Bool$ and $B \in [1..m] \rightarrow Bool$
2: **begin**
3: $B \leftarrow (1..m) \times \{false\}$; $TS(1)$
4: **end**

Enumerate all injective total functions

We extend the Algorithm $T(1)$ as $IT(1)$ in order to enumerate all injective total functions from $\mathtt{I} = 1..n$ to $\mathtt{F} = 1..m$.
For the call of order i, we discard from set $\mathtt{F}$ all elements that have already been selected in previous choices (*i.e.* in $X[1..i-1]$).

Algorithm $IT(i)$

Require: $i \in 1..n$

1: **for** j in $1..m \setminus codom(X[1..i-1])$ **do**
2: $\quad X[i] \leftarrow j$
3: $\quad$ **if** $i = n$ **then**
4: $\quad\quad$ Print(X)
5: $\quad$ **else**
6: $\quad\quad IT(i+1)$
7: $\quad$ **end if**
8: **end for**

Your turn!

Draw the recursion tree for the case $\mathtt{I} = 1..3$ *and* $\mathtt{F} = 1..4$.

Injective and bijective total functions, permutations

Your turn!

How many injective total functions are there from $\mathtt{I} = 1..n$ *to* $\mathtt{F} = 1..m$*?*
How many recursive calls of IT are there?

Injective and bijective total functions, permutations

Your turn!

How many injective total functions are there from $\mathtt{I} = 1..n$ *to* $\mathtt{F} = 1..m$*?*
How many recursive calls of IT are there?

There are $A_m^n = m!/(m-n)!$, whenever $m \geq n$.

Injective and bijective total functions, permutations

Your turn!

How many injective total functions are there from $\mathtt{I} = 1..n$ *to* $\mathtt{F} = 1..m$*?*
How many recursive calls of IT *are there?*

There are $A_m^n = m!/(m-n)!$, whenever $m \geq n$.

Algorithm $IT(i)$

Require: $i \in 1..n$

```
1: for j in 1..m \ codom(X[1..i − 1]) do
2:     X[i] ← j
3:     if i = n then
4:         Print(X)
5:     else
6:         IT(i + 1)
7:     end if
8: end for
```

Remark that for $\mathtt{I} = 1..n = \mathtt{F}$, Algorithm $IT(1)$ enumerates all bijections or equivalently all *permutations* over n elements.

Template for backtracking algorithms

We study templates for

- "AllSolutions"
- "OneSolution"
- "OptimalSolution", for some optimality criterion

Example (Zero-sum subset problems)

Instead of enumerating all solutions, we may ask for:

- *"OneSolution": i.e. is there one zero-sum subset?*
- *"OptimalSolution": e.g. a maximal zero-sum subset, if any.*

Templates for "AllSolutions"

Algorithm $AllSol(i)$

Require: $i \in 1..n$

```
1:  for j in 1..m do
2:      if Admissible(i, j) then
3:          X[i] ← j ; Update(i, j)
4:          if i = n and then SolutionFound then
5:              Sol ← Sol ∪ {X}
6:          else
7:              if i ≠ n then
8:                  AllSol(i + 1)
9:              end if
10:         end if
11:         Undo(i, j)
12:     end if
13: end for
```

- $Admissible(i, j)$ is used to prune,
- *Modify* et *Undo* go in pairs: they are needed to propagate the context of the current choice and to release this context, respectively.
- Condition *SolutionFound* (which may be useless) expresses that the structure X satisfies the contraints given by the very problem to be solved.
- Notice that condition $(i \neq n)$ is needed only when we use *SolutionFound*, since the current structure X might be incomplete and the search must still go on.

Templates for "OptimalSolution: a total ordering between candidates and we assume there is at leas one solution

(1) Y is a global variable that hosts an optimal solution, (2) *StoreCurrentContext* stores in global variables information to compare the solutions in order to evaluate *BetterSolution*.

Algorithm $OptSol(i)$

```
Require: i ∈ 1..n
 1: for j in 1..m do
 2:     if Admissible(i, j) then
 3:         X[i] ← j ; Modify(i, j)
 4:         if i = n and-then SolutionFound(n, j) then
 5:             if BetterSolution(X) then
 6:                 X* ← X
 7:                 StoreCurrentContext
 8:             end if
 9:         else
10:             if i ≠ n then
11:                 OptSol(i + 1)
12:             end if
13:         end if
14:         Undo(i, j)
15:     end if
16: end for
```

Templates for "OptimalSolution"

Your turn!

Write the context for the call of OptSol.

Templates for "OptimalSolution"

Your turn!

Write the context for the call of OptSol.

1: Constants $n \in \mathbb{N}$ and $n = ...$ and $T \in n \rightarrow \mathbb{Z}$ and $T = [...]$
2: Variables $X \in 1..n \rightarrow 1..m$, $Found \in Bool$ (for found a solution), and $X^* \in 1..n \rightarrow 1..m$ (best solution so far)
3: **begin**
4: $X^* \leftarrow \perp$; $OptSol(1)$; Print(X^*)
5: **end**

Templates for "OptimalSolution"

Your turn!

What should we write if we do not assume there is a solution?

Templates for "OptimalSolution"

Your turn!

What should we write if we do not assume there is a solution?

Algorithm $OptSol(i)$

Require: $i \in 1..n$
1: **for** j in $1..m$ **do**
2: **if** $Admissible(i, j)$ **then**
3: $X[i] \leftarrow j$; $Modify(i, j)$
4: **if** $i = n$ and-then $SolutionFound(n, j)$ **then**
5: $Found \leftarrow true$
6: **if** $BetterSolution(X)$ **then**
7: $X^* \leftarrow X$
8: $StoreCurrentContext$
9: **end if**
10: **else**
11: **if** $i \neq n$ **then**
12: $OptSol(i + 1)$
13: **end if**
14: **end if**
15: $Undo(i, j)$
16: **end if**
17: **end for**

1: Constants $n \in \mathbb{N}$ and $n = ...$ and $T \in n \rightarrow \mathbb{Z}$ and $T = [...]$
2: Variables $X \in 1..n \rightarrow 1..m$ and $Found \in Bool$
3: **begin**
4: $Found \leftarrow false$; $OptSol(1)$
5: **if** $Found$ **then**
6: Print(X^*)
7: **end if**
8: **end**

Template for "OneSolution"

Your turn!

Your job!

Branch-and-Bound
$\equiv$
Programmation par Séparation et Évaluation Progressive

Principles of of Branch-and-Bound

We are in the situation where for the optimization problem we want to solve we have:

- A set $\mathcal{C}$ of *candidates* with their cost given as $cost : \mathcal{C} \to \mathbf{R}_+$;
- We look for a solution, that is any candidate $c^* \in \mathcal{C}$ such that

$$cost(c^*) = \min_{c \in \mathcal{C}} cost(c)$$

Remark

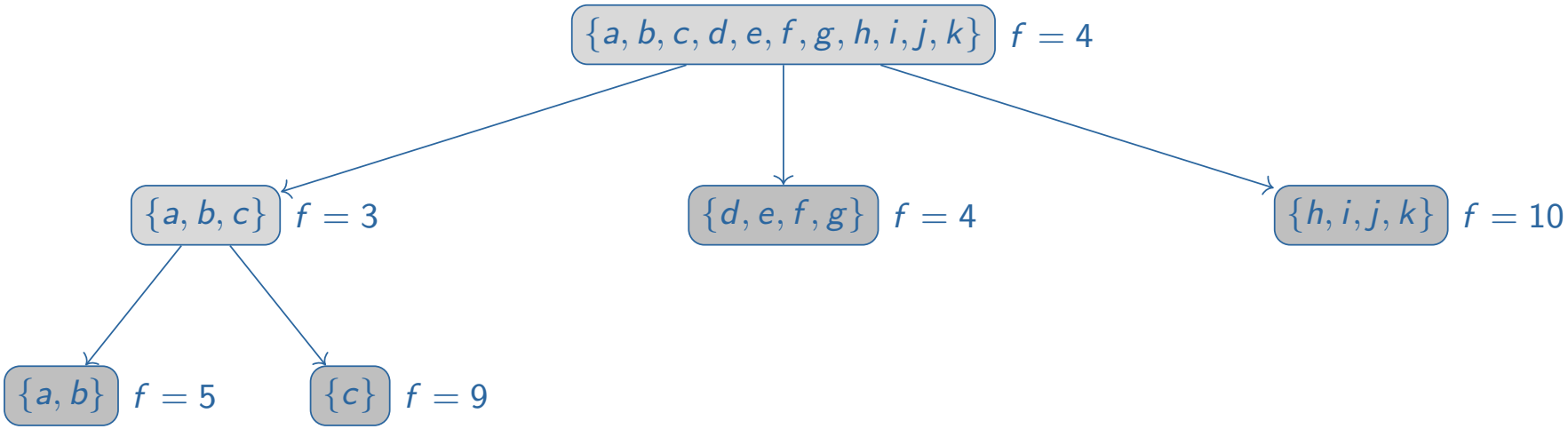
- *If the set of candidates $\mathcal{C}$ is big, it is not tractable to enumerate its elements.*
- *This is typically of* NPTIME*-complete problems.*

The Branch-and-Bound approach works:

- with collections of subsets of candidates that emerge from a partition of $\mathcal{C}$.
  - at the beginning the partition is $\{\mathcal{C}\}$;
  - over time, the partition is refined into $\{C_1, C_2, \ldots C_n\}$.
- with a function f, called a *heuristics* (build by the designer of the algorithm). A heuristics under-approximates the cost of all candidates in each element C_i of the current partition:

$$f(C_i) \leq \min cost(c)$$

A tree that reflect the execution of Branch-and-Bound

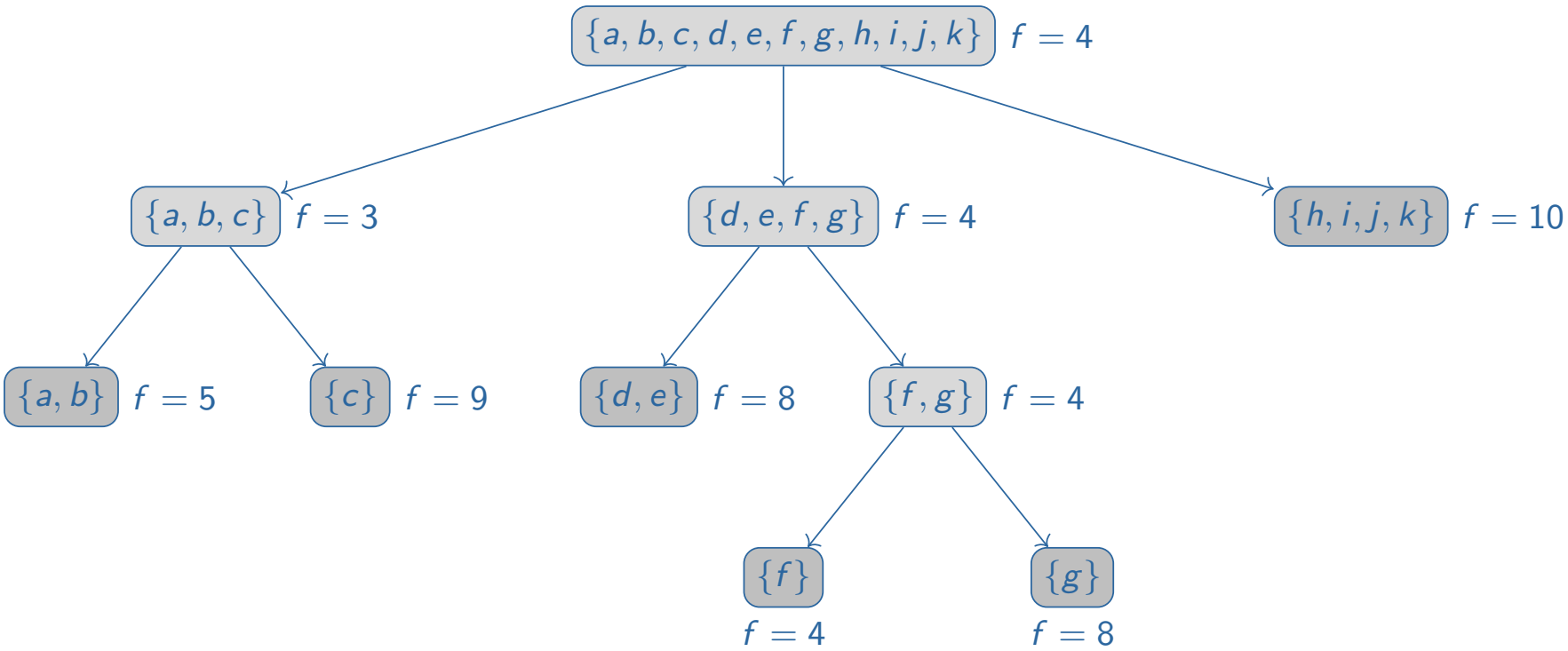


"Current" partition is $\{\{a, b\}, \{c\}, \{d, e, f, g\}, \{h, i, j, k\}\}$

The two steps of Branch-and-Bound

A Branch-and-Bound algorithm virtually searches through a part of the tree.

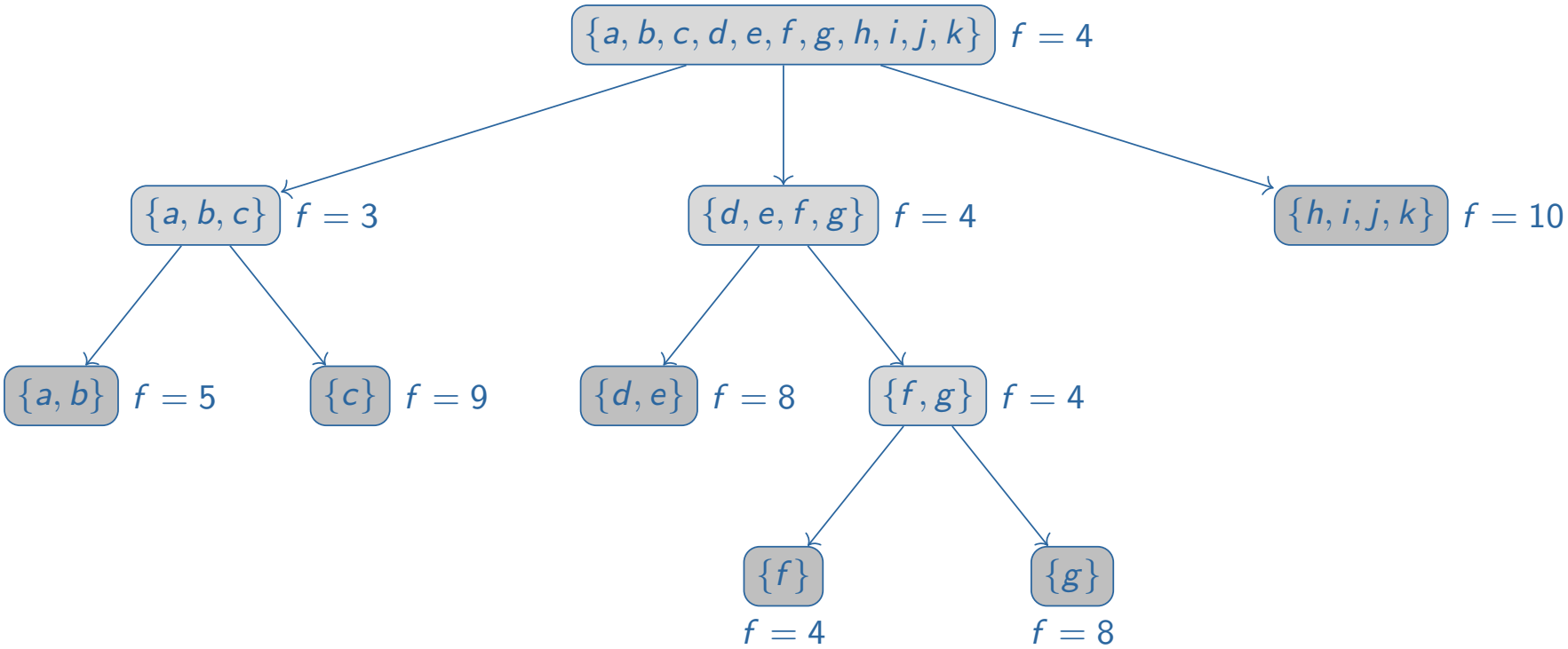
1. the branching step uses fonction f to select among the leaves the most promising subset of candidates and to generate its children (that is to refine this leaf node).
   $\Rightarrow$ the tree expansion is not fixed in advance, as opposed to backtracking.
2. the bounding step: each new leaf C is given its value $f(C)$.



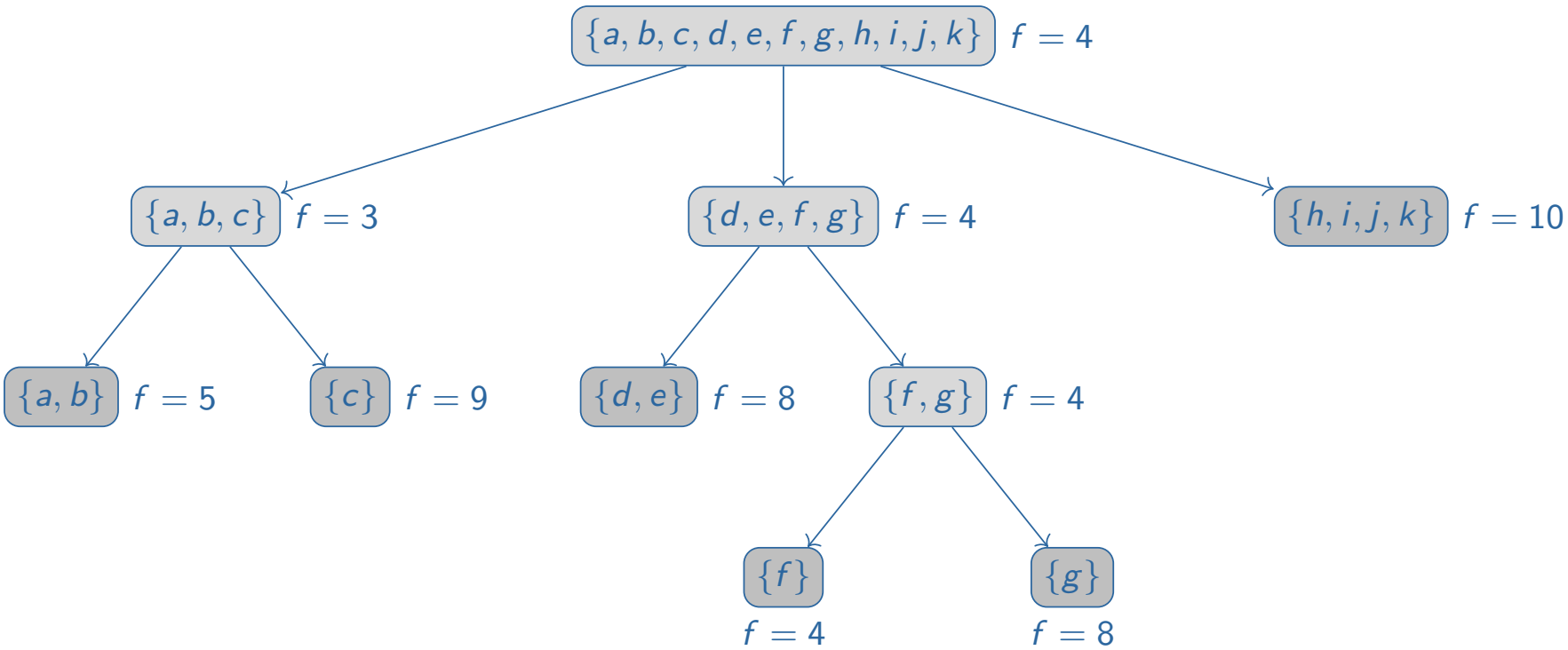
The two steps of Branch-and-Bound

Take a Priority Queue *OPEN* that sorts the leaves of the search in increasing values for f. Initially, the only leaf is $\mathcal{C}$, so $OPEN \leftarrow \{\mathcal{C}\}$.

1. *Branching* step:
   $\begin{cases} C \leftarrow OPEN.get \text{ with minimal } f(C) : OPEN.remove(C) \\ \textbf{if } C \text{ is a singelton } \{c\} \textbf{ then Returen } c \\ \textbf{else} \text{ expand } C \text{ into non-empty subsets } C_{i_1}, C_{i_2}, \ldots C_{i_m} \text{ and add them to } OPEN \end{cases}$
2. *Bounding* step: for each C_{i_k}, compute $f(C_{i_k})$; $OPEN.add(C_{i_k})$; goto 1.



Principles of Branch-and-Bound

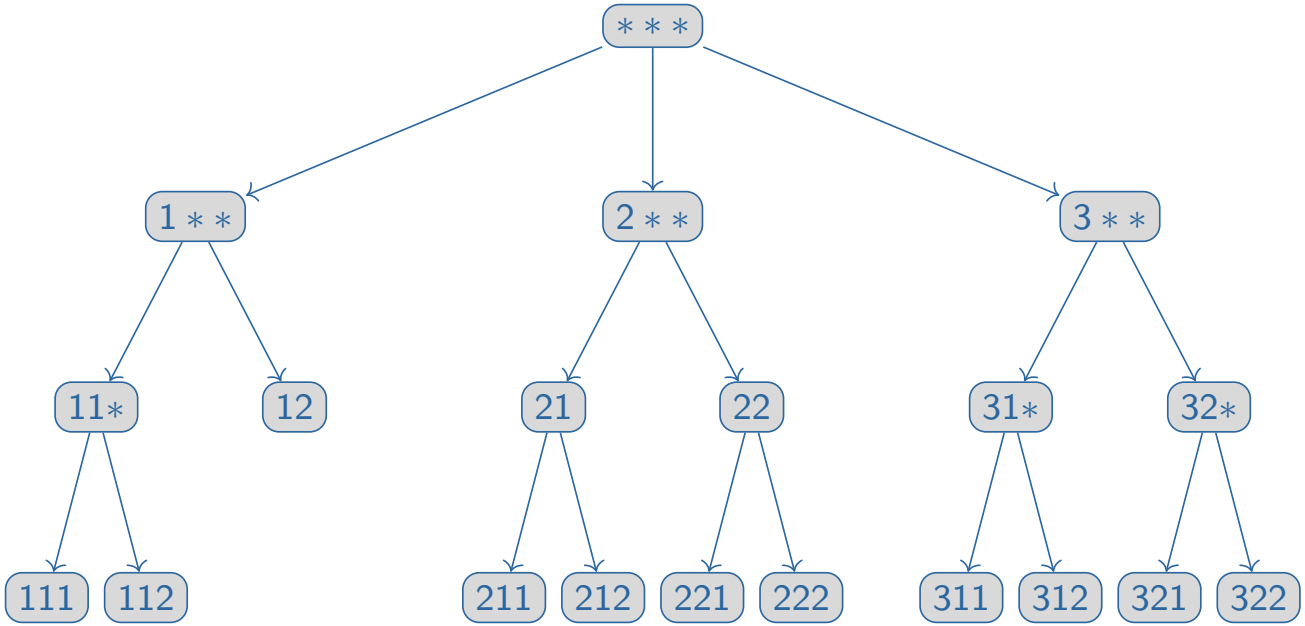


The two issues of the branching step

1. How to split a set C?
2. How to efficiently represent each node?

The answer depends on the problem to solve.
In many cases, we find an adaptation of the backtracking approach to expand a node:

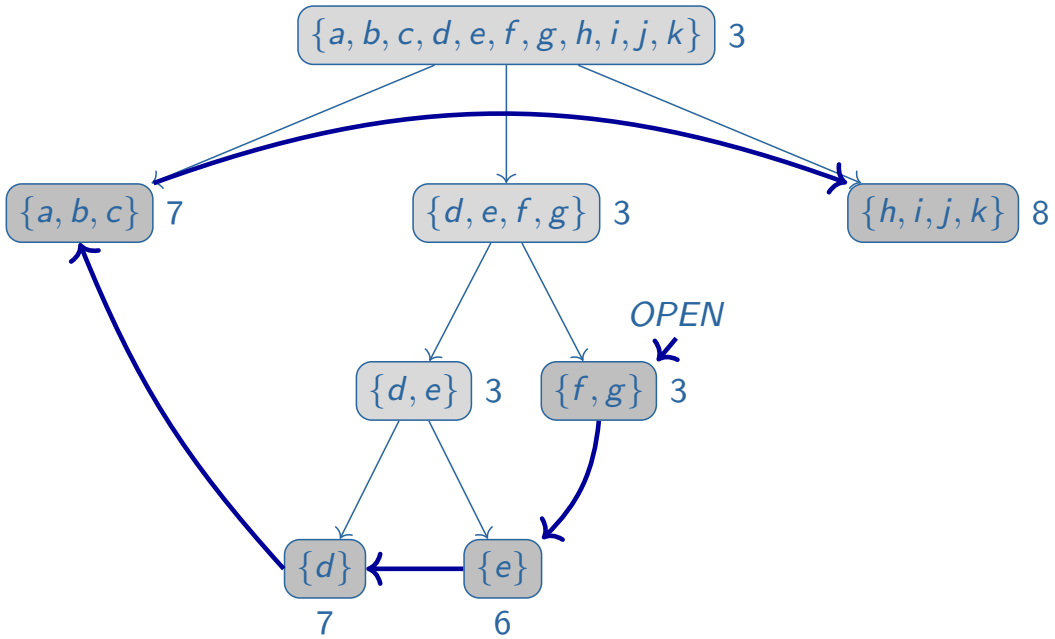
- The partially-filled vector $X[1..i]$ is seen as the set of all its completions i.e. all vectors $Y[1..n]$ such $Y[1..i] = X[1..i]$;
- The branching step amounts to instanciating component $X[i]$ in all possible manner.



The branching step

The set *OPEN* is implemented as a priority queue .

The node to expend next is *OPEN.first*.



The Branch-and-Bound boundaries lie in the size of *OPEN*

The bounding step

Before a new node is added to *OPEN*, we must compute its priority.

Definition

We extend $cost : \mathcal{C} \to \mathbf{R}$ *as* $cost : \mathcal{P}(\mathcal{C}) \setminus \emptyset \to \mathbf{R}$ *by:*

$$\begin{cases} cost(\{c\}) = cost(c) & \text{for every candidate } c \in C \\ cost(C) = min_{c \in C} cost(c) & \text{for } n > 1 \end{cases}$$

By definition, $cost(\mathcal{C})$ is the cost of a (best) solution. Also function *cost* may increase while we go down a branch of the tree because it satisfies $C' \subseteq C$ implies $cost(C) \leq cost(C')$.

Since function $cost(\mathcal{C})$ is unknown (this is what we search), we resort to an evaluation function f, called a *heuristics*, well-chosen by the designer of the Branch-and-Bound algorithm.

Theorem

If the evaluation function f *satisfies the following two conditions:*

(a) f *is positive and remains below cost :* $\text{for all } C \subseteq \mathcal{C},\ 0 \leq f(C) \leq cost(C)$;

(b) $\text{for every candidate } c \in \mathcal{C},\ f(\{c\}) = cost(c)$;

then the Branch-and-Bound algorithm that uses f *for the priority in OPEN returns an optimal solution.*

Template for Branch-and-Bound

We let $f(C)$ be an under-estimation of $cost(C)$ for each node C:

Algorithm basic Branch-and-Bound

```
1:  constants/types
2:  type: set 𝒞 = {...} and set ElmtPQ = {(C, p) | ∅ ⊂ C ⊆ 𝒞 and p ∈ R+}
3:  variables
4:  OPEN is a priority queue over ElmtPQ
5:  begin
6:  OPEN.new; OPEN.add(𝒞, f(𝒞)))
7:  while not(IsSingleton(OPEN.first)) do
8:      (C, p) ← OPEN.get; Algo        // this removes the element from OPEN
9:      for C' ∈ Children(C) do
10:         OPEN.add(C', f(C'))
11:     end for
12: end while
13: Print(OPEN.get)
14: end
```

ElmtPQ is the type of elements in the priority queue *OPEN*.

Correctness of Algorithm Branch-and-Bound

Algorithm basic Branch-and-Bound

```
1:  constants/types
2:  type: set 𝒞 = {...} and set ElmtPQ = {(C, p) | ∅ ⊂ C ⊆ 𝒞 and p ∈ R+}
3:  variables
4:  OPEN is a priority queue over ElmtPQ
5:  begin
6:  OPEN.new; OPEN.add(𝒞, f(𝒞)))
7:  while not(IsSingleton(OPEN.first)) do
8:      (C, p) ← OPEN.get; Algo        // this removes the element from OPEN
9:      for C′ ∈ Children(C) do
10:         OPEN.add(C′, f(C′))
11:     end for
12: end while
13: Print(OPEN.get)
14: end
```

Lemma

Algorithm Branch-and-Bound terminates.

Lemma

Loop invariant of line 7: "the priority queue OPEN contains a partition of $\mathcal{C}$".

Remarks on Branch-and-Bound algorithm

Algorithm basic Branch-and-Bound

```
1: constants/types
2: type: set C = {...} and set ELMTPQ = {(C, p) | ∅ ⊂ C ⊆ C and p ∈ R+}
3: variables
4: OPEN is a priority queue over ELMTPQ
5: begin
6: OPEN.new; OPEN.add(C, f(C)))
7: while not(IsSingleton(OPEN.first)) do
8:     (C, p) ← OPEN.get; Algo        // this removes the element from OPEN
9:     for C' ∈ Children(C) do
10:        OPEN.add(C', f(C'))
11:    end for
12: end while
13: Print(OPEN.get)
14: end
```

1. The current version of Branch-and-Bound puts all the candidates in *OPEN*, but it might be the case that some elements of $\mathcal{C}$ are not even candidates. Think of TSP when no Hamiltonian cycle exists in the graph.

Your turn!

What would you add in the algorithm ?

Remarks on Branch-and-Bound algorithm

1. The current version of Branch-and-Bound puts all the candidates in *OPEN*, but it might be the case that some elements of $\mathcal{C}$ are not even candidates. Think of TSP when no Hamiltonian cycle exists in the graph.

Algorithm basic Branch-and-Bound with possibly no admissible solution

```
1:  constants/types
2:  types: set C = {...} and set ElmtPQ = {(C, p) | ∅ ⊂ C ⊆ C and p ∈ R+}
3:  variables
4:  OPEN is a priority queue over ElmtPQ
5:  begin
6:  OPEN.new; OPEN.add(C, f(C)))
7:  while not(OPEN.isEmpty or then IsSingleton(OPEN.first)) do
8:      (C, p) ← OPEN.get;
9:      for C' ∈ Children(C) do
10:         if Admissible(C') then
11:             OPEN.add(C', f(C'))
12:         end if
13:     end for
14: end while
15: if OPEN.isEmpty then
16:     Print("No solution")
17: else                                          // IsSingleton(OPEN.first)
18:     Print(OPEN.get)
19: end if
20: end
```

Remarks on Branch-and-Bound algorithm

1. The current version of Branch-and-Bound puts all the candidates in *OPEN*, but it might be the case that some elements of $\mathcal{C}$ are not even candidates. Think of TSP when no Hamiltonian cycle exists in the graph.

Algorithm basic Branch-and-Bound with possibly no admissible solution

```
1: constants/types
2: types: set C = {...} and set ElmtPQ = {(C, p) | ∅ ⊂ C ⊆ C and p ∈ R+}
3: variables
4: OPEN is a priority queue over ElmtPQ
5: begin
6: OPEN.new; OPEN.add(C, f(C)))
7: while not(OPEN.isEmpty or then IsSingleton(OPEN.first)) do
8:     (C, p) ← OPEN.get;
9:     for C' ∈ Children(C) do
10:        if Admissible(C') then
11:            OPEN.add(C', f(C'))
12:        end if
13:    end for
14: end while
15: if OPEN.isEmpty then
16:    Print("No solution")
17: else                                          // IsSingleton(OPEN.first)
18:    Print(OPEN.get)
19: end if
20: end
```

Your turn!

What would the loop invariant become ?

Remarks on Branch-and-Bound algorithm

1. The current version of Branch-and-Bound puts all the candidates in *OPEN*, but it might be the case that some elements of $\mathcal{C}$ are not even candidates. Think of TSP when no Hamiltonian cycle exists in the graph.

Algorithm basic Branch-and-Bound with possibly no admissible solution

```
1: constants/types
2: types: set C = {...} and set ElmtPQ = {(C, p) | ∅ ⊂ C ⊆ C and p ∈ R+}
3: variables
4: OPEN is a priority queue over ElmtPQ
5: begin
6: OPEN.new; OPEN.add(C, f(C)))
7: while not(OPEN.isEmpty or then IsSingleton(OPEN.first)) do
8:     (C, p) ← OPEN.get;
9:     for C' ∈ Children(C) do
10:        if Admissible(C') then
11:            OPEN.add(C', f(C'))
12:        end if
13:    end for
14: end while
15: if OPEN.isEmpty then
16:    Print("No solution")
17: else                                   // IsSingleton(OPEN.first)
18:    Print(OPEN.get)
19: end if
20: end
```

Your turn!

What would the loop invariant become ?

Algorithm Branch-and-Bound with pruning

We may exploit value $f(C)$ to prune the tree.

Your turn!

Do modify Algorithm Branch-and-Bound to take pruning into account.

Algorithm basic Branch-and-Bound with possibly no admissible solution

```
1:  constants/types
2:  types: set 𝒞 = {...} and set ElmtPQ = {(C, p) | ∅ ⊂ C ⊆ 𝒞 and p ∈ R⁺}
3:  variables
4:  OPEN is a priority queue over ElmtPQ
5:  begin
6:  OPEN.new; OPEN.add(𝒞, f(𝒞)))
7:  while not(OPEN.isEmpty or then IsSingleton(OPEN.first)) do
8:      (C, p) ← OPEN.get;
9:      for C′ ∈ Children(C) do
10:         if Admissible(C′) then
11:             OPEN.add(C′, f(C′))
12:         end if
13:     end for
14: end while
15: if OPEN.isEmpty then
16:     Print("No solution")
17: else                                   // IsSingleton(OPEN.first)
18:     Print(OPEN.get)
19: end if
```

f v.s. *cost*: an interesting case

It may happen that each node of the tree has a value such that the cost $cost(c)$ of a candidate c corresponds to the sum of the edge cost from the root node $\mathcal{C}$ to the leaf node $\{c\}$. Think of the shortest path problem.

Then $cost(C)$, where $C \in \mathcal{C}$, can be decomposed as follows:
Write:

- $g(C)$ for the cost accumulated along of the path in the search tree – from $\mathcal{C}$ to C, and
- $cost_{future}(C)$ for the best cost that remains in the future.

Then set

$$cost(C) = g(C) + cost_{future}(C)$$

Remark

Notice that this particular case of cost is compatible with the generic version of the Branch-and-Bound algorithm.

For the heuristic fonction f using in the priority queue, we can then take

$$f(C) := g(C) + h(C)$$

where h under-estimates what can be expected "in the future", namely $cost_{future}$.

Heuristic functions: Branch-and-Bound admissibility

Theorem (Sufficient conditions for Branch-and-Bound admissibility)

Let f be an evaluation function defined by:

$$f(C) = g(C) + h(C)$$

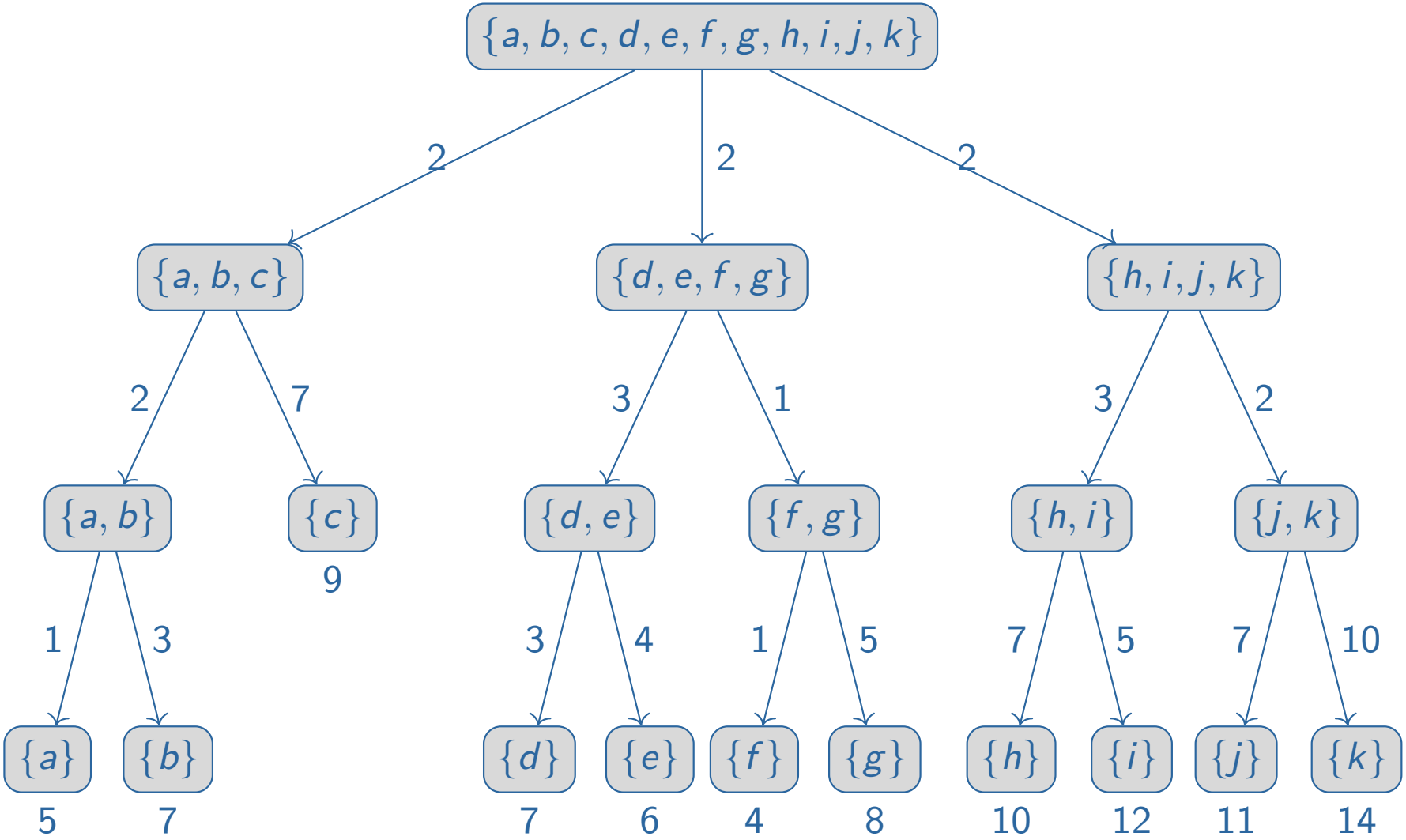
If the heuristic function satisfies:

$$0 \leq f(C) \leq cost(C), \text{ for all } C \subseteq \mathcal{C}$$

then Branch-and-Bound is correct.

Heuristic function: an example

with *cost* at the leaves equal to the sum of the edge values. Run Branch-and-Bound.



Heuritic functions: impact on Branch-and-Bound algorithm

If we write $v(C, C')$ for the value of edge (C, C') in the tree, the template for Algorithm Branch-and-Bound is as follows.

Algorithm Branch-and-Bound with heuristics

```
1:  constants
2:  C = {...} and ElmtPQ = {(C, past, past + future) | ∅ ⊂ C ⊆ C and past ∈
    R+ and future ∈ R+}
3:  variables
4:  OPEN ∈ PQ(ElmtPQ)
5:  begin
6:  OPEN.new; OPEN.add(C, 0, 0 + f(C))
7:  while not(OPEN.isEmpty or then IsSingleton(OPEN.first)) do
8:      (C, past, past + future) ← OPEN.get;
9:      for C' ∈ Children(C) do
10:         OPEN.add(C', past + v(C, C'), past + v(C, C') + h(C'))
11:     end for
12: end while
13: if OPEN.isempty then
14:     Print("No solution")
15: else                                          // IsSingleton(OPEN.first)
16:     Print(OPEN.get)
17: end if
18: end
```

Heuristic functions: comparison

The more the value h is closed to $cost_{future}$, the more efficient is the algorithm.

Lemma

Let f_1 and f_2 be two heuristic functions such that

$$f_1(C) \leq f_2(C) \leq cost(C) \qquad \text{for all } C \subseteq \mathcal{C}$$

and let A_1 (resp. A_2) be a version of the Branch-and-Bound algorithm that uses heuristic f_1 (resp. f_2).

Then, if node C is not chosen by A_1, neither it will be by A_2.

A trivial but ineffective choice is to take $h = 0$, namely we cannot tell anything about the future.

How do you choose the heuristic function $f(C)$?

- It must satisfy $f(C) \leq cost(C)$, for all $C \subseteq \mathcal{C}$
- It should be computed efficiently, say polynomial;
- Very powerful techniques rely on a relaxation of the original problem so that a greedy algorithm exists.

KNAPSACK (KP)

Input : n items, for each item i two values w_i (≥ 0 its weight) et p_i (its profit), a constant $K \geq 0$.
Output : An assignement of the variables $x_1, x_2, \ldots, x_n$ in $\{0, 1\}$ such that $\Sigma_{i=1}^{n} w_i.x_i \leq K$ and which maximizes the total profit $\Sigma_{i=1}^{n} p_i.x_i$.

and

FRACTIONAL KNAPSACK (KP*frac*)

Input : n items, for each item i two values w_i (≥ 0 its weight) et p_i (its profit), a constant $K \geq 0$.
Output : An assignement of the variables $x_1, x_2, \ldots, x_n$ in $[0, 1]$ such that $\Sigma_{i=1}^{n} w_i.x_i \leq K$ and which maximizes the total profit $\Sigma_{i=1}^{n} p_i.x_i$.

Example of the Job Assignment Problem

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

Your turn!

Take a look at more abstract Minimum weight perfect matching problem. What do you think?

Take a look at `https://www.youtube.com/watch?v=F4vI_qc_u0Q`

Your turn!

Formalize everything that this video presents.

Other kind of heuristics

Which branch should be developped first can be decided very locally.

Again: Job Assignement problem

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Worker A takes 8 units of time to finish job 4.

An example job assignment problem. Green values show optimal job assignment that is A-Job4, B-Job1 C-Job3 and D-Job4

Job Assignment Problem using Branch And Bound

For the Job Assignment Problem, there are two approaches to calculate the heuristic function:

1. For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row), or
2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers (take minimum entry from each column).

see `https://www.geeksforgeeks.org/job-assignment-problem-using-branch-and-bound/`

Job Assignment Problem using Branch And Bound

For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row).

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	6	4	**3**	7
C	5	8	1	8
D	7	6	9	4

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	6	4	**3**	7
C	**5**	8	1	8
D	7	6	9	**4**

Recall that a best solution had cost 13.

Videos to watch

Applications of Branch-and-Bound

- Integer linear programming
  https://www.youtube.com/watch?v=F-jFk1BczYg
- Job assignement https://www.geeksforgeeks.org/job-assignment-problem-using-branch-and-bound/
  https://www.youtube.com/watch?v=F4vI_qc_u0Q
- Knapsack problem
  https://fr.coursera.org/learn/discrete-optimization/lecture/66010/knapsack-5-relaxation-branch-and-bound (incomplete)
- A* https://duckduckgo.com/?q=branch+and+bound&t=h_&ia=videos&iax=videos&iai=gGQ-vAmdAOI

Approximation Algorithms

Main bibliography

Readings:

- Chapter 11 of "Algorithm Design" by J. Kleinberg and E. Tardos [KT05].
- Sections 1-2, Chapter 9 of "Algorithms" by S. Dasgupta, C.H. Papadimitriou, and U. Vazirani [DPV06].

How should we design algorithms for problems where polynomial time seems hopeless?

We may use approximation algorithms, which run in polynomial time and find solutions that are guaranteed to be close to optimal.

At the same time, we will be interested in proving that our algorithms find solutions that are guaranteed to be close to the optimum.

Chapter 11 of Kleinberg and Tardos

We will consider four general techniques for designing approximation algorithms.

1. Greedy algorithms. These algorithms are simple and fast with the challenge being to find a greedy rule that leads to solutions provably close to optimal.
2. Pricing method. This approach is motivated by an economic perspective; we consider a price one has to pay to enforce each constraint of the problem. The pricing method is often referred to as the primal-dual technique, a term inherited from the study of linear programming.
3. Linear programming and rounding. We exploit the relationship between the computational feasibility of linear programming and the expressive power of its more difficult cousin, integer programming.
4. Dynamic programming on a rounded version of the input. This technique can lead to extremely good approximations.

We now use slides from Kleinberg and Tardos at `http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/11ApproximationAlgorithms.pdf`.

The approximability hierarchy

Given any NPTIME-complete optimization problem, we seek the best approximation algorithm possible.
Failing this, we try to prove lower bounds on the approximation ratios that are achievable in polynomial time (we carried out such a proof for the general TSP).
NPTIME-complete optimization problems are classified as follows:

1. Those for which, like the TSP, no finite approximation ratio exists.
2. Those for which an approximation ratio exists, but there are limits to how small this can be. VerCov, k-CLUSTER (Center Selection), and the TSP with triangle inequality belong here.
   For such problems we have not established limits to their approximability, but these limits do exist, and their proofs constitute some of the most sophisticated results in this field.
3. Those for which approximability has no limit, and polynomial approximation algorithms with error ratios arbitrarily close to zero exist, KP resides here.
4. Finally, there is another class of problems, between the first two given here, for which the approximation ratio is about $log(n)$.
   SetCov is an example (find out how this works).

Final remarks on approximation algorithms

- All these results on approximation algorithms is contingent upon the assumption $\mathrm{PTIME} \neq \mathrm{NPTIME}$.

  Failing this, the hierarchy collapses down to PTIME, and all NPTIME-complete optimization problems can be solved exactly in polynomial time.

- Often these approximation algorithms, or their variants, perform much better on typical instances than their worst-case approximation ratio.

Local Search

What is Local Search?

Local search is a very general technique.

- It describes any algorithm that "explores" the space of possible solutions in a sequential fashion, moving in one step from a current solution to a "nearby" one.
- The generality and flexibility of this notion has the advantage that it is not difficult to design a local search approach to almost any computationally hard problem.
- The counterbalancing disadvantage is that it is often very difficult to say anything precise or provable about the quality of the solutions that a local search algorithm finds, and consequently very hard to tell whether one is using a good local search algorithm or a poor one.

Readings [KT05, Chapter 12] and [DPV06, Section 3 of Chapter 9]

We now use slides from Kleinberg and Tardos.


S. Dasgupta, C.H. Papadimitriou, and U. Vazirani.
Algorithms.
McGraw-Hill, Inc., 2006.


J. Kleinberg and E. Tardos.
Algorithm Design. 2005.
Addison Wesley, 2005.