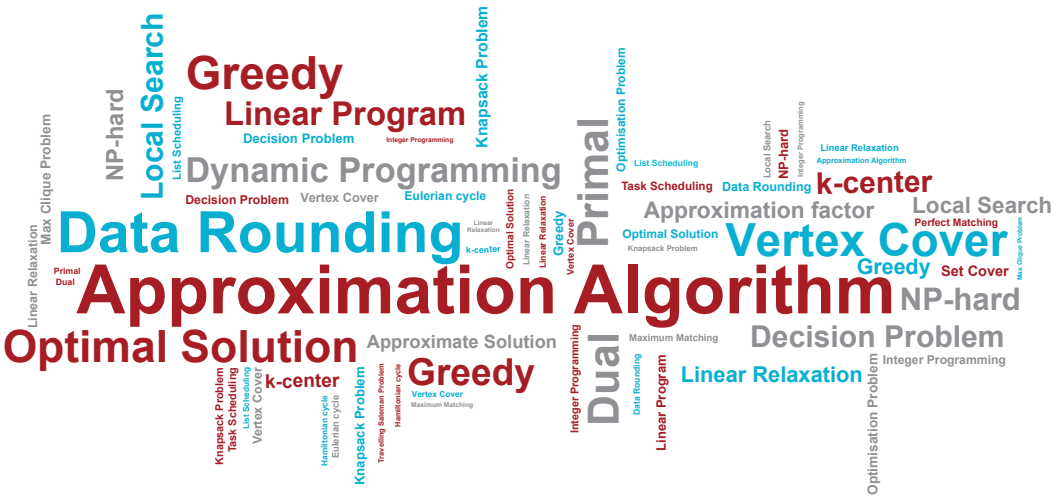



# Algorithmes d'approximation

*Cours*

*Xavier Dubuc*
(xavier.dubuc@umons.ac.be)

UMONS
Université de Mons

Faculté
des Sciences

22 novembre 2018

**Contributeurs :**

- 2016
  - Benoit Debled
  - Julien Delplanque (julien.delplanque@student.umons.ac.be)
  - Anthony Rouneau

# Table des matières

# 1 Introduction

## Introduction au cours

- Examen oral → tirer une question et préparation de la question avec les notes, puis on présente.
- Travail personnel après Pâques (présentation).

## Références

- **Williamson** & **Shmoys**
  "*The Design of Approximation Algorithms*"
  Cambridge University (version gratuite pdf : www.designofapproxalgs.com)
- **Vazirani**
  "*Approximation Algorithms*"
  Springer, 2001 *(difficile à comprendre)*
- **Ausiello**, **Cescenzi**, **Gambozi**, **Kann**, et al.
  "*Complexity and Approximation : Combinatorial Optimization Problems and their approximability properties*"
  Springer, 1999.

## 1.1 Introduction à la matière

On va voir les problèmes d'optimisation, en particulier les problèmes *NP-difficiles* (∼*NP-complets*, on verra les différences plus tard). A moins que $P = NP$, on ne peut résoudre ces problèmes d'optimisation de manière optimale, c'est-à-dire qu'il n'existe pas (encore ?) d'algorithme *efficace* (càd polynomial) pour trouver une solution optimale à ces problèmes.

**Que peut-on faire dans ces cas là ?** *(un des objectifs du cours)*

La première réponse est "*les heuristiques*". Elles sont rapides mais elles donnent une solution qui est une *"bonne"* solution et pas **la solution optimale** (en temps polynomial).
Dans ce cours on va prouver que l'heuristique donne une solution qui est proche de la solution optimale.

On utilise une analogie à la phrase bien connue :

*“ Rapide. Bon marché. Fiable. Choisissez-en deux ”*

⟶ Il faut en effet sacrifier un des 3 critères, c'est-à-dire que si P ≠ NP, on ne peut pas avoir simultanément un algorithme qui :

1. trouve la solution optimale ;
2. travaille en temps polynomial ;
3. fonctionne pour tout instance du problème (toute entrée possible).

On a donc 3 choix :

- **Laisser tomber** 3
  → Ce n'est pas toujours applicable en pratique car ça donne pas une solution générale.
  ⇝ Ex : la coloration d'un graphe où on s'intéresse qu'aux graphes bipartis par exemple.
- **Laisser tomber** 2
  → Ce n'est pas toujours une bonne idée.
  ⇝ Ex : $A^*$ en **IA** ou encore le Branch & Bound.
- **Laisser tomber** 1
  → En utilisant les métaheuristiques et les heuristiques.
  ⇝ Ex : la recherche Tabou.

**Dans ce cours on va s'intéresser aux heuristiques avec une garantie de performance.**

## 1.2 Objectifs du cours

1. Savoir que faire pour résoudre un problème NP-difficile.
2. Découvrir et revoir des problèmes "paradigmatiques" *(problèmes classiques, exemplaires, simplifiés, comme le voyageur de commerce par exemple ; qui ont beaucoup d'applications)*.
3. Tous les problèmes intraitables ne sont pas les mêmes. Les problèmes *NP-complet* sont identiques d'un point de vue "résolution exacte" mais peuvent être très différents d'un point de vue approximabilité *(certains algorithmes peuvent donner une très bonne approximation, d'autres une moins bonne et d'autres encore, aucune)*. L'objectif consistera à savoir différencier ces algorithmes.
4. Apprendre des techniques de conception et d'analyse d'algorithmes d'approximation. ($\rightsquigarrow$ avoir une "boite à outils", où les outils sont des algorithmes et heuristiques applicables à un grand nombre de problèmes).

## 1.3 Problèmes d'optimisation

**Définition 1.1 Problème d'optimisation** *(informel)*
*C'est un type de problème où il est demandé de trouver la solution optimale parmi les solutions réalisables.*

**Définition 1.2 Problème d'optimisation** *(formel)*
*Un problème d'optimisation $P$ est spécifié par $(I_P, SOL_P, m_P, goal_P)$ tels que :*

- $I_P$ *est un ensemble d'instances de* $P$.
  $\rightarrow$ *e.g. pour la coloration de graphe, tous les couples (graphe,entier).*
- $SOL_P$ *est une fonction qui associe à chaque instance* $x \in I_P$ *un ensemble de solutions réalisables de* $x$ *($SOL_P(x)$).*
  $\rightarrow$ *e.g. pour la coloration de graphe, ensemble des colorations légales possibles.*
- $m_P$ *est une fonction de mesure ou fonction objectif définie pour les paires* $(x, y)$ *tq* $x \in I_p$ *et* $y \in SOL_P(x)$. *Pour toute paire* $(x, y)$, $m_P(x, y)$ *donne une valeur non-négative.*
- $goal_P \in MIN, MAX$ *spécifiant si* $P$ *est une problème de minimisation ou de maximisation.*

$\rightsquigarrow$ *Quand le contexte est clair, on peut laisser tomber le* $_P$ *dans les notations.*
$\rightsquigarrow$ $SOL_P^*(x)$ *est l'ensemble des solutions optimales de* $x \in I_P$.

---

**Problème 1.1 MIN VERTEX_COVER (VC)**

* **Instance** : *graphe* $G = (V, E)$,
* **Solution** : *un ensemble de sommets* $U \subseteq V$ *tel que* $\forall (v_i, v_j) \in E$, $v_i \in U$ *ou* $v_j \in U$.
* **Mesure** : $|U|$

---

**Exemple 1.1** *Graphe étoile à* $n$ *sommets,* $S_n$.

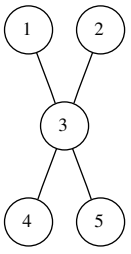



FIGURE 1 – $S_n$ $(n = 5)$, exemple pour VC

$C = \{3\}$ est une solution réalisable.
$VC(S_n) = 1 \ \forall n \geq 2$

**Exemple 1.2** *Graphe complet à $n$ sommets, $K_n$.*

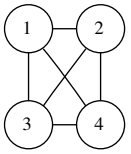



FIGURE 2 – $K_n$ $(n = 4)$, exemple pour VC

$C = \{1, 2, 3\}$ est une solution réalisable.
$VC(S_n) = n - 1 \ \forall n \geq 2$
Pour justifier cela, il suffit de voir que si on enlève toutes les arêtes concernant un sommet, on se retrouve à nouveau avec un graphe complet, on peut ainsi itérer récursivement jusqu'au cas de base $K_2$ où on sait que $VC(K_2) = 1$.

**Exemple 1.3** *Chemin à $n$ sommets, $P_n$.*

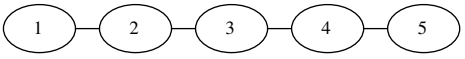



FIGURE 3 – $P_n$ $(n = 5)$, exemple pour VC

$C = \{2, 4\}$ est une solution réalisable.
$VC(P_n) = \left\lfloor \frac{n}{2} \right\rfloor \ \forall n \geq 2$

**Exemple 1.4** *Cycle à $n$ sommets, $C_n$.*

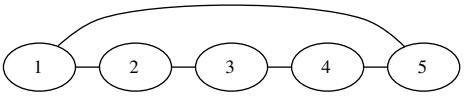



FIGURE 4 – $C_n$ $(n = 5)$, exemple pour VC

$C = \{1, 3, 5\}$ est une solution réalisable.
$VC(C_n) = \left\lceil \frac{n}{2} \right\rceil \ \forall n \geq 2$

**Exemple 1.5** *Graphe biparti complet à $n+m$ sommets, $K_{n,m}$.*

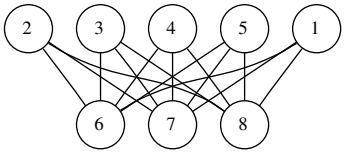



FIGURE 5 – $K_{n,m}$ $(n = 5,\ m = 3)$, exemple pour VC

$C = \{6, 7, 8\}$ est une solution réalisable.
$VC(K_{n,m}) = \min(n, m)\ \forall n, m \geq 1$

---

**Problème 1.2 MAX MATCHING (MM)**

* **Instance** *: graphe $G = (V, E)$,*
* **Solution** *: un ensemble d'arcs $F \subseteq E$ qui est un matching, c'est-à-dire qu'il n'y a pas 2 arêtes incidentes dans $F$ (extrémités communes).*
* **Mesure** *: $|F|$*

---

**Exemple 1.6** *Valeurs pour MM pour différents types de graphes :*

- $MM(S_n) = 1$
- $MM(C_n) = \lfloor \frac{n}{2} \rfloor$
- $MM(P_n) = \lfloor \frac{n}{2} \rfloor$
- $MM(K_{n,n}) = n$

**Remarques 1.1** *:*

- *Quand un graphe $G$ est biparti, $VC(G) = MM(G)$, sinon $MM(G) \leq VC(G)$ avec égalité ssi $G$ est* **biparti**.
- *VC est* **NP-Complet**, *MM ne l'est pas, c'est-à-dire qu'il existe un algorithme polynomial pour calculer MM. On va donc pouvoir utiliser MM pour approximer VC.*

## 1.4 Rappels et complexité des problèmes d'optimisation

L'intuition que l'on suit est que **VERTEX COVER** est beaucoup plus "difficile" que **MAXIMUM MATCHING** (pour lequel il existe des algorithmes polynomiaux).

### Rappels

- Un algorithme pour un problème est dit **polynomial** si le nombre d'instructions pour résoudre ce problème peut être borné par un polynôme en la taille de l'entrée.
- Un **problème de décision** est un problème dont la sortie est soit "Yes" soit "No".
- La classe $\mathcal{P}$ contient les problèmes de décisions qui admettent des algorithmes polynomiaux pour les résoudre.
- La classe $\mathcal{NP}$ est l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial de manière non-déterministe *(c.f. cours de Calculabilité et Complexité)*.
- De manière moins formelle, la classe $\mathcal{NP}$ est l'ensemble des problèmes de décision tels que pour toute instance "Yes" il y a une preuve "facile" (algorithme polynomial) pour montrer que la réponse est "Yes" *(c.f. métaphore avec la musique : "Je suis capable de vérifier qu'une musique est une bonne musique mais je suis incapable de la composer moi-même.")* .
- Tout problème d'optimisation peut être exprimé sous la forme d'un problème de décision.

→ Exemple :

**Problème 1.3 MIN VC-DECISION (VC-DEC)**
* **Instance** : *Graphe* $G = (V, E)$, $k \in \mathbb{N}$
* **Question** : *Existe-t-il une converture de* $G$ *de taille* $\leq k$ *?*

→ De manière générale, si $goal = min(max)$, demander, pour un certain $k > 0$, s'il existe une solution réalisable $y$ pour une instance $x$ telle que $m(x, y) \leq (\geq) k$ ($m$ étant la mesure).

### Définitions

- Un problème de décision $A$ se **réduit polynomialement** en $B$, noté $A \propto B$, s'il existe un algorithme polynomial permettant de transformer toute instance de $A$ en une instance de $B$ correspondante.
- Un problème de décision $B$ est dit $\mathcal{NP}$-**complet** si $B \in \mathcal{NP}$ et $\forall$ problème $A$ $\mathcal{NP}$-complet, il existe une réduction polynomiale $A \propto B$.
- Une **preuve par réduction** que $A$ est $\mathcal{NP}$-complet se fait en 2 étapes :
  1. Prouver que $A$ est dans $\mathcal{NP}$ *(il faut donc prouver que la vérification est polynomiale)* ;
  2. $\exists$ un problème $B$ tel que $B \propto A$ pour un problème $B$ connu comme étant $\mathcal{NP}$-complet.
- Soit un problème de décision ou d'optimisation $A$. On dit qu'un algorithme possède $A$ comme **oracle** si on suppose que l'algorithme peut résoudre une instance de $A$ en une seule instruction ($O(1)$).
- Un problème $A$ est $\mathcal{NP}$-**difficile** *($\mathcal{NP}$-hard)* s'il existe un algorithme polynomial pour un problème $\mathcal{NP}$-complet $B$ quand cet algorithme possède $A$ comme oracle.
  *(Intuition : Si $A$ était facile, alors $B$ le serait aussi)*
  Exemple : **VC** est $\mathcal{NP}$-difficile car si on connait **VC(G)** en temps constant, alors le problème **VC-décision** devient polynomial.

## 1.5 Algorithmes d'approximation : qu'est-ce ? Pourquoi ?

*" Probably close to optimal, but polynomial. "*

#### Définition

Un **algorithme d'**$\alpha$**-approximation** pour un problème d'optimisation est un algorithme polynomial qui pour toutes les instances du problème produit une solution dont la valeur est proche de la solution optimale à un facteur $\alpha$ près.

#### Remarques

- On va suivre la convention suivante :
  * $\boxed{\alpha > 1}$ si **MIN**, par exemple, un algorithme de 2-approximation implique que la solution de l'algorithme est le double de la solution optimale dans le pire des cas,
  * $\boxed{\alpha < 1}$ si **MAX**, par exemple, un algorithme de $\frac{1}{2}$-approximation implique que la solution de l'algorithme est la moitié de la solution optimale dans le pire des cas.
- $\alpha$ est la **garantie de performance**, aussi appelé **facteur** ou **ratio d'approximation**.
  Ce facteur n'est pas toujours constant, il peut dépendre de la taille de l'entrée.

#### Pourquoi ?

Nous donnons ici 6 raisons :

1. il y avait un certain besoin d'algorithmes pour obtenir des solutions rapidement aux problèmes d'optimisation ;
2. souvent la conception d'algorithmes se base sur des modèles idéalisés mais peuvent donner des idées pour les "vrais" problèmes ;
3. base mathématique rigoureuse pour étudier les heuristiques ;
4. donne des garanties :
   - à priori *(facteur $\alpha$)*,

- à fortiori dans certains cas, *(voir plus loin)* ;

5. donne un moyen de mesurer le niveau de difficultés des problèmes d'optimisation ;
6. c'est "fun".

### 1.5.1 Définitions

- **Un schéma d'approximation en temps polynomial** *(polynomial time approximation scheme* (**PTAS**)) est une famille d'algorithmes $A_\epsilon$, où il y a un algorithme pour tout $\epsilon > 0$ tel que $A_\epsilon$ est un algorithme de :
  * *$(1+\epsilon)$-approximation* si c'est une **MIN**imisation,
  * *$(1-\epsilon)$-approximation* si c'est une **MAX**imisation.
    $\rightarrow$ Exemple : problème du sac à dos et version euclidienne du TSP (voyageur de commerce).
- Il existe une classe (**MAX SNP**) de problèmes telle que :
  **Théorème 1.1** *Pour tout problème $MAXSNP$-difficile, il n'y a pas de PTAS (sauf si $P = NP$).*
- Il existe d'autres problèmes encore plus difficile. Par exemple :

---

**Problème 1.4 MAX CLIQUE**
* **Instance** *: graphe $G = (V, E)$,*
* **Solution** *: un sous-ensemble de sommets $S \subseteq V$ tel que $S$ est une* **clique** *(sous-graphe complet),*
* **Mesure** *: $|S|$*

---

**Théorème 1.2** *Soit $n = |V|$ et $0 < \epsilon < 1$ n'importe quelle constante, alors il n'existe pas d'algorithme d'approximation avec un facteur $O(n^{\epsilon-1})$ pour* **MAX CLIQUE** *(sauf si $P = NP$).*
**Commentaires** :

1. si $\epsilon = 1$, $\exists$ un algorithme d'approximation en $O(1)$,
2. si $\epsilon = 0$, $\exists$ un algorithme d'approximation en $O(\frac{1}{n})$,
   $\rightarrow$ $OPT = n$ et $SOL = 1 \Rightarrow \frac{SOL}{OPT} = \frac{1}{n}$
   $\rightarrow$ **Algorithme idiot et très mauvais** mais **on ne peut pas faire mieux que lui !**
   *Tout algorithme serait donc aussi bon que de prendre un seul sommet, on ne peut pas améliorer ce facteur d'approximation.*

$\hookrightarrow$ Voir exercices dans l'annexe A.

# 2 Set Cover et survol des techniques

Ce problème possède diverses applications dans la vie de tous les jours. Par exemple, les antivirus ou encore les "*Houses of Graphs*".

---

**Problème 2.1 MIN SET_COVER (SC)**

* **Instance** :
  — *Un ensemble* $E = \{e_1, e_2, \ldots, e_n\}$ *de* $n$ *éléments,*
  — *des sous-ensembles* $S_1$, $S_2$, …, $S_m$ *où chaque* $S_j \subseteq E$,
  — *un poids non-négatif* $w_j \geq 0$ *pour chaque sous-ensemble* $S_j$.
* **Solution** : *Une collection* $I$ *de sous-ensembles qui couvrent* $E$.
  *C'est-à-dire* $I \subseteq \{1, 2, \ldots, n\}$ *telle que* $\bigcup_{j \in I} S_j = E$.
* **Mesure** : $\sum_{j \in I} w_j$

---

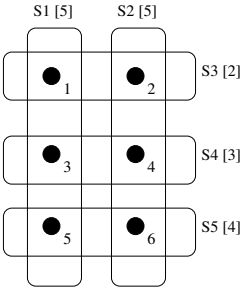



FIGURE 6 – Exemple d'instance de SC

$I_{OPT} = \{S_3, S_4, S_5\}$.
Et si $w_j = 1, \forall j$ *(version non-pondérée du problème)*, alors $I_{OPT} = \{S_1, S_2\}$.

## 2.1 Programmation linéaire et Set Cover

**Définition 2.1 (Programme Linéaire)** *Un programme linéaire (PL) est spécifié par :*

1. *des variables de décision,*
2. *des contraintes sous la forme d'inégalités/égalités linéaires,*
3. *une fonction objectif linéaire.*

*Un* **PL en nombres entiers (IP)** *impose une contrainte d'intégralité supplémentaire sur certaines variables.*
*Un* **PL 0-1** *impose une contrainte supplémentaire de valeur booléenne sur certaines variables.*

**Propriété 2.1** *Un PL où toutes les variables sont continues peut être résolus en temps polynomial (non pas via le simplexe mais via la méthode des points intérieurs ou via l'algorithme ellipsoïdale par exemple (il en existe d'autres)).*

⇒ **Ils appartiennent donc à** $\mathcal{P}$. *Un IP (Integer Programming) par contre, en général, est* $\mathcal{NP}$*-difficile.*

**Propriété 2.2** *Tout IP peut être* **relaxé**.
*Par exemple :* $x_j \in \{0, 1\} \Rightarrow_{relaxation} x_j \geq 0$
*(le* $x_j \leq 1$ *n'est pas utile car dès que c'est plus* $0$*, on sait qu'elle n'est plus à la valeur booléenne* $0$*).*

Grâce à cette idée de relaxation on peut dégager un algorithme qui semble être un algorithme d'approximation pour résoudre un problème **IP(*)** :

**Algorithme 1** RelaxationApprox

1: Résoudre le problème relaxé (**LP(**)**)
2: Arrondir la solution optimale trouvée.

Deux questions se posent alors : *"Comment arrondir ?"* et *"Quel est le facteur $\alpha$ ?"*.
On sait que cet algorithme est polynomial si l'algorithme utilisé pour arrondir est dans $\mathcal{P}$.
On sait également que **LP(**)** est une relaxation de **IP(*)**, ce qui signifie que :

- toute solution réalisable d'**IP(*)** est une solution réalisable de **LP(**)**,
- la valeur de toute solution réalisable dans **IP(*)** sera égale à la même valeur dans **LP(**)**.

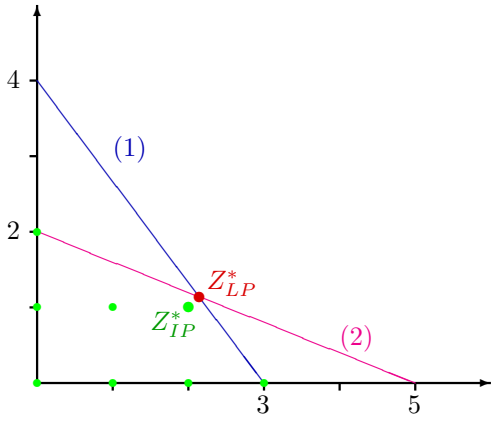
$\Rightarrow$ **LP(**)** a plus de solutions réalisables que **IP(*)**.
En d'autres mots, dans le cas d'une **MIN**imisation, si :

- $Z^*_{LP}$ est la solution optimale pour le **LP**
- $Z^*_{IP}$ est la solution optimale pour le **IP**

alors on a $\boxed{Z^*_{LP} \leq Z^*_{IP} = OPT}$.

**Exemple 2.1**

$$\max\ x_1 + x_2$$
$$\textbf{s.l.c}\ 4x_1 + 3x_2 \leq 12\ (1)$$
$$2x_1 + 5x_2 \leq 10\ (2)$$
$$x_1, x_2 \in \mathbb{N}$$




$\rightarrow Z^*_{LP} = (\frac{15}{7}, \frac{8}{7}) \simeq (2,1) = Z^*_{IP}$
*(on a de la chance on trouve la solution optimale via notre algorithme polynomial).*

### Solution pour l'arrondi SC

Posons $f$ comme le nombre maximum de sous-ensembles dans lesquels n'importe quel élément apparaît,

$$f = \max_{i=1,\ldots,n} (f_i) \text{ où } f_i = |\{j : e_i \in S_j\}|$$

Soit $x^* = (x^*_1, x^*_2, ..x^*_n)$ la solution optimale du **LP**, on va arrondir en incluant $S_j$ dans la solution si et seulement si $x^*_j \geq \frac{1}{f}$.

**Théorème 2.1** *Cette manière de procéder est un algorithme de $f$-approximation.*

**Corollaire 2.1** *L'adaptation de cet algorithme donne un algorithme de 2-approximation pour le VC.*

**Algorithme 2** Det_Rounding_SC

1: Résoudre le **LP** pour **SC**
2: Soit $x^*$ la solution optimale pour **LP**
3: A partir de $x^* = (x_1^*, x_2^*, \ldots, x_m^*)$,
on inclut $S_j$ dans la solution "arrondie" ssi $x_j^* \geq \frac{1}{f}$

où $f = \max_{i=1,\ldots,n}(f_i)$ où $f_i = |\{j : e_i \in S_j\}|$ et $I =$ ensemble des indices sélectionnés.

**Lemme 2.1** *La collection des $S_j$, $j \in I$ est une couverture.*

**Preuve** *On va montrer que tout $e_i$ est couvert.*
*Comme $x^*$ (la solution optimale du* **LP***) est une solution réalisable, on a :*

$$\sum_{j:e_i \in S_j} (x_j^*) \geq 1 \text{ pour un certain } e_i.$$

*(une contrainte pour un élément donné)*

*Par définition, $f_i \leq f$, on a donc au maximum $f$ termes dans la somme.*
*Donc, il y a au moins un terme $x_j^*$ qui doit être $\geq \frac{1}{f}$ car si ils sont tous $< \frac{1}{f}$, la somme n'est pas $\geq 1$.*
*$\Rightarrow$ Il existe un $j$ tel que $e_i \in S_j$ et $x_j^* \geq \frac{1}{f}$. En conséquence, $j \in I$ par l'algorithme et $e_i$ est couvert.*

□

**Théorème 2.2** *Det_Rounding_SC est un algorithme de $f$-approximation pour* **SC**.

**Preuve** *On voit que l'algorithme est polynomial.*
*Prouvons que la solution approché est égale à $f$ fois la solution optimale, càd :*

$$\sum_{j \in I} w_j = APP \leq f * OPT$$

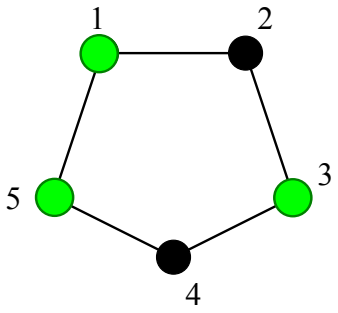
*Par construction, $1 \leq f * x_j^*$, $\forall j \in I$.* **(**)**
*Ensuite,*

$$\begin{aligned}
\sum_{j \in I}(w_j) &\leq \sum_{j=1}^{m}(w_j.1) && (1)\\
&\leq \sum_{j=1}^{m} w_j.f.x_j^* \text{ (par } \textbf{(**)}) && (2)\\
&= f\sum_{j=1}^{m} w_j.x_j^* && (3)\\
&= (f \text{ * valeur de la fonction objective pour la solution optimale du LP}) && (4)\\
&= f * Z_{LP}^* && (5)\\
&\leq f * OPT && (6)
\end{aligned}$$

□

Le facteur $f$ ainsi calculé est une **garantie à priori**, c'est-à-dire qu'avant même de lancer l'algorithme, on est sûr que ce facteur d'approximation sera vérifié. Cependant, l'algorithme **Det_Rounding_SC** permet d'avoir également une **garanti à fortiori**, c'est-à-dire une garantie que l'on a lorsque on a commencé à exécuter l'algorithme (dans ce cas-ci, lorsque l'on a la réponse au problème relaxé). En effet, prenons $\alpha = \frac{\sum_{j \in I}(w_j)}{Z_{LP}^*}$, de la preuve précédente on a que $\sum_{j \in I}(w_j) \leq f.Z_{LP}^*$ et donc $\alpha \leq f$ (dans certain cas, $\alpha$ peut être beaucoup plus petit que $f$). On a également $\frac{APP}{OPT} \leq \alpha$, ce qui fait d'$\alpha$ un meilleur facteur d'approximation que $f$ bien qu'il ne soit calculable qu'à partir de la solution obtenue via la résolution du problème relaxé ($Z_{LP}^*$).

**Exemple 2.2 (Voir feuilles des résultats obtenus avec CPLEX)**
**SC** : $\alpha = \frac{9}{9} = 1$
**VC** : $APP = 5$, $Z^*_{LP} = 2.5 \Rightarrow \alpha = 2$.




$OPT = 3$
$5 \leq 2 * 3 = 6$

**Corollaire 2.2** *L'algorithme* **Det_Rounding_SC** *peut être adapté au* **VC** *pour donner algorithme de 2-approximation (vu que $f = 2$ pour* **VC***).*

## 2.2 Primal $\leftrightarrow$ Dual

**Définition 2.2 (Problème dual)** *Pour tout* **LP**, *il existe un autre* **LP** *appelé son* **dual** *qui possède des propriétés intéressantes.*

**Exemple 2.3** *Voir feuilles des résultats obtenus avec* **CPLEX**.

L'intuition du **dual** est que l'on se met à la place de l'*entrepreneur* tandis que le **primal** correspond au point de vue du *client*.

Pour le **SC**, chaque élément $e_i$ est *"facturé"* d'un prix $y_i \geq 0$ à payer pour le couvrir. Pour que les prix soient raisonnables,

$$\sum_{i:e_i \in S_j} (y_i) \leq w_j \ (\forall j \in \{1, ..., m\})$$

c'est-à-dire, en français que l'on n'accepte pas que la somme des prix à payer pour couvrir les éléments d'un ensemble $S_j$ soit supérieur au poids de cet ensemble. Le problème **dual** est défini comme suit :

$\max \sum_{i=1}^{n} y_i$
s.l.c.
- $\sum_{i:e_i \in S_j} (y_i) \leq w_j \, (\forall j \in \{1, \ldots, m\})$
- $y_i \geq 0 \, (\forall i \in \{1, \ldots, n\})$

### 2.2.1 Remarques et propriétés du dual

**Remarques 2.1**
- *On appelle le* **LP** *"original" le* **problème primal**.
- *Le* **dual** *du* **dual** *est le* **primal**.
- *A chaque variable du* **dual** *correspond une contrainte du* **primal**.
- *A chaque variable du* **primal** *correspond une contrainte du* **dual**.

**Théorème 2.3 (Dualité faible)** *Soit $x$ une solution réalisable du primal (*MIN*) et $y$ une solution réalisable du dual ($\Rightarrow$* MAX*), alors la valeur de $y \leq$ la valeur de $x$ (et $\geq$ si le primal est une* MAX *et le dual une* MIN*).*

**Exemple 2.4 (Set Cover)**

$$\sum_{i=1}^{n} (y_i) \leq \sum_{j=1}^{m} (x_j . w_j)$$

**Théorème 2.4 (Dualité forte)** *S'il existe des solutions qui sont réalisables pour le primal* **ET** *pour le dual alors* **les valeurs optimales de ces 2 problèmes sont égales**.

**Exemple 2.5 (Set Cover)** *$x^*$ et $y^*$ sont des solutions optimales pour respectivement le* **primal** *et le* **dual** *alors :*

$$\sum_{i=1}^{n} y_i^* = \sum_{j=1}^{m} x_j^*.w_j$$

---

**Algorithme 3** Dual_Rounding_SC

---

1: Résoudre le **dual** pour **SC**
2: Soit $y^*$ la solution obtenue
3: Sélectionner $j$ dans $I'$ si l'inégalité correspondante dans le **dual** est serrée *(donc l'égalité). (c'est-à-dire* $\sum_{i:e_i \in S_j}(y_i^*) = w_j$*)*

---

**Exemple 2.6 (Set Cover)** *(Voir feuilles des résultats obtenus avec* **CPLEX***)*
$y^* = (0, 2, 0, 3, 4, 0)$
$I = \{2, 3, 4, 5\}$ *(les contraintes serrées sont $c_2$, $c_3$, $c_4$ et $c_5$)*
$APP = 9 + 5 = 14$.

**Exemple 2.7 (Vertex Cover)** *(Voir feuilles des résultats obtenus avec* **CPLEX***)*
$I' = \{1, 2, 3, 4, 5\}$

On peut montrer que cette $2^{\text{ème}}$ approche ne fait jamais mieux que la première. En effet, $I \subseteq I'$ c'est-à-dire que les indices sélectionnés par **Det_Rounding_SC** est un sous-ensemble des indices sélectionnés par **Dual_Rounding_SC**.

**Lemme 2.2** *La collection des $S_j$ tq $j \in I'$ est une couverture.*

**Preuve (par l'absurde)** *Supposons qu'il existe $e_k$ qui n'est pas couvert par les sous-ensembles $S_j$, $j \in I'$. Donc, par construction, et pour tout ensemble $S_j$ contenant $e_k$ :*

$$\sum_{i:e_i \in S_j} (y_i^*) < w_j$$

*(contrainte non serrée, sinon on aurait sélectionné $S_j$)*

*On définit $\epsilon$ comme la plus petite différence entre le* **membre droit** *(rhs) et le* **membre gauche** *(lhs) des contraintes/inégalités impliquant $e_k$. C'est-à-dire $\epsilon = \min_{j:e_k \in S_j}(w_j - \sum_{i:e_i \in S_j} y_i^*)$. $\Rightarrow$ On a que $\boxed{\epsilon > 0}$.*

*On définit à présent*

$$y'_k = y_k^* + \epsilon \tag{7}$$
$$y'_i = y_i^* \qquad \forall i \neq k \tag{8}$$

$\Rightarrow y'$ *est une solution réalisable car*

- *pour tout $j$ t.q. $e_k \in S_j$ :*

$$\sum_{i:e_i \in S_j} y'_i = \sum_{i:e_i \in S_j} (y_i^* + \epsilon) \leq w_j$$

*(par définition de $y'k$)*

- *pour tout $j$ t.q $e_k \notin S_j$ :*

$$\sum_{i:e_i \in S_j} y'_i = \sum_{i:e_i \in S_j} (y_i^*) \leq w_j$$

*Pourtant $\sum y'_i > \sum y_i^* \Rightarrow y'$ est meilleure que la solution optimale.*

⊠

**Théorème 2.5** *L'algorithme* **Dual_Rounding_SC** *est un algorithme d'approximation de facteur $f$.*

**Preuve** *(idée)*

* *Quand on choisit un ensemble $S_j$ dans la couverture, on "paye" en facturant $y_i^*$ à chacun de ses éléments $i$.*
* *Chaque élément est facturé au plus une fois pour chaque ensemble qui le contient (et donc au plus $f$ fois, par définition de $f$).*
* *Le coût total est au plus $f$ fois le cout de la solution optimale, c'est-à-dire (dualité forte) :*

$$COUT\_TOTAL = f.\sum_{i=1}^{n}(y_i^*) = f.Z_{LP}^* \leq f.OPT$$

## 2.3 La méthode primale-duale

On a vu 2 algorithmes avec facteur $f$ (sections 2.1 et 2.2) mais ils nécessitent de résoudre un **LP**.
⇒ On va utiliser la preuve du lemme de la section 2.2 *(dual rounding)* pour en tirer un algorithme.

**Algorithme 4** Primal_Dual_SC

```
1: ∀i, y_i ← 0
2: I ← {}
3: Tant que il existe un e_i non couvert faire
4:    Pour chaque S_k ∋ e_i faire
5:       Δ_k ← w_k − Σ_{j:e_j∈S_k}(y_j)
6:    fin Pour
7:    l ← arg min_{k:e_i∈S_k}(Δ_k)
8:    y_i ← y_i + Δ_l
9:    I ← I ∪ {l}
10: fin Tant que
```

**Remarque 2.1** *$arg\min_{i\in A}(x_i)$ donne le $k$ tel que $x_k$ est le minimum des $x_i$.*

**Exemple 2.8** *Voir section 1.4 de la feuille "Resultats obtenus avec CPLEX". Appliquons l'algorithme :*
$y \leftarrow 0$, $I \leftarrow \{\}$

- **Itération** 1 *($e_i = e_1$) :*
  *on considère toutes les contraintes $c_k$ correspondant aux $S_k \ni e_1$ et on calcule $\Delta_k$ :*
  — $c_1$ : $y_1 + y_3 + y_5 \leq 5 \Leftrightarrow 0 + 0 + 0 \leq 5 \Rightarrow \Delta_1 = 5$
  — $c_3$ : $y_1 + y_2 \leq 2 \Rightarrow \Delta_3 = 2$
  ↪ $l = 3$
  ↪ $y = (2, 0, 0, 0, 0, 0)$
  ↪ $I = \{3\}$

- **Itération** 2 *($e_i = e_3$ car $e_2$ est couvert par $S_3$) :*
  *on considère toutes les contraintes $c_k$ correspondant aux $S_k \ni e_3$ et on calcule $\Delta_k$ :*
  — $c_1$ : $2 + y_3 + y_5 \leq 5 \Rightarrow \Delta_1 = 3$
  — $c_4$ : $y_3 + y_4 \leq 3 \Rightarrow \Delta_4 = 3$
  ↪ $l = 1$ *(choix aléatoire ici)*
  ↪ $y = (2, 0, 3, 0, 0, 0)$
  ↪ $I = \{1, 3\}$

- **Itération** 3 *($e_i = e_4$) :*
  *on considère toutes les contraintes $c_k$ correspondant aux $S_k \ni e_4$ et on calcule $\Delta_k$ :*
  — $c_2$ : $y_2 + y_4 + y_5 \leq 5 \Rightarrow \Delta_2 = 5$
  — $c_4$ : $3 + y_4 \leq 3 \Rightarrow \Delta_4 = 0$
  ↪ $l = 4$
  ↪ $y = (2, 0, 3, 0, 0, 0)$
  ↪ $I = \{1, 3, 4\}$

- **Itération** 4 *($e_i = e_6$) :*
  *on considère toutes les contraintes $c_k$ correspondant aux $S_k \ni e_6$ et on calcule $\Delta_k$ :*
  — $c_2$ : ... $\Rightarrow \Delta_2 = 5$

— $c_5 \ : \ \ldots \ \Rightarrow \Delta_4 = 0$
$\hookrightarrow l = 5$
$\hookrightarrow y = (2, 0, 3, 0, 0, 4)$
$\hookrightarrow I = \{1, 3, 4, 5\}$
**fin.** *(tout est couvert)*

**Remarque 2.2** *Cet algorithme est intéressant car il est simple à implémenter et qu'il ne nécessite pas de résoudre le* **LP**.

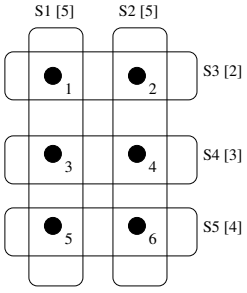
**Théorème 2.6** *L'algorithme Primal_Dual_SC est un algorithme de $f$-approximation.*

## 2.4 Algorithme d'approximation glouton

**Algorithme 5** Greedy_SC

1: $i \leftarrow \{\}$
2: $\forall j, \ \hat{S}_j \leftarrow S_j$ // *Cette variable représente les éléments non-couverts de* $S_j$
3: **Tant que** $I$ n'est pas une couverture **faire**
4: $\quad l \leftarrow arg\min_{j:\hat{S}_j \neq \{\}} \dfrac{w_j}{|\hat{S}_j|}$
5: $\quad I \leftarrow I \cup \{l\}$
6: $\quad \forall j, \ \hat{S}_j \leftarrow \hat{S}_j \setminus S_l$
7: **fin Tant que**

**Exemple 2.9** *Appliquons l'algorithme :*




$I \leftarrow \{\}$
$\hat{S}_j \leftarrow S_j$
**iteration 1** : $l = 3 \ \left(\frac{2}{2} < \frac{3}{2} < \frac{5}{3} = \frac{5}{3} < \frac{4}{2}\right)$
**iteration 2** : $l = 4 \ \left(\frac{3}{2} < \frac{4}{2} < \frac{5}{2} = \frac{5}{2}\right)$
**iteration 3** : $l = 5 \ \left(\frac{4}{2} < \frac{5}{1} < \frac{5}{1}\right)$
$I \leftarrow \{3, 4, 5\} \Longrightarrow$ **solution optimale !**

Cet algorithme est beaucoup plus simple et en fait le meilleur pour **SC** *(on peut pas faire meilleur algorithme d'approximation)* mais ce qu'on gagne en simplicité dans l'algorithme, on va le perdre dans le calcul du facteur d'approximation.

**Rappel 2.1** *Pour le* **VC**, *nous avons déjà appliqué ce type d'algorithme. Pour rappel, il donnait un facteur d'approximation $\sim \log k$ ce qui n'est pas un facteur borné. Cet algorithme ne donne donc pas toujours de très bons résultats mais c'est en général assez bon.*

**Interlude 2.1 (Nombre harmonique)** *On note $H_k$ le $k^{\text{ème}}$ nombre harmonique qui est calculé comme :*

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{k} = \sum_{i=1}^{k} \frac{1}{i} \sim \ln(k)$$

| $k$ | 1 | 2 | 3 | ... | 6 | ... | 100 | ... | 10000 |
|---|---|---|---|---|---|---|---|---|---|
| $H_k$ | 1 | 1.5 | 1.83 | ... | 2.45 | ... | 5.19 | ... | 9.79 |
| $\ln(k)$ | 0 | 0.69 | 1.09 | ... | 1.79 | ... | 4.6 | ... | 9.21 |

**Théorème 2.7** *L'algorithme* **Greedy_SC** *est un algorithme d'$H_n$-approximation où $n$ est le nombre d'éléments à couvrir.*

**Preuve**

- *L'algorithme est* **polynomial** *car $O(m)$ itérations (à chaque itération on sélectionne un sous-ensemble et il y en a maximum $m$) et pour chaque itération, un travail en $O(m)$ est effectué (calcul des ratios et du minimum) $\Rightarrow O(m^2)$.*
- *On définit :*
  - ▲ *$n_k$ comme le nombre d'éléments non-couverts au début de la $k^{\text{ème}}$ itération,*
  - ▲ *$l$ comme le nombre d'itérations utilisées pour* **Greedy_SC**,
    *$\Rightarrow$ ainsi,*
    - △ $n_1 = n$
    - △ $n_{l+1} = 0$
    - ▲ $\mathbf{n_k > n_{k+1}}$ **(*)**
  - ▲ *$I_k$ (pour une itération $k$) comme l'indice des ensembles sélectionnés jusque là (donc lors des itérations $1, \ldots, k-1$),*
  - ▲ *$\hat{S}_j$ comme l'ensemble des éléments non couverts dans $S_j$ au début de l'itération $k$,*

$$\hat{S}_j = S_j - \bigcup_{p \in I_k} S_p$$

- *On suppose que (on le prouvera par après), pour l'ensemble $S_j$ choisi à l'itération $k$ :*

$$w_j \leq \frac{(n_k - n_{k+1})}{n_k} OPT = \frac{|\hat{S}_j|}{n_k} OPT \textbf{ (1)}$$

- *Sous l'hypothèse que* **(1)** *est vraie, on a :*

$$APP = \sum_{j \in I} (w_j) \leq \sum_{k=1}^{l} \frac{(n_k - n_{k+1})}{n_k} OPT \tag{9}$$

$$= OPT \sum_{k=1}^{l} \frac{(n_k - n_{k+1})}{n_k} \tag{10}$$

$$= OPT \sum_{k=1}^{l} \left( \frac{1}{n_k} + \frac{1}{n_k} + \ldots + \frac{1}{n_k} \right) \text{ (car } n_k > n_{k+1}\text{)} \tag{11}$$

$$\leq OPT \sum_{k=1}^{l} \left( \frac{1}{n_k} + \frac{1}{(n_k - 1)} + \ldots + \frac{1}{(n_{k+1} + 1)} \right) \tag{12}$$

$$= OPT \sum_{i=1}^{l} \frac{1}{i} \tag{13}$$

$$= OPT.H_l \tag{14}$$

$$\Rightarrow \frac{APP}{OPT} \leq H_n$$

**Exemple 2.10** *($n_k = 6$ et $n_{k+1} = 2$)*
$\frac{n_k - n_{k+1}}{n_k} = \frac{6-2}{6} \leq \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} \leq \frac{1}{6} + \frac{1}{5} + \frac{1}{4} + \frac{1}{3}$

Il ne reste donc qu'à prouver l'inégalité **(1)**.

**Preuve** ($w_j \leq \frac{(n_k - n_{k+1})}{n_k} OPT$)
*A l'itération $k$, on a*

$$\min_{j:\hat{S}_j \neq \{\}} \left( \frac{w_j}{|\hat{S}_j|} \right) \leq \frac{OPT}{n_k} \qquad \textbf{(2)}$$

*On observe :*

- *il existe une solution optimale qui couvre $n$ éléments pour un coût* **OPT**,
- *au début, il existe un sous-ensemble qui couvre ses éléments avec un poids moyen $\frac{w_j}{|S_j|} \leq \frac{OPT}{n}$*
- *de manière similaire, quand $p$ éléments ont été couverts, à l'itération $k$, la même solution* **OPT** *peut couvrir les $n - p$ éléments non-couverts avec poids $\leq$* **OPT**.

$\Rightarrow$ *il existe un sous ensemble qui couvre ses propres éléments non-couverts avec un poids $\leq \frac{OPT}{n-p}$ et $n - p = n_k$*
$\Rightarrow$ **(2)** *est prouvée.*

*Donc, on choisit $S_j$ à l'itération $k$ tel que : $\frac{w_j}{|\hat{S}_j|} \leq \frac{OPT}{n_k}$ (par* **(2)**).
$\Longrightarrow$ *Il y aura donc $|\hat{S}_j|$ éléments nouvellement couverts ce qui signifie $n_{k+1} = n_k - |\hat{S}_j|$.*
*Dès lors $\frac{w_j}{|\hat{S}_j|} \leq \frac{OPT}{n_k}$ devient*

$$w_j \leq \frac{(OPT.|\hat{S}_j|)}{n_k} = \frac{(n_k - n_{k+1})}{n_k}.OPT$$

□

**Théorème 2.8** *L'algorithme* **Greedy_SC** *est un algorithme de $H_g$-approximation où $g = \max_j(|S_j|)$ ($g$ est en fait le maximum d'éléments dans un sous-ensemble).*

## 2.5 Conclusion du chapitre

On a testé plusieurs types d'algorithmes d'approximation et on peut en conclure qu'il n'y en a pas un qui est meilleur que tous les autres, cela dépendant des instances *(bien que pour le* **SC** *on a très peu d'espoir de faire mieux que l'algorithme glouton)*. On terminera cette section par l'énoncé d'un résultat bien connu sur lequel on va se baser pour conjecturer que $\mathcal{P} \neq \mathcal{NP}$.

**Théorème 2.9 (Hardness result)** *Sous l'hypothèse que la conjecture "jeux uniques" (conjecture disant que le problème des "jeux uniques" est $\mathcal{NP}$-difficile, ce qui n'est pas encore prouvé) est vraie et s'il existe un algorithme d'$\alpha$-approximation (général, évidemment) pour* **VC** *avec $\alpha < 2$, alors $\mathcal{P} = \mathcal{NP}$.*

↪ Voir exercices dans l'annexe B.

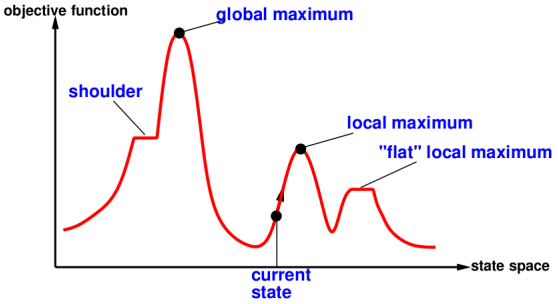
# 3 Algorithmes gloutons et de recherche locale

Pour rappel, on a utilisé ce type d'algorithme au chapitre 2 pour résoudre le **SC** avec un facteur d'approximation de $H_n$ (et même $H_g$). Dans ce chapitre nous allons changer d'approche, au lieu de définir un problème et d'utiliser plusieurs approches pour l'approximer, nous allons voir plusieurs problèmes que l'on va uniquement considérer via l'approche gloutone ou de la recherche locale. Mais qu'est-ce exactement un algorithme glouton ? et un algorithme de recherche locale ?

**Définition 3.1 (Algorithme glouton)** *Algorithme où la solution est construite* **pas à pas**. *A chaque pas la* **partie** *suivante est construite en prenant une décision qui est localement la meilleure possible.*

**Exemple 3.1 (Chapitre 2) Greedy_SC** *est un algorithme où le choix local consiste à prendre le sous-ensemble avec le meilleur ratio* $\frac{poids}{\#\text{éléments que l'on couverait en prenant le sous-ensemble}}$.

**Définition 3.2 (Algorithme de recherche locale)**




*Algorithme commençant avec une* **solution initiale réalisable** *qui est améliorée* **pas à pas** *en considérant le* **voisinage** *de cette solution. Dès que ce n'est plus possible, l'algorithme s'arrête.*

Comme on peut le voir sur le dessin, si on est bloqué dans un optimum local, on obtient une solution qui n'est pas l'optimum global.

**Exemple 3.2 (Chapitre 2)** *Pour le* **SC** *on peut prendre comme voisinage "Sélectionner un ensemble et en désélectionner un autre".*

## Comparaison

1. Les points communs :
   - ils prennent tous deux des décisions qui optimisent un choix local,
   - la solution obtenue est une solution approchée,
   - ils sont très facilement implémentables → ils sont souvent très populaires.
2. Les différences :
   - **Glouton** : solution partielle qui sera réalisable à la fin,
     **Recherche locale** : solution réalisable tout au long de l'exécution ;
   - **Glouton** : souvent évident de prouver que l'algorithme est polynomial,
     **Recherche locale** : rarement polynomial avec une implémentation simple mais dans certains cas elle l'est. Ces cas nécessitent en général certaines restrictions sur les changements possibles pour assurer un temps polynomial.

## 3.1 Ordonnancement de tâches sur une seule machine

*(De l'anglais "Scheduling jobs on single machine (**SSM**)")*

**Scénario :**

- Vous disposez d'une machine qui ne peut exécuter qu'une seule tâche à la fois.
- $A$,$B$ et $C$ sont 3 tâches à exécuter.
- L'ordonnancement commencera au temps $t = 0$.
- Chaque tâche prend un certain temps pour être réalisée : $A = 2$, $B = 1$, $C = 4$ *(unités de temps)*.
- Les tâches ne sont pas interruptibles.
- Chaque tâche n'est pas accessible dès le début, elles ont un temps de release(release date) : $A = 0$, $B = 2$, $C = 1$.
- Chaque tâche a une deadline : $A = -1$, $B = 1$, $C = 10$

↪ On cherche à minimiser le retard maximal ($L_{MAX}$) *(maximum lateness)*.

Quel ordonnancement est optimal ? ($ABC$, $ACB$, $BAC$, $BCA$, $CAB$ ou $CBA$ ?)

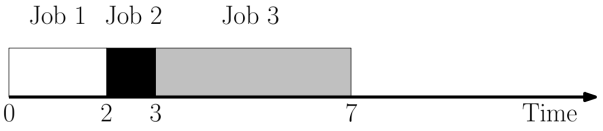



FIGURE 7 – Ordonnancement optimal *(Job 1 = A, Job 2 = B, Job 3 = C)*

En effet, on a :

- $retard(A) = 2 - (-1) = 3$
- $retard(B) = 3 - 1 = 2$
- $retard(C) = 7 - 10 = -3$, dans ce cas-ci on est en avance.

↪ $L_{MAX} = 3$ et on ne peut faire mieux, et ce, pour 2 raisons :

- C'est la tâche $A$ qui cause le retard maximum de 3,
- pour $A$, on ne peut pas faire mieux car on l'a faite le plus tôt possible.

Ceci était un exemple, nous généralisons à présent le problème. En général, on a :

- $n$ tâches,
- chaque tache $j$ possède :
  - ★ un **temps de complétion** : $p_j$,
  - ★ une **release date** : $r_j$,
  - ★ une **deadline** : $d_j$,

Aussi, pour chaque solution réalisable, nous introduisons certaines notations :

- $c_j$ = instant ou $j$ est finie,
- $L_j$ *(lateness)* $= c_j - d_j$,
- $L_{MAX} = \max_j(L_j)$.

**Exemple 3.3 (Pour la même instance)** *On a :*

- $p_1 = 2,\ r_1 = 0,\ d_1 = -1$
- $p_2 = 1,\ r_1 = 2,\ d_1 = 1$
- $p_3 = 4,\ r_1 = 1,\ d_1 = 10$

*Et pour l'ordonnancement* **ABC** *:* $L_{MAX} = L_1 = c_1 - d_1 = 2 - (-1) = 3$

---

**Problème 3.1 MIN Schedule Single-Machine(SSM)**

* **Instance** *: $n$ jobs (triplets ($p_j$,$r_j$,$d_j$)),*

* **Solution** *: schedule des $n$ jobs ($\simeq$ $c_j$ et $L_j$),*

* **Mesure** *: $L_{MAX}$*

---

Pour simplifier, pour tout $j$, on émet les hypothèses suivantes :

**Hyp$_1$** : $r_j$ et $p_j \geq 0$

**Hyp$_2$** : $d_j < 0$ *(pour être sur que $L_{MAX}$ soit $> 0$, et pour simplifier les choses vu qu'une seule deadline dans le passé suffirait)*

**Remarque 3.1** *Imaginons un algorithme de $f$-approximation et que la solution optimale est $= 0$. Par la définition, on a $APP.f = OPT$, c'est-à-dire $0.f = 0$ ce qui est gênant par rapport à la définition... alors $\mathcal{P} = \mathcal{NP}$ ?*
*$\Rightarrow$ On suppose que l'optimal est toujours $> 0$.*

**Exemple 3.4** *$p_j$ et $r_j$ identiques que précédemment, on pose :*

$$d'_j = d_j - (\max_j(d_j + 1))$$

$\rightarrow (\max_j(d_j + 1)) = 11$
$\rightarrow d'_1 = -12, d'_2 = -10, d'_3 = -1 \Rightarrow L'_{MAX} = L_{MAX} + 11 = 3 + 11 = 14$
$\hookrightarrow$ *on ne fait qu'une translation dans le temps, on peut retomber sur l'instance initiale simplement en soustrayant l'écart utilisé.*

Nous allons maintenant élaborer un premier algorithme permettant d'approcher la solution optimale. Il s'agit d'un algorithme privilégiant la tâche de deadline la plus proche ou en anglais *"Earliest Due Date (**EDD**)"* $\rightarrow$ on commence par celle la plus en retard dans le cas où il y a des deadlines négatives.

**Algorithme 6** EDD_SSM

```
1: t ← 0
2: todo ← {1, 2, ..., n}
3: Tant que todo n'est pas vide faire
4:    Si au moins une tâche est disponible (r_j ≤ t ?) alors
5:       j ← arg min_j d_j
6:       t ← t + p_j // le temps d'exécution est ajouté au temps courant
7:       c_j ← t // le temps à laquelle la tâche est terminée est mis à jour
8:       todo ← todo \ {j}
9:    Sinon
10:      t ← min_{j∈todo}(r_j) // On avance à la tâche la plus rapidement disponible
11:   fin Si
12: fin Tant que
13: Calculer et retourner L_MAX
```

Soit $S$ un sous-ensemble de tâches, on définit

- $r(S) = \min_{j \in S} r_j$ *(la date de dispo au plus tôt pour S)*
- $p(S) = \sum_{j \in S} p_j$ *(temps pour tout accomplir sans "trou")*
- $d(S) = \max_{j \in S} d_j$ *(la deadline la + éloignée)*
- $L^*_{MAX} = OPT$.

**Lemme 3.1** *Pour tout sous-ensemble de tâches $S$,*

$$L^*_{MAX} \geq r(S) + p(S) - d(S)$$

**Preuve** *Soit un ordonnancement optimal pour toutes les tâches ($n$ tâches), voyons le simplement comme un ordonnancement pour $S$.*

**Exemple 3.5** *(exemple précédent)*

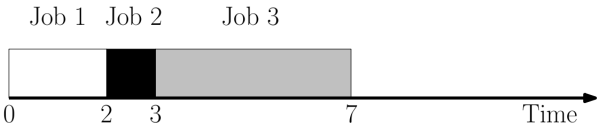



FIGURE 8 – Ordonnancement optimal *($Job\ 1 = A$, $Job\ 2 = B$, $Job\ 3 = C$)*

*Si on prend $S = \{A, C\}$ :*

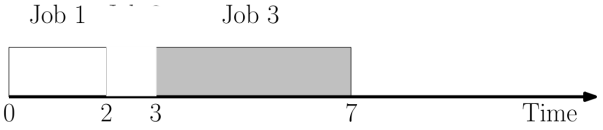



FIGURE 9 – Ordonnancement optimal concentré sur $S$ *($Job\ 1 = A$, $Job\ 3 = C$)*

*Soit $j$ la dernière tâche traitée dans $S$ :*
**(1)** *aucune tâche ne peut être exécutée avant $r(S)$,*
**(2)** *au total on a besoin de minimum $p(s)$ unité de temps.*
$\rightarrow$ *de* **(1)** *et* **(2)** *on tire que $j$ ne peut pas être finie avant $r(S) + p(S)$,*
$\rightarrow$ *on sait également, par définition de $d(S)$, que $d(j) < d(S)$.*
$\rightarrow$ *le retard pour la tâche $j$ est au moins $r(S) + p(S) - d(S)$*
$\hookrightarrow$ $L^*_{MAX} \geq r(S) + p(S) - d(S)$ *(vu que $L^*_{MAX}$ est le retard maximal)*

□

**Théorème 3.1 (EDD_SSM est un algorithme de 2-approximation pour le problème SSM)** *(si les hypothèses $\mathbf{Hyp_1}$ et $\mathbf{Hyp_2}$ sont vérifiées)*

**Preuve** *Soit l'ordonnancement donné par* **EDD_SSM** *et soit $j$ la tâche ayant le plus grand retard, on a donc :*

$$L_{MAX} = c_j - d_j \qquad \textbf{(1)}$$

*Faisons un focus sur $c_j$, soit $t \leq c_j$ l'instant le plus tôt tel que la machine est utilisée sans interruption pour toute la période $[t, c_j[$, c'est-à-dire la situation suivante :*

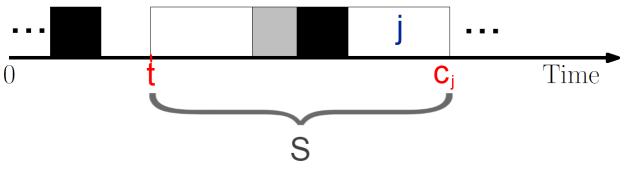



FIGURE 10 – Exemple général

*Sur l'intervalle* $[t, c_j[$ *et* $S$ *on sait :*

★ $r(S) = t$, *en effet, juste avant* $t$ *il y a un repos (par définition), donc aucune tâche de* $S$ *n'était disponible avant* $t$ *(et donc tous les* $r_j$ *de* $S$ *sont* $\geq t$ *(au moins une est égale à* $t$*, par définition de* $t$ *toujours).*

★ $p(S) = c_j - t$ *(vu qu'il n'y a pas de pause par construction)* $= c_j - r(S)$

$$\Longrightarrow c_j = r(S) + p(S) \qquad \textbf{(2)}$$

*Comme* $d(S) < 0$ *(par les hypothèses faites précédemment), par le lemme, on a*

$$L^*_{MAX} \geq r(S) + p(S) - d(S) \geq r(S) + p(S) = c_j \text{ par (2)} \qquad \textbf{(3)}$$

*Si on applique à nouveau le lemme avec* $S = \{j\}$*, on a :*

$$L^*_{MAX} \geq r_j + p_j - d_j \geq -d_j \qquad \textbf{(4)}$$

*En combinant (3) et (4) (en sommant les 2 cotés) on a :*

$$L_{MAX} =_{(1)} c_j - d_j \leq 2L^*_{MAX} \qquad \text{(par (3) et (4))}$$

*et donc*

$$\frac{L_{MAX}}{L^*_{MAX}} \leq 2$$

□

## 3.2 Le problème "$k$-*centre*"

La motivation de ce problème est de faire du clustering, c'est-à-dire diviser un ensemble de données en plusieurs sous-groupes contenant des données "proches" au sens d'une certaine distance à définir. Dans notre cas nous utiliserons la distance euclidienne.

---

**Problème 3.2 MIN $k$-centre**

* **Instance** *:*
— *graphe non-dirigé complet* $G = (V, E)$ *pondéré avec une distance* $d_{ij} \geq 0$ *entre toute paire* $i$*,* $j$ *de sommets (on a donc une matrice des distances),*
— *un entier* $k$ *;*

* **Solution** *:* $S \subseteq V$ *tel que* $|S| = k$*, sous-ensemble d'éléments appelés les centres.*

* **Mesure** *: distance maximum entre un sommet et son centre (le plus proche), cette distance est appelée le rayon.*

---

On fait l'hypothèse que la distance utilisée est bien une distance au sens mathématique, c'est-à-dire qu'elle réunit les 3 propriétés :

- $\forall i \in V,\ d_{ii} = 0$
- $\forall i, j \in V,\ d_{ij} = d_{ji}$
- $\forall i, j, k \in V,\ d_{ij} \leq d_{ik} + d_{kj}$ *(inégalité triangulaire)*

**Exemple 3.6**

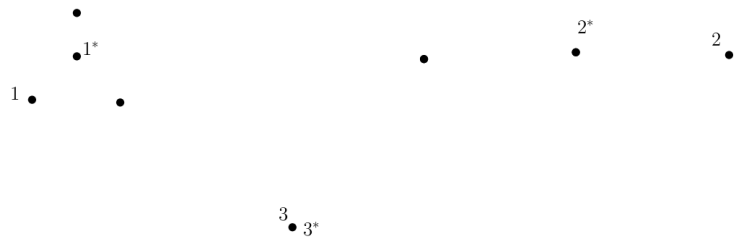



FIGURE 11 – Exemple de solution optimale d'une instance avec $k = 3$

*(les nombres avec une étoile sont ceux représentant les centres des groupes)*
*Dans cet exemple, la distance maximale (le rayon), et donc la mesure de cette solution, est la distance entre le point* $2$ *et le point* $2^*$*.*

Nous allons à présent essayer d'élaborer un algorithme glouton permettant de définir au moins les bons groupes, même s'il ne trouve pas forcément le bon point comme centre de chaque groupe. Nous proposons de commencer l'algorithme par sélectionner un point au hasard comme centre puis on va itérer en prenant à chaque fois le point le plus loin des centres déjà sélectionné. Pour ce faire, nous introduisons une notation : soit $S$ l'ensembles des centres, on note $d(i, S) = \min_{j \in S}(d_{ij})$, on a donc :

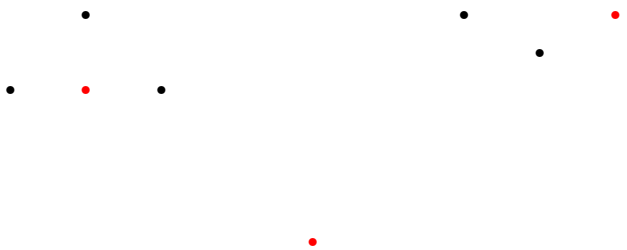
$$rayon = \max_{i \in V} d(i, S)$$

**Algorithme 7** Greedy_k_center

```
Choisir i ∈ V au hasard.
S ← {i}
Tant que |S| < k faire
    j ← arg max_{j∈V} d(j, S)
    S ← S ∪ {j}
fin Tant que
```

Quel est le pire des cas pour cet algorithme ? Voici un exemple serré ("tight") du pire des cas (ce qui signifie qu'il montre que le facteur d'approximation est atteint).

**Exemple 3.7**



*Sur le dessin (les centres sont en rouges) on a :*

$$d_{ab} = 2d_{ac} = 2d_{bc}$$

*Dans ce cas, on fait* $2$ *fois moins bien. L'algorithme peut donc faire* **au moins** $2$ *fois pire (et en fait* $2$ *est le facteur d'approximation).*

**Théorème 3.2 Greedy_$k$_center** *est un algorithme de 2-approximation pour le problème* $k$**-center**.

**Preuve**

- **L'algorithme s'effectue-t-il en temps polynomial ?** $\hookrightarrow$ *oui, de manière triviale.*
- **L'algorithme a-t-il un facteur d'approximation égal à 2 ?**
  *Soit* $S^* = \{j_1, \ldots, j_k\}$ *une solution optimale et* $r^*$ *le rayon de* $S^*$.
  $\Rightarrow$ *cette solution partitionne* $V$ *en groupes* $V_1, ..., V_k$ *où chaque sommet* $j \in V$ *est placé dans un groupe* $V_i$ *s'il est le plus proche du centre* $j_i$ *parmi tous les centres* $S^*$ *(égalités brisées arbitrairement).*

  **Fait (1)** *: chaque paire de points* $j, j'$ *dans un même groupe* $V_i$ *est telle que* $d_{jj'} \leq 2r^*$.
  *Ce fait est logique vu que* $r^*$ *est le rayon du plus gros cluster, les 2 points les plus éloignés possibles sont donc de part et d'autres du centre de ce cluster à une distance exacte de* $r^*$.

  1. *Supposons d'abord que chaque centre de* $S$ *est sélectionné par l'algorithme parmi chacun des groupes* $V_1, \ldots V_k$ *déterminés par* $S^*$. *Par le* **fait (1)**, *tout sommet de* $V$ *est à distance* $\leq 2r^*$ *d'un des centres de* $S \Rightarrow$ *le théorème est prouvé.*
  2. *Supposons maintenant que ce ne soit pas le cas, c'est-à-dire que l'algorithme sélectionne* $2$ *points dans le même groupe* $V_i$. *Cela veut dire que lors d'une certaine itération* $j \in V_i$ *est choisi alors que* $j' \in V_i$ *avait déjà été choisi. Par le* **fait (1)**, *on sait que*

     $$d_{jj'} \leq 2r^*$$

     *Or l'algorithme choisit* $j$ *car il est actuellement* **le point le plus éloigné** *des centres contenus dans* $S$ *à cette itération. Donc, tous les points sont à distances* $\leq 2r^*$ *d'un des centres déjà dans* $S$. *Cette observation reste vraie à la fin de l'algorithme (le fait d'ajouter des centres dans* $S$ *n'augmente pas la distance maximale).*

□

Nous allons maintenant prouver qu'il n'est pas possible de faire mieux comme garantie théorique. En effet, il existe des heuristiques effectuant des meilleurs choix que notre algorithme glouton mais elles n'ont pas de meilleure garantie théorique que ce facteur $2$. Ceci signifie que, même si elles donnent de meilleurs résultats en moyenne, il existe des instances pour lesquelles elles fourniront une solution équivalente à $2$ fois la solution optimale.
Afin d'effectuer cette preuve, nous allons nous ramener à un problème de décision connu pour être $\mathcal{NP}$-difficile, le problème du **Dominating Set**.

---

**Problème 3.3 Dominating Set (decision) (DS)**

* **Instance** *:*
— *Graphe* $G = (V, E)$,
— *entier positif* $k$ *;*
* **Question** *: Existe-t-il un ensemble* $S \subseteq V$ *de taille* $k$ *tel que chaque sommet de* $V$ *est soit dans* $S$ *soit adjacent à un sommet de* $S$ *?*

---

**Exemple 3.8**

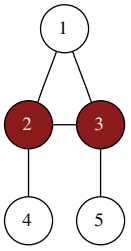



FIGURE 12 – Exemple d'instance du **DS**

*Si* $k = 1 \Rightarrow$ **NO**
*Si* $k = 2 \Rightarrow$ **YES** *(en prenant l'ensemble* $S = \{2, 3\}$*)*

Comment peut-on transformer une instance de **DS** en une instance de $k$-center ? Pour le graphe, rien de bien particulier, la notion importante à définir ici est la distance à utiliser, prenons :

- $d_{ij} = 1$ si $(i, j) \in E$
- $d_{ij} = 2$ sinon

En imaginant qu'il existe un algorithme d'approximation de facteur $f < 2$ pour résoudre $k$-**center**, alors si la réponse est $2$ pour cet algorithme cela veut dire que la réponse est **NO** pour le **DS**. On aurait alors là un algorithme polynomial capable de résoudre un problème $\mathcal{NP}$-difficile de manière exacte... ce qui signifierait que $\mathcal{P} = \mathcal{NP}$ ! On peut utiliser cet argument pour prouver qu'on ne peut faire mieux qu'un facteur $2$ pour résoudre $k$-**center**.

**Théorème 3.3** *Il n'y a pas d'algorithme d'*$\alpha$*-approximation pour le* $k$-**center** *avec* $\alpha < 2$.
*(à moins que* $\mathcal{P} = \mathcal{NP}$*)*

**Preuve**
*Soit* $A$ *un algorithme d'approximation de facteur* $< 2$*, montrons qu'on peut utiliser* $A$ *pour résoudre* **DS** *en temps polynomial. A partir d'une instance* $(G, k)$ *du* **DS***, on définit une instance pour le* $k$-**center** *comme suit :*

- ★ $d_i j = 1$ *si* $(i, j) \in E$
- ★ $d_i j = 2$ *sinon.*

$\Rightarrow$ *il y a un ensemble dominant de taille* $k$ *ssi le rayon optimal est* $1$ *pour le* $k$-**center**.
$\Rightarrow$ *comme l'algorithme* $A$ *possède un facteur* $< 2$*, la mesure du problème du* $k$*-center doit obligatoirement trouver une solution ayant un rayon égal à 1 si une telle solution existe (car la solution est un entier).*

□

## 3.3 Ordonnancement de tâches sur des machines identiques parallèles

*(un des deux premiers algorithmes d'approximation : années 60)*

Il s'agit d'une variation du **SSM** :

- il y a maintenant plusieurs machines,
- pas de *release date* ni de *deadline*,
- but : minimiser le temps utilisé pour terminer toutes les tâches.

---

**Problème 3.4 MIN Schedule Parralel Machine (SPM)**

* **Instance** :
— *entiers* $m$ *(machines) et* $n$ *(tâches),*
— *pour chaque tâche* $j$*, un temps de complétion* $p_j$ *(unités de temps sans interruption).*
* **Solution** : *Scheduling des* $n$ *tâches sur les* $m$ *machines.*
* **Mesure** : $c_{MAX} = \max_j c_j$

---

### 3.3.1 Approche par la recherche locale

**Exemple 3.9 (**$m = 5$ **et** $n = 10$**)**

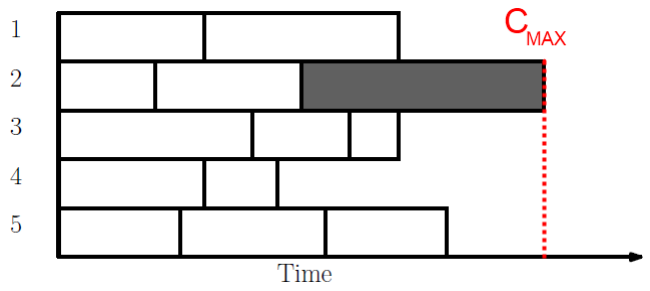



FIGURE 13 – Exemple de solution d'une instance du **SPM**

*Via un algorithme de recherche locale que l'on appellera* **LocalSearch_SPM**, *on pourrait choisir comme voisinage par exemple le fait de prendre la tâche qui se finit le plus tard pour la mettre sur la machine qui n'est plus utilisé le plus tôt. On va même simplifier en déplaçant la tâche vers une machine permettant de diminuer* $C_{MAX}$*. Imaginons que cette tâche est la tâche* $l$*, on va la déplacer vers une machine qui se termine avant* $c_l - p_l$*. Appliquons l'algorithme (une seule itération) :*

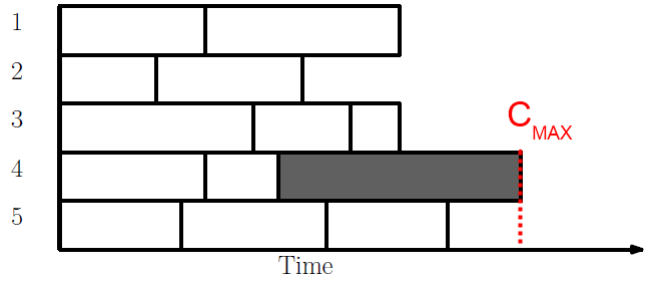



FIGURE 14 – Exemple de solution d'une instance du **SPM**

$\Longrightarrow$ **Optimum local.**

Notons $c^*_{MAX}$ la longueur d'un schedule optimal, on peut alors dire que ce schedule prendra au moins le temps d'exécution de la plus longue des tâches soumises :

$$\boxed{c^*_{MAX} \geq \max_{j=1,\ldots,n} p_j} \qquad \textbf{(1)}$$

D'autre part, il y a au total un temps de travail $P = \sum_{j=1}^{n} p_j$ et $m$ machines ; donc en moyenne une machine travaille $\frac{P}{m}$ unité de temps. Il y a donc au moins une machine qui travaille pendant un temps $t \geq \frac{P}{m}$ (sinon la moyenne n'est pas la moyenne . . . !) et donc :

$$\boxed{c^*_{MAX} \geq \frac{P}{m} = \frac{\sum_{j=1}^{n} p_j}{m}} \qquad \textbf{(2)}$$

Prouvons à présent que l'algorithme que nous avons montré sur l'exemple plus haut est bien un algorithme d'approximation.

**Théorème 3.4** *L'algorithme LocalSearch_SPM est un algorithme de 2-approximation pour le problème* **SPM**.

**Preuve**

1. **L'algorithme s'exécute-t-il en temps polynomial ?**
   *(Voir preuve plus formelle dans livre de référence, chapitre 2)*
   ↪ *Intuitivement : l'algorithme est polynomial car le nombre d'itérations est borné par une fonction polynomiale.*

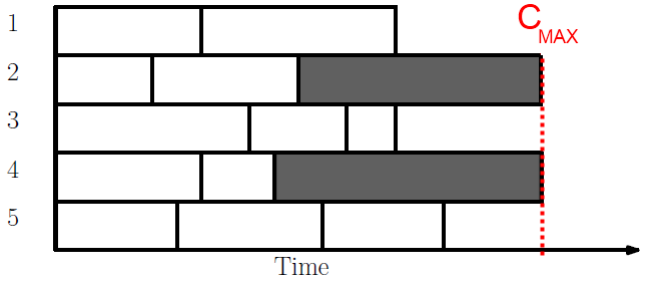



FIGURE 15 – Exemple où 2 machines atteignent $c_{MAX}$

   *A chaque itération on va soit réduire $c_{MAX}$, soit diminuer le nombre de machines atteignant $c_{MAX}$*
   ⇒ *nombre limité d'étapes.*

2. **L'algorithme possède-t-il un facteur d'approximation égal à 2 ?**
   *Soit une solution produite par* **LocalSearch_SPM**, *soit $l$ la tâche se terminant en dernière, càd que $c_l = c_{MAX}$. On est donc dans un cas comme suit (l'algorithme est terminé) :*

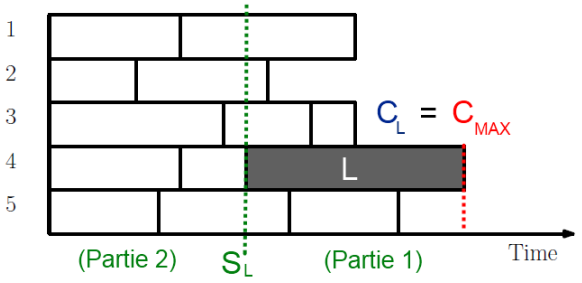



FIGURE 16 – Exemple de solution donnée par l'algorithme

*Par le fait que l'algorithme est terminé,* **toutes les machines sont au travail entre le temps** $0$ **et le début de la tâche** $l$*, c'est-à-dire jusqu'en* $S_l = c_l - p_l$*.*

- *Le temps pris dans la (partie 1) est égal à* $p_l$ *et par* **(1)** *ce temps est* $\leq c^*_{MAX}$*.*
- *Le temps pris dans la (partie 2) est égale à* $m.S_l$ *car toutes les machines sont au travail. Et on sait que* $m.S_l \leq P$ *(plus petit que le travail total) et donc*

$$S_l \leq \frac{P}{m} \leq c^*_{MAX} \qquad \textit{(par } \textbf{(2)}\textit{)}$$

*On conclut en observant que la fin du schedule se situe en* $c_{MAX} \leq 2.c^*_{MAX}$ *(en sommant les 2 parties du schedule qui sont majorées par* $c^*_{MAX}$*).*

□

### 3.3.2 Approche gloutone

L'idée dans cette approche gloutone est de considérer les tâches dans un certain ordre et de les ajouter au schedule les unes à la suite des autres. On peut alors imaginer trier les tâches dans un ordre aléatoire soit par ordre décroissant des $p_i$ et donc de considérer d'abord les plus grandes et garder les plus petites pour la fin.

**Exemple 3.10** $m = 2$ *et* $n = 3$ $p_1 = 2$, $p_2 = 3$ *et* $p_3 = 4$

*Si on les place selon l'ordre dans lequel les tâches sont données ("ListScheduling"), on obtient :*

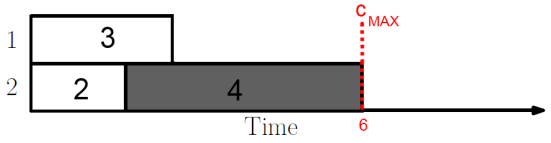



FIGURE 17 – Résultat du ListScheduling

*On remarque rapidement que cela dépend de l'ordre dans lequel on donne les tâches.*
*Par exemple, l'ordre* $3, 1, 2$ *donne la solution optimale :*

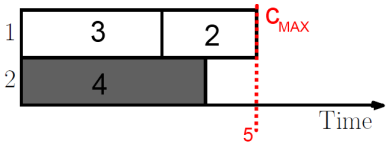



FIGURE 18 – Résultat du ListScheduling avec l'ordre 3, 1, 2

**Algorithme 8** ListScheduling

```
todo ← liste des tâches (liste = séquence ordonnée)
Pour chaque tâche t dans todo faire
    Attribuer t à une machine dont le temps de complétion est minimum
fin Pour
```

De cet algorithme on pourrait tirer un algorithme naïf exact pour ce problème, il suffirait d'énumérer chaque permutation (ordre) possible pour la liste et appliquer **ListScheduling** pour chacune de ces permutations et prendre celle qui donne la valeur la plus petite. Le problème est que cet algorithme a une complexité factorielle (si liste de $n$ éléments, $n!$ permutations).

**Théorème 3.5** *L'algorithme* **ListScheduling** *est un algorithme de* 2*-approximation pour* **SPM**.

**Preuve**
*Soit une solution donnée par l'algorithme* **ListScheduling**, *notons* $l$ *la tâche qui termine ce scheduling, càd*

$$c_l = c_{MAX}$$

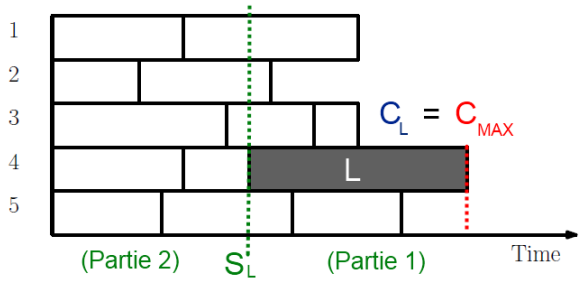



FIGURE 19 – Exemple de solution donné par l'algorithme

*(Exactement le même cas que pour LocalSearch vu que l'on place* $l$ *dans la machine dont le temps de complétion est minimum, donc on aura jamais une tâche* $l$ *que l'on pourrait placer plus tôt dans le scheduling courant)*

⇒ *Chaque machine est en activité jusque* $S_l$, *sinon on aurait pu attribuer* $l$ *à une autre machine.*

⇒ *Et donc, cela signifie que si on utilise* **ListScheduling** *pour la solution initiale du* **LocalSearch_SPM**, *ce dernier n'appliquera aucune itération car il se trouvera dans un optimum local.*

*Le facteur d'approximation* 2 *est donc valable ici aussi (car on a finalement utilisé que l'algorithme glouton* **ListScheduling** *dans cette "recherche locale").*

□

A présent, nous allons considérer que :

- la liste est triée par ordre décroissant de $p_j \Longrightarrow p_1 \geq p_2 \geq \ldots \geq p_n$
- $n > m$ (car sinon $n \leq m$ et tout ordre est optimal vu que **ListScheduling** va placer exactement une tâche par machine)

Cette approche s'appelle **LPT** *(Longest Processing Time rule)*, l'idée de manière formelle est :

- trier les tâches par ordre décroissant de $p_j \Longrightarrow$ **polynomial**,
- appliquer **ListScheduling** sur ces tâches $\Longrightarrow$ **polynomial**.

**Exemple 3.11** *Appliquons l'algorithme* **LPT** *sur l'exemple précédent.*
$\Rightarrow p_1 = 4$, $p_2 = 3$, $p_3 = 2$ *(on a retrié)*

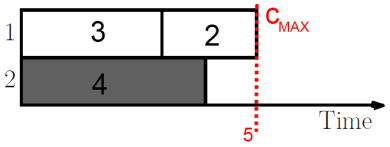



FIGURE 20 – Solution donnée par l'algorithme

**Lemme 3.2** *Pour toute instance d'un* **SPM** *telle que* $p_1 \geq p_2 \geq \ldots \geq p_n > \frac{c^*_{MAX}}{3}$, *l'algorithme* **LPT** *calcule un schedule optimal.*

**Théorème 3.6** *L'algorithme* **LPT** *est un algorithme de* $\frac{4}{3}$*-approximation pour* **SPM**.

**Preuve (par contradiction)**
*Supposons que le théorème est faux, c'est-à-dire qu'il existe une instance* $p_1 \geq p_2 \geq \ldots \geq p_n$ *qui est un contre exemple.*
*Soit un shedule obtenu par l'application de* **LPT** *sur cette instance, on peut supposer que la dernière tâche de la liste,* $p_n$*, est également la tâche* $l$ *qui termine le schedule.*

↪ *Si on ne pouvait supposer cela, alors il existe un autre contre-exemple plus petit (= moins de tâches) qui respecte cette hypothèse. En effet, soit* $l$ *la dernière tâche du schedule, il suffit alors d'ignorer toutes les tâches* $l+1, l+2, \ldots$ *(on ne modifie pas la valeur de* $c_{MAX}$ *vu que c'est* $l$ *qui cause sa valeur).*
$\rightarrow l$ *est maintenant la tâche la plus petite.*
*(ceci est vrai car* $n > m$*)*

**Exemple 3.12** *Ici* $l = 3$*, on ignore donc* $4$.

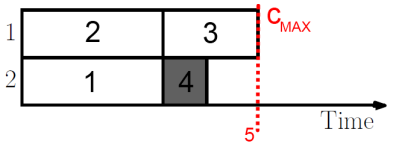



FIGURE 21 – Situation considérée

**Que savons nous de** $p_n$ **?**

*a) si* $p_n \leq \frac{C^*_{max}}{3}$

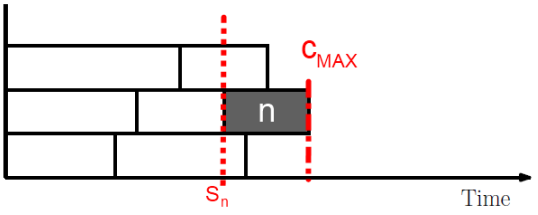



FIGURE 22 – Situation considérée

*Et donc on a :*

$$\begin{aligned} c_{MAX} &= S_n + p_n \\ &\leq c^*MAX + p_n \\ &< c^*MAX + \frac{c^*MAX}{3} = \frac{4}{3}c^*MAX \end{aligned}$$

*b) si* $p_n > \frac{c^*_{MAX}}{3}$*, par le lemme précédent,* **LPT** *donne la solution optimale.*

□

## 3.4 Traveling Saleman Problem (TSP)

**Problème 3.5 MIN Traveling Saleman Problem (TSP)**

* **Instance** : *ensemble de ville* $= \{1, 2, \ldots, n\}$ *ainsi qu'une matrice de distances (ou coûts)* $C = C_{ij}$.
  ↪ **Hypothèses :** *coûts* **symétriques** *et* **non négatifs** $(\forall i, j \ C_{ij} = C_{ji} \geq 0)$ *et* $\forall i, \ C_{ii} = 0$.
* **Solution** : *Un tour de villes, i.e. une permutation des nombres de 1 à n (on note k(1),k(2), ..., k(n)).*
* **Mesure** : $\left[\sum_{i=1}^{n-1} C_{k(i)k(i+1)}\right] + C_{k(n)k(1)}$

**Remarque 3.2** *De manière équivalente, on peut décrire une instance via un graphe complet non dirigé et pondéré avec les coûts/distances.*

**Exemple 3.13**

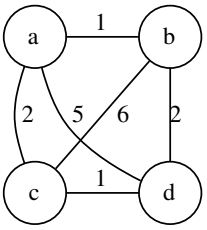



FIGURE 23 – Exemple d'instance de **TSP** sour forme de graphe pondéré

- *le tour optimal est* $abdc$ *et son coût est* $6$,
- $(n-1)!$ *possibilités (pas* $n!$ *car on peut considérer que la première ville est fixée),*
- *l'instance n'est pas métrique, c'est-à-dire qu'elle ne respecte pas l'inégalité triangulaire. (En effet,* $C_{ab} + C_{bd} \geq C_{ad}$ *est faux car* $1 + 2 = 3 < 5$*)*

Ce problème nous fait penser à un problème déjà vu lors des années précédentes, le problème de décision consistant à savoir si un graphe possède un cycle hamiltonien *(cycle passant une et une seule fois par tous les sommets)*. Voyons si on peut effectuer un rapprochement entre les deux.

**Problème 3.6 Problème Cycle-Hamiltonien ?**

* **Instance** : *Graphe* $G = (V, E)$ *non dirigé.*
* **Question** : $G$ *possède-t-il un cycle hamiltonien ?*

**Exemple 3.14** *Existe-t-il un cycle hamiltonien dans un cube ?*
$\Longrightarrow$ *oui, voir figure 25.*

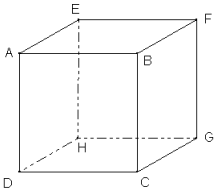



FIGURE 24 – Cube

*(il en existe plusieurs)*

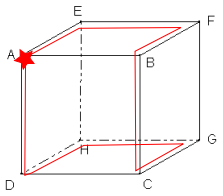



FIGURE 25 – Chemin hamiltonien dans un cube

Comment pourrait-on passer du problème du cycle Hamiltonien au **TSP** ?
↪ A partir de $G = (V, E)$ *(instance du problème Hamiltonien)*, on construit une instance **TSP** comme ceci :

- $c_{ij} = 1$ si $(i, j) \in E$
- $c_{ij} = n + 2$ sinon

La valeur optimale du **TSP** sur cette instance devrait être égal à $n$, cela signifiant qu'il existe un cycle hamiltonien. Sinon la solution optimale du **TSP** $\geq (n-1) + (n+2) = 2n + 1$ vu qu'il faut au moins sélectionner une arête qui n'existait pas dans l'instance du cycle.

Cette remarque nous dit qu'avec un algorithme de 2-approximation pour le **TSP**, on pourrait résoudre un problème **NP-complet** ! Ce facteur 2 vient du cout que l'on a placé pour les arêtes inexistantes. On peut donc le faire croître arbitrairement $\Rightarrow$ il n'existe donc pas d'algorithmes d'approximation pour le **TSP**. Pour s'en convaincre, il suffit de modifier l'instance précédente en plaçant comme coût pour les arêtes inexistantes un coût égal à $(\alpha - 1)n + 2$ et ce $\forall \alpha > 1$. On obtient donc :

$$\begin{aligned} OPT &= n \text{ s'il existe un cycle hamiltonien} \\ &= (n-1) + (\alpha - 1)n + 2 = \alpha n + 1 \text{ sinon} \end{aligned}$$

Ceci nous amène à une conclusion plutôt décevante, décrite dans le théorème suivant.

**Théorème 3.7** *Pour tout $\alpha > 1$, il n'y a pas d'algorithme d'approximation pour* **TSP**, *à moins que $\mathcal{P} = \mathcal{NP}$.*

**Remarque 3.3** *Ce résultat est dû au fait que les instances considérées ne sont pas métriques. Nous allons donc nous restreindre au* **TSP** *métrique.*

**Exemple 3.15** *(***TSP** *métrique)*

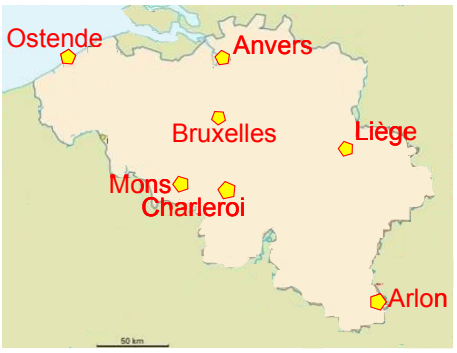
- $c(i, j) = 1$
- $c(j, k) = 1$
- $c(i, k) = 2$

$\Rightarrow$ *on ne peut mettre plus de 2 pour $c(i, k)$ sinon on ne respecte pas l'inégalité triangulaire.*
*Dans le cas du* **TSP** *non métrique, on mettait $(\alpha - 1)n + 2$ ... !*

Développons une approche gloutone pour résoudre **TSP**.
On note $S$ la permutation finale et $S_i$ la permutation partielle contenant que $i$ villes.
Nous allons construire la solution $S$ en ajoutant à chaque itération la ville la plus proche de l'ensemble des villes déja construite.
Voici cet algorithme, il est polynomial.

**Algorithme 9** NearestAddition

1: $i, j \leftarrow arg\min_{i,j \in S} C_{ij}$
2: $tour \leftarrow [i, j]$
3: $reste \leftarrow S \setminus \{i, j\}$
4: **Tant que** $reste \neq \emptyset$ **faire**
5: $\quad i, j \leftarrow arg\min_{i \notin reste, j \in reste} C_{ij}$
6: $\quad$ insérer $j$ dans $tour$ après $i$
7: $\quad reste \leftarrow reste\{j\}$
8: **fin Tant que**




| | Arlon | Bruxelles | Charleroi | Liège | Mons | Ostende |
|---|---|---|---|---|---|---|
| Anvers | 230 | 47 | 105 | 133 | 110 | 119 |
| | Arlon | 187 | 172 | 127 | 209 | 308 |
| | | Bruxelles | 60 | 97 | 67 | 113 |
| | | | Charleroi | 94 | 50 | 167 |
| | | | | Liège | 127 | 211 |
| | | | | | Mons | 145 |

FIGURE 26 – Carte et distances de la Belgique

**Exemple 3.16** *Les villes de Belgique*

*Appliquons l'algorithme* **NearestAddition** *:*
*On commence par* $S_2 = [Anvers, Bruxelles]$*, et on cherche la distance* $d_{ij}$ *la plus petite telle que* $i \in S_k$ *et* $j \notin S_k$*.*
*On trouve* $d_{ij} = 60$ *pour Bruxelles-Charleroi. On a donc* $S_3 = [Anvers, Bruxelles, Charleroi]$*.*
*On itère et on trouve :*

- $S_4 = [Anvers, Bruxelles, Charleroi, Mons]$
- $S_5 = [Anvers, Bruxelles, Charleroi, Liege, Mons]$
- $S_6 = [Anvers, Bruxelles, Ostende, Charleroi, Liege, Mons]$
- $S_7 = S = [Anvers, Bruxelles, Ostende, Charleroi, Liege, Arlon, Mons]$

*La valeur de la solution est :* **867***.*
*(La valeur optimale est de* **757***, en prenant* $S^* = [Anvers, Bruxelles, Liege, Arlon, Charleroi, Mons, Ostende]$*)*

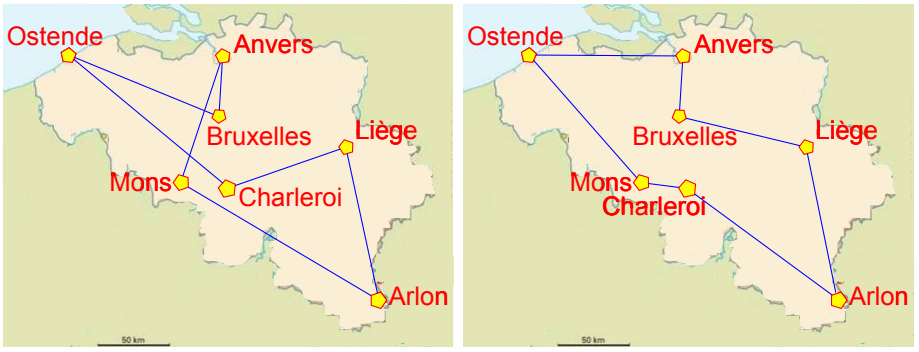



FIGURE 27 – Solution approchée par **NearestAddition** et solution optimale

Le coeur de l'analyse de cette algorithme nous amènera à une relation avec la notion d'*arbre couvrant minimum (Min Spanning Tree (MST))*. Un arbre couvrant d'un graphe connexe $G = (V, E)$ est un sous-ensemble minimal d'arêtes $F \subseteq E$ tel que $\forall i, j \in V$, il existe un chemin n'utilisant que des arêtes de $F$ allant de $i$ à $j$. Ce problème appartient à $P$ et l'algorithme de **Prim** est un algorithme exact polynomial permettant de le résoudre.

**Exemple 3.17**

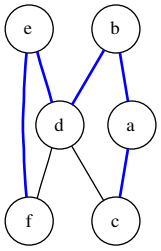



FIGURE 28 – Graphe et arbre couvrant

---

**Problème 3.7 MIN Spanning Tree (MST)**

* **Instance** : *Graphe* $G = (V, E)$ *pondéré,*
* **Solution** : *Arbre couvrant* $F \subseteq E$*,*
* **Mesure** : $\sum_{(i,j)\in F} C_{ij}$

---

**Algorithme 10** Prim (pour MST)

1: $S \leftarrow \{v\}$ où $v$ est un noeud arbitraire.
2: $F \leftarrow \emptyset$
3: **Tant que** $|S| < |V|$ **faire**
4: $\quad i, j \leftarrow arg\min_{i\in S, j\notin S} C_{ij}$
5: $\quad F \leftarrow F \cup \{(i, j)\}$
6: $\quad S \leftarrow S \cup \{j\}$
7: **fin Tant que**

**Exemple 3.18** *Appliquons l'algorithme de* **Prim** *sur l'algorithme de la Belgique. Il va sélectionner les arêtes :*

*(Ostende,BXL), (Anvers,BXL), (BXL,Charleroi), (Charleroi,Mons), (Charleroi,Liège), (Liège,Arlon)*

*soit exactement les mêmes que* **NearestAddition**.

**Lemme 3.3** *Pour toute instance du* **TSP** *métrique,* $OPT \geq$ **MST**.

**Preuve** *Soit* $n \geq 2$*, une instance de TSP métrique et son tour optimal : Supprimer une arête du tour*

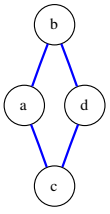



FIGURE 29 – Instance **TSP** métrique et son tour optimal

**OPT** *donne un arbre couvrant de poids* $w \leq OPT$*, or* $MST \leq w$ *par définition du problème* **MST** *et donc* $MST \leq OPT$*.*

□

**Théorème 3.8** *L'algorithme* **NearestAddition** *est un algorithme de* 2*-approximation pour le problème* **TSP** *métrique.*

**Preuve**
*Soient*

- $S_2$, $S_3$, …, $S_n = \{1, \ldots, n\}$ *où* $S_k$ *est le sous ensemble de sommets inclus dans le tour à une itération telle que* $|S_k| = k$,
- $F = \{(i_2, j_2), (i_3, j_3), \ldots, (i_n, j_n)\}$ *où* $(i_l, j_l)$ *sont les "i" et "j" identifiés dans la boucle de* **NearestAdd** *(ce sont les mêmes que pour* **Prim***).*

$T = (S, F)$ *est un arbre couvrant de* $G = (S, E)$ *et on sait que* $coût(T) = \sum_{l=2}^{n} C_{i_l j_l} = MST$.
↪ *Quel est le coût maximimal du tour construit par* **NearestAdd** *?*
→ *le premier tour sur* $i_2$ *et* $j_2 = 2C_{i_2 j_2}$ *(aller-retour).*
→ *Soit une itération où* $j$ *est inséré entre* $i$ *et* $k$*. La différence de coût est donnée par :*

$$C_{ij} + C_{jk} - C_{ik} \quad (\star)$$

*(ajout des 2 nouvelles arêtes, suppression de l'ancienne)*
*Par l'inégalité triangulaire, on sait que* $C_{jk} \leq C_{ji} + C_{ik}$ *et donc que*

$$C_{jk} - C_{ik} \leq C_{ij} \quad (\star\star)$$

*Le coût supplémentaire de cette itération est alors :* $C_{ij} + C_{jk} - C_{ik}$ *qui est borné par* $2C_{ij}$ *par* $(\star\star)$.

⇒ *En conclusion,* $APP$ *(coût du tour final donné par* **NearestAdd***) est tel que :*

$$\begin{aligned} APP &\leq 2 \sum_{l=2}^{n} C_{i_l j_l} \\ &= 2\ MST \\ &\leq 2\ OPT \text{ (par le lemme 3.3)} \end{aligned}$$

□

On peut également imaginer une autre approche avec les **multigraphes**, c'est-à-dire des graphes autorisant l'existence de plusieurs arêtes (ici 2) entre 2 sommets.

**Exemple 3.19**

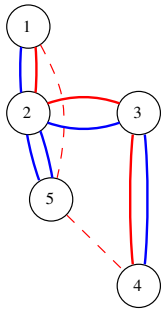



FIGURE 30 – Exemple de multigraphe

*Ce multigraphe est* **Eulérien***, c'est-à-dire qu'il existe une chemin empruntant chaque arête 1 et une seule fois. Essayons de trouver un tour sur ce graphe (en rouge) :*

- *on part de* 1 *et on va en* 2*, de* 2 *à* 3*, de* 3 *à* 4*,*
- *on veut retourner en* 3 *mais déjà visité, on va donc prendre un* **raccourci** *et aller directement en* 5*,*
- *on veut retourner en* 2 *mais déjà visité →* *aller directement en* 1*, bouclant ainsi le tour.*

Revenons sur la théorie des *cycles Eulériens*, via le problème de **Konïgsberg**.

**Exemple 3.20**

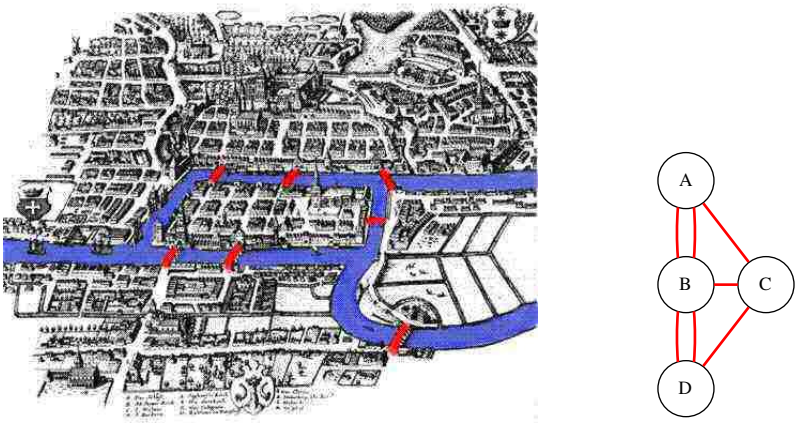



FIGURE 31 – Plan de la ville de Konïgsberg et le multigraphe associé

*Euler a dit en* **1736** *qu'une promenade passant une et une seule fois par chacun des ponts est impossible. Celui-ci n'avait pas utilisé la théorie des graphes pour prouver cela mais cela est facilement visible via un multigraphe où un noeud correspond à une île/rive et chaque pont est une arête.*

**Définition 3.3** *Un graphe est* **eulérien** *si et seulement si tous les sommets du graphes ont un degré pair et que le graphe est connexe.*

Le problème Eulérien $\in \mathcal{P}$, on peut donc l'utiliser pour approximer **TSP**.
Pour trouver un "bon" tour pour le **TSP** nous allons :

a) calculer un **MST**,

b) remplacer chaque arête par 2 copies $\Rightarrow$ multigraphe eulérien dont le coût est égal à $2\ MST \leq 2\ OPT$ (il est eulérien vu que tous les degrés sont multipliés par 2)

c) créer un tour qui utilise des "raccourcis" s'il le faut.
$\Rightarrow$ à partir d'un cycle eulérien, ne garder que les villes/sommets du parcours quand elles apparaissent pour la première fois $\Rightarrow$ "**Shortcut**"

On peut calculer une borne sur la longueur du tour obtenu en c). Les villes ommises n'augmentent pas le coût du multigraphe eulérien.
*(via l'inégalité triangulaire, si on prend un raccourci, on diminue (ou conserve) le coût)*
$\Rightarrow APP \leq 2\ OPT$, vu que la solution en c) $\leq$ celle en b) qui elle est $\leq 2\ OPT$.
$\Longrightarrow$ On a donc un algorithme de 2-approximation, on l'appellera **DoubleTree**.

**Théorème 3.9** *L'algorithme* **DoubleTree** *est un algorithme de 2-approximation pour* **TSP**.

**Preuve** *(cf analyse ci-dessus)*

□

**Algorithme 11** Double Tree

1: $MST \leftarrow PRIM(G)$
2: $DT \leftarrow double\_edges(MST)$
3: $walk \leftarrow eulerian\_walk(DT)$
4: $tour \leftarrow shortcut(walk)$
5: **Retourner** $cost(tour)$

**Exemple 3.21** *Appliquons* **Double Tree** *sur l'exemple de la Belgique.*
*On a déjà appliqué* **Prim** *précédemment (même résultat que* **NearestAddition***), on a donc :*
$F = \{(Anvers, Bruxelles),\ (Charleroi, Anvers),\ (Mons, Charleroi),$
$(Liege, Charleroi),\ (Ostende, Bruxelles),\ (Arlon, Liege)\}$
*On obtient donc l'arbre couvrant avec les arêtes données par* $F$. $\Rightarrow$ *3 raccourcis apparaissent :*

$$Arlon - Mons \qquad Mons - Ostende \qquad Ostende - Anvers$$

*On obtient le tour* $Anvers - Bruxelles - Charleroi - Liege - Arlon - Mons - Ostende = 801km$.

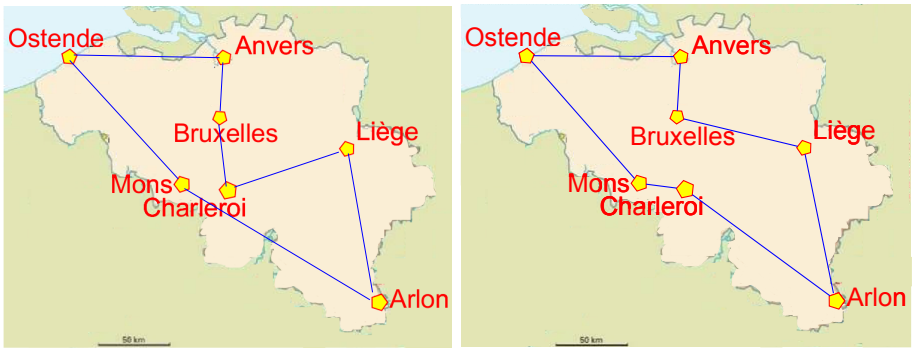



FIGURE 32 – Solution approchée par **DoubleTree** et solution optimale

L'idée clé utilisée pour prouver que l'algorithme est un algo de 2-approximation est que le cycle dans un graphe eulérien a un coût inférieur à $2\ OPT$. Donc, si on trouve un meilleur graphe eulérien, c'est-à-dire un graphe eulérien tel que le coût du cycle est $\leq \alpha OPT$, alors l'algorithme est d'$\alpha$-approximation. Il existe en effet un meilleur algorithme, donné par *Christophides*, qui est le meilleur algorithme connu à ce jour et qui a un facteur $\frac{3}{2}$.
Parlons d'abord des degrés des sommets d'un graphe et énonçons quelques propriétés :

1. La somme des degrés d'un graphe, $\Delta(G) = 2m$ où $m$ est le nombre d'arêtes car chaque arête est comptée 2 fois (ses 2 extrémités).
2. La somme des degrés est donc toujours **paire**.
3. La somme des degrés des sommets ayant un degré pair est toujours paire car c'est une somme de nombres pairs.
4. La somme des degrés des sommets ayant un degré impair est toujours paire. *(par 2. et 3., sinon la somme totale serait impaire)*
5. Le nombre de sommets de degrés impairs est pair. *(par 4.)*

Dans la suite, nous noterons $O$ l'ensemble des sommets de degré impair d'un graphe.

**Exemple 3.22** *Vérifions ces propriétés sur le graphe G (figure 33).*
*Le graphe possède* 7 *noeuds et* 6 *arêtes et les degrés sont :*

| *Sommet* $i$ | *Degré de* $i$ $d(i)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 3 |
| 4 | 1 |
| 5 | 2 |
| 6 | 2 |
| 7 | 1 |

- $\sum d(i) = 12 = 2 * 6 = 2$ *fois le nombre d'arêtes et c'est pair.*
- 4 *sommets de degrés impairs et la somme de leur degré* $\rightarrow 1 + 3 + 1 + 1 = 6$, *c'est pair.*
- $O = 2, 3, 4, 7$ *et* $|O| = 2 * k$ *(par 5)) pour un entier* $k$ *non négatif (ici* $k = 2$*).*

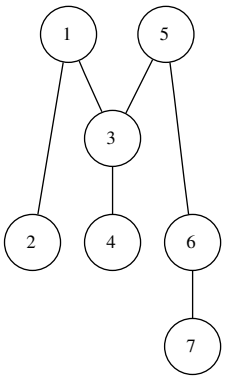



FIGURE 33 – Graphe G

Supposons qu'on groupe les sommets de degrés impairs par paires (ce qui est possible car $|0| = 2k$) : $(i_1, i_2)$, $(i_3, i_4)$ ... $(i_{2k-1}, i_{2k})$. On obtient ainsi un **perfect matching**, c'est-à-dire un ensemble d'arêtes non incidentes entre elles qui couvrent tous les sommets (ici, tous les sommets de $O$).

**Exemple 3.23** *Dans l'exemple précédent, associons* $(2, 3)$ *et* $(4, 7)$ *et on obtient un graphe avec des sommets comportants uniquement des sommets de degrés pairs.*

Il existe plusieurs **perfect matching** *(et plus k est grand plus il y en a)*
$\Longrightarrow$ le problème **MIN Perfect Matching** consiste à trouver le perfect matching de coût minimum.
$\rightsquigarrow$ Il existe un algorithme polynomial pour résoudre ce problème, l'algorithme d'**Edmonds** en $O(n^4)$.
*(voire* $O(n^3)$ *avec des SDD particulières)*

**Algorithme 12** Christophides (informel)

1: $MST \leftarrow PRIM(G)$
2: $O \leftarrow$ ensemble des sommets impairs de $MST$
3: $PM \leftarrow compute\_perfect\_matching(O)$
4: $EG \leftarrow MST \cup PM$
5: $tour \leftarrow shortcut(EG)$
6: **Retourner** $cost(tour)$

**Remarque 3.4** **EG** *est eulérien car il est connexe vu que* **MST** *l'était déjà et tous les degrés sont pairs puisque l'algorithme a ajouté exactement une arête à chaque sommet de degré impair, via le* **perfect matching**.

**Exemple 3.24** *Appliquons* **Christophides** *sur l'exemple de la Belgique.*

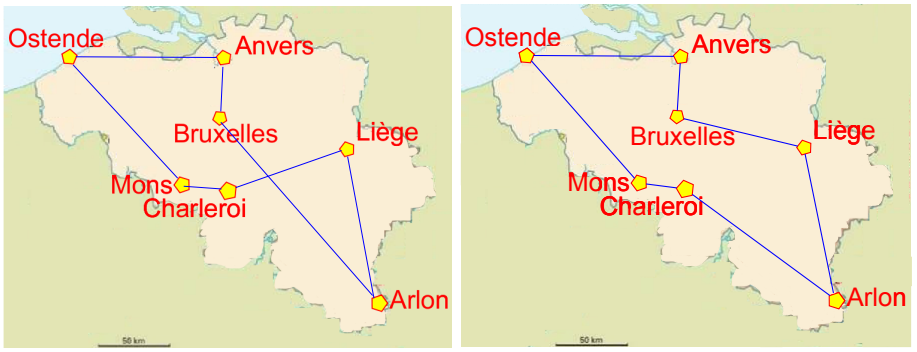



FIGURE 34 – Solution approchée par **Christophides** et solution optimale

*Seul Liège a un degré pair.*
*Le meilleur* **PM** *est :* $Anvers - Ostende$, $Charleroi - Mons$, $Bruxelles - Arlon$, *avec un poids de* 356 *km.*
*Chistophides donne donc le tour :*
$Anvers - Bruxelles - Arlon - Liege - Charleroi - Mons - Ostende \rightarrow 769$*km.*
*(Rappel :* $OPT = 757$ *km)*

**Théorème 3.10** *L'algorithme de Christophides est un algorithme de* $\frac{3}{2}$*-approximation pour le problème* **TSP-métrique**.

**Preuve**
*Il reste à prouver que le graphe eulérien EG a un coût total* $\leq \frac{3}{2}OPT$.
*On sait que* $cout(EG) = cout(MST) + cout(PM)$ *et que* $cout(MST) \leq OPT$, *on doit donc prouver :*

$$cout(PM) \leq \frac{OPT}{2}$$

.

*a) Observons d'abord qu'il existe un tour sur les sommets de* $O$ *qui a un coût* $\leq OPT$. *Soit un tour optimal sur toutes les villes, il suffit de prendre des shortcuts sur ce tour optimal pour obtenir un tel tour. Et via les arguments déjà cités (inégalité triangulaire), le tour ainsi créé est* $\leq OPT$.

*b) Sur ce tour, il y a 2* **Perfect Matching** *(pour rappel, il s'agit d'un ensemble d'arêtes non incidentes*

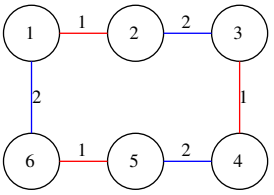



FIGURE 35 – Exemple de tour sur 6 villes $\in O$

*entre elles couvrant chaque noeud) (on numérote les arêtes) soit en prenant les arêtes numérotées impaires soit en prenant les arêtes numérotées paires.*
*Vu que la somme des couts de ces 2 perfect matching* $\leq OPT$, *au moins un des 2 (le minimum perfect matching est peut-être encore plus petit) est tel que* $PM \leq \frac{OPT}{2}$.

□

Ce facteur d'approximation est serré, nous allons le montrer sur l'exemple suivant.

**Exemple 3.25**

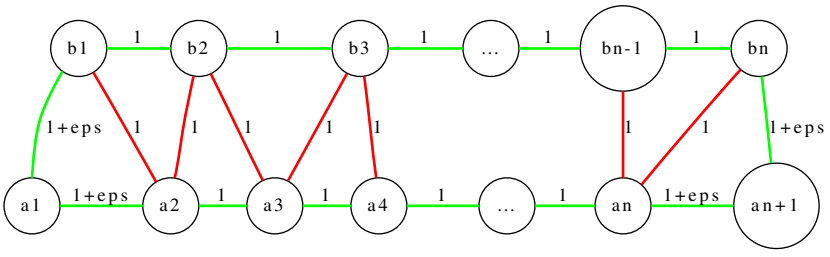



FIGURE 36 – Exemple serré pour le facteur d'approximation

*Si le tour ne prend que les arêtes vertes, on obtient la solution optimale dont le coût est :*

$$(n-1) + 4(1+\epsilon) + (n-2) = 2n - 1 + 4\epsilon$$

*Seulement, si l'arbre couvrant donné par l'algorithme est toutes les arêtes rouges (+ les 2 arêtes vertes extérieures), il n'y a que $2$ sommets de degrés impairs : $a_1$ et $a_{n+1}$. Le* **perfect matching** *va donc devoir les relier, on obtient alors une solution dont la valeur est :*

$$2(1+\epsilon) + 2(n-1) + n = 3n + 2\epsilon$$

*Le ratio est donc :*

$$\frac{3n+2\epsilon}{2n+4\epsilon-1} \rightarrow_{n\to\infty} \frac{3}{2}$$

$\Longrightarrow$ **Il n'y a pas de meilleur d'algorithme d'approximation à ce jour pour le TSP métrique.**

**Théorème 3.11 (Résultat d'inapproximabilité pour TSP)**
*A moins que $\mathcal{P} = \mathcal{NP}$, pour toute constante $\alpha \leq \frac{220}{219} \simeq 1,0045$, il n'existe pas d'algorithme d'approximation pour le TSP métrique.*

↪ Voir exercices dans l'annexe C.

# 4 Programmation dynamique et arrondissement (rounding) de données

La programmation dynamique est une technique classique en algorithmique où une solution optimale est construite à partir de solutions optimales de sous-problèmes, en général stockées dans une table ou un tableau multidimensionnel. Nous allons utiliser cette technique pour des problèmes "faiblement" NP-difficiles en ce sens où il existe un algorithme pseudo-polynomial, c'est-à-dire presque polynomial.

## 4.1 Le problème du sac à dos (knapsack problem)

---

**Problème 4.1 MAX sac à dos KP**

* **Instance** *:*
— *ensemble d'objets* $I = \{1, 2, \ldots, n\}$*, chaque objet a une valeur* $v_i$ *et une taille* $s_i$*,*
— *le sac à dos a une capacité* $B$ *;*
↪ *Hypothèses :*
— *toutes les données sont des entiers (*$v_i$*,* $s_i$ *et* $B$*),*
— $\forall i,\ s_i \leq B$
* **Solution** *:* $S \subseteq I$ *tel que* $\sum_{i \in S}(s_i) \leq B$
* **Mesure** *: le profit, c'est-à-dire* $\sum_{i \in S}(v_i)$.

---

Quelques applications :

- utilisé pour minimiser les chutes dans les problèmes de découpe,
- sélection de porte-feuilles et d'investissements financiers,
- génération de clés dans le système cryptographique de **Merkle-Hellman**.

**Exemple 4.1**
$I = \{1, 2, 3, 4\}$
$v_1 = 4,\ s_1 = 2$
$v_2 = 5,\ s_2 = 3$
$v_3 = 3,\ s_3 = 2$
$v_4 = 2,\ s_4 = 1$

$OPT$ *pour* $B = 8$ *?* $\rightarrow$ *tout prendre* $\longrightarrow size = 8$ *et* $profit = 14$
$OPT$ *pour* $B = 6$ *?* $\rightarrow \{1, 2, 4\} \longrightarrow size = 6$ *et* $profit = 11$
$OPT$ *pour* $B = 5$ *?* $\rightarrow \{1, 2\}$ *ou* $\{1, 3, 4\} \longrightarrow size = 5$ *et* $profit = 9$
$OPT$ *pour* $B = 4$ *?* $\rightarrow \{1, 3\}$ *ou* $\{2, 4\} \longrightarrow size = 4$ *et* $profit = 7$

Pourquoi ne pas essayer un algorithme glouton sur ce problème ? L'idée est de trier les objets selon un ordre choisi puis les ajouter dans cet ordre dans le sac. Nous allons envisager plusieurs ordres et montrer qu'à chaque ordre il existe une instance où on peut faire aussi mauvais que possible.

---

**Algorithme 13** GloutonKPGeneral

1: Trier les objets selon une certaine règle.
2: Placer les objets dans le sac dans cet ordre jusqu'à ce que le sac soit trop plein.

---

**Les règles possibles** :

- **taille croissante** : on prend un objet de taille $1$ et de valeur $1$ et un objet de taille $B$ et de valeur $M > 1$. La valeur optimale est $M$ et la valeur approchée est $1$. Le ratio est donc $1/M$ et on peut faire grandir $M$ tant que l'on veut.
- **valeur décroissante** : on prend un objet de taille $B$ et de valeur $M$ et $B$ objets de taille $1$ et de valeur $M - \epsilon$ ($\epsilon$ petit). La valeur optimale est donc $B(M - \epsilon) \sim BM$ (avec $\epsilon \to 0$) et la valeur approchée est $M$. Le ratio est donc $1/B$ avec $B$ que l'on peut faire grandir tant que l'on veut.

- **ratio valeur/taille décroissant** : on prend un objet de taille $1$ et de valeur $1+\epsilon$ et un objet de taille $B$ et de valeur $B$. La valeur optimale est donc $B$ et la valeur approchée est $1+\epsilon \sim 1$. Le ratio est donc $1/B$ avec $B$ que l'on peut faire grandir tant que l'on veut.

$\Rightarrow$ **les algorithmes gloutons ne sont pas très bons.**

Posons nous les bonnes questions : dans le cas d'une recherche exhaustive,

- combien y a-t-il de possibilités ? $\rightarrow 2^n$
- comment représenter graphiquement cela ? $\rightarrow$ arbre binaire complet où chaque niveau $i$ représente la question "prend-on l'objet $i$ dans le sac ?".

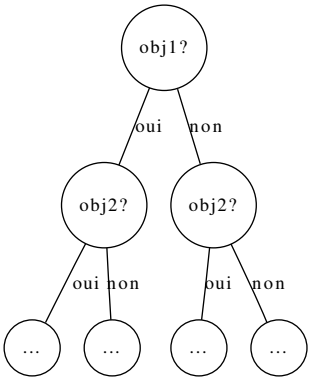



FIGURE 37 – Arbre de décision type

**Exemple 4.2** *(exemple précédent) La figure 38 contient l'arbre de décision relatif à l'exemple précédent. Chaque noeud contient un couple $(x, y)$, où $x$ = taille de la solution partielle et $y$ = profit de cette solution partielle.*

**Remarques 4.1** *(sur l'exemple)*

- *On peut ne pas énumérer tout l'arbre car à certains niveaux, on a des noeuds qui ont la même taille mais pas le même profit, ce qui signifie que pour les mêmes éléments avec la même taille on a un meilleur profit, il est donc inutile de générer le reste du sous-arbre concerné.*
- *On peut également avoir une situation identique si on a le même profit et que la taille est plus petite.*
- *On peut également supprimer 2 branches donnant la même taille et le même profit, vu que les 2 solutions intermédiaires sont identiques, on choisit celle que l'on veut.*

Formalisons nos observations : soient $P$ et $Q$, 2 sous-problèmes au "même niveau"
(i.e. qui ont considérés les mêmes objets) de $1$ à $j$ :

si $taille(P) = taille(Q)$
  si $profit(P) \geq profit(Q)$ alors calculer $P$
  sinon calculer $Q$
sinon si $profit(P) = profit(Q)$
  si $taille(P) > taille(Q)$ alors calculer $Q$
  sinon calculer $P$

Pour **KP** :

- la notion de sous-problème est simple et naturelle,
- il y a beaucoup de redondance dans la recherche exhaustive, d'où l'interêt d'avoir des données entières.

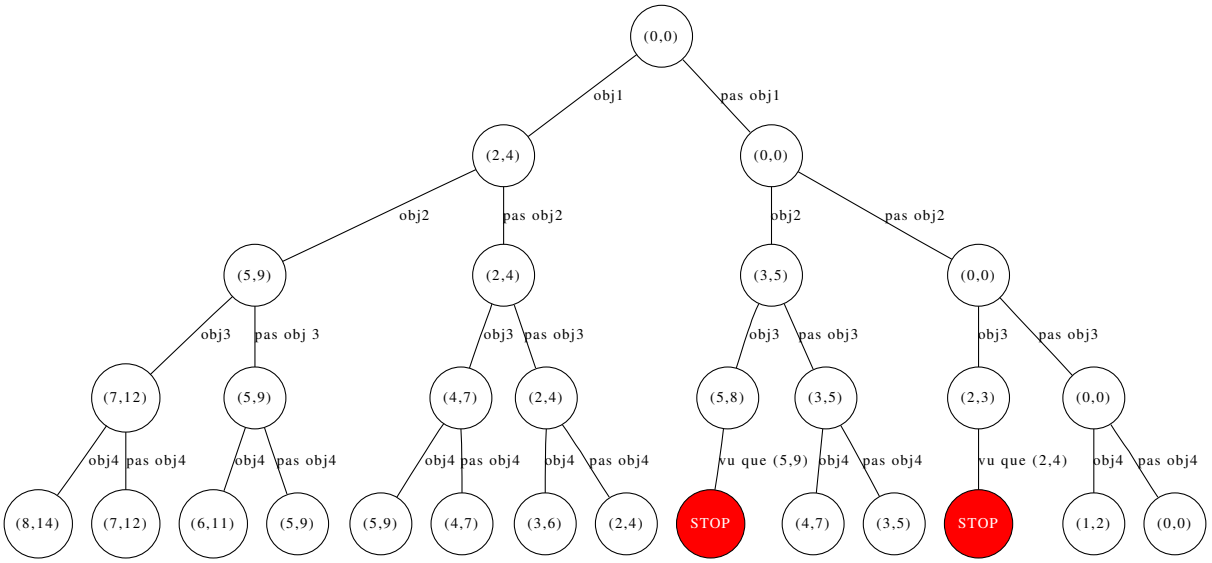



FIGURE 38 – Arbre de décision relatif à l'exemple

### 4.1.1 Programmation dynamique pour KP (1ère version)

→ on suppose qu'on a considéré les objets de 1 à $i$,
→ on veut se souvenir des profits qui sont possibles,
→ pour chaque profit possible on veut minimiser la taille associée.

Nous allons utiliser les notations suivantes,

- $S(i,p)$ est le sous-ensemble de $I$ tel que le **profit** est égal à $p$ et la **taille** est **minimisée**,
- $A(i,p)$ représente la taille de $S(i,p)$, vaut $\infty$ si il n'existe pas d'ensemble $S(i,p)$.

**Exemple 4.3** *(exemple précédent)*

$$\begin{aligned} A(1,p) &= s_1 \textit{ si } p = v_1 \\ &= 0 \textit{ sinon si } p = 0 \\ &= \infty \textit{ sinon} \end{aligned}$$

*Le tableau $A$ est donc de la forme :*

| $Objet \backslash P$ | 0 | 1 | 2 | 3 | 4 | ... |
|---|---|---|---|---|---|---|
| 1 | 0 | $\infty$ | $\infty$ | $\infty$ | 2 | ... |
| 2 | ... | ... | ... | ... | ... | ... |
| 3 | ... | ... | ... | ... | ... | ... |
| $\vdots$ | ... | ... | ... | ... | ... | ... |

*Il y a $V$ colonnes, où $V = \sum_{k=1}^{n} v_k$ et $n$ lignes, c'est donc une matrice $n \times V$.*

Nous voulons trouver une formule inductive pour calculer le tableau $A$, c'est-à-dire une formule pour calculer la $(i+1)^{\text{ème}}$ ligne à partir de la ligne $i$.

$$\begin{aligned} A(i,p) &= \text{taille minimum pour un profit p en utilisant les } i \text{ premiers objets} \\ A(i+1,p) &= A(i,p) \text{ si on a pas pris } i+1 \\ &= A(i, p - v_{i+1}) + s_{i+1} \text{ si on prend } i+1 \\ &= \text{en fait le minimum de ces 2 valeurs vu qu'on prend la taille minimum} \end{aligned}$$

**Exemple 4.4** *Remplir la table $A(i,p)$ pour l'exemple. (case blanche $= \infty$)*

| $A$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | | | | 2 | | | | | | | | | | |
| 2 | 0 | | | | 2 | 3 | | | | 5 | | | | | |
| 3 | 0 | | | 2 | 2 | 3 | | 4 | 5 | 5 | | | 7 | | |
| 4 | 0 | | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 6 | 6 | 7 | | 8 |

*On peut calculer l'optimum facilement, par exemple pour $B = 6$, le $6$ plus à droite donne $11$.*

L'algorithme utilisant cette table est donc en $O(n \times V)$, ce qui semble polynomial mais le problème est que $V$ n'est pas une constante vu qu'elle dépend des valeurs que prennent les entrées.
De **manière intuitive**, si $V$ est beaucoup plus grand que $n$, le nombre de colonnes dans le tableau ne pourra pas être bornée par un polynôme en $n$.
De **manière plus formelle**, soit "$size$" la taille utilisée pour représenter $V$, quelle est la complexité de notre programme dynamique en fonction de $n$ et "$size$" ? Sur nos machines *(binaires)* : $size = \log_2(V)$ et donc $V = 2^{size}$, $\Rightarrow O(n \times 2^{size})$.

**Définition 4.1** *Un algorithme* **polynomial** *quand les entrées sont représentées sous forme unaire est un algorithme* **pseudopolynomial**.

**Remarque 4.1** *En unaire : $(7)_{10} = (1111111)_1$, et dans ce cas $size = O(V)$ et donc $O(n \times size)$ mais évidemment size peut être énorme.*

### 4.1.2 Variation du programme dynamique

Nous allons utiliser un tableau de listes de paires :
$A(j)$ pour $j = 1, ..., n$ contient une liste de paires $(t, w)$ où une paire signifie qu'il existe un sous ensemble $S \subseteq I$ utilisant les $j$ premiers objets avec une **taille** $t$ et un **profit** $w$.
En d'autres mots si $(t, w)$ est dans la liste $A(j)$, alors il existe $S \subseteq \{1, 2, ..., j\}$ tel que $\sum_{j \in S} s_i = t \leq B$ et $\sum_{i \in S} v_i = w$.

**Définition 4.2** *On dit qu'une paire $(t, w)$* **domine** *une paire $(t', w')$ si $t \leq t'$ et $w \geq w'$. (relation transitive)*

Nous allons nous assurer que dans une liste, aucune paire n'en domine une autre. On peut donc supposer que $A(j)$ est de la forme $(t_1, w_1),\ (t_2, w_2),\ \ldots,\ (t_k, w_k)$ et est telle que :

$$t_1 < t_2 < \ldots < t_k$$
$$w_1 < w_2 < \ldots < w_k$$

Comme pour tout $i$, $v_i$ et $s_i$ sont des entiers :

a) dans chaque liste, il y a au plus $B+1$ paires ($+1$ car il y a la taille $0$)

b) dans chaque liste, il y a au plus $V+1$ paires ($+1$ car il y a le profit $0$)

c) pour tout sous-ensemble $S \subseteq \{1,...,j\}$ réalisable (i.e $\sum_{i\in S}(s_i) \leq B$), la liste $A(j)$ contient une paire $(t,w)$ qui donne la paire $(\sum_{i\in S}(s_i),\sum_{i\in S}(v_i))$.

**Algorithme 14** DynProg_KP

$A(1) \leftarrow \{(0,0),(s_1,v_1)\}$
**Pour** $j$ allant de 2 à $n$ **faire**
  $A(j) \leftarrow A(j-1)$
  **Pour** chaque paire $(t,w) \in A(j-1)$ **faire**
    **Si** $t+s_j \leq B$ **alors**
      Ajouter $(t+s_j,w+v_j)$ à $A(j)$
    **fin Si**
  **fin Pour**
  Retirer les paires dominées de $A(j)$
**fin Pour**
Retourner $\max_{(t,w)\in A(n)}(w)$

$\Longrightarrow O(n \times min(B,V))$

**Exemple 4.5** $A(1) \leftarrow \{(0,0),(2,4)\}$
*Itération* $j=2$
$A(2) \leftarrow \{(0,0),(2,4)\}$
- $(0,0)$ *:* $0+3 \leq_? 5 \rightarrow$ *oui donc ajouter* $(3,5)$
- $(2,4)$ *:* $2+3 \leq_? 5 \rightarrow$ *oui donc ajouter* $(5,9)$

...

**Théorème 4.1** **DynProg_KP** *calcule la valeur exacte pour* **KP**.

**Définition 4.3** *Un* **schéma d'approximation en temps polynomial** *(***PTAS***) est une famille d'algorithmes* $\{A_\epsilon\}$ *telle qu'il existe un algorithme pour chaque valeur* $\epsilon > 0$ *de sorte que* $A_\epsilon$ *est un algorithme de*

$$\begin{cases} (1-\epsilon) \text{ (pour un MAX)} \\ (1+\epsilon) \text{ (pour un MIN)} \end{cases} \text{-approximation.}$$

**Définition 4.4** *Si pour un* **PTAS** *chaque algorithme* $A_\epsilon$ *a un temps d'exécution borné par un polynome en* $\frac{1}{\epsilon}$*, alors on parle de schéma d'approximation complet (***FPAS***).*

**Algorithme 15** FPAS_KP

**Entrée :** $\epsilon > 0$
$M \leftarrow \max_{i\in I}(v_i)$
$\mu \leftarrow \frac{(\epsilon M)}{n}$
**Pour** tous les $i \in I$ **faire**
  $v'_i \leftarrow \left\lfloor (\frac{v_i}{\mu}) \right\rfloor$
**fin Pour**
Exécuter **DynProg_KP** sur instance modifiée

On peut réellement avec cet algorithme donner la précision désirée. Evidemment, plus la précision est grande plus l'algorithme sera lent. Pour une idée sur la précision, si on prend par exemple $\mu = 100$, on ne fait plus la différence entre la valeur $2000$ et la valeur $2099$.

↪ Comment pouvons-nous être sûrs que la garantie est bien $\epsilon$ ?

Idées :

- "mesurer" les profits en multiples entiers de $\mu$,
- exécuter sur instance $(s_i, v'_i) \rightarrow$ le nombre de colonnes diminue,
- retourner la valeur de l'instance modifiée comme la valeur approchée de l'instance originale.
- $M = \max_{i \in I}(v_i) \Rightarrow M \leq OPT$ **(1)**

Pourquoi choisir $\mu = \frac{\epsilon M}{n}$ ?

$\rightarrow$ supposons qu'on accepte une erreur d'au plus $\mu$ pour un profit,

$\rightarrow$ au total on a un profit maximum qui a été changé d'au plus $n\mu$,

$\rightarrow$ on veut que cette erreur $(n\mu)$ soit au plus un paramètre $\epsilon$ multiplié par une borne inférieur sur $OPT$ ($M$ par exemple) $\longrightarrow$ Choisissons $n\mu = M\epsilon$.

**Théorème 4.2** *L'algorithme* **FPAS_KP** *est un* **FPAS** *pour* **KP**.

**Preuve**

*a) Le temps d'exécution est borné par un polynôme en* $\frac{1}{\epsilon}$.
*On observe :*

$$\begin{aligned} V' &= \sum_{i=1}^{n} (v'_i) \\ &= \sum_{i=1}^{n} \left\lfloor \frac{v_i}{\mu} \right\rfloor \\ &= \sum_{i=1}^{n} \left\lfloor \frac{v_i}{\frac{\epsilon M}{n}} \right\rfloor \\ &\leq \sum_{i=1}^{n} \frac{Mn}{\epsilon M} \\ &= \frac{n \times n}{\epsilon} \\ &= n^2 \left(\frac{1}{\epsilon}\right) \end{aligned}$$

$\Rightarrow V' = O\left(n^2 \left(\frac{1}{\epsilon}\right)\right)$

$\Longrightarrow$ *Temps d'exécution* $O(n \min(B, V)) = O\left(n^3 \times \left(\frac{1}{\epsilon}\right)\right)$

*b) Le facteur d'approximation est de* $1 - \epsilon$, *c'est-à-dire* $APP \geq (1 - \epsilon)OPT$.
*Soit* $S$ *l'ensemble des objets utilisés dans la solution approchée (c'est-à-dire celui retourné par* **FPAS_KP***).*
*Soit* $O$ *l'ensemble optimal d'objets, on sait déjà que* $M \leq OPT$, *de plus,* $\mu v'_i \leq_{(2)} v_i \leq_{(3)} \mu(v'_i + 1)$
$\Longrightarrow$ *par* **(3)**, $\mu v'_i \geq v_i - \mu$ **(4)**

*Dès lors,*

$$
\begin{aligned}
APP &= \sum_{i \in S} v_i \\
&\geq \sum_{i \in S} \mu v'_i \text{ (par } \textbf{(2)}\text{)} \\
&\geq \sum_{i \in O} \mu v'_i \text{ (parce que } S \text{ est optimal sur les } v'_i \text{ et } O \text{ reste réalisable sur les } v'_i \text{ et vu que c'est une maximisation,} \\
&\quad \text{la valeur de } S \text{ est la plus grande et donc plus grande que celle de } O \text{ en particulier)} \\
&\geq \sum_{i \in O} (v_i - \mu) \text{ par } \textbf{(4)} \\
&= \sum_{i \in O} (v_i) - |O|\mu \\
&= OPT - |O|\mu \\
&\geq OPT - n\mu \text{ car } n \geq |O| \\
&= OPT - M\epsilon \text{ par définition de } \mu \\
&\geq OPT - OPT\epsilon \text{ par } \textbf{(1)} \\
&= (1 - \epsilon)\, OPT
\end{aligned}
$$

□

↪ Voir exercices dans l'annexe D.

# A Annexe A : Exercices chapitre 1

## A.1 Vertex Cover attaqué par un algorithme glouton simple

**Donner un algo/une heuristique qui va donner une solution approchée**

**Algorithme 16** MonAlgorithme

1: sommetsPris $\rightarrow 0$
2: **Tant que** sommetsPris $<$ nombreDeSommets **faire**
3: Ajouter le sommet de degré max à la couverture et le supprimer du graphe.
4: **fin Tant que**

**Algorithme 17** AlgorithmeMélot

1: Trouver un sommet $v$ de degré maximum
2: Ajouter v dans la solution et retirer $v$ et toutes les arêtes incidentes à $v$
3: Répéter jusqu'à ce qu'il n'y ait plus d'arêtes.

**Essayer l'algo sur l'exemple et trouver un facteur d'approx**

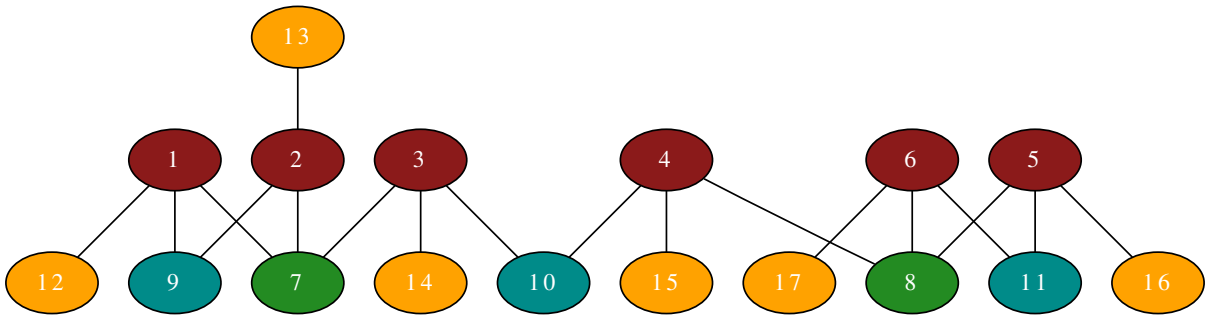



FIGURE 39 – Exemple pour le Vertex Cover

$\boxed{OPT = 6 = k!}$ *(on prend tous les rouges).*

Il y a 8 sommets de degré maximal, les 6 rouges et les 2 verts, supposons qu'on fasse les mauvais choix à chaque fois (on prend les verts). Ensuite on a plus que des degrés 2 (les rouges) et les bleus. On prend les bleus (mauvais choix) (...) $\Rightarrow$ **Au final on a 11 noeuds.**

**Y a-t-il moyen de généraliser le facteur d'approximation ?**

Analysons les sommets et leurs degrés, on a :

- les sommets $1, 2, 3, 4, 5$ et $6 \rightarrow k!$ sommets de degrés $k$,
- les sommets $7$ et $8 \rightarrow \frac{k!}{k}$ sommets de degrés $k$,
- les sommets $9, 10$ et $11 \rightarrow \frac{k!}{k-1}$ sommets de degrés $k-1$,
- les sommets $12, 13, 14, 15, 16$ et $17 \rightarrow k!$ sommets de degrés $1$.

En sachant que :

$$\frac{1}{k} + \frac{1}{(k-1)} + \ldots + 1 \sim \log k$$

On a

$$SOL = \frac{k!}{k} + \frac{k!}{(k-1)} + k! = k!\left(\frac{1}{k} + \frac{1}{(k-1)} + 1\right) \sim k! \log k$$

Et donc,

$$\boxed{\frac{SOL}{OPT} \sim \frac{k! \log k}{k!} \sim \log k}$$

**Cet algorithme possède-t-il un facteur d'approximation $\alpha$ constant ?**

Peut-être, mais on a pas prouvé que c'était le cas ni que c'était pas le cas, on a juste vu que dans ce cas là on avait un ratio de l'ordre de $\log k$.

# B Annexe B : Exercices chapitre 2

## Montrer par un exemple que VC est un cas particulier de SC

- Instance de $VC$ : $G = (V, F) \rightarrow F$ qui doit être couvert.

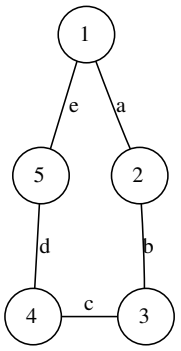



FIGURE 40 – $OPT = C = \{1, 3, 4\}$

- Instance de $SC$ : $E = F$, $S_j$ correspond à $V$ et contient toutes les arêtes incidentes au sommet $j$.

a b

e d c

★ $S1 = \{a, e\}$
★ $S2 = \{a, b\}$
★ $S3 = \{b, c\}$
★ $S4 = \{c, d\}$
★ $S5 = \{d, e\}$

Pour tout couverture $C$ dans $G$, il y a un set cover $I = C$. Vérifions donc que $S_1$, $S_3$ et $S_4$ couvrent bien $E$ ... c'est le cas ! $\longrightarrow$ **En particulier, c'est également vrai pour les solutions optimales.**

## B.1 Programmation Linéaire et Set Cover

### Écrire un problème SC sous la forme d'un PL en nombre entiers

Cet **IP** est donné par :

a) une solution $I \subseteq \{1, 2, ..., m\} \rightarrow$ "Le sous-ensemble $j$ est dans la solution"

$$\Rightarrow \text{Pour chaque } j,\ x_j = 1 \text{ si } j \in I, 0 \text{ sinon.}$$

b) "Chaque élément doit être couvert"

$$\Rightarrow \sum_{j:e_i \in S_j} (x_j) \geq 1, \forall i \in \{1, ..., n\}$$

c) "Il faut minimiser le poids total"

$$\Rightarrow f_{obj} = \sum_j x_j.w_j$$

On peut construire un **LP** en relaxant la contrainte $x_j \in \{0, 1\}$ en la modifiant en $x_j \geq 0$ et on obtient :
$\min \sum_{j=1}^{m} w_j.x_j$
**s.l.c** $\sum_{j:e_i \in S_j} (x_j) \geq 1,\ \forall i = 1, ..., n$
$x_j \geq 0,\ \forall j = 1, ..., m$

**Formuler l'IP de l'exemple ci-dessous**

$$\begin{aligned}
&\min\ 5x_1 + 5x_2 + 2x_3 + 3x_4 + 4x_5 \\
&\textbf{s.l.c}\ x_1 + x_3 \geq 1 \\
&\qquad x_1 + x_4 \geq 1 \\
&\qquad x_1 + x_5 \geq 1 \\
&\qquad x_2 + x_3 \geq 1 \\
&\qquad x_2 + x_4 \geq 1 \\
&\qquad x_2 + x_5 \geq 1 \\
&\qquad\\
&\qquad x_1,\ x_2,\ x_3,\ x_4,\ x_5 \in \{0,1\}
\end{aligned}$$

**Formuler l'IP du Vertex Cover**

Pour une instance $G = (V, E)$ :

$$\begin{aligned}
&\min \textstyle\sum_{v \in V}(x_v) \\
&\textbf{s.l.c}\ x_u + x_v \geq 1,\ \forall (u,v) \in E \\
&\qquad x_v \in \{0,1\},\ \forall v \in V
\end{aligned}$$

# Exercices d'algorithmes d'approximation.

# Chapitres 1 et 2.

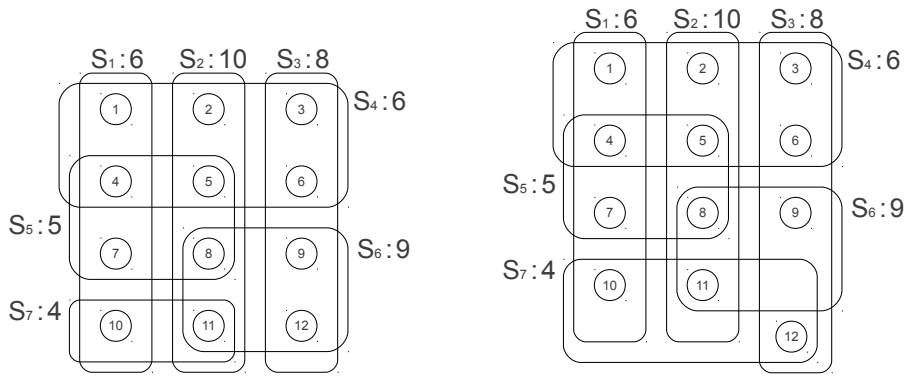
Absil Romain

6 avril 2011

**Exercice 1.** Soit $\mathcal{P}_1$ et $\mathcal{P}_2$ deux instances du problème SET-COVER, illustrées à la Figure 1.

1. Écrivez $\mathcal{P}_2$ sous la forme primale d'un problème d'optimisation linéaire en nombres entiers.
2. Écrivez la forme duale de la relaxation linéaire de ce problème.
3. Calculez la valeur du paramètre $f$ utilisé dans les algorithmes d'approximation DETERMINIST-ROUNDING-SC, DUAL-ROUNDING-SC et PRIMAL-DUAL-SC.
4. Utilisez l'arrondi déterministe DETERMINIST-ROUNDING-SC sur la solution de la relaxation linéaire du problème primal et vérifiez son facteur d'approximation. Calculez également le paramètre $\alpha$ afin de vérifier la garantie *a fortiori* de cet algorithme.
5. Utilisez la méthode d'arrondi déterministe DUAL-ROUNDING-SC sur la solution du problème dual pour résoudre ce problème et vérifiez son facteur d'approximation.
6. Utilisez l'algorithme primal-dual PRIMAL-DUAL-SC pour résoudre ce problème et vérifiez son facteur d'approximation.
7. Utilisez l'algorithme GREEDY-SC pour résoudre $\mathcal{P}_1$.
8. Calculez le paramètre $g$ pour cette méthode et vérifiez son facteur d'approximation.

Remarque : vous ne devez *pas* calculer vous-même les solutions des formulations primales et duales du problème. Elles vous seront données en séance d'exercices.




(a) $\mathcal{P}_1$ (b) $\mathcal{P}_2$

FIGURE 1 – Deux instances $\mathcal{P}_1$ et $\mathcal{P}_2$ du problème SET-COVER

*Solution.*

1. Formulation primale :

$$\min \; 6x_1 + 10x_2 + 8x_3 + 6x_4 + 5x_5 + 9x_6 + 4x_7$$

$$\begin{array}{lllll}
\text{s.c.} & v_1 & : & x_1 + x_4 & \geqslant 1 \\
& v_2 & : & x_2 + x_4 & \geqslant 1 \\
& v_3 & : & x_3 + x_4 & \geqslant 1 \\
& v_4 & : & x_1 + x_4 + x_5 & \geqslant 1 \\
& v_5 & : & x_2 + x_4 + x_5 & \geqslant 1 \\
( & v_6 & : & x_3 + x_4 & \geqslant 1 \quad ) \\
& v_7 & : & x_1 + x_5 & \geqslant 1 \\
& v_8 & : & x_2 + x_5 + x_6 & \geqslant 1 \\
& v_9 & : & x_3 + x_6 & \geqslant 1 \\
& v_{10} & : & x_1 + x_7 & \geqslant 1 \\
& v_{11} & : & x_2 + x_6 + x_7 & \geqslant 1 \\
& v_{12} & : & x_3 + x_7 & \geqslant 1 \\
& & & x_i \in \{0,1\} \quad \forall i = 1, \ldots, 7 &
\end{array}$$

2. Formulation duale :

$$\max \sum_{i=1}^{12} v_i$$

$$\begin{array}{lllll}
\text{s.c.} & s_1 & : & v_1 + v_4 + v_7 + v_{10} & \leqslant 6 \\
& s_2 & : & v_2 + v_5 + v_8 + v_{11} & \leqslant 10 \\
& s_3 & : & v_3 + v_6 + v_9 + v_{12} & \leqslant 8 \\
& s_4 & : & v_1 + v_2 + v_3 + v_4 + v_5 + v_6 & \leqslant 6 \\
& s_5 & : & v_4 + v_5 + v_7 + v_8 & \leqslant 5 \\
& s_6 & : & v_8 + v_9 + v_{11} & \leqslant 9 \\
& s_7 & : & v_{10} + v_{11} + v_{12} & \leqslant 4 \\
& & & v_i \geqslant 0 \quad \forall i = 1, \ldots, 12 &
\end{array}$$

3. Avant toute chose, on remarque que la solution optimale du problème a une valeur de 23 en sélectionnant les ensembles $S_3, S_4, S_5$ et $S_7$.

   Par ailleurs, pour les trois premiers algorithmes, on a

$$\begin{aligned} f &= \max_i f_i \\ &= \max_i \Big| \{ j : v_i \in S_j \} \Big| \end{aligned}$$

   Plus particulièrement, on a $f = 3$ (atteint sur les éléments $v_4$, $v_5$, $v_8$ et $v_{11}$).

   On pourra vérifier les résultats théoriques de ce facteur d'approximation sur les trois algorithmes.

4. Dans la méthode d'arrondi déterministe, on doit utiliser la règle d'arrondi suivante sur une solution $x^*$ :

$$x_i = \begin{cases} 1 & \text{si } x_i^* \geqslant \frac{1}{f}, \\ 0 & \text{sinon.} \end{cases}$$

   Comme la solution du problème primal relaxé vaut 22, avec

$$x^* = \left( \frac{1}{2}, 0, \frac{1}{2}, 1, \frac{1}{2}, \frac{1}{2}, \frac{1}{2} \right), \text{ [1]}$$

   à l'aide de cette règle, on obtient donc la solution $(1, 0, 1, 1, 1, 1, 1)$.

1. Vous ne devez pas chercher vous-même cette solution, elle a été donnée en séance d'exercices.

La garantie $\alpha$ *a fortiori* de cet algorithme est définie comme

$$\alpha = \frac{\sum_{i \in I} w_i}{z^*_{LP}}.$$

On a donc $\alpha = \frac{6+8+6+5+9+4}{22} = \frac{38}{22}$.

On remarque par ailleurs qu'on a $\frac{APP}{OPT} = 1.6521 < 1.7272 = \alpha < 3 = f$.

5. La règle d'arrondi dual est la suivante : *On sélectionne $j$ dans $I'$ si l'inégalité correspondante est serrée,* i.e., *si*

$$\sum_{i : e_i \in S_j} y^*_i = w_j.$$

Comme la solution du problème dual vaut 22, avec

$$v^* = (0, 6, 0, 0, 0, 0, 3, 2, 7, 3, 0, 1), \text{ [2]}$$

on sélectionne donc tous les ensembles à l'exception de $S_2$. Cette solution a une valeur d'objectif de $38 < f.OPT = 69$.

6. Pour l'algorithme PRIMAL-DUAL-SC, on notera $NC$ l'ensemble des sommets non couverts à l'itération courante. À chaque itération, on calcule les variables nécessaires à l'algorithme. Les notations utilisées sont identiques à celles du cours.
   – Itération 1.

$$NC = \left\{ \boxed{v_1}, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12} \right\}$$

$s_1 : \Delta_1 = 6$
$s_4 : \Delta_4 = 6$
$\hookrightarrow l = 1 \quad$ (ou 4)

$y = (6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$I = \{1\}$

2. Vous ne devez pas chercher vous-même cette solution, elle a été donnée en séance d'exercices.

– Itération 2.

$$NC = \left\{ \boxed{v_2}, v_3, v_5, v_6, v_8, v_9, v_{11}, v_{12} \right\}$$

$s_2 : \Delta_1 = 10$
$s_4 : \Delta_4 = 0$
$\hookrightarrow l = 4$

$y = (6, \underline{0}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$I = \{1, 4\}$

– Itération 3.

$$NC = \left\{ \boxed{v_8}, v_9, v_{11}, v_{12} \right\}$$

$s_2 : \Delta_2 = 10$
$s_5 : \Delta_5 = 5$
$s_6 : \Delta_6 = 9$
$\hookrightarrow l = 5$

$y = (6, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0)$
$I = \{1, 4, 5\}$

– Itération 4.

$$NC = \left\{ \boxed{v_9}, v_{11}, v_{12} \right\}$$

$s_3 : \Delta_3 = 8$
$s_6 : \Delta_6 = 4$
$\hookrightarrow l = 6$

$y = (6, 0, 0, 0, 0, 0, 0, 5, 4, 0, 0, 0)$
$I = \{1, 4, 5, 6\}$

– Itération 5.

$$NC = \left\{ \boxed{v_{12}} \right\}$$

$s_3 : \Delta_3 = 4$
$s_7 : \Delta_7 = 4$
$\hookrightarrow l = 3$

$y = (6, 0, 0, 0, 0, 0, 0, 5, 4, 0, 0, 4)$
$I = \{1, 3, 4, 5, 6\}$

Les ensembles sélectionnées sont donc $S_1, S_3, S_4, S_5$ et $S_6$, avec un objectif de valeur $34 < f.OPT = 69$.

7. À chaque itération, l'algorithme GREEDY-SC sélectionne l'ensemble ayant le meilleur rapport éléments non-couverts – coût de l'ensemble. Pour cet exercice, on notera $Q_i$ ce rapport pour chaque ensemble $S_i$. On a donc les résultats suivants :

| Itération | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $Q_1$ | $\frac{6}{4}$ | $\frac{6}{2}$ | $\frac{6}{1}$ | / |
| $Q_2$ | $\frac{10}{4}$ | $\frac{10}{2}$ | $\frac{10}{1}$ | / |
| $Q_3$ | $\frac{8}{4}$ | $\frac{8}{2}$ | $\frac{8}{2}$ | $\boxed{\frac{8}{2}}$ |
| $Q_4$ | $\boxed{\frac{6}{6}}$ | / | / | / |
| $Q_5$ | $\frac{5}{4}$ | $\frac{5}{2}$ | $\boxed{\frac{5}{2}}$ | / |
| $Q_6$ | $\frac{9}{4}$ | $\frac{9}{4}$ | $\frac{9}{3}$ | $\frac{9}{2}$ |
| $Q_7$ | $\frac{4}{2}$ | $\boxed{\frac{4}{2}}$ | / | / |
| Sélection | $S_4$ | $S_7$ | $S_5$ | $S_3$ |

L'algorithme sélectionne donc les ensembles $S_3, S_4, S_5$ et $S_7$, avec un objectif de valeur 23. La solution optimale a une valeur de 21 et est atteinte en sélectionnant $S_1, S_4$ et $S_6$.

8. L'algorithme GREEDY-SC est un algorithme de $H_g$-approximation, où $g = \max_i |S_i|$.

   Dans le cas de $\mathcal{P}_1$, on a $g = 6$, et donc GREEDY-SC est un algorithme de 2.45-approximation.

□

**Exercice 2.** Montrez que l'Algorithme 1 est un algorithme de 2-approximation pour le problème VERTEX-COVER.

*Solution.*
Cette heuristique est clairement polynomiale. En notant $C^*$ une couverture optimale, montrons que $|C| \leqslant 2\ |C^*|$.

L'ensemble $C$ des sommets sélectionnés est bel et bien une couverture, car l'algorithme s'arrête quand toute arête de $G$ a été couverte par un sommet dans $C$.

**Algorithm 1** Algorithme APPROX-VERTEX-COVER

**Entrées :** $G = (V, E)$ un graphe non orienté.
**Sortie :** $C$ : une couverture des arêtes par les sommets de $G$.

1: $C \leftarrow \emptyset$
2: $E' \leftarrow E$
3: **tant que** $E' \neq \emptyset$
4: Soit $(u, v)$ une arête arbitraire de $E'$
5: $C \leftarrow C \cup \{u, v\}$
6: Supprimer de $E'$ toutes les arêtes incidentes à $u$ ou $v$
7: **retourner** C

Afin de montrer que APPROX-VERTEX-COVER calcule une couverture de taille au plus deux fois la taille d'une couverture optimale, notons $A$ l'ensemble des arêtes sélectionnées à la ligne 4 de cet algorithme. On remarque que toute couverture – et en particulier une couverture optimale $C^*$ – doit inclure une des extrémités de chaque arête de $A$. De plus, il n'existe pas d'arêtes de $A$ ayant une extrémité commune, car lorsqu'une arête est sélectionnée à la ligne 4, toutes les arêtes incidentes sont supprimées de $E'$ à la ligne 6.

Dès lors, deux arêtes de $A$ ne sont jamais couvertes par le même sommet de $C^*$, et on a donc $|C^*| \geqslant |A|$. Par ailleurs, chaque exécution de la ligne 4 sélectionne une arête dont aucune des extrémités ne se trouve dans $C$, et on obtient donc $|C| = 2\,|A|$. En combinant ces deux équations, on obtient donc

$$\begin{aligned} |C| &= 2\,|A| \\ &\leqslant 2\,|C^*| \end{aligned}$$

□

**Exercice 3.** Un voleur veut cambrioler une maison, muni d'un sac à dos de taille $K$. Dans la maison se trouve une série de $n$ objets qu'il souhaite dérober. Chaque objet $e_i$ a une valeur $v_i$ et une taille $s_i$.
Le but du voleur est donc de remplir son sac à dos avec le plus d'objets possibles afin de maximiser son profit, en sachant qu'il ne peut pas remplir son sac au delà de sa capacité.
En optimisation combinatoire, on appelle ce problème le *problème de sac à dos binaire* (0-1-Knapsack Problem). Si le voleur a le droit de sélectionner autant de fois qu'il le désire un même objet, on appellera ce problème le *problème de sac à dos entier* (Integer-Knapsack Problem).

1. Modélisez le problème de sac à dos binaire sous la forme d'un problème d'optimisation en nombres entiers,
2. Décrivez une heuristique greedy pour ce problème,
3. Montrez que cette heuristique est un algorithme de 2-approximation (ou modifiez-là pour qu'elle le soit),

4. Décrivez une heuristique greedy pour le problème de sac à dos entier et montrez que c'est un algorithme de 2-approximation.

*Solution.*

1. On peut modéliser le problème de sac à dos binaire de la façon suivante :

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} v_i \; x_i \\ s.c. \quad & \sum_{i=1}^{n} s_i \; x_i < K \\ & x_i \in \{0,1\} \; \forall i = 1, \ldots, n \end{aligned}$$

   Les variables $x_i$ décrivent si l'on a sélectionné l'objet $e_i$ ou non.

2. Une heuristique Greedy va itérativement sélectionner les objets de profit maximal. Le profit d'un objet $i$ peut être décrit par $\frac{v_i}{s_i}$. On peut donc écrire l'heuristique suivante pour ce problème :

**Algorithm 2** Algorithme Greedy

**Entrées :** un problème de sac à dos binaire.
**Sortie :** $z^*_{GR}$, solution greedy pour le problème.

1: Trier et renuméroter les objets $i$ par ordre de profit $\frac{v_i}{s_i}$ décroissant.
2: Ajouter dans l'ordre les objets $i$ dans le sac tant que la capacité du sac n'est pas dépassée.
3: **retourner** le contenu du sac à dos.

3. L'heuristique `Greedy` n'est pas un algorithme d'approximation, et la solution peut être aussi mauvaise que possible, comme illustré par l'exemple suivant :
   - Un objet de taille 1 et de valeur 2,
   - Un objet de taille $K$ et de valeur $K$.

   L'algorithme `Greedy` va uniquement sélectionner le premier objet, et en faire une approximation de mauvaise qualité. Afin d'éviter ce problème et de transformer cette heuristique en algorithme d'approximation, il est nécessaire d'y apporter la modification suivante : soit $k$ le dernier objet ajouté dans le sac, il suffit de retourner $\max \left\{ \sum_{i=1}^{k} v_i, v_{k+1} \right\}$.

Montrons qu'à présent cette heuristique est bien un algorithme de 2-approximation, *i.e.,* l'algorithme est polynomial, et la solution retournée vaut au pire la moitié de la solution optimale.

Clairement, cet algorithme a une complexité dans le pire des cas en $\mathcal{O}(n \log n)$. En notant $z^*$ la solution optimale et $z^*_{GR}$ la solution de l'algorithme, on doit donc montrer que $z^* \leqslant 2\ z^*_{GR}$.

Avec cette heuristique, si la solution n'est pas optimale il reste de la place dans le sac. Précisément, il reste un espace de taille $K - \sum_{i=1}^{k} v_i$. Dès lors, on pourrait améliorer la solution en ajoutant des morceaux de l'objet $k+1$ (cette solution n'est évidemment pas admissible).

On a donc

$$\begin{aligned} z^* &\leqslant \sum_{i=1}^{k} v_i + \left(K - \sum_{i=1}^{k} v_i\right) \cdot \frac{v_{k+1}}{s_{k+1}} \\ &< \sum_{i=1}^{k} v_i + v_{k+1} \end{aligned}$$

Si $\sum_{i=1}^{k} v_i \geqslant v_{k+1}$, alors $z^* \leqslant 2 \sum_{i=1}^{k} v_i = 2\ z^*_{GR}$.

Sinon si $\sum_{i=1}^{k} v_i < v_{k+1}$, alors $z^* \leqslant 2\ p_{k+1} = 2\ z^*_{GR}$.

4. L'heuristique greedy pour ce problème procède de manière similaire à l'heuristique du point 2 : elle trie dans un premier temps les objets par profit $\frac{v_i}{s_i}$ décroissants, et sélectionne ensuite le premier objet $\left\lfloor \frac{K}{s_1} \right\rfloor$ fois.

   Montrons à présent que c'est un algorithme de 2-approximation, *i.e.,* que cette heuristique est polynomiale (trivial) et que la solution retournée vaut au pire la moitié de la solution optimale.

   On a $z^*_{GR} = v_1 \left\lfloor \frac{K}{s_1} \right\rfloor$. Notons $z_{LP}$ la solution de la relaxation linéaire. On a :

$$\begin{aligned} z^* &\leqslant z_{LP} && \text{par définition de relaxation} \\ &= v_1 \tfrac{K}{s_1} \\ &= v_1 \left( \left\lfloor \tfrac{K}{s_1} \right\rfloor + f \right) && \text{pour un certain } 0 \leqslant f < 1 \\ &\leqslant 2 v_1 \left\lfloor \tfrac{K}{s_1} \right\rfloor \end{aligned}$$

On a donc $\frac{z^*_{GR}}{z^*} \geqslant \frac{z^*_{GR}}{z_{LP}} \geqslant \frac{1}{2}$.

□

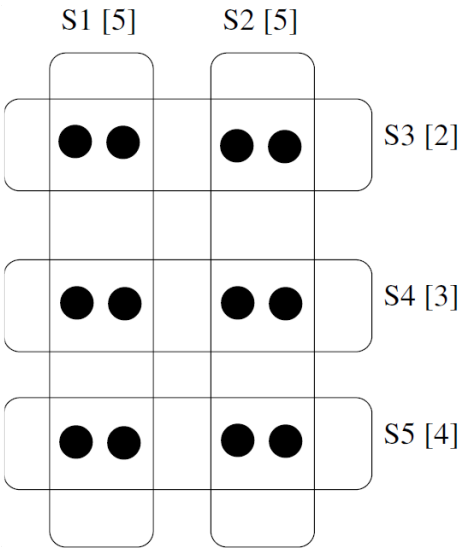



FIGURE 41 – Exemple d'instance de SC

# C Annexe C : Exercices chapitre 3

**Appliquer EDD_SSM à l'instance suivante**

$(p_j, r_j, d_j) =$

1. $(2, 0, -12)$
2. $(1, 2, -10)$
3. $(4, 1, -1)$

L'algorithme donne la solution optimale *(ordonnancement* **ABC** *mais c'est un coup de chance)*.

# D Annexe D : Exercices chapitre 4

(cf pdf inclus à la page suivante)

# Exercices d'algorithmes d'approximation. Programmation dynamique.

Absil Romain

18 mai 2011

## Table des matières



## 1 Introduction.

La programmation dynamique est une technique classique dans le design d'algorithmes d'optimisation. Le principe de base est de construire la solution optimale d'un problème à partir de solutions optimales de sous-problèmes, le plus souvent stockées dans des tableaux, matrices ou autres structures de données multidimensionnelles.

Quand elle est applicable et utilisée judicieusement, cette méthode offre souvent de meilleurs résultats qu'une approche naïve. En effet, lors de la

décomposition d'un problème en sous-problèmes, on peut être amené à résoudre plusieurs fois un même sous-problème. La clé de ce principe est donc de résoudre chaque sous-problème exactement une fois.

Cette technique va donc réduire considérablement le temps de calcul, plus particulièrement quand le nombre de "sous-problèmes répétés" est arbitrairement grand.

La première fois introduite par Bellmann [2], cette méthode est à présent largement utilisé en informatique, notamment en

- bioinformatique : algorithmes de Needleman-Wunsch et de Smith-Waterman d'alignement maximal de deux séquences de nucléotides,
- théorie des graphes : algorithme de Floyd de calcul de tous les plus courts chemins,
- infographie : algorithme de Boore d'évaluation des B-Splines,
- analyse numérique : calcul des moindres carrés récursifs, etc.

## 2 Exemple : la suite de Fibonacci.

Le calcul de la suite de Fibonacci $F(n)$, définie ci-dessous, illustre un bel exemple de l'utilité de la programmation dynamique.

$$F(n) = \begin{cases} 1 & \text{si } n \in \{1, 2\}, \\ F(n-1) + F(n-2) & \text{sinon.} \end{cases}$$

Il est bien connu qu'implémenter une récursion naïve pour calculer la suite de Fibonacci est peu efficace, dans la mesure où on calcule plusieurs fois un même résultat, comme illustré par la Figure 1. On peut même montrer que cette implémentation a une complexité en $\mathcal{O}(\varphi^n)$.

Supposons à présent que l'on dispose d'une structure de donnée $M$ capable d'associer à chaque valeur $k$ une valeur $v$, telle qu'un tableau. On notera $M(k)$ la valeur $v$ associée à $k$ dans $M$. On peut dès lors modifier l'algorithme naïf afin d'utiliser cette structure et la mettre à jour pour diminuer drastiquement la complexité des calculs. On obtient donc l'algorithme de programmation dynamique ci-dessous. Cet algorithme a une complexité en temps dans le pire des cas linéaire.

Notons que la programmation dynamique est une méthode exacte, et n'a donc à priori pas de lien direct avec les algorithmes d'approximation.

Toutefois, on peut concevoir des algorithmes d'approximation en arrondissant les données *en entrée* du problème (c'est d'ailleurs ce qui est généra-

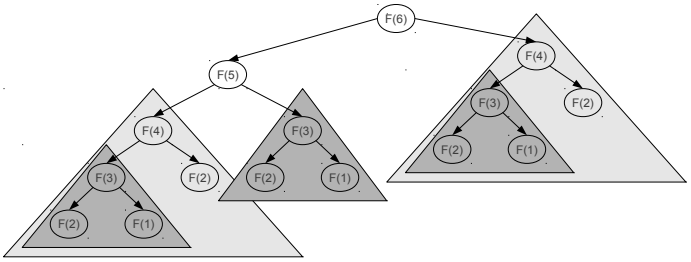



FIGURE 1 – Illustration de la redondance des calculs lors de l'évaluation de la suite de Fibonacci.

**Algorithm 1** Algorithme Fib(n)

1: $M(1) = M(2) = 1$
2: **retourner** $F(n)$

3: **Fonction** F(n)
4: **si** $M(n)$ n'est pas défini
5: $\quad M(n) = F(n-1) + F(n-2)$
6: **retourner** $M(n)$

lement effectué dans le cas de tels algorithmes).

Cette technique se distingue donc des autres méthodes d'approximation car un arrondi est effectué avant l'exécution d'un quelconque algorithme, sur son entrée, et non pas après son exécution, sur sa sortie.

# 3 Éléments de programmation dynamique.

Bien que l'exemple ci-dessus illustre l'utilité de la programmation dynamique pour améliorer l'efficacité de la résolution de certains problèmes, il est intéressant de caractériser la structure des problèmes pour lesquels cette technique est toute indiquée.

Dans le cas présent, on s'intéressera plus particulièrement aux problèmes d'optimisation, qui peuvent être résolus rapidement grâce à des algorithmes d'approximation.

Concrètement, ces problèmes doivent avoir les propriétés suivantes :
– Sous-structure d'optimalité,
– Recouvrement des sous-problèmes,
– Memoisation [1].

## 3.1 Sous-structure d'optimalité.

Souvent, la première étape dans la résolution d'un problème d'optimisation est la caractérisation de la structure d'une solution optimale.

Plus particulièrement, un problème a une *sous-structure optimale* si une solution optimale du problème « contient » des solutions optimales de sous-problèmes.

Quand un problème partage cette propriété, il peut être intéressant de le résoudre par programmation dynamique. Dans ce cas, on construira la solution optimale d'un problème à partir de solutions optimales de sous-problèmes, par une approche *bottom up*.

L'exemple ci-dessous illustre la sous-structure d'optimalité du problème de plus court chemin dans un graphe non-orienté. Toutefois, il faut parfois se montrer prudent avec cette propriété, et ne pas assumer qu'elle s'applique là où elle n'est pas préservée, comme illustré ci-après.

Considérons les problèmes suivants :
– Recherche du plus court chemin élémentaire entre deux sommets,
– Recherche du plus long chemin élémentaire entre deux sommets.

Le problème de recherche de plus court chemin exhibe une sous-structure d'optimalité. Soit un plus court chemin $p$ non trivial de $u$ à $v$, et $w$ un sommet sur ce chemin. On peut décomposer le chemin $u \overset{p}{\rightsquigarrow} v$ en sous-chemins $u \overset{p_1}{\rightsquigarrow} w$ et $w \overset{p_2}{\rightsquigarrow} v$.

Clairement, le nombre d'arêtes dans $p$ est égal à la somme du nombre d'arêtes dans $p_1$ et dans $p_2$. De plus, si $p$ est optimal (*i.e.,* de longueur minimum), alors $p_1$ et $p_2$ sont optimaux [2].

On pourrait donc assumer que le problème de recherche de plus long chemin entre deux sommets $u$ et $v$ a une sous-structure d'optimalité. Toutefois, ce n'est pas le cas, comme illustré par l'exemple de la Figure 2.

1. Ceci n'est pas une faute d'orthographe. Le terme est bien mémoisation et non mémorisation. Mémoisation vient de *memo*, cette technique consistant à enregistrer une valeur destinée à être consultée ultérieurement.
2. La preuve de cette propriété simple est laissée en exercice.

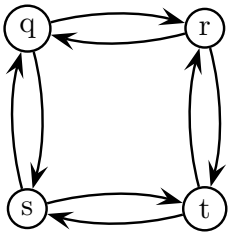



FIGURE 2 – Contre-exemple de la sous-structure d'optimalité pour le problème de plus long chemin.

Sur cet exemple, on voit que $q \longrightarrow r \longrightarrow t$ est un plus long chemin, de $q$ à $t$. Toutefois, ni $q \longrightarrow r$, ni $r \longrightarrow t$ ne sont des plus longs chemins de $q$ à $r$ ou de $r$ à $t$.

En effet, les chemins $q \longrightarrow s \longrightarrow t \longrightarrow r$ et $r \longrightarrow q \longrightarrow t \longrightarrow t$ sont des chemins de $q$ à $r$ et de $r$ à $t$ de longueur supérieurs.

Par ailleurs, il est impossible de construire un plus long chemin à partir de solutions de sous-problèmes. En effet, si l'on construit un plus long chemin de $q$ à $t$ à partir des chemins $q \longrightarrow s \longrightarrow t \longrightarrow r$ et $r \longrightarrow q \longrightarrow t \longrightarrow t$, on obtient le chemin $q \longrightarrow s \longrightarrow t \longrightarrow r \longrightarrow q \longrightarrow t \longrightarrow t$, qui n'est pas élémentaire.

Le problème du plus long plus court chemin ne partage donc pas la propriété de sous-structure d'optimalité. Rivest *et al.* [1] affirment même qu'aucun algorithme de programmation dynamique n'a jamais été trouvé pour ce problème.

### 3.2 Recouvrement des sous-problèmes.

Une seconde caractéristique que les problèmes devraient partager afin que la programmation dynamique soit applicable et efficace est la propriété de *recouvrement des sous-problèmes*.

Intuitivement, cela signifie d'un algorithme récursif pour un problème va résoudre de nombreuses fois des sous-problèmes identiques, plutôt que de générer des nouveaux sous-problèmes.

Typiquement, les algorithmes de programmation dynamique tirent avantage de cette propriété en résolvant chaque problème une unique fois et en stockant sa solution dans une table où elle pourra être consultée ou mise à jour rapidement.

On remarque que l'évaluation de la suite de Fibonacci illustrée à la Figure 1 partage cette propriété.

En effet, on remarque que la hauteur de l'arbre binaire d'évaluation de $F(n)$ est en $\mathcal{O}(n)$. Dès lors, cet arbre a un nombre $k$ de nœuds avec $k$ en $\mathcal{O}(2^n)$. Or, l'espace des sous-problèmes a une taille valant exactement $n$ (il y a exactement $n$ problèmes distincts à résoudre), on a donc un facteur de recouvrement de l'ordre de $2^n/n$.

En revanche, l'algorithme d'Euclide de calcul du pgcd illustré ci-dessous ne partage pas la propriété de recouvrement des sous-problèmes.

**Algorithm 2** Algorithme pgcd(a,b)

```
1: si b = 0
2:     retourner a
3: sinon
4:     retourner pgcd(b, a mod b)
```

En effet, tous les sous-problèmes générés par la procédure récursive sont indépendants, et l'on n'est jamais amené à résoudre plusieurs fois un même problème. Utiliser la programmation dynamique dans ce cas est donc inutile.

En considérant l'exemple plus général de la résolution d'un problème linéaire en nombre entiers, la taille de l'espace des sous-problèmes générés par une procédure de Branch and Bound ne peut pas être bornée (car ces problèmes sont tous indépendants). Dès lors, la programmation dynamique n'est pas applicable pour cet exemple.

### 3.3 Memoisation.

Ce principe a déjà été préalablement débattu. Étant donné qu'un algorithme de programmation dynamique résout chaque sous-problème du problème initial exactement une fois, il est nécessaire de conserver les solutions de ces sous-problèmes, dans le cas où ils seraient amenés à être résolus à nouveau.

L'idée de base est de *memoiser* l'algorithme récursif naturel (mais inefficace). D'ordinaire, on maintient une table avec les solutions des sous-problèmes, la structure de contrôle permettant de remplir cette table étant l'algorithme récursif.

Selon le problème, on peut utiliser des tableaux (unidimensionnels ou non), des arbres ou des tables de hachage dans le but de stocker et/ou mettre à jour les valeurs de solutions optimales de sous problèmes. Dans tous les

cas, l'espace mémoire nécessaire à la mémoisation doit être raisonnable (*i.e.,* borné par un polynôme de degré faible).

Dans un algorithme memoisé, la table est initialement vide, ou chacune de ses entrées contient une valeur symbolique dans le cas où le nombre d'entrées maximal est connu. Quand un sous-problème est rencontré la première fois dans l'exécution de l'algorithme récursif, sa solution est calculée et stockée dans la table. À chaque fois que ce sous-problème est rencontré à nouveau, la valeur stockée dans la table est simplement retournée.

L'Algorithme 3 d'évaluation de la suite de Fibonacci est un exemple d'algorithme memoisé. Dans ce cas, on peut utiliser un simple tableau $M$ de taille $n$ initialement rempli de zéros. On considère alors qu'un sous-problème $F(k)$ n'a pas encore été rencontré si $M(k) = 0$, *i.e.,* si la k$^{\text{ième}}$ case du tableau contient un zéro.

Par ailleurs, on remarque que l'espace mémoire nécessaire à la mémoisation n'est pas plus grand que celui occupé par la pile d'exécution de l'algorithme récursif naïf.

Pour reprendre l'exemple de l'exécution d'un Branch and Bound, l'espace mémoire occupé par une structure de donnée si l'on décidait de mémoiser cet algorithme ne pourrait pas être borné. Pour cette raison encore, la programmation dynamique n'est pas applicable pour cet exemple.

# 4 Exercices.

**Exercice 1.** Soit l'instance de problème de sac à dos à quatre objets suivante, définie comme dans le cours et le livre support :

$$\begin{array}{ll} v_1 = 3217 & s_1 = 2 \\ v_2 = 4000 & s_2 = 3 \\ v_3 = 2108 & s_3 = 2 \\ v_4 = 1607 & s_4 = 7 \end{array}$$

1. Si $\mu = 100$ dans l'algorithme FPAS-KP, quel est le facteur d'approximation de cet algorithme sur cette instance ?
2. Si l'on veut utiliser l'algorithme FPAS-KP sur cette instance avec une erreur de maximum 1%, sur quelle instance « arrondie » va-t-on lancer l'algorithme PROG-DYN-KP ?

*Solution.*

1. Rapellons que $\mu = \frac{\varepsilon M}{n}$, or on a $M = 4000$ et $n = 4$, on a donc $\varepsilon = \frac{1}{10}$. Cet algorithme a donc un facteur d'approximation de $1 - \varepsilon = 0.9$. On accepte donc une erreur de l'ordre de 10%.
2. On a $\mu = \frac{1}{100} \cdot \frac{4000}{4} = 10$. On aura donc :

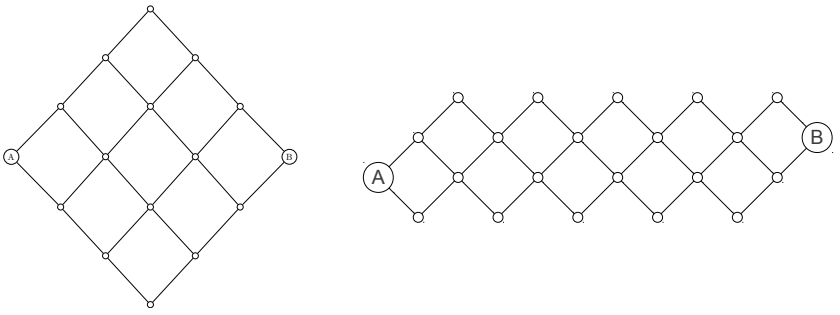
$$\begin{array}{ll} v_1 = 321 & s_1 = 2 \\ v_2 = 400 & s_2 = 3 \\ v_3 = 210 & s_3 = 2 \\ v_4 = 160 & s_4 = 7 \end{array}$$

□

**Exercice 2.** Soient les deux graphes $G$ et $H$ de la Figure 3. Il est demandé de calculer le nombre de plus courts chemins permettant d'aller de $A$ à $B$.

1. Concevez un algorithme récursif naïf pour résoudre ce problème.
2. Exhibez la sous-structure d'optimalité de ces problèmes.
3. Illustrez le recouvrement des sous-problèmes.
4. Transformez votre algorithme récursif naïf en algorithme de programmation dynamique en utilisant la mémoisation.

Notez que vos algorithmes doivent fonctionner sur des graphes de taille plus grande que les exemples illustrés. La grille du premier exemple et le serpentin du deuxième peuvent donc avoir une taille $n$ arbitraire.




(a) $G$ (b) $H$

FIGURE 3 – Deux graphes $G$ et $H$. Calculez le nombre de plus courts chemins allant de $A$ à $B$.

*Solution.*
Soit $v$ un sommet du graphe $G$ ou $H$, on note
- $up_v$ le prédécesseur supérieur de $v$,
- $down_v$ le prédécesseur inférieur de $v$,
- $v_{up}$ le successeur supérieur de $v$,
- $v_{down}$ le successeur inférieur de $v$,
- $N(v)$ le nombre de plus courts chemins allant de $A$ à $v$.

Notez que la notion prédécesseur / successeur est définie en considérant les graphes orientés de gauche à droite (*i.e.,* , on oriente une arête $(i, j)$ de $i$ vers $j$ si $i$ est à gauche de $j$.). Procéder de cette manière ne change pas le nombre de plus courts chemins de $A$ vers $B$.[3]

Le calcul de $N(v)$ est donc défini (dans les deux exemples) récursivement comme ci-dessous :

$$N(v) = \begin{cases} 1 & \text{si } v = A, \\ N(down_v) & \text{si } up_v \text{ n'est pas défini,} \\ N(up_v) & \text{si } down_v \text{ n'est pas défini,} \\ N(down_v) + N(up_v) & \text{sinon.} \end{cases}$$

On peut montrer la sous-structure d'optimalité de la façon suivante : soient $u$ et $v$ tels que $u_{up} = v_{down} = w$. Clairement, pour atteindre $w$ à partir de $A$, il est nécessaire de passer par $u$ ou $v$. De plus, si $p = A \rightsquigarrow u \longrightarrow w$ est un plus court chemin de $A$ à $w$, alors $A \rightsquigarrow u$ est un plus court chemin de $A$ à $u$. En considérant le même argument pour $v$, le nombre de plus courts chemins de $A$ à $w$ est donc égal à la somme du nombre de plus courts chemins de $A$ à $u$ et de $A$ à $v$.

On peut particulariser cet argument pour les sommets $v$ aux extrémités de la grille pour lesquels soit $v_{up}$ ou $v_{down}$ n'est pas défini.

Montrons à présent pourquoi l'algorithme récursif est inefficace, et qu'il est amené à calculer plusieurs fois le même résultat, *i.e.,* exhibons le recouvrement des sous-problèmes. En pivotant légèrement le graphe de la Figure 3(a) pour numéroter les sommets (sauf $A$ et $B$) de gauche à droite et de haut en bas, on obtient l'arbre d'appels récursifs de calcul de $N(B)$ de la Figure 4. On remarque notamment que l'on calcule plusieurs fois les sous-arbres de racine $N(10)$, $N(6)$, etc.

Afin de mémoiser l'algorithme récursif, il est nécessaire d'utiliser une structure de données capable d'enregistrer les valeurs de $N(v)$ pour chaque

3. En réalité, il existe de nombreuses autres paires de sommets $u$ et $v$ pour lesquels considérer ces graphes orientés comme tel ne change pas le nombre de plus courts chemins de $u$ vers $v$. Voyez-vous lesquels, et pourquoi ?

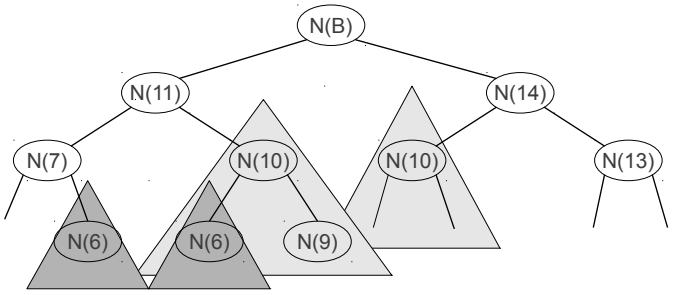



FIGURE 4 – Illustration du recouvrement des sous-problèmes du problème du nombre de plus courts chemins.

sommet $v$. Étant donné que le nombre de sommets du graphe est fixé avec le problème, on pourrait utiliser un simple tableau indexé judicieusement. Afin de simplifier les notations, on se contentera d'une simple table de hachage $T$ capable d'associer à un sommet $v$ la valeur $N(v)$.

L'algorithme mémoisé de programmation dynamique peut dès lors être conçu de la façon suivante :

**Algorithm 3** Algorithme Count-Paths(n)

1: $T(A) = 1$
2: **retourner** Count($B$)

3: **Fonction** Count($v$)
4: **si** $T(v)$ n'est pas défini
5: **si** $up_v$ n'est pas défini
6: $T(v) =$ Count($down_v$)
7: **sinon si** $down_v$ n'est pas défini
8: $T(v) =$ Count($up_v$)
9: **sinon**
10: $T(v) =$ Count($up_v$) + Count($down_v$)
11: **retourner** $T(v)$

□