# MMAD2 : MODELES ET METHODES POUR L'AIDE A LA DECISION 2

*Suite du cours MMAD1 - Durée : 20h*

*Problèmes d'optimisation combinatoire difficiles*

*Modélisation – Méthodes de résolution*

*2 parties*

*I) C. Briand 10h / Programmation Linéaire en Nombres Entiers*

*II) C. Artigues 10h : Autres méthodes*

*1. Introduction*

*2. Programmation dynamique*

*3. Recherche arborescente*

*3. Heuristiques et Métaheuristiques*

*4. Programmation par contraintes*

## 1 INTRODUCTION

### 1.1 Quels sont les problèmes auxquels on va s'intéresser ?

- **Problème d'Optimisation Combinatoire (POC)**

Chercher le minimum d'une fonction $f$ à variables prenant leurs valeurs sur un ensemble discret $S$. Soit $x$ est un vecteur de telles variables de décision. On note $x^*$ le minimum cherché.

$$f(x^*) = \min_{x \in S}(f(x))$$

$f$ est appelée fonction objectif, fonction économique ou critère.

On peut ajouter un ensemble de contraintes $C$ pour limiter $S$. Dans ce cas une solution réalisable est une solution qui respecte ces contraintes. Le POC revient alors à rechercher $x^*$, solution réalisable de coût minimal.

Pour des problèmes de maximisation, on se ramène à une minimisation ( $\max(g) = \min(-g)$ ).

- **Problème d'Existence (PE)**

On cherche dans un ensemble $S$ s'il existe une solution $x$ vérifiant les contraintes $C$ (solution réalisable). Variante : problème de décision : savoir si oui ou non un PE a une solution.

- **Lien entre les deux types de problèmes**

- POC : déterminer $x^*$ tel que $f(x^*) = \min_{x \in S}(f(x))$ [ et $C$ satisfait]

- pour tout $k$, définir PE($k$) : chercher s'il existe $x$ tel que $x \in S$ et $f(x) \leq k$ [ et $C$ satisfait]

Résoudre le POC revient à chercher le plus petit $k$ tel que PE($k$) soit réalisable

*Conséquences* :

Sil existe un algorithme (efficace) pour résoudre PE alors il existe un algorithme (efficace) pour résoudre POC. Si PE est difficile à résoudre alors POC est au moins aussi difficile à résoudre.

## 1.2 Notions de complexité ➔ cf. MMAD1

Complexité : mesure de l'efficacité d'un algorithme (en temps ou espace mémoire)

- Complexité dans le pire des cas.
- Complexité en moyenne.

➔ Complexité temporelle : évaluation du temps de calcul en fonction de la taille des données.

Exemple de mesure : $O(n)$, $O(n^2)$, $O(\log n)$, $O(2^n)$, $O(n!)$, etc.

Les algorithmes efficaces sont de complexité polynomiale. Les autres sont dits exponentiels.

- **Problèmes faciles et difficiles**

En théorie, tout POC peut être résolu par :

- énumération complète des solutions de l'espace $S$
- retenir la meilleure solution par rapport au critère

➔ Limite : taille de l'espace $S$

Pour certains POC : il existe des algorithmes polynomiaux. On parle de POC faciles.

Pour les autres : POC difficiles.

- **Exemples de problèmes faciles**

*1. Recherche d'une information parmi N (PE)*

- Recherche séquentielle dans un tableau $O(n)$
- Recherche dichotomique si tableau trié $O(\log n)$
- Recherche dans un arbre binaire de recherche $O(\log n)$

➔ Lien entre structure de données utilisées et complexité …..

*2. Quelques Problèmes vus dans le cours de Graphes*

- Rechercher des chemins de coût minimum dans un graphe orienté et valué
- Rechercher un flot maximum dans un graphe
- Rechercher un arbre couvrant de coût minimal dans un graphe non orienté et valué

*3. Programmation Linéaire*

$\min(Cx)$ / $Ax \leq b$ et $x \geq 0$ avec n variables, m contraintes

➔ Algorithme du Simplexe

On se ramène à un problème combinatoire (énumération d'un nombre fini de solutions de base).

- **Exemples de problèmes difficiles**

*1. Parcours Hamiltonien dans un graphe minimal ou non (PE ou POC)*

Exemple : problème du voyageur de commerce.

*2. Problème de satisfiabilité (SAT) (PE)*

$n$ variables booléennes $x_i$

$m$ clauses $C_j$ = disjonction de variables (Exemple $C_j = x1 \vee x3 \vee \neg x7$)

SAT : peut-on trouver une valeur pour chaque variable $x_i$ telle que toutes les clauses soient satisfaites (vraies) ($C_1 \wedge ..... \wedge C_m$.= vraie).

*3. Problème du sac à dos (knapsack) (POC)*

$n$ objets ayant chacun un poids $a_i$ et une valeur $c_i$.

Déterminer l'ensemble des objets que l'on peut emporter de manière à maximiser la valeur totale et sans dépasser un poids total $b$.

Max $\sum_{i=1,\ldots,n} c_i x_i$

s.c.

$\sum_{i=1,\ldots,n} a_i x_i \leq b$

$x_i \in \{0,1\}$, $i=1,\ldots,n$

*4. Problème de « Bin Packing » (POC)*

$n$ objets ayant chacun un poids $a_i$ et des boites de capacité $b$.

Répartir les objets dans un nombre minimal de boites

Problème de conditionnement, problème de chargement de véhicule, etc.

## 1.3 Résolution de problèmes difficiles

Objectif méthode de résolution :

- 1 (toutes) solution(s) PE
- 1 (toutes) solution(s) optimale(s) POC
- 1 (toutes) solution(s) approchées / optimum POC

Différentes classifications :

- exactes / approchées (résultat de la méthode / POC)
- déterministes / stochastiques (principe de la méthode)
- recherche locale / recherche globale (principe de la méthode)

- **Méthodes exactes**

Méthodes fournissant la (les) solution(s) optimale(s) d'un POC en prouvant son optimalité ou bien trouvant la (les) solution(s) d'un PE si elle(s) existe(nt) et prouvant sinon l'absence de solution.

➔ Elles sont basées sur une énumération intelligente (implicite) par **recherche arborescente** ou **programmation dynamique** de l'espace de recherche.

Ces méthodes ont une complexité exponentielle dans le pire des cas mais parfois performantes en moyenne ou bien pseudo-polynomiales. Elles sont en général utilisables pour des problèmes de petite taille. Les méthodes de ce type peuvent fournir des solutions intermédiaires pour les POC. Tronquées (par un temps ou nombre d'étapes limite d'exécution), elles deviennent ainsi des méthodes approchées.

- **Méthodes approchées ou méthodes heuristiques**

Méthodes permettant de trouver une solution de qualité correcte à un POC (sans forcément de garantie d'optimalité)

*- Méthodes gloutonnes (greedy search) ou constructives*

Construire une solution à partir de choix partiels et définitifs.

A chaque étape de résolution : faire un « bon » choix et ce choix ne sera pas remis en cause (pas de retour-arrière).

*- Méthodes locales (local search) ou par voisinage (neighborhood search)*

Partir d'une solution initiale ayant un coût donné et appliquer des transformations pour construire des solutions dont le coût est meilleur.

Arrêt : plus d'amélioration de la solution courante à partir des transformations.

➔ **méthodes dites de « descente »**

Les méthodes de descente peuvent être piégées dans des minima locaux.

Pour en sortir : construire une solution initiale et une suite de solutions voisines pour lesquelles le coût peut augmenter par rapport à la solution initiale.

➔ **méthodes dites « méta-heuristiques »**

Appliquer des principes généraux (indépendants du problème) pour ne pas être piégé dans un optimum local. Ces mécanismes peuvent ou non être basé sur le hasard (on parle alors de méthodes stochastiques par opposition aux méthodes déterministes). On trouve généralement des heuristiques dépendantes du problème comme composants élémentaires d'une méta-heuristique.

- descentes stochastiques avec redémarrages
- méthode à voisinages variables
- recuit simulé
- méthode avec tabous
- algorithmes génétiques ou évolutionnaires
- algorithmes basés sur les colonies de fourmi

- **Programmation par contraintes**

## I.1) Introduction

La programmation dynamique est un principe général applicable à de nombreux problèmes d'algorithmique et d'optimisation. A la base de la programmation dynamique, se trouve le *principe d'optimalité*, énoncé par R. Bellman (1957)

## I.2) Programmation dynamique en optimisation

### *problème d'optimisation*

La recherche opérationnelle utilise la programmation dynamique pour résoudre des problèmes d'optimisation du type

$$P : \underset{x \in X}{\text{Minimiser}} f(x)$$

où x est un vecteur $(x_1, x_2, \ldots, x_n)$ de variables de décision réelles et X est l'ensemble des solutions réalisables, représenté généralement par des contraintes.

Considérons par exemple le problème du sac à dos :

$$KP : \text{Maximiser } \Sigma_{i=1,\ldots,n} V_i x_i$$
$$\Sigma_{i=1,\ldots,n} T_i x_i \leq T$$
$$x_i \in \{0,1\} \qquad \forall I=1,\ldots,n$$

### *décomposition en sous-problèmes définis par les étapes de décision*

Pour résoudre P par programmation dynamique, il faut essayer de le décomposer en une suite de problèmes $P^0, P^1, \ldots, P^p$ tels que

- $P^0$ se calcule « facilement » et $P^n$ est équivalent à P
- $P^k$ se résout « facilement » à partir de la solution des $P^q$, $k>0$ et $q<k$
- $P^p$ est équivalent à P

Pour effectuer cette décomposition, il faut que le processus de décision menant à une solution soit décomposable en étapes.

Par exemple pour le sac-à-dos :

$1^{ère}$ étape : décider si on prend où non l'objet 1 (affecter la variable $x_1$)
$2^{ème}$ étape : décider si on prend où non l'objet 2 (affecter la variable $x_2$)…
$n^{ème}$ étape : décider si on prend où non l'objet n (affecter la variable $x_n$)
Après avoir déterminé les étapes, on peut en tirer deux manières de définir les sous problèmes à résoudre. La programmation dynamique avant consiste à considérer les décisions dans l'ordre croissant des étapes. La programmation dynamique arrière consiste à considérer les décisions dans l'ordre décroissant des étapes.

## I.3) - Programmation dynamique arrière

Dans l'ordre décroissant des étapes, on obtient la décomposition suivante :

$P^n$ sous problème d'optimisation réduit à la décision de la $n^{ème}$ étape.
$P^{n-1}$ sous problème d'optimisation réduit aux $n^{ème}$ et $n-1^{ème}$ étapes.
…
$P^1$ sous problème d'optimisation pour toutes les étapes

On essaye ensuite d'exprimer la valeur optimale d'un sous problème $P^k$ de manière récurrente : en fonction des valeurs optimales des sous problèmes $P^q$ $q>k$. Le concept central pour arriver à une telle décomposition est celui d'**état**. Une décision fait passer le « système » d'un état à un autre. Les états doivent être définis de telle sorte qu'à une étape donnée *k*, la meilleure décision à prendre ne dépend que de l'état atteint à l'étape précédente. L'état à l'étape *k* peut être la suite de décisions prises aux étapes $q<k$ mais la programmation dynamique est efficace si on peut exprimer cet état plus simplement.

Dans le cas du sac à dos, on définit l'état comme la place t encore disponible dans le sac à dos avec $0 \leq t \leq T$. Grace à ce concept d'état, on obtient la suite de sous problèmes

P(k,t) : problème du sac à dos de taille $0 \leq t \leq T$ et $1 \leq k \leq N$ avec les objets k,k+1,…,n

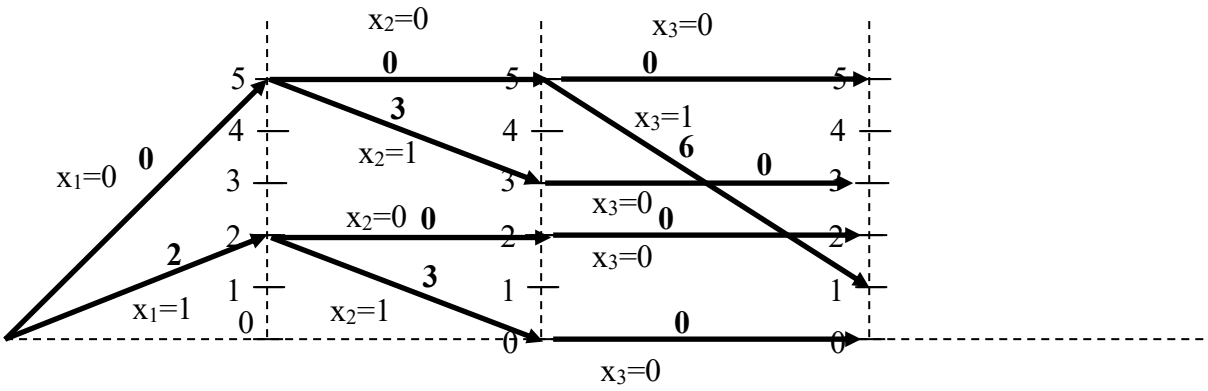
- On sait résoudre facilement P(n,t) pour $t \leq T$ : sa solution optimale est
  $f(n,t)=0$ si $T_n > t$ et $f(n,t)=V_n$ sinon
- On a P(0,T) = P
- On résout facilement P(k,t) en fonction de P(k+1,s) pour $k<N$ avec $s \leq t$ en posant
  $f(k,t) = \max( f(k+1,t) , V_k + f(k+1,t-T_k) )$ si $T_k \leq t$
  $f(k,t) = f(k+1,t)$ sinon

## I.4) - Programmation dynamique et plus (court) long chemin dans un graphe

On peut montrer que la décomposition peut se ramener à la recherche d'un plus long (ou plus court) chemin dans un graphe. Les sommets de ce graphe sont les couples (k,E) où k est un numéro d'étape et E un état.

Considérons le problème du sac-à-dos suivant
Maximiser $2 x_1 + 3 x_2 + 6 x_3$
Sous la contrainte :
$3x_1 + 2x_2 + 4 x_3 \leq 5$
$x_1, x_2, x_3 \in \{0,1\}$

Les décisions des étapes correspondant à la programmation dynamique peuvent se représenter par le graphe suivant un sommet est défini par son abscisse représentant les objets, et son ordonnée représentant la taille du sac à dos.




En valuant les arcs par la valeur de la décision (0 ou $V_k$), on obtient la solution optimale du sac à dos en calculant le plus long chemin du nœud (0,0) aux nœuds (3,t)

## I.5) - Programmation dynamique avant

Comme suggéré par la représentation sous forme de graphe, on peut procéder dans l'ordre croissant des étapes et on obtient alors la décomposition suivante :

$P^1$ sous problème d'optimisation réduit à la première étape.
$P^2$ sous problème d'optimisation réduit à première étape et à la deuxième étape.
…
$P^n$ sous problème d'optimisation pour toutes les étapes

On essaye ensuite d'exprimer la valeur optimale d'un sous problème $P^k$ de manière récurrente : en fonction des valeurs optimales des sous problèmes $P^q$ q<k.

Dans le cas du sac à dos, on obtient par exemple la suite de sous problèmes

P(k,t) : problème du sac à dos de taille $0 \le t \le T$ et $0 \le k \le N$ avec les objets 1,2,…,k

- On sait résoudre facilement P(0,t) pour $t \le T$ : sa solution optimale est f(0,t)=0
- On a P(N,T) = P
- On résout facilement P(k,t) en fonction de P(k-1,s) pour k>0 avec $s \le t$ en posant
  f(k,t) = max( f(k-1,t) , $V_k$ + f(k-1,t+$T_k$) ) si $T_k \le t$
  f(k,t) = f(k-1,t) sinon

*Dans le cas du sac à dos la programmation dynamique avant et arrière sont équivalentes, mais ce ne sera pas le cas en général, l'une pouvant être plus efficace que l'autre.*

## I.6) - Obtention des solutions optimales

La résolution par programmation dynamique permet d'obtenir la valeur optimale de la fonction objectif, mais nous sommes également intéressés par les valeurs optimales des variables de décision. Pour cela, lors des calculs de la solution optimale d'un sous-problème correspondant à l'affectation d'une variable, il suffit de mémoriser la valeur de la variable ayant mené à cette solution.

Considérons par exemple la programmation dynamique avant pour le sac à dos :

```
Fonction SacADosAvant
    Début
    Pour t=0 à T faire
        F[0,T]=0
    Pour k=1 à N faire
        Pour t=0 à T faire
            Si (Tk > t) Alors
                F[k,t]= F[k-1,t]
                x[k,t]=0
            Sinon si (F[k-1,t] > Vk+F[k-1,t+Tk]) alors
                F[k,t]= F[k-1,t]
                x[k,t]=0
            Sinon
                F[k,t]= Vk+F[k-1,t+Tk]
                x[k,t]=1
            Fin si
        Fin Pour
    Fin Pour
```

La complexité de l'algorithme est de O(NT) et il nécessite un tableau T[] de taille N×T pour le stockage des valeurs optimales de la fonction objectif et un tableau x[] de même taille pour le stockage des décisions optimales.

A Partir de la valeur de la fonction objectif on peut obtenir ces valeurs en remontant à partir de $x_N$ en posant :

```
t=T
Pour k=N à 1
xk=x[k,t]
t = t - TkXk
```

# 2 PROGRAMMATION DYNAMIQUE (SUITE)

## 2.1 Schéma général de conception d'un algorithme de programmation dynamique

1. Caractérisation de la structure d'une solution optimale
2. Définition récursive de la valeur de la solution optimale
3. Calcul ascendant de la valeur de la solution optimale
4. Construction de la solution optimale à partir des informations obtenues à l'étape précédente

## 2.2 Problème du sac à dos

> $n$ objets ayant chacun un poids $a_i$ et une valeur $c_i$.
>
> Déterminer l'ensemble des objets que l'on peut emporter de manière à maximiser la valeur totale et sans dépasser un poids total $b$.

**1. Caractérisation de la structure d'une solution optimale**

Pour résoudre *P*, nous avons deux possibilités. Soit on prend l'objet *n.* On a un gain de $c_n$ et il reste à résoudre un problème sac à dos avec *n*-1 objets et un poids maximum de $b$-$a_n$. Soit on ne prend pas l'objet *n*. On a un gain nul et il reste à résoudre un problème de sac à dos avec *n*-1 objets et un poids maximum de *b*. On a donc mis en évidence une sous-structure optimale.

**2. Définition récursive de la valeur de la solution optimale**

*n «* étapes » : variables de décision $x_1,x_2,\ldots,x_n$ avec $x_i \in \{0,1\}$ (on prend l'objet ou non)

Variable d'état $s_p \in \{0,1,\ldots,b\}$ capacité restante du sac à la fin de l'étape $p$ avec $s_n$ inconnue a priori.

Fonction de transfert : $t_p(x_p,s_p)= s_p + a_p x_p$ (capacité restante du sac avant d'ajouter l'objet *n*)

Fonction de coût (gain) $f_p(x_p,s_p) = c_p x_p$ si $a_p x_p \leq s_p$

Récurrence : $v_p(s_p) = \max\{ f_p(x_p,s_p) + v_{p-1}(s_p + a_p x_p) \mid x_p \in \{0,1\}, a_p x_p \leq s_p \}$

*Initialisation* $v_0(s_1 + a_1 x_1)=0 \ \forall \ s_1, x_1$

**3. Calcul ascendant de la valeur de la solution optimale** (dans l'ordre $v_1(s_1)$, $v_2(s_2)$, … ) . Remarque comme $s_0 = b$, les seuls états atteignables pour $s_1$ sont $b$ et $b$-$a_1$ on peut donc se limiter au calcul de $v_1(b)$ et $v_1(b\text{-}a_1)$. De même pour l'étape 2 seuls $b$, $b$-$a_2$, $b$-$a_1$ et $b$-$a_1$-$a_2$ peuvent être atteints… Si pour un état donné la fonction de transfert donne un état non atteignable on peut l'ignorer.

Dans le pire des cas, on énumère *nb* états. L'algorithme de programmation dynamique a donc une complexité de $O(nb)$ ce qui est *pseudopolynomial* (La valeur de *b* peut être aussi grande qu'on veut, et par exemple égale à $2^n$).

Exemple :

Maximiser $2\ x_1 + 3\ x_2 + 6\ x_3$

Sous la contrainte :

$2x_1 + 2x_2 + 4\ x_3 \leq 4$

$x_1, x_2, x_3 \in \{0,1\}$

$v_0(4)=0$ permet d'atteindre les états 4 ($x_1$=0) et 2 ($x_1$=1) de l'étape 1

$v_1(4) = 0$ permet d'atteindre les états 4 ($x_2$=0) et 2 ($x_2$=1) de l'étape 2

$v_1(2) = 2$ permet d'atteindre l'état 2 ($x_2$=0) et 0 ($x_2$=1) de l'étape 2

$v_2(4) = 0$ permet d'atteindre les états 4 ($x_3$=0) et 0 ($x_3$=1) de l'étape 3

$v_2(2) = \max (0+v_1(2),3+ v_1(4))=3$ permet d'atteindre l'état 2 ($x_2$=0) et 0 ($x_2$=1) de l'étape 3

$v_2(0) = \max ( 3+ v_1(2),\sout{0+ v_1(0)}) = 5$ permet d'atteindre l'état 0 ($x_3$=0) de l'étape 3 (~~barré~~ : état non atteignable)

$v_3(4) = 0$

$v_3(2) = 0+v_2(2) =3$

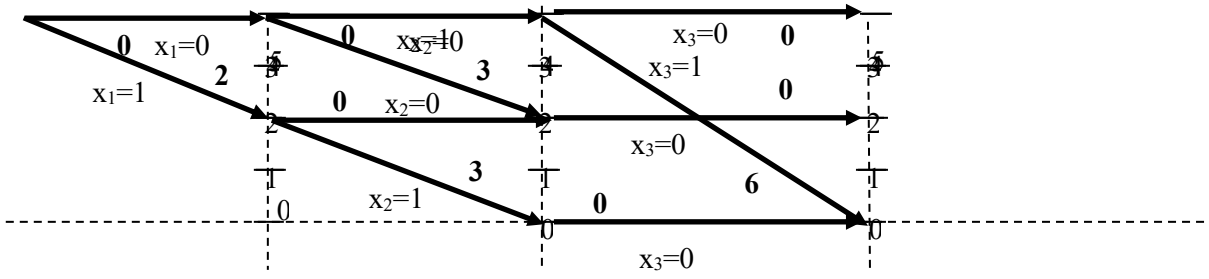
$v_3(0) = \max ( 6+ v_2(4),0+ v_2(0)) = 6$

**4. Construction de la solution optimale à partir des informations obtenues à l'étape précédente**

Il suffit de mémoriser un tableau $x[p,s_p]$ la décision prise pour maximiser $v_p(s_p)$ puis de remonter dans l'espace des états. La décision donnant l'état précédant par la fonction d transfert.

**Remarque :**

On peut montrer que dans ce cas, la décomposition se ramène à la recherche d'un plus long (ou plus court) chemin dans un graphe. Les décisions des étapes correspondant à la programmation dynamique peuvent se représenter par le graphe suivant un sommet est défini par son abscisse représentant les objets, et son ordonnée représentant la taille du sac à dos.




**Exercice**

Proposer un algorithme de programmation dynamique pour résoudre le problème du voyageur de commerce à $n$ villes telles que $d_{ij}$ donne la distance minimale entre deux villes $i$ et $j$. Le dépôt est supposé situé dans la ville 1. Definir de manière récursive la distance optimale $D(S,k)$ qui donne la distance minimale partant de la ville 1 visitant toutes les villes de S et terminant à la ville $k$.

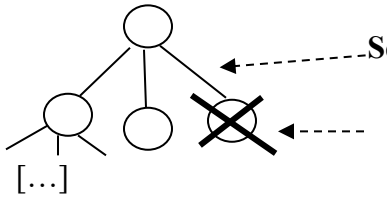
## 3. RECHERCHE ARBORESCENTE – PROCEDURE DE SEPARATION ET EVALUATION (PSE) OU BRANCH & BOUND.

### 3.1 . Principe de la méthode

**Enumération implicite** de l'espace de recherche de solutions :

- Partage de l'espace des solutions en espaces plus petits ;
- Elimination de certains de ces espaces avec des calculs de bornes sur le critère.

Recherche arborescente : un nœud = un sous-espace de solution. Nœuds fils d'un nœuds : sous-espaces de solution dont l'union donne l'espace représenté par le nœud ascendant.




**Séparation** ou Branchement (Branch)

Elimination d'un nœud après **évaluation**
souvent par calcul de borne inférieure (supérieure) de la fonction objectif $f(x)$ (bound) pour un problème de minimisation (maximisation).

Dans le pire cas : énumération complète ….. Seulement pour des problèmes de taille moyenne !

Dans une PSE on doit disposer de :

- une règle de séparation (partitionnement) de l'espace de solutions ;
- une fonction d'évaluation des solutions;
- une stratégie d'exploration (règles de branchement).

- **Règle de séparation**

On part de l'ensemble $S_0$ des solutions initiales. On sépare cet ensemble en sous ensembles en appliquant des décisions successives. Toute règle de séparation est possible si on ne perd aucune solution :

Union des sous ensembles générés = ensemble initial

Souvent séparation = énumération des valeurs possibles d'une des variables de décision.

- **Fonction d'évaluation**

Pour éviter l'énumération complète : supprimer des sous ensembles en utilisant une fonction d'évaluation. Eliminer ainsi des sous ensembles ne contenant pas la solution optimale.
*Evaluation d'un sous ensemble* (= évaluation d'un nœud de l'arborescence)
= estimation de coût de la meilleure solution de cet ensemble.

*Pour un problème de minimisation : Evaluation par BORNE INFERIEURE*

A tout nœud $S_i$, éval($S_i$) = borne inférieure des coûts des solutions de $S_i$

éval($S_i$) $\leq f(x), \forall x \in S_i$.

Dans $S_i$, il y a au mieux une solution dont le coût vaut éval($S_i$).

Si on trouve un nœud $S_j$, tel que éval($S_j$) ≤ éval(solution courante) alors

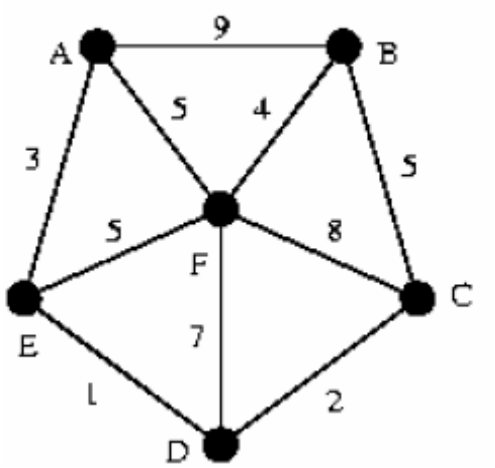
Ne pas explorer le nœud $S_j$

➔ **Elagage de l'arborescence**

Pour obtenir une fonction d'évaluation :

Relaxation de contraintes dans le sous problème correspondant au nœud $S_i$ pour résoudre un problème plus facile.

### 3.2 . Branch and Bound pour le problème du voyageur de commerce

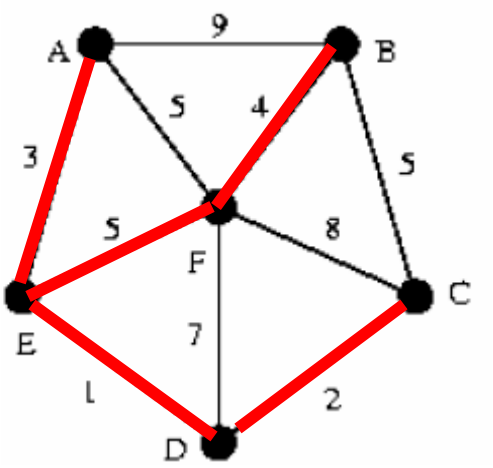



- **Exemple de séparation :** choix du prochain sommet à visiter
- **Exemples d'évaluation par borne inférieure :**

a) Borne inférieure triviale :Si *n* sommets restent à visiter : *n* plus petits arcs restant à visiter

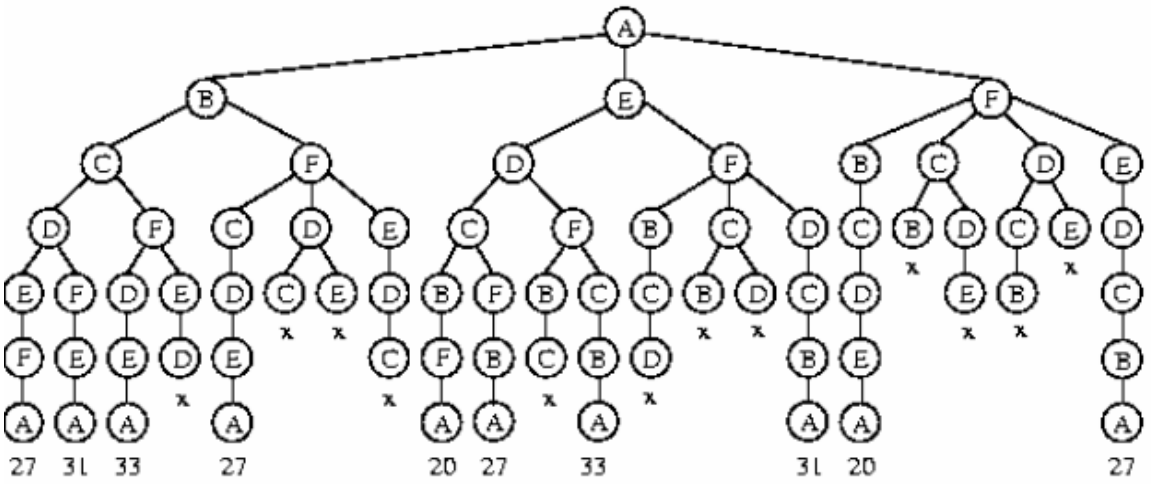
Classements des arcs dans l'ordre croissant des valeurs 1,2,3,4,5,5,5,7,8,9 => Borne inférieure pour 5 arcs 15.

b) Arbre couvrant de poids minimal
Borne inférieure = 15

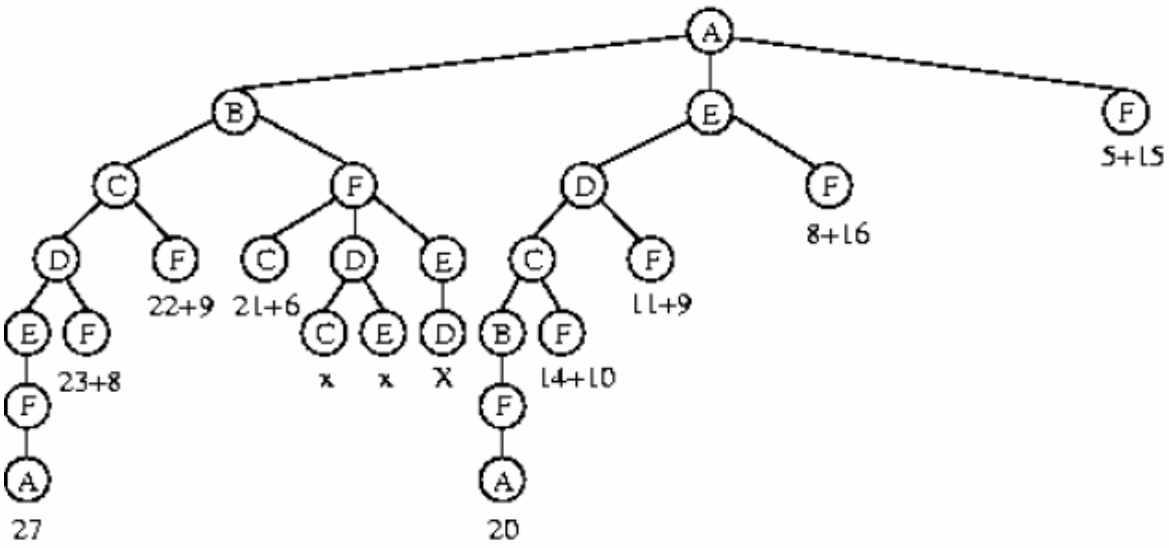


- **Recherche arborescente : énumération complète avec ou sans évaluation**

**3.2 . PSE pour le problème du sac-à-dos**




Backtracking/exhaustive search




$$\max(\sum_{i=1}^{n} c_i.x_i) \ / \ \sum_{i=1}^{n} a_i.x_i \le b \ \text{ et } \ x = 0 \ ou \ 1$$

On va prendre $n = 5$ ; $A = (18,12,15,16,13)$, $C = (27,9,30,16,6.5)$ et $b = 45$

1 nœud = 1 état du problème : certains objets sont sélectionnés ; d'autres non.

Séparation : choisir une variable $x_i$ et poser $x_i \leftarrow 1$ ou $x_i \leftarrow 0$. Pour choisir entre les variables $x_i$ : on les classe par utilité maximale. L'utilité est définie par $u_i = \frac{c_i}{a_i}$

Evaluation : (méthode heuristique ignorant les contraintes d'intégrité) Trier les objets par utilité

décroissante, prendre les objets dans l'ordre jusqu'à ce que ne soit plus possible (on peut éventuellement fractionner la quantité du dernier objet pris). On aura une solution non binaire. A partir de cette solution, calculer son coût (pas forcément entier). Ici on cherche à maximiser : on aura une évaluation par borne supérieure.

Stratégie d'exploration : plus grande évaluation d'abord

Utilité :

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $c_i$ | 27 | 9 | 30 | 16 | 6,5 |
| $a_i$ | 18 | 12 | 15 | 16 | 13 |
| $u_i$ | 1,5 | 0,75 | 2 | 1 | 0,5 |

Ordre des objets : 3 1 4 2 5

Etat initial $S_0$ : en cherche à évaluer une borne supérieure, pour cela on va calculer une solution non en $\{0,1\}$.
On part de la contrainte :

$$a_3.x_3 + a_1.x_1 + a_4.x_4 + a_2.x_2 + a_5.x_5 \leq 45$$

On peut prendre $x_3 = 1$ et $x_1 = 1$, ensuite on ne peut pas prendre $x_4 = 1$ car la contrainte serait violée.
On pose donc $x_2 = x_5 = 0$, et il faut calculer la valeur de $x_4$.
$15 + 18 + 16x_4 = 45$, d'où $x_4 = 3/4$.
On peut alors évaluer $S_0$ : $c_3.x_3 + c_1.x_1 + c_4.x_4 + c_2.x_2 + c_5.x_5 = 69$
➔ on a bien une borne supérieure.

Séparation de $S_0$ : $x_3 \leftarrow 0$ (état $S_1$) ou $x_3 \leftarrow 1$ (état $S_2$). Il faut évaluer ces 2 états et continuer à partir de celui d'évaluation maximale

Ainsi de suite jusqu'à ne plus pouvoir développer l'arborescence. Comparez avec la méthode de programmation dynamique. **A FAIRE**

**Solution : (** $x_3 = 1$, $x_1 = 1$, $x_4 = 0$, $x_2 = 1$, $x_5 = 0$ **) coût de 66.**

# 4 HEURISTIQUES

## 4.1 Evaluation des méthodes approchées

Méthode approchée (ou heuristique) pour un problème d'optimisation combinatoire : fournit une solution non nécessairement optimale, avec ou sans garantie d'optimalité. Soit $P$ un problème d'optimisation combinatoire et soir $i$ une instance de ce problème. La performance d'une heuristique $H$ sur l'instance $i$ est donnée par

$$Perf_H(i)=\frac{H(i)}{OPT(i)}$$

où $OPT(i)$ est la valeur de la solution optimale de $i$ et $H(i)$ est la valeur obtenue par l'heuristique. Pour un problème de minimisation : $H(i)$ est une borne supérieure de $OPT(i)$ et $Perf_H(i) \geq 1$.
En pratique $OPT(i)$ peut être difficile à calculer. Par contre (voir cours précédents), on dispose souvent de méthodes de calcul de bornes inférieures BI telles que $BI(i) \leq OPT(i)$ et donc telles que
$Perf_H(i)=H(i)/OPT(i) \leq H(i)/BI(i)$
Il s'ensuit que l'évaluation d'une heuristique relativement à une borne inférieure donne une borne supérieure de la performance de l'heuristique.

On peut chercher à évaluer la performance d'une heuristique a priori ou a posteriori.

L'évaluation a priori consiste à évaluer la performance de l'heuristique dans le pire des cas (c'est à dire pour la pire des instances) relativement à une borne inférieure et donc fournit une véritable garantie de performance.

*[Voir l'exemple de la performance de l'heuristique donnée en cours pour le voyageur de commerce euclidien]*

L'évaluation a posteriori mesure l'écart constaté entre l'heuristique et la borne inférieure pour une instance donnée. Cette évaluation permet d'effectuer des mesures statistiques de la performance des heuristiques, par exemple en générant des instances aléatoirement.

## 4.2 Heuristiques gloutonnes

A chaque étape de résolution, prendre une décision (selon une règle de « bon sens » ou heuristique) sans remise en cause de cette décision (pas de retour en arrière). On obtient ainsi une borne supérieure de l'optimum (pour un problème de minimisation).

Exemple : la méthode du plus proche voisin pour le voyageur de commerce et la méthode du classement par utilités décroissantes pour le problème du sac à dos.

Ces méthodes ont l'avantage d'être très rapides et l'inconvénient de ne pouvoir en général garantir une performance acceptable dans le pire des cas ou même en moyenne.

## 4.3 Méthode de recherche locale

Partir d'une solution initiale ayant un coût donné et appliquer des transformations ou mouvements pour construire des solutions dont le coût est meilleur. La méthode s'arrête lorsque le coût de la

solution courante ne peut plus être amélioré. On note $V(s)$ le voisinage (ensemble de solutions dites voisines), défini à partir de transformations, d'une solution $s$ .

s ← solution initiale
z ← f(s)
Tant qu'il existe s'∈ $V(s)$ telle que $f(s') < f(s)$ faire
    $s \leftarrow s'$
    $z \leftarrow f(s)$
Fin tant que

La solution initiale peut être fournie par une heuristique gloutonne ou être choisie de manière aléatoire. On peut relancer la méthode pour plusieurs solutions initiales et conserver ensuite la meilleure des solutions obtenues. Lors de l'exploration du voisinage : on peut choisir la meilleure solution de tout le voisinage ou la 1ère solution améliorante trouvée dans la voisinage. Il faut définir des transformations permettant d'avoir des voisinages pas trop grand par rapport à l'ensemble des solutions (sinon on revient à une énumération complète).

Cette méthode conduit à un optimum **local relativement au voisinage choisi**.

*[voir l'exemple de la méthode 2-opt pour le voyageur de commerce vue en cours]*

### 4.4 Méta-heuristiques

Les méthodes locales peuvent être piégées dans des minima locaux. Les méta-heuristiques proposent pour sortir de ces minima locaux d'accepter des solutions dont le coût peut augmenter par rapport à la solution initiale. Chaque méta-heuristique propose un mécanisme de contrôle de ces dégradations des solutions. Ces méthodes permettent ainsi de sortir du piège d'un minimum local et parte explorer une autre partie de l'espace des solutions. Les inconvénients principaux de ces méthodes sont les difficultés de paramétrage et un temps de calcul relativement élevé. Il y a différentes méta-heuristiques et il est difficile de savoir comment choisir une méthode plutôt qu'une autre. Parmi les méta-heuristiques les plus répandues, on trouve le recuit simulé, la méthode tabou et les algorithmes génétiques. On présente ici la méthode tabou *[voir exercice donné en cours]*.

Comme pour une méthode de recherche locale, on a une solution initiale (une seule solution courante à la fois) et un voisinage
- Lors d'une itération donnée, on évalue le coût pour chaque élément de $V(s)$ ;
- On retient comme nouvelle solution $s'$ celle ayant le meilleur coût sur tout le voisinage $V(s)$ . On accepte donc une solution même si elle a un coût supérieur à la solution courante $s$ .
- Pour éviter de boucler sur des solutions déjà obtenues, on utilise une liste appelée liste de solutions tabou ou liste tabou interdisant de revenir sur des solutions déjà explorées.
- On passe ensuite à l'itération suivante.

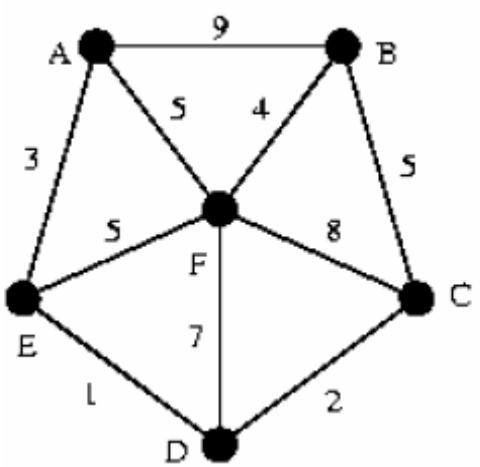
La méthode s'arrête lorsqu'un critère d'arrêt est atteint :
- nombre maximum d'itérations autorisé ;
- nombre maximum d'itérations sans amélioration de la solution

Il faut conserver la meilleure solution obtenue au cours des itérations.

Liste tabou : plutôt que mémoriser les solutions visitées, on mémorise pendant un nombre fini d'itérations les mouvements effectués pour ne pas refaire les mouvements inverses (par exemple : ne pas échanger les variables $x_i$ et $x_j$ si le voisinage est défini en termes d'échanges ).

# Correction des exercices (MMAD2)

## 1 . Programmation dynamique pour le problème du voyageur de commerce




> Proposer un algorithme de programmation dynamique pour résoudre le problème du voyageur de commerce à *n* villes (passer une et une seule fois par chaque sommet du graphe en minimisant la distance totale parcourue et en empruntant uniquement les arcs matérialisés sur le graphe). $d_{ij}$ donne la distance minimale entre deux villes (sommets) *i* et *j*. Tester cet algorithme sur l'exemple ci-dessus. Le dépôt (point de départ du circuit) est supposé situé dans la ville A.

*Equation de récurrence de la programmation dynamique*

$D(S,k)$ donne la distance minimale partant de la ville 1 visitant toutes les villes de S et terminant à la ville $k \in S$.

Cas élémentaire : $D(\{k\},k) = d_{1k}$
Relation de récurrence : $D(S,k) = \min_{m \in S\setminus\{k\}} ( D(S\setminus\{k\},m)+d_{mk} )$ pour tout $k \in S$ et pour tout $S \subseteq \{1,\ldots,n\}$

*Test sur l'exemple*

Calcul ascendant des $D(S,k)$ pour tout $S$ et pour tout $k \in S$. Les cas impossibles ne sont pas écrits :

Cas $|S| = 1$
(1) $D(\{B\},B) = 9$ (2) $D(\{F\},F) = 5$ (3) $D(\{E\},E) = 3$

Cas $|S| = 2$
(4) $D(\{B,F\},B)=(2)+4=9$ (5) $D(\{B,F\},F)=(1)+4=13$ (6) $D(\{B,C\},C)=(1)+5=14$
(7) D({F,C},C) =(2)+8=13 (8) D({F,D},D}=(2)+7=12 (9) D({E,F},E}=(2)+5=10
(10) D({E,F},F}=(3)+5=8 (11) D({E,D},D}=(3)+1=4

Cas $|S| = 3$
(12) D({B,F,E},B)=(10)+4=12 (13) D({B,F,E},E)=(5)+5=18
(14) D({B,F,D},D)=(5)+7=20

(15) D({B,F,C},B)= (7)+5=18 (16) D({B,F,C},F)= (6)+8=22
(17) D({B,C,D},D)= (6)+2=16
(18) D({F,C,D},C)=(8)+2= 14 (19) D({F,C,D},D)=(7)+2=15
(20) D({F,E,D},E)=(8)+1=13 (21) D({F,E,D},F)=(11)+7=11
(22) D({E,D,C},C)=(11)+2=6
(23) D({E,F,C},C)=(10)+8=16
(24) D({B,F,C},C)= min (**(4)+5**,(5)+8)=14
(25) D({F,E,D},D)= min (**(9)+1**,(10)+7)= 11

Cas $|S| = 4$
(26) D({B,F,E,D},B)= (21)+4=15 (27) D({B,F,E,D},E)= (14)+1=21
(28) D({B,F,E,D},D)=(13)+1=19
(29) D({B,F,E,C},B)=(23)+5=21 (30) D({B,F,E,C},E)= (16)+5=27
(31) D({B,F,E,C},C)= (12)+5=17
(32) D({B,F,D,C},F)=(17)+7=23 (33) D({B,F,D,C},B)=(18)+5=19
(34) D({B,F,D,C},D)= min ((16)+7,**(24)+2**)= 16
(35) D({B,F,D,C},C)=(14)+2=22
(36) D({B,C,D,E},E)=(17)+1=17 (37) D({B,C,D,E},B)=(22)+5=11
(38) D({F,C,D,E},F)=(22)+8=14 (39) D({F,C,D,E},D)= (23)+2=18
(40) D({F,C,D,E},E)=(19)+1=16
(41) D({F,C,D,E},C)= min ((21)+8,**(25)+2**))=13

Cas $|S| = 5$
(42) D({B,C,D,E,F},B)= min (**(38)+ 4**, **(41)+5**)= 18 [chemins A,E,D,C,F,B et A,F,E,D,C,B]
(43) D({B,C,D,E,F},C)= min (**(26)+5**, (28)+2)=20 [chemin A,E,D,F,B,C]
(44) D({B,C,D,E,F},D)= min((30)+1,**(31)+2**)= 19 [chemin A,E,F,B,C,D]
(45) D({B,C,D,E,F},E)= min((32)+5,**(34)+1**))=17 [chemin A,F,B,C,D,E]
(46) D({B,C,D,E,F},F)= min((36)+5,**(37)+4**))=15 [chemin A,E,D,C,B,F]

Cas $|S| = 6$
(47) D({B,C,D,E,F},A)=min((42)+9,**(45)+3**,**(46)+5**)=20

On obtient 2 cycles optimaux de durée 20 A,F,B,C,D,E,A et A,E,D,C,B,F,A.

*Comparaison avec l'énumération complète naïve*
On a effectué 47 calculs de valeurs D(S,k). L'énumération de toutes les solutions possibles en ignorant les arcs impossibles (permutations de B,C,D,E,F ) donne 5 !=120 permutations. La programmation dynamique donne donc de meilleurs temps de calculs que l'énumération complète stupide.

**Exercice : Prouvez ce résultat en calculant la complexité temporelle dans le pire des cas des algorithmes.**

*Comparaison avec la recherche arborescente exhaustive*
La recherche arborescente exhaustive éliminant les chemins interdits (haut de la figure du support de cours) ne comporte que 72 nœuds. Elle reste donc moins performante que la programmation dynamique.

*Comparaison avec le branch and bound*
L'arbre de recherche du branch and bound (bas de la figure du support de cours) ne comporte que 26

nœuds. Il est donc plus efficace que la programmation dynamique (si on ne prend pas en compte le temps de calcul de la borne inférieure !).

# Correction des exercices suite et fin (MMAD2)

## 1. Complexité de l'énumération exhaustive et de la programmation dynamique pour le voyageur de commerce

Considérons un problème de voyageur de commerce à $n$ villes.

Pour l'énumération exhaustive, en fixant arbitrairement le point de départ à un sommet on doit énumérer (n-1) ! permutations. Disons que la complexité de l'énumération exhaustive est de $O(n\,!)$.

Pour la programmation dynamique, on rappelle la récurrence utilisée :
$D(\{k\},k) = d_{1k}$ pour $k = 2,\ldots, n$
$D(S,k) = \min_{m \in S\setminus\{k\}} ( D(S\setminus\{k\},m)+d_{mk} )$ pour tout ensemble $S$ et pour tout $m \in S\setminus\{k\}$

Cette récurrence suppose qu'on part du sommet 1. Comptons le nombre d'additions et de comparaisons nécessaires au calcul de tous les $D(S,k)$.

On considère les ensembles $S$ par cardinalité » croissante (voir correction de l'exerice précédent).

Pour $|S| = 1$ on effectue n-1 initialisations (pour $k = 2,\ldots, n$)
Pour un ensemble $S$ tel que $|S| \geq 2$ on effectue on calcule $|S|$ valeurs $D(S,k)$ différentes, Pour chacune de ces valeurs on effecteur $|S|$-1 comparaisons pour calculer le min. soit $q$ un entier $\geq 2$. le nombre d'ensembles $S$ tels que $|S| = q$ est égal au nombre de parties à $q$ éléments dans un ensemble à $n$-1 éléments donné par le coefficient binomial :

$$\binom{n-1}{q}$$

On obtient donc pour une valeur de $q$ donnée un nombre d'opérations élémentaires égal à

$$q(q\text{-}1)\binom{n-1}{q}+(\text{n-1})$$

La cardinalité de l'ensemble variant de 2 à n-1, on obtient un nombre d'opérations élémentaires total égal à (NOP désigne le nombre d'opérations élémentaires) :

$$\text{NOP}=\sum_{q=2}^{n-1} q(q-1)\binom{n-1}{q} +(\text{n-1}) \qquad (1)$$

Pour simplifier cette expression, on peut utiliser les propriétés suicanets des coefficents binomiaux :

$$\sum_{q=0}^{n}\binom{n}{q} = 2^n \qquad (2)$$

$$\binom{n}{q} = \frac{n}{q}\binom{n-1}{q-1} \qquad (3)$$

En utilisant un changement de variables dans (1) on obtient

$$\text{NOP}=\sum_{q=0}^{n-3}(q+2)(q+1)\binom{n-1}{q+2}+(\text{n-1})$$

En appliquant (3), il vient

$$\text{NOP}=\sum_{q=0}^{n-3}\frac{(q+2)(q+1)(n-1)}{(q+2)}\binom{n-2}{q+1}+(\text{n-1})$$

En appliquant de nouveau (3), on obtient

$$\text{NOP}=\sum_{q=0}^{n-3}\frac{(q+2)(q+1)(n-1)(n-2)}{(q+2)(q+1)}\binom{n-3}{q}+(\text{n-1})$$

Et donc

$$\text{NOP}=(n-1)(n-2)\sum_{q=0}^{n-3}\binom{n-3}{q}+(\text{n-1})$$

Finalement avec (2) le nombre total d'opérations élémentaires est

$$\text{NOP}=(n\text{-}1)(n\text{-}2)2^{n-3}+(\text{n-1})$$

Soit une complexité de $O(n^2 2^n)$. La programmation dynamique est donc beaucoup plus rapide que l'énumération exhaustive même si elle reste de complexité exponentielle. A titre d'exemple, voici pour quelques valeurs de $n$ la croissance comparée des fonctions (n-1) ! et $n^2 2^n$ :

| n | (n-1) ! | $n^2 2^n$ |
|---|---|---|
| 4 | 6 | 256 |
| 10 | 362880 | 41472 |
| 20 | 121.645.100.408.832.000 | 189.267.968 |

Par contre la programmation dynamique requiert une place mémoire (ou complexité spatiale) également exponentielle. On doit conserver toutes les valeurs D($S$,$k$) soit (ND désigne le nombre d'éléments à mémoriser) en utilisant le même principe que pour le nombre d'opérations élémentaires:

$$ND=\sum_{k=1}^{n-1}k\binom{n-1}{k}=(n\text{-}1)2^{n-2}=\text{O}(\text{n}^2 2^{\text{n}})$$

## 2. Branch-and-bound et programmation dynamique pour le sac-à-dos

On considère le problème du sac-à-dos

$$\max(\sum_{i=1}^{n}c_i.x_i)\ /\ \sum_{i=1}^{n}a_i.x_i\le b\ \text{ et }\ x=0\ ou\ 1$$

et une instance définie par :
e $n=5$ ; $a=(18,12,15,16,13)$, $c=(27,9,30,16,6.5)$ et $b=45$

**1) On définit l'arbre de recherche**

On choisit un arbre de recherche binaire. A chaque nœud on choisit si on prend un objet *i* ou non (décisions $x_i$=1 et $x_i$=0) Un nœud correspond ainsi à un ensemble d'objets pris (noté P), un ensemble d'objets non pris (noté $\overline{P}$) et un ensemble d'objets au statut indéterminé (noté *I*).

**2) On définit une méthode de calcul d'une borne supérieure (voir cours sur la PLNE) :**

Soit l'utilisé d'un objet *i* définie par $u_i=c_i/a_i$. A un nœud donné, trier les objets indéterminés par utilité décroissante, prendre les objets dans l'ordre jusqu'à ce que ne soit plus possible compte tenu de la capacité de sac à dos et desobjets pris (on peut éventuellement fractionner la quantité du dernier objet pris). On aura une solution non binaire (donc non nécessairement réalisable qui constitue une borne supérieure)

**2) On définit une stratégie de branchement**

A un nœud donné brancher sur l'objet de plus grande utilité. Pour le choix du prochain nœud, on applique la stratégie du « meilleur d'abord », c'est-à-dire, qu'on sélectionne le nœud ouvert de plus grande borne supérieure en espérant trouver des bonnes solutions par cette stratégie.

**<u>Question</u> : Appliquer la méthode de branch-and-bound ainsi définie à l'instance donnée et comparer avec la méthode de programmation dynamique pour le sac-à-dos.**

**<u>Réponse</u> :**

Calculons les utilités de objets.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $c_i$ | 27 | 9 | 30 | 16 | 6,5 |
| $a_i$ | 18 | 12 | 15 | 16 | 13 |
| $u_i$ | 1,5 | 0,75 | 2 | 1 | 0,5 |

On obtient l'ordre initial 3,1,4,2,5.

**Nœud racine (noté $S_0$)** I={3,1,4,2,5} $P = \overline{P} = \{\}$
On calcule une borne supérieure. On part de la contrainte :

$$a_3.x_3 + a_1.x_1 + a_4.x_4 + a_2.x_2 + a_5.x_5 \leq 45$$

On peut prendre $x_3 = 1$ et $x_1 = 1$, ensuite on ne peut pas prendre $x_4 = 1$ car la contrainte serait violée.

On pose donc $x_2 = x_5 = 0$, et il faut calculer la valeur de $x_4$.

$15+18+16x_4 = 45$, d'où $x_4 = 3/4$.

On peut alors évaluer BS( $S_0$) = $c_3.x_3 + c_1.x_1 + c_4.x_4 + c_2.x_2 + c_5.x_5 = 69$

Notons qu'on dispose également d'une borne inférieure donné par la solution réalisable consistant à prendre les objets 3 et 1. Soit BI=30+27=57. On pourra ainsi couper tout nœud de borne supérieure plus petite que BI=57.

On branche en générant deux nœuds fils $x_3 \leftarrow 1$ (nœud $S_1$) ou $x_3 \leftarrow 0$ (nœud $S_2$). Il faut évaluer ces 2 nœuds et continuer à partir de celui d'évaluation maximale

**Nœud $S_1$** I={1,4,2,5} $P$ ={3} $\overline{P}$ = {}
Calcul de la borne inférieure. On obtient la même borne supérieure que le nœud $S_0$ soit BS($S_2$)=69.

**Nœud $S_2$** I={1,4,2,5} $P$ ={} $\overline{P}$ = {3}
Calcul de la borne inférieure : Dans l'ordre des utilités croissantes des objets indéterminés, on peut prendre les objets 1 et 4. Posons $18+16+12x_2=45$. On obtient $x_2=11/12$. On calcule donc BS($S_1$)=51,25. On a BS($S_1$)<BI. On ne peut obtenir mieux que ma solution réalisable déjà trouvée au nœud racine. On stoppe donc l'exploration de ce nœud.

$S_1$ étant le seul nœud « ouvert », on branche à partir de ce nœud sur le prochain objet non pris dans l'ordre des utilités décroissantes, soit 1. On obtient ainsi les nœuds $S_3$ et $S_4$.

**Nœud $S_3$** I={4,2,5} $P$ ={3,1} $\overline{P}$ = {}
Calcul de la borne inférieure. On obtient la même borne supérieure que le nœud $S_0$ soit BS($S_2$)=69.

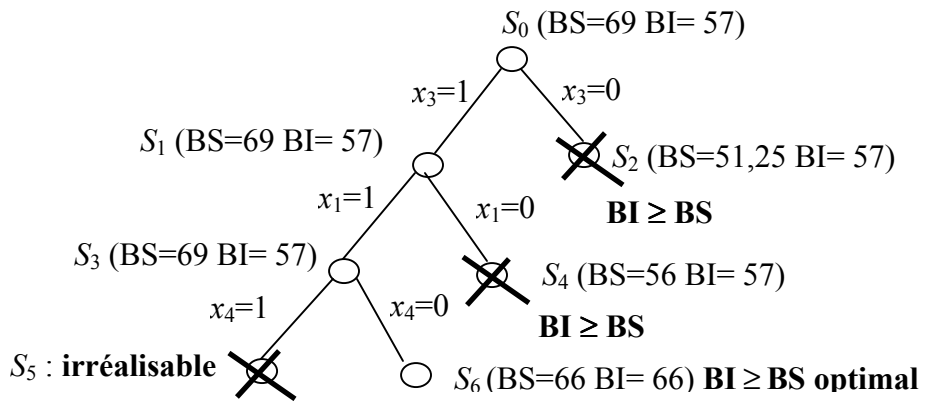
**Nœud $S_4$** I={4,2,5} $P$ ={3} $\overline{P}$ = {1}
Calcul de la borne inférieure. On doit prendre l'objet 3. Dans l'ordre des utilités croissantes des objets indéterminés, on peut ajouter 4 et 2. Posons $15+16+12+13x_5=45$. On obtient $x_5=2/13$. On calcule donc BS($S_4$)=30+16+9+6.5×2/13=56. On a BS($S_4$)<BI. On ne peut obtenir mieux que ma solution réalisable déjà trouvée au nœud racine. On stoppe donc l'exploration de ce nœud.

$S_3$ étant le seul nœud « ouvert », on branche à partir de ce nœud sur le prochain objet non pris dans l'ordre des utilités décroissantes, soit 4. On obtient ainsi les nœuds $S_5$ et $S_6$.

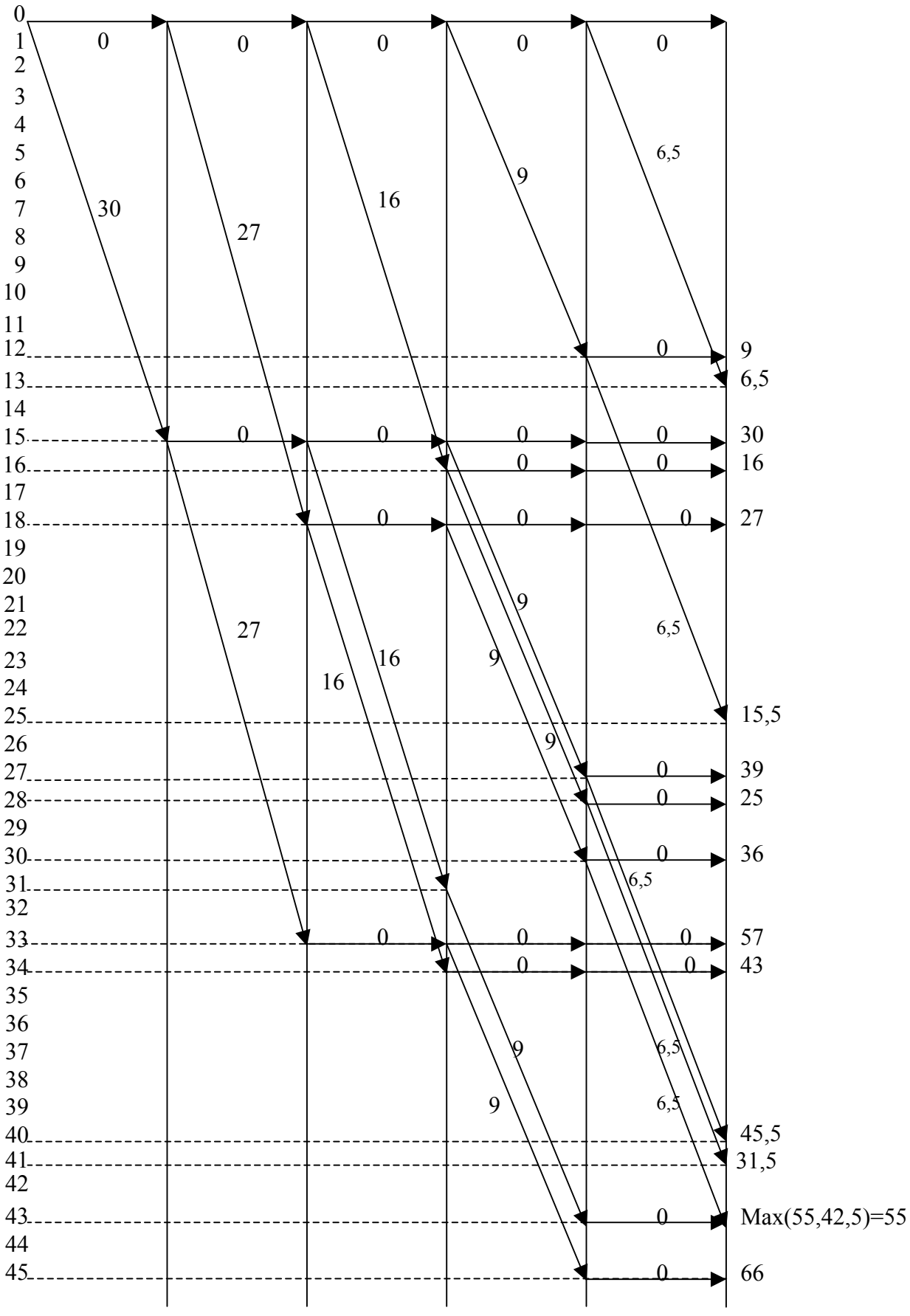
**Nœud $S_5$** I={2,5} $P$ ={3,1,4} $\overline{P}$ = {}
La somme des poids des objets 3,1 et 4 dépasse 45. On ne peut donc continuer la recherche à partir de ce nœud.

**Nœud $S_6$** I={2,5} $P$ ={3,1} $\overline{P}$ = {4}
Calcul de la borne inférieure. On doit prendre les objets 3 et 1. Dans l'ordre des utilités croissantes des objets indéterminés, on peut ajouter 2. Posons $15+18+12+13\ x_5=45$. On obtient $x_5=0$ et BS($S_6$)=30+27+9=66. Par ailleurs puisque $x_5=0$, on a également une solution réalisable de valeur 66 qui consiste à prendre les objets 3,1 et 2. On peut mettre à jour la borne inférieure et poser BI=66. Comme BS($S_6$)=BI. On stoppe la recherche à partir de ce nœud.

Il n'y a plus de nœud ouvert la meilleure solution est donc de prendre les objets 1,2 et 3 de valeur totale 66.



**Application de  la méthode de programmation dynamique**



**CORRECTION DE L'EXERCICE– METHODE TABOU**

- **Problème d'affectation quadratique**

  n objets et flots $f_{ij}$ entre toute paire d'objet i et j

  n emplacements et distance $d_{pipj}$ entre les places des objets i et j (respectivement pi et pj)

  Le problème : trouver une place pour chaque objet de telle sorte que le produit flot par distance soit minimisé : $Min(\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij}.d_{pipj})$

  Application : problème de taille 5x5

  Matrice de flots

| | O1 | O2 | O3 | O4 | O5 |
|---|---|---|---|---|---|
| O1 | 0 | 5 | 2 | 4 | 1 |
| O2 | 5 | 0 | 3 | 0 | 2 |
| O3 | 2 | 3 | 0 | 0 | 0 |
| O4 | 4 | 0 | 0 | 0 | 5 |
| O5 | 1 | 2 | 0 | 5 | 0 |

  Matrice de distances

| | p1 | p2 | p3 | p4 | p5 |
|---|---|---|---|---|---|
| p1 | 0 | 1 | 1 | 2 | 3 |
| p2 | 1 | 0 | 2 | 1 | 2 |
| p3 | 1 | 2 | 0 | 1 | 1 |
| p4 | 2 | 2 | 2 | 0 | 2 |
| p5 | 3 | 2 | 2 | 1 | 0 |

- **Représentation d'une solution :**

  un tableau de 5 éléments donnant la place de chaque objet. Par exemple :

| 01 | 02 | 03 | 04 | 05 |
|---|---|---|---|---|
| 2 | 4 | 1 | 5 | 3 |

  Ce tableau représente le fait que O1 est en place 2, O2 en place 4 etc.

- **Définition du voisinage :**

  Echanger les places de 2 objets i et j quelconques ($i \neq j$)

- **Contenu de la liste TABOU**

  Interdire le mouvement inverse de celui sélectionné. Par exemple si on a échangé les places de i et j on interdit que i revienne à sa place initiale ET que j revienne à sa place initiale.

  Représentation par une matrice donnant pour chaque objet la liste des places tabou.

  Durée des interdiction : fixe ou variable.

  Ici on choisira durée fixe (t=9)

- **Déroulement de la méthode**

  Solution initiale (2, 4, 1, 5, 3) – Coût = 72 (OK – à vérifier)

  **Itération 1 –**

  Solution courante : (2, 4, 1, 5, 3) – Coût = 72

  Meilleure solution : (2, 4, 1, 5, 3) – Coût = 72

  Voisinage :

| Mvts | (1, 2) | (1, 3) | (1, 4) | (1, 5) | (2, 3) | (2, 4) | (2, 5) | (3, 4) | (3, 5) | (4, 5) |
|---|---|---|---|---|---|---|---|---|---|---|
| Aut ? | OK | OK | OK | OK | OK | OK | OK | OK | OK | OK |
| Δ coût | 2 | -12 | -12 | 2 | 0 | -10 | -12 | 4 | 8 | 6 |

  - Parmi tous les mouvements possibles : aucun n'est interdit.
  - Pour tous ces mouvements : on calcule la variation de coût (sans générer la solution) et on choisit le meilleur mouvement. Ici 3 mouvements entraînant une variation de –12. On choisit le mouvement (1, 3)
  - Interdire le mouvement inverse : à savoir remettre 01 en place 2 ET O3 en place 1 pendant 9 itérations.

| | p1 | p2 | p3 | p4 | p5 |
|---|---|---|---|---|---|
| O1 | 0 | 1+9=10 | 0 | 0 | 0 |
| O2 | 0 | 0 | 0 | 0 | 0 |
| O3 | 1+9=10 | 0 | 0 | 0 | 0 |
| O4 | 0 | 0 | 0 | 0 | 0 |
| O5 | 0 | 0 | 0 | 0 | 0 |

  Solution courante : (1, 4, 2, 5, 3) – Coût = 60.

**Itération 2 –**

Solution courante : (1, 4, 2, 5, 3) – Coût = 60 / Meilleure solution : IDEM

Voisinage :

| Mvts | (1, 2) | (1, 3) | (1, 4) | (1, 5) | (2, 3) | (2, 4) | (2, 5) | (3, 4) | (3, 5) | (4, 5) |
|---|---|---|---|---|---|---|---|---|---|---|
| Aut ? | OK | NON | OK | OK | OK | OK | OK | OK | OK | OK |
| Δ coût | 14 | 12 | -8 | 10 | 0 | 10 | 8 | 12 | 12 | 8 |

- Parmi tous les mouvements possibles : un seul est interdit.
- On choisit le meilleur mouvement : ici (1, 4)
- Interdire le mouvement inverse : à savoir remettre 01 en place 1 ET O4 en place 5 pendant 9 itérations.

| | p1 | p2 | p3 | p4 | p5 |
|---|---|---|---|---|---|
| O1 | 2+9=11 | 10 | 0 | 0 | 0 |
| O2 | 0 | 0 | 0 | 0 | 0 |
| O3 | 10 | 0 | 0 | 0 | 0 |
| O4 | 0 | 0 | 0 | 0 | 2+9=11 |
| O5 | 0 | 0 | 0 | 0 | 0 |

Solution courante : (5, 4, 2, 1, 3) – Coût = 52

**Itération 3 –**

Solution courante : (5, 4, 2, 1, 3) – Coût = 52 - Meilleure solution : IDEM

Voisinage :

| Mvts | (1, 2) | (1, 3) | (1, 4) | (1, 5) | (2, 3) | (2, 4) | (2, 5) | (3, 4) | (3, 5) | (4, 5) |
|---|---|---|---|---|---|---|---|---|---|---|
| Aut ? | OK | OK | NON | OK | OK | OK | OK | OK | OK | OK |
| Δ coût | 10 | 24 | 8 | 10 | 0 | 22 | 20 | 8 | 8 | 14 |

- Meilleur mouvement : (2, 3)
- Interdire le mouvement inverse : à savoir remettre 02 en place 4 ET O3 en place 2 pendant 9 itérations.

| | p1 | p2 | p3 | p4 | p5 |
|---|---|---|---|---|---|
| O1 | 11 | 10 | 0 | 0 | 0 |
| O2 | 0 | 0 | 0 | 3+9=12 | 0 |
| O3 | 10 | 3+9=12 | 0 | 0 | 0 |
| O4 | 0 | 0 | 0 | 0 | 11 |
| O5 | 0 | 0 | 0 | 0 | 0 |

Solution courante : (5, 2, 4, 1, 3) – Coût = 52

**Itération 4 –**

Solution courante : (5, 2, 4, 1, 3) – Coût = 52 - Meilleure solution : IDEM

Voisinage :

| Mvts | (1, 2) | (1, 3) | (1, 4) | (1, 5) | (2, 3) | (2, 4) | (2, 5) | (3, 4) | (3, 5) | (4, 5) |
|---|---|---|---|---|---|---|---|---|---|---|
| Aut ? | OK | OK | NON | OK | NON | OK | OK | OK | OK | OK |
| Δ coût | 24 | 10 | 8 | 10 | 0 | 8 | 8 | 22 | 20 | 14 |

- Meilleur mouvement : (2, 4) **➔ Augmente le coût …**
- Interdire le mouvement inverse : à savoir remettre 02 en place 2 ET O4 en place 1 pendant 9 itérations.

| | p1 | p2 | p3 | p4 | p5 |
|---|---|---|---|---|---|
| O1 | 11 | 10 | 0 | 0 | 0 |
| O2 | 0 | 4+9=13 | 0 | 12 | 0 |
| O3 | 10 | 12 | 0 | 0 | 0 |
| O4 | 4+9=13 | 0 | 0 | 0 | 11 |
| O5 | 0 | 0 | 0 | 0 | 0 |

Solution courante : (5, 1, 4, 2, 3) – Coût = 60

**Itération 5 –**

Solution courante : (5, 1, 4, 2, 3) – Coût = 60

Meilleure solution : (5, 2, 4, 1, 3) – Coût = 52

Voisinage :

| Mvts | (1, 2) | (1, 3) | (1, 4) | (1, 5) | (2, 3) | (2, 4) | (2, 5) | (3, 4) | (3, 5) | (4, 5) |
|---|---|---|---|---|---|---|---|---|---|---|
| Aut ? | OK | OK | NON | OK | NON | OK | OK | NON | OK | OK |
| Δ coût | 12 | -10 | 12 | 10 | 0 | -8 | 4 | 14 | 20 | 10 |

- Meilleur mouvement : (1, 3)
- Interdire le mouvement inverse : à savoir remettre 01 en place 5 ET O3 en place 4 pendant 9 itérations.

| | p1 | p2 | p3 | p4 | p5 |
|---|---|---|---|---|---|
| O1 | 11 | 10 | 0 | 0 | 5+9=14 |
| O2 | 0 | 13 | 0 | 12 | 0 |
| O3 | 10 | 12 | 0 | 5+9=14 | 0 |
| O4 | 13 | 0 | 0 | 0 | 11 |
| O5 | 0 | 0 | 0 | 0 | 0 |

Solution courante : (4, 1, 5, 2, 3) – Coût = 50 (ici valeur optimale)