

Table des matières

1 Généralités	11
	11
I Définitions de quelques problèmes classiques d'optimisation combinatoire	12
1.1 Problème du rendu de monnaie	13
1.1.1 Problème	13
1.1.2 Formalisation mathématique	13
1.2 Problème du voyageur de commerce	14
1.2.1 Problème	14
1.2.2 Formalisation mathématique	14
1.2.3 Voisin et voisinage	15
	16
II Heuristiques	17
1.3 Algorithmes gloutons	19
1.3.1 Principe	19
1.3.2 Exemples d'algorithmes gloutons pour le problème du rendu de monnaie	19

1.3.3	Exemples d'algorithmes gloutons pour le problème du voyageur de commerce	22
1.4	Algorithmes gloutons randomisés	30
1.4.1	Principe	30
1.4.2	Exemple d'algorithme glouton randomisé pour le problème du rendu de monnaie	30
1.4.3	Exemple d'algorithme glouton randomisé pour le problème du voyageur de commerce	31
1.5	Recherche locale	31
1.5.1	Principe	31
1.5.2	Exemple de recherche locale pour le problème du voyageur de commerce	34

37

III Métaheuristiques 39

1.6	Procédure de recherche itérative en deux phases (GRASP)	40
1.6.1	Principe	40
1.6.2	Exemple d'algorithme GRASP pour le problème du voyageur de commerce	41
1.7	Recuit simulé	41
1.7.1	Principe	41
1.7.2	Exemple d'algorithme de recuit simulé pour le problème du voyageur de commerce	42
1.8	Recherche taboue	44
1.8.1	Principe	44

	46
IV Méthodes exactes	48
1.9 Programmation linéaire	49
1.9.1 Principe	49
1.9.2 Exemple de programme linéaire pour le problème du rendu de monnaie	50
1.9.3 Exemple de programme linéaire pour le problème du voyageur de commerce	52
	55
V Etude comparative entre les méthodes étudiées en utilisant les données de la Tsplib	57
1.10 Description des données et de l'environnement de développement	58
1.11 Coût des solutions	58
1.11.1 Heuristiques	58
1.11.2 Métaheuristiques	58
1.12 Temps d'exécution	59
1.13 Observations	59
1.14 Gap	61
1.14.1 Instance de taille 131	61
1.14.2 Instance de taille 423	62
1.14.3 Instance de taille 436	62
1.14.4 Instance de taille 662	62
1.15 Complexité	64
VI Arbres et arborescences	66
1.16 Propriétés des arbres	67
1.17 Arborescences	68
1.18 Problème de l'arbre couvrant de poids maximum (maximum spanning tree)	68

1.18.1	L'algorithme de Kruskal	69
1.18.2	L'algorithme de Prim	71
VII	Algorithme de plus court chemin	73
1.19	Algorithme de Dijkstra	74
VIII	Problèmes de flots maximum	77
1.20	Énoncé	78
1.21	Algorithme de Ford - Fulkerson	78
IX	Algorithmes d'approximations pour le problème du voyageur de commerce	79
1.22	Algorithme d'insertion au plus proche	81
2	Définition et notations du système	83
X	Formalisation du système	84
2.1	Introduction	85
2.2	Description générale du système	87
2.2.1	Problème	87
2.2.2	Caractéristiques de l'énergie	88
2.2.3	Critères de qualité	90
2.2.4	Hypothèses	90
2.3	Formalisation du système	91
2.3.1	Système « véhicules »	91
2.3.2	système « production d'énergie »	107
2.4	Évaluation des coûts énergétiques	112
2.5	Programme mathématique du système	112
2.5.1	Programme mathématique du « système véhicules »	112
2.5.2	Programme mathématique du système « production d'énergie » . . .	121
2.5.3	Coût	125

2.6	Perspectives	126
-----	------------------------	-----

Liste des algorithmes

1	Heuristique constructive	18
2	Heuristique d'amélioration	18
3	Rendu de monnaie - glouton	20
4	TSP - heuristique du plus proche voisin	27
5	TSPCHOISIR	27
6	TSP - heuristique d'insertion	28
7	Glouton randomisé	30
8	Rendu de monnaie - glouton randomisé	32
9	RENDREPIECE	33
10	TSP - plus proche voisin randomise	33
11	AJOUTER	34
12	SWITCH	35
13	2-OPT	37
14	TSP - recherche_locale_descente	38
15	Procédure GRASP	40
16	TSP - GRASP	41
17	Recuit simulé	43
18	TSP - recuit	45
19	P	45
20	ProgramDeRecuit	45
21	Kruskal	70
22	Variante Kruskal	71
23	Prim	71

24	TSP - tree	72
25	Dijkstra	75
26	Ford - Fulkerson	78

Table des figures

1.1	Tournée Tour1.	15
1.2	Exemple d'exécution de l'algorithme 3	21
1.3	Exemple d'application de l'heuristique du plus proche voisin et de l'heuristique du voisin le plus éloigné.	22
1.4	Exemple d'application de l'heuristique d'insertion au moindre coût et d'insertion au plus proche.	25
1.5	Exemple d'application de l'heuristique de Clarke et Wright.	26
1.6	Exemple d'exécution de l'algorithme de plus court chemin du TSP.	29
1.7	Exemple de transformation d'une tournée à l'aide du 2-opt	36
1.8	Organigramme de l'algorithme du recuit simulé	44
1.9	Organigramme de l'algorithme de la recherche taboue.	47
1.10	Exemple d'exécution du PLNE du rendu de monnaie.	52
1.11	Exemple d'exécution du PLNE du TSP.	56
1.12	Heuristiques constructives	59
1.13	Heuristiques d'amélioration	60
1.14	Heuristiques (instance de taille maximal que nous pouvons traiter)	61
1.15	Metaheuristiques du TSP	62
1.16	Temps d'exécution obtenu de plusieurs tailles d'instances du TSP	64
1.17	Les poids encadrés des arêtes sont les poids de l'arbre de poids maximum de ce graphe	69
1.18	Avant	70
1.19	Après	70
1.20	Exemple d'application de l'algorithme Kruskal	70

2.1	Illustre les composantes du dépôt. On a la micro-usine qui fabrique de l'hydrogène, la citerne en hydrogène qui alimente les véhicules en hydrogène, les bornes de recharge électrique qui permettent de recharger les véhicules en électricité et les véhicules qui effectueront les tâches de réaliser les requêtes.	86
2.2	Réseau réel représentant l'ensembles des stations (par exemple les stations A,B,C) et les temps de déplacement pour se déplacer d'une station à une autre (par exemple pour se déplacer de A à D un véhicule fera 4 unité de temps). Nous avons deux requêtes associées à ce réseau réel, la première requête doit récupérer 5 unités de charge à A et l'apporter à B.	87
2.3	Illustre une tournée (dessinée en pointillés). Les arcs pleins, les stations et le dépôt représentent le réseau réel. Les requêtes sont : (A,B,10) et (D,C,15)	88
2.4	Division du système global en deux sous-systèmes.	92
2.5	Représentation du tableau des requêtes.	93
2.6	Illustre un pré-traitement effectué sur un réseau réel. Sur le graphe virtuel, chaque petit carré représente une duplication du dépôt, les ronds représentent les stations origines ou destinations et les arcs représentent les déplacements possibles entre les dépôts et les stations. Sur chaque arc on retrouve le temps qui vaut aussi la dépense énergétique du déplacement.	96
2.7	Illustre une situation où on doit synchroniser des activités de logistique/-transport avec des production et consommation d'énergie renouvelable. On transforme le réseau réel en faisant abstraction de certains nœuds qui ne sont ni origine, ni destination et en calculant le plus court chemin d'une origine à une destination. Les arcs représentent les déplacements possibles dans le graphe.	98
2.8	Récapitulatif des inputs et outputs du système « véhicules ».	102
2.9	Illustration d'une liste d'une tournée dont le s ième élément est $i1$ et le $s + 1$ ième élément est $i2$.	105
2.10	Fonction en escalier du rendement en hydrogène.	108
2.11	Récapitulatif des inputs et outputs du système « production d'énergie ».	110
2.12	Illustration des trois cas de figure possible pour $V_{i,j}$.	124

Liste des tableaux

1.1	Instance de taille 131.	63
1.2	Instance de taille 423.	63
1.3	Instance de taille 436.	63
1.4	Instance de taille 662.	63
1.5	Complexité en temps	65
2.1	Tableau représentant les actions qui peuvent se faire simultanément au dé- pôt.	90
2.2	représentation du tableau des labels des stations virtuelles de la figure 2.6, avec $tour = (tour[1], tour[2])$	101

Chapitre 1

Généralités

Nous présenterons dans cette partie quelques généralités.

Première partie

Définitions de quelques problèmes classiques d'optimisation combinatoire

1.1 Problème du rendu de monnaie

1.1.1 Problème

Un système de pièces de monnaie est constitué d'un ensemble de pièces de différentes valeurs. Étant donné un système de pièces de monnaie, nous cherchons à donner à un individu un certain montant en utilisant seulement ces pièces. Quel est le nombre minimal de pièces que nous pouvons utiliser pour rendre ce montant ?

1.1.2 Formalisation mathématique

— Données du problème

Soit $S = (C_1, C_2, \dots, C_n)$ le système de pièces et M le montant à rendre. $\forall i \in \llbracket 1, n \rrbracket$, C_i représente la valeur de la pièce i . Dans un premier temps nous considérons que le nombre de chaque pièce qu'on a à notre disposition est infini.

— Résultats

$\forall i \in \llbracket 1, n \rrbracket$, on cherche x_i la quantité de pièces de type i à rendre. Une solution du problème du rendu de monnaie sera un n -uplets (x_1, x_2, \dots, x_n) .

Exemple 1 Soit $S=(1,3,4)$ le système de pièces c'est-à-dire que $C_1 = 1, C_2 = 3, C_3 = 4$, le montant à rendre est 6. Les solutions possibles sont des 3-uplets. Deux solutions possibles sont $X1=(2,0,1)$, $X2=(0,2,0)$. $X1$ utilise 3 pièces et $X2$ utilise 2 pièces. La solution $X2$ est meilleure car elle utilise le plus petit nombre de pièces.

Exemple 2 Soit $S=(1,16,18,25,30)$ le système de pièces c'est-à-dire que $C_1 = 1, C_2 = 16, C_3 = 18, C_4 = 25, C_5 = 30$, le montant à rendre est 100. Les solutions possibles sont des 5-uplets. Deux solutions possibles sont $X1=(10,0,0,0,3)$, $X2=(100,0,0,0,0)$. $X1$ utilise 13 pièces et $X2$ utilise 100 pièces. La solution $X1$ est meilleure que la solution $X2$ car elle utilise le plus petit nombre de pièces.

1.2 Problème du voyageur de commerce

1.2.1 Problème

Le problème du voyageur de commerce (ou TSP pour Traveling Salesman Problem) est le suivant : un représentant de commerce ayant un nombre fixe de villes à visiter doit planifier sa tournée de manière à passer par toutes les villes en minimisant la distance totale parcourue. Sa tournée doit commencer et se terminer sur la même ville. La notion de distance peut-être remplacée par d'autres notions comme le temps qu'il met ou l'argent qu'il dépense.

1.2.2 Formalisation mathématique

— Données du problème

Soit $G = (V, E, dist)$ un graphe complet, non-orienté et étiqueté sur les arêtes où $v_i \in V$ est l'ensemble des N villes à visiter avec $i \in \llbracket 1, N \rrbracket$. $\forall (v_i, v_j) \in V^2, (v_i, v_j) \in E$ représente la route qui lie la ville v_i à la ville v_j et $dist_{v_i, v_j} \in \mathbb{R}$ est la distance qui sépare la ville v_i de la ville v_j . La matrice des distances $dist$ satisfait l'inégalité triangulaire c'est-à-dire $\forall (v_i, v_j, v_k) \in V^3, dist_{v_i, v_j} \leq dist_{v_i, v_k} + dist_{v_k, v_j}$ car on passera par les villes une unique fois et on reviendra à la ville de départ.

— Résultats

Un tour $t \subseteq E$ est un ensemble de N arêtes qui constitue un cycle hamiltonien dans le graphe G et est défini comme $t = \{(\pi_i, \pi_{(i+1) \bmod N}) \in E \mid 0 \leq i < N, 0 \leq j < N\}$, où π_i est la ville visitée à la i ème position dans le tour. L'espace des solutions S d'une instance du TSP est l'ensemble de tous les tours et la fonction coût $f : S \rightarrow \mathbb{R}$ est définie par $f(t) = \sum_{(i,j) \in t} dist(i, j)$. On cherche un tour t tel que $f(t)$ est minimal.

Exemple 3 Soit $G = (V, E, dist)$ où $V = \{1, 2, 3, 4, 5\}$, $N = 5$ et $\forall i \in \llbracket 1, N \rrbracket, v_i = i$.

La matrice $dist$ est indexée de la ville 1 à 5 sur les lignes et les colonnes :

$$\begin{pmatrix} 0 & 2 & 3 & 4 & 1 \\ 2 & 0 & 2 & 4 & 2 \\ 3 & 2 & 0 & 3 & 5 \\ 4 & 4 & 3 & 0 & 7 \\ 1 & 2 & 5 & 7 & 0 \end{pmatrix}$$

un exemple de tournées : $Tour1 = \{(v_1, v_5), (v_5, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1)\}$. La tournée construite est représentée par la figure 1.1. La distance parcourue par $Tour1$ est :

$$f(Tour1) = dist_{v_1, v_5} + dist_{v_5, v_2} + dist_{v_2, v_3} + dist_{v_3, v_4} + dist_{v_4, v_1} = 1 + 2 + 2 + 3 + 4 = 12.$$

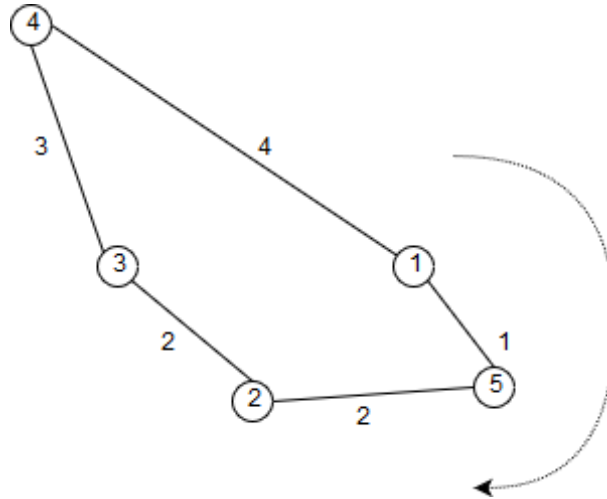


FIGURE 1.1 – Tournée Tour1.

1.2.3 Voisin et voisinage

Le voisin S' d'une solution S est une solution qu'on obtient en appliquant à S un certain opérateur. Un opérateur sert à modifier une solution localement. En utilisant $Tour1 = \{(v_1, v_5), (v_5, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1)\}$ construit à l'exemple 3 et SWITCH un opérateur qui permute deux villes placées en paramètres, on obtient comme voisins de $Tour1$:

- $Tour1' = \{(v_5, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5)\}$ Si les villes permutées sont 1 et 5.

- $Tour1' = \{(v_2, v_5), (v_5, v_1), (v_1, v_3), (v_3, v_4)(v_4, v_2)\}$ Si les villes permutées sont 1 et 2.
- $Tour1' = \{(v_3, v_5), (v_5, v_2), (v_2, v_1), (v_1, v_4)(v_4, v_3)\}$ Si les villes permutées sont 1 et 3.
- $Tour1' = \{(v_4, v_5), (v_5, v_2), (v_2, v_3), (v_3, v_1)(v_1, v_4)\}$ Si les villes permutées sont 1 et 4.
- $Tour1' = \{(v_1, v_2), (v_2, v_5), (v_5, v_3), (v_3, v_4)(v_4, v_1)\}$ Si les villes permutées sont 2 et 5.
- $Tour1' = \{(v_1, v_3), (v_3, v_2), (v_2, v_5), (v_5, v_4)(v_4, v_5)\}$ Si les villes permutées sont 5 et 3.

Le voisinage d'une solution est l'ensemble de ses voisins. Le voisinage d'une solution contient toutes les solutions que l'on obtient en modifiant localement celle-ci à l'aide d'un opérateur donné.

Deuxième partie

Heuristiques

Algorithme 1 Heuristique constructive

Entrées: α : données pour construire une solution**Sorties:** S : solution

- 1: $S \leftarrow \emptyset$
 - 2: **Tant que** S n'est pas une solution **faire**
 - 3: Choisir un nouvel élément de solution e //construire e à l'aide de α
 - 4: $S \leftarrow S + e$
 - 5: **Fin tant que**
 - 6: Retourner S
-

Algorithme 2 Heuristique d'amélioration

Entrées: S' : solution**Sorties:** S : solution

- 1: $S \leftarrow S'$
 - 2: $nonstop \leftarrow 1$
 - 3: **Tant que** $nonstop == 1$ et S n'est pas améliorée **faire**
 - 4: **Si** critère d'arrêt **alors**
 - 5: $nonstop \leftarrow 0$
 - 6: **sinon**
 - 7: $S \leftarrow \text{Transformer}(S)$
 - 8: **Fin si**
 - 9: **Fin tant que**
 - 10: Retourner S
-

Les heuristiques ou méthodes approximatives conduisent (en temps polynomial) à une solution mais pas nécessairement à une solution optimale. Il existe deux types d'heuristiques : les heuristiques constructives et les heuristiques d'amélioration. Les heuristiques constructives construisent une solution tandis que, les heuristiques d'amélioration construisent une solution puis l'améliore. Le pseudo code d'une heuristique constructive est donné par l'algorithme 1 et celui d'une heuristique d'amélioration est donné par l'algorithme 2.

Dans les algorithmes 1 et 2, α est une donnée fournit par l'utilisateur qui servira à construire une solution S . Par exemple dans le cas du TSP, α représente la matrice des distances (matrice contenant la distance entre toutes les paires de villes) et S représente la tournée construite. Dans l'algorithme 1, si nous sommes dans le cas du TSP, le nouvel élément de solution e choisit est une ville. La transformation effectuée sur S dans l'algorithme 2 consiste à sélectionner un voisin de S .

1.3 Algorithmes gloutons

1.3.1 Principe

Soit (P) un problème qu'on cherche à résoudre c'est-à-dire que nous cherchons la meilleure solution (solution optimale) ou une solution dont la valeur se rapproche de l'optimale. Un algorithme glouton est une méthode intuitive qui construit une solution de (P) étape par étape en faisant un choix. Les choix effectués à chaque étape ne sont jamais remis en question. Autrement dit, la méthode gloutonne est une heuristique constructive qui fait une succession de choix optimaux localement et ce sans retour en arrière possible.

1.3.2 Exemples d'algorithmes gloutons pour le problème du rendu de monnaie

Idées d'heuristiques

Dans le cas du problème du rendu de monnaie, les algorithmes gloutons sont :

- Choix de la pièce ayant la plus grande valeur.

Une solution à ce problème serait de choisir à chaque étape la pièce qui a la plus grande valeur et qui est inférieur au montant restant à rendre. En reprenant l'exemple 2, la solution X1 serait celle qu'on obtiendrait avec cette méthode donc on utiliserait 13 pièces.

- Choix de la pièce ayant la plus petite valeur.

Pour résoudre ce problème nous pouvons choisir à chaque étape la pièce qui a la plus petite valeur. Mais si nous avons par exemple une pièce de valeur 1 alors le nombre de pièces utilisées sera maximal, or notre but est de minimiser ce nombre. En reprenant l'exemple 2, la solution X2 serait celle qu'on obtiendrait avec cette méthode donc on utiliserait 100 pièces. Dans l'exemple 2 cette technique a été utilisée pour trouver la solution (2,0,1), or elle n'est pas la meilleure solution car la solution (0,2,0) utilise deux pièces et elle trois pièces.

- Nous pouvons alterner entre le choix d'une pièce de plus grande valeur et d'une pièce de plus petite valeur jusqu'à atteindre le montant voulu. Donc à la première itération on sélectionne la pièce ayant la plus grande valeur puis à l'itération suivante on

sélectionne la pièce ayant la plus petite valeur et on itère ce procédé jusqu'à obtenir la montant voulu.

Les algorithmes gloutons donnent très rarement les solutions optimales. Par exemple, en prenant le cas de l'exemple 2, on remarque que la solution $(0,0,0,4,0)$ est meilleure que celles que nous avons trouvé jusqu'ici.

Hypothèses

Avant d'écrire l'algorithme glouton dont le principe est de choisir à chaque étape la pièce qui a la plus grande valeur nous allons faire certaines hypothèses qui nous simplifient l'écriture de l'algorithme.

- Les pièces sont rangées dans l'ordre croissant car ceci nous aidera lors de la sélection de la pièce ayant la plus grande valeur.
- M est un nombre entier.
- $\exists i \in \llbracket 1, n \rrbracket, C_i = 1$, avec cette hypothèse on est sûr de pouvoir rendre le montant M .

Algorithme

Algorithme 3 Rendu de monnaie - glouton

Entrées: M : montant à rembourser,

n : nombre de valeurs différentes pour les pièces,

$C[n]$: valeurs des pièces

Sorties: $X[n]$: quantité de chaque pièce sélectionnée,

Nb_{piece} : nombre de pièces total

1: $Nb_{piece} \leftarrow 0$

2: **pour** $i = 1$ **à** n **faire**

3: $X[i] \leftarrow 0$

4: **fin pour**

5: $reste \leftarrow M$

6: **Tant que** $reste > 0$ **faire**

7: Choisir la plus grande valeur de i telle que $C[i] \leqslant reste$

8: $reste \leftarrow reste - C[i]$

9: $X[i] \leftarrow X[i] + 1$

10: $Nb_{piece} \leftarrow Nb_{piece} + 1$

11: **Fin tant que**

12: Retourner $X[n], Nb_{piece}$

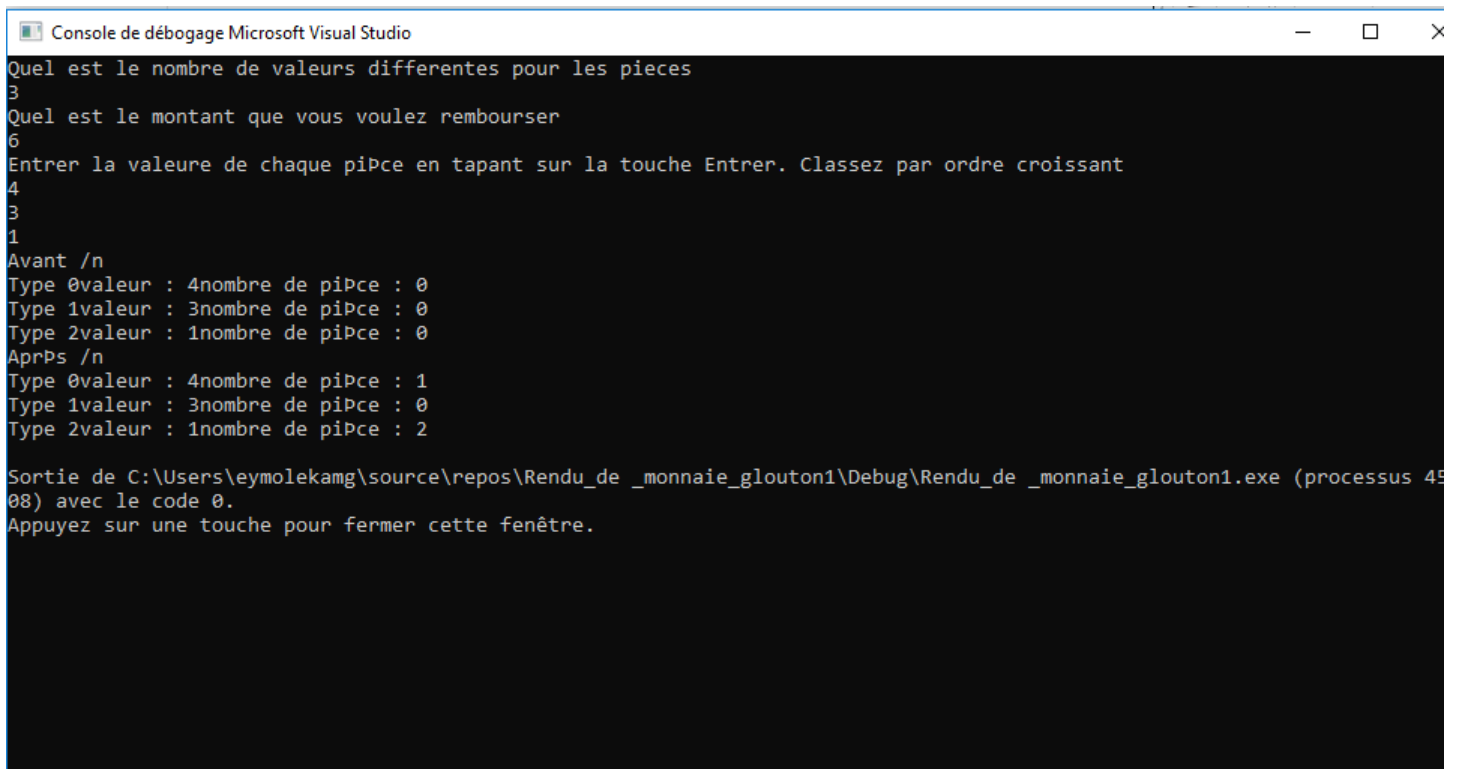
L'algorithme 3 décrit l'algorithme glouton qui consiste à choisir à chaque étape la pièce qui a la plus grande valeur jusqu'à atteindre le montant M . Dans cet algorithme

nous considérons les hypothèses posées dans la section précédente.

Expérimentation

En ce qui concerne l'implémentation de l'algorithme 3 nous avons créé trois fichiers : *rendu_monnaie.h*, *rendu_de_monnaie.cpp* et *main.cpp*.

- *rendu_monnaie.h* contient la classe *Rendu_de_monnaie* qui déclare les attributs (entrées et sorties) dont nous aurons besoin et la signature des fonctions dont nous aurons besoin.
- *rendu_de_monnaie.cpp* contient le code source des fonctions déclarées dans *rendu_monnaie.h*.
- *main.cpp* appelle les fonctions pour déterminer la quantité de chaque pièce à rembourser. La figure 1.2 présente un exemple d'exécution du programme. Nous constatons que la solution fournie par l'algorithme 3 n'est pas la meilleure solution car (0,2,0) est meilleure.



```
Console de débogage Microsoft Visual Studio
Quel est le nombre de valeurs différentes pour les pièces
3
Quel est le montant que vous voulez rembourser
6
Entrer la valeur de chaque pièce en tapant sur la touche Entrer. Classez par ordre croissant
4
3
1
Avant /n
Type 0valeur : 4nombre de pièce : 0
Type 1valeur : 3nombre de pièce : 0
Type 2valeur : 1nombre de pièce : 0
Après /n
Type 0valeur : 4nombre de pièce : 1
Type 1valeur : 3nombre de pièce : 0
Type 2valeur : 1nombre de pièce : 2
Sortie de C:\Users\eymolekam\source\repos\Rendu_de_monnaie_glouton1\Debug\Rendu_de_monnaie_glouton1.exe (processus 4508) avec le code 0.
Appuyez sur une touche pour fermer cette fenêtre.
```

FIGURE 1.2 – Exemple d'exécution de l'algorithme 3

1.3.3 Exemples d’algorithmes gloutons pour le problème du voyageur de commerce

Idées d’heuristiques

Dans cette section, on appellera tour trivial un tour composé de deux villes uniquement.

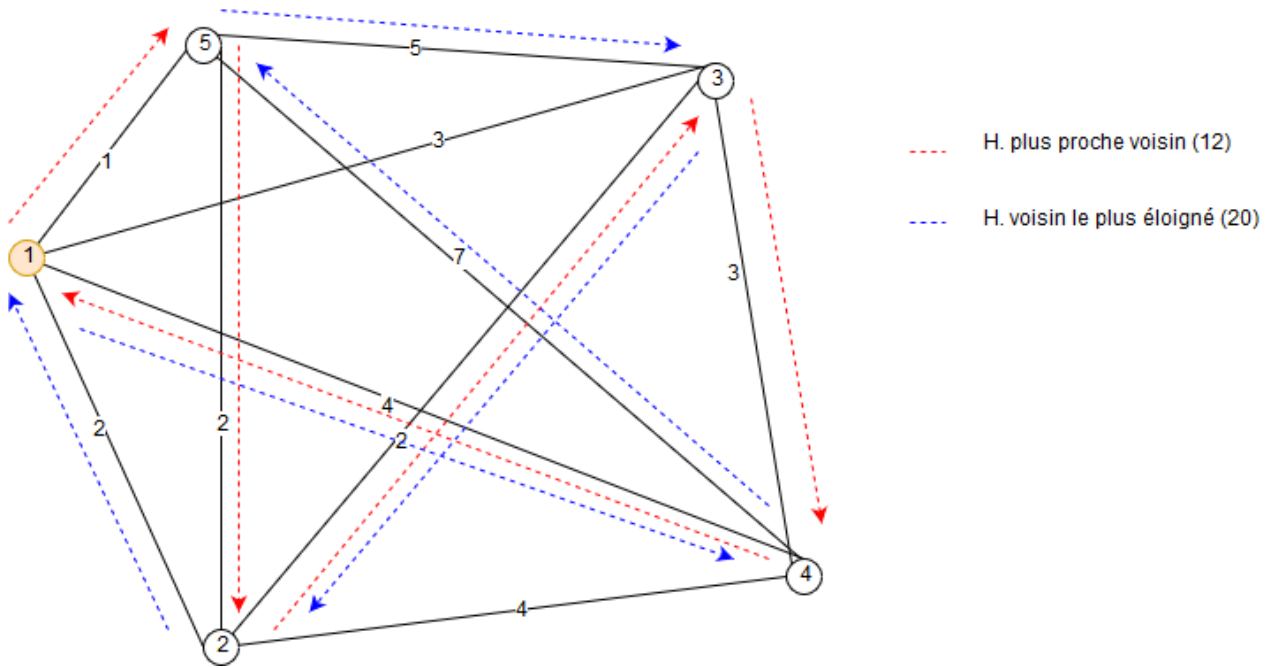


FIGURE 1.3 – Exemple d’application de l’heuristique du plus proche voisin et de l’heuristique du voisin le plus éloigné.

1. Heuristique du plus proche voisin.

On commence à la ville de départ et à chaque étape, pour choisir la nouvelle ville à visiter, on sélectionnera la plus proche (la ville qui se trouve à la plus petite distance de la ville courante) et qui n’a pas encore été visitée. On revient à la ville de départ lorsqu’on a parcouru toutes les villes.

En reprenant l’exemple 3 et en commençant le tour par la ville 1, pour choisir la prochaine ville à visiter on a le choix entre les villes $\{2, 3, 4, 5\}$, on va choisir la ville la plus proche de la ville 1. D’après la matrice des distances, $dist(1, 2) = 2$, $dist(1, 3) = 3$, $dist(1, 4) = 4$ et $dist(1, 5) = 1$ donc la ville la plus proche de la ville 1 est la ville 5. Donc après la ville 1 on visitera la ville 5. Partant de la ville 5 on recommence le processus en choisissant la prochaine ville à visiter parmi les villes $\{2, 3, 4\}$. Le tour

trouvé avec cette heuristique est $\{(v_1, v_5), (v_5, v_2), (v_2, v_3), (v_3, v_4)(v_4, v_1)\}$ et le coût vaut 12 (voir figure 1.3).

2. Heuristique du voisin le plus éloigné.

On commence à la ville de départ et à chaque étape, pour choisir la nouvelle ville à visiter, on choisit la ville la plus éloignée et qui n'a pas encore été visitée. On rentre à la ville de départ lorsqu'on a parcouru toutes les villes.

En reprenant l'exemple 3 et en commençant le tour par la ville 1, pour choisir la prochaine ville à visiter on a le choix entre les villes $\{2, 3, 4, 5\}$, on va choisir la ville la plus éloignée de la ville 1. D'après la matrice des distances, $dist(1, 2) = 2$, $dist(1, 3) = 3$, $dist(1, 4) = 4$ et $dist(1, 5) = 1$ donc la ville la plus éloignée de la ville 1 est la ville 4. Donc après la ville 1 on visitera la ville 4. Partant de la ville 4 on recommence le processus en choisissant la prochaine ville à visiter parmi les villes $\{2, 3, 5\}$. Le tour trouvé avec cette heuristique est $\{(v_1, v_4), (v_4, v_5), (v_5, v_3), (v_3, v_2)(v_2, v_1)\}$ et le coût vaut 20 (voir figure 1.3).

3. Heuristique d'insertion.

— Insertion au moindre coût (respectivement au coût maximal).

Partir d'une tournée triviale puis ajouter les autres villes une par une en les insérant à chaque fois entre les deux villes de la tournée telle que l'augmentation du coût de la tournée soit minimale (respectivement maximale). On s'arrête lorsqu'il n'y a plus de ville à ajouter.

En reprenant l'exemple 3 et en commençant avec la tournée $\{(v_1, v_5), (v_5, v_1)\}$, il reste les villes $\{2, 3, 4\}$ à ajouter dans le tour pour obtenir un tour contenant toutes les villes. On commence par la ville 2, elle sera insérée soit entre la ville 5 et la ville 1, soit entre la ville 1 et la ville 5 en fonction de la position qui minimise le coût du tour obtenu. Le coût du tour $\{(v_1, v_2), (v_2, v_5), (v_5, v_1)\}$ est 5 et le coût du tour $\{(v_1, v_5), (v_5, v_2), (v_2, v_1)\}$ est 5 donc peu importe la position de la ville 2 les tours obtenus ont le même coût. On choisit d'insérer la ville 2 entre la ville 1 et la ville 5, la nouvelle tournée est donc $\{(v_1, v_2), (v_2, v_5), (v_5, v_1)\}$ et il reste les villes $\{3, 4\}$ à insérer dans la nouvelle tournée, pour le faire on procédera comme pour la ville 2. La tournée complète obtenue est

$\{(v_1, v_4), (v_4, v_3), (v_3, v_2), (v_2, v_5), (v_5, v_1)\}$ et le coût vaut 12 (voir figure 1.4).

— Insertion au plus proche.

On construit un tour trivial puis on choisit à chaque étape la ville la plus proche du tour pour l'ajouter dans le tour. On insère le nœud le plus proche du tour, pour cela, on insère la ville dont la distance à une ville du tour est minimale. La distance d'une ville à une tournée est la plus petite distance entre cette ville et les villes de la tournée.

En reprenant l'exemple 3 et en commençant par le tour trivial $\{(v_1, v_5), (v_5, v_1)\}$, il reste les villes $\{2, 3, 4\}$ à insérer, pour le faire, dans un premier temps on calcule les distances des villes $\{1, 5\}$ aux villes $\{2, 3, 4\}$ en choisissant la distance minimale à la tournée. On obtient $\text{Min}\{\text{Min}\{\text{dist}(2, 1), \text{dist}(2, 5)\}, \text{Min}\{\text{dist}(3, 1), \text{dist}(3, 5)\}, \text{Min}\{\text{dist}(4, 1), \text{dist}(4, 5)\}\} = \text{Min}\{\text{dist}(2, 1), \text{dist}(3, 1), \text{dist}(4, 1)\} = \text{dist}(2, 1)$ donc la nouvelle tournée sera $\{(v_1, v_5), (v_5, v_2), (v_2, v_1)\}$. En utilisant le même procédé il faut insérer l'une des villes $\{3, 4\}$ à la nouvelle tournée.

Le tour trouvé avec cette heuristique est $\{(v_1, v_5), (v_5, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1)\}$ et le coût vaut 12 (voir figure 1.4).

— Insertion au plus loin.

On construit un tour trivial puis on choisit à chaque étape la ville la plus loin du tour pour l'ajouter dans le tour (on insère la ville la plus loin du tour). Pour l'insertion, on peut insérer la ville dont la distance minimale à une ville du tour est maximale.

4. Heuristique de sélection gloutonne d'arêtes.

Il s'agit d'un algorithme glouton, où l'on sélectionne les paires de villes (arêtes) une à une, sans nécessairement avoir une chaîne à tout moment. On commence donc par l'arête (paire de villes) de poids le plus faible, puis on sélectionne l'arête de poids minimal qui ne crée pas de sous-cycle, et ainsi de suite jusqu'à obtenir une tournée qui contient toutes les villes. Une ville apparaît dans maximum deux paires de villes sélectionnées. Cette procédure devient un peu plus complexe dans le cas de graphes non complets.

En reprenant l'exemple 3, La première paire de ville sélectionnée sera (1,5) car

$dist(1, 5)$ est la plus petite distance de la matrice des distances. Les prochaines plus petites distances de la matrice des distances sont $dist(1, 2)$, $dist(2, 3)$ et $dist(2, 5)$, on choisit la paire (1,2). En suivant le même procédé on choisit (3,4), puis (2,4). Le tour trouvé avec cette heuristique est $\{(v_5, v_1), (v_1, v_2), (v_2, v_4), (v_4, v_3), (v_3, v_5)\}$ et le coût vaut 15 (voir figure 1.4).

5. Heuristique de Clarke et Wright.

On choisit une ville $v_i \in V$ qu'on suppose être la ville de départ et on crée $n-1$ tours triviales : des boucles (de la ville v_i au $n - 1$ autres villes restantes, voir figure 1.5). On procède ensuite à $n-2$ étapes de fusion où l'on prend deux tours que l'on fusionne en un. Une telle fusion permet d'éviter un aller-retour au sommet v_i et diminue la somme des longueurs des tours. L'algorithme consiste à effectuer à chaque fois la fusion qui permettra de diminuer le plus possible la somme de ces longueurs.

On fusionne les tournées en connectant la dernière ville visitée d'une sous-tournée à la première ville visitée d'une autre sous-tournée telles que la distance parcourue soit minimale. Une sous-tournée est une tournée qui ne contient pas toutes les villes. On s'arrête lorsqu'on obtient une tournée qui contient toutes les villes.

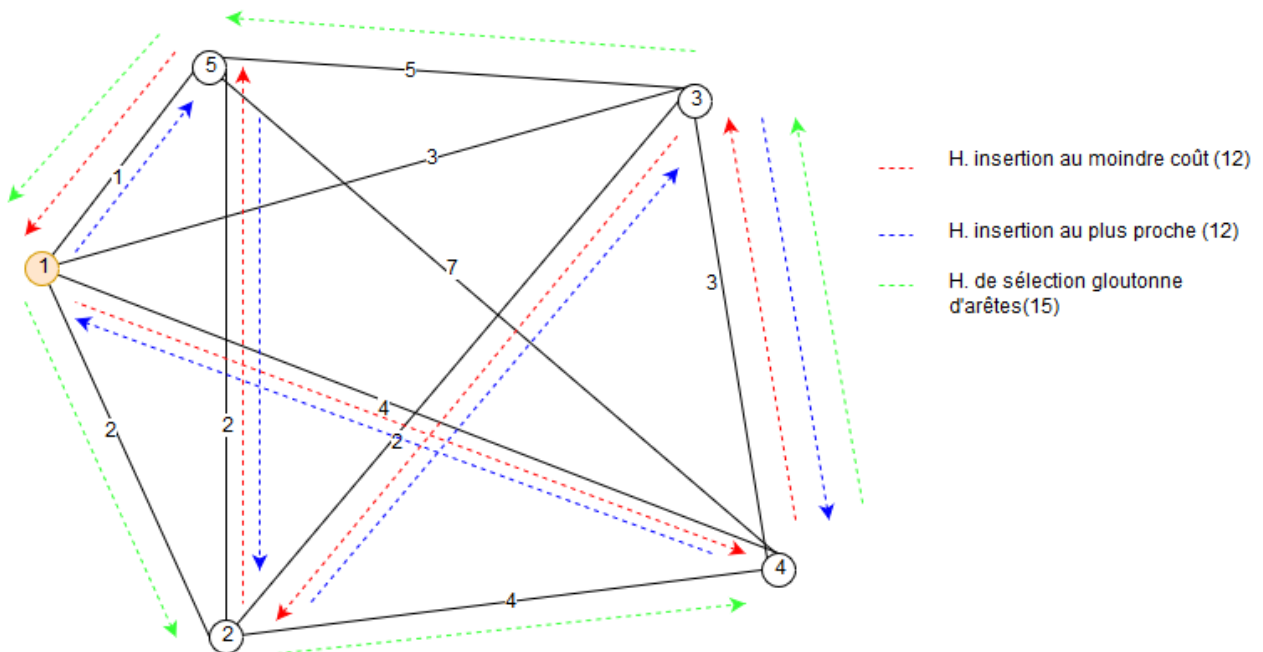


FIGURE 1.4 – Exemple d'application de l'heuristique d'insertion au moindre coût et d'insertion au plus proche.

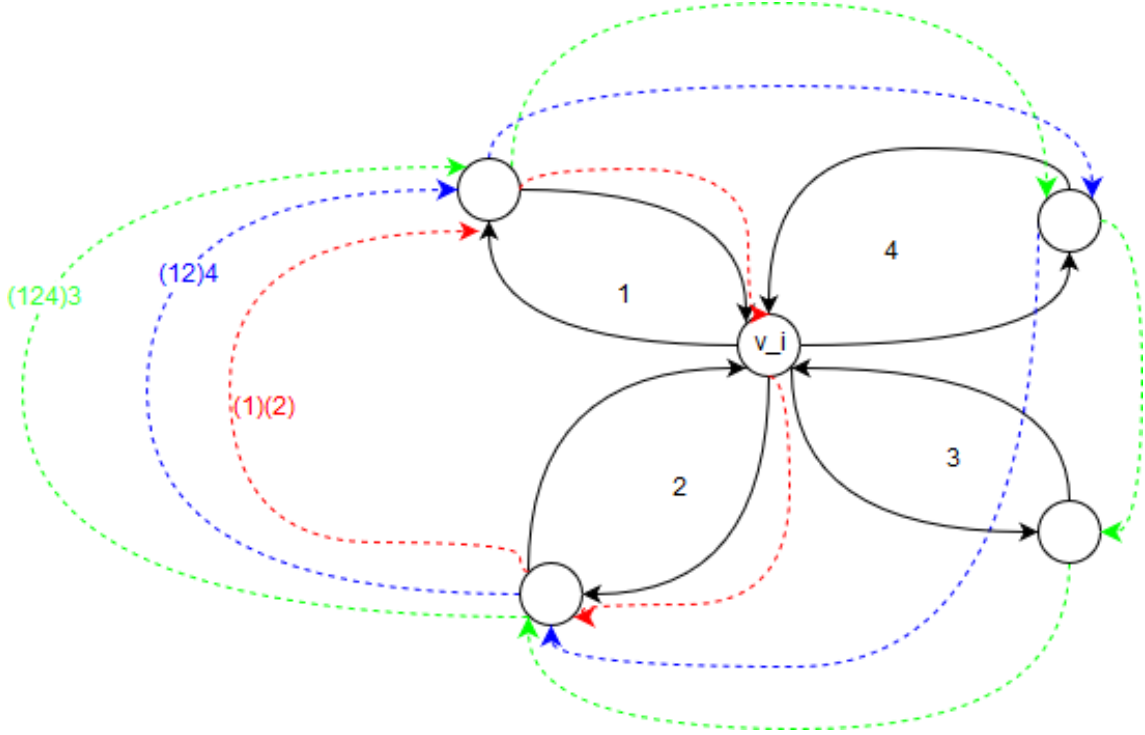


FIGURE 1.5 – Exemple d’application de l’heuristique de Clarke et Wright.

Algorithmes

Soit N le nombre de villes à visiter ($N \geq 2$) et $dist$ une matrice de taille $N * N$ contenant les distances entre toutes les paires de villes. $\forall (i, j) \in \llbracket 0, N - 1 \rrbracket^2$, $dist[i][j]$ est la distance de la ville i à la ville j . Nous représenterons la tournée en notant le successeur et l’antécédent de chaque ville : $\forall j \in N$, $succ[j]$ est le successeur de la ville j , $ante[j]$ est l’antécédent de la ville j dans la tournée. Par exemple si nous représenterons la tournée 0120 où $\{0, 1, 2\} \in V$ et $\{(0, 1), (1, 2), (2, 0)\} \in E$, on obtiendrait $succ[0] = 1$, $succ[1] = 2$, $succ[2] = 0$, $ante[0] = 2$, $ante[1] = 0$ et $ante[2] = 1$.

$_ville_visit[j]$ permet de savoir si la ville j a déjà été visitée :

$$_ville_visit[j] = \begin{cases} 1 & \text{si la ville } j \text{ a déjà été visitée} \\ -1 & \text{sinon.} \end{cases}$$

L’algorithme 4 est l’heuristique du plus proche voisin et l’algorithme 6 est l’heuristique d’insertion. La ville 0 est la ville de départ.

La procédure choisir *indicemin* de la ligne 12 de l’algorithme 4 est donnée par l’algorithme 5.

Algorithme 4 TSP - heuristique du plus proche voisin

Entrées: N : Nombre de villes,

$dist[N][N]$: matrice des distances

Sorties: $succ[N]$: successeur de chaque ville,

$ante[N]$: antécédent de chaque ville

- 1: initialiser le vecteur $_ville_visit[N]$ à -1 // ce vecteur enregistre si une ville a été visitée
 - 2: $villeCourante \leftarrow 0$
 - 3: $_ville_visit[0] \leftarrow 1$
 - 4: $compteur \leftarrow 1$
 - 5: $_Nb_villes \leftarrow N - 1$
 - 6: **Tant que** $compteur \leq N$ **faire**
 - 7: **TSPCHOISIR** ($N, dist, villeCourante, _ville_visit$)
 - 8: $succ[villeCourante] \leftarrow indicemin$
 - 9: $ante[indicemin] \leftarrow villeCourante$
 - 10: $_ville_visit[indicemin] \leftarrow 1$
 - 11: $villeCourante \leftarrow indicemin$
 - 12: $compteur \leftarrow compteur + 1$
 - 13: **Fin tant que**
 - 14: $succ[villeCourante] \leftarrow 0$ //retour à la ville de départ
 - 15: $ante[0] \leftarrow villeCourante$
 - 16: Retourner $succ, ante$
-

Algorithme 5 TSPCHOISIR

Entrées: N : nombre de villes,

$dist[N][N]$: matrice des distances,

$villeCourante$: indice de la ville courante,

$_ville_visit[N]$: état des villes (villes visitées ou non)

Sorties: $indicemin$: indice de la ville non visitée la plus proche de la ville $villeCourante$

- 1: $min \leftarrow _infini$
 - 2: **pour** $villeC=0$ à $N-1$ **faire**
 - 3: **Si** $_ville_visit[villeC] \neq 1$ **alors**
 - 4: **Si** $dist[villeCourante][villeC] < min$ **alors**
 - 5: $indicemin \leftarrow villeC$
 - 6: **Fin si**
 - 7: **Fin si**
 - 8: **fin pour**
 - 9: Retourner $indicemin$
-

Algorithme 6 TSP - heuristique d'insertion

Entrées: N : nombre de villes,

$dist[N][N]$: distance entre toutes les paires de villes,

Sorties: $succ[N]$: successeur de chaque ville

$ante[N]$: antécédent de chaque ville

1: $succ[1] \leftarrow 2$

2: $ante[2] \leftarrow 1$

3: **pour** $i = 3$ à N **faire**

4: $min \leftarrow \infty$

5: $indice \leftarrow -1$

6: **pour** $j=1$ à $i-1$ **faire**

7: $S \leftarrow succ[j]$

8: **Si** $dist[i][j] + dist[i][S] - dist[j][S] < min$ **alors**

9: $indice \leftarrow j$ //chercher quel sera l'antécédent de la ville i qui minimisera la distance parcourue

10: $min \leftarrow dist[i][j] + dist[i][S] - dist[j][S]$

11: **Fin si**

12: **fin pour**

13: $P \leftarrow succ[indice]$

14: $succ[i] \leftarrow succ[indice]$

15: $succ[indice] \leftarrow i$

16: $ante[i] \leftarrow indice$

17: $ante[P] \leftarrow i$

18: **fin pour**

19: Retourner $succ, ante$

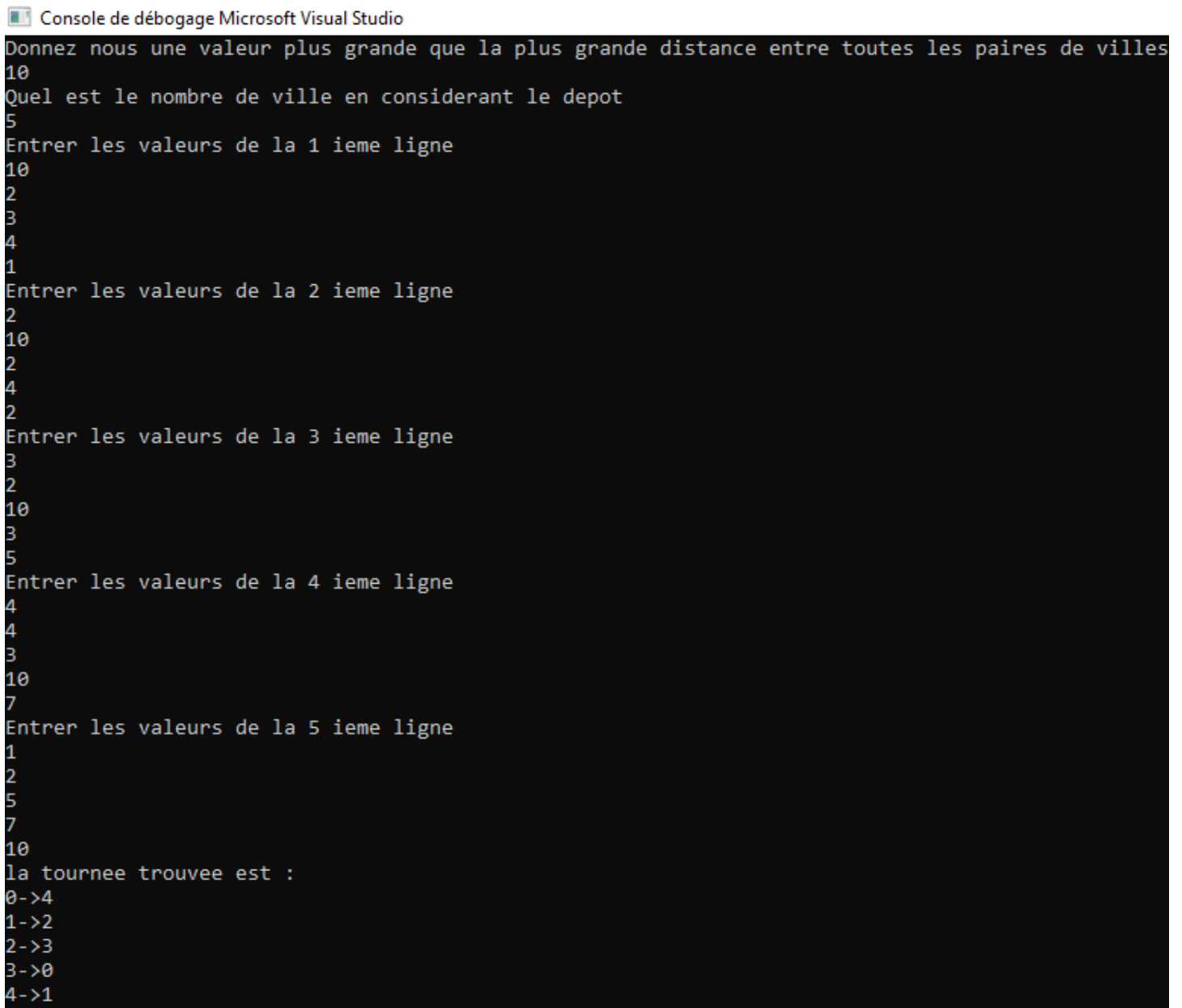
Implémentation

En ce qui concerne l'implémentation de l'algorithme 4 nous avons créé trois fichiers :

Tsp.h, *Tsp.cpp* et *main.cpp*. La figure 1.6 présente un exemple d'exécution du programme.

— *Tsp.h*

— *main.cpp*



```
Console de débogage Microsoft Visual Studio
Donnez nous une valeur plus grande que la plus grande distance entre toutes les paires de villes
10
Quel est le nombre de ville en considerant le depot
5
Entrer les valeurs de la 1 ieme ligne
10
2
3
4
1
Entrer les valeurs de la 2 ieme ligne
2
10
2
4
2
Entrer les valeurs de la 3 ieme ligne
3
2
10
3
5
Entrer les valeurs de la 4 ieme ligne
4
4
3
10
7
Entrer les valeurs de la 5 ieme ligne
1
2
5
7
10
la tournée trouvée est :
0->4
1->2
2->3
3->0
4->1
```

FIGURE 1.6 – Exemple d'exécution de l'algorithme de plus court chemin du TSP.

Nous avons implémenté l'algorithme d'insertion 6.

1.4 Algorithmes gloutons randomisés

1.4.1 Principe

Avec une algorithmes glouton, on effectue un choix à chaque étape (voir section 1.3), ce choix est déterministe. Dans un algorithme glouton randomisé, ce choix est aléatoire, on choisit un élément aléatoirement parmi une liste de candidats possible (noté *Listecandidat*). La liste de candidats est constituée à l'aide d'une heuristique H . Par exemple, dans le TSP, en considérant que nous sommes sur le sommet de départ et que l'heuristique H est l'heuristique du plus proche voisin, la liste *Listecandidat* à cette étape représente la liste des sommets qui peuvent être visités après le sommet de départ et cette liste peut être constituée des deux sommets les plus proches du sommet départ.

Dans un algorithme glouton randomisé, à chaque étape on choisira aléatoirement un élément parmi les éléments possibles de la liste *Listecandidat*. L'algorithme 7 décrit le processus général des algorithmes gloutons randomisés, pour plus de précision consulter [siarry__metaheuristiques__2014]. Le paramètre α représente les données nécessaires pour construire la solution S . Par exemple, dans le TSP, α représente la matrice des distances et S la tournée construite.

Algorithme 7 Glouton randomisé

Entrées: α : données pour construire une solution

Sorties: S : solution

$S \leftarrow \emptyset$

Tant que la solution S n'est pas complète **faire**

 Construire la liste *Listecandidat* à l'aide de l'heuristique H et de α
 choisir aléatoirement un élément s_h dans la liste *Listecandidat*

$S \leftarrow S \cup \{s_h\}$

 Mettre à jour *Listecandidat*

Fin tant que

Retourner S

1.4.2 Exemple d'algorithme glouton randomisé pour le problème du rendu de monnaie

Ici, nous considérons que le nombre de chaque type de pièces que nous avons est fini. Soit N le vecteur contenant le nombre de chaque type de pièces dont on dispose : $\forall i \in \llbracket 1, n \rrbracket, N_i$ représente le nombre de pièces de type i dont on dispose. Dans l'algorithme

8, au départ, l'ensemble des pièces remboursées est vide. Ensuite, on construit la liste *Listecandidat* qui sera constituée des deux pièces ayant les plus grandes valeurs inférieures au montant restant à rembourser, on choisit au hasard une pièce de la liste *Listecandidat* et on ajoute cette pièce dans la liste des pièces à rembourser. La solution est le nombre de chaque type de pièces remboursées.

1.4.3 Exemple d'algorithme glouton randomisé pour le problème du voyageur de commerce

Une ville j est la ville la plus proche d'une ville i si la distance $dist(i, j)$ est la plus petite valeur de la ligne i de la matrice des distances $dist$. L'algorithme 10 décrit une heuristique gloutonne randomisée pour le problème du voyageur de commerce. Dans cette heuristique, on commence la tournée sur une ville de départ qu'on appellera ville 1, pour choisir la prochaine ville à visiter on construit la liste *Listecandidat* qui est constituée des deux villes les plus proches de la ville 1 et qui n'ont pas encore été visitées, on choisit la prochaine ville à visiter aléatoirement parmi les villes de la liste *Listecandidat*. On recommence le processus en prenant comme ville de départ la dernière ville visitée. En sortie de l'algorithme 10, on a la tournée construite.

Le paramètre $\beta \in [0, 1]$ est utilisé pour choisir aléatoirement entre 2 villes A et B dans l'algorithme 10. Par exemple, si $\beta = 0.5$ alors on a 50% de chance de choisir A et 50% de chance de choisir B . Si $\beta = 0.7$, on 30% de chance de choisir A et 70% de chance de choisir B .

1.5 Recherche locale

1.5.1 Principe

La recherche locale est un mécanisme qui vise à améliorer une solution existante d'un problème en explorant le voisinage de cette solution pour voir s'il contient une meilleure solution (solution plus proche de l'optimum). Le critère d'arrêt lors de cette exploration est défini dès le départ par une stratégie de décision.

Comme exemples de stratégies de décisions on peut citer par exemple : la descente, la plus grande descente, la marche aléatoire, etc.

Algorithme 8 Rendu de monnaie - glouton randomisé

Entrées: M : montant à rembourser,

n : nombre de valeurs différentes pour les pièces,

$C[n]$: valeurs des pièces,

$N[n]$: quantité de chaque type de pièces,

β : probabilité

Sorties: $X[n]$: quantité de chaque pièce sélectionnée,

$Nbpiece$: nombre de pièces total

$Nbpiece \leftarrow 0$; $max1 \leftarrow 0$; $max2 \leftarrow 0$

$i1 \leftarrow -1$; $i2 \leftarrow -1$

$hasard \leftarrow -1$

pour $i = 1$ à n **faire**

$X[i] \leftarrow 0$

fin pour

$reste \leftarrow M$

Tant que $reste > 0$ **faire**

pour $i = 1$ à n **faire**

Si $N[i] - X[i] > 0$ et $C[i] \leq reste$ **alors**

Si $C[i] > max1$ **alors**

$i1 \leftarrow i$

$max1 \leftarrow C[i1]$ //On cherche les deux pièces de plus grande valeurs qu'on peut rembourser ($max1 \geq max2$)

$i2 \leftarrow i1$

$max2 \leftarrow C[i2]$

sinon Si $C[i] > max2$ **alors**

$max2 \leftarrow C[i]$

$i2 \leftarrow i$

Fin si

Fin si

fin pour

Si $i1 \neq -1$ et $i2 \neq -1$ **alors**

$hasard \leftarrow random()$ //random() est une fonction qui renvoie un nombre aléatoire entre 0 et 1

Si $hasard < \beta$ **alors**

RENDREPIECE($n, X, reste, i1, C, Nbpiece$)

sinon

RENDREPIECE($n, X, reste, i2, C, Nbpiece$)

Fin si

sinon

Si $i1 \neq -1$ **alors**

RENDREPIECE($n, X, reste, i1, C, Nbpiece$)

sinon Si $i2 \neq -1$ **alors**

RENDREPIECE($n, X, reste, i2, C, Nbpiece$)

sinon

 Retourner "impossible de construire une solution"

Fin si

Fin si

$max1 \leftarrow 0$

$max2 \leftarrow 0$

$i1 \leftarrow -1$

$i2 \leftarrow -1$

Fin tant que

Retourner $X[n]$, $Nbpiece$

Algorithme 9 RENDREPIECE

Entrées: n : nombre de valeurs différentes pour les pièces,

$X[n]$: quantité de chaque pièce sélectionnée,

$reste$: montant restant à rembourser,

i : indice de la pièce à rembourser,

$C[n]$: valeur de la pièce à rembourser,

$Nbpiece$: nombre de pièces total

$X[i] \leftarrow X[i] + 1$

$reste \leftarrow reste - C[i]$

$Nbpiece \leftarrow Nbpiece + 1$

Algorithme 10 TSP - plus proche voisin randomise

Entrées: N : nombre de villes,

$dist[N][N]$: distance entre toutes les paires de villes,

$_ville_visit[N]$: enregistre si une ville a été visitée (est initialisée à -1),

β : nombre utilisé lors du choix

Sorties: $succ[N]$: successeur de chaque ville dans la tournée construite,

$ante[N]$: antécédent de chaque ville dans la tournée construite

1: $compteur \leftarrow 1$ //nombre de villes visitées

2: $p \leftarrow 0$ //dernière ville ajoutée

3: $_ville_visit[0] \leftarrow 1$

4: **Tant que** $compteur < N$ **faire**

5: choisir les deux villes A et B les plus proches de la dernière ville visitée et n'ayant pas encore été visitée

6: **Si** les deux villes A et B existent **alors**

7: tirer un nombre au hasard appelé *hasard*

8: **Si** $hasard > \beta$ **alors**

9: **AJOUTER**($N, p, A, succ, ante, compteur$) //on ajoute la ville A au tour en construction avec la probabilité β

10: **sinon**

11: **AJOUTER**($N, p, B, succ, ante, compteur$) //on ajoute la ville B au tour en construction avec la probabilité $1 - \beta$

12: **Fin si**

13: **sinon Si** une des villes A et B existe **alors**

14: ajouter au tour la ville qui existe

15: **sinon**

16: Retourner "impossible de construire une solution"

17: **Fin si**

18: **Fin tant que**

19: connecter la dernière ville ajoutée à la ville de départ

20: Retourner $succ, ante$

Algorithme 11 AJOUTER

Entrées: N : nombre de villes,

p, A : villes,

$succ[N]$: successeur de chaque ville dans la tournée construite,

$ante[N]$: antécédent de chaque ville dans la tournée construite,

$compteur$: nombre de villes visitées

1: $succ(p) \leftarrow A$

2: $ante(A) \leftarrow p$

3: $p \leftarrow A$

4: $compteur \leftarrow compteur + 1$

1.5.2 Exemple de recherche locale pour le problème du voyageur de commerce

Une recherche locale appliquée à une solution du problème du voyageur de commerce consiste à améliorer une tournée trouvée par exemple grâce à un algorithme glouton, afin que la distance parcourue par la tournée soit la plus petite possible.

Opérateurs

Soit *tour* une tournée construite à l'aide d'une méthode heuristique du problème du voyageur de commerce. Par la suite nous allons présenter SWITCH et 2-OPT deux opérateurs.

— L'opérateur SWITCH.

Dans le cas du TSP, l'opérateur SWITCH consistera à sélectionner deux villes dans *tour* et à les permuter. L'algorithme 12 décrit l'opérateur SWITCH sur une solution du voyageur de commerce. Dans cet algorithme, on permute deux villes consécutives.

— L'opérateur k-OPT.

La technique k-opt consiste à supprimer k arêtes dans le tour *tour* et, à recomposer un autre tour en reconnectant ces chaînes d'une autre manière. L'algorithme k-opt qui essaye toutes les possibilités avec une complexité de $O(n^k)$.

Par exemple, pour $k=2$, ces transformations entraînent un changement dans le sens de parcours de certaines sous-chaînes de *tour*. Changer le sens de parcours des sous-chaînes dans le cas du TSP consiste à inverser l'ordre dans lequel on visite les villes de ces sous-chaînes.

Algorithme 12 SWITCH

Entrées: *tour* : tournée construite par une méthode heuristique représentée par *succ* (successeurs) et *ante* (antécédent),

i : indice de la ville sur laquelle on appliquera le switch

Sorties: tournée modifiée par le switch : *tour*

//permutation de la position de *i* et de son successeur

$j \leftarrow succ[i]$

$j_temp \leftarrow succ[j]$

$i_temp \leftarrow ante[i]$

$succ[i_temp] \leftarrow j$

$succ[i] \leftarrow j_temp$

$ante[j] \leftarrow i_temp$

$ante[j_temp] \leftarrow i$

$succ[j] \leftarrow i$

$ante[i] \leftarrow j$

Retourner *tour*

Plus spécifiquement, dans le 2-OPT, on sélectionnera deux arêtes. L'algorithme 13 décrit le procédé de transformation du 2-OPT. La figure 1.7 montre un exemple de transformation d'une tournée à l'aide du 2-opt [**2opting**], ici, la tournée est parcourue de la ville *a* vers la ville *i*.

Dans la figure 1.7, la figure 1.3(a) montre la tournée et la figure 1.3(b) montre le résultat obtenu après avoir appliqué le 2-OPT sur la tournée de la la figure 1.3(a), en utilisant comme paramètres d'entrées du 2-OPT les villes *i* et *j*. Nous constatons que le sens de parcours de la sous-chaîne qui va de la ville *c* à la ville *j* a été inversé après l'application du 2-OPT.

Avant d'appliquer le 2-OPT, comment choisir les arêtes à réorganiser dans la tournée ? Il existe plusieurs stratégies que nous présenterons par la suite.

Stratégies de décisions

Dans le cadre de la recherche locale, on doit être capable de connaitre à quel moment il faut s'arrêter lors du processus de recherche, pour cela, plusieurs techniques ou stratégies de décisions existent.

Les stratégies de décisions permettent de savoir à quel moment appliquer ou non une transformation à la tournée à améliorer. Parmi les stratégies nous pouvons citer la de descente, la plus grande descente et la marche aléatoire.

— La descente.

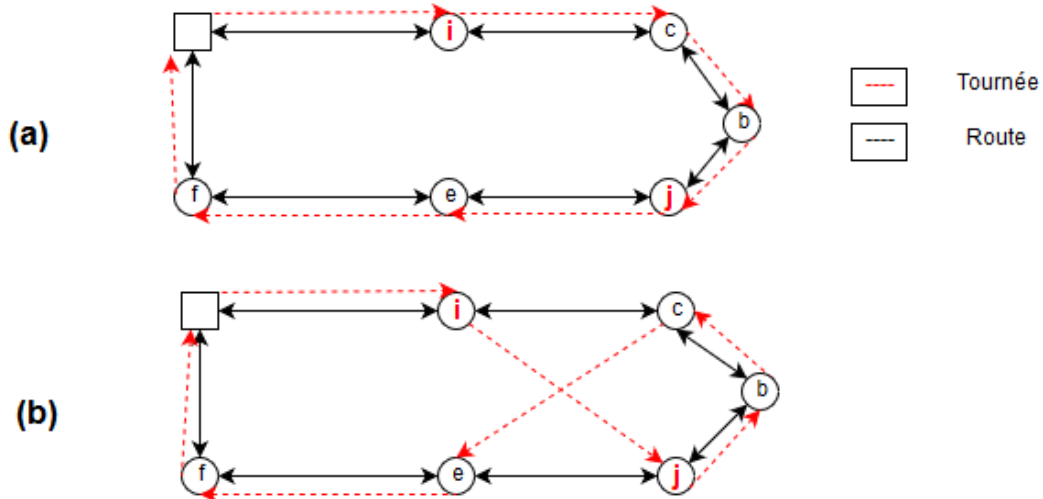


FIGURE 1.7 – (a) tournée initiale, (b) tournée après la transformation 2-OPT(i,j).

Elle consiste à explorer le voisinage d'une solution à l'aide d'un opérateur et à s'arrêter lorsqu'on trouve une solution améliorante. Une solution améliorante est une solution qui est meilleure que la solution courante.

Dans le cas du TSP, si on a comme opérateur le 2-opt, cette stratégie consistera à explorer les couples de villes jusqu'à trouver un couple qui minimise la longueur du chemin parcouru. La transformation 2-opt sera donc effectuée sur le premier couple de villes trouvé et on arrête l'exploration des couples de villes. L'algorithme 14 est une recherche locale pour le problème du voyageur de commerce. Dans cet algorithme, pour la ville de départ du tour, on parcourt l'ensemble de ses voisins (en respectant l'ordre du tour), si en appliquant le 2-OPT avec l'un de ses voisins on obtient une meilleure tournée alors le processus s'arrête et on revoit cette tournée. Si on ne trouve aucun couple qui améliore la tournée alors on passe à la ville suivante dans le tour et le processus recommence. Si on trouve deux villes consécutives qui améliorent la tournée alors on applique l'opérateur SWITCH sur la tournée avec comme paramètre les deux villes consécutives.

— La plus grande descente.

Elle consiste à parcourir tout le voisinage d'une solution à l'aide d'un opérateur et à sélectionner la meilleure solution trouvée.

Dans le TSP, si l'on a comme opérateur le 2-opt, cette stratégie consistera à parcourir tous les couples de villes et à sélectionner le couple qui permet d'obtenir une tournée

Algorithme 13 2-OPT

Entrées: *tour* : tournée construite par une méthode heuristique représentée par *succ* (successeurs) et *ante* (antécédent),
i, j : indices des villes sur lesquelles on appliquera le 2-OPT,
N : nombre de villes

Sorties: tournée modifiée par le 2-OPT : *tour*

```
i_temp ← succ[i]  
j_temp ← succ[j]  
k ← j  
Tant que k ≠ i_temp faire  
    k_temp ← ante[k] //changement des successeurs  
    succ[k] ← k_temp  
    k ← k_temp  
Fin tant que  
succ[i] ← j  
succ[i_temp] ← j_temp  
pour l = 1 à N faire  
    k_temp ← succ[i] //changement des antécédents  
    ante[k_temp] ← i  
fin pour  
Retourner tour
```

de longueur la plus petite possible. Et enfin, à appliquer le 2-opt sur ce couple de villes.

— La marche aléatoire.

Elle consiste à partir d'une solution, à appliquer sur cette solution un certain opérateur, les paramètres de cet opérateur sont choisis aléatoirement, par exemple pour le 2-OPT on choisit les paramètres *i* et *j* aléatoirement. On applique l'opérateur sur la solution et on garde le résultat obtenu même si il n'est pas améliorant. Cette stratégie permet de sortir des optimum locaux (contrairement à la descente).

Algorithme 14 TSP - recherche_locale_descente

Entrées: *tour* : tournée, *dist* : matrice des distances

Entrées: *tour* : tournée

courant_i \leftarrow succ(1) // successeur de la première ville de la tournée

I \leftarrow 0 // ville qui sera utilisée pour faire 2-opt

J \leftarrow 0 // ville qui sera utilisée pour faire 2-opt

notstop \leftarrow true

Tant que *courant_i* \neq 1 **et** *notstop* **faire**

courant_j \leftarrow tour.succ[*courant_i*] /*On cherche la deuxième ville pour le 2-OPT

Tant que *courant_j* \neq 1 **faire**

i_temp \leftarrow tour.succ[*courant_i*]

j_temp \leftarrow tour.succ[*courant_j*]

gain \leftarrow $dist[courant_i][i_temp] + dist[courant_j][j_temp] - dist[courant_i][courant_j] - dist[i_temp][j_temp]$

Si *gain* $>$ 0 **alors**

I \leftarrow *courant_i*

J \leftarrow *courant_j*

notstop \leftarrow false

Fin si

courant_j \leftarrow tour.succ[*courant_j*]

Fin tant que

courant_i \leftarrow tour.succ[*courant_i*]

Fin tant que

Si *I* \neq 0 **ou** *J* \neq 0 **alors**

Si *J* = tour.succ[*I*] **ou** *I* = tour.succ[*J*] **alors**

SWITCH(*tour*, *I*)

sinon

2 - OPT(*tour*, *I*, *J*)

Fin si

sinon

Retourner "Aucun couple de ville qui diminue la distance parcourue n'a été trouvé"

Fin si

Troisième partie

Métaheuristiques

Dans cette partie, on présentera quelques métaheuristiques : le Greedy randomized adaptive search procedure (GRASP), le recuit simulé et la recherche taboue.

1.6 Procédure de recherche itérative en deux phases (GRASP)

1.6.1 Principe

L'algorithme Greedy randomized adaptive search procedure (GRASP) [Feo_1989] a été formalisé par Thomas A Feo et Mauricio G.C Resende en 1989. La méthode GRASP consiste à générer plusieurs solutions dans l'espace des solutions possibles d'un problème (P) donné, à l'aide d'un algorithme glouton randomisé puis, à améliorer chaque solution obtenue à l'aide d'une recherche locale et enfin à sélectionner la meilleure solution parmi ces solutions améliorées. Cela s'effectue en deux principales phases : la phase de construction d'une solution et la phase d'amélioration de celle-ci. Ce processus est répété tant qu'une certaine condition préalablement fixée n'est pas satisfaite. Cette condition peut par exemple être un certain nombre d'itérations à effectuer. L'algorithme 15 [siarry__metaheuristiques__2014] décrit le GRASP de façon générale.

Algorithme 15 Procédure GRASP

Entrées: α : paramètres pour construire une solution,

$temps - limite$: nombre d'itérations maximum

Sorties: X^* : solution optimale trouvée

Tant que temps CPU $>$ $temps - limite$ **faire**

$X \leftarrow$ Glouton randomisé(α)

$X \leftarrow$ Recherche locale(X)

Si $z(X)$ meilleur que $z(X^*)$ **alors**

$X^* \leftarrow X$

Fin si

Modifier temps CPU

Fin tant que

Retourner X^*

L'algorithme 15 décrit la procédure GRASP et la condition d'arrêt est de faire un nombre d'itérations. La fonction z calcule le coût d'une solution passée en paramètre.

1.6.2 Exemple d'algorithme GRASP pour le problème du voyageur de commerce

Algorithme GRASP

En ce qui concerne le problème du voyageur de commerce, on utilisera l'algorithme du plus proche voisin randomisé pour générer plusieurs tournées, puis on améliorera ces tournées avec une recherche locale en utilisant la descente et l'opérateur 2-opt. Et enfin, on choisira la meilleure solution parmi les solutions améliorées. L'algorithme 16 présente ce procédé.

Algorithme 16 TSP - GRASP

Entrées: *Nbexecute* : Nombre de tournées générés

Sorties: *tour2* : Tournée construite

min \leftarrow *infini*

D \leftarrow 0

pour *i*=0 à *Nbexecute* **faire**

tour \leftarrow voyageur_de_commerce_plus_proche_voisin_randomise()

 recherche_locale_descente_aleatoire(*tour*)

D \leftarrow Calculer la longueur de *tour*

Si *D* < *min* **alors**

min \leftarrow *D*

tour2 \leftarrow *tour*

Fin si

fin pour

Retourner *tour2*

1.7 Recuit simulé

1.7.1 Principe

L'algorithme du recuit simulé a été formalisé par Kirkpatrick et al en 1983 [**kirkp**]. Initialement, l'utilisateur fixe une température initiale appelée *T*, une température finale et un processus de diminution lente de la température initiale vers la température finale appelé programme de recuit. Par exemple si la température initiale est fixée à 10 et que la température finale est fixée à 1, on peut dire que le programme de recuit consiste à diminuer la température de 1, ce qui fera faire 10 itérations.

En 2014, d'après [**siarry_metaheuristiques_2014**], pour appliquer l'algorithme du recuit simulé, on part d'une solution initiale (configuration initiale) et d'une tempéra-

ture initiale, on fait subir à cette solution initiale une modification élémentaire ; si cette transformation a pour effet de diminuer la fonction objectif du système de ΔE , elle est acceptée ; si elle provoque au contraire une augmentation de ΔE , elle peut être acceptée tout de même, avec la probabilité $\exp(-\Delta E/T)$ (ce processus est dit de Métropolis). On itère ensuite ce procédé, en gardant la température constante, jusqu'à ce que l'équilibre thermodynamique soit atteint. L'équilibre thermodynamique est atteint au bout d'un nombre "suffisant" de modifications. On abaisse alors la température, avant d'effectuer une nouvelle série de transformations : la loi de décroissance de la température est souvent empirique, tout comme le critère d'arrêt de l'algorithme. L'algorithme s'arrête lorsqu'on atteint la température finale, cet état est appelé système figé. L'algorithme 17 présente la structure générale d'un algorithme de recuit simulé.

Dans le TSP, par exemple, une solution initiale est une tournée, une modification élémentaire est la permutation de deux villes d'une tournée ou un 2-OPT et la fonction objectif est la longueur d'une tournée.

Les difficultés de cette méthode sont de définir la température initiale, de déterminer le processus de dégradation de la température initiale, de déterminer combien de temps garder la température constante avant de la dégrader et de déterminer la modification élémentaire à faire afin d'obtenir une solution proche de l'optimum. L'organigramme de l'algorithme du recuit simulé est présenté à la figure 1.8. La configuration initiale représente la solution initiale. Le programme de recuit définit la façon dont la température initiale va décroître au cours du temps. Le système est figé lorsqu'on a atteint la température finale qui est un seuil fixé par l'utilisateur. Une modification élémentaire est par exemple dans le cas du TSP un SWITCH ou un 2-OPT.

1.7.2 Exemple d'algorithme de recuit simulé pour le problème du voyageur de commerce

Concernant le voyageur de commerce, l'algorithme 18 est un algorithme du recuit simulé du voyageur de commerce. La configuration initiale est une tournée que nous avons obtenue avec l'heuristique du plus proche voisin. On appellera T la température initiale et on dégradera cette température de 2 (programme de recuit), voir algorithme 20, au fil du temps jusqu'à atteindre une température inférieure ou égale à zéro avant de s'arrêter (sys-

Algorithme 17 Recuit simulé

Entrées: S : solution initiale,

T : température initiale,

F : température finale

Sorties: S' : solution finale

$temperatureCourante \leftarrow T$

$S' \leftarrow S$

Tant que $temperatureCourante \neq F$ **faire**

Tant que l'équilibre thermodynamique n'est pas atteint **faire**

 mettre dans $S1$ la solution S' modifiée

$\Delta E \leftarrow \text{cout}(S1) - \text{cout}(S')$ // la fonction cout calcule le coût d'une solution

Si $\Delta E \leq 0$ **alors**

$S' \leftarrow S1$

sinon

$S' \leftarrow S1$ avec la probabilité $\exp(-\Delta E/T)$

Fin si

Fin tant que

$temperatureCourante \leftarrow \text{ProgramDeRecuit}(temperatureCourante)$

Fin tant que

Retourner S'

tème figé). L'équilibre thermodynamique sera atteint lorsqu'on aura fait $P(\text{seuilcourant})$ itérations, autrement dit la température constante sera maintenue $P(\text{seuilcourant})$ itérations. P représenté par l'algorithme 19 est une fonction qui détermine le nombre d'itérations maximales durant lesquelles la température sera constante. Dans les expérimentations, on incrémente seuilcourant de 1 pour avoir $P(\text{seuilcourant})$, mais $P(\text{seuilcourant})$ peut être défini d'une autre façon.

En fonction de la valeur du nombre qu'on obtiendra avec la fonction *random*, on appliquera ou non le 2-OPT lorsque le couple de villes n'est pas améliorant. La modification élémentaire consiste à appliquer le 2-OPT avec les couples de villes améliorants trouvés. Un couple de villes est améliorant lorsqu'en appliquant le 2-OPT avec ce couple on obtient une tournée plus proche de l'optimum c'est-à-dire qu'il y a eu diminution de sa longueur après le 2-OPT. Il existe plusieurs méthodes pour générer $i1$ et $i2$: dans cet algorithme on a choisi de parcourir tous les couples de villes de façon ordonnée mais on pouvait aussi sélectionner les couples de villes connectés à des arêtes d'étiquettes maximales.

gain fait la différence entre les étiquettes arêtes enlevées et les arêtes ajoutées dans la tournée. Lorsque le gain est positif cela signifie qu'en appliquant le 2-OPT à la tournée on obtiendra une tournée de coût plus minimale.

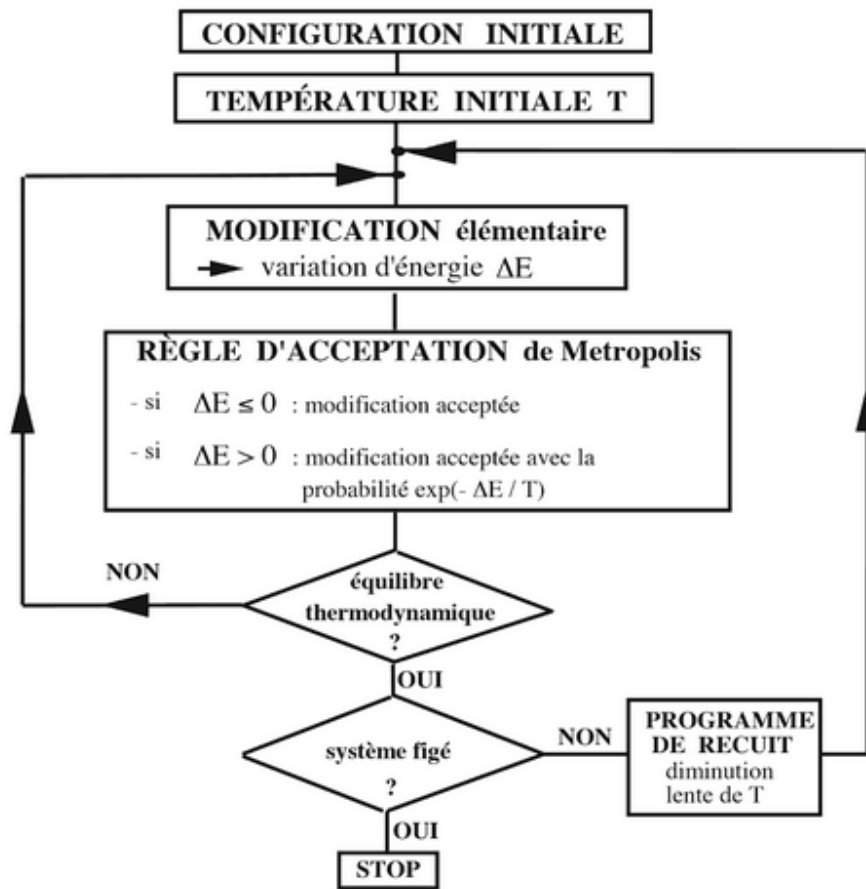


FIGURE 1.8 – Organigramme de l’algorithme du recuit simulé [siarry_metaheuristiques_2014]. La configuration initiale représente la solution initiale. Le programme de recuit définit la façon dont la température initiale va décroître au cours du temps. Le système est figé lorsqu’on a atteint la température finale qui est un seuil fixé par l’utilisateur.

$hasard$, $\beta \in [0, 1]$ sont utilisés pour choisir si le 2-OPT sera ou non appliqué dans le cas d’un couple de villes qui n’est pas améliorant dans l’algorithme 10. Par exemple, dans les expérimentations on a pris $\beta = 0.5$ ce qui signifie qu’on a 50% de chance d’appliquer le 2-OPT avec le couple qui n’est pas améliorant. Par contre, Si $\beta = 0.7$, on 30% de chance d’appliquer le 2-OPT avec le couple qui n’est pas améliorant.

1.8 Recherche taboue

1.8.1 Principe

L’algorithme de la recherche taboue [GLove] a été formalisé par F. Glover en 1986. Pour faire de la recherche taboue, on commence par générer une solution initiale ou

Algorithme 18 TSP - recuit

Entrées: *tour* : tournée construite,

T : température initiale,

dist[*N*][*N*] : distance entre toutes les paires de villes,

Sorties: *tour* : tournée construite

seuilCourant $\leftarrow T$

Tant que *seuilCourant* ≥ 0 **faire**

pour *i* = 1 à *P*(*seuilCourant*) **faire**

 Générer *i1* et *i2* et calculer *gain* // *i1* et *i2* sont deux villes

Si *gain* > 0 **alors**

deux_opt(*tour*, *i1*, *i2*)

sinon

 tirer au hasard *hasard* avec la fonction *random*

Si *hasard* > β **alors**

deux_opt(*tour*, *i1*, *i2*)

Fin si

Fin si

fin pour

seuilCourant $\leftarrow \text{ProgramDeRecuit}(\text{seuilCourant})$ //programme de recuit

Fin tant que

Retourner *tour*

Algorithme 19 P

Entrées: *seuilCourant* : température à un instant

Sorties: *seuilCourant* : température à un instant

Retourner *seuilCourant* + 1

Algorithme 20 ProgramDeRecuit

Entrées: *seuilCourant* : température à un instant

Sorties: *seuilCourant* : température à un instant

Retourner *seuilCourant* + 2

configuration initiale à l'aide d'une heuristique, puis on initialise une liste vide appelée *listeTaboue*. On génère plusieurs voisins de la solution initiale, puis, on choisit le meilleur voisin. On perturbe la solution initiale à l'aide de l'opérateur et des paramètres de ce meilleur voisin, même si cet opérateur dégrade la solution initiale, puis on insère cet opérateur et ses paramètres dans la liste taboue. La nouvelle solution initiale devient la solution perturbée et on recommence ces étapes un certain nombre de fois (placé en paramètre).

Par exemple dans le TSP, comme solution initiale on peut générer une tournée et la perturber avec un 2-OPT. L'idée ici est d'accepter de dégrader sa solution mais sans tournée en rond c'est-à-dire sans boucler sur les mêmes solutions. Les opérateurs présents dans la liste taboue ne seront pas appliqués pendant un certain temps défini au préalable.

Les difficultés de cette méthode sont de savoir quoi conserver dans la liste *listeTaboue* (garder par exemple la tournée entière ou alors la signature de la tournée si on est dans le cas du voyageur de commerce) et quand vider la liste *listeTaboue* (car à un certain moment elle sera trop longue). L'organigramme de l'algorithme de la recherche taboue est présenté à la figure 1.9.

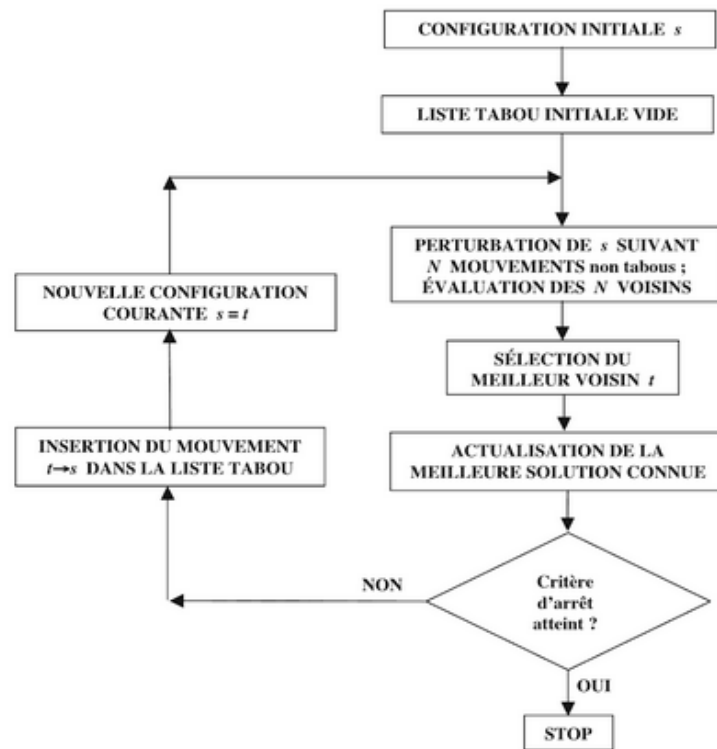


FIGURE 1.9 – Organigramme de l’algorithme de la recherche taboue [siarry_metaheuristiques_2014]. La configuration initiale est la solution initiale. Le mouvement est l’opérateur et ses paramètres. Le meilleur voisin est celui qui est plus proche de l’optimum.

Quatrième partie

Méthodes exactes

Les méthodes exactes permettent de trouver des solutions optimales en temps polynomial pour des instances de problèmes de petite taille. Lorsque les problèmes sont de grande taille ces techniques ont du mal à trouver des solutions car le temps de calcul nécessaire pour obtenir une solution est trop long.

1.9 Programmation linéaire

1.9.1 Principe

Dans cette partie, nous allons définir les notions de programme linéaire en nombres entiers (PLNE) et programme linéaire (PL).

Définition 1 [Sakar84] *soient A une matrice de taille $n \times m$, b un vecteur colonne de taille m et c un vecteur ligne de taille n . Un programme linéaire en nombres entiers est le problème d'optimisation :*

$$(P) \begin{cases} Ax \leq b \\ cx = z(Max) || cx = z(Min) \end{cases}$$

avec $x_j \in \mathbf{N}$, $j = \{1, 2, \dots, n\}$ et la fonction objectif z est une fonction linéaire à maximiser ($z(Max)$) ou à minimiser ($z(Min)$) ; x est un vecteur inconnu de taille n . On dit qu'on a un programme linéaire en variables bivalentes si les contraintes $x_j \in \mathbf{N}$ sont remplacées par $x_j \in \{0, 1\}$.

Définition 2 [Sakar84] *Quand on relâche les contraintes d'intégrité sur les variables ($x_j \in \mathbf{N}$) dans le programme linéaire en nombres entiers (P) de la définition 1, on obtient le programme linéaire :*

$$(P') \begin{cases} Ax \leq b \\ cx = z(Max) || cx = z(Min) \end{cases}$$

avec $x_j \geq 0$, $j = \{1, 2, \dots, n\}$.

Il existe une relation entre (P) et (P') : si par hasard, la solution optimale de (P') est entière, c'est aussi une solution de (P).

Un programme linéaire est constitué de variables de décision, de contraintes auxquelles sont soumises ces variables de décision et de la fonction objectif à optimiser. Il existe plusieurs types de contraintes :

- les contraintes d'intégrités ou de signe par exemple $x_j \in \mathbf{N}$ du problème (P) et $x_j \geq 0$ du problème (P') avec $j = \{1, 2, \dots, n\}$
- les contraintes d'inégalité par exemple $Ax \leq b$ du problème (P')
- les contraintes d'égalité par exemple $Ax = b$.

1.9.2 Exemple de programme linéaire pour le problème du rendu de monnaie

Données d'entrée

Les données d'entrée sont :

- N est le nombre de type de pièces différentes.
- Tab est un vecteur de N entiers qui donne la valeur de chaque type de pièces.
- M est un nombre entier qui donne la somme à rendre.
- Max est un vecteur de N entiers qui représente le nombre maximum de chaque type de pièces.

Variable de décision

La variable de décision X est un vecteur qui représente le nombre de chaque type de pièces rendue. X est un vecteur indexé de 1 à N tel que $\forall i \in \llbracket 1, N \rrbracket$, $X[i]$ est le nombre de pièces de type i à rendre.

Fonction objectif

Dans ce problème l'objectif est de minimiser le nombre de pièces remboursé, donc $Min(\sum_{i=1}^N X[i])$.

Contraintes

Les contraintes du problème sont :

- $\sum_{i=1}^N X[i] * Tab[i] = M$: la somme remboursée doit être égale à M .
- $\forall i \in \llbracket 1, N \rrbracket, X[i] \leq Max[i]$: pour chaque type de pièces, le nombre de pièces remboursé doit être inférieur ou égale au nombre maximum de ce type.
- Contrainte d'intégrité : $\forall i \in \llbracket 1, N \rrbracket X[i] \in \mathbf{N}$.

Implémentation

Nous avons implémenté ce programme linéaire dans OPL. La figure 1.10 représente le résultat de l'exécution.

- *rendudemonnaie.mod* est le fichier qui décrit le modèle.

```

1      //Données input
2      int N =...;
3      range I =1..N;
4      int Tab[I] =...;
5      int M =...;
6      int Max[I] =...;
7
8      //Variables
9      dvar int+ X[I];
10
11     //PL
12     minimize
13     sum(i in I) X[i];
14
15     //contraintes
16     subject to
17     {
18     sum(i in I) X[i]*Tab[i] == M;
19     forall(i in I) X[i]<=Max[i];
20     }
21
22
23

```

- *rendudemonnaie.dat* est le fichier qui décrit les données qui seront utilisées pour tester le modèle.

```

1      N=3;
2      Tab=[ 1 , 3 , 4 ];
3      M=6;
4      Max=[ 3 , 1 , 2 ];
5

```

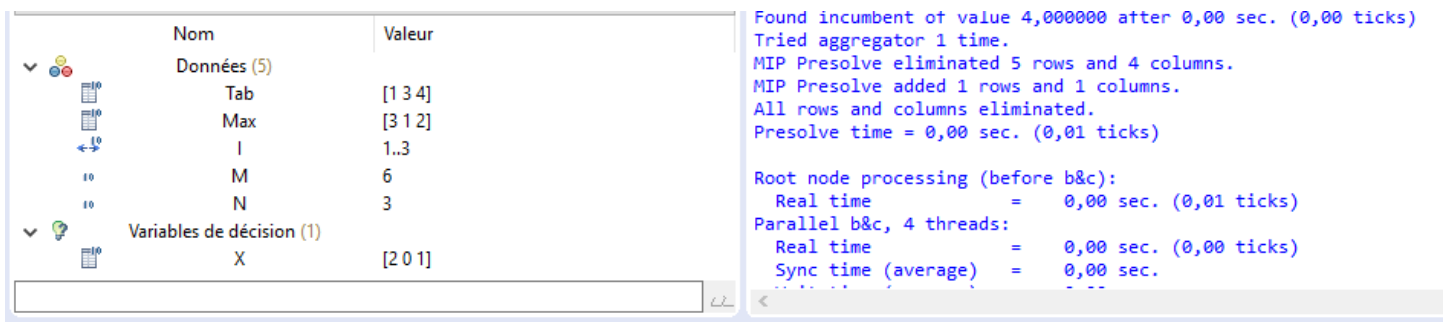


FIGURE 1.10 – Exemple d’exécution du PLNE du rendue de monnaie.

1.9.3 Exemple de programme linéaire pour le problème du voyageur de commerce

Données d’entrée

- N est le nombre de ville.
- $dist$ est un vecteur de taille $N \times N$ d’entiers qui représente la distance entre toutes les paires de villes.

Variables de décision

$arrete_selectionne$ est un vecteur de taille $N \times N$ de booléens et t la tournée construite.

$$arrete_selectionne[i][j] = \begin{cases} 1 & \text{si } (i,j) \in t \\ 0 & \text{sinon.} \end{cases}$$

Soit i une ville, $date[i]$ est un vecteur qui contient l’ordre de passage de la ville i dans

la tournée construite.

Fonction objectif

L'objectif est de minimiser la distance parcourue en passant par toutes les villes :

$$\text{Min}(\sum_{i=1}^N \sum_{j=1}^N \text{dist}[i][j] * \text{arrete_selectionne}[i][j]).$$

Contraintes

Les contraintes sont :

- $\forall i \in \llbracket 1, N \rrbracket, \sum_{j=1}^N \text{arrete_selectionne}[i][j] = 1,$
 $\forall j \in \llbracket 1, N \rrbracket, \sum_{i=1}^N \text{arrete_selectionne}[i][j] = 1$: on arrive et repart exactement une fois de chaque ville.
- $\forall i \in \llbracket 1, N \rrbracket, \text{arrete_selectionne}[i][i] = 0$: on ne doit pas se déplacer d'une ville vers elle-même.
- $\text{date}[1] = 1$ et $\forall i \in \llbracket 1, N \rrbracket, \text{date}[i] \leq n.$
- Contraintes d'intégrité sont : $\forall (i, j) \in \llbracket 1, N \rrbracket^2, \text{arrete_selectionne}[i][j] \in \llbracket 0, 1 \rrbracket.$
- Contrainte de sous-tours.

Un sous-tour est un cycle qui ne contient pas toutes les villes. Dans le TSP notre objectif est de construire un cycle contenant toutes les villes donc nous devons ajouter des contraintes qui empêchent que des sous-tours se forment. $\forall i \in \llbracket 1, N \rrbracket$ et $\forall j \in \llbracket 1, N \rrbracket$

$\text{arrete_selectionne}[i][j] = 1 \Rightarrow \text{date}[i] + 1 \leq \text{date}[j].$ Cette expression étant non linéaire il faut la linéariser. Pour cela on utilise un "big M" qui est un nombre suffisamment grand. La contrainte se réécrit :

$\forall (i, j) \in \llbracket 1, N \rrbracket^2, \text{date}[i] + 1 \leq \text{date}[j] + M(1 - \text{arrete_selectionne}[i][j]).$ Le "big M" doit être un nombre suffisamment grand pour que cette expression soit toujours vérifiée dans le cas où $\text{arrete_selectionne}[i][j] = 0$ mais pas trop grand aussi pour éviter de ralentir les calculs dans OPL. On prendra $M = N - 1.$

Implémentation

- *TSP.mod* est le fichier qui décrit le modèle.

```

1      //Données input
2      int N =...;
3      range I =1..N;
4      int dist[I][I] =...;
5
6      //Variables
7      dvar int+ arrete_selectionne[I][I];
8      dvar int+ date[I];
9
10     //PL
11     minimize
12     sum(i in I) sum(j in I) dist[i][j]*arrete_selectionne[i][j];
13
14     //contraintes
15     subject to
16     {
17     date[1]==1;
18     forall(i in I)
19     {
20     sum(j in I) arrete_selectionne[i][j]==1;
21     sum(j in I) arrete_selectionne[j][i]==1;
22     arrete_selectionne[i][i]==0;
23     date[i]<=V;
24     forall(j in 2..N)
25     date[i]+1<=date[j]+(V-1)*(1-arrete_selectionne[i][j]);
26
27     }
28     }
29
30     //affichage de la matrice arrete_selectionne
31     execute DISPLAY
32     {
33     for(var i in I)
34     {
35     for(var j in I)
36     {
37     if(arrete_selectionne[i][j]==1)

```

```

38     write(" | 1");
39     else
40     write(" | ");
41     }
42     write("\n");
43     }
44     }
45     execute DISPLAY2
46     {
47     for(var i in I)
48     {
49     for(var j in I)
50     {
51     if(arrete_selectionne[i][j]==1)
52     {
53     write(i,"->",j,"\n");
54     }
55     }
56
57     }
58     }
59

```

— *TSP.dat* est le fichier qui décrit les données qui seront utilisées pour tester le modèle.

```

1      //TSP
2      N=5; //Nombre de villes
3      dist
      =[[10,2,3,4,1],[2,10,2,4,2],[3,2,10,3,5],[4,4,3,10,7],[1,2,5,7,10]];
4

```

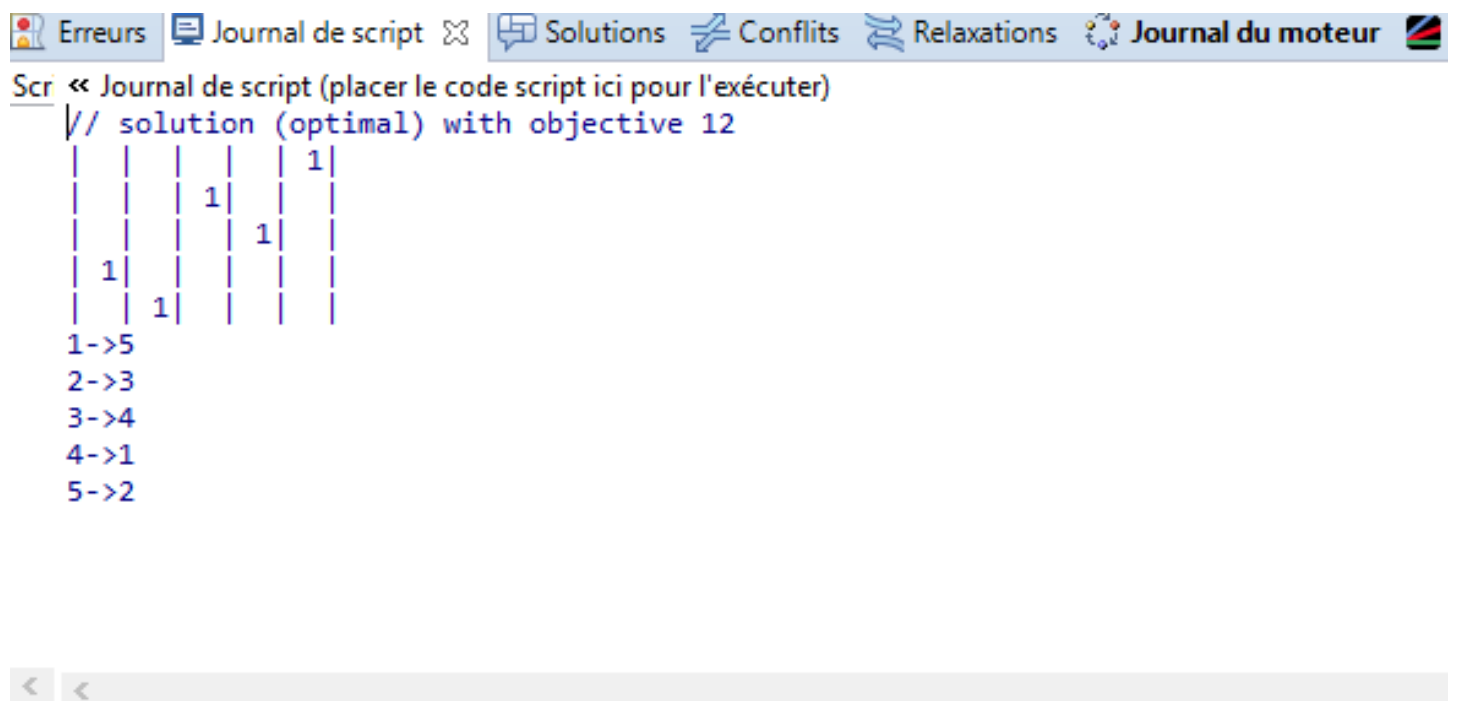


FIGURE 1.11 – Exemple d'exécution du PLNE du TSP.

Cinquième partie

Etude comparative entre les méthodes étudiées en utilisant les données de la Tsplib

1.10 Description des données et de l’environnement de développement

C’est une collection de 102 instances de TSP qui a été fournie par Andre Rohe, sur la base d’ensembles de données VLSI (Very Large Scale Integration) étudiés au Forschungsinstitut für Diskrete Mathematik, Universität Bonn [vlsi]. L’Institut de Bonn est un site universitaire de premier plan pour la recherche appliquée en matière de conception VLSI.

La taille des instances de la collection VLSI varie de 131 villes à 744 710 villes. Dans ces exemples, le coût des déplacements entre les villes est spécifié par la distance euclidienne arrondie au nombre entier le plus proche (la norme TSPLIB `EUC_2D`). La collection VLSI est présentée dans un groupe de 11 pages, avec 10 instances par page. L’ensemble de la collection peut également être téléchargé sous la forme d’un seul fichier tar gzipé `vlsi_tsp.tgz`. Un résumé de l’état actuel de la solution pour les instances a été fourni.

On a effectué des expérimentations sur 5 instances de tailles : 131, 423, 436, 662, 22777.

On a effectué les expérimentations sous un système d’exploitation Windows 10 de 64 bits avec une mémoire RAM de 16 Gigaoctets et un processeur core i5-6500 3.20 GHz.

1.11 Coût des solutions

L’optimum est donné par la `tsplib`. La recherche locale ici a été faite sur une tournée construite par l’heuristique du plus proche voisin.

1.11.1 Heuristiques

Les figures 1.13 et 1.14 montrent les Gap des solutions obtenues (de plusieurs tailles d’instances du TSP) en expérimentant les heuristiques étudiées.

1.11.2 Métaheuristiques

La figure 1.15 montre les coûts des solutions obtenues (de plusieurs tailles d’instances du tsp) en exécutant les métaheuristiques étudiées.

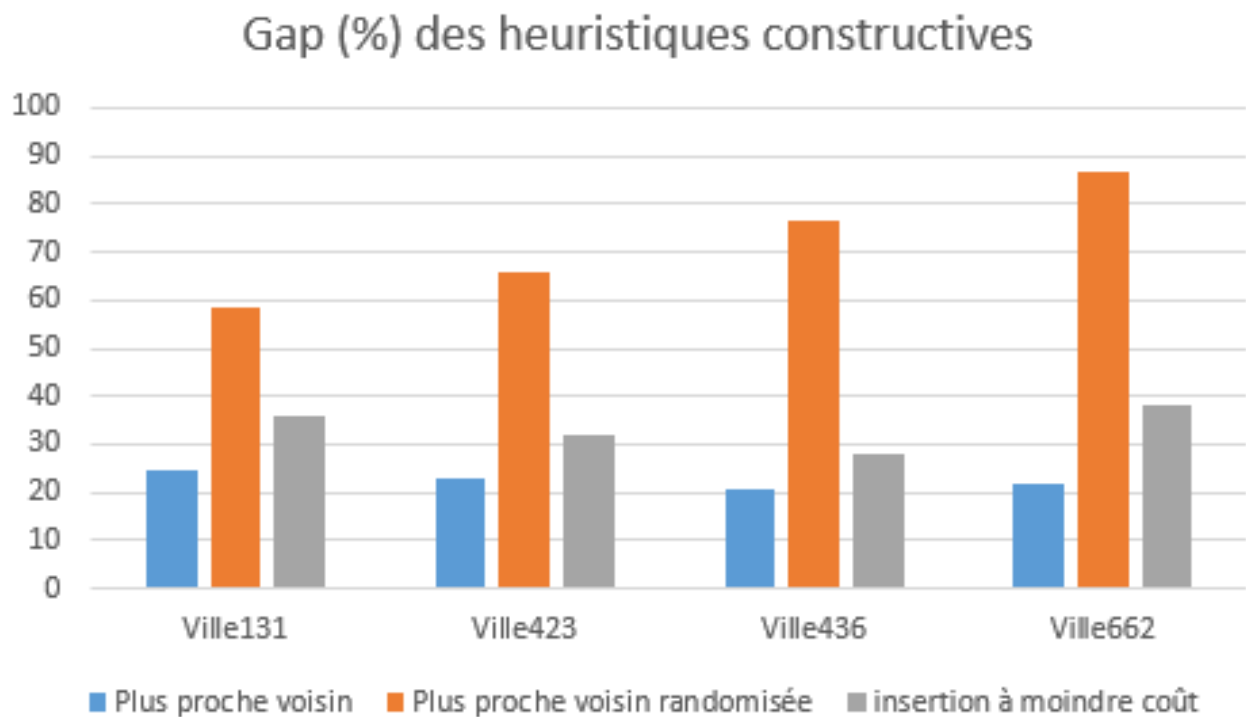


FIGURE 1.12 – Heuristiques constructives. Gap des solutions obtenues de plusieurs tailles d’instances du TSP. Le nombre présent à côté du mot ville représente la taille de l’instance.

1.12 Temps d’exécution

Le temps CPU 1.16 est mesuré avec la fonction `<clock>` de la bibliothèque `time` en C++.

1.13 Observations

1. Temps d’exécution.

- L’heuristique la plus rapide est l’heuristique du plus proche voisin.
- La métaheuristique la plus rapide est le recuit simulé.
- Il existe un grand écart entre le temps d’exécution des heuristiques et le temps d’exécution des métaheuristiques. Le temps d’exécution des métaheuristiques est supérieur à celui des heuristiques.

2. Coût.

- L’heuristique qui donne le meilleur coût c’est l’heuristique du plus proche voisin couplée avec une recherche locale de plus grande descente.

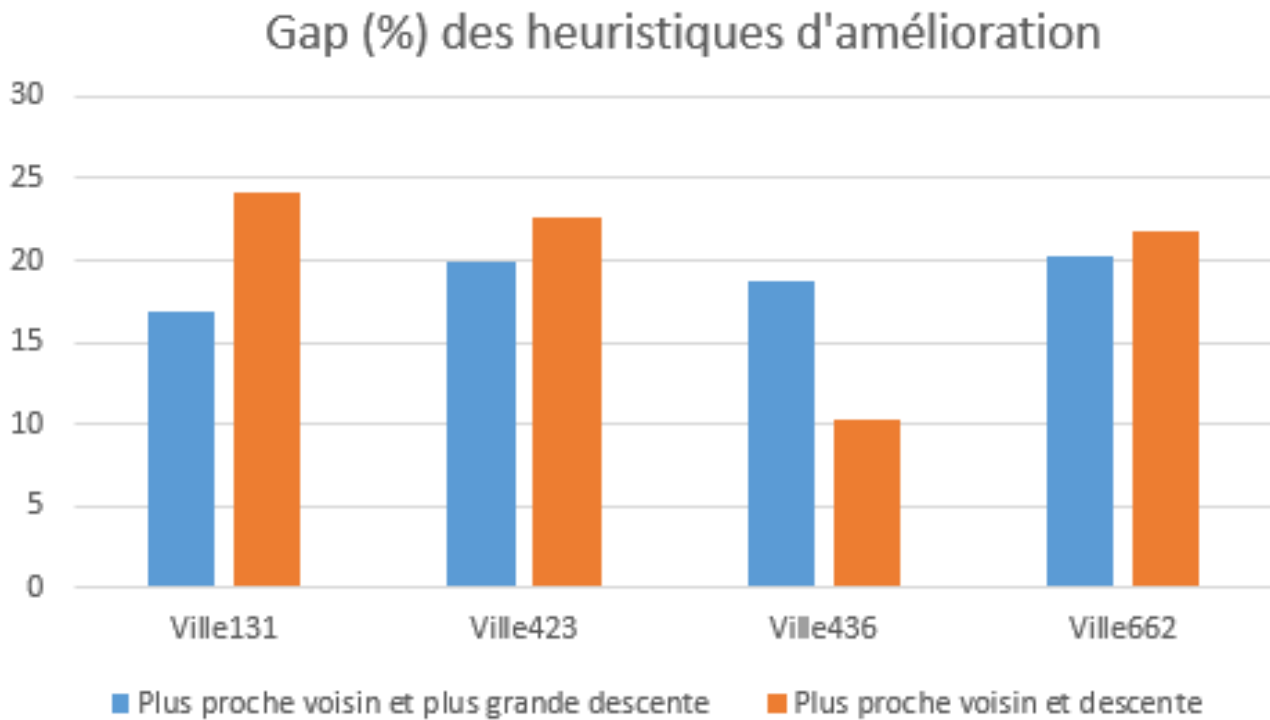


FIGURE 1.13 – Heuristiques d'amélioration. Gap des solutions obtenues de plusieurs tailles d'instances du TSP. Le nombre présent à côté du mot ville représente la taille de l'instance.

- La métaheuristique qui donne le meilleur coût est GRASP.
- La technique de descente et la technique de plus grande descente trouvent des tournées ayant des coûts proches.
- Le plus mauvais coût est obtenu par l'algorithme du plus proche voisin randomisé.
- Lorsqu'on fait de la recherche locale sur une tournée construit par l'heuristique du plus proche voisin on constate qu'il y a diminution du coût des solutions.
- En faisant plusieurs itérations de la recherche locale (plus grande descente) on constate qu'on décroît vers l'optimum mais le temps de calcul croît aussi.
- En faisant une recherche locale après un recuit simulé on constate qu'il y a une amélioration du coût d'une solution.
- Lorsqu'on couple un recuit simulé et une recherche locale (plus grande descente) on constate qu'on abouti au même résultat peu importe l'ordre (recuit simulé puis recherche locale ou recherche locale puis recuit simulé). Mais en faisant

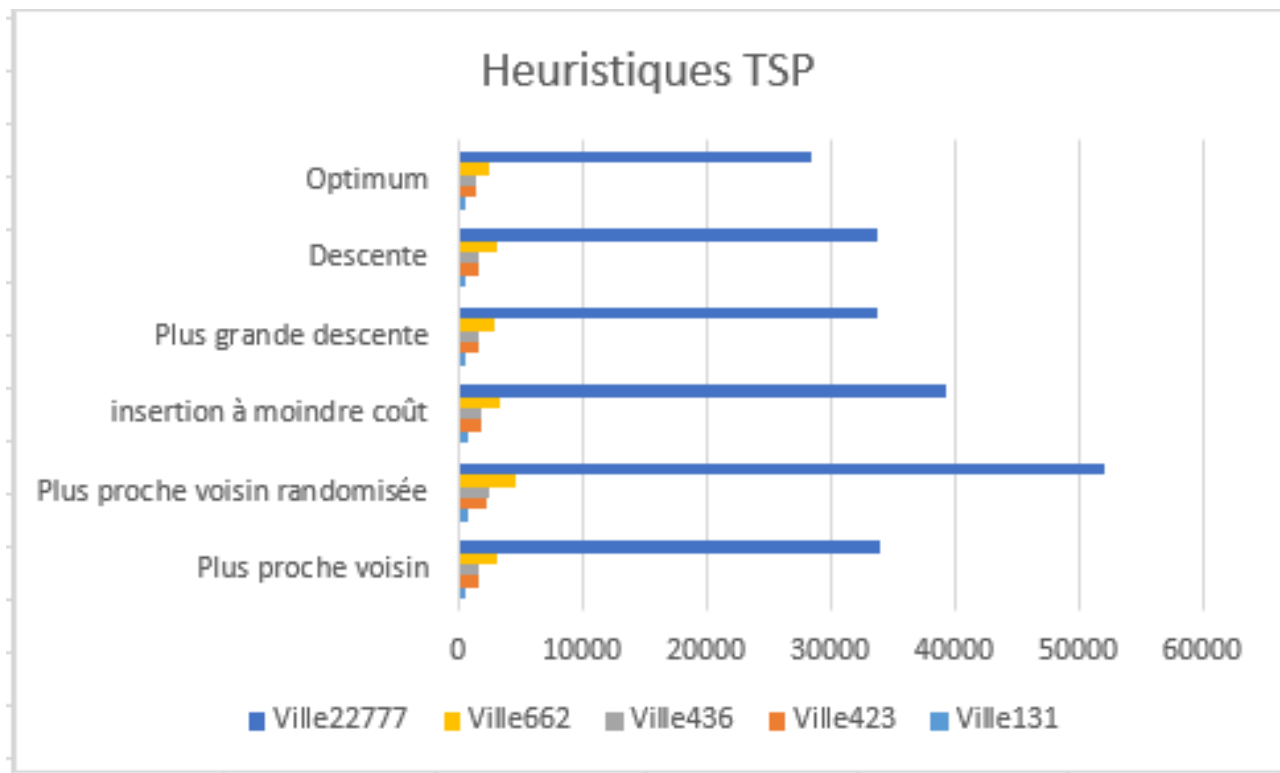


FIGURE 1.14 – Coûts des solutions obtenues de plusieurs tailles d’instances du TSP. Le nombre présent à côté du mot ville représente la taille de l’instance.

d’abord la recherche locale on constate que l’exécution est plus rapide. L’on constate que selon ces données, la recherche locale améliore plus une solution qu’un recuit simulé.

- En faisant une recherche locale (plus grande descente), lors de la recherche des paramètres du 2-opt, on constate que lorsqu’on parcourt la chaîne dans le sens inverse de la tournée construite et que l’instance est de grande taille on a diminution du coût des solutions.

1.14 Gap

Le Gap en % se calcule à l’aide de la formule $\frac{cout-opt}{opt} * 100$. *opt* est l’optimum.

1.14.1 Instance de taille 131

Le tableau 1.1 montre le coût et le temps CPU de l’expérimentation des algorithmes étudiés sur l’instance de 131 villes.

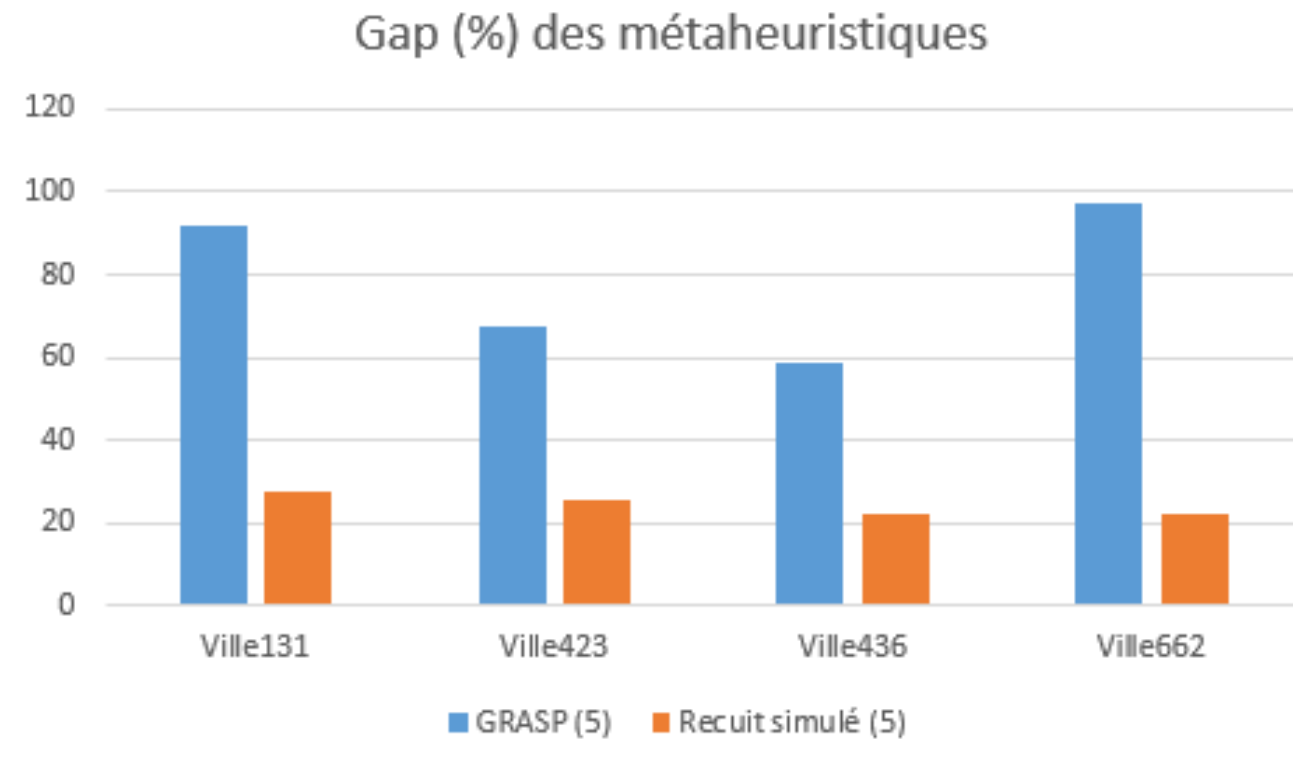


FIGURE 1.15 – Métaheuristiques. Gap des solutions obtenues de plusieurs tailles d’instances du TSP. Le nombre présent à côté du mot ville représente la taille de l’instance. Le paramètre de GRASP et du recuit simulé utilisé lors de cette expérimentation est 5.

1.14.2 Instance de taille 423

Le tableau 1.2 montre le coût et le temps CPU de l’expérimentation des algorithmes étudiés sur l’instance de 423 villes.

1.14.3 Instance de taille 436

Le tableau 1.3 montre le coût et le temps CPU de l’expérimentation des algorithmes étudiés sur l’instance de 436 villes.

1.14.4 Instance de taille 662

Le tableau 1.4 montre le coût et le temps CPU de l’expérimentation des algorithmes étudiés sur l’instance de 662 villes.

Taille instance(NB villes)	Méthodes	Temps CPU	Gap (%)
131	H. plus proche voisin	3	24,64
131	plus proche voisin randomisé	4	58,51
131	H. insertion	4	35,81
131	RL. descente	3	24,11
131	RL. plus grande descente	3	16,84
131	M. GRASP(5)	6	91,66
131	M. recuit simulé(5)	5	27,30

TABLE 1.1 – Instance de taille 131.

Taille instance(NB villes)	Méthodes	Temps CPU	Gap(%)
423	H. plus proche voisin	12	22,78
423	plus proche voisin randomisé	12	65,56
423	H. insertion	15	32,16
423	RL. descente	13	22,63
423	RL. plus grande descente	14	19,92
423	M. GRASP(5)	17	67,39
423	M. recuit simulé(5)	13	25,71

TABLE 1.2 – Instance de taille 423.

Taille instance(NB villes)	Méthodes	Temps CPU	Gap(%)
436	H. plus proche voisin	13	20,58
436	plus proche voisin randomisé	13	76,50
436	H. insertion	12	28,06
436	RL. descente	12	10,30
436	RL. plus grande descente	13	18,78
436	M. GRASP(5)	20	58,35
436	M. recuit simulé(5)	15	21,96

TABLE 1.3 – Instance de taille 436.

Taille instance(NB villes)	Méthodes	Temps CPU	Gap(%)
662	H. plus proche voisin	22	21,88
662	plus proche voisin randomisé	22	86,90
662	H. insertion	23	37,92
662	RL. descente	21	21,72
662	RL. plus grande descente	25	20,29
662	M. GRASP(5)	35	96,97
662	M. recuit simulé(5)	25	22,40

TABLE 1.4 – Instance de taille 662.

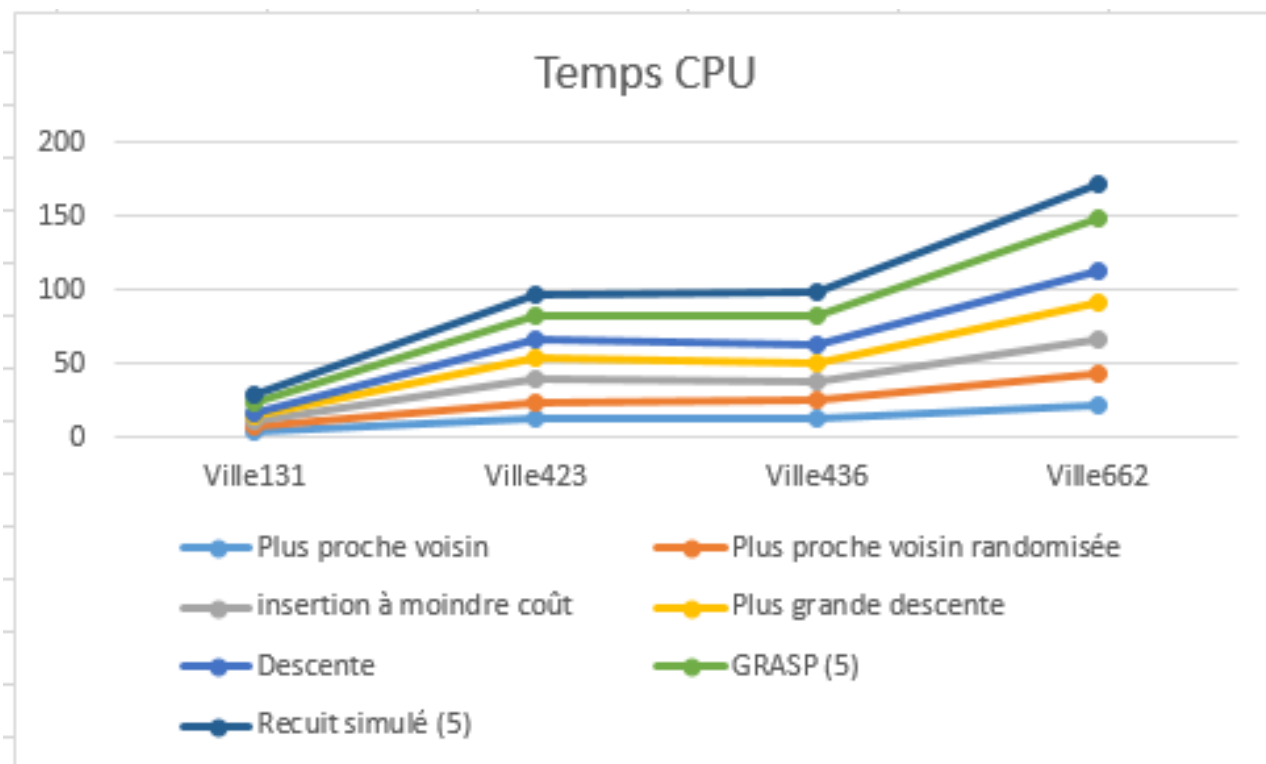


FIGURE 1.16 – Temps d’exécution obtenu de plusieurs tailles d’instances du TSP. Le nombre présent à coté du mot ville représente la taille de l’instance.

1.15 Complexité

D’après [bookTEg], « La théorie de la complexité en temps des algorithmes a pour objet d’étudier l’évolution du temps de calcul d’un algorithme en fonction de la dimension du problème à traiter. ». Nous allons évaluer la complexité des algorithmes étudiés dans ce document.

- L’heuristique du plus proche voisin (algorithme 4) : la procédure *Choisir* qui consiste à choisir la ville la plus proche de la ville courante *villeCourante* se fait en $O(n)$, avec n le nombre de villes car il n’y a qu’une seule boucle *pour* qui parcourt l’ensemble des villes. Cette procédure est faite pour toutes les n villes donc ce qui fait $O(n \times n)$. Donc cette heuristique a une complexité de $O(n^2)$.
- L’heuristique d’insertion à moindre coût (algorithme 6) : la procédure qui consiste à choisir pour une ville sa position dans la tournée se fait en $O(n)$ au pire des cas car il faut parcourir toutes les arêtes de la tournée. Cette procédure est répétée $n - 1$ fois ce qui fait $O(n \times n)$. Donc cette heuristique a une complexité de $O(n^2)$.

- La recherche locale avec la technique de descente (algorithme 14) Pour la première ville, on fera $n - 2$ comparaisons pour avoir un couple améliorant, pour la deuxième ville on fera $n - 3$ comparaisons ainsi de suite et pour la $n - 1$ ème ville on ne fera aucune comparaison donc on obtient une complexité de $O(n^2)$.

Le tableau 1.5 montre la complexité des algorithmes étudiés.

Méthodes	Complexité en temps
H. plus proche voisin	$O(n^2)$
plus proche voisin randomisé	$O(n^2)$
H. insertion	$O(n^2)$
RL. descente	$O(n^2)$
RL. plus grande descente	$O(n^2)$

TABLE 1.5 – Complexité en temps. n est le nombre de ville de l'instance du problème

Sixième partie

Arbres et arborescences

1.16 Propriétés des arbres

Définition 3 [sakarovitchoptimisation1984] *Un graphe est défini par deux ensembles V et E dits ensemble de sommets et d'arêtes respectivement ; deux applications I et $T : E \rightarrow V$ qui associent à chaque arête ses extrémités.*

Exemple 4 $V = \{1, 2, 3, 4, 5\} ; E = \{a, b, c, d, e, f\}$

$$I(a) = 1, T(a) = 2$$

$$I(b) = 1, T(b) = 5$$

$$I(c) = 2, T(c) = 3$$

$$I(d) = 5, T(d) = 4$$

$$I(e) = 4, T(e) = 3$$

$$I(f) = 5, T(f) = 5$$

Définition 4 [sakarovitchoptimisation1984] *Un graphe est connexe lorsque pour tout sommet x et y appartenant à ce graphe, il existe toujours un chaîne qui relie x et y .*

Le graphe de l'exemple 4 est connexe.

Définition 5 [sakarovitchoptimisation1984] *Un arc et une arête u d'un graphe $G = (V, E)$ orienté et non orienté respectivement, dont la suppression augmente le nombre de composantes connexes est appelé isthme.*

Un isthme n'appartient à aucun cycle car, si tel est le cas, en supprimant u on aura pas d'augmentation du nombre de composantes connexes.

Définition 6 [sakarovitchoptimisation1984] *Un arbre est un graphe connexe sans cycle.*

Soit $G = (V, E)$ un graphe, avec $|V| = n$, Si G est connexe alors $|E| \geq n - 1$, Si G est sans cycle alors $|E| \leq n - 1$. Donc dans un arbre nous avons la relation $|E| = n - 1$ est vérifiée.

Exemple 5 $G = (V, E)$ est un arbre avec $V = \{1, 2, 3, 4, 5\} ; E = \{a, b, c, d, e, f\}$

$$I(a) = 1, T(a) = 2$$

$$I(b) = 1, T(b) = 5$$

$$I(c) = 2, T(c) = 3$$

$$I(d) = 5, T(d) = 4$$

Définition 7 [sakarovitchoptimisation1984] Une forêt est un graphe dont chaque composante connexe est sans cycle orienté.

1.17 Arborescences

Soit $G = (V, E)$ un graphe orienté.

Définition 8 Un sommet " a " du graphe G est une racine (respectivement une antiracine) s'il existe dans G un chemin joignant a à x (respectivement x à a) $\forall x \in V$.

Un nœud qui est une racine et une antiracine peut encore être vue comme un dépôt.

Définition 9 G avec $n \geq 2$ sommets est une arborescence de racine a si a est une racine de G et G est un arbre.

G avec $n \geq 2$ sommets est une anti-arborescence admettant le sommet a pour racine si a est une anti-racine de G et G est un arbre.

Une arborescence est un arbre mais la réciproque est fausse. Par exemple, $A = (V, E)$ est une arborescence avec $V = 1, 2, 3$, $E = a, b$, $I(a) = 1, T(a) = 2$, $I(b) = 2, T(b) = 3$ et comme $|V| = |E| + 1$ alors A est un arbre. Mais en ajoutant à A le nœud 4 et l'arc c tel que $I(c) = 4$, $T(c) = 3$ on obtient un arbre qui n'est plus une arborescence.

1.18 Problème de l'arbre couvrant de poids maximum (maximum spanning tree)

Soit G un graphe connexe dont les arêtes portent des poids, dans le problème de l'arbre de poids maximum, il s'agit de trouver un sous-graphe de G contenant tous les sommets, qui est un arbre couvrant tel que la somme des poids des arêtes de cet arbre soit maximale. La figure 1.17 montre un exemple d'arbre de poids maximum d'un graphe. Un arbre couvrant d'un graphe G est un arbre contenant tous les sommets de G .

Par la suite nous allons présenter deux algorithmes (Kruskal et Prim) qui permettent de trouver des solutions optimales à ce problème. Les algorithmes de Kruskal et Prim sont des algorithmes gloutons exactes car à chaque étape on fait des choix qui ne sont jamais remis en question et de plus les solutions obtenues sont optimales.

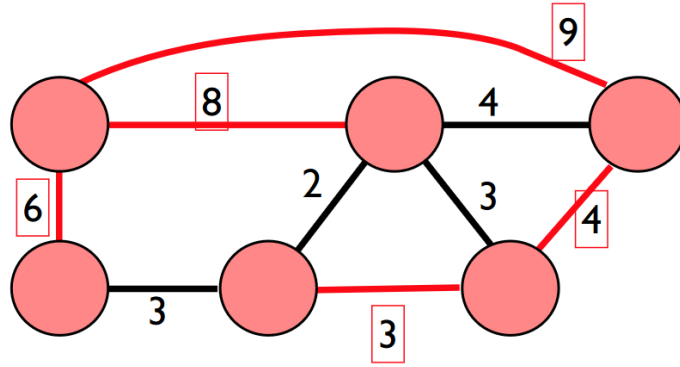


FIGURE 1.17 – Les poids encadrés des arêtes sont les poids de l’arbre de poids maximum de ce graphe

Ces algorithmes prennent en entrée un graphe connexe qu’on appellera $G = (V, E, c)$, avec $c : E \rightarrow \mathbf{R}$ est une fonction ou le vecteur poids $c = (c(e_1), c(e_2), \dots, c(e_m))$.

L’objectif est de construire un arbre de poids maximum $A = (V, T)$, avec $T \subseteq E$ et la somme des valeurs $\sum_{e_i \in T} c(e_i)$ maximale.

Pour avoir une solution au problème de l’arbre de poids minimum, en utilisant les algorithmes qui trouvent une solution au problème de l’arbre de poids maximum, il suffit de multiplier les poids des arêtes du graphe par -1 lorsque les poids ont le même signe.

1.18.1 L’algorithme de Kruskal

Pour déterminer l’arbre de poids maximum, initialement on ajoute à A l’arête de G ayant le plus grand poids en sachant qu’une arête doit être sélectionnée au plus une fois. Ensuite, on répète cette action tant que l’ensemble formé par les arêtes (A) choisies ne contient pas de cycle et que tous les nœuds n’ont pas été sélectionnés. Pour vérifier que A ne contient pas de cycle on vérifie que $|T| \leq |V| - 1$

L’algorithme 21 est l’algorithme de Kruskal [sakarovitchoptimisation1984]. On classe les poids des arêtes E dans l’ordre décroissant et on initialise l’ensemble T au vide. On parcourt l’ensemble des arêtes e_j classés et on ajoute à T une arête si le nouvel ensemble $T \cup e_j$ formé ne contient pas de cycle. Pour former un arbre, nous allons donc ajouter au plus $n - 1$ arêtes à l’ensemble T . La complexité de cet algorithme est $O(|E|)$.

Exemple 6 L’arbre de la figure 1.19 représente l’arbre obtenu au terme de l’application de l’algorithme Kruskal sur le graphe de la figure 1.18 dans lequel les étiquettes des arêtes

Algorithme 21 Kruskal

Entrées: $G = (V, E, c)$: graphe**Sorties:** T : ensemble d'arêtesClasser les poids des arêtes par ordre décroissant et les numéroter de 1 à m // $c(e_1) \geq c(e_2) \geq \dots \geq c(e_m)$ $T \leftarrow \emptyset$ **Tant que** $|T| < |V| - 1$ **faire** **Si** $(V, T \cup e_j)$ ne contient pas de cycle **alors** $T \leftarrow T \cup e_j$ **Fin si****Fin tant que**Retourner T

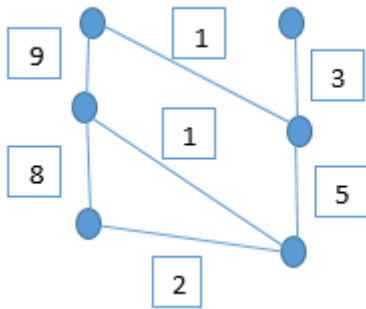


FIGURE 1.18 – Avant

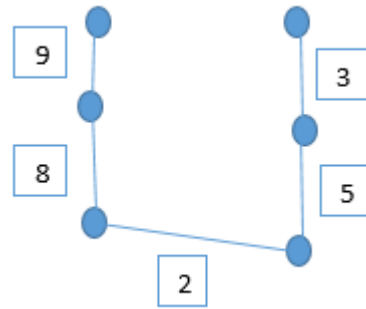


FIGURE 1.19 – Après

FIGURE 1.20 – Exemple d'application de l'algorithme Kruskal

représentent leurs poids.

Il existe une autre variante de l'algorithme de Kruskal qui consiste à initialiser A à G puis, on enlève à A des arêtes ayant le poids le plus petit tant que A reste connexe. Les paramètres d'entrée de l'algorithme de la variante de Kruskal sont pareilles que ceux de l'algorithme de Kruskal. On classe les poids e_j des arêtes par ordre croissant. On initialise T à E . On parcourt l'ensemble des arêtes e_j classés et on enlève à T une arête si le nouvel ensemble $T \setminus \{e_j\}$ formé est connexe. Pour former un arbre, T doit contenir $n - 1$ arêtes. L'algorithme 22 est l'algorithme de la variante de Kruskal [sakarovitchoptimisation1984].

La solution obtenue par ces deux variantes de Kruskal est optimale car à chaque étape on conserve l'arête ayant le plus grand poids et donc on satisfait la contrainte qui est de maximiser le poids des arêtes.

Algorithme 22 Variante Kruskal

Entrées: $G = (V, E, c)$: graphe**Sorties:** T : ensemble d'arêtesClasser les poids des arêtes par ordre croissant et les numéroter de 1 à m // $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$ $T \leftarrow E$ **pour** $i = 1$ **à** m **faire****Si** $(V, T \setminus \{e_j\})$ est connexe **alors** $T \leftarrow T \setminus \{e_j\}$ **Fin si****fin pour**Retourner T

1.18.2 L'algorithme de Prim

Soit $G = (V, E, c)$ un graphe, l'algorithme de Prim est basé sur la propriété suivante : si (V_1, V_2) est une partition de V , un arbre maximal contient toujours l'arête $i \in V_1, j \in V_2$ de valeur maximale. Une partition d'un ensemble V est une famille de parties non vides de V , disjointes deux à deux, et dont la réunion est l'ensemble V . L'initialisation de l'algorithme de Prim consiste à choisir un sommet quelconque $x \in V$ et poser $V_1 = \{x\}$, $V_2 = V \setminus V_1$. La phase itérative consiste à déterminer l'arête e_{kl} vérifiant $c(e_{kl}) = \text{Max}_{i \in V_1, j \in V_2} (c(e_{ij}))$ et poser $V_1 \leftarrow V_1 \cup l$, $V_2 \leftarrow V \setminus V_1$. La solution est l'ensemble des e_{kl} trouvés. L'algorithme s'arrête lorsque $V_1 = V$.

L'algorithme 23 est l'algorithme de Prim [sakarovitchoptimisation1984].

Algorithme 23 Prim

Entrées: $G = (V, E, c)$: graphe**Sorties:** T : ensemble d'arêtes $T \leftarrow \emptyset$ $V_1 \leftarrow$ Choisir n'importe quel $x \in V$ $V_2 \leftarrow V \setminus V_1$ **Tant que** $V_1 \neq V$ **faire**Déterminer l'arête e_{kl} vérifiant $c(e_{kl}) = \text{Max}_{i \in V_1, j \in V_2} (c(e_{ij}))$ $T \leftarrow e_{kl}$ $V_1 \leftarrow V_1 \cup l$ $V_2 \leftarrow V \setminus V_1$ **Fin tant que**Retourner T

Si nous prenons par exemple le cas du voyageur de commerce avec un graphe de n sommets et de m arêtes, nous nous posons la question suivante : peut-on trouver une tournée en exécutant le procédé décrit à l'algorithme 24.

Algorithme 24 TSP - tree

Entrées: G : Graphe**Sorties:** T : Tournée

- 1: Construire l'arbre de poids minimum à l'aide de l'algorithme Kruskal ou Prim.
 - 2: Avec l'arbre trouvé, construire la plus longue chaîne élémentaire possible et la mettre dans T . Pour cela on utilisera l'algorithme de Dijkstra en sélectionnant la plus grande chaîne parmi les plus courts chemins trouvés.
 - 3: Ajouter à T les nœuds isolés (nœuds n'appartenant pas à la chaîne) en minimisant le poids des arêtes choisies et fermer la chaîne. On utilisera l'algorithme d'insertion à moindre coût.
 - 4: Retourner T
-

1. Construire l'arbre de poids minimum à l'aide de l'algorithme Kruskal ou Prim.

La complexité de l'algorithme de Kruskal est $O(n \times m)$ et celle de l'algorithme de Prim est $O(n^2)$.

2. Avec l'arbre trouvé, construire la plus longue chaîne élémentaire possible et la mettre dans T . Pour cela on utilisera l'algorithme de Dijkstra en sélectionnant la plus grande chaîne parmi les plus courts chemins trouvés.

Par la suite nous allons donc évaluer la complexité de l'algorithme de Dijkstra. S'il y a n sommets, l'algorithme nécessite au plus $n - 1$ étapes. À l'étape k , il faut calculer un minimum sur l'ensemble provisoires (de cardinal au plus égal à $n - k$), puis, pour le sommet choisi x , examiner ses successeurs non calculés (leur nombre est au plus égal au nombre d'arcs issus de x). Globalement, il y a donc au plus $n^2 + m$ opérations, où m est le nombre d'arcs du graphe (somme du nombre d'arcs issus de chacun des sommets). Comme $m \leq n^2$, le nombre d'opérations est, au pire, proportionnel au carré du nombre de sommets. Donc $O(n^2)$.

3. Ajouter à T les nœuds isolés (nœuds n'appartenant pas à la chaîne) en minimisant le poids des arêtes choisies et fermer la chaîne. On utilisera l'algorithme d'insertion à moindre coût.

Au pire des cas, on obtiendra un graphe en étoile et la complexité du processus de construction de la chaîne entière si on utilise l'heuristique d'insertion à moindre coût sera $(n - 3) * n^2$ donc $O(n^3)$.

La dernière étape est la randomisation de l'algorithme 3.

Septième partie

Algorithme de plus court chemin

Définition 10 Un graphe orienté $G = (V, U)$ est défini par V un ensemble de sommets et U un ensemble d'arcs u qui relient, de manière orientée, un sommet $i \in V$ à un sommet $j \in V : i$ est l'extrémité initiale de u et j son extrémité terminale. Un arc u correspond donc à un couple ordonné (i, j) de sommets.

Pour un arc $u \in G$, $I(u)$ et $T(u)$ représenteront respectivement l'extrémité initiale et l'extrémité finale de l'arc u .

Exemple 7 $V = \{1, 2, 3, 4, 5\} ; U = \{a, b, c, d, e, f\}$

$$I(a) = 1, T(a) = 2$$

$$I(b) = 1, T(b) = 5$$

$$I(c) = 2, T(c) = 3$$

$$I(d) = 5, T(d) = 4$$

$$I(e) = 4, T(e) = 3$$

$$I(f) = 5, T(f) = 5$$

Dans un graphe orienté $G = (V, U)$, s'il existe un arc $u_{i,j}$, on dit que l'arc u est adjacent aux sommets i et j . Deux arcs adjacents à un même sommet sont dits adjacents. Une chaîne est une séquence d'arcs adjacents. Un chemin dans un graphe orienté est une chaîne dont tous les arcs sont orientés dans le même sens.

Soit $G = (V, U, c)$ un graphe orienté valué avec $c : U \rightarrow \mathbf{R}$ une fonction ou le vecteur poids $c = (c(u_1), c(u_2), \dots, c(u_m))$. Le but des algorithmes de plus court chemin est de trouver des chemins $(u_1, u_2, \dots, u_r, \dots, u_q)$ qui minimisent la valeur $\sum_{r=1}^q c(u_r)$. Si $c(u) = 1 \forall u \in U$ les algorithmes de plus court chemin chercheront le chemin le plus court, donc constitué du plus petit nombre d'arcs. Il existe deux situations : déterminer les plus courts chemins pour aller d'un sommet u à tous les autres sommets $U \setminus u$ ou déterminer les plus courts chemins entre toutes les paires de sommets.

1.19 Algorithme de Dijkstra

L'algorithme de Dijkstra présenté à l'algorithme [bookTEg] sert à résoudre le problème du plus court chemin. Il permet déterminer les plus courts chemins pour aller d'un sommet u à tous les autres sommets $U \setminus u$, par exemple, on peut déterminer un plus court

chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région. Plus précisément, il calcule des plus courts chemins à partir d'une source dans un graphe orienté pondéré par des réels positifs.

L'algorithme prend en entrée un graphe orienté valué G par des réels positifs et un sommet source. Il s'agit de construire progressivement un sous-graphe dans lequel sont classés les différents sommets par ordre croissant de leur distance minimale au sommet de départ. La distance correspond à la somme des poids des arcs empruntés.

Au départ, on considère que les distances de chaque sommet au sommet de départ sont infinies, sauf pour le sommet de départ pour lequel la distance est nulle. Le sous-graphe de départ est l'ensemble vide.

Au cours de chaque itération, on choisit en dehors du sous-graphe un sommet de distance minimale et on l'ajoute au sous-graphe. Ensuite, on met à jour les distances des sommets voisins de celui ajouté. La mise à jour s'opère comme suit : la nouvelle distance du sommet voisin est le minimum entre la distance existante et celle obtenue en ajoutant le poids de l'arc qui se trouve entre ce sommet et le sommet précédemment ajouté au sous-graphe. On s'arrête lorsque le sous-graphe contient tous les sommets du graphe G . uij signifie qu'il existe un arc qui va du sommet i au sommet j .

Algorithme 25 Dijkstra

Entrées: $G = (V, U, c)$: graphe orienté,

$x = 1$: sommet de départ

Sorties: D : ensemble d'arêtes

$\lambda_1 \leftarrow 0$ // λ_i représente l'étiquette du sommet i

pour tous les $i \neq 1$, $\lambda_i \leftarrow c(u_{1i})$

$p(i) \leftarrow 1$ // $p(i)$ est le sommet qui permet d'obtenir l'étiquette λ_i

$D \leftarrow 1$ // D est l'ensemble des sommets à étiquette définitive $D \in X$

Tant que $D \neq X$ **faire**

déterminer la plus petite étiquette provisoire $\lambda_i = \min_{j \in X \setminus D} \lambda_j$; si elle devient définitive alors $D \leftarrow D \cup i$

calculer les nouvelles étiquettes provisoires $\lambda_j = \min(\lambda_j, \lambda_i + c(u_{ij}))$ avec $j \in X \setminus D$.

Si λ_j a diminué **alors**

$p(j) \leftarrow i$

Fin si

Fin tant que

Retourner D, P

Exemple 8 Soit le $G = (V, U, c)$ avec $V = \{1, 2, 3, 4, 5, 6, 7\}$,

$U = \{u_{12}, u_{13}, u_{34}, u_{43}, u_{42}, u_{53}, u_{25}, u_{62}, u_{27}, u_{67}, u_{56}, u_{65}\}$

D	λ_1	λ_2	λ_3	λ_4	λ_5	λ_6	λ_7
1	0	8(1)	2(1)	∞	∞	∞	∞
1,3	-	8	-	5(3)	11(2)	∞	18(2)
1,3,4	-	7(4)	-	-	11(2)	15(5)	18(2)
1,3,4,2	-	-	-	-	10(2)	15(5)	17(6)
1,3,4,2,5	-	-	-	-	-	14(5)	17(6)
1,3,4,2,5,6	-	-	-	-	-	-	16(6)

Les poids des arcs sont : $c(u_{12}) = 8$, $c(u_{13}) = 2$, $c(u_{34}) = 3$, $c(u_{43}) = 4$, $c(u_{42}) = 2$,
 $c(u_{53}) = 5$, $c(u_{25}) = 3$, $c(u_{62}) = 5$, $c(u_{27}) = 10$, $c(u_{67}) = 2$, $c(u_{56}) = 4$, $c(u_{65}) = 2$

On trouve ainsi les plus courts chemins : 1-3-4-2 de valeur 7, 1-3 de valeur 2, 1-3-4 de valeur 5, 1-3-4-2-5 de valeur 10, 1-3-4-2-5 de valeur 14 et 1-3-4-2-5-6-7 de valeur 16.

Huitième partie

Problèmes de flots maximum

1.20 Énoncé

Flot maximum

1.21 Algorithme de Ford - Fulkerson

Algorithme 26 Ford - Fulkerson

Entrées: $G = (V, U, c)$: graphe orienté

Sorties: F : Flot maximum

initialiser le flot F de tous les arcs à 0

Tant qu'il existe un chemin améliorant p de capacité ϵ , modifier les arcs correspondant dans G

Retourner F

Neuvième partie

Algorithmes d'approximations pour le problème du voyageur de commerce

Soient un ensemble de villes $1, 2, \dots, n$ et une matrice symétrique ($c_{ij} = c_{ji}$) $C = (c_{ij})$ de taille $n \times n$ qui représente la distance de la ville i à la ville j . Si $c_{ij} \neq c_{ji}$ alors C est une matrice asymétrique. Cette matrice représente la matrice d'adjacence d'un graphe complet et non orienté dont les arêtes portent les distances entre les villes et les nœuds représentent les villes. Le but du voyageur est de construire une tournée qui passera par toutes les villes et qui minimisera la distance parcourue.

Une tournée est vu comme :

- Un cycle hamiltonien.

Un cycle qui passe exactement une fois par chaque sommet d'un graphe est dit « hamiltonien ». Lorsque le nombre de sommet n devient de plus en plus grand, déterminer le cycle hamiltonien de coût minimal en faisant une liste exhaustive des cycles hamiltoniens du graphe entraînerait une explosion combinatoire, car le nombre de cycles hamiltoniens d'un graphe complet de n sommets est $(n-1)!/2$.

Dans le cas d'un graphe non complet $G = (V, E)$, on peut vérifier qu'il existe un cycle dans le graphe si nous faisons la transformation suivante :

$$c_{ij} = \begin{cases} 1 & \text{si } (i, j) \in E \\ n+2 & \text{sinon.} \end{cases}$$

Si un cycle hamiltonien (C.H) existe alors le coût $C(C.H) = n$ sinon $C(C.H) \geq 2n+1$

- Une permutation cyclique de villes.

Une permutation cyclique de villes est une traversé ordonnée des villes tel que chaque ville est classée de façon unique et la ville de départ est identique à la ville d'arrivée.

Un algorithme est une k -approximation (k représente le facteur d'approximation, plus k est proche de 1, meilleure est l'approximation) lorsque la solution S fournie par cet algorithme se trouve à une "distance" k de la solution optimale OPT . Par exemple dans le cas du voyageur de commerce, si on appelle T le tournée construite par un algorithme d'approximation, $cout(T) \leq k.OPT$. OPT représente le coût d'une la tournée optimale, $cout$ calcule le coût d'une solution pour l'algorithme d'approximation.

1.22 Algorithme d'insertion au plus proche

On construit la tournée triviale puis on choisit à chaque étape la ville la plus proche de la tournée pour l'ajouter dans la tournée (on insère le nœud le plus proche de la tournée). Dans cette partie nous allons montrer que l'algorithme d'ajouts successifs au plus proche est une 2-approximation. Soit T la tournée construite par l'algorithme d'insertion au plus proche, montrons que $c(T) \leq 2OPT$, c est la fonction qui calcule le coût.

Pour faire cette démonstration nous allons utiliser l'algorithme de Prim du problème de l'arbre de poids minimum, le coût de la tournée déterminée avec l'algorithme d'insertion (AI) au plus proche est au moins le coût de l'arbre de poids minimum (APM) déterminé avec l'algorithme de Prim ($c(T) \geq c(APM)$).

Soit G un graphe constitué de l'arête (i, j) , nous voulons ajouter le nœud k à G , la différence de coût de l'arbre de poids minimum formé est $c_k = c_{ik} + c_{ij} - c_{kj}$ si $c_{ik} \leq c_{kj}$. En généralisant ceci à n nœuds on obtient $C = \sum_{k=2}^n c_k$ or d'après l'inégalité triangulaire, $c_{ij} - c_{kj} \leq c_{ik}$ donc $c_k \leq 2c_{ik}$.

Avec la méthode de construction nous pouvons affirmer que ($c(T) \leq 2c(APM)$). Or $c(APM) = \sum_{k=2}^n c_{ik}$ et $2c(APM) \leq 2OPT$. Donc $c(T) \leq 2OPT$.

- Modules validés : 5 au 7 décembre 2018 art de négociier (SP14,groupe 1), 24 et 25 janvier 2019 (Communication)(SP11,groupe 1), 5 et 6 février 2019 Du corps à la parole (SP), Processus d'innovation (SPI 2), Ethique et intégrité scientifique+conférence, enseigner à l'université
- Logiciels : OPL de IBM CPLEX, Gusek, Microsoft Visual Studio 2017.
- Language : C++.

Chapitre 2

Définition et notations du système

Dixième partie

Formalisation du système

2.1 Introduction

Les véhicules autonomes ou partiellement autonomes ont besoin d'énergie pour fonctionner. De nos jours, les principales sources d'énergie sont les ressources fossiles par exemple le pétrole, le gaz naturel et le charbon. Ces ressources fossiles mettent des millions d'années (environ 250 millions) à se former, de plus, elles sont présentes en quantités limitées et se renouvellent beaucoup plus lentement que nous les utilisons. Dans quelques années, l'humanité pourra se retrouver confronté à un problème : le manque de ressources fossiles pour combler tous les besoins. Face à cette difficulté, les constructeurs de véhicules autonomes ou partiellement autonomes font appel à la mixité d'énergie. Il s'agit, en effet, de coupler plusieurs types d'énergies pour alimenter les véhicules autonomes ou partiellement autonomes. Cette solution consiste souvent à coupler les ressources fossiles et les énergies renouvelables. Les tournées des véhicules autonomes ou partiellement autonomes étant planifiées avant leurs mises en service, on doit tenir compte lors de la planification, des instants durant lesquels les véhicules iront se recharger en utilisant l'énergie non renouvelable et l'énergie renouvelable. L'énergie renouvelable sur laquelle l'on travaillera est l'hydrogène, nous devons donc considérer le processus de fabrication de celui-ci et l'énergie non-renouvelable sur laquelle l'on travaillera sera l'électricité. L'objectif de ce projet est de construire et d'optimiser des algorithmes de planification de tournées de véhicules en utilisant comme source énergétique une combinaison de l'énergie électrique et hydrogène.

Dans un premier temps nous allons travailler sur des problèmes de tournées de véhicules avec collectes (*General Pick Up and Delivery*). Nous considérons que la structure des demandes est *one-to-one* c'est-à-dire qu'à chaque demande de collecte correspond une unique demande de livraison située sur une autre station. On a donc des requêtes de transport et les véhicules doivent acheminer les marchandises de leurs stations d'« origine » vers leurs stations « destination ». Plusieurs décisions devront donc être prises : à quel moment un véhicule ira-t-il se charger en électricité ou en hydrogène ? A quel moment on produit de l'hydrogène ? Quelle station visite un véhicule durant un instant donné ? Quel chemin emprunter pour atteindre une station ?

On aura un dépôt (voir figure 2.1) dont les rôles seront de :

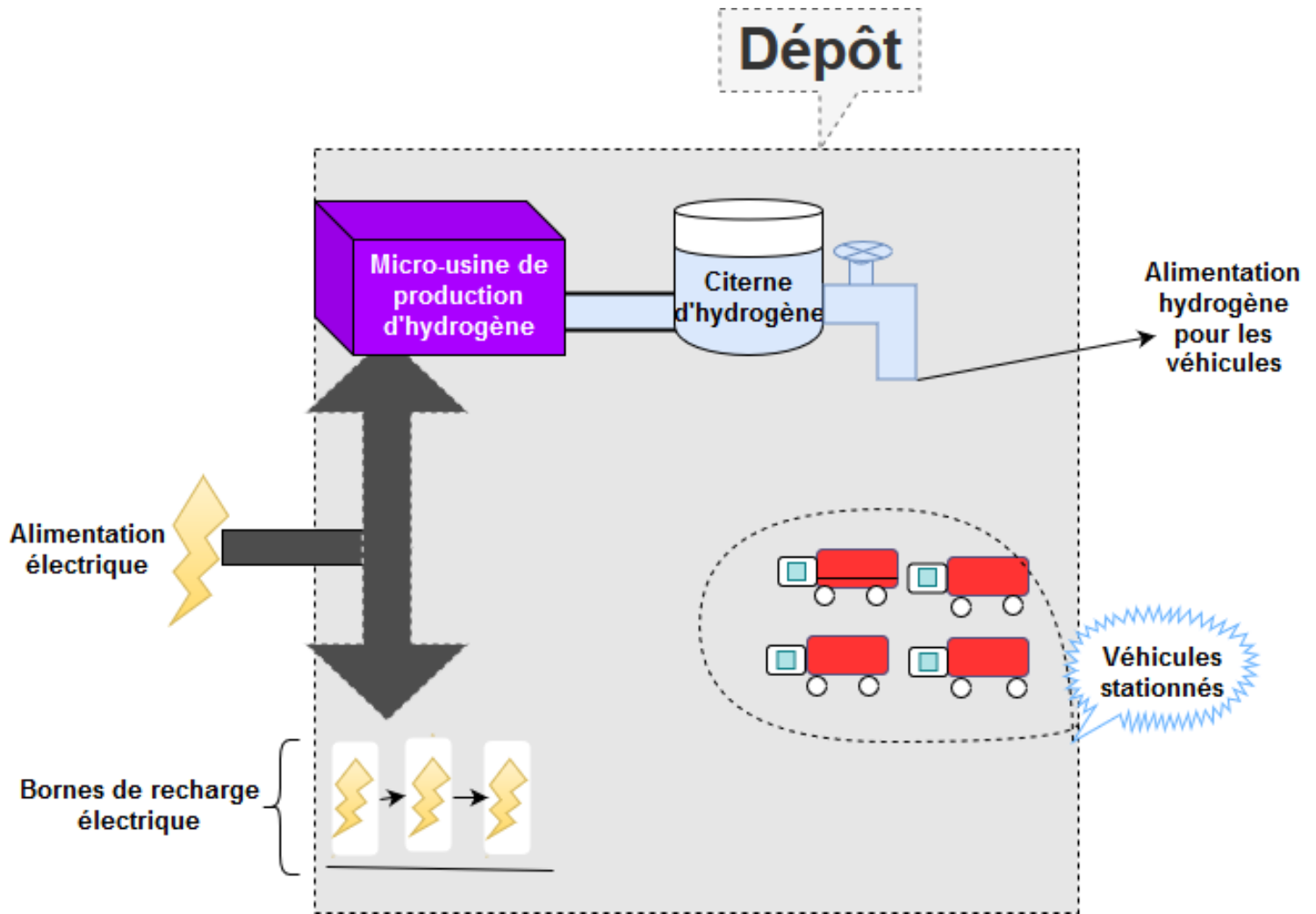


FIGURE 2.1 – Illustre les composantes du dépôt. On a la micro-usine qui fabrique de l'hydrogène, la citerne en hydrogène qui alimente les véhicules en hydrogène, les bornes de recharge électrique qui permettent de recharger les véhicules en électricité et les véhicules qui effectueront les tâches de réaliser les requêtes.

- recharger en électricité les véhicules
- recharger en hydrogène les véhicules
- produire de l'hydrogène, ce qui sera fait par une micro-usine
- alimenter la micro-usine en électricité

De plus, on aura un ensemble de stations (voir figure 2.2) dont certaines seront des points de collectes ou de livraisons des charges des requêtes. Ces stations sont connectées et cela forme un réseau réel car ça représente exactement les carrefours et les routes de l'espace réel sur lequel les véhicules vont circuler.

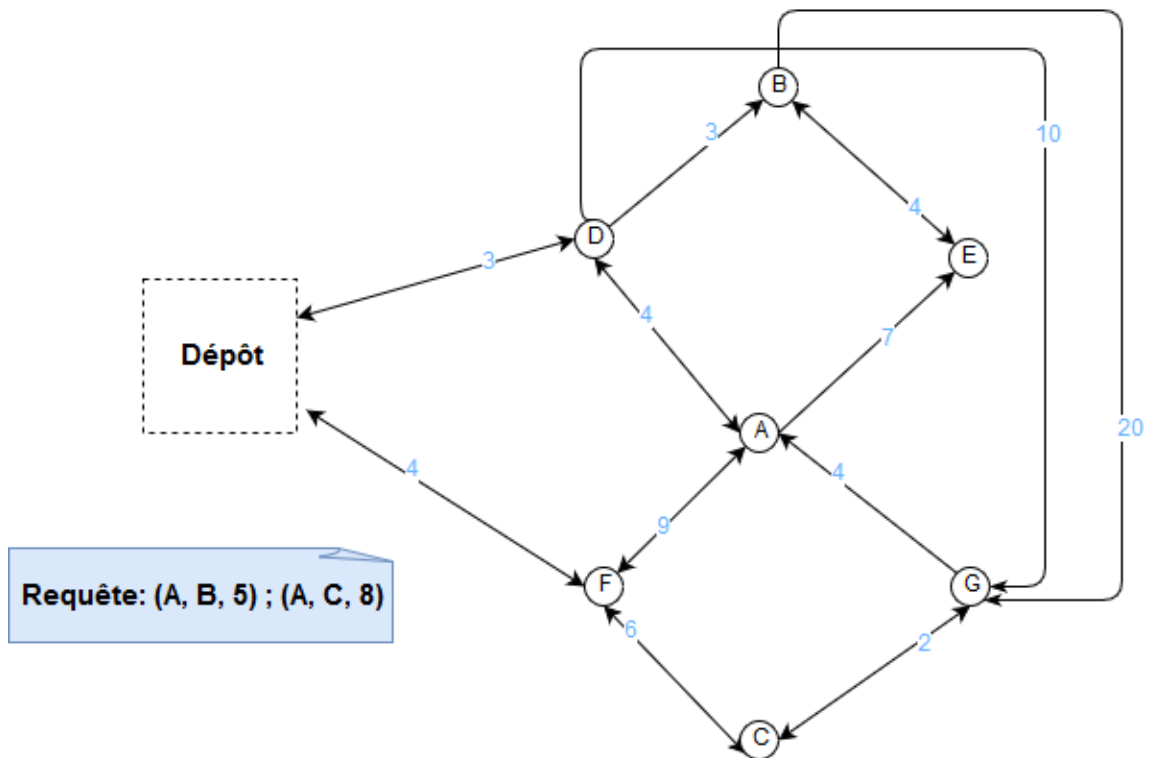


FIGURE 2.2 – Réseau réel représentant l’ensemble des stations (par exemple les stations A,B,C) et les temps de déplacement pour se déplacer d’une station à une autre (par exemple pour se déplacer de A à D un véhicule fera 4 unité de temps). Nous avons deux requêtes associées à ce réseau réel, la première requête doit récupérer 5 unités de charge à A et l’apporter à B.

2.2 Description générale du système

2.2.1 Problème

On considère une flotte de véhicules ayant des tâches à réaliser (de type *Pick Up and Delivery*) sur un horizon de temps donné. Le but est de calculer des tournées qui permettent de réaliser l’ensemble de ces tâches tout en prenant en compte la consommation et le ravitaillement en énergie des véhicules. On cherche donc à synchroniser la production d’énergie avec la recharge des véhicules. Autrement dit, le but est à la fois de planifier l’activité des véhicules (calcul des tournées) et de planifier l’activité de la micro-usine (production de l’hydrogène pour les véhicules, recharge en hydrogène des véhicules et recharge en électricité des véhicules).

Un exemple de tournée est présenté à la figure 2.3, cette tournée effectuée par le véhicule 1 commence au dépôt puis parcourt les stations A et B (10 unité de charge sont

transportées de A à B), ensuite retourne au dépôt se charger en électricité ou en hydrogène, puis parcourt les stations D et C (15 unité de charge sont transportées de D à C), et enfin le véhicule retourne se garer au dépôt.

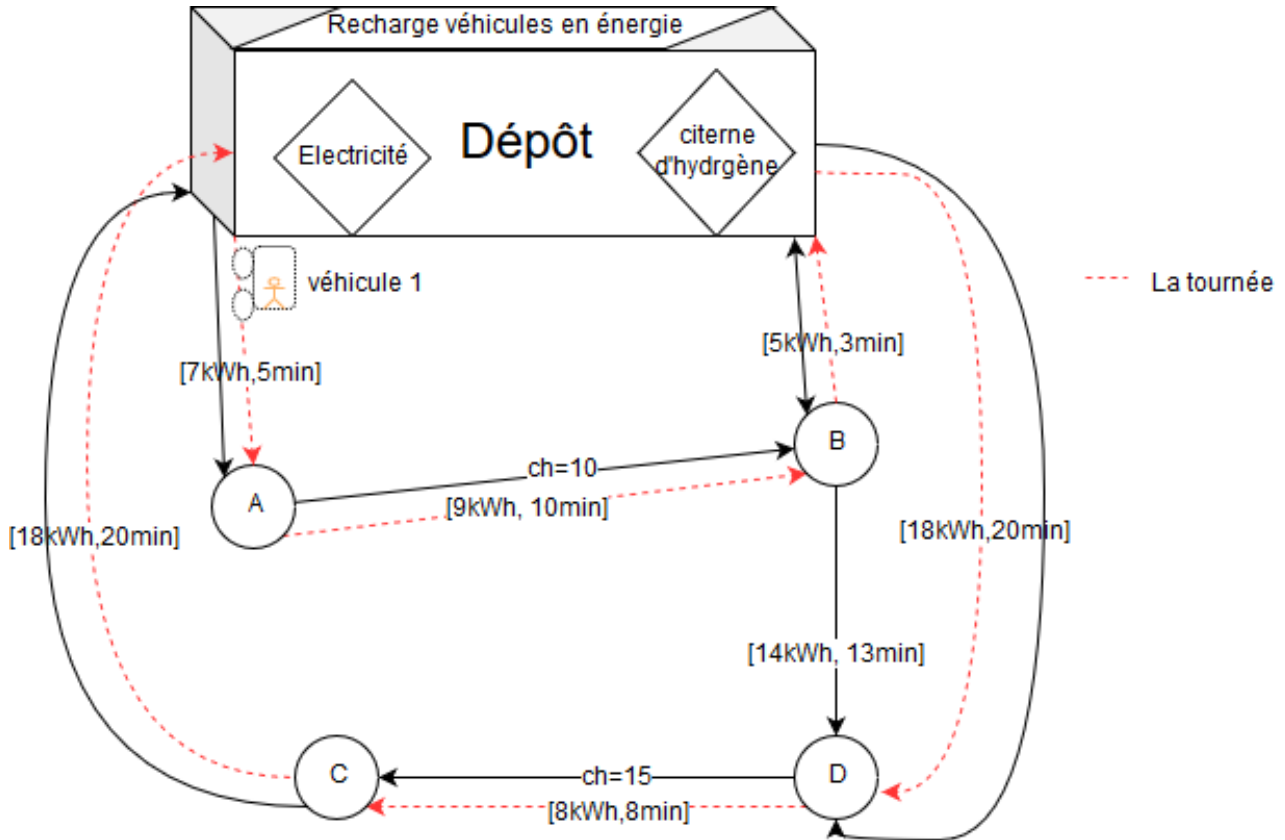


FIGURE 2.3 – Illustre une tournée (dessinée en pointillés). Les arcs pleins, les stations et le dépôt représentent le réseau réel. Les requêtes sont : (A,B,10) et (D,C,15)

On suppose que les véhicules possèdent à la fois une batterie (énergie électrique) et une pile à hydrogène (énergie hydrogène). Deux sources d'énergie sont donc possibles :

1. l'énergie électrique
2. l'énergie hydrogène

2.2.2 Caractéristiques de l'énergie

Caractéristiques de l'énergie électrique

On branche le véhicule sur une prise électrique, on a les caractéristiques suivantes :

- l'énergie est disponible de manière immédiate mais il faut un temps de chargement de la batterie

- il n'y a pas de stockage au dépôt
- on a une limite de puissance électrique c'est-à-dire une limite sur la quantité d'électricité qui peut être délivrée sur une même période. On a une puissance électrique par unité de temps qui arrive de l'extérieur vers le dépôt pour :
 1. alimenter les batteries des véhicules
 2. alimenter l'activité générale de la micro-usine de production d'hydrogène
- le prix est variable suivant les périodes (grille tarifaire)
- deux véhicules peuvent se charger simultanément en électricité
- on considère que la charge initiale en électricité peut ne pas être identique à tous les véhicules
- à chaque instant , la consommation électrique liée sur le dépôt à la recharge des véhicules et au fonctionnement de la micro-usine ne peut excéder un seuil de puissance disponible.

Caractéristiques de l'hydrogène

La production de l'hydrogène est soumise à un processus de fabrication qui nécessite de l'eau, de l'énergie solaire, de l'électricité et un temps de fabrication, on a les caractéristiques suivantes :

- le coût de *setup* est le coût de mise en fonctionnement du processus de fabrication
- le processus prend du temps pour produire l'hydrogène
- il y a un coût (fixe) par unité de temps
- la quantité d'hydrogène produite dépend de la météo (plus il y a de soleil, plus c'est rapide)
- l'hydrogène doit être stocké dans une citerne (capacité de stockage limitée) après production avant d'être utilisé par les véhicules
- on peut supposer que le temps de remplir le réservoir des véhicules en hydrogène est négligeable
- un véhicule à la fois au plus se charge sur la citerne d'hydrogène

- on considère que la charge initiale en hydrogène peut ne pas être identique à tous les véhicules
- on ne peut produire et consommer de l'hydrogène simultanément.

Le tableau 2.1 représente les actions qui peuvent se faire simultanément au dépôt (k_1 et k_2 sont deux véhicules).

	Production	Recharge hydrogène (k_1)	recharge électricité (k_1)
Production		×	✓
Recharge hydrogène (k_2)	×	×	✓
Recharge électricité (k_2)	✓	✓	✓

TABLE 2.1 – Tableau représentant les actions qui peuvent se faire simultanément au dépôt.

2.2.3 Critères de qualité

Il peut arriver que notre problème n'ait pas de solution réalisable, c'est-à-dire qu'on ne puisse effectuer les tâches de transport dans le temps imparti et avec l'énergie disponible. Un premier objectif va donc être de tester l'existence d'un planning réalisable et d'en construire un.

Si des plannings réalisables existent alors se pose la question de leurs qualités. Il existe deux pistes qui sont envisagées :

1. "écologique" : on utilise un maximum d'hydrogène et un minimum d'électricité. Une autre question se pose donc : comment prendre en compte le prix variable de l'électricité ?
2. "économique" : on planifie de sorte que les recharges en énergie coûtent le moins cher possible
3. critère de qualité sur les tournées de véhicules ??

Ici on se focalise dans un premier temps sur le coût économique.

2.2.4 Hypothèses

Avant de faire la formalisation du système, nous posons les conditions suivantes :

1. on a le droit de charger simultanément en hydrogène et en électricité

2. en ce qui concerne la quantité d'énergie présente dans les véhicules, on impose de finir comme on a démarré c'est-à-dire que le niveau de charge à la fin d'une tournée doit être au moins équivalent (\geq) au niveau de charge au début de cette tournée
3. Pour les stations différentes du dépôt (origines ou destinations de requêtes ou nœuds intermédiaires du transit réel) :
 - on connaît le temps pour aller d'une station à une autre et on le suppose fixe
 - on connaît l'énergie dépensée par les véhicules pour aller d'une station à une autre et on le suppose fixe
 - on considère le temps de service associés aux requêtes négligeable ou fixe

2.3 Formalisation du système

Pour formaliser le système global, nous l'avons subdivisé en deux sous-systèmes (voir figure 2.4) :

- le système « véhicules » qui est formé par les véhicules qui répondent, dans un réseau de transit, à des tâches de logistique urbaine ou interne à un site
- le système « production d'énergie » qui est formé par le dépôt, où sont effectuées des tâches de production et ré-alimentation d'énergie hydrogène et électrique.

Nous devons planifier ces deux systèmes en les synchronisant, cette dualité va nous conduire à formaliser notre problème, puis à le traiter algorithmiquement, en distinguant ces 2 sous-systèmes, et en identifiant leurs points d'articulation.

2.3.1 Système « véhicules »

Input du système « véhicules »

A) Input réel du système « véhicules »

Ce qu'on nous donne est :

(a) un réseau de transit $G = (STATIONS, E)$

- dans ce réseau on a un dépôt, et des nœuds qui peuvent être l'origine ou la destination d'une requête, ou les deux à la fois, ou encore un nœud

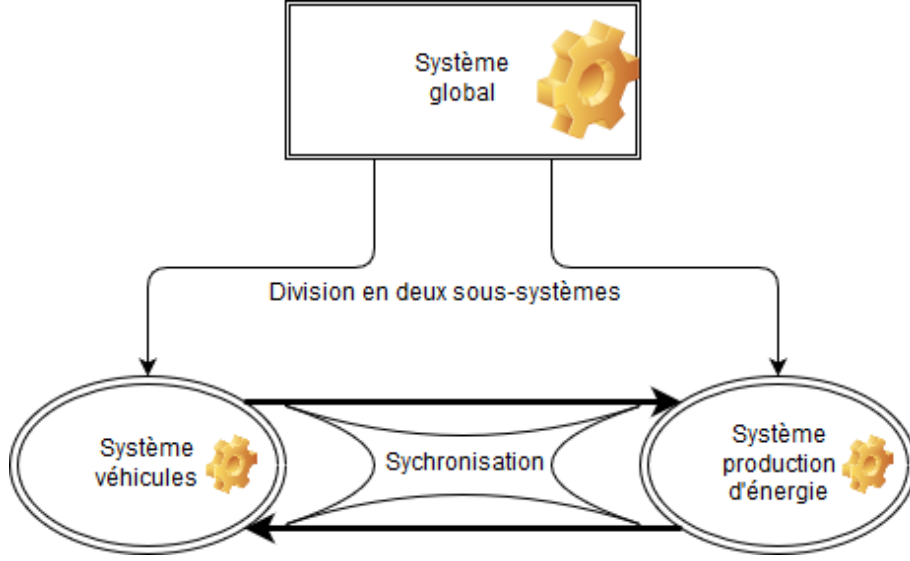


FIGURE 2.4 – Division du système global en deux sous-systèmes.

intermédiaire. $STATIONS = \{0, 1, \dots, N\}$ ~ ensemble de stations où la station 0 est le dépôt

— les arcs représentent les voies de passage entre deux nœuds. Chaque arc est muni de deux coefficients tmp et $energie$

— tmp ~ le tableau des temps qu'il faut pour aller d'une station à une autre $\forall (i, j) \in \{0, \dots, N\}, tmp[i][j] =$ temps mis par le véhicule pour aller de la station i à la station j . Ces temps sont calculés dans le réseau réel.

— $energie$ ~ le tableau des consommations d'énergie pour aller d'une station à une autre $\forall (i, j) \in \{0, \dots, N\}, energie[i][j] =$ consommation d'un véhicule pour aller de la station i à la station j . Ces consommations sont calculées dans le réseau réel.

(b) un ensemble de requête REQ , on note R le nombre de requêtes. Une requête est un triplet de la forme $(o, d, load)$ où $o \in \{1, \dots, N\}$ est la station origine, $d \in \{1, \dots, N\}$ la station destination et $load \in \{1, \dots, CAP\}$ la charge (quantité de marchandises à transporter) REQ ~ un tableau des requêtes (voir figure 2.5) constitué de trois lignes représentant l'origine, la destination et la charge de chaque requête $r \in \{1, \dots, R\}$.

— $REQ[r][origine]$ est la station origine de la requête r

- $REQ[r][destination]$ est la station destination de la requête r
- $REQ[r][load]$ est la charge de la requête r

	1	...	r	...	R
origine		...	I	...	
destination		...	J	...	
load		...	L	...	

FIGURE 2.5 – Représentation du tableau des requêtes.

(c) un ensemble de véhicule $\{1, \dots, K\}$, tous identiques et pour lesquels on connaît les informations suivantes :

- $K \sim$ le nombre de véhicules et l'ensemble des véhicules $\{1, \dots, K\}$, on notera k un véhicule
- les capacités
 - $CAP_load \sim$ la capacité maximum de chaque véhicule
 - $CAP_elect_veh \sim$ la capacité maximum en électrique des véhicules
 - $CAP_h2_veh \sim$ la capacité maximum en hydrogène des véhicules
- les temps de recharge
 - $tmp_h2 \sim$ le temps de recharge d'un véhicule en hydrogène . Cette valeur est fixe
 - $tmp_elect \sim$ le temps de recharge d'un véhicule en électricité par unité énergie
- les charges initiales
 - $init_elect(k) \sim$ la charge initiale en électricité du véhicule k
 - $init_h2(k) \sim$ la charge initiale en hydrogène du véhicule k

(d) la durée maximale $TMAX$ du processus d'exécution des requêtes

B) Pré-traitement de l'input réel : Duplication des stations et réseau virtuel

Nous allons faire un pré-traitement de l'input réel pour obtenir un réseau virtuel parce que :

- on veut éliminer les stations où il ne se passe rien
- on veut pouvoir identifier chaque station **par ce qui va s’y passer** : ça nous amène à faire des copies de stations physiquement identifiées, pour faire apparaître l’action qui va s’y passer, et la date de cette action.

Donc on a :

- (a) éliminer les stations qui ne sont ni dépôt, ni origine ou destination de requêtes
- (b) pour toutes stations, origine ou destination de requêtes, on va considérer cette station en fonction qu’elle est origine ou destination
- (c) pour le dépôt, on va distinguer :
 - le dépôt vu comme début de tournée pour un véhicule $k = \{1, \dots, K\}$
 - le dépôt vu comme fin de tournée pour un véhicule $k = \{1, \dots, K\}$
 - le dépôt vu comme point de chargement électrique
 - le dépôt vu comme point de chargement hydrogène
 - le dépôt vu comme point de production hydrogène

Nous allons effectuer un pré-traitement sur l’ensemble des stations *STATIONS* (appelées stations réelles) et sur l’ensemble des requêtes données en entrées, on obtient un ensemble X constitué de stations virtuelles qu’on divisera en cinq types :

- (a) $X_{req} = \{1, \dots, N_{Req}\} \sim$ ensemble des origines et destinations des requêtes, $N_{Req} = 2R$, car chaque requête a une origine et une destination, donc l’ensemble $\{1, \dots, R\}$ représente les origines des requêtes et l’ensemble $\{R + 1, \dots, N_{Req}\}$ représente les destinations des requêtes
- (b) $X_{depot_debut} = \{N_{Req} + 1, \dots, N_{Req} + K + 1\} \sim$ ensemble des copies du dépôt correspondant au début de chaque tournée. La tournée k démarre par $N_{Req} + k$ et $N_{Req} + K + 1$ est le début de la production d’hydrogène
- (c) $X_{depot_fin} = \{N_{Req} + K + 2, \dots, N_{Req} + 2K + 2\} \sim$ ensemble des copies du dépôt correspondant à la fin de chaque tournée. La tournée k finit par $N_{Req} + 2K + k$ et $N_{Req} + 2K + 2$ est la fin de la production d’hydrogène
- (d) Nous avons fait une estimation **pessimiste** du **nombre de fois maximum** que les véhicules iront se charger en électricité au dépôt.

$X_depot_elect = \{N_Req + 2K + 2, \dots, N_Req + 2K + K \times (N_max_elect)\}$
 \sim ensemble des copies du dépôt correspondant aux chargements d'électricité de chaque véhicule, où N_max_elect est le nombre de fois maximum qu'un véhicule va au dépôt se charger en électricité

- (e) Nous avons fait une estimation **pessimiste** du **nombre de fois maximum** que les véhicules iront se charger en hydrogène au dépôt.

$X_depot_h2 = \{N_Req + 2K + K \times (N_max_elect) + 1, \dots, N_Req + 2K + K \times (N_max_elect + N_max_h2)\}$ \sim ensemble des copies du dépôt correspondant aux chargements d'hydrogène de chaque véhicule, où N_max_h2 est le nombre de fois maximum qu'un véhicule va au dépôt se charger en hydrogène

- (f) Nous avons fait une estimation **pessimiste** du **nombre de fois maximum** que la micro-usine produira de l'hydrogène au dépôt.

$X_depot_prod = \{N_Req + 2K + K \times (N_max_elect + N_max_h2) + 1, \dots, N_Req + 2K + K \times (N_max_elect + N_max_h2) + N_max_prod\}$ \sim ensemble des copies du dépôt correspondant aux productions d'hydrogène, où N_max_prod est le nombre maximum de fois qu'on produit de l'hydrogène

donc, $X = X_req \cup X_depot_debut \cup X_depot_fin \cup X_depot_elect \cup X_depot_h2 \cup X_depot_prod$, c'est-à-dire que $X = \{1, \dots, N_Req + 2K + K \times (N_max_elect + N_max_h2) + N_max_prod\}$, X est l'ensemble de toutes les stations virtuelles qui ont été créées.

Exemple 9 La figure 2.6, illustre un exemple de pré-traitement des inputs d'un réseau réel :

(a) données

- $K = 2$
- $N_max_h2 = 3$
- $N_max_elect = 2$
- $N_max_prod = 1$
- $RED = \{(A, B, 20), (A, C, 30)\}$

(b) stations virtuelles :

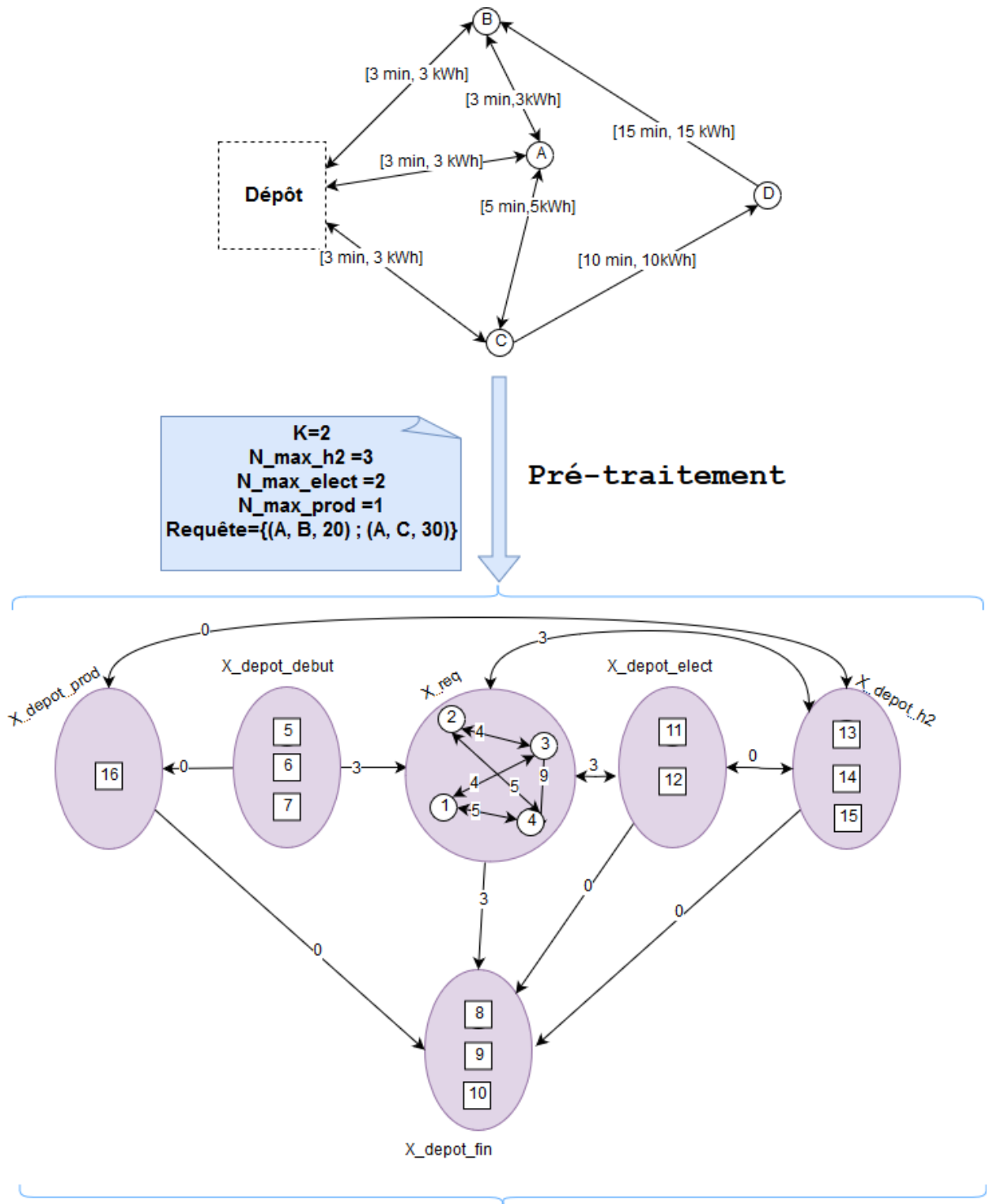


FIGURE 2.6 – Illustre un pré-traitement effectué sur un réseau réel. Sur le graphe virtuel, chaque petit carré représente une duplication du dépôt, les ronds représentent les stations origines ou destinations et les arcs représentent les déplacements possibles entre les dépôts et les stations. Sur chaque arc on retrouve le temps qui vaut aussi la dépense énergétique du déplacement.

- $X_{req} = \{1, 2, 3, 4\}$ avec 1 et 2 qui représentent la station réel A, 3 est la station réel B et 4 est la station réel C
- $X_{depot_debut} = \{5, 6, 7\}$
- $X_{depot_fin} = \{8, 9, 10\}$
- $X_{depot_elect} = \{11, 12\}$
- $X_{depot_h2} = \{13, 14, 15\}$
- $X_{depot_prod} = \{16\}$

Pour modéliser notre système, nous travaillerons dans un réseau virtuel c'est-à-dire que nous ferons un pré-traitement sur le réseau de transport existant dans la réalité. Ce pré-traitement consistera donc à faire abstraction de certaines routes qui ne mènent pas directement aux stations « origine » et aux stations « destination ». On transformera le réseau réel en faisant abstraction de certains nœuds qui ne sont ni origine, ni destination et en calculant le plus court chemin d'une origine à une destination (voir figure 2.7).

A chaque station virtuelle $x \in X$ correspond une station réelle et à une station réelle peut correspondre à plusieurs stations virtuelles.

Soit m la fonction surjective qui à une station virtuelle associe la station réelle correspondante :

$$m : \left\{ \begin{array}{lcl} X & \longrightarrow & STATIONS \\ x & \longmapsto & m(x) \end{array} \right.$$

Pour chacun de ces stations de l'ensemble X des stations virtuelles, on va joindre un certain nombre d'informations. On prolongera les fonctions temps et énergie, définies au niveau des inputs réels en deux tableaux tmp et $energie$ définis sur $X \times X$ pour :

- $tmp[i, j] \sim$ distance au sens du plus court chemin de i vers j qui est induite par la fonction tmp de l'input réel
- $energie[i, j] \sim$ quantité d'énergie consommée du sens du plus court chemin de i vers j qui est induite par la fonction $energie$ de l'input réel

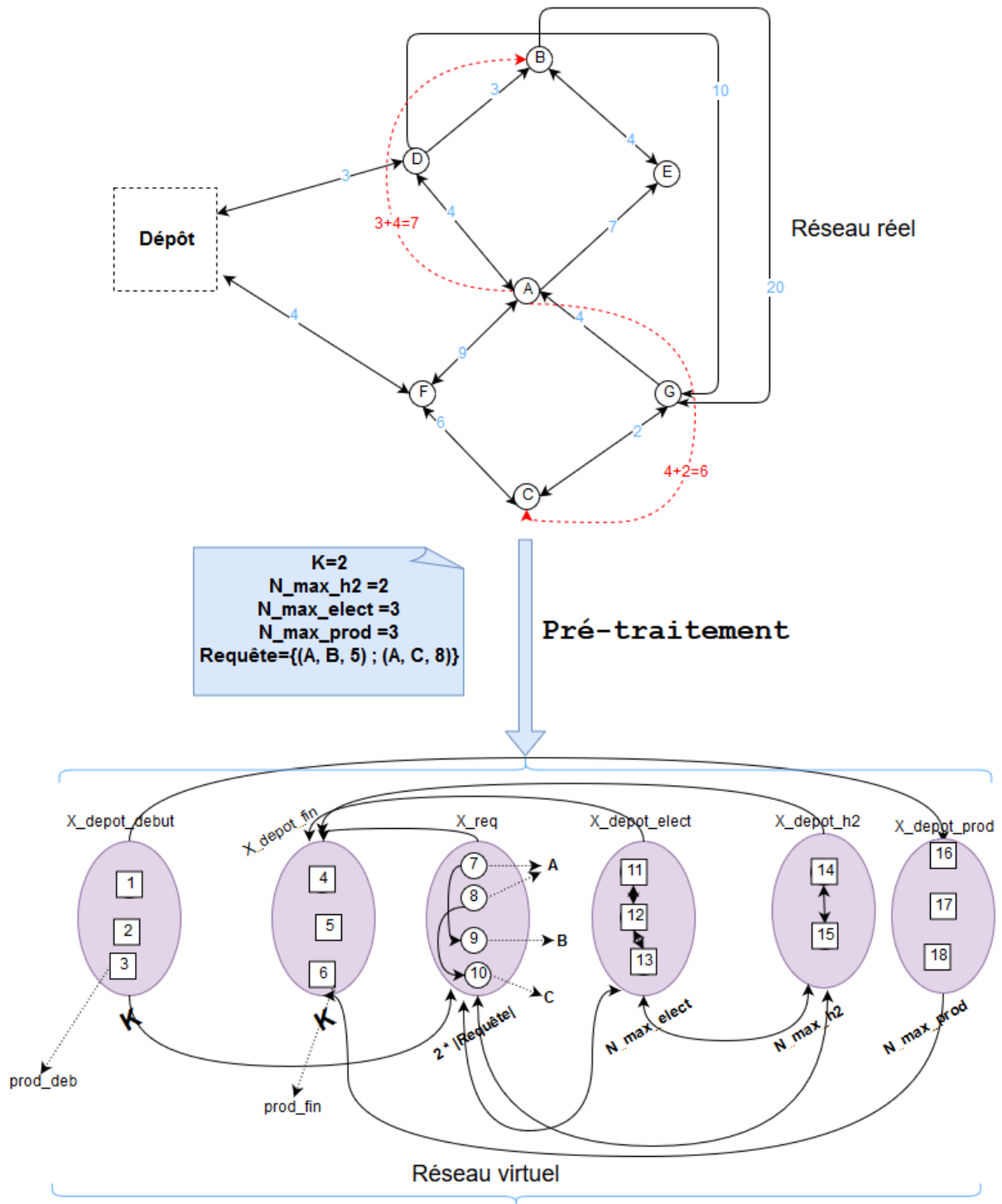


FIGURE 2.7 – Illustre une situation où on doit synchroniser des activités de logistique/-transport avec des production et consommation d'énergie renouvelable. On transforme le réseau réel en faisant abstraction de certains nœuds qui ne sont ni origine, ni destination et en calculant le plus court chemin d'une origine à une destination. Les arcs représentent les déplacements possibles dans le graphe.

On crée un tableau $labels_in$ indexé en colonne sur les stations virtuelles (c'est-à-dire de 1 à $N_Req + 2K + K \times (N_max_elect + N_max_h2) + N_max_prod$) et indexé en ligne sur les informations suivantes :

Pour tout $i \in X$,

— $labels_in[Id_req, i]$ dit si la station i est une station de X_req

$$labels_in[Id_req][i] = \begin{cases} 1 & \text{si } i \in X_req \\ 0 & \text{sinon.} \end{cases}$$

— $labels_in[Id_arrive, i]$ dit si la station i est une station de X_depot_debut

$$labels_in[Id_arrive][i] = \begin{cases} 1 & \text{si } i \in X_depot_debut \\ 0 & \text{sinon.} \end{cases}$$

— $labels_in[Id_depart][i]$ dit si la station i est une station de X_depot_fin

$$labels_in[Id_depart, i] = \begin{cases} 1 & \text{si } i \in X_depot_fin \\ 0 & \text{sinon.} \end{cases}$$

— $labels_in[lieu_geographique, i] \sim$ le point réel ($m(i)$) correspondant à la station virtuelle i , cette valeur va valoir -1 pour toutes les stations X_depot_prod

— $labels_in[status, i] \sim$ le status de la station i c'est-à-dire

$$labels_in[status, i] = \begin{cases} 1 & \text{si la station } i \text{ est une origine} \\ -1 & \text{si la station } i \text{ est une destination} \\ 0 & \text{sinon.} \end{cases}$$

— $labels_in[load, i] \sim$ la charge de $m(i)$ si $labels_in[status][i] = 1$ ou $labels_in[status][i] = -1$

— $labels_in[depot_elect, i] \sim$ qui nous renseigne si la station virtuelle i correspond au dépôt pour la recharge en électricité

$$labels_in[depot_elect, i] = \begin{cases} 1 & \text{si } i \text{ est un dépôt électrique} \\ 0 & \text{sinon.} \end{cases}$$

- $labels_in[depot_h2, i] \sim$ qui nous renseigne si la station virtuelle i correspond au dépôt pour la recharge en hydrogène

$$labels_in[depot_h2, i] = \begin{cases} 1 & \text{si } i \text{ est un dépôt hydrogène} \\ 0 & \text{sinon.} \end{cases}$$

- $labels_in[depot_prod, i] \sim$ qui nous renseigne si la station virtuelle i correspond au dépôt pour la production d'hydrogène

$$labels_in[depot_prod, i] = \begin{cases} 1 & \text{si } i \text{ est un dépôt production d'hydrogène} \\ 0 & \text{sinon.} \end{cases}$$

pour tout $i \in X_req$,

- $labels_in[requete, i]$ est le numéro de la requête traitée à la station i

Output du système « véhicules »

Ce qu'on cherche à calculer est un vecteur de tournées $tour = (tour[1], tour[2], \dots, tour[K])$.

Une tournée $tour[k]$, avec $k \in \{1, \dots, K\}$ est une liste de $X - X_depot_prod$ dans laquelle chaque station $i \in X - X_depot_prod$ apparaît une fois.

De plus, le tableau $labels_out$ fournit, pour chaque $i \in X - X_depot_prod$, qui apparaît dans un des tours $tour[k]$, avec $k \in \{1, \dots, K\}$, un certain nombre d'informations :

pour tout $i \in X - X_depot_prod$,

- $labels_out[i].actif$ qui indique si une station a été traversée par un véhicule

$$labels_out[i].actif = \begin{cases} 1 & \text{si } i \text{ a été traversée par un véhicule} \\ 0 & \text{sinon.} \end{cases}$$

- $labels_out[i].veh$ est le véhicule qui dessert la station i

- $labels_out[i].time_depart$ est la date d'arrivée du véhicule $labels_out[i].veh$ en i (début de l'action fait en i)
- $labels_out[i].time_arrive$ est la date de départ du véhicule $labels_out[i].veh$ depuis i (fin de l'action fait en i)
- $labels_out[i].load$ est la charge (bagage) dans le véhicule $labels_out[i].veh$ à son arrivé en i (ici le véhicule peut contenir la charge de plusieurs stations)
- $labels_out[i].CH_elect$ est la charge électrique du véhicule $labels_out[i].veh$ à l'arrivée en i
- $labels_out[i].CH_h2$ est la charge d'hydrogène du véhicule $labels_out[i].veh$ à l'arrivée en i
- $labels_out[i].energ$ est la quantité d'énergie chargée du véhicule $labels_out[i].veh$ en i
- $labels_out[i].mode_arr_elect$ est le pourcentage d'électricité que le véhicule $labels_out[i].veh$ va utiliser quand il va aller de i vers son successeur dans la tournée.

Exemple 10 En utilisant le réseau virtuelle de la figure 2.6, dont les stations virtuelles sont $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$, un exemple de représentation des tournées à l'aide de ce formalisme est : $tour[1] = [5, 1, 11, 3, 8]$ et $tour[2] = [6, 2, 13, 12, 4, 9]$. On suppose que $tmp_elect = 1$ et que le véhicule ne consommera jamais de l'électricité. De plus, $init_elect = 10$, $init_h2 = 11$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>actif</i>	1	1	1	1	1	1	0	1	1	0	1	1	2	0	0
<i>veh</i>	1	2	1	2	1	2	-	1	2	-	1	2	2	-	-
<i>time_depart</i>	3		$6+1 \times 5$		0		-	14		-	$3+3$			-	-
<i>time_fin</i>	3		$6+1 \times 5$		0		-	14		-	$6+1 \times 5$			-	-
<i>load</i>	20		20		0		-	0		-	20			-	-
<i>CH_elect</i>					10		-			-				-	-
<i>CH_h2</i>						11	-			-				-	-
<i>energ</i>	-	-	-	-	-	-	-	-	-	-	5	+	25	+	+
<i>mode_arr_elect</i>	0	0	0	0	0	0	-	0	0	-	0	0	0	-	-

TABLE 2.2 – représentation du tableau des labels des stations virtuelles de la figure 2.6, avec $tour = (tour[1], tour[2])$.

Le véhicule 1 charge 5 kWh d'électricité à la station virtuelle 11, c'est-à-dire au dépôt

considéré comme lieu de recharge électrique. Le véhicule 2 charge 25 kWh d'hydrogène à la station 13, c'est-à-dire au dépôt considéré comme lieu de recharge hydrogène.

Le signe « - » signifie que ces valeurs n'existeront jamais pour chaque station. Le signe « + » signifie que ces valeurs auraient pu exister. Et les espaces blancs signifient que ces valeurs existent mais on les a omises par souci de lisibilité.

L'avantage de cette formalisation est qu'elle permet d'avoir rapidement accès aux informations d'une tournée. Le tableau 2.8 présente le récapitulatif des inputs et des outputs du système « véhicules ».

X "Labels in" qui sont les inputs du système véhicule						X-X_prod "Labels out" qui sont les outputs du système véhicule				
Id_req				veh			...	
Id_arrive				time_arrive			...	
Id_depart				time_depart			...	
lieu_geographique				load			...	
status				CH_elect			...	
load				CH_h2			...	
depot_elect				energ			...	
depot_h2				actif			...	
depot_prod				mode_arr_elect			...	
requete								
tmp[i][j]	K			init_h2(k)		Liste des stations virtuelles X-X_prod pour chaque véhicule k suivant l'ordre de visite de chaque station				
energie[i][j]	CAP_load			init_elect(k)						
REQ	CAP_elect_veh			tmp_h2						
TMAX	CAP_h2_veh			tmp_elect						

FIGURE 2.8 – Récapitulatif des inputs et outputs du système « véhicules ».

Contraintes du système « véhicules »

Pour chaque requête $r \in 1 \dots R$, r désigne l'origine de r et $R+r$ désigne la destinations de r . Pour chaque véhicule $k \in 1 \dots K$, $2R+k$ désigne l'arrivé de k en i et $2R+K+k$ désigne le départ de k depuis i .

Les contraintes sur le problème de tournées du système « véhicules » sont :

1. contraintes globales

- (a) chaque station de X_{req} doit apparaître exactement une fois dans l'ensemble des tours
- (b) Chaque station de $X_{depot_elect} \cup X_{depot_h2} \cup X_{depot_prod}$ doit apparaître au plus une fois dans l'ensemble des tours
- (c) Pour tout $i \in X_{depot_debut}, X_{depot_fin}, labels_out[i].load = 0$
- (d) toutes les requêtes doivent être traitées

2. contraintes structurelles

- (a) le premier élément de chaque tournée $tour[k]$ s'écrit :
 $(0, 0, 0, 0, init_elect, init_h2, 0)$
- (b) le dernier élément de chaque tournée $tour[k]$ s'écrit :
 $(1, time_arrive, time_depart, 0, fin_elect(\geq init_elect(k)),$
 $fin_h2(\geq init_h2(k)), 0)$: les véhicules reviennent au dépôt comme ils sont parties c'est-à-dire que la quantité d'électricité dans le véhicule doit être supérieur ou égale à la quantité initiale.
- (c) si $i = REQ[r].origine$ figure dans la tournée, alors $j = REQ[r].destination$ est aussi dans la tournée et après i ; et si $j = REQ[r].destination$ figure dans la tournée, alors $i = REQ[r].origine$ est aussi dans la tournée et avant j

3. contraintes locales

Chaque tournée $tour[k]$ doit être réalisable. Une tournée est réalisable lorsque :

- (a) Contraintes de borne des variables. Pour chaque élément i de $X - X_{depot_prod} \cup \{2R + K + 1, 2R + 2K + 2\}$

i. contraintes temporelles

- A. Pour tout $i \in X - X_{depot_prod} \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $labels_out[i].time_depart \leq TMAX$: les véhicules doivent arrivés aux stations avant la fin de l'horizon $TMAX$,
- B. Pour tout $i \in X - X_{depot_prod} \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $0 \leq labels_out[i].time_depart$: les véhicules débutent après l'instant 0,

- C. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $labels_out[i].time_arrive \leq TMAX$: les véhicules doivent arrivés
aux stations avant la fin de l'horizon $TMAX$,
- D. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $0 \leq labels_out[i].time_arrive$: les véhicules finissent après l'instant
0
- E. Pour tout $i \neq 2R + k$ et $2R + K + k$,
 $labels_out[i].time_arrive \leq labels_out[i].time_depart$
- F. Pour tout $i \in X_req, X_depot_debut, X_depot_fin$,
 $labels_out[i].time_arrive = labels_out[i].time_depart$
- ii. contraintes sur la capacité (charge portée) du véhicule
 - A. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $0 \leq labels_out[i].load$: la charge d'un véhicule n'excède jamais sa
capacité de charge minimale
 - B. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $labels_out[i].load \leq CAP_load$: la charge d'un véhicule n'excède ja-
mais sa capacité de charge maximale
- iii. contraintes sur l'électricité
 - A. Pour tout $i \in X_depot_elect$, $0 \leq labels_out[i].CH_elect$: la quan-
tité d'électricité au dépôt est toujours supérieure à zéro
 - B. Pour tout $i \in X_depot_elect$,
 $labels_out[i].CH_elect + labels_out[i].energ \times labels_in[depot_elect][i]$
 $\leq CAP_elect_veh$: la charge d'un véhicule n'excède jamais sa capa-
cité électrique maximale
 - C. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $0 \leq labels_out[i].CH_elect \leq CAP_elect_veh$
- iv. contraintes sur l'hydrogène
 - A. Pour tout $i \in X_depot_h2$,

- B. Pour tout $i \in X_depot_h2$, $0 \leq labels_out[i].CH_h2$
 $labels_out[i].CH_h2 + labels_out[i].energ \times labels_in[depot_h2][i]$
 $\leq CAP_h2_veh$: la charge d'un véhicule n'excède jamais sa capacité
hydrogène maximale
- C. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $0 \leq labels_out[i].CH_h2 \leq CAP_h2_veh$,

v. Autres contraintes

- A. si $i \neq X_depot_elect$,
 X_depot_h2 alors $labels_out[i].energ = 0$: aucune recharge ne s'ef-
fectue dans une station autre que le dépôt
- B. $tour[k]$ débute par $2R + k$ et finit par $2R + K + k$
- C. $labels_out[i].veh = k$ et $labels_out[i].actif = 1$ si et seulement si
 $i \in tour[k]$
- D. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $0 \leq labels_out[i].mode_arr_elect$
- E. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $labels_out[i].mode_arr_elect \leq 1$

(b) pour deux éléments consécutifs $i1$ et $i2$ de $tour[k]$ (voir figure 2.9),

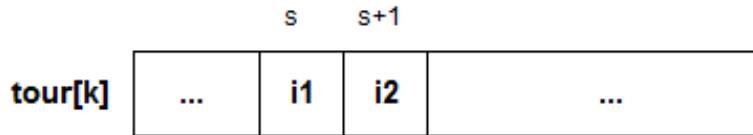


FIGURE 2.9 – Illustration d'une liste d'une tournée dont le s ième élément est $i1$ et le $s + 1$ ième élément est $i2$.

- i. $labels_out[i2].time_arrive \geq labels_out[i1].time_depart + tmp[i1, i2] +$
 $labels_in[depot_elect][i1] \times tmp_elect \times labels_out[i1].energ + labels_in[depot_h2][i1] \times$
 tmp_h2 Cette contrainte peut être divisée en trois contraintes :
- si $i1 \neq X_depot_elect, X_depot_h2$ alors $labels_out[i2].time_arrive \geq$
 $labels_out[i1].time_depart + tmp[i1, i2]$: pour se déplacer d'une sta-
tion $i1$ à une station $i2$ un véhicule fait un temps supérieur ou égal à
 $tmp[i1, i2]$

- si $i1 \in X_depot_h2$ (on charge le véhicule en hydrogène en $i1$) alors
 $labels_out[i2].time_arrive \geq labels_out[i1].time_depart + tmp[i1, i2] + tmp_h2$: pour faire une recharge en hydrogène un véhicule mettra le temps $tmp[i1, i2]$ pour se rendre au dépôt et le temps tmp_h2 que dure la recharge
- si $i1 \in X_depot_elect$ (on charge le véhicule en électricité en $i1$) alors
 $labels_out[i2].time_arrive \geq labels_out[i1].time_depart + tmp[i1, i2] + tmp_elect \times energ$: pour faire une recharge en électricité un véhicule mettra le temps $tmp[i1, i2]$ pour se rendre au dépôt et le temps $tmp_elect \times energ$ que dure la recharge
- ii. $labels_out[i2].load = labels_out[i1].load + (labels_in[status][i2] \times labels_in[load][i2])$: la charge d'un véhicule diminue si on est sur une station destination et cette charge augmente si on est sur une station origine
- iii. Les véhicules consomment de l'électricité ou de l'hydrogène pour se déplacer d'une station à une autre
 si $i1 \neq X_depot_elect, X_depot_h2$ alors
 - $labels_out[i2].CH_elect \leq labels_out[i1].CH_elect - [labels_out[i1].mode_arr_elect \times energie[i1, i2]]$
 - $labels_out[i2].CH_h2 \leq labels_out[i1].CH_h2 - [(1 - labels_out[i1].mode_arr_elect) \times energie[i1, i2]]$
 - $labels_out[i2].CH_elect + labels_out[i2].CH_h2 = labels_out[i1].CH_elect + labels_out[i1].CH_h2 - [(1 - labels_out[i1].mode_arr_elect) \times energie[i1, i2]] + labels_out[i1].mode_arr_elect \times energie[i1, i2]]$
- iv. la quantité d'hydrogène dans un véhicule augmente après sa recharge en hydrogène au dépôt ($i1 \in X_depot_h2$), les véhicules consomment de l'électricité ou de l'hydrogène pour aller au dépôt se charger en hydrogène
 - $labels_out[i2].CH_elect \leq labels_out[i1].CH_elect - [labels_out[i1].mode_arr_elect \times energie[i1, i2]]$

$$\begin{aligned}
& \text{— } labels_out[i2].CH_h2 \leq labels_out[i1].CH_h2 + \\
& \quad labels_in[depot_h2][i1] \times energ - \\
& \quad [(1 - labels_out[i1].mode_arr_elect) \times energie[i1, i2]] \leq CAP_h2_veh \\
& \text{— } labels_out[i2].CH_elect + labels_out[i2].CH_h2 \leq \\
& \quad labels_out[i1].CH_elect + labels_out[i1].CH_h2 + \\
& \quad labels_in[depot_h2][i1] \times labels_out[i1].energ \\
& \quad - [(1 - labels_out[i1].mode_arr_elect) \times energie[i1, i2]] \\
& \quad + labels_out[i1].mode_arr_elect \times energie[i1, i2]]
\end{aligned}$$

v. la quantité d'électricité dans un véhicule augmente après sa recharge en électricité au dépôt $i1 \in X_depot_elect$, les véhicules consomment de l'électricité ou de l'hydrogène pour aller au dépôt se charger en électricité

$$\begin{aligned}
& \text{— } labels_out[i2].CH_elect \leq labels_out[i1].CH_elect + labels_in[depot_elect][i1] \times \\
& \quad energ - [labels_out[i1].mode_arr_elect \\
& \quad \times energie[i1, i2]] \leq CAP_elect_veh \\
& \text{— } labels_out[i2].CH_h2 \leq labels_out[i1].CH_h2 - \\
& \quad [(1 - labels_out[i1].mode_arr_elect) \times energie[i1, i2]] \\
& \text{— } labels_out[i2].CH_elect + labels_out[i2].CH_h2 \leq \\
& \quad labels_out[i1].CH_elect + labels_out[i1].CH_h2 + \\
& \quad labels_in[depot_elect][i1] \times labels_out[i1].energ \\
& \quad - [(1 - labels_out[i1].mode_arr_elect) \times energie[i1, i2]] \\
& \quad + labels_out[i1].mode_arr_elect \times energie[i1, i2]]
\end{aligned}$$

2.3.2 système « production d'énergie »

Input du système « production d'énergie »

Les entrées du système « production d'énergie » sont :

1. concernant l'électricité

- p_max ~ la puissance électrique disponible (maximum)
- P ~ la puissance électrique requise pour charger chaque véhicule

- $Q \sim$ la puissance électrique requise pour fabriquer de l'hydrogène
- $\text{cout_elect}(t) \sim$ une fonction qui donne le prix d'une unité de puissance électrique entre $[t, t + 1]$ (9 euro par kWh par exemple), cette fonction est une fonction en escalier (voir figure 2.10)

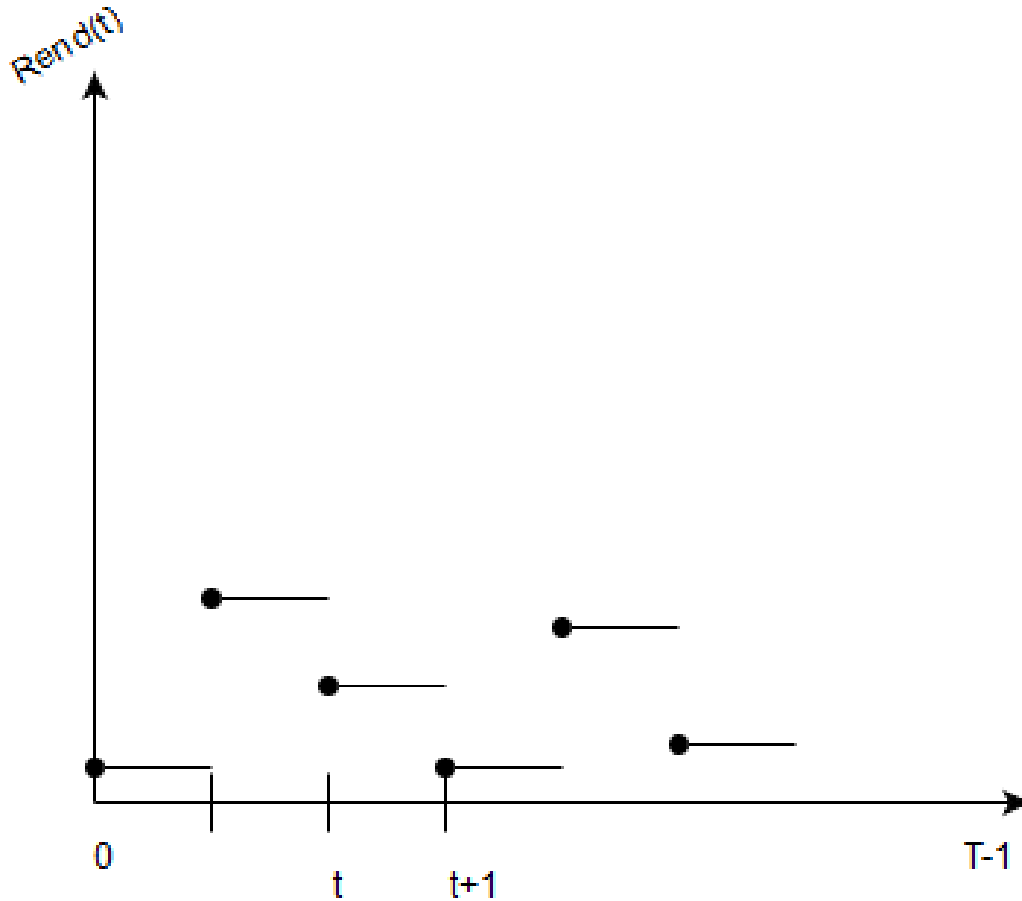


FIGURE 2.10 – Fonction en escalier du rendement en hydrogène.

2. concernant l'hydrogène

- $CAP_citerne \sim$ la capacité maximum de la citerne
- la fonction $t \longrightarrow \text{rend}(t)$ qui associe à tout instant t la quantité d'énergie produite par la centrale par unité de temps si elle est active
- $\text{init_cit} \sim$ le chargement initiale de la citerne
- $\text{cout_setup} \sim$ le coût de démarrage de la machine qui produit l'hydrogène (en euro par exemple)

- $\text{cout_h2} \sim$ le coût de fabrication de l'hydrogène par unité de temps (par exemple 15 euro/unité de temps)

Output du système « production d'énergie »

L'objet qu'on cherche à calculer est la liste liste_production , qui est une liste de stations virtuelles dans $X_depot_prod \cup X_depot_h2 \cup \{2R + K + 1, 2R + 2K + 2\}$: chaque élément de cette liste est donc un indice.

La sémantique de cette liste, est que les opérations de production et consommation d'hydrogène, vont se dérouler sur ces stations, dans l'ordre selon lequel les stations sont classées.

Exemple 11 *En reprenant l'illustration de la figure 2.7,*

- 3 désigne le dépôt vu comme début du processus de production
- 6 désigne le dépôt vu comme fin du processus de production
- $X_depot_h2 = \{14, 15\}$, désigne le dépôt vu comme lieux de recharge en hydrogène pour les véhicules
- $X_depot_prod = \{16, 17, 18\}$ désigne le dépôt vu comme lieux de production d'hydrogène

alors la liste $\text{liste_production} = \{3, 14, 16, 15, 17, 6\}$ signifie :

1. début du processus (station virtuelle 3)
2. un véhicule charge de l'hydrogène au dépôt (station virtuelle 14)
3. puis, la micro-usine de production produit une première fois (station virtuelle 16)
4. puis, un véhicule vient charger en hydrogène (station virtuelle 15)
5. puis, la micro-usine de production produit une deuxième fois (station virtuelle 17)
6. et le processus se termine (station virtuelle 6)

Cette liste liste_production est labellisée, c'est-à-dire que tout $i \in X_depot_prod \cup X_depot_h2 \cup \{2R + K + 1, 2R + 2K + 2\}$ est accompagné dans le tableau labels_out d'un certain nombre d'informations :

1. si $i \in X_depot_h2$, les labels de i ont déjà été mentionnés à la section 2.3.1

2. si $i \in X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$, on trouve :

- $labels_out[i].cit \sim$ le contenu de la citerne d'hydrogène à « l'arrivée » en j , c'est-à-dire le processus de production ou consommation associé à i débute.
- $labels_out[i].time_arrive \sim$ le début de la phase i de production
- $labels_out[i].time_depart \sim$ la fin de la phase i de production
- $labels_out[i].energ \sim$ la quantité d'énergie produite durant la phase i de production
- $labels_out[i].actif \sim$ qui indique si une station a déjà été traversée.

Le tableau 2.11 présente le récapitulatif des inputs et des outputs du système « production d'énergie ».

X "Labels in" qui sont les inputs du système prod						X_prod	"Labels out" qui sont les outputs du système prod			
Id_req					cit		...	
Id_arrive					time_arrive		...	
Id_depart					time_depart		...	
lieu_geographique					energ		...	
status					actif		...	
load				
depot_elect				
depot_h2				
depot_prod				
requete				
tmp[i][j]		K			init_h2(k)		Liste des stations virtuelles X_depot_prod U X_depot_elect suivant l'ordre de visite de chaque station			
energie[i][j]		CAP_load			init_elect(k)					
REQ		CAP_elect_veh			tmp_h2					
TMAX		CAP_h2_veh			tmp_elect					
P	Q	PMAX	init_cit	CAP_citerne		Rend(t)				

FIGURE 2.11 – Récapitulatif des inputs et outputs du système « production d'énergie ».

Contraintes du système « production d'énergie »

Les contraintes sont :

1. le premier élément de la liste *liste_production* est $2R + K + 1$ et le dernier élément de la liste *liste_production* est $2R + 2K + 2$, le dépôt est vu comme lieu de départ et de fin d'un véhicule virtuel « citerne »
2. $labels_out[2R + K + 1].cit = init_cit$ et $labels_out[2R + 2K + 2].cit \geq init_cit$
3. Pour tout $i \in liste_production$, $0 \leq labels_out[i].cit \leq CAP_citerne$: la quantité d'hydrogène dans la citerne ne dépasse jamais la capacité maximale de la citerne
4. pour tout i, j qui se suivent dans *liste_production*,

$$labels_out[j].cit = labels_out[i].cit + labels_in[depot_prod, i] \times energ[i] - labels_in[depot_h2, i] \times energ[i]$$

la quantité d'hydrogène dans la citerne augmente après une production et diminue après une recharge
5. $labels_out[N_Req + K + 1].time_arrive = 0$: la production commence au temps 0
6. $labels_out[N_Req + 2K + 2].time_depart = TMAX$: la production finie au temps *TMAX*
7. pour i, j qui se succède dans *liste_production*, on a $labels_out[i].time_depart \leq labels_out[j].time_arrive$: on ne peut pas avoir recharge d'hydrogène et production simultanément
8. si $i \in X_depot_h2$ alors $labels_out[i].time_depart = labels_out[i].time_arrive + tmp_h2$: la recharge d'un véhicule dure *tmp_h2*
9. si $i \in X_depot_prod$ alors $energ[i] = \int_{t=labels_out[i].time_arrive}^{labels_out[i].time_depart} rend(t) dt$: la quantité d'énergie produite par la citerne dépend du rendement d'hydrogène
10. contraintes limitant le nombre de recharge électrique simultanée et les productions d'hydrogène car on ne peut consommer plus de *p_max* électricité à un instant *t*.
 Pour écrire cette contrainte, à tout instant *t* on pose :
 - (a) $N_CH_elect[t] \sim$ le nombre de véhicule *k* entrain de se charger en électricité entre *t* et *t + 1*, c'est-à-dire le nombre d'indice $i \in X_depot_elect$ tel que $actif[i] = 1$ et
 $labels_out[i].time_arrive \leq t, t+1 \leq labels_out[i].time_depart$ et $i \in X_depot_prod$

- (b) $actif_prod[t] \sim$ un indicateur de production qui indique si la machine de production d'hydrogène est entrain de produire entre t et $t + 1$,
avec $labels_out[i].time_arrive \leq t, t + 1 \leq labels_out[i].time_depart$ et $i \in X_depot_prod$

$$actif_prod[t] = \begin{cases} 1 & \text{si la machine de production produit de l'hydrogène entre } t \text{ et } t + 1 \\ 0 & \text{sinon.} \end{cases}$$

la contrainte est : $\forall t, P \times [\sum_{k=1}^K N_CH_elect[t]] + Q \times actif_prod[t] \leq p_max$

2.4 Évaluation des coûts énergétiques

Soit, j le nombre de stations virtuelles dites de production,

les coûts sont :

— Hydrogène : $[cout_setup \times \sum_{j \in X_depot_prod} actif[j]] + [cout_h2 \times \sum_{j \in X_depot_prod} actif[j] \times time_depart[j] - time_arrive[j]]$

— Électricité :

$$P \times [\sum_{j \in X_depot_elect} actif[j] \times \int_{t=time_arrive[j]}^{time_depart[j]} cout_elect(t) dt] \\ + Q \times [\sum_{j \in X_depot_prod} actif[j] \times \int_{t=time_arrive[j]}^{time_depart[j]} cout_elect(t) dt]$$

2.5 Programme mathématique du système

2.5.1 Programme mathématique du « système véhicules »

Output du système « véhicules »

Définition 11 Vecteur principal Z^k

On va poser K vecteurs : Z^1, Z^2, \dots, Z^K , et chaque vecteur Z^k , $k \in \{1, 2, \dots, K\}$ sera indexé sur les couples (i, j) avec $i \neq j$ et $i, j \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$. La sémantique de $Z_{i,j}^k$ est :

$$Z_{i,j}^k = \begin{cases} 1 & \text{si le véhicule } k \text{ (correspondant au tour}[k]) \text{ passe en } i \text{ puis en } j \\ 0 & \text{sinon.} \end{cases}$$

L'ensemble des tournées sera représenté par :

1. K vecteurs principaux

2. un tableau *labels_out* indexé en ligne par les labels des stations

time_arrive, *time_depart*, *load*, *CH_elect*, *CH_h2*, *energ*, *veh* et indexé en colonne par $X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$

— *labels_out[i].actif* qui indique si une station a été traversée par un véhicule

$$labels_out[i].actif = \begin{cases} 1 & \text{si } i \text{ a été traversée par un véhicule} \\ 0 & \text{sinon.} \end{cases}$$

— *labels_out[i].veh* est le véhicule qui dessert la station i

— *labels_out[i].time_depart* est la date d'arrivée du véhicule *labels_out[i].veh* en i (début de l'action fait en i)

— *labels_out[i].time_arrive* est la date de départ du véhicule *labels_out[i].veh* depuis i (fin de l'action fait en i)

— *labels_out[i].load* est la charge (bagage) dans le véhicule *labels_out[i].veh* à son arrivé en i (ici le véhicule peut contenir la charge de plusieurs stations)

— *labels_out[i].CH_elect* est la charge électrique du véhicule *labels_out[i].veh* à l'arrivée en i

— *labels_out[i].CH_h2* est la charge d'hydrogène du véhicule *labels_out[i].veh* à l'arrivée en i

— *labels_out[i].energ* est la quantité d'énergie chargée du véhicule *labels_out[i].veh* en i

— *labels_out[i].mode_arr_elect* est le pourcentage d'électricité que le véhicule *labels_out[i].veh* va utiliser quand il va aller de i vers son successeur dans la tournée.

Contraintes du système « véhicules »

En ce qui concerne les contraintes on a trois types :

1. les contraintes sur les variables du système sauf Z

$$labels_out[i].time_depart \leq TMAX \quad (i)$$

$$labels_out[i].time_depart \geq 0 \quad (ii)$$

$$labels_out[i].time_arrive \leq TMAX \quad (iii)$$

$$labels_out[i].time_arrive \geq 0 \quad (iv)$$

$$labels_out[i].time_arrive \leq labels_out[i].time_depart \quad (v)$$

$$labels_out[i].time_arrive = labels_out[i].time_depart \quad (vi)$$

(a) contraintes temporelles

- i. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,

$labels_out[i].time_depart \leq TMAX$: les véhicules doivent arrivés aux stations avant la fin de l'horizon $TMAX$,

- ii. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$, $0 \leq$

$labels_out[i].time_depart$: les véhicules débutent après l'instant 0,

- iii. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,

$labels_out[i].time_arrive \leq TMAX$: les véhicules doivent arrivés aux stations avant la fin de l'horizon $TMAX$,

- iv. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$, $0 \leq$

$labels_out[i].time_arrive$: les véhicules finissent après l'instant 0

- v. Pour tout $i \neq 2R + k$ et $2R + K + k$,

$$labels_out[i].time_arrive \leq labels_out[i].time_depart$$

- vi. Pour tout $i \in X_req, X_depot_debut, X_depot_fin$,

$$labels_out[i].time_arrive = labels_out[i].time_depart$$

(b) contraintes sur la capacité (charge portée) du véhicule

$$labels_out[i].load \geq 0 \quad (vii)$$

$$labels_out[i].load \leq CAP_load \quad (viii)$$

- i. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$, $0 \leq labels_out[i].load$: la charge d'un véhicule n'excède jamais sa capacité de charge minimale
- ii. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$, $labels_out[i].load \leq CAP_load$: la charge d'un véhicule n'excède jamais sa capacité de charge maximale

(c) contraintes sur l'électricité

- i. Pour tout $i \in X_depot_elect$, $0 \leq labels_out[i].CH_elect$: la quantité d'électricité au dépôt est toujours supérieure à zéro
- ii. Pour tout $i \in X_depot_elect$, $labels_out[i].CH_elect + labels_out[i].energ \times labels_in[depot_elect][i] \leq CAP_elect_veh$: la charge d'un véhicule n'excède jamais sa capacité électrique maximale
- iii. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$, $0 \leq labels_out[i].CH_elect \leq CAP_elect_veh$

(d) contraintes sur l'hydrogène

- i. Pour tout $i \in X_depot_h2$,
- ii. Pour tout $i \in X_depot_h2$, $0 \leq labels_out[i].CH_h2$
 $labels_out[i].CH_h2 + labels_out[i].energ \times labels_in[depot_h2][i] \leq CAP_h2_veh$: la charge d'un véhicule n'excède jamais sa capacité hydrogène maximale
- iii. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$, $0 \leq labels_out[i].CH_h2 \leq CAP_h2_veh$,

(e) Autres contraintes

- i. si $i \neq X_depot_elect$,
 X_depot_h2 alors $labels_out[i].energ = 0$: aucune recharge ne s'effectue dans une station autre que le dépôt
 - ii. $tour[k]$ débute par $2R + k$ et finit par $2R + K + k$
 - iii. $labels_out[i].veh = k$ et $labels_out[i].actif = 1$ si et seulement si $i \in tour[k]$
 - iv. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$, $0 \leq labels_out[i].mode_arr_elect$
 - v. Pour tout $i \in X - X_depot_prod \cup \{2R + K + 1, 2R + 2K + 2\}$,
 $labels_out[i].mode_arr_elect \leq 1$
2. les contraintes liant Z aux autres variables du système $\forall k \in \{1, \dots, K\}, \forall \text{ arc segment } \vec{i, j}, i \neq j$,
- a) $\forall i, j \in X_req \cup X_arrive, (Z_{i,j}^k = 1) \implies (time_arrive_j \geq time_depart_i + tmp[i, j])$: pour se déplacer d'une station i à une station j un véhicule fait un temps supérieur ou égal à $tmp[i, j]$
 - b) $\forall i \in X_depot_elect, (Z_{i,j}^k = 1) \implies (time_arrive_j \geq time_depart_i + tmp[i, j] + tmp_elect \times energ_i)$: pour faire une recharge en électricité un véhicule mettra le temps $tmp[i, j]$ pour se rendre au dépôt et le temps $tmp_elect \times energ_i$ que dure la recharge
 - c) $\forall i \in X_depot_h2, (Z_{i,j}^k = 1) \implies (time_arrive_j \geq time_depart_i + tmp[i, j] + tmp_h2)$: pour faire une recharge en hydrogène un véhicule mettra le temps $tmp[i, j]$ pour se rendre au dépôt et le temps tmp_h2 que dure la recharge
 - d) $\forall i \in X_req \cup X_arrive, (Z_{i,j}^k = 1) \implies (load_j \geq load_i + labels_in[status, j] \times labels_in[load, j])$: la charge d'un véhicule diminue si on est sur une station destination et cette charge augmente si on est sur une station origine
 - e) Les véhicules consomment de l'électricité ou de l'hydrogène pour se déplacer d'une station à une autre (différentes des stations $X_depot_elect, X_depot_h2$)
 $\forall i \in X_req, X_depot_debut$
 - i) $(Z_{i,j}^k = 1) \implies (CH_electj \leq CH_electi)$

$$\text{ii) } (Z_{i,j}^k = 1) \implies (CH_h2_j \leq CH_h_i)$$

$$\text{iii) } (Z_{i,j}^k = 1) \implies (CH_elect_j + CH_h2_j = CH_elect_i + CH_h2_i - energie[i, j])$$

- f) la quantité d'hydrogène dans un véhicule augmente après sa recharge en hydrogène au dépôt, les véhicules consomment de l'électricité ou de l'hydrogène pour aller au dépôt se charger en hydrogène

$$\forall i \in X_depot_h2,$$

$$\text{i) } (Z_{i,j}^k = 1) \implies (CH_elect_j \leq CH_elect_i)$$

$$\text{ii) } (Z_{i,j}^k = 1) \implies (CH_h2_j \leq CH_h2_i + energi \leq CAP_h2_veh)$$

$$\text{iii) } (Z_{i,j}^k = 1) \implies (CH_elect_j + CH_h2_j \leq CH_elect_i + CH_h2_i + labels_in[depot_h2][i] \times energi - energie[i, j])$$

- g) la quantité d'électricité dans un véhicule augmente après sa recharge en électricité au dépôt, les véhicules consomment de l'électricité ou de l'hydrogène pour aller au dépôt se charger en électricité

$$\forall i \in X_depot_elect,$$

$$\text{i) } (Z_{i,j}^k = 1) \implies (CH_h2_j \leq CH_h2_i)$$

$$\text{ii) } (Z_{i,j}^k = 1) \implies (CH_elect_j \leq CH_elect_i + energi \leq CAP_elect_veh)$$

$$\text{iii) } (Z_{i,j}^k = 1) \implies (CH_elect_j + CH_h2_j \leq CH_elect_i + CH_h2_i + labels_in[depot_elect][i] \times energi - energie[i, j])$$

$$(time_arrive_i, 0, init_elect, init_h2, 0, i)$$

- h) le premier élément de chaque tournée $tour[k]$ s'écrit :

$$(0, 0, 0, 0, init_elect, init_h2, 0)$$

- i) le dernier élément de chaque tournée $tour[k]$ s'écrit :

$$(1, time_arrive, time_depart, 0, fin_elect(\geq init_elect(k)), fin_h2(\geq init_h2(k)), 0) :$$

les véhicules reviennent au dépôt comme ils sont parties c'est-à-dire que la quantité d'électricité dans le véhicule doit être supérieur ou égale à la quantité initiale.

3. *****

(a) $labels_out[j].time_arrive \geq labels_out[i].time_depart + tmp[i, j] + labels_in[depot_elect][i] \times tmp_elect \times labels_out[i].energ + labels_in[depot_h2][i] \times tmp_h2$

(b) $labels_out[j].load = labels_out[i].load + (labels_in[status][j] \times labels_in[load][j]) :$

la charge d'un véhicule diminue si on est sur une station destination et cette charge augmente si on est sur une station origine

(c) Les véhicules consomment de l'électricité ou de l'hydrogène pour se déplacer d'une station à une autre

si $i \neq X_depot_elect, X_depot_h2$ alors

— $labels_out[j].CH_elect \leq labels_out[i].CH_elect - [labels_out[i].mode_arr_elect \times energie[i, j]]$

— $labels_out[j].CH_h2 \leq labels_out[i].CH_h2 - [(1 - labels_out[i].mode_arr_elect) \times energie[i, j]]$

— $labels_out[j].CH_elect + labels_out[j].CH_h2 = labels_out[i].CH_elect + labels_out[i].CH_h2 - [(1 - labels_out[i].mode_arr_elect) \times energie[i, j]] + labels_out[i].mode_arr_elect \times energie[i, j]]$

(d) la quantité d'hydrogène dans un véhicule augmente après sa recharge en hydrogène au dépôt ($i \in X_depot_h2$), les véhicules consomment de l'électricité ou de l'hydrogène pour aller au dépôt se charger en hydrogène

— $labels_out[j].CH_elect \leq labels_out[i].CH_elect - [labels_out[i].mode_arr_elect \times energie[i, j]]$

— $labels_out[j].CH_h2 \leq labels_out[i].CH_h2 + labels_in[depot_h2][i] \times energ - [(1 - labels_out[i].mode_arr_elect) \times energie[i, j]] \leq CAP_h2_veh$

— $labels_out[j].CH_elect + labels_out[j].CH_h2 \leq labels_out[i].CH_elect + labels_out[i].CH_h2 + labels_in[depot_h2][i] \times labels_out[i].energ - [(1 - labels_out[i].mode_arr_elect) \times energie[i, j]] + labels_out[i].mode_arr_elect \times energie[i, j]]$

- (e) la quantité d'électricité dans un véhicule augmente après sa recharge en électricité au dépôt $i \in X_depot_elect$, les véhicules consomment de l'électricité ou de l'hydrogène pour aller au dépôt se charger en électricité

$$\begin{aligned}
& \text{--- } labels_out[j].CH_elect \leq labels_out[i].CH_elect + labels_in[depot_elect][i] \times \\
& \quad energ - [labels_out[i].mode_arr_elect \\
& \quad \times energie[i, j]] \leq CAP_elect_veh \\
& \text{--- } labels_out[j].CH_h2 \leq labels_out[i].CH_h2 - \\
& \quad [(1 - labels_out[i].mode_arr_elect) \times energie[i, j]] \\
& \text{--- } labels_out[j].CH_elect + labels_out[j].CH_h2 \leq \\
& \quad labels_out[i].CH_elect + labels_out[i].CH_h2 + \\
& \quad labels_in[depot_elect][i] \times labels_out[i].energ \\
& \quad - [(1 - labels_out[i].mode_arr_elect) \times energie[i, j]] \\
& \quad + labels_out[i].mode_arr_elect \times energie[i, j]]
\end{aligned}$$

4. les contraintes sur Z qui traduisent le fait que les vecteurs Z^k définissent des tournées
 "ad hoc"

$$\sum_j Z_{i,j}^k \leq 1 \quad (\text{ix})$$

$$\sum_j Z_{j,i}^k \leq 1 \quad (\text{x})$$

$$\sum_j Z_{N_Req+k,j}^k = 1 \quad (\text{xi})$$

$$\sum_j Z_{j,N_Req+K+k}^k = 1 \quad (\text{xii})$$

$$\sum_{j,k} Z_{i,j}^k \leq 1 \quad (\text{xiii})$$

$$\sum_{j,k} Z_{i,j}^k = 1 \quad (\text{xiv})$$

$$\sum_j Z_{i,j}^k = \sum_j Z_{j,i}^k \quad (\text{xv})$$

$$time_arrive_{REQ[r][destination]} \geq time_depart_{REQ[r][origine]} \quad (\text{xvi})$$

$$\sum_j Z_{origine[r],j}^k = \sum_j Z_{destination[r],j}^k \quad (\text{xvii})$$

$$(actif_i = 1) \iff \left(\sum_{j,k} Z_{i,j}^k = 1 \right) \quad (\text{xviii})$$

- a) $\forall i \in X - X_dep\hat{o}t_prod, \forall k \in \{1, \dots, K\}, \sum_j Z_{i,j}^k \leq 1, \sum_j Z_{j,i}^k \leq 1$: Chaque station $i \in X - X_dep\hat{o}t_prod$ apparait au plus une fois dans un tour
- b) $\forall j \in X - X_dep\hat{o}t_prod \forall k \in \{1, \dots, K\} \sum_j Z_{N_Req+k,j}^k = 1$ et $\sum_j Z_{j,N_Req+K+k}^k = 1$: chaque tournée k part de $N_Req + k$ et finit en $N_Req + K + k$
- c) $\sum_{j,k} Z_{i,j}^k \leq 1$: l'ensemble des véhicules passe au plus une fois par un point donné
- d) $\forall i \in X_req, \sum_{j,k} Z_{i,j}^k = 1$: dans l'ensemble des requêtes chaque station doit être visitée exactement une fois
- e) $\forall i \in X_req, X_dep\hat{o}t_elect, X_dep\hat{o}t_h2, \forall k \in \{1, \dots, K\}, \sum_j Z_{i,j}^k = \sum_j Z_{j,i}^k$: pour toute station visitée par k on doit entrer et sortir au plus une fois.
- f) $\forall k \in \{1, \dots, K\} \forall r \in \{1, \dots, X_req\},$

$time_arrive_{REQ[r][destination]} \geq time_depart_{REQ[r][origine]}$ et

$\sum_j Z_{origine[r],j}^k = \sum_j Z_{destination[r],j}^k$: si une tournée passe par une origine (

respectivement une destination) de requête alors elle passe par la destination (respectivement une destination)

$$g) \forall i \in X_depot_h2, X_depot_elect, (actif_i = 1) \iff (\sum_{j,k} Z_{i,j}^k = 1)$$

Remarque 1 Dans ce système nous n'aurons pas de contraintes de sous-tours parce que les sous-tours ne peuvent pas se produire car les dates d'arrivées sont incluses dans le modèle et ces dates sont croissantes.

2.5.2 Programme mathématique du système « production d'énergie »

Output du système « production d'énergie »

L'objet calculé est une tournée appelée *liste_production* qui prend ses éléments dans $X_depot_h2 \cup X_depot_prod \cup \{Debut_cit, Fin_cit\}$ avec $Debut_cit = N_Req + K + 1$, $Fin_cit = N_Req + 2K + 2$. Cette tournée sera représentée par un vecteur principal Y .

Définition 12 Vecteur principal Y

Le vecteur principal Y sera indexé sur les couples (i, j) avec $i \neq j$ et $i, j \in X_depot_prod \cup X_depot_h2 \cup \{Debut_cit, Fin_cit\}$. La sémantique de $Y_{i,j}$ est :

$$Y_{i,j} = \begin{cases} 1 & \text{si } i \text{ précède } j \text{ dans la liste de la citerne (liste_production)} \\ 0 & \text{sinon.} \end{cases}$$

Les opérations de productions et de recharges d'hydrogène seront représentées par :

1. le vecteur principal Y
2. un tableau *label_out_citerne* indexé en ligne par $X_depot_prod \cup X_depot_h2 \cup \{Debut_cit, Fin_cit\}$ et en colonne par :

$$j \in X_depot_prod \cup X_depot_h2 \cup \{Debut_cit, Fin_cit\},$$

— *Actif*

$$Actif[j] = \begin{cases} 1 & \text{si } j \in \text{liste_production} \\ 0 & \text{sinon.} \end{cases}$$

$$Actif[j] = \sum_i Y_{j,i}$$

— la date de départ *time_arrive_j*

- la date d'arrivé $time_depart_j$
- la quantité d'énergie chargée $energ_j$
- la quantité d'hydrogène dans la citerne cit_j

Contraintes du système « production d'énergie »

1. $(Y_{i,j} = 1) \wedge (i = N_Req + K + 1) \implies time_arrive[i] = 0$ le temps de commencement de la production est 0
2. $(Y_{i,j} = 1) \wedge (i = N_Req + 2K + 2) \implies time_depart[i] = TMAX$ la production finie au temps $TMAX$
3. $(Y_{i,j} = 1) \implies time_depart[j] \geq time_arrive[i]$
4. $(Y_{i,j} = 1) \wedge i \in X_depot_h2 \implies time_depart[i] = time_arrive[i] + tmp_h2$
5. $(Y_{i,j} = 1) \wedge i \in X_depot_prod \implies energ[i] = \int_{t=time_arrive[i]}^{time_depart[i]} rend(t) dt$. Le calcul de cet intégrale est difficile car $rend(t)$ est variable
6. Pour tout $i, j \in X_depot_prod \cup X_depot_h2$, $(Y_{i,j}^k = 1) \implies cit[j] = cit[i] + imput[depot_prod][i] \times energ[i] - imput[depot_h2][i] \times energ[i]$
7. $cit[Debut_cit] = init_cit$
8. $cit[Fin_cit] \geq init_cit$
9. pour tout $i \neq Debut_cit, Fin_cit$, $CAP_citerne \geq cit[i] \geq 0$
10. $actif_i = 1 \wedge i \in X_depot_prod \implies energ_i = \int_{t=time_arrive_i}^{time_depart_i} rend(t) dt$

Pour tout $i \in X_depot_prod \cup X_depot_h2$, on notera $Actif^{h2}[i] = \sum_j Y_{i,j}$ et $i \in X_depot_elect \cup X_depot_h2$ $Actif^{veh}[i] = \sum_{j,k} Z_{i,j}^k$

Les contraintes structurelles sur Y sont :

$$Y_{N_Req+K+1,i} = \sum_i Y_{i,N_Req+2K+2} \quad (xix)$$

$$Y_{N_Req+K+1,i} = 1 \quad (xx)$$

$$Actif^{h2}[i] = \sum_j Y_{i,j} \quad (xxi)$$

$$\sum_j Y_{i,j} = \sum_j Y_{j,i} \quad (xxii)$$

$$Actif^{h2}[i] = Actif^{veh}[i] \quad (xxiii)$$

1. $\sum_i Y_{N_Req+K+1,i} = \sum_i Y_{i,N_Req+2K+2} = 1$: lorsqu'on entre en un point on en ressort sauf si c'est le début ou la fin de la production
2. si $i \in X_depot_prod \cup X_depot_h2$, $Actif^{h2}[i] = \sum_j Y_{i,j} = \sum_j Y_{j,i} \leq 1$
3. si $i \in X_depot_elect \cup X_depot_h2$, $Actif^{h2}[i] = Actif^{veh}[i]$

Pour traduire le fait qu'à un instant donné t , la quantité d'électricité ne dépasse pas la puissance maximale disponible nous allons ajouter une variable additionnelle $Actif^*[i, t]$ avec t entre 0 et $TMAX$ et $i \in X_depot_elect \cup X_depot_h2 \cup X_depot_prod$

$$Actif^*[i, t] = \begin{cases} 1 & \text{si } i \text{ est actif à l'instant } t \text{ (période entre } t \text{ et } t+1) \\ 0 & \text{sinon.} \end{cases}$$

1. $Actif^*[i, t] = 1 \implies t \leq time_depart[i] - 1$
2. $Actif^*[i, t] = 1 \implies t \geq time_arrive[i]$
3. $Actif^*[i, t] = 0 \implies t \leq time_depart[i] - 1 \vee t \geq time_arrive[i]$
4. $\forall t, P \times (\sum_{j \in X_depot_elect} Actif^*[i, t]) + Q \times \sum_{j \in X_depot_prod} Actif^*[i, t] \leq PMAX$:
A un instant donné t on ne doit pas être entrain d'utiliser plus de $PMAX$ puissance électrique
5. avec la nouvelle variable $Actif^*$, le calcul de $energ$ devient $energ[i] = \int_{t=time_arrive[i]}^{time_depart[i]} rend(t) dt = \sum_t Actif^*[i, t] \times rend(t)$

Remarque 2 En introduisant $Actif^*[t]$ et les contraintes sur t , le nombre de variables augmentent en même temps que $TMAX$. Par exemple si on mesure le temps en seconde, pour une heure on créerait 3600 variables ce qui est énorme. Quelle méthode pouvons-nous mettre sur pied pour nous passer de $Actif^*$?

Piste de solution :

Soit $i, j \in X_depot_elect$, $U_{i,j}$ est une variable qui exprime le fait que i vient avant j dans le tour, c'est-à-dire que $time_depart[i] \leq time_arrive[j]$.

$$U_{i,j} = \begin{cases} 1 & \text{si } i \text{ vient avant } j \text{ (} time_depart[i] \leq time_arrive[j] \text{)} \\ 0 & \text{sinon.} \end{cases}$$

Soit $V_{i,j} = U_{i,j} + U_{j,i}$, qui permet de traduire le fait que i et j se coupent, c'est-à-dire

qu'à un certain moment il y a chargement simultané de deux véhicules aux stations i et j .
On a trois cas de figure (voir figure 2.12).

$$V_{i,j} = \begin{cases} 0 & \text{si } i \text{ et } j \text{ se coupent} \\ 1 & \text{sinon.} \end{cases}$$

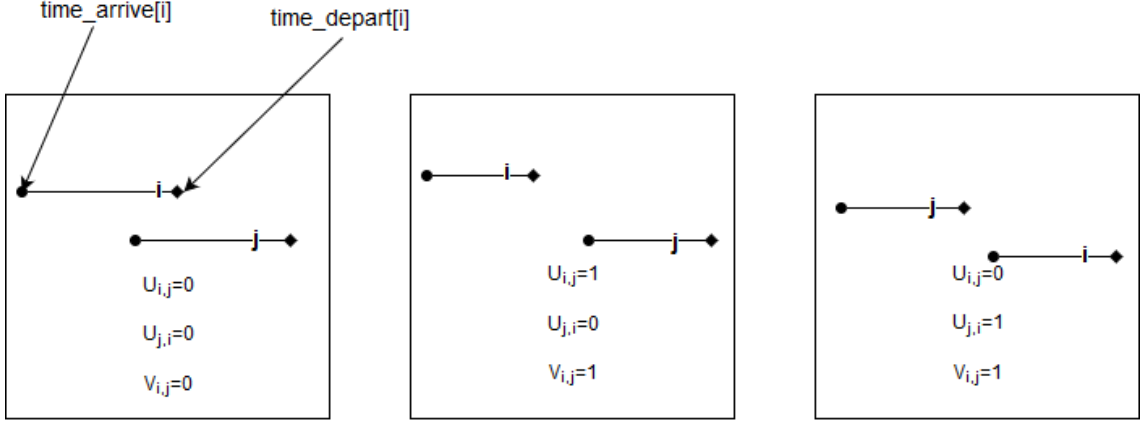


FIGURE 2.12 – Illustration des trois cas de figure possible pour $V_{i,j}$.

Soit $i \in X_depot_prod$, $j \in X_depot_elect$, $U_{i,j}^*$ est une variable qui exprime le fait que i vient avant j dans le tour, c'est-à-dire que $time_depart[i] \leq time_arrive[j]$.

$$U_{i,j}^* = \begin{cases} 1 & \text{si } i \text{ vient avant } j \text{ (} time_depart[i] \leq time_arrive[j] \text{)} \\ 0 & \text{sinon.} \end{cases}$$

Soit $V_{i,j}^* = U_{i,j}^* + U_{j,i}^*$, qui permet de traduire le fait que i et j se coupent, c'est-à-dire qu'à un certain moment il y a production d'hydrogène en i et chargement d'un véhicule j c'est-à-dire que simultanément la micro-usine et un véhicule utilisent de l'électricité.

$$V_{i,j}^* = \begin{cases} 0 & \text{si } i \text{ et } j \text{ se coupent} \\ 1 & \text{sinon.} \end{cases}$$

Avec ces variables $(U_{i,j}^*, V_{i,j}^*)$, nous pouvons contrôler la quantité de puissance électrique utilisée à un instant donné tel que cette puissance ne dépasse pas P_{MAX} en limitant le nombre d'appareils qui utilisent simultanément de l'électricité au dépôt. Si la micro-usine est en marche alors le nombre de véhicule qu'on peut charger simultanément est valeur

entière de $\left(\frac{PMAX-Q}{P}\right)$, sinon le nombre de véhicule qu'on peut charger simultanément est valeur entière de $\left(\frac{PMAX}{P}\right)$. Par exemple, si $P = 25kWh$, $PMAX = 80kWh$ et $Q = 40kWh$, on sait qu'on ne peut pas charger plus d'un véhicule $\left(\frac{80-40}{25}\right) = 1$ pendant qu'on produit de l'hydrogène.

Nous avons donc les contraintes :

- $\sum_{i,j} V_{i,j} \geq 1$ avec $j \in i_1, i_2, \dots, i_{\left(\frac{PMAX-Q}{P}\right)}$?
- $\sum_{i,j} V_{i,j}^* \geq 1$ avec $j \in i_1, i_2, \dots, i_{\left(\frac{PMAX-Q}{P}\right)}$?

Remarque 3 Nous devons trouver une solution pour calculer $energ[i]$ en enlevant l'intégrale ($energ[i] = \int_{t=time_arrive[i]}^{time_depart[i]} rend(t) dt$), pour cela nous devons trouver une solution pour linéariser cette expression. En effet, la difficulté vient du faite que $rend(t)$ est variable, si cette valeur était constante on pourrait très facile calculer cet intégrale.

Piste de solution :

Soit W_{i,j_1,j_2} une variable booléenne :

$$W_{i,j_1,j_2} = \begin{cases} 1 & \text{si } t_{j_1} \leq time_arrive[i] \leq t_{j_1+1} \text{ et } t_{j_2} \leq time_depart[i] \leq t_{j_2+1} \\ 0 & \text{sinon.} \end{cases}$$

Le calcul de $energ[i]$ devient $energ[i] = \int_{t=time_arrive[i]}^{time_depart[i]} rend(t) dt = [(t_{j_1+1} - time_arrive[i]) \times rend(t_{j_1}) + \dots + (time_depart[i] - t_{j_2}) \times rend(t_{j_2})]$?

Par exemple si on a $W_{i,t_1,t_4} = 1$ et $rend(t_1) = 2$, $rend(t_2) = 3$, $rend(t_3) = 4$, $rend(t_4) = 4$, l'énergie est : $energ[i] = [(t_2 - time_arrive[i]) \times rend(t_1) + (t_3 - t_2) \times rend(t_2) + (t_4 - t_3) \times rend(t_3) + (time_depart[i] - t_4) \times rend(t_4)]$

2.5.3 Coût

1. Coût hydrogène

En utilisant le vecteur $Y_{i,j}$, le coût hydrogène si la production de l'hydrogène est constante par unité de temps (et vaut $cout_h2$) est :

$$\sum_{i \in X_depot_prod} [\sum_{j \in X_depot_prod} actif[j] \times Y_{i,j}] \times cout_setup + [cout_h2 \times (time_depart[i] - time_arrive[i])].$$

2. Coût électricité

$$\begin{aligned}
& - [P \times (\sum_{j \in X_depot_elect} \sum_t cout_elect(t) \times Actif^*[j, t]) \\
& \quad + Q \times \sum_{j \in X_depot_prod} \sum_t cout_elect(t) \times Actif^*[j, t] \\
& - [P \times (\sum_{j \in X_depot_elect} \int_{t=time_arrive[j]}^{time_depart[j]} cout_elect(t) dt) \\
& \quad + Q \times \sum_{j \in X_depot_prod} \int_{t=time_arrive[j]}^{time_depart[j]} cout_elect(t) dt]
\end{aligned}$$

2.6 Perspectives

A cause de la modélisation par duplication de stations, il peut arriver qu'à un moment deux véhicules se retrouvent sur des stations physiquement identiques, dans le cadre de véhicules autonomes, nous devons mettre en place une solution qui empêche la collision entre les véhicules. De plus, au cas où il y a perturbation sur le plus court chemin qu'on devrait emprunter, il faut pouvoir faire emprunter un autre chemin aux véhicules.

Pour résoudre ce problème trois approches sont possibles :

1. **route first production last** : cette technique consiste à construire d'abord les tournées réalisables et ensuite à injecter la recharge des véhicules et la production d'Hydrogène tel que cela match avec les tours construits.
2. **production first route last** : consiste à planifier d'abord la production et la recharge des véhicules et ensuite à construire les tournées pour qu'elle match avec cette planification.
3. **Intégrer toutes les variables de décisions du système globale**, c'est-à dire résoudre les deux sous-systèmes simultanément.