Université de Montréal

Le calcul des plus courts chemins statiques et temporels: synthèse, implantations séquentielles et parallèles

par

Nicolas TREMBLAY

Département d'informatique et de recherche opérationnelle
Faculté des arts et sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)
Informatique

Février 1998

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Le calcul des plus courts chemins statiques et temporels: synthèse, implantations séquentielles et parallèles

présenté par:

Nicolas TREMBLAY

a été évalué par un jury
composé des personnes suivantes:

Bernard Gendron	Président-rapporteur
Michael Florian	Directeur de recherche
Jacques Ferland	Membre du jury

Mémoire accepté le: ...20 février 1998.....

Sommaire

Ce mémoire traite du calcul des plus courts chemins dans un graphe orienté. Plus précisément, nous nous intéressons au développement d'implantations *séquentielles* et *parallèles* d'algorithmes de calcul des plus courts chemins *statiques* et *temporels* dans un contexte de **planification des transports.**

Bien que ce mémoire soit axé davantage sur l'aspect développement, nous accordons une place importante à l'étude approfondie des algorithmes de calcul des plus courts chemins par le biais d'une revue de la littérature qui se veut la plus complète possible. Nous nous intéressons, dans cette synthèse, aux principales approches de calcul statiques incluant la variante considérant les délais et les interdictions sur les mouvements aux intersections. De plus, nous traitons de la dimension temporelle du problème en présentant notamment des approches de résolution très récentes.

Nous réalisons des implantations séquentielles d'algorithmes de calcul des plus courts chemins statiques et temporels. Dans le cas statique, nous développons des implantations permettant de calculer les plus courts chemins dans un réseau comportant des délais et des interdictions sur les mouvements aux intersections ainsi que des implantations spécialisées dans le calcul des plus courts chemins conventionnels. Dans le cas temporel, nous développons des implantations de trois algorithmes récemment proposés dans la littérature[1]. Le cadre pratique de la *planification des transports* sert de base au développement de nos implantations en définissant notre manière de modéliser les réseaux et de calculer les plus courts chemins. Les implantations séquentielles effectuées sont analysées afin de vérifier certains résultats déjà connus et d'en étudier de façon plus approfondie les comportements. Les réseaux de transport urbains des villes canadiennes de Winnipeg, Ottawa et Montréal sont utilisés afin d'évaluer les performances des différentes implantations.

[1] Il s'agit des algorithmes proposés par Ziliaskopoulos et Mahmassani [57], Chabini [12] et Pallottino et Scutellà [46].

Dans le même ordre d'idées que l'étude d'impact du calcul parallèle effectuée par Florian, Chabini et Le Saux [25][2], nous réalisons des implantations parallèles des trois algorithmes de calcul des plus courts chemins temporels traités dans le cas séquentiel. Plus spécifiquement, nous nous intéressons, d'une part, à l'utilisation de l'environnement **PVM** (*Parallel Vitual Machine*) dans le but d'étudier les performances du calcul distribué sur un réseau de 16 stations de travail *SUN SPARC Ultra 1/140* et, d'autre part, à l'usage de la **programmation multithreads** dans le but d'évaluer les performances du calcul sur une machine à 8 processeurs à mémoire partagée (*SUN SPARC Server 1000*). Les implantations parallèles développées suivent le schéma *maître-esclave*.

L'impact du calcul parallèle sur le calcul des plus courts chemins temporels est mesuré en présentant les résultats obtenus pour les réseaux de transport urbains des trois villes canadiennes citées précédemment. Nous remarquons que les deux environnements de parallélisation sont très efficaces dans un contexte de calcul brut. Toutefois, l'utilisation de l'environnement *PVM* pour le calcul sur une architecture à mémoire distribuée (réseau de 16 stations de travail *SUN SPARC Ultra 1/140*) devient en fait très inefficace lorsque tous les *esclaves* communiquent leurs résultats au *maître* en raison de la taille imposante des messages transmis. Par contre, l'utilisation de la *programmation multithreads* pour le calcul sur une architecture à mémoire partagée (*SUN SPARC Server 1000*) n'implique aucune communication de données ou de résultats étant donné l'utilisation d'une mémoire globale. Cet environnement de parallélisation est, par conséquent, plus efficace.

Finalement, nous vérifions l'impact de l'utilisation du langage orienté objet *C++* dans le développement d'outils de calcul des plus courts chemins statiques et temporels. Nous démontrons l'intérêt d'utiliser ce langage de programmation, d'une part, en développant des implantations efficaces et, d'autre part, en développant des outils facilement réutilisables par tous modèles d'optimisation de réseaux (notamment en *gestion des transports*) nécessitant un calcul des plus courts chemins.

Mots clés: plus courts chemins, statique, temporel, traitement parallèle, environnement *PVM*, *programmation multithreads*.

[2]Cette étude traite entre autre de la parallélisation du calcul des plus courts chemins statiques.

Table des matières

Liste des tableaux

Liste des figures

Remerciements

Je veux maintenant accorder une attention particulière aux personnes qui ont rendu possible la réalisation de ce mémoire.

En tout premier lieu, je désire remercier humblement mon directeur de recherche, Monsieur le Professeur Michael Florian. En tout temps, lors de la réalisation de ce projet, j'ai pu profiter de ses précieux conseils, de son immence expérience en plus de jouir de sa confiance. Merci profondément de m'avoir convaincu de terminer mes études de deuxième cycle et de m'avoir offert le support financier nécessaire.

Je tiens également à remercier le Centre de recherche sur les transports, et plus particulèrement toute l'équipe administrative incluant évidemment celle au support technique. Leur professionnalisme se voit à chaque jour et favorise une ambiance de recherche exemplaire.

Bien sûr, je veux aussi remercier le Département d'informatique et de recherche opérationnelle et le Centre de recherche sur les transports pour leurs contributions respectives à ma formation académique et professionnelle.

Je désire également remercier une amie, Isabelle, qui a bien voulu prendre le temps de lire ce mémoire afin d'en vérifier le français.

Merci aussi à tous ceux que je ne nomme pas et qui ont su m'encourager lorsqu'ils le sentaient nécessaire.

Finalement, je souhaite accorder tout mon respect à mes parents qui m'ont supporté plus que quiconque, et ce, bien avant la réalisation de ce mémoire...

Introduction

Le calcul des plus courts chemins représente un problème fondamental en optimisation de réseaux. L'imposante bibliographie traitant de ce sujet nous confirme l'intérêt y étant rattaché. Les travaux de Denardo et Fox [18], Glover, Klingman, Philips et Shneider [30], Gallo et Pallottino [28, 27] et Cherkassky, Goldberg et Radzik [15] constituent quelques exemples de l'importance accordée au calcul des plus courts chemins. La littérature contient plusieurs modèles appartenant à des domaines de recherche très variés et requérant un algorithme de calcul des plus courts chemins. Un de ces domaines de recherche, qui nous intéresse particulièrement, est celui de la **planification des transports**.

Dans le domaine de la *planification des transports*, la nature du calcul des plus courts chemins demandée peut varier beaucoup d'un modèle à l'autre. Par exemple, les *modèles de simulation* ([4, 38]) exigent un algorithme de calcul des plus courts chemins tenant compte de la dimension temporelle. On peut également être intéressé, comme dans le cas du problème d'affectation statique du trafic ([24, 51]), au calcul des plus courts chemins à partir d'une origine vers un sous-ensemble de noeuds destinations. De plus, il est possible que le modèle considéré exige de prendre en considération les mouvements aux intersections, c'est-à-dire les délais et les interdictions associés à chaque intersection. Par conséquent, étant donné les besoins variés des différents modèles reliés à la *planification des transports*, les algorithmes de calcul des plus courts chemins se doivent d'être particulièrement bien adaptés, voire même spécialisés.

Dans ce mémoire, nous nous intéressons principalement au calcul des plus courts chemins dans un contexte de *planification des transports*. Notre objectif consiste dans un premier temps à développer des implantations séquentielles d'algorithmes de calcul des plus courts chemins statiques et temporels. Les implantations développées constituent un ensemble d'outils qui pourront facilement être réutilisés par la suite. Dans un deuxième temps, nous visons le développement d'implantations parallèles d'algorithmes de calcul des

plus courts chemins temporels récemment proposés [3]. Nous utilisons, d'une part, l'environnement **PVM** (*Parallel Vitual Machine*) dans le but d'étudier les performances du calcul distribué sur un réseau de 16 stations de travail *SUN SPARC Ultra 1/140* et, d'autre part, la **programmation multithreads** dans le but d'évaluer les performances du calcul sur une machine à 8 processeurs à mémoire partagée (*SUN SPARC Server 1000*). L'ensemble des résultats obtenus nous permettent, dans le cadre du calcul des plus courts chemins temporels, de procéder à une analyse profonde des deux environnements de parallélisation concernés.

Ce mémoire est organisé de la façon suivante. Dans le **chapitre 1**, nous présentons une revue de la littérature concernant les algorithmes de calcul des plus courts chemins statiques et temporels. Nous nous intéressons, d'une part, au calcul des plus courts chemins statiques, incluant la variante considérant les délais et les interdictions sur les mouvements aux intersections et d'autre part, au calcul des plus courts chemins temporels. Le **chapitre 2** présente des implantations séquentielles d'algorithmes de calcul des plus courts chemins statiques et temporels et comparent leurs efficacités sur différents réseaux urbains. Finalement, le **chapitre 3** traite de la parallélisation d'algorithmes de calcul des plus courts chemins temporels utilisant les deux environnements parallèles mentionnés plus haut. Enfin, la conclusion résume les pricipaux résultats de ce mémoire et présente différentes avenues de recherche possibles.

[3] Il s'agit des algorithmes proposés par Ziliaskopoulos et Mahmassani [57], Chabini [12] et Pallottino et Scutellà [46].

Chapitre 1

Revue de la littérature concernant les algorithmes de calcul des plus courts chemins statiques et temporels

Ce premier chapitre propose une revue de la littérature concernant les algorithmes de calcul des plus courts chemins dans un graphe orienté. Nous nous intéressons aux algorithmes de calcul statiques et temporels. Notre étude du cas statique inclut les algorithmes de calcul des plus courts chemins considérant les délais et les interdictions aux intersections. Cette synthèse approfondie sert de base théorique pour le développement effectué dans les chapitres suivants. De plus, elle représente une source d'information essentielle à tout travail futur relié au sujet.

Dans la section 1.1, nous traitons du calcul des **plus courts chemins statiques**. Cette variante est bien sûr celle comportant le plus de travaux. Nous tentons donc de résumer ceux qui nous apparaissent comme les plus importants. Le calcul des **plus courts chemins temporels** est le sujet de la section 1.2. On retrouve, dans cette section, les algorithmes récemment proposés ainsi que ceux qui ont servi de base au développement de ces derniers. Enfin, la section 1.3 est consacrée au cas où il existe des délais ou des interdictions associés aux mouvements à certains noeuds (*intersections*). Comme dans le cas temporel, ce type de problème est particulièrement intéressant dans un contexte de **planification en transport**.

1.1 Algorithmes de calcul des plus courts chemins statiques

Cette section est divisée en quatre parties. D'abord, nous donnons la formulation du problème, de même que la notation qui sera utilisée. La deuxième partie discute de l'algorithme générique et des classes d'algorithmes. Les deux dernières parties traitent respectivement des méthodes utilisant les techniques d'**ajustements progressifs** (*label-correcting methods*) et d'**extensions sélectives** (*label-setting methods*) comme approche de calcul.

1.1.1 Terminologie et notation

Soient $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ un graphe connexe orienté avec $\mathcal{N}$ et $\mathcal{A}$ représentant respectivement l'ensemble des noeuds i et l'ensemble des arcs $a = (i, j)$. Nous posons $m = |\mathcal{A}|$ et $n = |\mathcal{N}|$. Au graphe $\mathcal{G}$ est associée une fonction de longueur (coût) $d : \mathcal{A} \rightarrow \Re$. On note d_{ij} (ou d_a) la longueur (coût) de l'arc $a = (i, j)$. Les longueurs d_{ij} peuvent être positives ou négatives avec l'hypothèse qu'il n'existe pas de circuits de longueur négative dans $\mathcal{G}$. La longueur d'un chemin est la somme des longueurs de ses arcs. Un chemin $\mathcal{P} = i_1, i_2, i_3, ..., i_k$ est élémentaire si, pour $p \neq q$ et $i_p, i_q \in \mathcal{P}$, on a que $i_p \neq i_q$. On suppose dans la suite du texte que chaque chemin $\mathcal{P}$ est élémentaire.

Soit un noeud origine r, on définit par $\mathcal{T}(r)$, l'arborescence de racine r contenant les plus courts chemins de r à i, pour tous les noeuds $i \neq r$. On définit également $\mathcal{A}_i^+ = \{(i, j) \in \mathcal{A}\}$ et $\mathcal{A}_i^- = \{(j, i) \in \mathcal{A}\}$, représentant respectivement l'ensemble des arcs sortant (*forward structure*) et l'ensemble des arcs entrant (*backward structure*) au noeud i. Finalement, soit π_i, la longueur d'un plus court chemin de r à i.

Le problème consiste à trouver, à partir d'un noeud r (*origine*), l'arborescence $\mathcal{T}(r)$ des plus courts chemins vers tous les autres noeuds $i \in \mathcal{N}(\textit{destinations}), i \neq r$.

1.1.2 Algorithme générique et classes d'algorithmes

Conditions d'optimalité

Soit $\mathcal{P}$ un plus court chemin de r à j. Nous voulons formuler un ensemble d'équations qui doivent être satisfaites par les longueurs π_k des plus courts chemins reliant un noeud r à chaque noeud $k \in \mathcal{P}$. Montrons d'abord que les longueurs π_k représentent bien les longueurs des plus courts chemins de r à k. Soit (k, j) l'arc terminal du plus court chemin $\mathcal{P}$, alors nous avons que $\pi_j = \pi_k + d_{kj}$. Donc, la portion de $\mathcal{P}$ reliant r gà k doit être un plus court chemin de r à k (le principe d'optimalité en programmation dynamique s'applique en raison de l'additivité des longueurs d_{ij}). Clairement, si nous voulons le plus court chemin de r à

j, alors il nous faut choisir le noeud k pour lequel $\pi_k + d_{kj}$ est aussi petit que possible. Ce raisonnement s'applique évidemment pour chaque arc $(i,j) \in \mathcal{P}$. Donc, les longueurs des plus courts chemins doivent **nécessairement** satisfaire le système d'équations suivant:

$$\begin{cases} \pi_r & = 0 \\ \pi_j & = \min_{i:(i,j)\in\mathcal{A}_j^-}\{\pi_i + d_{ij}\},\ \forall j \neq r \end{cases}$$

Ces équations sont connues sous le nom d'*équations de Bellman* (Bellman [5]). En fait, Bellman a montré que, si le graphe $\mathcal{G}$ est tel qu'il n'existe pas de circuit de longueurs négatives et qu'il existe un chemin de r à j, pour chaque noeud j, alors les longueurs des plus courts chemins sont entièrement décrites par ces équations.

La plupart des algorithmes de plus courts chemins proposés dans la littérature cherchent à résoudre les *équations de Bellman* afin d'atteindre l'optimalité. L'optimalité des longueurs des plus courts chemins peut être définie ainsi:

Conditions d'optimalité de Bellman: *Supposons que nous avons un vecteur* $\pi = (\pi_1, \pi_2, ..., \pi_n)$ *qui vérifie* $\pi_j \leq \pi_i + d_{ij}, \forall (i,j) \in \mathcal{A}$. *Alors,* $\forall (i,j) \in \mathcal{T}(r)$, *nous avons*

$$\pi_j = \pi_i + d_{ij}.$$

Les *conditions d'optimalité de Bellman* sont des **conditions nécessaires et suffisantes** pour qu'un vecteur de longueurs $\pi = (\pi_1, \pi_2, ..., \pi_n)$ représentent les longueurs des plus courts chemins d'un noeud origine r vers tous les autres noeuds $j \in \mathcal{N}$.

Algorithme générique

Quelque soit l'algorithme de calcul des plus courts chemins considéré, on peut dire qu'ils cherchent tous, pour la plupart, à exécuter les opérations suivantes [27, 28]:

1. Initialiser une arborescence $\mathcal{T}(r)$ et, pour chaque sommet $i \in \mathcal{N}, i \neq r$, on pose π_i égal à la longueur du chemin de r à i dans $\mathcal{T}$. S'il n'existe pas de chemin de r à i, alors poser $\pi_i = +\infty$.

2. Soit $(i,j) \in \mathcal{A}$ un arc tel que $\pi_i + d_{ij} < \pi_j$. Alors, poser $\pi_j = \pi_i + d_{ij}$ et mettre à jour $\mathcal{T}(r)$ en remplaçant l'arc incident dans j par l'arc (i,j).

3. Répéter (2) jusqu'à ce que les conditions d'optimalité soient satisfaites. C'est-à-dire:

$$\pi_j \leq \pi_i + d_{ij}, \forall (i,j) \in \mathcal{A}.$$

L'efficacité d'un algorithme exécutant ces opérations sera grandement affectée par la façon dont les arcs violant la condition d'optimalité seront sélectionnés. Étant donné que, dans plusieurs cas, $n \ll m$, il peut être raisonnable de sélectionner les noeuds plutôt que les arcs [27, 28]. De cette manière, on peut explorer l'ensemble des arc sortant d'un noeud et, ensuite, poursuivre la suite des opérations.

Un algorithme générique [27, 28] effectuant le calcul des plus courts chemins d'une origine r vers tous les noeuds et basé sur l'exploration des listes d'arc sortant, peut être donné comme suit. Associons à chaque noeud i un prédécesseur p_i (le plus récent noeud à partir duquel le noeud i a vu son étiquette modifiée) et considérons un ensemble de candidats $\mathcal{Q}$ où les opérations de sélection et d'insertion d'élements sont permises. L'algorithme générique est alors:

$\pi_r = 0;\ p_r = -1;\ \pi_i = +\infty, \forall i \in \mathcal{N} - r;\ \mathcal{Q} = \{r\};$
tant que $\mathcal{Q} \neq \emptyset$ faire
  choisir $i \in \mathcal{Q};\ \mathcal{Q} = \mathcal{Q} - \{i\};$
  pour chaque $j : (i,j) \in \mathcal{A}_i^+$ faire
    si $\pi_j > \pi_i + d_{ij}$ alors
      $\pi_j = \pi_i + d_{ij};\ p_j = i;$
      si $j \notin \mathcal{Q}$ alors $\mathcal{Q} = \mathcal{Q} + \{j\};$

Classes d'algorithmes

La procédure générique présentée précédemment ne spécifie pas la façon dont les noeuds sont sélectionnés. En fait, chaque règle de sélection affecte la façon dont le graphe $\mathcal{G}$ est exploré et, par conséquent, implique une stratégie particulière de recherche. On peut classer les différentes stratégies de recherche en trois groupes qui représentent les trois approches les plus utilisées [27, 28]. Ces stratégies sont la recherche en largeur (*breath-first-search*), la recherche en profondeur (*deph-first-search*) et la recherche selon la plus petite longueur (*short-first-search*). L'utilisation d'une **liste** comme structure de données est la caractéristique principale des stratégies de recherche en largeur et en profondeur. De son côté, la recherche selon la plus petite longueur est la plupart du temps implantée en utilisant une **queue de priorité** comme structure de données. Les algorithmes utilisant les deux premières stratégies sont souvent regroupés sous le nom de méthodes d'**ajustements**

progressifs (*label-correcting methods*) tandis que ceux utilisant la recherche selon la plus petite longueur se retrouvent dans la classe des méthodes d'**extensions sélectives** (*label-setting methods*). Cette classification des algorithmes est plus répandue que la première et sera adoptée dans la suite du texte. Toutefois, Gallo et Pallottino [27, 28] suggère la première classification plutôt que la seconde parce qu'elle fait référence à la structure des algorithmes et non à leurs comportements. (En fait, certains algorithmes peuvent appartenir à l'une ou l'autre des classes selon le type de données considéré. Johnson [33] montre, par exemple, que l'*algorithme de Dijkstra* (voir section 1.1.4) appartient à la classe des méthodes d'*extensions sélectives* si les longueurs des arcs sont non-négatives mais qu'il devient une méthode d'*ajustements progressifs* s'il existe certains arcs dont les longueurs sont négatives.)

1.1.3 Méthodes d'ajustements progressifs (*label-correcting methods*)

Comme il a été mentionné précédemment, les méthodes qui utilisent cette stratégie représentent l'ensemble des candidats $\mathcal{Q}$ sous forme de liste. Dans la plupart des ouvrages, on représente une liste sous forme de **queue** ou de **pile** (*stack*) [52]. Une *queue* est une liste qui permet l'ajout d'éléments à la fin de la liste et le retrait d'éléments au début de la liste. Ce type de structure est utilisé pour implanter la recherche en largeur dans le graphe. Une *pile* est une liste permettant l'ajout et le retrait d'éléments au début de la liste et elle permet l'implantation de la recherche en profondeur.

Quoiqu'il est facile d'imaginer un algorithme utilisant une *pile*, la littérature ne contient aucun exemple d'un tel algorithme. En fait, l'utilisation de la recherche en profondeur pour le calcul des plus courts chemins ne semble pas naturelle. Il s'avère q'une stratégie qui explore le graphe de façon concentrique autour de l'origine, comme c'est le cas avec la recherche en largeur, est plus avantageuse. (Gallo et Pallottino ont vérifié numériquement que l'utilisation d'une *pile* donne une procédure inefficace comparativement à l'utilisation d'une *queue*. En fait, Kershenbaum [36] montre au moyen d'un exemple qu'un algorithme de calcul des plus courts chemins utilisant une *pile* a un temps d'exécution qui n'est pas polynômialement borné.)

Implantation avec queue

L'utilisation d'une *queue* (figure 1.1) permet une implantation efficace du premier algorithme par *ajustements progressifs* crédité à Bellman [5], Ford [26] et Moore [41]. Cet algorithme consiste à maintenir à chaque itération k, une étiquette $\pi_i^{(k)}$ qui représente la plus courte

distance de l'origine r au noeud i utilisant au plus k arcs. Puisqu'il n'existe pas de circuits de longueur négative, alors les chemins de r à i, pour chaque noeud i sont élémentaires et contiennent au plus $n-1$ arcs. Dans ce cas, on est assuré que $\pi_i^{(k+1)} = \pi_i$, $\forall i$. La procédure est la suivante. Initialement, on pose que $\pi_r^{(1)} = 0$, $\pi_j^{(1)} = d_{rj}$, $\forall (r,j) \in \mathcal{A}$ et $\pi_j^{(1)} = +\infty$ $\forall (r,j) \notin \mathcal{A}$. Ensuite on a que:

$$\pi_j^{(k+1)} = \min\{\pi_j^{(k)}, \min_{i:(i,j)\in\mathcal{A}_j^-} \pi_i^{(k)} + d_{ij}\}, \ \forall j \neq r, \ k = 1,2,...,n-2.$$

Cette procédure itérative se base donc sur les équations d'optimalité développées par Bellman [5] (voir section 1.1.2).

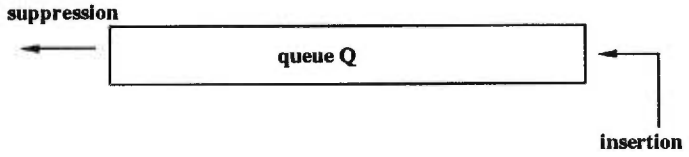

Figure 1.1: Structure de données *queue*.

La complexité de l'algorithme de *Bellman-Ford-Moore* est $\mathcal{O}(nm)$. En effet, chaque noeud peut être inséré dans la *queue* au plus $(n-1)$ fois puisque son étiquette sera modifiée lors du traitement des $(n-1)$ autres noeuds, dans le pire des cas. De plus, pour chaque noeud i sélectionné on doit vérifier, pour $(i,j) \in \mathcal{A}$, si les conditions d'optimalité sont respectées. Donc, on doit faire $\sum_{i\in\mathcal{N}} |\mathcal{A}_i^+| = m$ opérations. La complexité est donc $\mathcal{O}((n-1)m) = \mathcal{O}(nm)$.

Implantation avec deque

D'Esopo et Pape [47] proposent une nouvelle politique d'insertion des candidats dans la liste. Cette nouvelle stratégie peut être appliquée à l'aide d'une structure de données qui combine les propriétés de la queue et de la pile. Une *deque* (figure 1.2) est une structure de données permettant les insertions en début et en fin de liste et les suppressions au début de la liste. La politique d'insertion proposée par D'Esopo et Pape est la suivante:

- la première insertion d'un noeud i se fait en fin de liste;
- les insertions subséquentes d'un même noeud i se font en début de liste;
- les suppressions se font en début de liste.

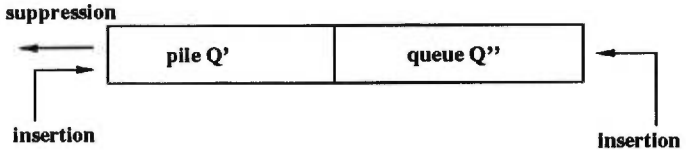

Figure 1.2: Structure de données *deque*.

Cette stratégie a pour effet de combiner la recherche en largeur et la recherche en profondeur. En effet, lorsqu'un noeud i redevient candidat, procéder à l'insertion de ce noeud en début de liste signifie que l'on effectue une recherche en profondeur sur ce noeud. En d'autres mots, on essaie immédiatement de diminuer les étiquettes des noeuds successeurs du noeud i (recherche en profondeur). De cette manière, on peut considérablement diminuer le nombre de mises à jour d'étiquettes normalement effectuées par un algorithme utilisant une recherche en largeur pure.

La complexité d'un algorithme utilisant cette stratégie est $\mathcal{O}(n2^n)$. Pour établir que cette implantation a un temps d'exécution exponentiel en pire cas, Kershenbaum [36] considère le réseau de la figure 1.3. Supposons que la liste d'adjacence du noeud origine 1

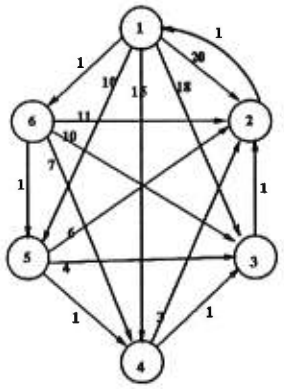

Figure 1.3: Cas pathologique pour l'implantation avec *deque*.

est ordonnée selon l'ordre croissant des numéros de noeuds et que, pour les autres noeuds, cette liste d'adjacence est ordonnée selon l'ordre décroissant des numéros de noeuds. Alors, à chaque fois qu'un noeud est visité, le noeud 2 obtient une nouvelle étiquette et il est ensuite immédiatement revisité puisque, dans ce cas, il est inséré au début de la liste. On remarque que l'étiquette du noeud 2 prendra respectivement toutes les valeurs entre 20 et

5 et que, pour chaque nouvelle étiquette, ce noeud sera revisité. Donc, le noeud numéro 2 sera traité 16 fois. Ceci correspond à $2^{n-2} = 2^4 = 16$, puisque dans cet exemple $n = 6$.

Kershenbaum donne une procédure permettant de construire des réseaux de n noeuds possédant les caractéristiques de l'exemple précédent. Le résultat général est alors que l'étiquette du noeud numéro 2 prendra toutes les valeurs entre $2^{n-2} + n - 2$ et $n - 1$ et qu'il sera visité 2^{n-2} fois. À partir de ce résultat, puisque chaque noeud peut, dans le pire des cas, être visité 2^{n-2} fois, on obtient un temps d'exécution dans l'ordre de $\mathcal{O}(n2^n)$.

Toutefois, malgré son temps d'exécution exponentiel en pire cas, cette implantation est très efficace en pratique. Son efficacité a, entre autre, été vérifiée par Gallo et Pallottino [28] et par Mondou, Crainic et Nguyen [40]. En fait, les caractéristiques du réseau pathologique décrit par Kershenbaum (ordonnancement des listes d'adjacence des noeuds et coûts sur les arcs) sont, généralement, rarement présentes simultanément. Donc, la plupart du temps, cette implantation n'atteindra pas sa borne exponentielle.

Implantation avec 2queue

La structure de données *2queue* (figure 1.4) est une combinaison de deux queues contrairement au jumelage d'une *queue* et d'une *pile* dans le cas d'une *deque*. Pallottino [44, 45]

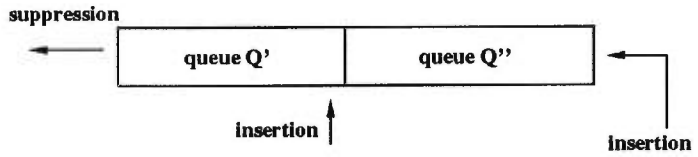

Figure 1.4: Structure de données *2queue*.

suggère l'utilisation de cette structure de données pour implanter l'idée de D'Esopo et Pape. La faiblesse d'une implantation utilisant la structure *deque* est son temps d'exécution exponentiel en pire cas. Ce mauvais comportement est dû, comme nous l'avons vu, à la présence de la *pile* dans la structure de données *deque*. En remplaçant cette *pile* par une *queue*, on élimine le pire cas exponentiel.

L'insertion est la seule opération sur $\mathcal{Q}$ qui est modifiée. En effet, pour cette implantation, les insertions subséquentes d'un noeud i se font à la fin de la queue $\mathcal{Q}'$. De cette manière, lorsque toutes les étiquettes ont été modifiées au moins une fois, l'algorithme se comporte comme une implantation avec *queue*. Donc, puisque dans une implantation avec *queue*, chaque noeud voit son étiquette modifiée au plus n fois, l'utilisation d'une *2queue*

nous donnera $\mathcal{O}(n^2)$ modifications d'étiquettes pour chaque noeud. Les opérations d'insertion et de sélection se font bien sûr en temps constant, tandis que l'initialisation nécessite $\mathcal{O}(n)$ opérations. De plus, pour chaque noeud i sélectionné, on doit toujours vérifier, pour $(i,j) \in A$, si les conditions d'optimalité sont respectées. Ce qui résulte en $\sum_{i \in N} |A_i^+| = m$ opérations. La complexité de cette implantation est donc $\mathcal{O}(mn^2)$.

Les expérimentations de Gallo et Pallottino [28] et Mondou, Crainic et Nguyen [40] montrent qu'une implantation utilisant la structure *2queue* est aussi performante que celle utilisant la structure *deque*. Cette conclusion suggère fortement d'implanter l'idée de D'Esopo et Pape avec une *2queue* afin d'éviter les risques de mauvais comportements, même si, en pratique, l'efficacité de l'implantation avec *deque* a été vérifiée.

Implantation avec seuil

Ce type d'implantation a été proposé par Glover et *al.* [30]. L'idée est de partionner l'ensemble des noeuds candidats $\mathcal{Q}$ en deux ensembles disjoints $\mathcal{Q}'$ et $\mathcal{Q}''$. Ces deux ensembles sont représentés sous forme de *queue*. L'insertion des noeuds candidats dans un de ces deux ensembles est sujet à un certain critère. On définit d'abord une valeur s que l'on appelle le seuil et avec laquelle nous pourrons comparer les étiquettes des noeuds à insérer dans $\mathcal{Q}$. Il s'agit maintenant d'insérer, dans la *queue* $\mathcal{Q}'$, les noeuds dont l'étiquette est inférieure ou égale à s et, ensuite, d'insérer les autres noeuds dans la *queue* $\mathcal{Q}''$. La suppression des noeuds se fait au début de $\mathcal{Q}'$. Lorsque la *queue* $\mathcal{Q}'$ est vide, le seuil est augmenté et les éléments de $\mathcal{Q}''$ qui satisfont la condition énoncée précédemment sont transférés dans $\mathcal{Q}'$. Évidemment, lorsque $\mathcal{Q}'$ et $\mathcal{Q}''$ sont vides, l'algorithme se termine.

La complexité de cette implantation est $\mathcal{O}(n^2m)$. L'opération de sélection d'un noeud nécessite $\mathcal{O}(n)$ opérations en pire cas puisque la *queue* $\mathcal{Q}''$ doit être balayée lorsque $\mathcal{Q}'$ est vide. Notons que la valeur du seuil s ne peut pas décroître et que les valeurs des étiquettes sont décroissantes. Ceci implique qu'un noeud i dont l'étiquette est telle que $d_i \leq s$ n'entrera plus dans $\mathcal{Q}''$. Donc, le nombre de fois que la *queue* $\mathcal{Q}''$ est balayée lorsque $\mathcal{Q}'$ est vide est borné par n et, alors, le coût total de ces opérations de mise à jour des listes est borné par n^2. Entre deux mises à jour des listes, l'algorithme se comporte comme une implantation avec *queue*. Ainsi, un noeud peut être retiré de $\mathcal{Q}'$ au plus n fois. Par conséquent, le nombre de sélections est borné par n^2 et alors le coût total de l'opération de sélection est $\mathcal{O}(n^3)$. Le coût de l'opération d'insertion est $\mathcal{O}(n^2m)$ puisque pour chaque noeud i sélectionné, on doit parcourir la liste d'adjacence $\mathcal{A}_i^+$. On peut donc conclure que la complexité de cette implantation est $\mathcal{O}(n^2m)$.

L'avantage d'utiliser cette implantation avec seuil est d'augmenter la probabilité de choisir l'étiquette de coût minimum (donc de possiblement diminuer les recorrections d'étiquettes) tout en profitant de la simplicité des opérations fournies par les structures de données utilisées.

Stratégie de la plus petite étiquette d'abord (*Small Label First Approach*)

Bertsekas [8] propose une nouvelle façon d'ordonner les noeuds candidats dans la *queue*. Le but étant encore une fois d'essayer de traiter les noeuds ayant les plus petites étiquettes le plutôt possible. (On cherche en fait à simuler l'*algorithme de Dijkstra*[1] à un plus petit coût.) Son heuristique consiste à comparer l'étiquette π_j d'un noeud j qui doit entrer dans la *queue* $\mathcal{Q}$ avec l'étiquette π_i du noeud i à la tête de $\mathcal{Q}$. Si $\pi_j \leq \pi_i$, alors le noeud j est inséré à la tête de la queue $\mathcal{Q}$. Autrement, l'insertion de j se fait à la fin de $\mathcal{Q}$.

Plusieurs variantes de cette méthode peuvent être produites. Cette stratégie peut, entre autre, être combinée avec l'implantation avec seuil, présentée précédemment. Bertsekas a comparé les performances de deux implantations différentes de cette stratégie. Ces implantations sont des modifications des algorithmes LDEQUE et LTHRESH produits par Gallo et Pallottino [28]. Les résultats obtenus par Bertsekas montrent que l'utilisation de la stratégie de la *plus petite étiquette d'abord* est plus efficace que la méthode de d'Esopo et Pape et requiert moins d'itérations que l'algorithme de *Bellman-Ford-Moore*. La combinaison de cette approche avec la méthode avec seuil nécessite également moins d'itérations que l'algorithme LTHRESH mais l'efficacité des deux algorithmes est souvent non distinguable. Cependant, lorsque le choix d'un seuil adéquat est difficile, alors l'algorithme combinant les deux approches devient significativement meilleur.

La complexité d'un algorithme utilisant la stratégie de la *plus petite étiquette d'abord* demeure inconnue jusqu'à présent. Bien qu'il est possible, dans le cas d'un réseau où les longueurs sont non-négatives, de produire un algorithme combinant l'approche avec seuil et ayant un temps d'exécution $\mathcal{O}(nm)$, cet algorithme ne sera pas aussi efficace en pratique. D'après les tests effectués par Bertsekas, on peut supposer que la complexité est pire que $\mathcal{O}(nm)$.

[1] Voir section 1.1.4.

1.1.4 Méthodes d'extensions sélectives (*label-setting methods*)

Les algorithmes qui utilisent cette approche sont des implantations particulières de la méthode originale proposée par Dijkstra [20]. La propriété de base de ces algorithmes est que **si $d_{ij} \geq 0$, $\forall (i,j) \in \mathcal{A}$, alors chaque noeud est retiré de $\mathcal{Q}$ exactement une fois.** Les implantations basées sur cette stratégie partitionnent les noeuds en deux ensembles: l'ensemble des noeuds ayant une étiquette permanente (c'est-à-dire ceux qui ont été retirés de $\mathcal{Q}$) et l'ensemble des noeuds qui sont encore candidats. L'*algorithme de Dijkstra* s'énonce comme suit:

$\pi_r = 0$; $p_r = -1$; $\pi_i = +\infty, \forall i \in \mathcal{N} - r$; $\mathcal{Q} = \{r\}$;
tant que $\mathcal{Q} \neq \emptyset$ faire
  choisir $i \in \mathcal{Q}$ tel que $\pi_i = \min\{\pi_j | j \in \mathcal{Q}\}$; $\mathcal{Q} = \mathcal{Q} - \{i\}$;
  pour chaque $j : (i,j) \in \mathcal{A}_i^+$ faire
    si $\pi_j > \pi_i + d_{ij}$ alors
      $\pi_j = \pi_i + d_{ij}$; $p_j = i$;
      si $j \notin \mathcal{Q}$ alors $\mathcal{Q} = \mathcal{Q} + \{j\}$;

Les implantations efficaces de l'*algorithme de Dijkstra* utilisent habituellement une *queue de priorité* pour représenter l'ensemble des candidats $\mathcal{Q}$. Une *queue de priorité* est un ensemble d'éléments auxquels sont associés une valeur réelle (étiquette). L'opération de suppression d'un élément permet de retirer l'élément d'étiquette minimale. Il existe plusieurs façons d'implanter une *queue de priorité*. À chaque type d'implantation de cette structure de données correspond une implantation particulière de l'*algorithme de Dijkstra*.

Implantations avec listes ordonnée et non ordonnée

La plus simple implantation de l'*algorithme de Dijkstra* est celle qui utilise une liste non ordonnée comme structure de données. Ce type d'implantation n'est pas très efficace puisqu'il faut effectuer une recherche de l'élément d'étiquette minimale à chaque fois que l'on choisit un noeud. L'opération de sélection d'un noeud i est exécutée n fois. À chaque fois, on doit choisir respectivement parmi $n, (n-1), (n-2), ..., 2, 1$ noeuds. Ce qui implique que cette opération est dans l'ordre de $\mathcal{O}(n^2)$. Ensuite, pour chaque noeud i, on doit explorer la liste des arc sortant $\mathcal{A}_i^+$, ce qui se fait en au plus m opérations. L'insertion d'un noeud dans $\mathcal{Q}$ se fait évidemment en temps constant. Donc, la complexité de cette implantation est $\mathcal{O}(n^2 + m) = \mathcal{O}(n^2)$.

L'utilisation d'une liste ordonnée rend plus efficace la sélection de l'élément minimal (en temps constant) au détriment, toutefois, d'une insertion plus coûteuse. En effet, dans ce cas, les opérations d'insertions exécutées lors de l'exploration des arcs incidents vers l'extérieur du noeud i sont dans l'ordre de $\mathcal{O}(n)$. Donc, le coût total de ces opérations est dans l'ordre de $\mathcal{O}(n^2)$ puisque chaque noeud est visité au plus une fois. Donc, cette implantation a également une complexité de $\mathcal{O}(n^2)$. Les expériences numériques de Gallo et Pallottino [28] ont clairement démontrées que ces implantations sont inefficaces.

Implantation avec liste de compartiments

Une manière plus efficace de représenter la *queue de priorité* est de conserver les noeuds candidats dans une liste de *compartiments* (Dial [19], Denardo et Fox [18]). La figure 1.5 nous montre une manière intuitive de disposer les *compartiments*. Dial [19] a été le premier

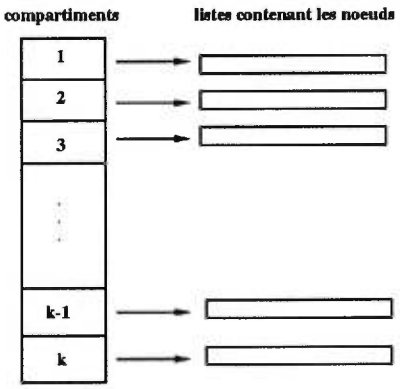

Figure 1.5: Liste de compartiments.

à proposer d'implanter l'*algorithme de Dijkstra* en utilisant ce type de structure de données. Son implantation suppose l'intégralité des longueurs d_{ij} et est basée sur la propriété suivante:

Propriété: *Dans l'algorithme de Dijkstra, les étiquettes π_i qui sont désignées permanentes sont non-décroissantes.*

Cette propriété découle du fait que l'*algorithme de Dijkstra* donne une étiquette permanente au noeud i ayant la plus petite étiquette π_i et, lors de la visite des arcs $(i,j) \in \mathcal{A}_i^+$ menant à la mise à jour des étiquettes π_j, ne décroît pas ces dernières étant donné que les distances d_{ij} sont non négatives.

L'idée de Dial est de maintenir une table de $nL+1$ *compartiments* numérotés $0, 1, 2, ..., nL$. En effet, si on note $L = \max\{d_{ij}\}$, on a que $0 \leq \pi_i \leq nL$ (un plus court

chemin ne peut contenir plus de $(n-1)$ arcs). Le compartiment numéro k contient alors tous les noeuds dont l'étiquette temporaire $\pi_i = k$. L'opération de sélection consiste alors à parcourir la table jusqu'à ce que l'on identifie le premier compartiment non vide k. Les noeuds i du compartiment k sont ensuite sélectionnés un par un et leurs étiquettes sont alors désignées permanentes. Les noeuds j tels que $(i,j) \in \mathcal{A}$ et dont l'étiquette π_j est modifiée lors du traitement d'un des noeuds i du compartiment k sont déplacés dans un autre compartiment k' $(k' > k)$ dans la table. La propriété précédente implique que les compartiments $0, 1, 2, ..., k$ seront toujours vides lors des itérations subséquentes.

Cette structure de données permet de réaliser la suppression et l'ajout d'un élément en un temps constant. De même, on peut vérifier en temps unitaire si un compartiment contient des éléments. Par conséquent, la mise à jour d'une étiquette se fait en temps constant et, alors, il nous faut effectuer au plus m opérations de mise à jour des étiquettes. Toutefois, l'utilisation de cette structure amène un coût supplémentaire qui provient du parcours des $nL+1$ compartiments de la table dans la phase de sélection des noeuds. Par conséquent, la complexité de cette méthode est $\mathcal{O}(m+nL+1) = \mathcal{O}(m+nL)$.

Afin de réduire l'espace mémoire utilisé par cette implantation, il est possible d'utiliser une table de $L+1$ compartiments [1]. Ceci vient du fait que pour tout noeud $j \in \mathcal{Q}$, $\pi_j \leq \pi_i + L$ où i est un noeud avec une étiquette minimale dans $\mathcal{Q}$. Ceci ne change évidemment rien à la complexité de l'algorithme. Cette implantation est donc pseudo-polynômiale puisqu'elle dépend d'un attribut du problème ($L = \max\{d_{ij}\}$). En effet, si $L = 2^n$, on obtient un algorithme qui requiert un temps exponentiel en pire cas [1]. Toutefois, ce type d'implantation obtient de bons résultats en pratique. Gallo et Pallottino [28] et Mondou, Crainic et Nguyen [40] montrent que l'implantation de l'*algorithme de Dijkstra* avec *liste de compartiments* performe assez bien pour la plupart des types de graphes. De plus, Mondou, Crainic et Nguyen remarquent que pour certaines catégories de graphes, la valeur de L n'influence pas le comportement de l'algorithme.

Denardo et Fox [18] proposent une amélioration à l'implantation de Dial en considérant le cas où les longueurs d_{ij} ne sont pas nécessairement entières. Chaque *compartiment* peut alors contenir la liste de noeuds pour lesquels l'étiquette π_i se situe dans l'intervalle $[\lfloor k \rfloor, \lfloor k \rfloor + 1)$. Ils proposent également, dans le but encore de résuire l'espace mémoire utilisé par la structure de données, d'utiliser plusieurs niveaux de compartiments. Goldberg et Silverstein [31] décrivent clairement cette idée. Dans le cas d'une *liste de compartiments* à deux niveaux telle que montrée dans la figure 1.6, nous avons essentiellement un niveau supérieur et un niveau inférieur de compartiments. Le niveau supérieur contient $\sqrt{L+1}$ compartiments, chacun d'eux contenant $\sqrt{L+1}$ compartiments de niveau inférieur. Chaque

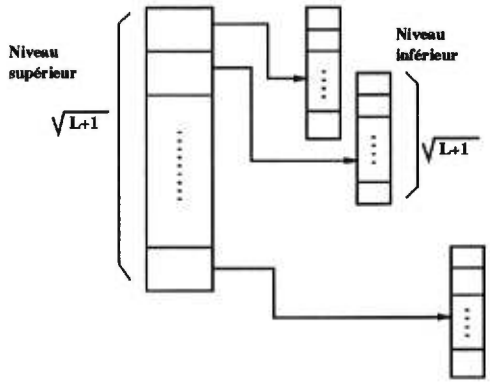

Figure 1.6: Liste de compartiments à deux niveaux.

compartiment du niveau inférieur conserve les étiquettes correspondant à une seule distance (comme dans l'implantation originale à un niveau), tandis que chaque compartiment du niveau supérieur conserve les étiquettes comprises dans un intervalle de largeur $\sqrt{L+1}$. Cet intervalle correspond aux $\sqrt{L+1}$ étiquettes des compartiments du niveau inférieur y étant associé. Afin de connaître la position dans la structure, deux indices, I_{sup} et I_{inf} sont conservés. Lorsque l'étiquette d'un noeud est modifiée, le noeud est déplacé (si nécessaire) au compartiment supérieur correspondant et, ensuite, vers le compartiment inférieur approprié.

L'économie en temps et en espace survient lorsque seule la liste de compartiments inférieurs associée au compartiment supérieur courant I_{sup} est conservée. Dans ce cas, lorsque l'étiquette d'un noeud est modifiée, le noeud est alors déplacé (si nécessaire) vers un nouveau compartiment supérieur. Lorsque I_{sup} change de valeur (dans le cas où tous les compartiments du niveau inférieur sont vides), il devient alors nécessaire d'étendre le compartiment supérieur situé à cette nouvelle valeur I_{sup} et placer les noeuds de ce compartiment dans les compartiments inférieurs correspondants. L'espace occupé par les compartiments inférieurs associés au compartiment supérieur précédent peut alors être réutilisé par les nouveaux compartiments inférieurs du niveau supérieur courant.

Cette façon de procéder implique clairement l'utilisation de seulement $2\sqrt{L+1}$ compartiments. La complexité de cette implantation est de l'ordre de $\mathcal{O}(m+n(1+\sqrt{L}))$ et Goldberg et Silverstein montrent que l'on peut généraliser la structure afin d'utiliser k niveaux de compartiments. Dans ce cas, on obtient une complexité dans l'ordre de $\mathcal{O}(m+n(k+L^{1/k}))$.

Implantation avec arbre partiellement ordonné

Un arbre complet partiellement ordonné [52] est une structure de données dans laquelle chaque élément de la *queue de priorité* correspond à un noeud d'un arbre complet. Une étiquette est associée à chacun des éléments et ceux-ci sont maintenus dans un ordre partiel de sorte que l'élément qui se retrouve à la racine de l'arbre est celui d'étiquette minimale. Un **b-arbre** ($b \geq 2$) partiellement ordonné permet de réaliser l'insertion et la modification d'étiquettes en $\mathcal{O}(\log_b n)$ opérations et requiert un temps $\mathcal{O}(b \log_b n)$ pour effectuer la sélection de l'élément minimal. Par conséquent, la complexité d'un algorithme utilisant ce type de structure est $\mathcal{O}(m \log_b n + nb \log_b n)$. La valeur optimale de b est donnée par $b = \max\{2, \lceil \frac{m}{n} \rceil\}$ [1]. Ceci nous donne alors un temps d'exécution $\mathcal{O}(m \log_b n)$. Cependant, Johnson [34] a démontré empiriquement que le choix de $b = 2$ donne l'implantation la plus efficace. On peut, entre autre, expliquer ce résultat par le fait que les ordinateurs conventionnels fonctionnent en mode binaire. Ceci permet donc une implantation efficace des opérations de multiplication et de division par 2.

Un b-arbre partiellement ordonné est appelé **monceau** lorsque $b = 2$. Dans un monceau chaque élément de la *queue de priorité* est associé à un noeud d'un arbre binaire. Cette structure de données permet la modification d'une étiquette ainsi que l'insertion et la suppression d'un élément en un temps $\mathcal{O}(\log n)$ où n est le nombre d'éléments du monceau [52]. La complexité d'un algorithme utilisant cette structure de données est $\mathcal{O}(m \log n)$. En effet, on doit retirer l'élément à la racine de l'arbre au plus $(n-2)$ fois, ce qui résulte en $\mathcal{O}(n \log n)$ opérations. De plus, le nombre de modifications d'étiquettes est au plus m puisque pour chaque noeud i sélectionné, il est nécessaire de parcourir la liste d'adjacence de ce noeud. Puisque la modification d'une étiquette se fait en un temps $\mathcal{O}(\log n)$, alors l'algorithme effectura $\mathcal{O}(m \log n)$ modifications d'étiquettes. Donc, le temps total requis par cette implantation est dans $\mathcal{O}((m+n) \log n) = \mathcal{O}(m \log n)$. Si le réseau est dense, alors $m \approx n^2$ et la complexité de l'algorithme devient $\mathcal{O}(n^2 \log n)$. Toutefois, si le réseau est tel que $m \approx n$ alors la complexité de cette implantation est $\mathcal{O}(n \log n)$. On remarque donc que l'utilisation d'un monceau est plus efficace, comparativement à une implantation $\mathcal{O}(n^2)$, lorsque le réseau n'est pas complet. On remarque également que si le nombre d'arcs est tel que $m = \mathcal{O}(n^2/\log n)$, alors ce type d'implantation n'est pas plus avantageux qu'une implantation avec liste ordonnée. Ce seuil est donc une façon de justifier l'utilisation d'un monceau pour implanter l'*algorithme de Dijkstra.*

Plusieurs implantations de l'*algorithme de Dijkstra* utilisant un monceau ont été réalisées. Les études effectuées entre autre par Gallo et Pallottino [28] et Mondou, Crainic et

Nguyen [40] ont démontré l'efficacité de ce type d'implantation sur tous les types de réseaux. Il faut toutefois noter que certaines implantations utilisant une technique d'*ajustements progressifs* sont parfois plus efficaces que cette implantation sur les réseaux incomplets.

Autres implantations

Il existe d'autres types d'implantations qui ont été développées au cours des dernières années. Le but étant évidemment de trouver une implantation avec une complexité de calcul inférieure aux implantations connues jusqu'à maintenant. Cependant, ces nouvelles méthodes sont souvent plus intéressantes d'un point de vue théorique qu'en pratique puisqu'il s'agit d'études en pire cas de la complexité de ces implantations. Souvent même, on ne retrouve aucun résultat numérique associé à ces implantations.

L'utilisation d'une structure appelée *arbres de Fibonacci* [1] permet une implantation de l'*algorithme de Dijkstra* dans laquelle toutes les opérations sur la structure sont réalisées dans un temps $\mathcal{O}(1)$ à l'exception de la sélection de l'élément minimal qui requiert un temps $\mathcal{O}(\log n)$. Cette implantation possède donc un temps d'exécution $\mathcal{O}(m + n \log n)$, ce qui représente le meilleur temps polynômial pour le calcul des plus courts chemins.

Mondou, Crainic et Nguyen [40] ont vérifié les performances d'une implantation proposée par Ahuja, Mehlorn, Orlin et Tarjan [2] et reprise par Ahuja, Magnanti et Orlin [1]. Il s'agit d'une méthode hybride utilisant l'idée de Dial [19] et l'implantation originale de l'*algorithme de Dijkstra* avec *queue* où les noeuds d'étiquettes temporaires sont conservés dans une seule et même *queue*. La stratégie suggérée par Dial utilise en quelque sorte $L+1$ queues. Chacune de ces queues contient les noeuds dont l'étiquette temporaire équivaut à une certaine valeur k. L'idée de cet algorithme hybride consiste à conserver dans la k^e queue les noeuds dont l'étiquette est comprise dans un certain intervalle $[pk, pk + p - 1]$ où $p \in N$. Ceci permet de réduire le nombre de compartiments à $1 + \lceil \log L \rceil$. La complexité de ce type d'implantation est $\mathcal{O}(m + n \log(nL))$. Les résultats obtenus par Mondou, Crainic et Nguyen montrent que, dans le cas des réseaux complets ($m \approx n^2$), cette implantation a une efficacité comparable aux méthodes les plus rapides lorsque la taille des problèmes augmente. Cependant, malgré sa meilleure borne théorique, ce type d'implantation n'est pas efficace pour les réseaux incomplets ($m \approx n$). Ahuja, Magnanti et Orlin [1] montrent que la complexité d'une implantation utilisant cette approche jumelée à l'utilisation d'*arbres de Fibonacci* a un temps d'exécution $\mathcal{O}(m + n\sqrt{\log L})$.

1.2 Algorithmes de calcul des plus courts chemins temporels

La section 1.1 nous a permis de constater l'étendue des recherches effectuées au sujet des algorithmes de calcul des plus courts chemins statiques. Cependant, les *modèles de planification en transport* dans lesquels le facteur temps est considéré, ont pris ces dernières années, de plus en plus d'importance. L'introduction de la dimension temporelle dans ces modèles a suffi à renouveler l'intérêt accordé au calcul des plus courts chemins.

Dans la présente section, nous présentons une synthèse des travaux traitant du calcul des plus courts chemins temporels. Dans la première partie de cette section, nous élargissons la notation de la section précédente en introduisant la dimension temporelle et nous définissons certaines propriétés que peut posséder le nouveau modèle temporel. Les deuxième et troisième parties traitent de deux types d'approches que l'on retrouve dans la littérature. Il s'agit respectivement des algorithmes de recherche *origine-vers-noeuds* pour un temps de départ fixe et des algorithmes de recherche *noeuds-vers-destination* pour tous les temps de départ.

1.2.1 Notation, formulations et propriétés

Pour un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, on définit pour chaque lien $(i,j) \in \mathcal{A}$, $d_{ij}(t)$, le temps de parcours requis pour voyager au temps t du noeud i au noeud j. Étant donné un temps de départ t du noeud i, nous avons alors que $t + d_{ij}(t)$ est le temps d'arrivée au noeud j. De même, on définit pour chaque lien $(i,j) \in \mathcal{A}$, $c_{ij}(t)$, le coût encouru pour voyager au temps t du noeud i au noeud j. Généralement, on a que $d_{ij}(t)$ et $c_{ij}(t)$ sont définis pour t appartenant à un intervalle de temps discret $\mathcal{S} = \{t_0, t_1, t_2, ..., t_{\mathcal{M}}\}$ ou à un intervalle de temps continu. Le modèle discret, c'est-à-dire où t est défini pour un intervalle de temps discret $\mathcal{S}$, est d'un intérêt particulier puisque dans le domaine du transport, une discrétisation du temps est généralement effectuée. Étant donné que ce mémoire s'intéresse au contexte de la planification en transport, nous mettrons donc plus d'emphase sur l'aspect discret du problème. Toutefois, le cas continu ne sera pas négligé et nous discuterons également des approches y faisant référence.

Dans un premier temps, la façon de traiter les coûts sur les liens permet de distinguer deux types d'algorithmes de calcul des plus courts chemins temporels. Dans le cas où le coût $c_{ij}(t)$ est ignoré, cela revient à déterminer, étant donné un noeud origine r et un temps de départ t, les chemins minimisant les temps d'arrivée vers tous les autres noeuds $i \neq r$ (*time-dependent least-time path*). Dans le cas où l'on considère le coût $c_{ij}(t)$, nous tentons alors de

déterminer, étant donné un noeud origine r et un temps de départ t, les chemins de coûts minima vers tous les autres noeuds $i \neq r$ (*time-dependent least-cost path*).

La littérature concernant le calcul des plus courts chemins temporels comprend également plusieurs types de modèles se rapportant soit à la définition des temps de parcours (discret ou continu), soit à la possibilité d'avoir des temps d'attente aux noeuds (temps d'attente interdit, permis à tous les noeuds,...) ou soit au choix des temps de départ (par exemple, plus courts chemins temporels pour un temps de départ fixe ou pour tous les temps de départ).

Dans les modèles temporels, il peut être utile que certaines propriétés locales soient satisfaites. Une première propriété importante concerne les temps de parcours $d_{ij}(t)$. Un lien (i,j) est dit **FIFO** (*Fisrt-In-First-Out*) si «le plus tôt on quitte le noeud i en empruntant le lien (i,j), le plus tôt on arrive au noeud j». Plus formellement, un lien (i,j) possède la propriété *FIFO* si, pour $t_a \leq t_b$, nous avons que

$$t_a + d_{ij}(t_a) \leq t_b + d_{ij}(t_b), \ \forall t_a, t_b \in \mathcal{S}.$$

En général, on dira qu'un réseau est *FIFO* si tous les liens du réseau satisfont la propriété.

Lorsque la propriété *FIFO* n'est pas satisfaite, ce qui est généralement le cas dans les différents modèles en transport, il peut être préférable d'attendre un certain temps au noeud i avant d'emprunter le lien (i,j). Une propriété similaire peut alors être imposée aux coûts sur les liens. On dit qu'un lien (i,j) est **CC** (*Cost Consistent*) si «quitter le noeud i en empruntant le lien (i,j) plus tôt ne coûte pas plus cher que quitter plus tard». Étant donné que ce domaine de recherche est relativement récent, la définition formelle de cette propriété varie parfois d'un auteur à l'autre. Nous présentons ici celle suggérée par Pallotino et Scutellà [46]. Définissons $w_i(t)$ comme le coût d'attendre au noeud i à l'instant t et, pour $t_a < t_b$, $t_c = t_a + d_{ij}(t_a)$ et $t_d = t_b + d_{ij}(t_b)$. Alors les deux cas suivants sont considérés par Pallotino et Scutellà:

1) (i,j) est **FIFO** $(t_c \leq t_d)$: alors (i,j) est **CC** si, $\forall t_a < t_b$:

$$c_{ij}(t_a) + \sum_{k=c}^{d-1} w_j(t_k)(t_{k+1} - t_k) \leq c_{ij}(t_b).$$

2) (i,j) n'est pas **FIFO** $(t_c > t_d)$: alors (i,j) est **CC** si, $\forall t_a < t_b$:

$$c_{ij}(t_a) \leq c_{ij}(t_b) + \sum_{k=d}^{c-1} w_j(t_k)(t_{k+1} - t_k).$$

1.2.2 Algorithmes de recherche *origine-vers-noeuds* pour un temps de départ fixe

Nous considérons, dans cette partie, les approches développées pour calculer les plus courts chemins temporels selon une recherche *origine-vers-noeuds* pour un temps de départ fixe t_0.

Dreyfus [21] propose, de manière heuristique, de généraliser l'*algorithme de Dijkstra* [2] afin d'être en mesure de calculer les plus courts chemins temporels avec temps d'attente interdits pour tous les noeuds. Il est le premier à proposer cette généralisation qui sera reprise et prouvée plus tard par plusieurs auteurs ([35, 12, 46]). Dans son approche, on suppose que l'on quitte le noeud origine r au temps t_0 et on définit, pour chaque noeud i, une étiquette π_i qui représente une borne supérieure sur le plus court temps d'arrivée au noeud i. On tente alors de rendre ces étiquettes permanentes afin d'obtenir le temps de parcours minimal du noeud r vers chacun des autres noeuds i sachant que le temps de départ du noeud r est t_0. À une itération quelconque, on cherche à satisfaire la relation suivante:

$$\pi_j = \min_{i:(i,j)\in\mathcal{A}_j^-} \{\pi_j, \pi_i + d_{ij}(\pi_i)\}.$$

L'algorithme cherche en fait à résoudre les équations fonctionnelles suivantes:

$$\pi_j = \begin{cases} \min_{i:(i,j)\in\mathcal{A}_j^-} \{\pi_i + d_{ij}(\pi_i)\}, & \forall j \neq r; \\ t_0, & j = r. \end{cases}$$

Le résultat important venant de cette généralisation est qu'il est possible de résoudre le problème de plus courts chemins temporels avec la même complexité de calcul que l'*algorithme de Dijkstra*, c'est-à-dire $\mathcal{O}(n^2)$. La validité de la généralisation de l'*algorithme de Dijkstra* proposée par Dreyfus, est formellement prouvée par Kaufman et Smith [35]. Cependant, Kaufman et Smith démontrent également que cette dernière n'est valide que si tous les liens du réseau possèdent la propriété *FIFO*. En fait, sans cette condition, ils notent (ainsi que plusieurs autres auteurs) que cette méthode ne réussit pas à identifier les plus

[2] Voir section 1.1.4.

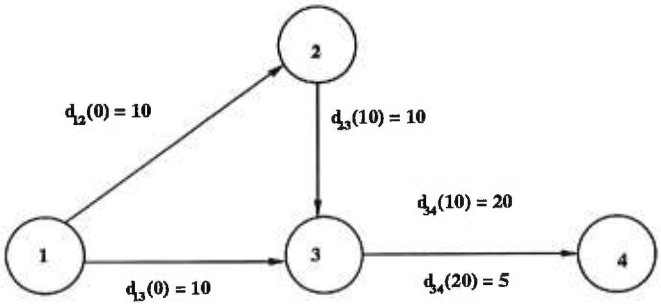

Figure 1.7: Violation de la condition FIFO.

courts chemins. Kaufman et Smith donnent, entre autre, un contre-exemple (figure 1.7) où la procédure de Dreyfus échoue. Le chemin optimal du noeud 1 au noeud 4 est $(1, 2, 3, 4)$ avec coût 25. Par contre, l'algorithme de Dreyfus nous donne plutôt, comme solution optimale, le chemin $(1, 3, 4)$ de coût 30.

Orda et Rom [42, 43] sont les premiers à soumettre une approche qui n'est pas restreinte par la condition *FIFO*. Cependant, il est important de noter que leur approche est conçue pour le modèle continu. Ils sont en fait les pionniers dans l'étude de ce modèle. Dans leurs travaux, ils étudient trois différentes politiques de temps d'attente aux noeuds du réseau:

- temps d'attente illimité pour tous les noeuds;
- temps d'attente interdit pour tous les noeuds;
- temps d'attente illimité au noeud origine seulement.

L'approche qu'ils proposent est une modification de la méthode suggérée par Dreyfus permettant de vérifier si un temps d'attente peut être bénéfique aux noeuds visités. Étant donné un noeud origine r et un temps de départ t_0, on cherche à déterminer les plus courts chemins de r vers tous les autres noeuds. Supposons que l'on atteint le noeud i au temps t, que l'on attend pendant un certain temps Δt, et que l'on quitte ensuite le noeud i en direction du noeud j. Le temps d'arrivée au noeud j sera alors $t + [\Delta t + d_{ij}(t + \Delta t)]$. Soit $\tilde{d}_{ij}(t, \Delta t) \equiv \Delta t + d_{ij}(t + \Delta t)$, qui combine le temps d'attente et le délai prévu pour traverser le lien (i, j). Alors, pour minimiser le temps de parcours du lien (i, j), il faut déterminer le temps d'attente optimal Δt^* tel que $\tilde{d}_{ij}(t, \Delta t^*) \leq \tilde{d}_{ij}(t, \Delta t)$, $\forall \Delta t \geq 0$. À l'aide de cette approche, il est donc possible de déterminer les temps d'attente optimaux pour chaque noeud i ou encore le temps d'attente optimal au noeud origine lorsque les pauses sont interdites

pour chacun des autres noeuds du réseau. L'algorithme proposé est identique à un algorithme utilisant une méthode d'extensions sélectives. La seule opération qui diffère est celle qui consiste à modifier l'étiquette non permanente d'un noeud. Cette opération est, dans ce cas, fonction de la valeur $\tilde{d}_{ij}(t)$. Nous avons que

$$\pi_j = \min_{i:(i,j)\in\mathcal{A}_j^-} \{\pi_j, \pi_i + \tilde{d}_{ij}(\pi_i)\}$$

où π_i représente bien sûr le temps de parcours minimal du noeud origine r au noeud i. La complexité de l'approche de Orda et Rom est par conséquent $\mathcal{O}(n^2)$, comme pour la méthode proposée par Dreyfus. Cependant, la méthode proposée ne permet pas de trouver efficacement les chemins lorsque les temps d'attente sont interdits dans tout le réseau.

Orda et Room [43] sont également les premiers à introduire le problème de recherche des chemins de coûts minima dans un réseau temporel (*time-dependent least-cost path*) et à proposer un algorithme pour ce problème. Leur approche calcule les chemins de coûts minima entre un noeud origine r et tous les autres noeuds avec temps de départ t_0 au noeud origine. Ils proposent un algorithme qui permet de calculer les chemins lorsque les temps d'attente sont permis aux noeuds. Ils montrent également que lorsque les temps d'attente sont interdits à chaque noeud du réseau, le problème de recherche des chemins sans circuits est *NP-difficile*. Aucune implantation de leur algorithme n'a été présentée et, par conséquent, il n'y a pas de résultats qui permettent d'analyser les performances de leur approche.

Drissi [22] montre également qu'il est possible d'utiliser l'*algorithme de Dijkstra* dans un réseau qui n'est pas nécessairement *FIFO*. Soit

$$\tilde{d}_{ij}(\pi_i) = \min\{d_{ij}(t) + t - \pi_i : t \geq \pi_i\}\ \forall (i,j) \in \mathcal{A},\ \forall \pi_i \in \mathcal{S}.$$

Dans cette expression, $d_{ij}(t) + t - \pi_i$ pour $t \geq \pi_i$ représente le temps total requis pour traverser le lien (i,j) lorsque le temps d'arrivée au noeud i est π_i et que l'entrée effective sur le lien (i,j) est t. $\tilde{d}_{ij}(\pi_i)$ représente donc le meilleur temps de parcours du lien (i,j), quand le temps d'arrivée au noeud i est π_i. Dans l'algorithme, l'étiquette π_i qui devient permanente représente le temps de parcours minimum du noeud origine r au noeud i. En supposant que S est un ensemble discret constitué de $\mathcal{M}$ intervalles de temps, alors la complexité de cet algorithme est $\mathcal{O}((n+m)\log n + m\mathcal{M})$.

Dernièrement, les travaux traitant du calcul des plus courts chemins temporels ont donné droit à de nouveaux résultats et à de nouveaux algorithmes. Chabini [12] propose

entre autre quelques nouveaux résultats basés sur une formulation légèrement différente du problème étudié initialement par Dreyfus. Il fait d'abord l'hypothèse que $d_{ij}(t)$ prend des valeurs entières positives et que $S = \{0, 1, 2, ..., \mathcal{M} - 1\}$. Alors, l'idée principale à la base du développement de ses résultats consiste à écrire les conditions d'optimalité de façon à ne considérer, pour un noeud j, que les chemins qui visitent le noeud précédent i à un temps plus grand ou égal à π_i.

Proposition 1 (Chabini): *Si la condition* **FIFO** *est satisfaite, alors l'équation fonctionnelle (1.1) représentant les temps de parcours minima est équivalente à l'équation fonctionnelle (1.2):*

$$\pi_j = \begin{cases} \min_{i:(i,j)\in\mathcal{A}_j^-} \min_{t\geq\pi_i}\{t + d_{ij}(t)\}, \ \forall j \neq r; \\ 0, \ j = r. \end{cases} \tag{1.1}$$

$$\pi_j = \begin{cases} \min_{i:(i,j)\in\mathcal{A}_j^-}\{\pi_i + d_{ij}(\pi_i)\}, \ \forall j \neq r; \\ 0, \ j = r. \end{cases} \tag{1.2}$$

Cette proposition montre que tout algorithme de calcul des plus courts chemins statiques peut être généralisé, sans coût supplémentaire, au cas temporel. Chabini résume ce résultat dans la proposition suivante:

Proposition 2 (Chabini): *Si la condition* **FIFO** *est satisfaite, alors le calcul des plus courts chemins temporels est algorithmiquement équivalent au calcul des plus courts chemins statiques.*

La généralisation de l'*algorithme de Dijkstra* proposée par Dreyfus est en fait un cas spécial du résultat de la **proposition 2**. Chabini ajoute également une autre restriction à laquelle sont soumises ces généralisations. La proposition suivante énonce explicitement cette restriction qui, apparamment, ne se retrouve pas dans la littérature:

Proposition 3 (Chabini): *Si la condition* **FIFO** *est satisfaite, alors tout algorithme utilisant une technique d'extensions sélectives avec recherche des chemins à partir d'une origine vers tous les autres noeuds et basé sur les équations fonctionnelles (1.2) de la* **proposition 1** *calcule les plus courts chemins temporels avec une complexité de calcul identique au cas statique.*

Dans le cas où la condition *FIFO* n'est pas satisfaite, Chabini mentionne qu'il est possible de concevoir un algorithme qui ne travaille pas directement sur le réseau *espace-*

temps, c'est-à-dire qui n'utilise pas explicitement l'expansion temporelle du réseau initial. Le cas où les temps d'attente sont permis est également étudié et il montre que les **propositions 1**, **2** et **3** restent valides sans la condition *FIFO*. De plus, il note que cette variante du problème est un cas particulier du problème avec des temps d'attente interdits pour tous les noeuds.

Indépendamment à Chabini, Pallotino et Scutellà [46] proposent également de calculer les plus courts chemins temporels en utilisant implicitement l'information contenue dans le réseau *espace-temps*. Étant donné un réseau $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ temporel et un ensemble discret $\mathcal{S} = \{t_0, t_1, t_2, ..., t_{\mathcal{M}}\}$, alors le réseau *espace-temps* $\mathcal{R} = (\mathcal{V}, \mathcal{E})$ est défini de la façon suivante:

$$\begin{aligned} \mathcal{V} &= \{i_h : i \in \mathcal{N}, 1 \leq h \leq \mathcal{M}\}; \\ \mathcal{E} &= \{(i_h, i_k) : (i,j) \in \mathcal{A}, t_h + d_{ij}(t_h) = t_k, 1 \leq h < k \leq \mathcal{M}\}. \end{aligned}$$

$\mathcal{R}$ est un réseau avec $|\mathcal{V}| = n\mathcal{M}$ noeuds et $|\mathcal{E}| = (m+n)\mathcal{M}$ liens et il ne contient pas de circuits. En particulier, chaque visite *chronologique* des noeuds de $\mathcal{R}$, c'est-à-dire dans laquelle les noeuds sont visités selon un ordre non-décroissant de leurs indices de temps, procure une visite *topologique* de $\mathcal{R}$. Considérons par exemple le réseau de la figure 1.8 où les temps de parcours pour les liens $(1,2)$, $(1,3)$, $(2,3)$ et $(3,2)$ sont 1 pour toutes les périodes de temps. Alors le réseau *espace-temps* $\mathcal{R}$ associé à ce dernier est illustré dans la figure 1.9.

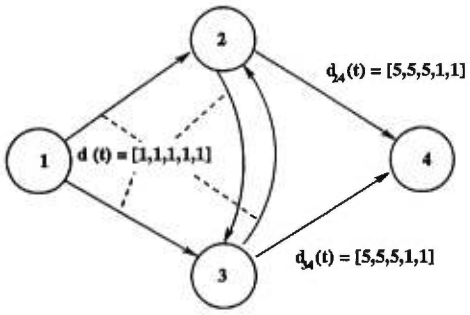

Figure 1.8: Réseau temporel $\mathcal{G}$.

Comme Chabini, Pallotino et Scutellà notent qu'il est possible de calculer les plus courts chemins temporels en travaillant implicitement sur $\mathcal{R}$ au moyen d'une visite topologique de ce réseau acyclique. L'approche qu'il propose, et qu'il nomme **Chrono-SPT** effectue une sélection des noeuds de $\mathcal{R}$ selon l'ordre *chronologique* (on considère en premier lieu les noeuds correspondant au temps t_0, ensuite ceux correspondant au temps t_1 et ainsi de

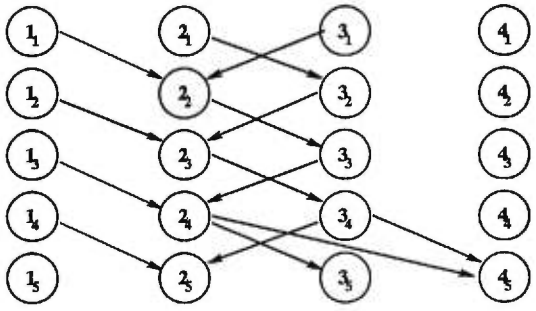

Figure 1.9: Réseau espace-temps $\mathcal{R}$ associé à $\mathcal{G}$.

suite). L'opération de sélection est effectuée au moyen d'une *liste de compartiments*[3] $L = \{L_0, L_1, L_2, ..., L_{\mathcal{M}}\}$ où L_h dénote le compartiment contenant les noeuds à visiter au temps t_h. L'algorithme proposé permet de calculer les chemins temporels de coûts minima avec ou sans temps d'attente pour chaque noeud i du réseau. Soit $\pi_i(t)$ l'étiquette représentant le coût du chemin de l'origine au noeud i_t. Alors, l'algorithme est:

1. Initialisation

$L_0 = \{r\}$; $L_h = \emptyset, 0 < h \leq \mathcal{M}$;

$\pi_i(t) = \infty$, $\forall t, i \neq r$;

$\pi_q(t) = 0$, $\forall t$;

2. Itération principale

sélectionner i de L_h et $L_h = L_h - \{i\}$

pour chaque $j : (i,j) \in \mathcal{A}_i^+$, faire

$t_k = t_h + d_{ij}(t_h)$

si $\pi_i(t_h) + c_{ij}(t_h) < \pi_j(t_k)$ alors

$\pi_j(t_k) = \pi_i(t_h) + c_{ij}(t_h)$ et $b_j(t_k) = i_h$

si $j \notin L_k$, alors $L_k = L_k + \{j\}$

3. Seulement s'il est permis d'attendre au noeud i

si $\pi_i(t_h) + w_i(t_h)(t_{h+1} - t_h) < \pi_i(t_{h+1})$ alors

$\pi_i(t_{h+1}) = \pi_i(t_h) + w_i(t_h)(t_{h+1} - t_h)$ et $b_i(t_{h+1}) = i_h$

si $i \notin L_{h+1}$, alors $L_{h+1} = L_{h+1} + \{i\}$

Remarque: Dans le réseau *espace-temps*, les temps d'attente $w_i(t_h)$ sont représentés par des liens (i_h, i_{h+1}).

L'algorithme présenté précédemment se modifie facilement pour le cas du calcul des chemins minimisant les temps de parcours. En effet, il suffit dans ce cas d'ignorer les coûts

[3]Voir section 1.1.4.

$c_{ij}(t)$ sur les liens et de ne considérer que les temps de parcours $d_{ij}(t)$. L'approche de Pallotino parcours implicitement la portion de $\mathcal{R}$ comprenant les chemins dont les temps d'arrivée sont compris dans l'intervalle considéré. La complexité de cet algorithme est $\Theta(\mathcal{M} + |\mathcal{E}^*|)$, où $\mathcal{E}^* \subset \mathcal{E}$ est le sous-ensemble des liens de $\mathcal{R}$ qui sont implicitement parcourus par l'algorithme. Puisque $|\mathcal{E}^*| \leq m\mathcal{M}$, la complexité est $\mathcal{O}(m\mathcal{M})$ en pire cas.

Les approches vues dans cette partie permettent de calculer les plus courts chemins temporels pour une origine r vers tous les autres noeuds et ce, pour un temps de départ fixe. Toutefois, les modèles de *planification des transports* qui intéressent présentement les chercheurs nécessitent la plupart du temps de connaître les plus courts chemins à partir de tous les noeuds vers une destination donnée et ce, pour tous les temps de départ possibles. Bien sûr, il est possible d'utiliser les algorithmes déjà vus (il suffit d'appliquer l'algorithme de son choix pour chaque noeud et pour chaque temps de départ possible) mais à un coût qui n'est clairement pas minimal.

1.2.3 Algorithmes de recherche *noeuds-vers-destination* pour tous les temps de départ

Cooke et Halsey [16] sont les premiers à développer un algorithme calculant les plus courts chemins temporels de chaque noeud vers une destination donnée. En se basant sur le principe d'optimalité développé par Bellman [5], ils proposent une fonction itérative qui donne les chemins minimisant les temps de parcours à partir de chaque noeud i vers la destination q et ce, pour chaque période de temps t. Ils font l'hypothèse que $S = \{t_0, t_0+1, t_0+2, ..., t_0+\mathcal{M}\}$ est un ensemble discret d'intervalles de temps et que les temps de parcours $d_{ij}(t)$ sont entiers pour $t \in S$. L'entier $\mathcal{M}$ est choisi de sorte que les temps de parcours soient définis pour tout $t \in \mathcal{S}$. Les temps de parcours pour $t > t_0 + \mathcal{M}$ prennent la valeur infinie, ce qui a pour effet d'éliminer les chemins dont le temps d'arrivée excède $t_0 + \mathcal{M}$. Soit $\pi_i(t)$, le temps de parcours du chemin reliant le noeud i à la destination q où le temps de départ du noeud i est t. Alors, le principe d'optimalité leur permet d'établir que, pour $t \in \mathcal{S}$,

$$\pi_i(t) = \begin{cases} \min_{j:(i,j)\in\mathcal{A}_i^+}\{d_{ij}(t) + \pi_j(t + d_{ij}(t))\}, & \forall i \neq q,\ t \in \mathcal{S}; \\ 0, & i = q,\ t \in \mathcal{S}. \end{cases} \tag{1.3}$$

L'algorithme qu'ils proposent mantient, à chaque itération k, un ensemble $\Pi_i^{(k)}(t)$ de tous les chemins utilisant au plus k liens et qui peuvent rejoindre la destination q avant le temps $t_0 + \mathcal{M}$ lorsque le temps de départ du noeud i est t. Au début de l'algorithme, les temps de

parcours $d_{ij}(t)$ sont modifiés de la façon suivante:

$$\tilde{d}_{ij}(t) = \begin{cases} d_{ij}(t), & si\ t + d_{ij}(t) \leq t_0 + \mathcal{M}; \\ \infty, & si\ t + d_{ij}(t) > t_0 + \mathcal{M}. \end{cases}$$

On pose alors, comme conditions de départ,

$$\pi_q^{(0)}(t) = 0,\ et\ \pi_i^{(0)}(t) = \tilde{d}_{iq}(t), \forall i \neq d,$$

où $\pi_i^{(0)}(t)$ représente le temps de parcours minimal reliant le noeud i et la destination q et constitué de seulement un lien. De manière générale, on a que $\pi_q^{(k)}(t) = 0$, et pour $i \neq q$ et $k = 1, 2, 3, ..., n-2$, on a que

$$\pi_i^{(k)}(t) = \begin{cases} \textit{temps minimum pour un chemin dans } \Pi_i^{(k)}(t), & si\ \Pi_i^{(k)}(t) \neq \emptyset \\ \infty, & si\ \Pi_i^{(k)}(t) = \emptyset \end{cases}.$$

Cooke et Halsey montrent que, par le principe d'optimalité, la forme précédente est équivalente à la fonction itérative suivante:

$$\pi_i^{(k)}(t) = \begin{cases} \min_{j:(i,j)\in\mathcal{A}_i^+}\{\tilde{d}_{ij}(t) + \pi_j^{(k-1)}(t + \tilde{d}_{ij}(t))\}, & \forall i \neq q,\ t \in \mathcal{S}; \\ 0, & i = q,\ t \in \mathcal{S}. \end{cases}$$

Cette méthode a une complexité de $\mathcal{O}(n^3\mathcal{M}^2)$ en pire cas. Cependant, la littérature ne contient aucune implantation de cet algorithme.

Ziliaskopoulos et Mahmassani [57] ont également proposé une méthode s'attaquant à cette variante du problème. Il est important de mentionner que leur algorithme peut calculer les plus courts chemins sans nécessairement permettre les temps d'attente aux noeuds, ce que Orda et Room [42, 43] n'avaient pas réussi à faire antérieurement. Ils considèrent $\mathcal{S} = \{t_0, t_0+\delta, t_0+2\delta, ..., t_0+(\mathcal{M}-1)\delta\}$ comme un ensemble discret d'intervalles de temps où δ est un petit intervalle de temps et $\mathcal{M}$ est un entier assez grand pour que $[t_0, t_0+(\mathcal{M}-1)\delta]$ soit la période d'intérêt (heure de pointe, par exemple). Les temps de parcours $d_{ij}(t)$ prennent des valeurs réelles non-négatives et ils sont définis sur $\mathcal{S} = \{t_0, t_0+\delta, t_0+2\delta, ..., t_0+(\mathcal{M}-1)\delta\}$. Ils supposent que $d_{ij}(\tilde{t}) = d_{ij}(t_0 + k\delta)$ pour $\tilde{t}$ dans l'intervalle $t_0 + k\delta < \tilde{t} < t_0 + (k+1)\delta$. De plus, ils supposent également que $d_{ij}(t)$, pour $t > t_0 + (\mathcal{M}-1)\delta$, est constant et égal à $d_{ij}(t_0+(\mathcal{M}-1)\delta)$. Ils justifient cette hypothèse par le fait que l'on peut supposer des temps de voyage constants après l'heure de pointe. On définit $\pi_i(t)$ le temps de parcours minimal du chemin reliant le noeud i à la destination q où le temps de départ du noeud i est t. Enfin,

définissons $\Pi_i = [\pi_i(t_0), \pi_i(t_0+\delta), ..., \pi_i(t_0+(\mathcal{M}-1)\delta)]$ comme étant un vecteur associé au noeud i contenant chacune des étiquettes $\pi_i(t)$ pour chaque période de temps $t \in \mathcal{S}$.

À la base de l'algorithme, on retrouve l'équation d'optimalité suivante qui est une extension de la fonction itérative proposée par Cooke et Halsey [16]:

$$\pi_i(t) = \begin{cases} \min_{j:(i,j)\in\mathcal{A}_i^+}\{d_{ij}(t) + \pi_j(t+d_{ij}(t))\}, & \forall i \neq q,\ t \in \mathcal{S}; \\ 0, & i = q,\ t \in \mathcal{S}. \end{cases}$$

Plutôt que de visiter tous les noeuds à chaque itération, une liste des noeuds ayant le potentiel d'améliorer au moins une étiquette d'un autre noeud est maintenue. Alors, à partir de la destination, l'algorithme tente de résoudre récursivement l'équation d'optimalité en visitant tous les noeuds de la liste. Plus précisément, lorsqu'un noeud i de la liste est visité, les étiquettes $\pi_j(t)$ $(t \in \mathcal{S})$ de tous les noeuds j tels que $(i,j) \in \mathcal{A}$ sont, s'il y a lieu, mises à jour. Si au moins un des noeuds voit une de ses étiquettes modifiées, alors il est inséré dans la liste. Cet algorithme procède donc par *ajustements progressifs*. Supposons que $\mathcal{Q}$ soit la liste des noeuds à visiter. Alors, initialement, la liste $\mathcal{Q}$ contient seulement la destination q. La première itération consiste à mettre à jour les étiquettes des noeuds pouvant rejoindre directement la destination q et de les insérer dans la liste $\mathcal{Q}$:

$$\pi_i(t) = d_{iq}(t),\ \forall i : (i,q) \in \mathcal{A}_q^-,\ \forall t \in \mathcal{S}.$$

Ensuite, pour chaque noeud $j \in \mathcal{Q}$, on exécute l'itération suivante:

$$\pi_i(t) = \min\{\pi_i(t), d_{ij}(t) + \pi_j(t+d_{ij}(t))\},\ \forall i : (i,j) \in \mathcal{A}_j^-,\ \forall t \in \mathcal{S}.$$

Si au moins une des composantes de Π_i est modifiée, alors le noeud i est inséré dans $\mathcal{Q}$. Cette dernière étape est répétée jusqu'à ce que $\mathcal{Q} = \emptyset$. À la fin de l'algorithme, les vecteurs Π_i de chaque noeud i contiennent les temps de parcours minima, pour chaque période de temps $t \in \mathcal{S}$, des plus courts chemins les reliant à la destination. Les étapes de l'algorithme sont les suivantes:

1. **Initialisation**

   $\Pi_q = (0, 0, 0, ..., 0)$

   $\Pi_i = (+\infty, +\infty, +\infty, ..., +\infty)$

2. **Étape principale**

   si $\mathcal{Q} = \emptyset$, alors aller à 4.

   sélectionner le premier noeud j de la queue $\mathcal{Q}$

pour chaque noeud $i : (i,j) \in \mathcal{A}_j^-$

pour chaque période de temps $t \in \mathcal{S}$

si $\pi_i(t) > d_{ij}(t) + \pi_j(t + d_{ij}(t))$, alors remplacer $\pi_i(t)$ par la nouvelle valeur $\pi_j(t + d_{ij}(t))$.

si au moins une des étiquettes $\pi_i(t)$ a été modifiée, alors le noeud i est inséré dans la queue $\mathcal{Q}$.

3. Répéter l'étape 2.

4. L'algorithme est terminé.

En supposant que la liste $\mathcal{Q}$ soit implantée sous forme de queue simple, alors la complexité de l'algorithme est dans l'ordre de $\mathcal{O}(n^3\mathcal{M}^2)$. Initialement, la queue $\mathcal{Q}$ contient un seul élément, la destination q. Pendant la première itération, au plus $(n-1)$ noeuds sont insérés dans $\mathcal{Q}$. En exécutant $(n-1)$ répétitions de l'étape (2), un des noeuds verra une de ses étiquettes devenir permanentes. Nous avons que chacun des $(n-1)$ noeuds peut entrer au plus $\mathcal{M}$ fois dans la queue $\mathcal{Q}$, puisque chaque noeud possède $\mathcal{M}$ étiquettes. Ce qui fait que la procédure qui exécute $(n-1)$ répétitions de l'étape (2) sera exécutée au plus $\mathcal{M}(n-1)$ fois. Commme l'étape (2) nécessite clairement $\mathcal{O}(\mathcal{M}(n-1))$ opérations, la complexité totale de l'algorithme est $\mathcal{O}(n^3\mathcal{M}^2)$ en pire cas.

Ziliaskopoulos et Mahmassani proposent également un algorithme permettant de résoudre le problème de recherche des chemins de coûts minima dans un réseau temporel (*time-dependent least-cost path*). On définit $\pi_i(t)$ comme étant le coût minimal pour atteindre la destination q à partir du noeud i au temps t et $\Pi_i = [\pi_i(t_0), \pi_i(t_0+\delta), ..., \pi_i(t_0 + (\mathcal{M}-1)\delta)]$ comme étant un vecteur associé au noeud i contenant chacune des étiquettes $\pi_i(t)$ pour chaque période de temps $t \in \mathcal{S}$. Les valeurs de $\pi_i(t)$ sont alors déterminées par l'équation d'optimalité suivante:

$$\begin{aligned} \pi_i(t) &= \min_{j:(i,j)\in\mathcal{A}_i^+} \{c_{ij}(t) + \pi_j(t + d_{ij}(t))\},\ \forall t \in \mathcal{S},\ \forall i \in \mathcal{N}\backslash q; \\ \pi_D(t) &= 0,\ \forall t \in \mathcal{S}. \end{aligned}$$

Cet algorithme utilise également une technique d'*ajustements progressifs* en maintenant une liste des noeuds qui ont le potentiel d'améliorer une des étiquettes d'au moins un des autres noeuds. Les étapes de l'algorithmes sont:

1. Initialisation

$\Pi_q = (0, 0, 0, ..., 0)$

$\Pi_i = (+\infty, +\infty, +\infty, ..., +\infty)$

2. Étape principale

si $\mathcal{Q} = \emptyset$, alors aller à 4.

sélectionner le premier noeud j de la queue $\mathcal{Q}$

pour chaque noeud $i : (i, j) \in \mathcal{A}_j^-$

pour chaque période de temps $t \in \mathcal{S}$

si $\pi_i(t) > c_{ij}(t) + \pi_j(t + d_{ij}(t))$, alors remplacer $\pi_i(t)$ par la nouvelle valeur $c_{ij}(t) + \pi_j(t + d_{ij}(t))$.

si au moins une des étiquettes $\pi_i(t)$ a été modifiée, alors le noeud i est inséré dans la queue $\mathcal{Q}$.

3. Répéter l'étape 2.

4. L'algorithme est terminé.

On peut voir que cet algorithme est une généralisation de l'algorithme présenté précédemment. En effet, si $c_{ij}(t) = d_{ij}(t)$ pour tous les liens du réseau, alors cet algorithme est équivalent à celui permettant de calculer les chemins minimisants les temps de parcours (*time-dependent least-time path*).

Pallotino et Scutellà [46] proposent d'utiliser une modification de son algorithme calculant les plus courts chemins temporels pour une origine, vers tous les noeuds étant donné un temps de départ fixe au noeud origine. Son approche est basée sur une visite *chronologique* inverse du réseau *espace-temps*. Cette opération peut être réalisée en effectuant une visite inversée de la structure de *compartiments* L. Cette idée utilise en fait la propriété acyclique du réseau *espace-temps*. Une fois son algorithme modifié, nous obtenons une approche permettant de calculer les chemins minimisant les temps de parcours pour chaque période de temps et à partir de chaque noeud vers la destination q:

1. Initialisation

$L_h = \{q\}$; $0 \leq h \leq \mathcal{M}$;

$\pi_i(t) = \infty$, $\forall t, i \neq q$; $\pi_q(t) = 0$, $\forall t$;

2. Itération principale

sélectionner j de L_h et $L_h = L_h - \{j\}$

pour chaque $i : (i, j) \in \mathcal{A}_j^-$, faire

$t_k = t_h + d_{ij}(t_h)$

si $\pi_i(t_h) + c_{ij}(t_h) < \pi_j(t_k)$ alors

$\pi_j(t_k) = \pi_i(t_h) + c_{ij}(t_h)$ et $b_j(t_k) = i_h$

si $j \notin L_k$, alors $L_k = L_k + \{j\}$

La complexité de calcul de cet algoritme est $\Theta(\mathcal{M} + |\mathcal{E}^*|)$ et $\mathcal{O}(m\mathcal{M})$ en pire cas.

Une autre approche utilisant la propriété acyclique du réseau *espace-temps* est également développée par Chabini [12]. L'algorithme qu'il propose est basé sur le fait que les étiquettes peuvent être mises à jour selon un ordre décroissant des intervalles de temps. Il justifie ce raisonnement en notant que puisque les temps de parcours sont des entiers positifs, les étiquettes correspondant au temps t ne peuvent mettre à jour celles correspondant au temps plus grand que t. On peut vérifier cette affirmation en observant la forme fonctionnelle donnée par l'équation (1.3).

Dans son algorithme, Chabini suppose que pour les temps de départ plus grands ou égaux à la dernière période de temps, soit $\mathcal{M}-1$, le calcul des plus courts chemins temporels est équivalent au cas statique. Cette hypothèse est tout à fait raisonnnable puisque l'on peut supposer que les temps de parcours sont constants après un certain intervalle de temps. Alors, en supposant qu'un algorithme de calcul des plus courts chemins statiques avec un temps d'exécution optimal (SSP) soit disponible, l'algorithme proposé est le suivant:

1. Initialisation

$\pi_i(t) = \infty$, $\forall t < \mathcal{M} - 1, i \neq q$;

$\pi_q(t) = 0$, $\forall t < \mathcal{M} - 1$;

$\pi_i(\mathcal{M} - 1) = SSP(d_{ij}(\mathcal{M}), q)$;

note: $\pi_i(t) = \pi_i(\mathcal{M} - 1)$, $\forall t \geq \mathcal{M} - 1, i \neq q$;

2. Itération principale

pour $t = \mathcal{M} - 2$ jusqu'à 0 faire

pour tous les arcs $(i, j) \in \mathcal{A}$ faire

si $\pi_i(t) > d_{ij}(t) + \pi_j(t + d_{ij}(t))$, alors $\pi_q(t) = d_{ij}(t) + \pi_j(t + d_{ij}(t))$

La complexité de cet algorithme s'évalue facilement. L'*étape 1* nécessite $n\mathcal{M}$ opérations pour initialiser $\pi_i(t)$, $\forall t < \mathcal{M} - 1, i \neq q$ et exige le calcul des plus courts chemins statiques (se fait en un temps SSP) pour la période de temps $\mathcal{M}$. De son côté, l'*étape 2* prend clairement un temps dans $\mathcal{O}(m\mathcal{M})$. Donc, globalement, nous obtenons une complexité de calcul qui est $\Omega(SSP + n\mathcal{M} + m\mathcal{M})$.

Nous attirons maintenant votre attention sur la variante du problème avec temps d'attente interdits pour tous les noeuds dans un réseau qui n'est pas *FIFO*. Cette variante est en fait une des plus réalistes dans le contexte d'un réseau de transport. Le point que nous voulons faire ressortir est que les trois algorithmes développés respectivement par Ziliaskopoulos et Mahmassani, Pallotino et Chabini permettent de calculer les plus courts chemins dans ces conditions. Cependant, ces chemins peuvent contenir des boucles, comme nous pouvons le voir dans l'exemple de la figure 1.8. En effet, en appliquant un de ces trois

algorithmes sur cet exemple on obtient, pour le noeud 1 avec temps de départ $t = 1$, le chemin 1-2-3-2-4 avec temps d'arrivée 5. En fait, lorsque les temps d'attente sont interdits, il est souvent nécessaire d'utiliser soit un chemin alternatif, soit une boucle, afin d'atteindre le noeud destination dans un temps au mieux égal à celui qu'on obtiendrait si les temps d'attente étaient permis. Ziliaskopoulos et Mahmassani [57] avaient déjà remarqué ce fait en démontrant que le principe d'optimalité de Bellman est satisfait dans le réseau *espace-temps*, même si le réseau n'est pas **FIFO**.

1.3 Algorithmes de calcul des plus courts chemins statiques avec délais et interdictions aux intersections

La majorité des algorithmes de calcul des plus courts chemins suggérés dans la littérature supposent qu'il n'existe pas de délais ou d'interdictions associés aux mouvements aux intersections. Cette hypothèse, qui peut être acceptable dans plusieurs types d'applications, devient souvent trop restrictive dans certains autres. Par exemple, si l'on désire calculer les plus courts chemins dans un réseau urbain, alors les délais aux intersections peuvent prendre beaucoup d'importance, particulièrement lorsque le réseau est congestionné.

Dans cette section, on présente l'évolution des recherches effectuées sur ce problème en discutant des différentes approches proposées pour le résoudre. Ces approches peuvent être classées en fonction de deux stratégies de résolution. La première stratégie vise à effectuer une modélisation explicite des virages aux intersections et, par conséquent, implique une modification du réseau original. Par contre, la deuxième stratégie cherche à résoudre le problème par un traitement implicite des virages aux intersections et, donc, utilise le réseau original pour effectuer la recherche des plus courts chemins.

Pour les besoins de cette section, nous définissons $\bar{\mathcal{N}} \subseteq \mathcal{N}$, un sous-ensemble de noeuds pour lesquels il existe des mouvements pénalisés ou interdits et, d_{ijk} (ou $d_{a_1 a_2}$), le délai associée au mouvement de $a_1 = (i, j)$ vers $a_2 = (j, k)$.

1.3.1 Modélisation explicite des mouvements aux intersections

Une première approche permettant de calculer les plus courts chemins dans un réseau avec délais sur les mouvements aux intersections fût proposée par Caldwell [9] en 1961. L'énoncé fondamental à la base du raisonnement de Caldwell est le suivant: "*Il n'est pas nécessairement vrai que le meilleur chemin de l'origine à un noeud i passant par un noeud j, coïncide, à partir de l'origine jusqu'à j, avec le meilleur chemin de l'origine au noeud*

j." On peut facilement vérifier cet énoncé dans l'exemple de la figure 1.10. En effet, le

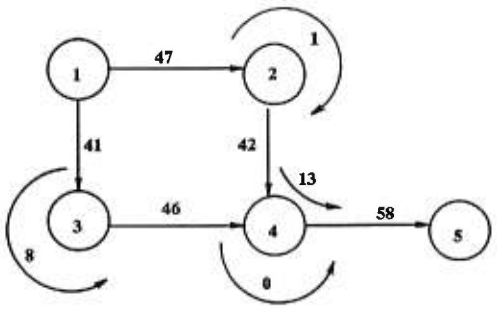

Figure 1.10: Réseau original.

meilleur chemin du noeud 1 au noeud 5 est celui passant par les noeuds 1-3-4-5. Cependant, le meilleur chemin du noeud 1 au noeud 4 est plutôt celui transitant par les noeuds 1-2-4.

Afin de pouvoir réutiliser les algorithmes déjà existants, l'approche de Caldwell consiste à modifier le réseau en redéfinissant les coûts de façon à ce qu'ils tiennent compte des délais associés aux mouvements aux intersections. Soit C_{ik} le coût du mouvement nous faisant transiter du noeud i au noeud k en passant par les arcs (i,j) et (j,k). Alors, nous avons que $C_{ik} = d_{ij} + d_{ijk} + d_{jk}$. Maintenant, considérons le chemin $r = i_1, i_2, i_3, ..., i_k$. Le coût de ce chemin est:

$$\begin{aligned} C_{ri_k} &= d_{ri_2} + d_{ri_2i_3} + d_{i_2i_3} + ... + d_{i_{k-1}i_k} \\ &= d_{ri_2} + (d_{ri_2i_3} + d_{i_2i_3}) + ... + (d_{i_{k-2}i_{k-1}i_k} + d_{i_{k-1}i_k}) \\ &= d_{ri_2} + \tilde{d}_{ri_2i_3} + ... + \tilde{d}_{i_{k-2}i_{k-1}i_k} \end{aligned}$$

où $\tilde{d}_{ijk} = d_{ijk} + d_{jk}$. Si l'on pose que $d_{i_0ri_2} = 0$, alors on obtient que

$$\begin{aligned} \tilde{d}_{i_0ri_2} &= d_{ri_2}; \\ C_{ri_k} &= \textstyle\sum_{n=0}^{k-2} \tilde{d}_{i_ni_{n+1}i_{n+2}}. \end{aligned}$$

Les nouveaux coûts $\tilde{d}_{ijk}$ permettent de calculer les plus courts chemins en utilisant un algorithme conventionnel et alors, le coût total C_{ri_k} inclura tous les délais sur les mouvements aux intersections. On remarque que, dans cette approche, le réseau est implicitement transformé. En effet, à chaque arc (i,j) du réseau original correspond un noeud (i,j) dans le réseau implicite. Les arcs du réseau implicite sont ensuite modélisés par chacun des mouvements de (i,j) vers (j,k) présents dans le réseau original. La figure 1.11 nous donne un exemple de cette transformation appliquée sur le réseau de la figure 1.10. Bien que cette

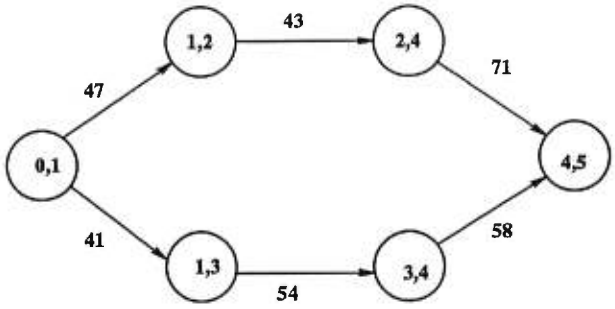

Figure 1.11: Réseau implicite.

approche ne considère pas explicitement les interdictions de virage, on peut représenter ces mouvements en affectant un coût infini à la variable d_{ijk} correspondante.

La figure 1.12 nous montre le résultat de la modélisation explicite proposée par Caldwell sur une intersection urbaine typique. On remarque que l'expansion de l'intersection

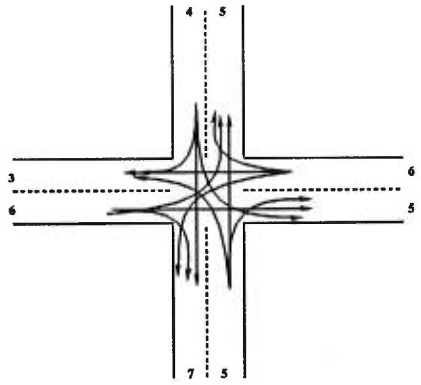

Figure 1.12: Modélisation des mouvements à une intersection.

peut devenir facilement assez complexe. Sur de grands réseaux, l'utilisation de cette stratégie peut être coûteuse en espace mémoire et en temps de calcul en raison de la dimension que peut prendre le réseau implicite. Caldwell n'a pas formulé d'algorithme pouvant être utilisé directement pour le calcul des plus courts chemins dans des réseaux avec pénalités de virages. Par conséquent, aucun résultat numérique n'est présenté dans son article.

L'idée de modéliser explicitement les mouvements aux intersections dans le but d'utiliser un algorithme conventionnel de calcul des plus courts chemins fût aussi adoptée par Yagar [54]. En fait, l'idée de Yagar permet d'éliminer correctement les mouvements inter-

dits. Toutefois, Easa [23] montre que l'algorithme proposé introduit d'autres mouvements habituellement non permis (virages en U) dans les réseaux urbains.

1.3.2 Traitement implicite des mouvements aux intersections

Kirby et Potts [37] sont les premiers à proposer une approche travaillant sur la structure originale du réseau. Dans cette approche, nous posons $d_{a_1 a_2} = \infty$ si le mouvement de l'arc $a_1 = (i, j)$ vers l'arc $a_2 = (j, k)$ est interdit et, pour $d_{a_1 a_2}$ prenant une valeur finie, on doit nécessairement avoir que $i \neq k$. Un chemin admissible $\mathcal{P}$ est une séquence d'arcs $a_1, a_2, a_3, ..., a_k$ distincts tels que $d_{a_n a_{n+1}} < \infty$, $n = 1, 2, ..., k-1$. Le coût $C_{a_1 a_k}$ d'un chemin admissible de l'arc a_1 jusqu'au début de l'arc a_k est donné par:

$$C_{a_1 a_k} = \sum_{n=1}^{k-1} (d_{a_n} + d_{a_n a_{n+1}}).$$

Le problème est de déterminer les plus courts chemins pour un arc donné a_1 vers chaque arc $a_k \in \mathcal{A}$.

L'approche qu'ils proposent est basée sur le travail de Caldwell [9] et elle est formulée dans le théorème suivant:

Théorème: *Pour un arc donné a_1 et pour tous les arc $a_k \in \mathcal{A}$, les coûts $C^*_{a_1 a_k}$ des plus courts chemins sont l'unique solution des équations fonctionnelles*

$$\begin{aligned} C_{a_1 a_k} &= \min_{a_i \in B(a_k)} \{C_{a_1 a_i} + d_{a_i} + d_{a_i a_k}\}, \ a_k \neq a_1; \\ C_{a_1 a_1} &= 0; \end{aligned}$$

si, et seulement si, il n'existe pas de circuit de coût négatif.

L'ensemble $B(a_k)$ contient les arcs a_i tels que $d_{a_i a_k} < \infty$. Ces équations fonctionnelles peuvent être résolues par un algorithme de calcul des plus courts chemins conventionnels. Cependant, les auteurs ne fournissent aucun resultat numérique permettant de vérifier les performances de leur méthode. Il est important de noter que cette approche calcule les plus courts chemins d'un arc origine vers la fin de chaque arc contrairement aux méthodes traditionnelles qui visent à déterminer l'arborescence des plus courts chemins reliant un noeud origine avec tous les autres noeuds du réseau. Cette approche associe donc une étiquette à chaque arc du réseau.

Une autre méthode évitant la modification du réseau original fût proposée par Easa [23]. L'approche de Easa vise à déterminer les plus courts chemins dans un réseau compor-

tant certaines intersections où un ou plusieurs mouvements sont interdits. Pour $i \in \bar{\mathcal{N}}$, on définit $\mathcal{A}_i^-(a_l)$ qui représente l'ensemble des arcs entrant au noeud i et pouvant rejoindre l'arc a_l. Si $i \notin \bar{\mathcal{N}}$, alors clairement $\mathcal{A}_i^-(a_l) \equiv \mathcal{A}_i^-$. On suppose que les noeuds origines et destinations i sont dans $\bar{\mathcal{N}}$ et on pose $\mathcal{A}_i^-(a_l) = \emptyset, \forall a_l \in \mathcal{A}$, de manière à prévenir toute transition par ces noeuds. Supposons que $d_{a_l} \geq 0$, $\forall a_l \in \mathcal{A}$, et soit π_i^k est le coût du chemin du noeud origine r au noeud i. Pour $i \in \bar{\mathcal{N}}$, on conserve quatre coûts correspondant aux coûts des chemins possibles du noeud origine r au noeud i; donc $k = 1, 2, 3, 4$. Pour $i \notin \bar{\mathcal{N}}$, seulement un coût est conservé et, par conséquent $\pi_i^k \equiv \pi_i$. Enfin, définissons $P_k(i)$ comme l'ensemble des arcs entrant au noeud i et pour lesquels il est possible de rejoindre le noeud i au coût π_i^k.

Un chemin admissible entre le noeud origine r et un noeud j dans le réseau peut être défini par la suite de noeuds $r = i_1, i_2, ..., i_v, ..., i_q = j$ (pas nécessairement distincts) et la suite $a_1, a_2, ..., a_l, ..., a_m$ d'arcs distincts dans $\mathcal{A}$, telles que $a_l \in \mathcal{A}_{i_v}^-(a_{l+1})$ pour $i_v \in \bar{\mathcal{N}}$, où a_l et a_{l+1} sont respectivement l'arc entrant et l'arc sortant du noeud i_v. Le coût de ce chemin admissible est donné par $\sum_{l=1}^{m} d_{a_l}$. L'algorithme proposé par Easa s'applique à résoudre les équations fonctionnelles énoncées dans le théorème suivant:

Théorème: *Les coûts des plus courts chemins du noeud r vers tous les autres noeuds $j \in \mathcal{N}$ du réseau comportant un ensemble $\bar{\mathcal{N}} \subseteq \mathcal{N}$ de noeuds avec mouvements interdits et un coût positif d_{a_l} pour chaque arc $a_l = (i, j) \in \mathcal{A}$ sont représentés par l'unique solution des équations fonctionnelles*

$$\begin{aligned} \pi_r^k &= 0; \\ \pi_j^k &= \min_{i:(i,j)\in\mathcal{A}_j^-} \{\tilde{\pi}_i + d_{a_l}\},\ j \neq r; \end{aligned}$$

où

$$\tilde{\pi}_i = \begin{cases} \min_{P_k(i)\in A_i^-(a_l)} \pi_i^k, & i \in \bar{N} \\ \pi_i, & i \notin \bar{N} \end{cases}.$$

On peut remarquer que si $\bar{\mathcal{N}} = \emptyset$, alors ces équations correspondent aux équations d'optimalité formulées par Bellman [5] et décrivent les plus courts chemins dans un réseau sans mouvement interdit.

Les résultats obtenus par Easa semblent montrer que sa procédure est efficace. Cependant, l'algorithme suggéré ne considère pas le cas où les intersections contiennent également des mouvements pénalisés. L'utilisation de cette procédure est donc limitée puis-

qu'en pratique il est souvent nécessaire de considérer des réseaux possédant des pénalités associées aux mouvements aux intersections.

Une variante de l'algorithme proposé par Kirby et Potts [37] a été développée par Spiess [51]. L'algorithme proposé associe également des étiquettes aux arcs contrairement à un algorithme conventionnel qui procède à un étiquetage sur les noeuds. (Son approche est entre autre utilisée dans le logiciel de planification en transport *emme/2* [32].) L'idée suggérée par Spiess exploite le fait que les mouvements pénalisés ou interdits ne se trouvent que sur un sous-ensemble de noeuds $\bar{\mathcal{N}} \subseteq \mathcal{N}$. Aux autres noeuds de $\mathcal{N}$, tous les mouvements sont permis sans délai additionnel, à l'exception des virages en U qui sont interdits par définition. La proposition suivante permet d'exploiter le fait précédent:

Proposition (Spiess): *Soit $(i_1, i_2, i_3, ..., i_{k-1}, i_k)$ la séquence de noeuds décrivant un plus court chemin $\mathcal{P}$ reliant $r = i_1$ et i_k sur un réseau avec coûts positifs sur les liens et pénalités de virage non-négatives sur un sous-ensemble de noeuds $\bar{\mathcal{N}}$. Si le même noeud i apparaît deux fois dans le chemin $\mathcal{P}$, c'est-à-dire $i = i_m = i_n$, où $m < n$, alors il existe un noeud $i_{m'} \in \bar{\mathcal{N}}$ avec $m' \leq m$ et un noeud $i_{n'} \in \bar{\mathcal{N}}$ avec $n' \geq m$.*

Preuve: *Procédons par contradiction. Supposons qu'il n'existe pas de noeuds $i_{m'}$ et $i_{n'}$. Ceci implique que le chemin $\mathcal{P}' = (i_1, i_2, ..., i_{m-1}, i_m = i_n, i_{n+1}, ..., i_k)$ est un chemin qui ne contient aucun noeud appartenant à $\bar{\mathcal{N}}$. La longueur du chemin $\mathcal{P}$ est alors la somme de la longueur du chemin $\mathcal{P}'$ et la longueur circuit $(i_m, i_{m+1}, ..., i_{n-1}, i_n = i_m)$. Puisque ce circuit contient au moins deux arcs, sa longueur est strictement positive. Ceci implique que le chemin $\mathcal{P}'$ est un chemin de plus petite longueur que le chemin $\mathcal{P}$. Ce qui contredit le fait que le chemin $\mathcal{P}$ est un plus court chemin.*

Ce résultat implique que nous n'avons plus à considérer les arcs $a = (i, j)$ quand le plus court chemin de l'origine r au noeud j est connu et ne contient pas de noeuds $i' \in \bar{\mathcal{N}}$. Ceci permet de réduire considérablement l'effort requis pour calculer les plus courts chemins à partir d'une origine r vers toutes les destinations. En effet, cette observation permet d'éliminer les chemins vers des arcs qui ne peuvent pas faire partie d'un plus court chemin d'une origine vers une destination.

Étant donné un sous-ensemble $\tilde{\mathcal{N}} \in \mathcal{N}$ de destinations q. Soient π_a, le coût d'un plus court chemin de l'origine r jusqu'à la fin de l'arc $a = (i, j) \in \mathcal{A}$ et π_q, le coût d'un plus court chemin de l'origine r jusqu'à la destination q. Soient $\mathcal{Q}$, l'ensemble des arcs qui ont été rejoints (arcs candidats) et b_a (b_q) l'arc prédécesseur de l'arc a (de la destination q).

Étant donné que cet algorithme utilise une technique d'*extensions sélectives* (on choisit l'arc $\bar{a}$ possédant la plus petite étiquette parmi les arcs de l'ensemble $\mathcal{Q}$), il est possible d'utiliser les structures de données présentées à la section 1.1.4 pour implanter cet algorithme. Enfin, soit ϕ_i, $i \in \mathcal{N}$ qui prend la valeur 1 si le chemin de r à i (inclusivement) ne contient pas de noeud $i' \in \bar{\mathcal{N}}$ et 0 autrement. L'algorithme s'écrit alors comme suit:

1. Initialisation

$\pi_q = \infty, b_q = \infty, \ \forall q \in \bar{\mathcal{N}}$;

$\phi_i = 0, \ i \in \mathcal{N} - p; \ \phi_p = 1$;

$\pi_a = \infty, b_a = \infty, \ \forall a \in \mathcal{A} - \mathcal{A}_p^+$;

$\pi_a = d_a, b_a = 0, \ \forall a \in \mathcal{A}_p^+$;

$\mathcal{Q} = \mathcal{A}_p^+$;

2. Sélection des arcs à traiter

tant que $\mathcal{Q} \neq \emptyset$ faire

déterminer $\bar{a} = (i, j) \in \mathcal{Q}$ tel que $\pi_{\bar{a}} \leq \pi_a, \forall a \in \mathcal{A}$

si $j \in \bar{\mathcal{N}}$, alors aller à 3

si $j \in \bar{\mathcal{N}}$, alors aller à 4

autrement aller à 5

$\mathcal{Q} = \mathcal{Q} - \{\bar{a}\}$;

3. Noeud j est une destination

$\pi_q = \pi_{\bar{a}}; b_q = \bar{a}$;

4. Noeud j est une intersection

pour $a = (j, l) \in \mathcal{A}_j^+$ tels que $\phi_l = 0$ faire

si $\pi_{\bar{a}} + d_{\bar{a}a} + d_a < \pi_a$ alors

$\pi_a = \pi_{\bar{a}} + d_{\bar{a}a} + d_a$;

$b_a = \bar{a}$;

$\mathcal{Q} = \mathcal{Q} + \{a\}$;

5. Noeud j est un noeud régulier

pour $a = (j, l) \in \mathcal{A}_j^+$ tels que $l \neq i$ faire

si $\pi_{\bar{a}} + d_a < \pi_a$ alors

$\pi_a = \pi_{\bar{a}} + d_a$;

$b_a = \bar{a}$;

$\mathcal{Q} = \mathcal{Q} + \{a\}$;

$\phi_j = \phi_i$;

L'optimisation visant à éliminer les arcs inintéressants est effectuée au moyen de la variable ϕ associée à chaque noeud. Il est également important de noter que cet algorithme s'assure de ne pas considérer, dans le cas où le noeud terminal du lien traité est un noeud régulier, les chemins contenant des virages en U. (Lorsque le noeud terminal du lien considéré est une

intersection, il n'est pas nécessaire d'éviter le mouvement en U puisque seuls les mouvements permis sont définis pour cette intersection.) Cette caractéristique ne fait que rendre le comportement de l'algorithme plus réaliste. Les expérimentations réalisées par Spiess lui permettent de conclure que, dans le cas où $\bar{\mathcal{N}} = \emptyset$, l'élimination de certains arcs produit un algorithme qui n'est que légèrement plus lent qu'un algorithme avec étiquettes sur les noeuds.

Dans le but de calculer les plus courts chemins dans un réseau où les virages pénalisés sont définis pour un sous-ensemble de noeuds, Della Valle et Tartaro [17] proposent un algorithme effectuant un étiquetage mixte, c'est-à-dire que des étiquettes sont associées aux noeuds ainsi qu'aux arcs. L'algorithme *hybride* qu'ils suggèrent permet, comme celui proposé par Spiess, de calculer les plus courts chemins à partir d'une origine r vers un sous-ensemble de noeuds destinations. On peut le résumer de la façon suivante:

1. Sélection du noeud ou de l'arc à traiter

tant que $\mathcal{Q} \neq \emptyset$ faire

sélectionner le premier élément de la liste $\mathcal{Q}$

si c'est un noeud, aller à l'étape 2

si c'est un arc, aller à l'étape 3

2. Traitement d'un noeud i

pour chaque arc $(i,j) \in \mathcal{A}_i^+$ faire

si j est un noeud pénalisé, aller à l'étape 2a

si j est un noeud non pénalisé, aller à l'étape 2b

2a. Le noeud j est pénalisé

pour chaque arc $a = (j,k) \in \mathcal{A}_j^+$ faire

si $\pi_a > \pi_i + d_{ij} + d_{ijk}$, alors $\pi_a = \pi_i + d_{ij} + d_{ijk}$;

2b. Le noeud j est non pénalisé

si $\pi_j > \pi_i + d_{ij}$, alors $\pi_j = \pi_i + d_{ij}$;

3. Traitement d'un arc $a = (i,j)$

si j est un noeud pénalisé, aller à l'étape 3a

si j est un noeud non pénalisé, aller à l'étape 3b

3a. Le noeud j est pénalisé

pour chaque arc $\bar{a} = (j,k) \in \mathcal{A}_j^+$ faire

si $\pi_{\bar{a}} > \pi_a + d_{ij} + d_{ijk}$, alors $\pi_{\bar{a}} = \pi_a + d_{ij} + d_{ijk}$;

3. Le noeud j est non pénalisé

si $\pi_j > \pi_a + d_{ij}$, alors $\pi_j = \pi_a + d_{ij}$;

Afin de comparer les performances de leur algorithme, Della Valle et Tartaro ont implanté une approche utilisant un étiquetage sur les noeuds jumelé à une expansion du réseau

(modélisation explicite des virages) ainsi qu'une approche utilisant un étiquetage sur les arcs. Les expérimentations réalisées sur deux réseaux urbains relativement petits leurs permettent de conclure que l'algorithme *hybride* est très compétitif vis-à-vis les deux autres approches implantées. En fait, leur algorithme domine le premier type d'approche lorsqu'au moins 30% des intersections sont pénalisées, tandis qu'il domine toujours l'approche utilisant un étiquetage sur les arcs lorsqu'un certain pourcentage de noeuds sont non pénalisés. Lorsque tous les noeuds sont pénalisés, les temps d'exécution de l'algorithme *hybride* et de celui avec étiquettes sur les arcs coïncident tandis que l'approche avec étiquettes sur les noeuds obtient un temps d'exécution 15% plus élevé.

Bien que les résultats présentés laissent entendre que cet algorithme *hybride* est plus performant qu'une approche utilisant seulement un étiquetage sur les arcs, quelques caractéristiques importantes de la méthode *hybride* doivent être prises en considération avant de conclure à sa supériorité. D'abord, cet algorithme utilise une *deque* comme structure de données lui permettant de conserver les noeuds et les arcs candidats. L'utilisation de cette structure de données ne permet pas de terminer le calcul des plus courts chemins lorsque toutes les destinations sont atteintes. Une autre caractéristique importante repose sur la manière dont les étiquettes sont affectées sur les arcs. En effet, contrairement à l'approche suggérée par Spiess qui associe des étiquettes à la fin des arcs, cet algorithme donne des étiquettes au début des arcs. Par conséquent, même une nouvelle implantation de cet algorithme utilisant une technique d'*extensions sélectives* ne pourrait profiter des optimisations possibles dans le cas de l'algorithme proposé par Spiess.

En conclusion, si l'on veut déterminer les plus courts chemins à partir d'une origine vers tous les noeuds, alors l'algorithme *hybride* s'avère un meilleur choix qu'un algorithme avec étiquetage sur les arcs (comme celui proposé par Spiess). Cependant, cet algorithme *hybride* ne s'avère pas la meilleure option dans le cas où seul un sous-ensemble de noeuds destinations doivent être considérés.

Récemment, Ziliaskopoulos et Mahmassani [58] ont proposé un algorithme procédant par *ajustements progressifs* et modifié de façon à tenir compte des délais et des mouvements interdits aux intersections. Leur algorithme est basé sur une extension de la structure d'adjacence utilisée normalement pour représenter les réseaux. Pour chaque noeud j, ils considèrent l'ensemble des mouvements vers les noeuds k : $(j,k) \in \mathcal{A}_j^+$ et, à chacun de ces mouvements, ils associent le délai d_{ijk} qui représente le délai associé au mouvement au noeud j vers le noeud k lorsque l'on vient du noeud i. Une valeur infinie est donnée aux délais correspondant aux mouvements interdits. Pour chaque noeud j, on maintient $|\mathcal{A}_j^+|+1$ étiquettes π_j^k qui repésentent la longueur d'un plus court chemin jusqu'à j lorsque le noeud

destination est le noeud k. À chaque itération, nous avons les $|\mathcal{A}_j^+|$ premières étiquettes qui représentent des bornes supérieures sur les plus courts chemins de l'origine au début des arcs sortant du noeud j et la $(|\mathcal{A}_j^+|+1)^{ième}$ étiquette, notée simplement π_j, qui conserve la valeur du plus court chemin terminant exactement au noeud j. La procédure proposée procède de la même façon qu'un algorithme d'*ajustements progressifs* conventionnel. Si au moins une des étiquettes du noeud j est modifiée, alors le noeud j est inséré dans une liste[4]. L'algorithme cherche à résoudre les équations fonctionnelles suivantes:

$$\pi_j^k = \begin{cases} \min_{i:(i,j)\in\mathcal{A}_j^-} d_{ijk} + d_{ij} + \pi_i^j, \ \forall j \in N, \ k:(j,k) \in \mathcal{A}_j^+ \\ = 0, \ \forall k:(p,k) \in \mathcal{A}_r^+ \end{cases}$$

où r est le noeud origine. Ziliaskopoulos et Mahmassani font remarquer que cette approche satisfait le principe d'optimalité. En effet, au meilleur chemin de l'origine r au noeud j en direction du noeud k correspond le meilleur chemin au noeud i en direction du noeud j. Soit $\mathcal{Q}$ une liste contenant les noeuds à visiter. Alors l'algorithme est le suivant:

1. Initialisation

$\pi_i^j = \infty, \ \forall i \in \mathcal{N}, \ j:(i,j) \in \mathcal{A}_i^+$;
$\pi_r^k = 0, \ \forall k:(r,k) \in \mathcal{A}_r^+$;
$\mathcal{Q} = \{r\}$;

2. Étape principale

si $\mathcal{Q} = \emptyset$, alors aller à 4
sélectionner le premier noeud i de la liste $\mathcal{Q}$
pour chaque noeud $j:(i,j) \in \mathcal{A}_i^+$ faire
pour chaque noeud $k:(j,k) \in \mathcal{A}_j^+$ faire
si $\pi_j^k > d_{ijk} + d_{ij} + \pi_i^j$ alors remplacer π_j^k par la nouvelle valeur $d_{ijk} + d_{ij} + \pi_i^j$;
(*pour le mouvement terminal au noeud* j)
si $\pi_j > d_{ij} + \pi_i^j$, alors remplacer π_j par la nouvelle valeur $d_{ij} + \pi_i^k$;
si au moins une des étiquettes π_j^k a été modifiée, alors le noeud j est inséré dans $\mathcal{Q}$;

3. Répéter l'étape 2

4. L'algorithme est terminé

Dans leur implantation de cet algorithme, Ziliaskopoulos et Mahmassani utilisent la structure de données *deque*. Les résultats qu'ils obtiennent leur permettent de conclure que leur algorithme performe mieux que l'utilisation d'un algorithme conventionnel sur un réseau dont les mouvements aux intersections ont été explicités et qu'il est également plus

[4]Voir section 1.1.3.

efficace qu'une procédure utilisant une technique d'extensions sélectives. Une observation attentive de l'algorithme proposé par Ziliaskopoulos et Mahmassani permet cependant de constater que cette approche est équivalente à celle consistant à associer des étiquettes au début des arcs et jumelée à une *recherche par parcours arrière* (*backward star*). Ainsi, dans un réseau où $\tilde{\mathcal{N}} = \mathcal{N}$, le nombre d'étiquettes traité par cet algorithme est équivalent à celui traité par une méthode associant des étiquettes aux arcs. Si, de plus, seul un sous-ensemble d'origines doit être traité, alors il est plus avantageux d'utiliser une méthode d'*extension sélective* jumelée à un étiquetage sur les arcs.

Chapitre 2

Implantations séquentielles d'algorithmes de calcul des plus courts chemins statiques et temporels

Dans ce chapitre, nous nous intéressons à l'implantation séquentielle d'algorithmes de calcul des plus courts chemins statiques et temporels traités au chapitre 1. Dans le cas statique, nous présentons des implantations d'algorithmes pour le calcul des plus courts chemins «conventionnels» et des implantations d'un algorithme considérant les délais et les interdictions aux intersections. Dans le cas temporel, trois algorithmes récemment développés sont implantés.

Le contexte de **planification des transports** nous procure un cadre pratique pour le développement de nos implantations. L'objectif de ce chapitre est de fournir des outils de base (outils de calcul des plus courts chemins) pouvant être utilisés par des modèles de planification plus complexes (modèles d'équilibre dans les réseaux [24], modèles de simulation [4], *etc*).

La première section de ce chapitre discute du langage de programmation choisi, de la modélisation des données et de l'environnement de travail utilisé lors du développement et des expérimentations. Les implantations d'algorithmes de calcul des plus courts chemins statiques sont présentées dans la section 2.2 tandis que la section 2.3 s'attarde aux implantations d'algorithmes de calcul des plus courts chemins temporels.

2.1 Langage de programmation, modélisation des données et environnement de travail

Le processus de développement d'une implantation comprend généralement une étape préliminaire très importante impliquant, d'une part, le choix du langage de programmation et, d'autre part, la façon de représenter les données. Le choix du langage de programmation est une phase déterminante, puisqu'elle restreint les possibilités du programmeur aux capacités du langage choisi. De son côté, la phase consacrée à la modélisation des données a également un impact important puisque les données doivent pouvoir être traitées efficacement.

2.1.1 Langage de programmation

Le choix du langage de programmation constitue la phase initiale de notre processus de développement. Le type d'application considéré, le niveau d'efficacité nécessaire et le degré de réutilisation désiré sont, entre autre, des questions que l'on doit se poser préalablement. Cette réflexion a comme objectif de faire ressortir les aspects que l'on juge importants. Dans notre cas, le langage de programmation orienté objet *C++* [53] a été retenu pour diverses raisons. Comme les langages de programmation *FORTRAN* et *C*, le langage *C++* permet de développer des applications pour lesquelles la performance est jugée importante. Cependant, notre choix est également motivé par les raisons suivantes:

- le concept de classe présent dans le langage *C++* rend les programmes plus faciles à lire, à maintenir et à réutiliser;
- plusieurs mécanismes offerts en *C++*, comme l'héritage et les types paramétrés (*template*) permettent une réutilisation du code et une généralisation de son utilisation;
- la souplesse offerte par le langage *C++*, comparativement aux langages *C* et *FORTRAN*, a très peu d'impact (s'il y en a) sur la performance.

En fait, le langage de programmation *C++* est maintenant le langage utilisé pour le développement d'applications d'envergures. Choisir ce langage de programmation ne fait que confirmer le désir de développer des applications performantes et durables.

Dans les sections qui suivent, nous présentons les codes *C++* associés à nos implantations. Dans la mesure du possible, des identificateurs significatifs ont été utilisés afin de rendre les programmes lisibles par toute personne n'ayant pas nécessairement une connais-

sance approfondie du langage *C++*. Cependant, quelques opérateurs peuvent être inconnus et sont donc définis préalablement:

- *: utilisé pour définir un pointeur sur un type quelconque;
- . ou →: utilisés pour accéder à une méthode ou aux champs d'une classe (structure);
- *new*: permet d'allouer l'espace nécessaire à la création d'un objet de type défini.

2.1.2 Modélisation des données

Dans cette partie, nous traitons de la modélisation des données, c'est-à-dire des réseaux de transport. Notre discussion débute d'abord par une description des composantes des réseaux de transport utilisés. Nous poursuivons ensuite en définissant notre façon de représenter ces réseaux de transport et en donnant les caractéristiques des réseaux de transport utilisés pour tester nos implantations.

Composantes d'un réseau de transport

Étant donné le contexte précis dans lequel ce mémoire est réalisé, nous adoptons une terminologie relativement commune dans le domaine du transport. Considérons le réseau de la figure 2.1. Ce réseau est composé de quatre **noeuds** (*nodes*), six **liens** (*links*) et trois **virages** (*turns*) pénalisés définis explicitement. Parmi l'ensemble des noeuds, nous dénotons par **centroïde** (*centroid*) une zone origine et/ou destination. Dans la modélisation d'un réseau, on représente le centroïde comme un *noeud virtuel* connecté à un ou plusieurs noeuds

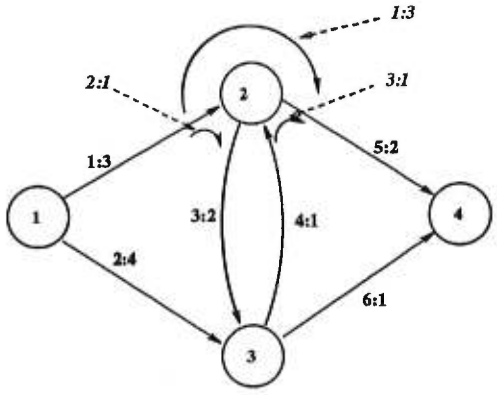

Figure 2.1: Exemple de réseau.

et par lequel il n'est pas possible de transiter. Les noeuds 1 et 4 de la figure 2.1 pourraient, par exemple, agir comme deux centroïdes.

Enfin, parmi l'ensemble des noeuds qui ne sont pas des centroïdes, on définit par **intersection**, un noeud dont au moins un virage est pénalisé et par **noeud régulier**, un noeud où tous les mouvements sont permis sans délai additionnel. Les concepts d'intersection et de noeud régulier sont implicites, c'est-à-dire qu'ils n'apparaissent pas explicitement dans la définition du réseau. (Ces concepts sont généralement utilisés dans la conception d'algorithme traitant les délais et les interdictions aux intersections.)

Représentation des réseaux de transport

Notre façon de représenter les réseaux de transport doit nous permettre de faciliter notre travail lors du développement des implantations. La représentation choisie doit également nous permettre de pouvoir traiter les données efficacement et être la plus générale possible afin de favoriser les échanges de données entre différents outils développés.

Nous disposons au **Centre de recherche sur les transports** (*C.R.T.*) d'un format de fichier nous permettant de satisfaire les exigences mentionnées précédemment. Il s'agit du format de fichier **NFF** (*Network File Format*)[1]. Bien que le format de fichier *NFF* ait été conçu de manière à pouvoir représenter divers types de données, il est particulièrement bien adapté à la représentation des réseaux. À titre d'exemple, le réseau de la figure 2.1 a été représenté à l'aide du format de fichier *NFF*. Le fichier résultant est présenté dans la figure 2.2. Chaque section (*Nodes*, *Centroids*, *Links* et *Turns*) contient un certain nombre d'attributs. Par exemple, la section *Nodes* comporte trois attributs: un identificateur (ID) et deux coordonnées (X et Y).

Notre utilisation du format de fichier *NFF* se justifie davantage par la présence de la librairie de support *C++* y étant associée. Cette librairie de support permet un traitement efficace des données et facilite le développement d'outils utilisant ces données. Par exemple, il est possible d'accéder facilement et efficacement aux listes de liens incidents vers l'extérieur (ou vers l'intérieur) d'un noeud. Ce genre d'opération apparaît très fréquemment dans les algorithmes concernant les problèmes de réseaux. Après un traitement particulier des données du fichier présenté dans la figure 2.2 à l'aide de la librairie, nous obtenons un nouveau fichier (figure 2.3) contenant certaines informations additionnelles. La section *Nodes* contient maintenant les listes des liens sortants et entrants pour chacun des noeuds du réseau. De même, la section *Links* contient, pour chaque lien, les listes de virages sortants

[1]Une documentation détaillée du format de fichier *NFF* est disponible à l'adresse *WWW* suivante: *http : //www.crt.umontreal.ca/~lab_sit/DOC − NFF.*

```
<nff 3.0>
<title>exemple</title>
<section Nodes>
 <structure>
   long ID;
   float X;
   float Y;
 </structure>
 <data>
   1, 23.45, 34.45;
   2, 44.67, 44.12;
   3, 12.34, 66.90;
   4, 43.68, 44.22;
 </data>
 </structure>
</section>
<section Centroids>
 <structure>
   long ID;
   ref Nodes Node;
 </structure>
 <data>
   1,1;
   2,4;
 </data>
</section>
```

```
<section Links>
 <structure>
   long ID;
   ref Nodes From;
   ref Nodes To;
   float Cost;
 </structure>
 <data>
   1,1,2,3;
   2,1,3,4;
   3,2,3,2;
   4,3,2,1;
   5,2,4,2;
   6,3,4,1;
 </data>
</section>
```

```
<section Turns>
 <structure>
   long ID;
   long At;
   ref Links From;
   ref Links To;
   float Cost;
 </structure>
 <data>
   1,2,1,5,3;
   2,2,1,3,1;
   3,2,4,5,1;
 </data>
</section>
</nff>
```

Figure 2.2: Fichier NFF original.

```
<nff 3.0>
<title>exemple</title>
<section Nodes>
 <structure>
   long ID;
   float X;
   float Y;
   ref Centroids Centroid;
   (ref) Links IncomingLinks;
   (ref) Links OutgoingLinks;
 </structure>
 <data>
   1, 23.45, 34.45,1,(),(1,2);
   2, 44.67, 44.12,-1,(1,4),(3,5);
   3, 12.34, 66.90,-1,(2,3),(4,6);
   4, 43.68, 44.22,2,(5,6),();
 </data>
 </structure>
</section>
<section Centroids>
 <structure>
   long ID;
   ref Nodes Node;
 </structure>
 <data>
   1,2;
   2,4;
 </data>
</section>
```

```
<section Links>
 <structure>
   long ID;
   ref Nodes From;
   ref Nodes To;
   float Cost;
   (ref) Turns IncomingTurns;
   (ref) Turns OutgoingTurns;
 </structure>
 <data>
   1,1,2,3,(),(1,2);
   2,1,3,4,(),();
   3,2,3,2,(2),();
   4,3,2,1,(),(3);
   5,2,4,2,(1,3),();
   6,3,4,1,(),();
 </data>
</section>
```

```
<section Turn
 <structure>
   long ID;
   long At;
   ref Links F
   ref Links T
   float Cost;
 </structure>
 <data>
   1,2,1,5,3;
   2,2,1,3,1;
   3,2,4,5,1;
 </data>
 </section>
</nff>
```

Figure 2.3: Fichier NFF modifié.

et entrants. Par exemple, le noeud 2 a deux liens entrants (1 et 4) et deux arcs sortants (3 et 5) tandis que le lien 3 a un virage entrant (2). La librairie de support associée au format de fichier *NFF* permet en fait de générer toute l'information généralement nécessaire et ce, à partir des données minimales fournies au départ. Notre utilisation du format de fichier *NFF* s'appuie donc sur la facilité d'accès à l'information et l'efficacité dans le traitement de cette information.

Caractéristiques des réseaux de transport utilisés

Afin de vérifier les performances des implantations présentées dans les sections 2.2 et 2.3 ainsi que dans le chapitre 3, nous disposons de données réelles relatives aux réseaux de transport des villes canadiennes de Winnipeg, Ottawa et Montréal. Ces réseaux proviennent de la banque de données du logiciel planification en transport *emme/2* [32]. Les caractéristiques de ces réseaux sont fournies dans le tableau 2.I.

Ville	Nb. de noeuds	Nb. de liens	Nb. de centroïdes	Nb. de virages
Winnipeg	1057	2535	338	154
Ottawa	2569	6963	258	703
Montréal	6906	17157	699	490

Tableau 2.I: Caractéristiques des réseaux de transport.

2.1.3 Description de l'environnement de travail

Dans la suite de ce travail, un certain nombre d'expérimentations visant à étudier le comportement des diverses implantations d'algorithmes de plus courts chemins statiques et temporels sont effectuées. Le tableau 2.II donne la description des deux types d'environnement de travail utilisés lors de ces expérimentations. Nous disposons au *C.R.T.* de 16 stations

Environnement	Nombre de processeurs	Mémoire vive	Fréquence de l'horloge	Système d'opération (*OS*)
SUN SPARC Ultra 1/140	1	64 Mb	140 MHz	SOLARIS 2.5
SUN SPARC Server 1000	8	256 Mb	40 MHz	SOLARIS 2.5

Tableau 2.II: Environnements de travail utilisés.

de travail *SUN SPARC Ultra 1/140* connectées sur un réseau *Ethernet* 100 *Mbits/sec.* La performance de ces machines ainsi que de la configuration du réseau seront particulièrement utiles lors du traitement parallèle [2]. Le *SUN SPARC Server 1000* est une machine à 8 processeurs à *mémoire partagée* [3]. En plus d'être aussi très utile pour le traitement parallèle, cet environnement de travail nous donne la possibilité d'utiliser des réseaux de taille importante puisque nous avons accès à 256 Mo de mémoire vive. Enfin, pour conclure cette section, notons que toutes les implantations présentées dans ce travail sont compilées à l'aide du compilateur *SUN C++*.

[2] Voir chapitre 3.

[3] La section 3.1 du chapitre 3 définit ce type d'architecture.

2.2 Implantations séquentielles d'algorithmes de calcul des plus courts chemins statiques

Dans cette section, nous nous attardons à l'implantation séquentielle d'algorithmes de calcul des plus courts chemins statiques. Cependant, l'objectif n'est pas d'essayer de reproduire les résultats qui ont déjà été obtenus dans les recherches antécédentes ([18, 28, 40, 15]). Notre objectif est plutôt d'utiliser un langage de programmation orienté objet afin de fournir des outils de calcul de plus courts chemins statiques simples, facilement réutilisables et pouvant répondre à divers besoins et ce, dans un contexte de **planification des transports**. Par conséquent, nos implantations sont conçues en fonction d'une utilisation sur des réseaux possédants certaines caractéristiques. Dans un premier temps, nous avons effectué un certain nombre d'implantations d'algorithmes de calcul des plus courts chemins «conventionnels». Ensuite, étant donné le contexte dans le lequel ce mémoire est réalisé, nous avons également conçu des implantations d'un algorithme considérant les délais et les interdictions aux intersections.

Dans la première partie de cette section, nous discutons brièvement des algorithmes implantés. La deuxième partie traite du développement des implantations et, en conclusion, nous présentons l'analyse des résultats obtenus suite aux expérimentations réalisées.

2.2.1 Algorithmes implantés

Rappelons brièvement la notation utilisée dans le chapitre 1. Soit $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ un réseau avec, pour chaque lien $a = (i, j) \in \mathcal{A}$, une fonction de longueur (coût) $d : \mathcal{A} \rightarrow \Re$. On note d_{ij} (ou d_a) la longueur du lien $a = (i, j)$. Étant donné un noeud origine r, on définit par $\mathcal{T}(r)$, l'arborescence de racine r contenant les plus courts chemins de r à i, pour tous les noeuds $i \neq r$. On définit également $\mathcal{A}_i^+ = \{(i, j) \in \mathcal{A}\}$ et $\mathcal{A}_i^- = \{(j, i) \in \mathcal{A}\}$ représentant respectivement l'ensemble des liens sortant (*forward structure*) et l'ensemble des liens entrant (*backward structure*) au noeud i. Finalement, soit π_i (π_a), la longueur d'un plus court chemin de r à i (de r à a). Le problème consiste à trouver, à partir d'un noeud origine r, l'arborescence $\mathcal{T}(r)$ des plus courts chemins vers tous les autres noeuds $i \in \mathcal{N}, i \neq r$.

Algorithmes de calcul des plus courts chemins conventionnels

Parmi les implantations utilisant une technique d'*ajustements progressifs*, décrites dans la section 1.1.2, nous considérons celles utilisant respectivement les structures de données *queue*, *deque* et *2queue*. Une implantation correspondant à une méthode d'*extensions sélectives* est

également considérée. Il s'agit d'une l'implantation avec *monceau*. Pour chacune de ces structures de données, des implantations utilisant une recherche par parcours avant (*forward star*) et une recherche par parcours arrière (*backward star*) sont réalisées. Rappelons que dans le cas de la recherche par parcours avant, les plus courts chemins sont calculés à partir d'un noeud origine vers tous les autres noeuds tandis que, dans le cas de la recherche par parcours arrière, le calcul des plus courts chemins est effectué pour un noeud destination à partir de tous les autres noeuds.

Dans le cas de l'implantation avec *monceau*, une spécialisation de l'algorithme est également implantée. Cette spécialisation permet de calculer les plus courts chemins vers ou à partir d'un ensemble restreint de noeuds (centroïdes), selon le type de recherche implantée.

Algorithme de calcul des plus courts chemins considérant les délais et les interdictions au intersections

Un certain nombre d'algorithmes de calcul des plus courts chemins considérant les délais et les interdictions aux intersections ont été conçus jusqu'à maintenant. Parmi ceux-ci, nous avons choisi celui proposé par Spiess[4] puisqu'il s'est avéré, jusqu'à présent, un des plus efficaces. Initialement, cet algorithme fût conçu pour être utilisé dans une approche permettant de résoudre le modèle d'*affectation d'équilibre dans un réseau urbain à demande fixe avec pénalités de virages sur les noeuds d'intersection* [51]. L'approche de résolution en question implique en fait le calcul des plus courts chemins reliant seulement chaque paire **origine/destination** du réseau. Par conséquent, l'algorithme développé par Spiess se termine dès que toutes les destinations sont atteintes. De plus, lors du calcul, il ne considère que les liens pouvant faire partie d'un plus court chemin vers une destination.

Contrairement aux méthodes de calcul conventionnelles qui associent des étiquettes aux noeuds, l'approche développée par Spiess procède en donnant des étiquettes aux liens. Soient une origine r et un ensemble de destinations q, soient π_a (π_q), le coût d'un plus court chemin de l'origine r jusqu'à la fin du lien $a = (i, j) \in \mathcal{A}$ (jusqu'à la destination $q \in \mathcal{D}$) et b_a (b_q), le lien précédant le lien a (la destination q). Définissons également $\mathcal{Q}$, l'ensemble des liens qui ont été rejoints (liens candidats). Enfin, soit ϕ_i, $i \in \mathcal{N}$ qui prend la valeur 1 si le chemin de r à i (inclusivement) ne contient pas de noeuds $k \in \bar{\mathcal{N}}$ et 0 autrement. L'algorithme s'écrit alors ainsi:

[4]Voir section 1.3.2.

1. Initialisation

$\pi_q = \infty, b_q = -1, \ \forall q \in \mathcal{D}$;

$\phi_i = 0, \ i \in \mathcal{N} - r; \ \phi_r = 1$;

$\pi_a = \infty, b_a = -1, \ \forall a \in \mathcal{A} - \mathcal{A}_r^+$;

$\pi_a = d_a, \forall a \in \mathcal{A}_r^+$;

$\mathcal{Q} = \mathcal{A}_r^+$

2. Sélection des liens à traiter

tant que $\mathcal{Q} \neq \emptyset$ faire

déterminer $\bar{a} = (i, j) \in \mathcal{Q}$ tel que $\pi_{\bar{a}} \leq \pi_a, \forall a \in \mathcal{A}$

si $j \in \mathcal{D}$, alors aller à 3

si $j \in \mathcal{N}$, alors aller à 4

autrement aller à 5

$\mathcal{Q} = \mathcal{Q} - \{\bar{a}\}$;

3. Noeud j est une destination

$\pi_q = \pi_{\bar{a}}; b_q = \bar{a}$;

4. Noeud j est une intersection

pour $a = (j, l) \in \mathcal{A}_j^+$ tels que $\phi_l = 0$ faire

si $\pi_{\bar{a}} + d_{\bar{a}a} + d_a < \pi_a$ alors

$\pi_a = \pi_{\bar{a}} + d_{\bar{a}a} + d_a$;

$b_a = \bar{a}$;

$\mathcal{Q} = \mathcal{Q} + \{a\}$;

5. Noeud j est un noeud régulier

pour $a = (j, l) \in \mathcal{A}_j^+$ tels que $l \neq i$ faire

si $\pi_{\bar{a}} + d_a < \pi_a$ alors

$\pi_a = \pi_{\bar{a}} + d_a$;

$b_a = \bar{a}$;

$\mathcal{Q} = \mathcal{Q} + \{a\}$;

$\phi_j = \phi_i$;

Bien que cet algorithme s'avère très efficace pour le modèle pour lequel il a été conçu, d'autres modèles de planification routière peuvent avoir des besoins différents. Par exemple, il peut être intéressant d'avoir un algorithme permettant de donner une étiquette à chaque lien du réseau. (Rappelons que l'algorithme de Spiess ne donnera pas d'étiquettes permanentes aux liens qui ne font pas partie d'un plus court chemin vers une destination.) Plus spécifiquement, dans certains modèles de simulation, il est nécessaire de connaître les coûts des plus courts chemins reliant chaque lien du réseau à un ensemble de destinations données. Ce type de comportement peut être obtenu à partir de l'implantation avec monceau de l'algorithme de Spiess. Il suffit dans ce cas de ne pas utiliser la variable ϕ_i associée à chaque noeud et de poursuivre les calculs même si toutes les centroïdes sont atteints. Ce comportement peut également être obtenu à l'aide d'une implantation utilisant une technique d'*ajustements progressifs*. En effet, ce type d'implantations visitera nécessairement

chaque lien du réseau, même dans le cas où seuls les plus courts chemins vers ou à partir d'un sous-ensemble de noeuds (centroïdes) sont nécessaires.

Nous proposons donc, dans cette section, différentes implantations basées sur cet algorithme. Dans un premier temps, nous présentons des implantations avec *queue*, *deque* et *2queue*, ainsi qu'avec *monceau*. Ce premier type d'implantations se concentre sur le calcul des plus courts chemins d'un noeud origine vers chaque lien du réseau ou à partir de chaque lien du réseau vers un noeud destination. Ensuite, dans le but d'obtenir une implantation se spécialisant dans le calcul des plus courts chemins d'un noeud origine vers un sous-ensemble de noeuds (centroïdes), ou à partir d'un sous-ensemble de noeuds (centroïdes) vers un noeud destination, nous présentons des implantations avec *monceau* basées sur l'idée originale de Spiess. Comme précédemment, les deux types de recherche (parcours avant et parcours arrière) sont implantés dans chaque cas.

2.2.2 Développement des implantations

Remarque: Étant donné la similitude existant entre les implantations effectuant une recherche par parcours avant et celles effectuant une recherche par parcours arrière, seuls les détails concernant le premier type d'implantations sont donnés.

Comme il a déjà été mentionné dans la section 2.1.1, le développement des implantations a été effectué à l'aide du langage *C++*. L'utilisation de ce langage orienté objet permet de développer des outils efficaces pouvant être réutilisés aisément par d'autres outils. En fait, plusieurs caractéristiques du langage facilitent la conception, l'entretien et l'utilisation des classes développées. Entre autre, afin d'éviter la duplication de l'information et de «simplifier» le développement, il est possible d'utiliser le concept d'héritage.

La figure 2.4 permet de visualiser la hiérarchie reliant les classes développées. Au plus haut niveau, nous retrouvons les classes **SP** et **SPT**. Ces classes regroupent respectivement les outils de calcul des plus courts chemins conventionnels (*SP*) et ceux considérant les délais et les interdictions aux intersections (*SPT*). Au deuxième niveau, une classification selon le type de recherche (parcours avant ou parcours arrière) est effectuée. Dérivant de la classe *SP* (*SPT*) on retrouve les classes **FORWARDSP** (**FORWARDSPT**) et **BACKWARDSP** (**BACKWARDSPT**). Les outils de calcul des plus courts chemins se retrouvent bien entendu au niveau inférieur de la hiérarchie. La classe *SP* comprend donc les outils de calcul de plus courts chemins «conventionnels», c'est-à-dire qu'ils calculent les chemins reliant des paires de noeuds en ne considérant pas les délais et les interdictions qui peuvent exister aux intersections. Par contre, les outils appartenant à la classe *SPT* considèrent les

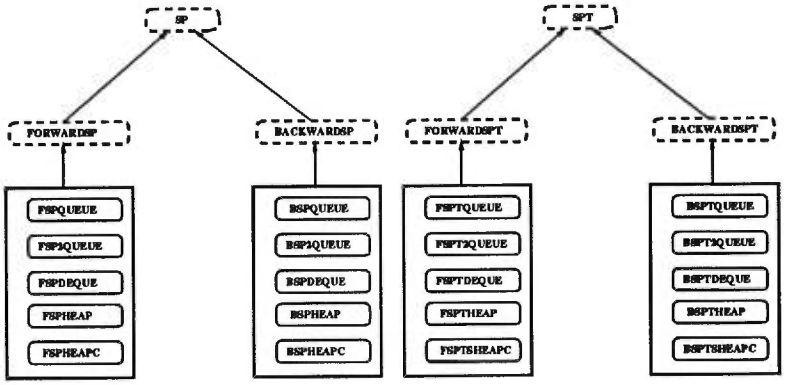

Figure 2.4: Hiérarchie des classes développés.

mouvements pénalisés et interdits aux intersections. De plus, l'algorithme utilisé procède en associant des étiquettes aux liens. Cette caractéristique est importante puisqu'elle permet, dans le cas de certains outils, de connaître le coût des plus courts chemins vers chaque lien ou à partir de chaque lien. Par exemple, dans le cas d'un modèle de simulation, ce genre d'information est essentiel puisqu'un véhicule se trouvant sur un lien quelconque doit être en mesure de connaître le coût du meilleur chemin vers la destination à partir de ce lien. Si l'algorithme utilisé donne des étiquettes aux noeuds, alors il est plus difficile d'obtenir l'information désirée.

Méthodes[a]	Fonctionnalités
FSP?? (NFF&); *FSP??* ();	Constructeurs.
void setNetwork (NFF&);	Permet de sélectionner un réseau.
void sp (const long id, float* a, int* b);	Calcule les *pccs* à partir d'un noeud origine vers tous les noeuds (toutes les destinations).
NFF* network () const;	Donne accès à l'objet NFF.
void setCostLink (float*);	Permet de sélectionner les coûts sur les liens
int numberOfLinks () const; int numberOfNodes () const; int numberOfCentroids () const;	Donne accès à l'information concernant le reseau.

[a]Des caractères ?? sont placés pour éviter de nommer chacune des classes concernées.

Tableau 2.III: Interface partagée par les classes dérivées de la classe FORWARDSP.

Toutes les classes représentant chacun des outils partagent une même interface. De cette façon, nous facilitons d'une part, l'utilisation de ces outils et d'autre part, le travail de mise à jour de la hiérarchie. Bien sûr, le comportement de certaines méthodes varie en

fonction du type de recherche (parcours avant ou parcours arrière) utilisé. Toutefois, c'est parfois le prix qu'il faut payer lorsque l'on désire unifier l'interface. Les tableaux 2.III et 2.IV présentent respectivement l'interface unique partagée par les classes dérivant de *FORWARDSP* et *FORWARDSPT*. La méthode $sp(\cdot)$ du tableau 2.III accepte trois paramètres.

Méthodes[a]	Fonctionnalités
FSPT?? (NFF&); *FSPT??* ();	Constructeurs.
void setNetwork (NFF&);	Permet de sélectionner un réseau.
void sp (const long id, float* a, int* b, int* c);	Calcule les *pccs* à partir d'un noeud origine vers tous les noeuds (toutes les destinations) en considérant les délais aux intersections.
NFF* network () const;	Donne accès à l'objet NFF.
void setCostLink (float*); void setCostTurn (float*);	Permet de sélectionner les coûts sur les liens et les virages.
int numberOfLinks () const; int numberOfNodes () const; int numberOfTurns () const; int numberOfCentroids () const;	Donne accès à l'information concernant le reseau.

[a]Des caractères ?? sont placés pour éviter de nommer chacune des classes concernées.

Tableau 2.IV: Interface partagée par les classes dérivées de la classe FORWARDSPT.

Le premier paramètre correspond à l'identificateur du noeud origine à considérer. Ensuite, afin de pouvoir conserver l'information résultant du calcul, la méthode requiert deux vecteurs de dimension n. Ces vecteurs sont définis de la façon suivante:

- *a[i]:* Coût du chemin reliant l'origine au noeud i;
- *b[i]:* Noeud précédant le noeud i sur le plus court chemin le reliant à l'origine.

Compte tenu que les entiers et les réels à simple précision requièrent un espace mémoire de 4 *bytes* sur le type de machines utilisées, l'espace mémoire total nécessaire pour conserver l'information en question est $2 \times 4 \times n = 8n$ *bytes*.

La méthode $sp(\cdot)$ du tableau 2.IV accepte toutefois quatre paramètres. Le premier paramètre correspond toujours à l'identificateur du noeud origine à considérer. Cependant, afin de pouvoir conserver l'information résultant du calcul, la méthode requiert deux vecteurs de dimensions m ainsi qu'un vecteur de dimension $|\mathcal{C}|$, où $\mathcal{C}$ est l'ensemble des centroïdes. Ces vecteurs sont définis de la façon suivante:

- *a[i]:* Coût du chemin reliant l'origine à la fin du lien i;
- *b[i]:* Noeud précédant le lien i sur le plus court chemin le reliant à l'origine;
- *c[i]:* Lien précédant le centroïde i sur le plus court chemin le reliant à l'origine.

Dans ce cas, l'espace mémoire total nécessaire pour conserver l'information résultant du calcul des plus courts chemins pour un origine est $2 \times 4 \times m + 4 \times |\mathcal{C}| = 8m + 4|\mathcal{C}|$ *bytes.*

L'utilisation d'un outil est relativement simple. En effet, supposons que l'on désire utiliser l'outil *FSPTQUEUE* pour calculer les plus courts chemins pour chaque noeud origine en considérant les virages aux intersections. Alors, le programme *C++* correspondant peut se résumer comme suit:

```
#include <FSPTQUEUE.h>
...
void main ()
{
  ...
  ifstream inputFile (''winnipeg.nff'');

  NFF network;
  inputFile >> network;
  ...
  FSPTQUEUE tool (network);
  int nbLinks = tool.numberOfLinks();
  int nbCentroids = tool.numberOfCentroids();

  float* pathCost = new float*[nbLinks];
  int* previousLink = new int[nbLinks];
  int* firstLink = new int[nbCentroids];

  for (i = 0; i < nbCentroids; i++) {
     tool.sp (i, pathCost, previousLink, firstLink);
     ...
  }
}
```

Dans cet exemple, les méthodes $numberOfLinks()$ et $numberOfCentroids()$ sont utilisées pour connaître respectivement le nombre de liens et le nombre de centroïdes dans le réseau. À l'aide de ces informations, nous sommes en mesure de construire les structures (vecteurs) permettant d'obtenir les résultats du calcul des plus courts chemins. Ensuite, pour chacun des centroïdes i, nous utilisons la méthode $sp(\cdot)$ pour déterminer ces plus courts chemins vers chacun des liens et implicitement, vers chacune des destinations. Il est important de noter que le premier paramètre transmis à la méthode $sp(\cdot)$ doit correspondre à l'identificateur d'un centroïde. Comme on peut le remarquer, ce programme ne conserve pas les résultats du calcul des plus courts chemins. On suppose en fait qu'une opération quelconque est effectuée avec les résultats obtenus pour un centroïde donné avant d'en traiter un nouveau. Certains modèles de planification en transport peuvent toutefois exiger de conserver les résultats des calculs pour chacun des centroïdes considérés. Dans ce cas, une structure plus coûteuse en

espace mémoire devient nécessaire. En effet, pour conserver les résultats des calculs des plus courts chemins pour chaque centroïde, un espace mémoire de $|\mathcal{C}| \times (8m + 4|\mathcal{C}|)$ *bytes* est requis.

Dans ce qui suit, nous discutons des implantations avec *queue*, *2queue*, *deque* et *monceau* de la méthode $sp(\cdot)$ pour les classes dérivant de *FORWARDSP* et de *FORWARDSPT*. Dans chaque cas, nous décrivons la structure de données correspondante et nous donnons les détails importants relatifs à l'implantation traitée. Les codes *C++* correspondant à l'implantation de la structure de données ainsi qu'à l'implantation même de la méthode $sp(\cdot)$ sont également fournis dans chacun des cas. Afin de rendre la lecture des codes *C++* associés aux implantations plus facile, un certain nombre de variables sont définies préalablement [5]. Puisque ces variables sont privées, c'est-à-dire qu'elles sont utilisées uniquement par les méthodes de la classe à laquelle elles appartiennent[6], elles sont précédées du caractère « _ »:

- *_costLink[i]*: coût du lien i;
- *_costTurn[i]*: coût du virage i;
- *_outgoingLinks[i]*: liste des arcs sortant du noeud i;
- *_outgoingTurns[i]*: liste des virages sortant du lien i;
- *_turnsAtNode[i]*: liste des virages présents au noeud i;
- *_centroid[i]*: référence au centroïde correspondant au noeud i (vaut -1 s'il n'existe pas de correspondance);
- *_nodeCentroid[i]*: référence au noeud associé au centroïde i;
- *_bestTripCost[i]*: coût du chemin le plus court reliant l'origine et le noeud (centroïde) i;
- *_fromNode[i]*: référence au noeud origine du lien i;
- *_toNode[i]*: référence au noeud terminal du lien i;
- *_toLink[i]*: référence au lien destination du virage i.

Implantations avec queue

Comme nous l'avons déjà vu au chapitre 1, une *queue* est une structure de données qui permet les opérations d'insertion et de suppression respectivement à la fin et au début de la structure. La figure 2.5 nous permet de voir cette structure de données ainsi que différents états pouvant apparaître lors de son utilisation. Trois états de la *queue* y sont représentés: l'état initial et deux états intermédiaires.

[5] Les définitions correspondent à l'implantation avec recherche par parcours avant.

[6] Pour plus de détails sur ce concept, consulter Stroustrup [53].

Notre implantation de cette structure de données utilise quatre pointeurs destinés à gérer les opérations permises. Les pointeurs **top** et **eoq** (**end-of-queue**) sont constants, c'est-à-dire fixes. Leur fonction est, en quelque sorte, de délimiter l'espace qu'occupe la *queue*. Le pointeur *top* pointe sur le début de la *queue* tandis que le pointeur *eoq* délimite la fin de la *queue* en pointant sur la première case mémoire à l'extérieur de la structure. D'un autre côté, les pointeurs **first** et **end** pointent respectivement sur la case mémoire occupée par le premier élément et sur la première case mémoire venant après celle qu'occupe le dernier élément de la *queue*. Chaque nouvel élément est ajouté dans la case pointée par *end*. Une incrémentation (déplacement d'une case mémoire vers la droite) du pointeur *end* suit chaque opération d'ajout d'un élément et, une fois le pointeur *eoq* atteint, une mise à jour vers le

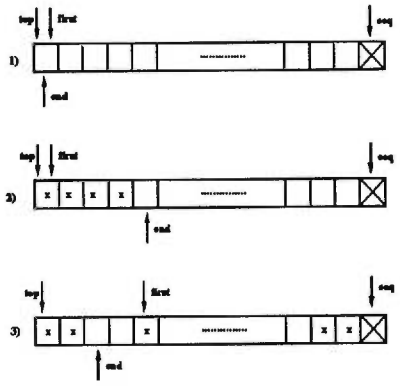

Figure 2.5: Structure de données *queue*.

pointeur *top* est effectuée. Le comportement du pointeur *first* est très semblable à celui du pointeur *end*. Chaque opération de sélection d'un élément provoque une incrémentation du pointeur *first* jusqu'à ce que le pointeur *eoq* soit atteint. Comme précédemment, une mise à jour vers le pointeur *top* est effectuée, permettant ainsi de poursuivre le processus. Évidemment, la *queue* est vide lorsque *first=end*.

Afin d'implanter cette structure de données, la classe paramétrée **QUEUE<T>** a été développée. L'interface de cette classe est donnée dans le tableau 2.V. Le développement d'une classe paramétrée se fait en utilisant le concept de *template* propre au langage C++. Ce concept permet de créer une classe conteneur dont le type des objets qu'elle contient doit être défini préalablement par l'utilisateur. Dans notre cas, il s'agit du type des éléments contenus dans la *queue* (*int*, *long*, *float*, *double*, *char*, etc).

Méthodes	Fonctionnalités
QUEUE<T> (long size = 0);	Constructeur (un type T doit être spécifié) .
void setSize (long size);	Permet de choisir la taille de la structure.
void flush ();	Permet de vider la structure.
T remove ();	Permet de retirer le premier élément de la structure.
void insert (T value);	Permet d'ajouter un élément.
long size () const;	Permet de connaître la taille de la structure.
int empty () const;	Permet de savoir si la structure est vide.

Tableau 2.V: Interface de la classe QUEUE.

```
template <classe T>
class QUEUE {
  long _size, _maxSize;
  T*   _queue;
  T*   _first, _end;
  T*   _top, _eoq;

public:
  ...
  void flush() {
    _first = _end = _top;
    _size = 0;
  }

  void setSize (long size) {
    _maxSize = size;
    _queue = new T[size];
    _top = _queue;
    _eoq = _queue+size-1;
    flush();
  }

  T remove () {
    T value = *_first;
    _size--;
    if (_first == _eoq) {
      _first = _top;
      return value;
    }
    else {
      _first++;
      return value;
    }
  }

  void insert (T value) {
    if (_end == _eoq) {
      *_end = value;
      _end = _top;
      _size++;
    }
    else {
      *_end = value;
      _end++;
      _size++;
    }
  }

  long size () const { return _size; }
  int empty () const { return _first == _end; }
}
```

Étant donné t, le nombre de *bytes* requis par le type T et s la taille de la structure, alors la création d'un objet de la classe $QUEUE<T>$ requiert $28 + st$ *bytes*. (En supposant qu'un pointeur sur un type quelconque nécessite 4 *bytes*.)

A) Implantation de la méthode *FSPQUEUE:: sp (·)*

L'implantation de la méthode *FSPQUEUE:: sp (·)* utilise un objet **queue** de type $QUEUE<float>$. La taille de la *queue* est fixée à $\frac{n}{2}$. Les expérimentations effectuées nous permettent de conclure que cette borne est raisonnable puisque le nombre d'élements présents

dans la *queue* n'est jamais très élevé, comparativement au nombre total de noeuds. Cette valeur nous assure, dans tous les cas, qu'il n'y aura pas de débordement lors de l'exécution. L'espace mémoire occupé par l'objet *queue* est donc $28 + 4\frac{n}{2} = 28 + 2n$ *bytes.*

Afin d'éviter d'insérer un même élement plusieurs fois dans la *queue*, l'implantation associe à chaque noeud i une variable permettant de savoir si ce noeud i est contenu dans la structure. Le tableau $elementOfQ[\cdot]$ permet de conserver, pour chaque noeud i, l'information en question. Pour un noeud i, nous avons que

$$elementOfQ[i] = \begin{cases} 1, si\ i \in Q; \\ 0, sinon. \end{cases}$$

```
void FSPQUEUE:: sp (const long origin,
     float* bestTripCost, int* nextLink)
{
  // Initialisation
  for (int i = 0; i < nbNodes; i++ ) {
    nextNode[i] = -1;
    bestTripCost[i] = INFINITY;
    _elementOfQ[i] = 0;
  }

  bestTripCost[origin] = 0;

  long link;
  long size = _outgoingLinks[origin].size();
  for (i = 0; i < size; i++) {
    link = _outgoingLinks[origin][i];
    int node = _toNode[link];
    bestTripCost[node] = _costLink[link];
    nextNode[node] = origin;
    _elementOfQ[node] = 1;
    _queue->insert (node);
  }

  // Etape principale
  while (!_queue->empty()) {
    long current = _queue->remove();
    _elementOfQ[current] = 0;
    float bestCost = bestTripCost[current];

    long nextLink;
    float newCost;

    int size = _outgoingLinks[current].size();
    for (i = 0; i < size; i++) {

      nextLink = _outgoingLinks[current][i];
      int candidat = _toNode[nextLink];
      newCost = bestCost+_costLink[nextLink];

      if (newCost < bestTripCost[candidat]) {
        bestTripCost[candidat] = newCost;
        nextNode[candidat] = current;

        if (!_elementOfQ[candidat]) {
          _queue->insert (candidat);
          _elementOfQ[candidat] = 1;
        }
      }
    }
  }
}
```

B) Implantation de la méthode *FSPTQUEUE:: sp (·)*

L'implantation de la méthode *FSPTQUEUE:: sp (·)* utilise également un objet queue de type *QUEUE<float>*. La taille de la *queue* est toutefois fixée à $\frac{m}{2}$ puisque les éléments à insérer dans la structure sont, dans ce cas, des identificateurs de liens. L'espace mémoire occupé par l'objet *queue* est donc $28 + 4\frac{m}{2} = 28 + 2m$ *bytes.*

Le tableau $elementOfQ[\cdot]$ permet, dans cette implantation, d'associer à chaque lien i une valeur permettant de savoir si ce lien i est contenu dans la structure. La définition formelle de la variable est cependant la même que précédemment.

```
void FSPTQUEUE:: sp (const long origin,
     float* pathCost, int* nextLink, int* firstLink)
{
  // Initialisation
  for (int i = 0; i < nbLinks; i++) {
    _elementOfQ[i] = 0;
    pathCost[i] = INFINITY;
    nextLink[i] = -1;
  }

  for (i = 0; i < nbNodes; i++ ) {
    firstLink[i] = -1;
    _bestTripCost[i] = INFINITY;
  }

  _bestTripCost[origin] = 0;
  int nodeOrigin = _nodeCentroid[origin];

  long link;
  long size = _outgoingLinks[nodeOrigin].size();
  for (i = 0; i < size; i++) {
    link = _outgoingLinks[nodeOrigin][i];
    pathCost[link] = _costLink[link];
    nextLink[link] = -1;
    _elementOfQ[link] = 1;
    _queue->insert (link);
  }

  // Etape principale
  while (!_queue->empty()) {

    long link = _queue->remove();
    _elementOfQ[link] = 0;
    float costMin = pathCost[link];
    long toNode = _toNode[link];
    long centroid = _centroid[toNode];

    if (centroid != -1) {

      if (costMin < _bestTripCost[centroid]) {
        _bestTripCost[centroid] = costMin;
        firstLink[centroid] = selectedLink;
      }
    }
    else {

      long next;
      float newCost;
      if (_turnsAtNode[toNode].empty()) {

        // Le noeud terminal est un noeud regulier

        int size = _outgoingLinks[toNode].size();
        for (i = 0; i < size; i++) {

          next = _outgoingLinks[toNode][i];

          // Virages en U interdits
          if (_toNode[nextLink] == _fromNode[link])
            continue;

          newCost = costMin+_costLink[next];
          if (newCost < pathCost[next]) {
            pathCost[next] = newCost;
            nextLink[next] = link;

            if (!_elementOfQ[next]) {
              _queue->insert (next);
              _elementOfQ[next] = 1;
            }
          }
        }
      }
      else {

        // Le noeud terminal est une intersection

        long outgoingTurn;
        int size = _outgoingTurns[link].size();
        for (i = 0; i < size; i++) {

          outgoingTurn = _outgoingTurns[link][i];
          next = _toLink[outgoingTurn];

          newCost = costMIn+
          _costTurn[outgoingTurn]+
          _costLink[next];
          if (newCost < pathCost[next]) {
            pathCost[next] = newCost;
            nextLink[next] = link;
```

```
        if (!_elementOfQ[next]) {
          _queue->insert (next);
          _elementOfQ[next] = 1;
        }
      }
                                          }
                                        }
                                      }
                                    }
                                  }
```

Implantations avec 2queue

Les implantations suivantes utilisent une structure de données légèrement plus complexe que la précédente. Toutefois, l'implantation de cette structure est relativement simple maintenant que nous avons la classe *QUEUE<T>* implantée. En effet, la structure de données *2queue* est simplement constituée d'une combinaison de deux *queues*. La figure 2.6 permet de voir une représentation de la structure et des pointeurs utilisés pour la gérer. Les

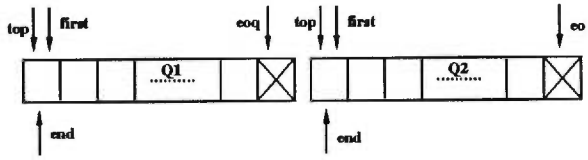

Figure 2.6: Structure de données *2queue*.

pointeurs associés aux deux *queues* (Q1 et Q2) se comportent exactement comme dans l'exemple de la figure 2.5. Comme dans le cas de la structure *queue*, nous développons une classe paramétrée **2QUEUE<T>**. Cette classe utilise directement deux objets de la classe *QUEUE<T>*.

```
template <classe T>
class 2QUEUE {

   QUEUE<T> _queue1;
   QUEUE<T> _queue2;

 public:
   ...

   void flush() {

     _queue1->flush();
     _queue2->flush();
   }

   void setSize (long size) {
     _queue1->setSize (size);
     _queue1->setSize (size);
     flush();
   }

   T remove () {

     if (!_queue1->empty())
       return _queue1->remove();
     else
       return _queue2->remove();
   }

   void insertInQ1 (T value) {
```

```
  _queue1->insert (value);
}

void insertInQ2 (T value) {

  _queue2->insert (value);
}

long size () const {
  return _queue1->size()+_queue2->size();
}

int empty () const {
  return _queue1->empty() && _queue2->empty();
}
}
```

Étant donné t, le nombre de *bytes* requis par le type **T** et s la taille de la structure, alors le nombre de *bytes* nécessaires à la création d'un objet de la classe *2QUEUE* est $8 + 2(28 + st)$. L'interface de la classe *2QUEUE<T>* est résumée dans le tableau 2.VI.

Méthodes	Fonctionnalités
2QUEUE<T> (long size = 0);	Constructeur (un type T doit être spécifié).
void setSize (long size);	Permet de choisir la taille de la structure.
void flush ();	Permet de vider la structure.
T remove ();	Permet de retirer le premier élément de la structure.
void insertInQ1 (T value); void insertInQ2 (T value);	Permettent d'ajouter un élément (dans Q1 ou Q2).
long size () const;	Permet de connaître la taille de la structure.
int empty () const;	Permet de savoir si la structure est vide.

Tableau 2.VI: Interface de la classe 2QUEUE.

A) Implantation de la méthode *FSP2QUEUE:: sp (·)*

L'implantation de la méthode *FSP2QUEUE:: sp (·)* utilise un objet **2queue** de type *2QUEUE<float>*. Comme dans le cas de l'implantation avec *queue*, la taille de la *2queue* est fixée à $\frac{n}{2}$. Ce qui implique que l'objet *2queue* utilise un espace mémoire de $8 + 2(28 + 2n) = 64 + 4n$ *bytes*. L'insertion des éléments dans la structure est également gérée par l'état d'une variable ($elementOfQ[i]$) associée à chaque noeud. Toutefois, ce type d'implantation implique une stratégie particulière pour l'insertion des élements dans la structure. Par conséquent, la signification de la variable $elementOfQ[i]$ est légèrement différente. Pour être en mesure de déterminer dans laquelle des deux queues le noeud i doit être inséré, il faut savoir s'il a déjà appartenu ou s'il appartient présentement à la structure. On définit donc la variable $elementOfQ[i]$ de la façon suivante:

$$elementOfQ[i] = \begin{cases} 1, si\ i \in Q; \\ -1, si\ i\ a\ déjà\ été\ dans\ Q; \\ 0, sinon. \end{cases}$$

```
void FSP2QUEUE:: sp (const long origin,
     float* bestTripCost, int* nextLink)
{
  // Initialisation
  for (int i = 0; i < nbNodes; i++ ) {
    nextNode[i] = -1;
    bestTripCost[i] = INFINITY;
    _elementOfQ[i] = 0;
  }

  bestTripCost[origin] = 0;

  long link;
  long size = _outgoingLinks[origin].size();
  for (i = 0; i < size; i++) {
    link = _outgoingLinks[origin][i];
    int node = _toNode[link];
    bestTripCost[node] = _costLink[link];
    nextNode[node] = origin;
    _elementOfQ[node] = 1;
    _2queue->insertInQ2 (node);
  }

  // Etape principale
  while (!_2queue->empty()) {

    long current = _2queue->remove();
    _elementOfQ[current] = -1;
    float bestCost = bestTripCost[current];

    long nextLink;
    float newCost;

    int size = _outgoingLinks[current].size();
    for (i = 0; i < size; i++) {

      nextLink = _outgoingLinks[current][i];
      int candidat = _toNode[nextLink];
      newCost = bestCost+_costLink[nextLink];

      if (newCost < bestTripCost[candidat]) {
        bestTripCost[candidat] = newCost;
        nextNode[candidat] = current;

        if (_elementOfQ[candidat] == -1) {
          _2queue->insertInQ1 (candidat);
          _elementOfQ[candidat] = 1;
        }
        else if (!_elementOfQ[candidat]) {
          _2queue->insertInQ2 (candidat);
          _elementOfQ[candidat] = 1;
        }
      }
    }
  }
}
```

B) Implantation de la méthode *FSPT2QUEUE:: sp (·)*

L'implantation de la méthode *FSPT2QUEUE:: sp (·)* utilise également un objet **2queue** de type *2QUEUE<float>*. Comme les éléments à insérer dans la structure sont maintenant des identificateurs de liens, la taille de la *2queue* est fixée à $\frac{m}{2}$. L'espace mémoire occupé par l'objet **2queue** est alors $8 + 2(28 + 2m) = 64 + 4m$ *bytes*. La définition de la variable $elementOfQ[i]$ est la même que pour l'implantation précédente. Bien entendu, cette variable permet, dans cette implantation, d'associer à chaque lien i une valeur permettant de savoir si ce lien i est contenu dans la structure.

```
void FSPT2QUEUE:: sp (const long origin,
     float* pathCost, int* nextLink, int* firstLink)
{
  // Initialisation

  for (int i = 0; i < nbLinks; i++) {
    _elementOfQ[i] = 0;
    pathCost[i] = INFINITY;
    nextLink[i] = -1;
  }

  for (i = 0; i < nbNodes; i++ ) {
    firstLink[i] = -1;
    _bestTripCost[i] = INFINITY;
  }
```

```
_bestTripCost[origin] = 0;
int nodeOrigin = _nodeCentroid[origin];

long link;
long size = _outgoingLinks[nodeOrigin].size();
for (i = 0; i < size; i++) {
  link = _outgoingLinks[nodeOrigin][i];
  pathCost[link] = _costLink[link];
  nextLink[link] = -1;
  _elementOfQ[link] = 1;
  _2queue->insertInQ2 (link);
}

// Etape principale
while (!_2queue->empty()) {

  long link = _2queue->remove();
  _elementOfQ[link] = -1;
  float costMin = pathCost[link];

  long toNode = _toNode[link];
  long centroid = _centroid[toNode];

  if (centroid != -1) {

    if (costMin < _bestTripCost[centroid]) {
      _bestTripCost[centroid] = costMin;
      firstLink[centroid] = selectedLink;
    }
  }
  else {

    long next;
    float newCost;

    if (_turnsAtNode[toNode].empty()) {

      // Le noeud terminal est un noeud regulier

      int size = _outgoingLinks[toNode].size();
      for (i = 0; i < size; i++) {

        next = _outgoingLinks[toNode][i];

        // Virages en U interdits
        if (_toNode[nextLink] == _fromNode[link])
          continue;

        newCost = costMin+_costLink[next];

        if (newCost < pathCost[next]) {
          pathCost[next] = newCost;
          nextLink[next] = link;

          if (_elementOfQ[next] == -1) {
            _2queue->insertInQ1 (next);
            _elementOfQ[next] = 1;
          }
          else if (!_elementOfQ[next]) {
            _2queue->insertInQ2 (next);
            _elementOfQ[next] = 1;
          }
        }
      }
    }
    else {

      // Le noeud terminal est une intersection

      long outgoingTurn;
      int size = _outgoingTurns[link].size();
      for (i = 0; i < size; i++) {

        outgoingTurn = _outgoingTurns[link][i];
        next = _toLink[outgoingTurn];

        newCost = costMIn+
        _costTurn[outgoingTurn]+
        _costLink[next];

        if (newCost < pathCost[next]) {
          pathCost[next] = newCost;
          nextLink[next] = link;

          if (_elementOfQ[next] == -1) {
            _2queue->insertInQ1 (next);
            _elementOfQ[next] = 1;
          }
          else if (!_elementOfQ[next]) {
            _2queue->insertInQ2 (next);
            _elementOfQ[next] = 1;
          }
        }
      }
    }
  }
}
```

Implantations avec deque

La structure de données utilisée pour ces implantations est représentée dans la figure 2.7. Contrairement à la structure de données *2queue* qui est formée de deux *queues*, une *deque* est constituée d'une *queue* et d'une *pile*. Cependant, en pratique, l'implantation de cette

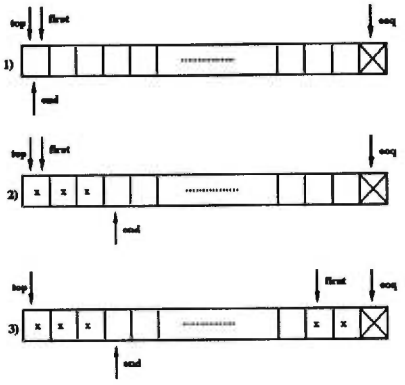

Figure 2.7: Structure de données *deque*.

structure se fait sans distinction entre les deux sous-structures. La figure 2.7 nous montre trois états de la *deque* ainsi que les positions des pointeurs pour chacun de ces états. Le comportement des pointeurs est le même que celui décrit pour la structure *queue*. Cette structure offre toutefois la possibilité d'insérer des éléments au début de la *deque*. Pour cette opération d'insertion, le pointeur *first* doit subir une décrémentation (déplacement d'une case mémoire vers la gauche) et, dans le cas où le pointeur *top* est rencontré une mise à jour vers le pointeur *eoq* doit être effectuée. Le nouvel élément est alors inséré dans la case mémoire pointée par le pointeur *first*.

Pour l'implantation de cette structure de données, la classe paramétrée **DEQUE<T>** a été développée. L'interface de cette classe est résumée dans le tableau 2.VII.

```
template <classe T>
class DEQUE {

   long _size, _maxSize;
   T*   _deque;
   T*   _first, _end;
   T*   _top, _eoq;

 public:
   ...
```

```
void flush() {
  _first = _end = _top;
  _size = 0;
}

void setSize (long size) {
  _maxSize = size;
  _deque = new T[size];
  _top = _deque;
  _eoq = _deque+size-1;
  flush();
```

```
}

T remove () {
  T value = *_first;
  _size--;
  if (_first == _eoq) {
    _first = _top;
    return value;
  }
  else {
    _first++;
    return value;
  }
}

void insertAtEnd (T value) {
  if (_end == _eoq) {
    *_end = value;
    _end = _top;
    _size++;
  }
  else {
    *_end = value;
    _end++;
    _size++;
  }
}

void insertAtFront (T value) {
  if (_first == _top) {
    *_first = _eoq;
    *_first = value;
    _size++;
  }
  else {
    _first--;
    *_first = value;
    _size++;
  }
}

long size () const { return _size; }
int empty () const { return _first == _end; }
}
```

Le nombre de *bytes* nécessaires à la création d'un objet de la classe *DEQUE* est le même que dans le cas de la classe *QUEUE*.

Méthodes	Fonctionnalités
DEQUE<T> (long size = 0);	Constructeur (un type **T** doit être spécifié).
void setSize (long size);	Permet de choisir la taille de la structure.
void flush ();	Permet de vider la structure.
T remove ();	Permet de retirer le premier élément de la structure.
void insertAtEnd (T value); void insertAtFront (T value);	Permettent d'ajouter un élément. (à la fin ou au début de la structure)
long size () const;	Permet de connaître la taille de la structure.
int empty () const;	Permet de savoir si la structure est vide.

Tableau 2.VII: Interface de la classe DEQUE.

A) Implantation de la méthode *FSPDEQUE:: sp (·)*

L'implantation de la méthode *FSPDEQUE:: sp (·)* utilise un objet **deque** de type *DEQUE<float>*. Comme précédemment, nous fixons la taille de la *deque* à $\frac{n}{2}$. L'espace mémoire occupé est donc $28+4\frac{n}{2} = 28+2n$ *bytes*. La définition de la variable $elementOfQ[i]$ est la même que dans le cas de l'implantation avec *2queue*. De cette façon, nous sommes en mesure de déterminer si l'insertion d'un noeud i doit se faire au début ou à la fin de la *deque*.

```
void FSPDEQUE:: sp (const long origin,
     float* bestTripCost, int* nextLink)
{
  // Initialisation
  for (int i = 0; i < nbNodes; i++ ) {
    nextNode[i] = -1;
    bestTripCost[i] = INFINITY;
    _elementOfQ[i] = 0;
  }

  bestTripCost[origin] = 0;

  long link;
  long size = _outgoingLinks[origin].size();
  for (i = 0; i < size; i++) {
    link = _outgoingLinks[origin][i];
    int node = _toNode[link];
    bestTripCost[node] = _costLink[link];
    nextNode[node] = origin;
    _elementOfQ[node] = 1;
    _deque->insertAtEnd (node);
  }

  // Etape principale
  while (!_deque->empty()) {

    long current = _deque->remove();
    _elementOfQ[current] = -1;
    float bestCost = bestTripCost[current];

    long nextLink;
    float newCost;

    int size = _outgoingLinks[current].size();
    for (i = 0; i < size; i++) {

      nextLink = _outgoingLinks[current][i];
      int candidat = _toNode[nextLink];
      newCost = bestCost+_costLink[nextLink];

      if (newCost < bestTripCost[candidat]) {
        bestTripCost[candidat] = newCost;
        nextNode[candidat] = current;

        if (_elementOfQ[candidat] == -1) {
          _deque->insertAtFront (candidat);
          _elementOfQ[candidat] = 1;
        }
        else if (!_elementOfQ[candidat]) {
          _deque->insertAtEnd (candidat);
          _elementOfQ[candidat] = 1;
        }
      }
    }
  }
}
```

B) Implantation de la méthode *FSPTDEQUE:: sp (·)*

Pour l'implantation de la méthode *FSPTDEQUE:: sp (·)*, nous utilisons également un objet **deque** de type *DEQUE<float>*. En fixant la taille de la *deque* à $\frac{m}{2}$, l'espace mémoire requis par l'objet *deque* est $28 + 4\frac{m}{2} = 28 + 2m$ *bytes.*

```
void FSPTDEQUE:: sp (const long origin,
     float* pathCost, int* nextLink, int* firstLink)
{
  // Initialisation

  for (int i = 0; i < nbLinks; i++) {
    _elementOfQ[i] = 0;
    pathCost[i] = INFINITY;
    nextLink[i] = -1;
  }

  for (i = 0; i < nbNodes; i++ ) {
    firstLink[i] = -1;
    _bestTripCost[i] = INFINITY;
  }

  _bestTripCost[origin] = 0;
  int nodeOrigin = _nodeCentroid[origin];

  long link;
  long size = _outgoingLinks[nodeOrigin].size();
  for (i = 0; i < size; i++) {
    link = _outgoingLinks[nodeOrigin][i];
    pathCost[link] = _costLink[link];
    nextLink[link] = -1;
    _elementOfQ[link] = 1;
```

```
  _deque->insertAtEnd (link);
}

// Etape principale
while (!_deque->empty()) {

  long link = _deque->remove();
  _elementOfQ[link] = -1;
  float costMin = pathCost[link];

  long toNode = _toNode[link];
  long centroid = _centroid[toNode];

  if (centroid != -1) {

    if (costMin < _bestTripCost[centroid]) {
      _bestTripCost[centroid] = costMin;
      firstLink[centroid] = selectedLink;
    }
  }
  else {

    long next;
    float newCost;

    if (_turnsAtNode[toNode].empty()) {

      // Le noeud terminal est un noeud regulier

      int size = _outgoingLinks[toNode].size();
      for (i = 0; i < size; i++) {

        next = _outgoingLinks[toNode][i];

        // Virages en U interdits
        if (_toNode[nextLink] == _fromNode[link])
          continue;

        newCost = costMin+_costLink[next];

        if (newCost < pathCost[next]) {
          pathCost[next] = newCost;
          nextLink[next] = link;

          if (_elementOfQ[next] == -1) {
            _deque->insertAtFront (next);
            _elementOfQ[next] = 1;
          }
          else if (!_elementOfQ[next]) {
            _deque->insertAtEnd (next);
            _elementOfQ[next] = 1;
          }
        }
      }
    }
    else {

      // Le noeud terminal est une intersection

      long outgoingTurn;
      int size = _outgoingTurns[link].size();
      for (i = 0; i < size; i++) {

        outgoingTurn = _outgoingTurns[link][i];
        next = _toLink[outgoingTurn];

        newCost = costMIn+
        _costTurn[outgoingTurn]+
        _costLink[next];

        if (newCost < pathCost[next]) {
          pathCost[next] = newCost;
          nextLink[next] = link;

          if (_elementOfQ[next] == -1) {
            _deque->insertAtFront (next);
            _elementOfQ[next] = 1;
          }
          else if (!_elementOfQ[next]) {
            _deque->insertAtEnd (next);
            _elementOfQ[next] = 1;
          }
        }
      }
    }
  }
}
```

Implantations avec monceau

Un *monceau* est une structure de données plus complexe que celles vues précédemment. L'intérêt d'une implantation avec monceau réside dans la possibilité d'obtenir un outil de calcul des plus courts chemins utilisant une technique d'*extensions sélectives*[7]. Le monceau utilisé a été implanté de façon à être le plus efficace possible dans le genre d'application traité dans ce travail[8]. Le monceau est représenté en mémoire comme un tableau dont les entrées comportent les champs suivants: **data**, **parent**, **child1** et **child2**. La figure 2.8 permet de voir une représentation partielle du tableau correspondant à la structure. Chaque case mémoire du tableau correspond en fait à un noeud de l'arbre binaire associé

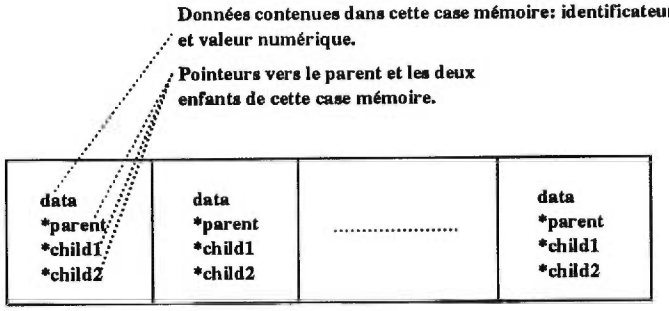

Figure 2.8: Structure de données *monceau.*

à la représentation arborescente du monceau. L'efficacité des opérations sur cette structure de données peut être améliorée d'abord en fixant la taille de la structure et, ensuite, en précalculant la hiérarchie implicite existant entre les entrées du tableau. L'interface de cette classe est résumée dans le tableau 2.VIII. Elle permet les opérations standards associées à la structure de données implantée, en plus de quelques fonctionnalités additionnelles. Étant donné s la taille de la structure, alors l'espace mémoire nécessaire à la création d'un objet de la classe **HEAP** est $40 + 16s$ *bytes.*

Dans nos implantations avec monceau, la variable *elementOfQ[i]* a une définition différente que dans celles avec *queue*, *2queue* et *deque.* Dans le cas présent, la définition fait plutôt référence à l'état de l'étiquette associée au noeud (ou au lien) i:

$$elementOfQ[i] = \begin{cases} 1, si\ i\ a\ une\ étiquette\ permanente; \\ 0, sinon. \end{cases}$$

[7]Voir section 1.1.4.

[8]L'implantation du monceau a été réalisée par Éric Le Saux dans le cadre du développement de la librairie *NFF*: *http* : *//www.crt.umontreal.ca/~lab_sit/DOC − NFF.*

Méthodes	Fonctionnalités
HEAP (long size);	Constructeur.
void setSize (long size);	Permet de choisir la taille de la structure.
void flush ();	Permet de vider la structure.
long top ();	Permet de retirer le premier élément de la structure.
void add (long id, double value);	Permet d'ajouter un élément.
double minValue () const;	Permet de connaître la valeur minimum de la structure.
long size () const;	Permet de connaître la taille de la structure.
int empty () const;	Permet de savoir si la structure est vide.

Tableau 2.VIII: Interface de la classe HEAP.

De cette manière, on s'assure qu'un noeud (lien) possédant une étiquette permanente n'est plus visité par la suite.

A) Implantation de la méthode *FSPHEAP:: sp (·)*

L'implantation de la méthode *FSPHEAP:: sp (·)* utilise évidemment un objet **heap** de la classe *HEAP*. Comme mentionné précédemment, la taille du monceau est fixe. Nous choisissons $s = \frac{n}{2}$ afin d'assurer une exécution sans débordement. L'espace mémoire requis par l'objet *heap* est donc dans ce cas $40 + 16\frac{n}{2} = 40 + 8n$ *bytes*.

```
void FSPHEAP:: sp (const long origin,
     float* bestTripCost, int* nextNode)
{
  // initialisation
  for (int i = 0; i < _nbNodes; i++ ) {
    nextNode[i] = -1;
    bestTripCost[i] = INFINITY;
    _elementOfQ[i] = 0;
  }

  bestTripCost[origin] = 0;

  long link;
  long size = _outgoingLinks[origin].size();
  for (i = 0; i < size; i++) {
    link = _outgoingLinks[origin][i];
    int node = _toNode[link];
    bestTripCost[node] = _costLink[link];
    nextNode[node] = origin;
    _heap->add (node, bestTripCost[node]);
  }

  // Etape principale

  while (!_heap->empty()) {
    long node = _heap->top();
    if (_elementOfQ[node])
      continue;

    float bestCost = bestTripCost[node];
    long nextLink;
    float newCost;

    int size = _outgoingLinks[node].size();
    for (i = 0; i < size; i++) {

      nextLink = _outgoingLinks[node][i];
      int candidat = _toNode[nextLink];
      if (_elementOfQ[candidat])
        continue;

      newCost = bestCost+_costLink[nextLink];

      if (newCost < bestTripCost[candidat]) {
        bestTripCost[candidat] = newCost;
        nextNode[candidat] = node;
        _heap->add (candidat, newCost);
      }
    }
```

```
    _elementOfQ[node] = 1;
                                        }
                                      }
```

B) Implantation de la méthode *FSPTHEAP:: sp (·)*

Comme dans l'implantation précédente, nous utilisons un objet **heap** de la classe *HEAP*. La taille du *monceau* est toutefois fixée à $\frac{m}{2}$, impliquant ainsi l'utilisation d'un espace mémoire de $40 + 8m$ *bytes*.

```
void FSPTHEAP:: sp (const long origin,
     float* pathCost, int* nextLink, int* firstLink)
{
  // Initialisation

  for (int  i = 0; i < _nbLinks; i++) {
    _elementOfQ[i] = 0;
    pathCost[i] = INFINITY;
    nextLink[i] = -1;
  }

  for ( i = 0; i < _nbCentroids; i++ )
    firstLink[i] = -1;

  int nodeOrigin = _nodeCentroid[origin];
  int size = _incomingLinks[nodeOrigin].size();
  for (i = 0; i < size; i++)
    _elementOfQ[_incomingLinks[nodeOrigin][i]] = 1;

  long link;
  size = _outgoingLinks[nodeOrigin].size();
  for (i = 0; i < size; i++) {
    link = _outgoingLinks[nodeOrigin][i];
    pathCost[link] = _costLink[link];
    nextLink[link] = -1;
    _heap->add (link, pathCost[link]);
  }

  // Etape principale

  while (!_heap->empty()) {

    long link = _heap->top();

    if (_elementOfQ[link])
      continue;

    float costMin = pathCost[link];
    long toLink = _toNode[link];
    long centroid = _centroid[endLink];

    if (centroid != -1) {

      if (firstLink[centroid] == -1)
        firstLink[centroid] = linkMin;
    }
    else {

      long next;
      float newCost;

      if (_turnsAtNode[endLink].empty()) {

        // Le noeud terminal est un noeud regulier

        int size = _outgoingLinks[endLink].size();
        for (i = 0; i < size; i++) {

          next = _outgoingLinks[endLink][i];
          if (_elementOfQ[nextLink])
            continue;

          // Virages en U interdits
          if (_toNode[nextLink] == _fromNode[link])
            continue;

          newCost = costMin+_costLink[next];

          if (newCost < pathCost[next]) {
            pathCost[next] = newCost;
            nextLink[next] = link;
            _heap->add (next, newCost);
          }
        }
      }
      else {

        // Le noeud terminal est une intersection
```

```
long outgoingTurn;
int size = _outgoingTurns[link].size();
for (i = 0; i < size; i++) {

  outgoingTurn = _outgoingTurns[link][i];
  next = _toLink[outgoingTurn];

  if (_elementOfQ[nextLink])
    continue;

  newCost = costMin+_costTurn[outgoingTurn]+
  _costLink[next];
```

```
    if (newCost < pathCost[next]) {
      pathCost[next] = newCost;
      nextLink[next] = link;
      _heap->add (next, newCost);
    }
  }
 }
}

_elementOfQ[link] = 1;
}
```

Implantation avec monceau pour le calcul des plus courts chemins reliant un noeud origine à un sous-ensemble de noeuds destinations

L'implantation que l'on présente dans cette partie est conçue spécialement pour le calcul des plus courts chemins reliant un noeud origine à un sous-ensemble de noeuds destinations. Il s'agit en fait d'une spécialisation de l'implantation avec monceau vue précédemment. L'intérêt d'utiliser une implantation appliquant une technique d'*extensions sélectives* pour ce problème particulier vient du fait qu'il est possible d'arrêter de calculer les plus courts chemins dès que toutes les destinations sont atteintes. Une implantation avec *queue* ou toute autre implantation utilisant une technique d'*ajustements progressifs*, n'offrent pas cette possiblité.

Dans cette implantation, nous avons la possibilité, comme l'algorithme proposé par Spiess le montre, de limiter notre exploration du réseau. Il s'agit en fait d'éviter certains liens inintéressants. Afin de pouvoir éliminer ces liens, notre implantation utilise une technique qui diffère de celle proposée par Spiess. Dans un premier temps, notons que lorsqu'un noeud destination (centroïde) est atteint pour la première fois, nous obtenons le plus court chemin le reliant à l'origine. Dès lors, nous sommes en mesure d'éliminer (à l'aide de la variable *elementOfQ[·]*) les autres liens entrant à cette destination. Il est également possible de procéder à la même opération lorsque nous atteignons un noeud régulier. Cependant, ce cas est légèrement plus complexe car il faut d'abord s'assurer qu'il n'existe pas de mouvement en U au noeud en question.

Comme dans l'implantation précédente, nous utilisons, pour l'implantation de la méthode *FSPTHEAPO:: sp (·)*, un objet **_heap** de la classe *HEAP*. Étant donné que l'on fixe la taille à $\frac{m}{2}$, l'espace mémoire requis par l'objet *heap* est $40 + 8m$ *bytes*.

```
void FSPTHEAP0:: sp (const long origin,
     float* pathCost, int* nextLink,
     int* firstLink)
{
  // Initialisation

  for (int  i = 0; i < _nbLinks; i++) {
    _elementOfQ[i] = 0;
    pathCost[i] = INFINITY;
    nextLink[i] = -;
  }

  for ( i = 0; i < _nbCentroids; i++ )
    firstLink[i] = -1;

  int node0 = _nodeCentroid[origin];
  int size = _incomingLinks[ node0].size();
  for (i = 0; i < size; i++)
    _elementOfQ[_incomingLinks[ node0][i]] = 1;

  long link;
  size = _outgoingLinks[ node0].size();
  for (i = 0; i < size; i++) {
    link = _outgoingLinks[ node0][i];
    pathCost[link] = _costLink[link];
    nextLink[link] = -1;
    _heap->add (link, pathCost[link]);
  }

  // Etape principale

  int nbCentroids = _nbCentroids-1;

  while (!_heap->empty()) {

    long link = _heap->top();

    if (_elementOfQ[link])
      continue;

    float costMin = pathCost[link];
    long node = _toNode[link];
    long centroid = _centroid[node];

    if (centroid != -1) {

      _bestTripCost[centroid] = costMin;
      _firstLink[centroid] = linkMin;

      if (!(--nbCentroids))
        break;

    size = _incomingLinks[node].size();
    for (i = 0; i < size; i++)
      _elementOfS[_incomingLinks[node][i]] = 1;
  }
  else {

    long next;
    float newCost;

    if (_turnsAtNode[node].empty()) {

      // Le noeud terminal est un noeud regulier

      int reverseLink = 0;

      int size = _outgoingLinks[node].size();
      for (i = 0; i < size; i++) {

        next = _outgoingLinks[node][i];
        if (_elementOfQ[next])
          continue;

        // Virages en U interdits
        if (_toNode[next] == _fromNode[link]) {
          reverseLink = 1;
          continue;
        }

        newCost = costMin+_costLink[next];

        if (newCost < pathCost[next]) {
          pathCost[next] = newCost;
          nextLink[next] = link;
          _heap->add (next, newCost);
        }
      }

      // Les liens entrant dans ce noeud
      // sont elimines
      if (!reverseLink) {
        size = _incomingLinks[node].size();
        for (int j = 0; j < size; j++)
          _elementOfS[_incomingLinks[node][j]] = 1;
      }
    }
    else {

      // Le noeud terminal est une intersection
```

```
long outgoingTurn;
int size = _outgoingTurns[link].size();
for (i = 0; i < size; i++) {

  outgoingTurn = _outgoingTurns[link][i];
  next = _toLink[outgoingTurn];

  if (_elementOfQ[next])
    continue;

  newCost = costMin+
  _costTurn[outgoingTurn]+_costLink[next];
                                              if (newCost < pathCost[next]) {
                                                pathCost[next] = newCost;
                                                nextLink[next] = link;
                                                _heap->add (next, newCost);
                                              }
                                            }
                                          }
                                        }

                                        _elementOfQ[link] = 1;
                                      }
                                    }
```

2.2.3 Expérimentations et analyse des résultats

Cette partie est consacrée à l'analyse des performances des implantations séquentielles développées dans cette section. Afin de tester ces implantations, nous avons utilisé les réseaux urbains des villes canadiennes de Winnipeg, Ottawa et Montréal[9]. Les résultats présentés ont été obtenus sur une station de travail *SUN SPARC Ultra 1/140.*

Les tableaux 2.IX et 2.X nous montrent les temps de calcul obtenus par les différentes implantations traitées dans cette section[10]. Dans chaque cas, on retrouve le temps de calcul global correspondant au traitement de toutes les origines et, entre parenthèses, le temps de calcul moyen par origine. Le temps mesuré est le temps réel pris par l'application et il est exprimé en secondes. (Dans le cas présent, le temps réel nous donne une mesure valide puisque les tests sont effectués lorsque le processeur est entièrement dédié à notre tâche. La mesure obtenue correspond en fait au temps utilisé par le processeur.) En premier lieu, le tableau 2.IX nous permet de constater que les implantations *FSP2QUEUE* et *FSPDEQUE* se comportent similairement et sont les plus efficaces parmi les quatre implantations comparées. (Ce résultat a déjà été vérifié plusieurs fois dans des travaux antérieurs (voir entre autre [30, 28, 27]).) L'implantation *FSPHEAP* est, pour les trois réseaux, environ 1.5 fois plus lente que les implantations *FSP2QUEUE* et *FSPDEQUE.* On remarque également que l'implantation *FSPQUEUE* est 1.7, 2.1 et 2.7 fois plus lente que les implantations *FSP2QUEUE* et *FSPDEQUE* pour les réseaux de Winnipeg, Ottawa et Montréal respectivement.

En ce qui concerne les performances des implantations considérant les mouvements aux intersections, nous pouvons remarquer dans le tableau 2.X que l'implantation *FSPT-*

[9]Les caractéristiques de ces réseaux sont fournies à la section 2.1.

[10]Les résultats obtenus par les implantations effectuant une recherche par parcours arrière ne sont pas présentés, étant donné la similitude des résultats.

Ville	FSPQUEUE	FSP2QUEUE	FSPDEQUE	FSPHEAP
Winnipeg	0.54 (0.004)	0.32 (0.002)	0.32 (0.002)	0.40 (0.003)
Ottawa	2.55 (0.010)	1.19 (0.005)	1.13 (0.004)	2.07 (0.008)
Montréal	36.12 (0.051)	13.27 (0.019)	13.68 (0.020)	17.34 (0.025)

Tableau 2.IX: Temps d'exécution des implantations de type *FSP*.

2QUEUE est la plus efficace parmi celles calculant les plus courts chemins vers tous les liens du réseau. On constate également que l'implantation *FSPTDEQUE* n'est pas aussi

Ville	FSPTQUEUE	FSPT2QUEUE	FSPTDEQUE	FSPTHEAP	*FSPTSHEAPO*
Winnipeg	1.30 (0.008)	0.83 (0.005)	2.94 (0.019)	1.03 (0.007)	*0.79* *(0.005)*
Ottawa	6.89 (0.027)	4.93 (0.019)	9.16 (0.036)	6.22 (0.024)	*4.22* *(0.016)*
Montréal	115.97 (0.166)	41.72 (0.060)	61.45 (0.088)	58.55 (0.084)	*36.92* *(0.053)*

Tableau 2.X: Temps d'exécution des implantations de type *FSPT*.

performante que précédemment. En fait, cette implantation performe moins bien que l'implantation *FSPTQUEUE* pour les réseaux de Winnipeg et d'Ottawa. Enfin, l'implantation *FSPTHEAP* obtient des résultats satisfaisants mais l'implantation *FSPT2QUEUE* demeure supérieure dans tous les cas.

La dernière colonne du tableau 2.X nous montre les résultats obtenus par l'implantation *FSPTHEAPO*. Rappelons que cette implantation s'intéresse uniquement au calcul des plus courts chemins vers chaque destination (centroïde). On remarque que cette spécialisation de l'implantation *FSPHEAP* permet un calcul plus rapide que l'implantation *FSPT2QUEUE*. Donc, dans le cas où seuls les chemins vers chaque destination sont nécessaires, une implantation utilisant une technique d'*extensions sélectives* permet certaines optimisations rendant le calcul très efficace. Cependant, lorsque les plus courts chemins reliant l'origine à chaque lien du réseau sont désirés, une implantation utilisant une technique d'*ajustements progressifs* (*FSPT2QUEUE*) s'avère plus efficace.

Afin d'analyser plus en profondeur le comportement des différentes implantations, nous reportons dans les figures 2.9 et 2.10 des statistiques concernant les structures de données utilisées par chaque type d'implantations. Les graphiques de gauche et de droite, inclus dans ces deux figures, montrent respectivement le nombre maximum d'éléments que

contiennent les structures de données et le nombre total d'élements insérés dans ces mêmes structures pendant l'exécution de l'implantation.

La figure 2.9 s'attarde aux implantations ne considérant pas les mouvements aux intersections. On remarque dans le graphique de gauche que le nombre maximum d'éléments

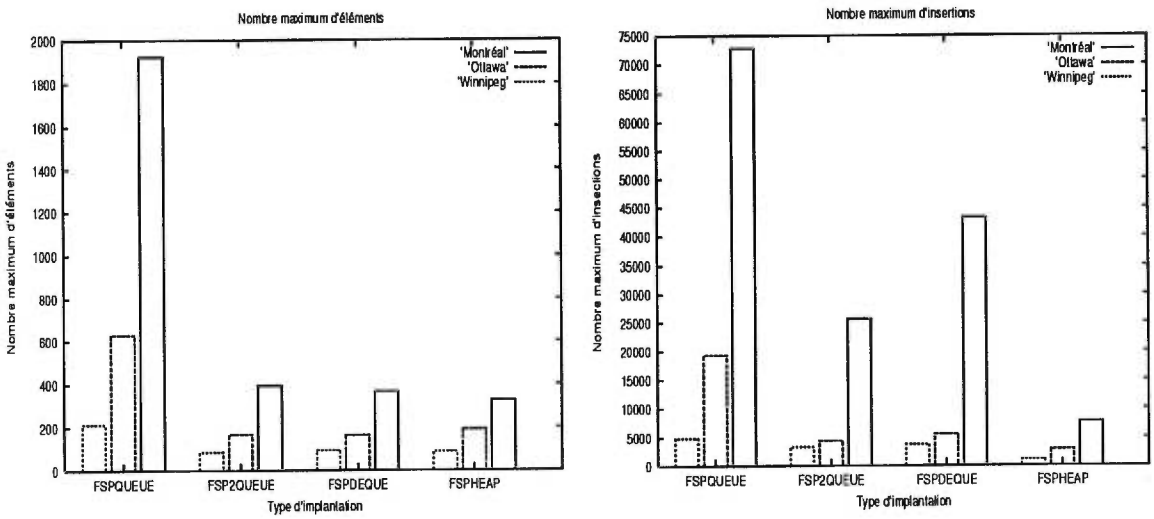

Figure 2.9: Implantations de type *FSP*: Statistiques des structures de données.

(noeuds) pouvant se trouver dans la structure de données utilisée par l'implantation *FSPQUEUE*, ne dépasse pas 30% du nombre total de noeuds pour chacun des trois réseaux. Les trois autres structures de données associées aux implantations *FSP2QUEUE*, *FSPDEQUE* et *FSPHEAP*, montrent des taux d'occupations similaires et moins élevés que l'implantation *FSPQUEUE*. Par conséquent, la taille des structures de données que nous avons fixée à $n/2$ est ainsi justifiée bien que, dans le cas des implantations *FSP2QUEUE*, *FSPDEQUE* et *FSPHEAP*, une taille inférieure à $n/2$ aurait pu être utilisée. Par ailleurs, on remarque dans le graphique de droite que le nombre peu élevé d'insertions pour l'implantation *FSPHEAP* n'est pas suffisant pour rendre celle-ci plus efficace que les implantations *FSP2QUEUE* et *FSPDEQUE*. Ceci s'explique par le coût élevé des opérations de la structure de données utilisée (*monceau*), comparativement aux structures de données *queue* et *deque*. Il est également intéressant de voir que le nombre plus élevé d'insertions obtenus pour l'implantation *FSPDEQUE*, comparativement à l'implantation *FSP2QUEUE*, n'est pas suffisant pour en affecter les résultats.

Les statistiques traitées précédemment, sont également comptabilisées pour les implantations considérant les mouvements aux intersections dans les graphiques de la figure 2.10. Dans le cas des implantations *FSPTQUEUE*, *FSPT2QUEUE*, *FSPTDEQUE* et *FSPTHEAP*, les nombres maximaux d'éléments contenus dans les structures de données sont proportionnellement équivalents à ceux obtenus pour les implantations ne considérant pas les mouvements aux intersections. Par conséquent, en fixant la taille des structures utilisées à $m/2$, nous nous sommes assurés de ne pas excéder l'espace disponible. Les résultats présentés dans le graphique de droite de la figure 2.10 sont particulièrement intéressants, puisqu'ils expliquent en partie la mauvaise performance de l'implantation *FSPTDEQUE*. On

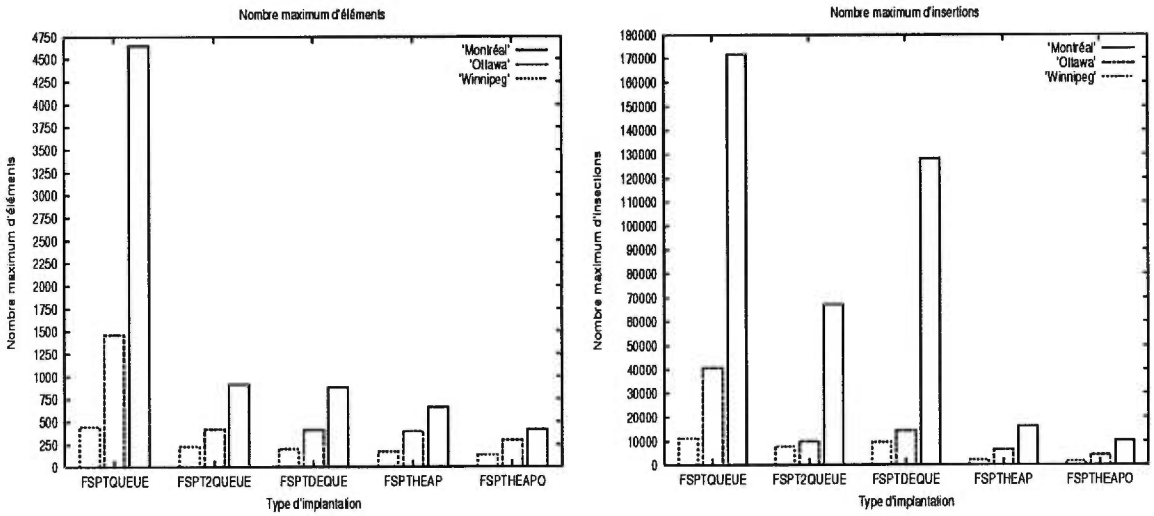

Figure 2.10: Implantations *FSPT*: Statistiques des structures de données.

constate que, pour cette implantation, les nombres d'insertions obtenus pour chaque réseau sont très élevés comparativement à ceux obtenus pour l'implantation *FSPT2QUEUE*. En fait, bien que les écarts remarqués sont proportionnellement équivalents à ceux obtenus pour les implantations *FSP2QUEUE* et *FSPDEQUE*, ils sont plus importants en absolu. Étant donné ces nombres élévés d'insertions, on peut conclure que l'implantation *FSPDEQUE* effectue un plus grand nombre de mises à jour d'étiquettes (une même étiquette est mise à jour plus d'une fois). Ce qui explique la performance observée.

En observant attentivement le graphique de droite, on remarque également que les implantations *FSPTQUEUE*, *FSPT2QUEUE* et *FSPTDEQUE* présentent, pour chaque réseau, des maxima d'insertions beaucoup plus élevés que le nombre de liens présents dans le

réseau. Ce comportement, qui caractérise les méthodes utilisant une technique d'*ajustements progressifs*, impliquent que plusieurs des étiquettes sont mises à jour plus d'une fois. D'autre part, les deux implantations utilisant une technique d'*extensions sélectives* montrent des maxima d'insertions peu élevés, puisqu'une étiquette qui devient permanente ne peut être modifiée par la suite. L'implantation *FSPTHEAPO* permet, comparativement à l'implantation *FSPTHEAP*, de réduire le nombre d'insertions de 20% pour le réseau de Winnipeg, de 31% pour le réseau d'Ottawa et de 38% pour le réseau de Montréal. Cette réduction du nombre des insertions justifie les performances qu'obtient cette implantation et confirme qu'il est avantageux d'utiliser ce type de stratégie dans le cas où seuls les chemins vers chaque destination sont nécessaires.

2.3 Implantations séquentielles d'algorithmes de calcul des plus courts chemins temporels

Comme nous l'avons déjà vu au chapitre 1, la recherche d'algorithmes de calcul des plus courts chemins temporels constitue un champ d'étude relativement nouveau. Notre revue de la littérature nous a permis de distinguer un certain nombre d'algorithmes développés pour résoudre différentes variantes du problème de base.

Dans cette section, nous nous limitons à l'implantation d'algorithmes de calcul des plus courts chemins temporels en temps discret et visant la recherche des chemins minimisant les temps de parcours (*time-dependent least-time path*)[11]. De plus, nous nous concentrons sur la variante du problème visant à déterminer les plus courts chemins joignant chaque noeud à un noeud destination donné et ce, pour toutes les périodes de temps. Actuellement, cette variante du problème attire de plus en plus l'attention des chercheurs puisque généralement, dans les nouveaux modèles de gestion des transports, elle permet d'obtenir directement l'information désirée. Les algorithmes implantés sont ceux proposés par Ziliaskopoulos et Mahmassani [57], Chabini [12] et Pallottino et Scutellà [46][12].

La présente section est organisée comme suit. Un bref retour sur les algorithmes implantés est d'abord effectué. Ensuite, nous traitons du développement des implantations et, en conclusion, nous présentons l'analyse des résultats obtenus suite aux expérimentations réalisées.

2.3.1 Algorithmes implantés

Rappelons d'abord la notation utilisée dans le chapitre 1. Soit $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ un réseau comportant, pour chaque lien $(i, j) \in \mathcal{A}$, un temps de parcours $d_{ij}(t)$ représentant le temps requis pour voyager au temps t du noeud i au noeud j. Généralement, $d_{ij}(t)$ est non négatif et défini pour t appartenant à un ensemble discret d'intervalles de temps $\mathcal{S} = \{t_0, t_1, t_2, ..., t_{\mathcal{M}-1}\}$, où t_0 est le plus petit temps de départ pour chaque noeud du réseau, et $\mathcal{M}$ est un entier assez grand pour que $[t_0, t_{\mathcal{M}-1}]$ soit la période d'intérêt. On définit également $\pi_i(t)$ comme le temps de parcours minimal du chemin reliant le noeud i à la destination q, où t est le temps de départ du noeud i et $\Pi_i = [\pi_i(t_0), \pi_i(t_1), \pi_i(t_2)..., \pi_i(t_{\mathcal{M}-1})]$ comme étant un vecteur associé au noeud i contenant chacune des étiquettes $\pi_i(t)$ pour chaque période de

[11] Dans la suite de cette section, nous utilisons l'expression plus courts chemins pour référer aux chemins minimisant les temps de parcours (*time-dependent least-time path*).

[12] Voir chapitre 1, section 1.2.3.

temps $t \in \mathcal{S}$. Enfin, on note $b_i(t)$, le noeud suivant le noeud i au temps t sur le plus court chemin vers la destination.

Le problème consiste à déterminer, pour chaque période de temps t, les plus courts chemins reliant chaque noeud i à la destination q.

Ziliaskopoulos et Mahmassani 1993

L'algorithme proposé par Ziliaskopoulos et Mahmassani consiste en une extension d'une approche classique de calcul des plus courts chemins statiques. Il s'agit en fait d'un algorithme utilisant une technique d'*ajustements progressifs*. Il est donc possible de l'implanter en utilisant l'une des structures de données vues dans la section 1.1.3 du chapitre 1 (*queue*, *2queue* ou *deque*). Pour le moment, supposons que $\mathcal{Q}$ soit simplement la liste des noeuds à visiter. Alors l'algorithme, nommé **TDLTP**, est le suivant:

1. Initialisation

$\Pi_q = (0, 0, 0, ..., 0), \forall t \in \mathcal{S}$;

$\Pi_i = (+\infty, +\infty, +\infty, ..., +\infty), \forall (t \in \mathcal{S}, i \neq q)$;

2. Étape principale

si $\mathcal{Q} = \emptyset$, alors aller à 4

sélectionner le premier noeud j de la queue $\mathcal{Q}$, $\mathcal{Q} = \mathcal{Q} - \{j\}$

pour chaque noeud $i | (i, j) \in \mathcal{A}_j^-$ faire

pour chaque période de temps $t \in \mathcal{S}$ faire

si $\pi_i(t) > d_{ij}(t) + \pi_j(t + d_{ij}(t))$, alors $\pi_i(t) = d_{ij}(t) + \pi_j(t + d_{ij}(t))$ et $b_i(t) = j$

si au moins une des étiquettes $\pi_i(t)$ a été modifiée, alors $\mathcal{Q} = \mathcal{Q} + \{i\}$

3. Répéter l'étape 2.

4. L'algorithme est terminé.

À la fin de l'algorithme, le vecteur Π_i, associé à chaque noeud $i \in \mathcal{N}$, contient les étiquettes $\pi_i(t)$ pour chaque période de temps $t \in \mathcal{S}$. Dans cet algorithme, on considère $\mathcal{S} = \{t_0, t_0 + \delta, t_0 + 2\delta, ..., t_0 + (\mathcal{M} - 1)\delta\}$ comme un ensemble discret d'intervalles de temps, où δ est un petit intervalle de temps, et $\mathcal{M}$ est un entier assez grand pour que $[t_0, t_0 + (\mathcal{M} - 1)\delta]$ soit la période d'intérêt (heure de pointe, par exemple). Les temps de parcours $d_{ij}(t)$ prennent des valeurs réelles non-négatives et ils sont définis sur $\mathcal{S} = \{t_0, t_0 + \delta, t_0 + 2\delta, ..., t_0 + (\mathcal{M} - 1)\delta\}$. Ils supposent que $d_{ij}(\tilde{t}) = d_{ij}(t_0 + k\delta)$ pour $\tilde{t}$ dans l'intervalle $t_0 + k\delta < \tilde{t} < t_0 + (k + 1)\delta$. De plus, on suppose également que $d_{ij}(t)$, pour $t > t_0 + (\mathcal{M} - 1)\delta$, est constant et égal à $d_{ij}(t_0 + (\mathcal{M} - 1)\delta)$.

Chabini 1997

L'approche suggérée par Chabini ne requiert l'utilisation d'aucune structure de données. Cependant, cet algorithme suppose des temps de parcours entiers et strictement positifs définis dans $\mathcal{S} = \{0, 1, 2, 3, ..., \mathcal{M}-1\}$. En fait, cette hypothèse est à la base du raisonnement menant à la conception de cet algorithme. Son algorithme, qu'il nomme **DOT** (*Decreasing Order of Time*), utilise implicitement la propriété acyclique du réseau espace-temps associé à l'expansion dans le temps du réseau temporel original. Supposons qu'un algorithme de plus courts chemins statiques (SSP) est disponible. Alors, l'algorithme proposé par Chabini est le suivant:

1. Initialisation

$\pi_i(t) = +\infty$, $\forall (t < \mathcal{M}-1, i \neq q)$;
$\pi_q(t) = 0$, $\forall t < \mathcal{M}-1$;
$\pi_i(\mathcal{M}-1) = SSP(d_{ij}(\mathcal{M}-1), q)$;
Note: $\pi_i(t) = \pi_i(\mathcal{M}-1)$, $\forall (t \geq \mathcal{M}-1, i \neq q)$;

2. Itération principale

pour $t = \mathcal{M}-2$ jusqu'à 0 faire
  pour tous les arcs $(i,j) \in \mathcal{A}$ faire
    si $\pi_i(t) > d_{ij}(t) + \pi_j(t + d_{ij}(t))$, alors $\pi_i(t) = d_{ij}(t) + \pi_j(t + d_{ij}(t))$ et $b_i(t) = j$

Comme précédemment, le vecteur Π_i, associé à chaque noeud $i \in \mathcal{N}$, contient les étiquettes $\pi_i(t)$ pour chaque période de temps $t \in \mathcal{S}$ lorsque l'algorithme se termine. Notons que dans cet algorithme, le calcul des plus courts chemins pour des temps de départ plus grands ou égals à $\mathcal{M}-1$ est équivalent au calcul des plus courts chemins statiques.

Pallottino et Scutellà 1997

Pallottino et Scutellà proposent, indépendamment de Chabini, un algorithme exploitant également la propriété acyclique du réseau espace-temps. L'algorithme, nommé **CHRONOSPT**, utilise un parcours inversé d'une *liste de compartiments* L, dans le but d'exploiter cette propriété. Chaque compartiment corrrespond à une période de temps t et contient la liste des noeuds à visiter pour cette période de temps. L'algorithme suggéré est le suivant:

1. Initialisation

$\pi_i(t) = +\infty$, $\forall (t \in \mathcal{S}, i \neq q)$;
$\pi_q(t) = 0$, $\forall t \in \mathcal{S}$;
$L_t = \{q\}$, $\forall t \in \mathcal{S}$;
$t_1 = \mathcal{M}-1$;

2. Itération principale

tant que $L_{t_1} \neq \emptyset$ faire
sélectionner j de L_{t_1}
$L_{t_1} = L_{t_1} - \{j\}$;
pour chaque $i : (i,j) \in \mathcal{A}_j^-$ faire
$t_2 = t_1 + d_{ij}(t_1)$;
si $t_2 > \mathcal{M} - 1$ alors aller à 2
si $\pi_i(t_1) > d_{ij}(t_1) + \pi_j(t_2)$ alors
$\pi_i(t_1) = d_{ij}(t_1) + \pi_j(t_2)$;
$b_i(t) = j$;
si $i \notin L_{t_1}$ alors $L_{t_1} = L_{t_1} + \{i\}$

3. Mise à jour.

si $t_1 - 1 < 0$ aller à 4
$t_1 = t_1 - 1$ et aller à 2

4. L'algorithme est terminé.

À la fin de l'algorithme, le vecteur Π_i, associé à chaque noeud $i \in \mathcal{N}$, contient également les étiquettes $\pi_i(t)$ pour chaque période de temps $t \in \mathcal{S}$. Cependant, contrairement à l'algorithme de Chabini ainsi qu'à celui de Ziliaskopoulos et Mahmassani, l'algorithme de Pallottino et Scutellà ne considère que les temps d'arrivée n'excédant pas la dernière période de temps. Par conséquent, certaines étiquettes $\pi_i(t)$ demeurent avec une valeur infinie s'il est impossible d'atteindre la destination avant la dernière période de temps.

2.3.2 Développement des implantations

Afin d'implanter les trois algorithmes discutés précédemment, nous avons développé les classes **TDLTP**, **DOT** et **CHRONOSPT**. Dans le but d'éviter la duplication de l'information et de «simplifier» le développement, ces trois classes dérivent de la classe de base **TDSP** (*Time-Dependent Shortest-Path*). La figure 2.11 permet de visualiser la hiérarchie reliant les classes développées. De façon à faciliter d'une part, l'utilisation des outils et, d'autre part, le travail de mise à jour de la hiérarchie, une interface unique est partagée par les classes *TDLTP*, *DOT* et *CHRONOSPT*. Une description de la fonctionnalité de chaque méthode est donnée dans la tableau 2.XI. La méthode $sp(\cdot)$, qui permet de calculer les plus courts chemins temporels, accepte quatre paramètres. Le premier paramètre correspond à l'identificateur du noeud destination à considérer. Ensuite, afin de pouvoir conserver l'information pertinente, la méthode requiert trois matrices entières de dimensions $|\mathcal{M}| \times n$. Ces matrices sont définies de la façon suivante:

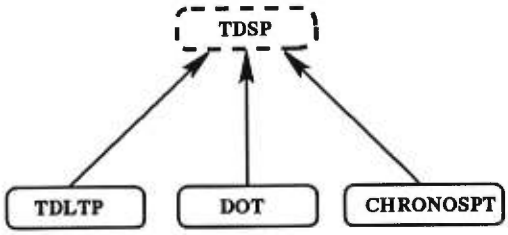

Figure 2.11: Hiérarchie des classes.

Méthodes[a]	Fonctionnalités
?? (NFF& network); ?? ();	Constructeurs.
void setNetwork (NFF& network);	Permet de sélectionner un réseau.
int sp (const long id, int** a, int** b, int** c);	Calcule les plus courts chemins temporels pour un noeud destination. Retourne 1 si le calcul est effectué et 0 sinon.
NFF* network () const;	Donne accès à l'objet NFF.
int numberOfLinks () const; int numberOfNodes () const; int numberOfPeriods () const; int numberOfCentroids () const;	Retournent respectivement le nombre de liens, de noeuds, de périodes de temps et de centroïdes.

[a]Des caractères ?? sont placés pour éviter de nommer chacune des classes concernées.

Tableau 2.XI: Interface partagée par les outils.

- *a[t][i]:* Temps d'arrivée à la destination à partir du noeud i et pour un temps de départ t;
- *b[t][i]:* Noeud suivant le noeud i au temps t;
- *c[t][i]:* Temps d'arrivée au noeud suivant le noeud i au temps t.

Compte tenu que, sur le type de machines utilisées, un entier requiert un espace mémoire de 4 *bytes*, l'espace mémoire total nécessaire pour conserver l'information résultant du calcul des plus courts chemins pour une destination est $3 \times 4 \times n \times |\mathcal{M}| = 12n|\mathcal{M}|$ *bytes*.

L'utilisation d'un outil se fait relativement facilement. Supposons que l'on veuille calculer les plus courts chemins temporels en utilisant l'outil *DOT*, alors le programme *C++* correspondant prend la forme suivante:

```
#include <DOT.h>
...
void main ()
{
  ...
  NFF network;
```

```
  ifstream inputFile (''ottawa.nff'');
  inputFile >> network;
  ...
  NFFSection& centroids = network.sectionRef ("Centroids");
  NFFAttributeRef& nodeCentroid = centroids.refAttribute ("Node");
  ...
  DOT tool (network);

  int nbPeriods = tool.numberOfPeriods();
  int nbNodes = tool.numberOfNodes();
  int nbCentroids = tool.numberOfCentroids();

  int** arrivalTimes = new int[nbPeriods];
  for (int i = 0; i < nbPeriods; i++)
    arrivalTimes[i] = new int[nbNodes];

  int** nextNodes = new int[nbPeriods];
  for (i = 0; i < nbPeriods; i++)
    nextNodes[i] = new int[nbNodes];

  int** nextTimes = new int[nbPeriods];
  for (i = 0; i < nbPeriods; i++)
    nextTimes[i] = new int[nbNodes];

  for (i = 0; i < nbCentroids; i++) {
    tool.sp (nodeCentroid[i], arrivalTimes, nextNodes, nextTimes);
    ...
  }
  ...
}
```

Dans cet exemple, les méthodes *numberOfPeriods()*, *numberOfNodes()* et *numberOfCentroids()* sont utilisées pour connaître respectivement le nombre de périodes de temps, le nombre de noeuds et le nombre de centroïdes dans le réseau. À l'aide de ces informations, nous sommes en mesure de construire les structures (matrices) permettant de contenir les résultats du calcul des plus courts chemins. Ensuite, pour chacun des centroïdes i, nous utilisons la méthode $sp(\cdot)$ pour déterminer ces plus courts chemins reliant chaque noeud au centroïde traité. Il est important de noter que le premier paramètre transmis à la méthode $sp(\cdot)$ doit correspondre à l'identificateur d'un noeud. Pour cette raison, nous utilisons l'attribut *Node* (de la section *Centroids*) qui réfère à l'identificateur du noeud associé au centroïde traité.

Comme on peut le remarquer, ce programme ne conserve pas les résultats du calcul des plus courts chemins. On suppose en fait qu'une opération quelconque est effectuée avec les résultats obtenus pour un centroïde donné avant d'en traiter un nouveau. Certains

modèles de planification en transport peuvent toutefois exiger de conserver les résultats des calculs pour chacun des centroïdes considérés. Dans ce cas, une structure plus coûteuse en espace mémoire devient nécessaire. En effet, pour conserver les résultats des calculs des plus courts chemins pour chaque centroïde, un espace mémoire de $|\mathcal{C}| \times 12n|\mathcal{M}|$ *bytes* (où $|\mathcal{C}|$ est le nombre de centroïdes dans le réseau) est requis. Cet espace mémoire est considérable et, pour la plupart des exemples traités dans ce travail, nous ne sommes pas en mesure de satisfaire à cette demande. À titre d'exemple, le réseau d'Ottawa exige pour un intervalle contenant 120 périodes de temps, un espace mémoire d'approximativement 910 *Mb*.

Dans ce qui suit, nous discutons des implantations des algorithmes de calcul des plus courts chemins temporels associées à chacune des trois classes. Dans chaque cas, nous présentons les détails relatifs à l'implantation traitée ainsi que le code *C++* correspondant. Afin de rendre la lecture des codes *C++* associés aux implantations plus facile, un certain nombre de variables sont définies préalablement:

- *travelTimes[t][i]:* Temps de parcours au temps t associé au lien i;
- *fromNode[i]:* Noeud origine du lien i;
- *toNode[i]:* Noeud terminal du lien i;
- *incomingLinks[i]:* Liste des liens entrant au noeud i.

Implantation de l'algorithme TDLTP

La structure de données utilisée dans notre implantation de l'algorithme proposé par Ziliaskopoulos et Mahmassani est une *deque*[13]. Cette structure de données a également été utilisée par Ziliaskopoulos et Mahmassani dans leur implantation de l'algorithme *TDLTP* [57]. Bien que les résultats de la section 2.2 nous montrent que la structure de données *2queue* semble plus efficace, les résultats obtenus par Chabini [12] prouvent que l'utilisation d'une *2queue* n'améliore pas la performance de cet algorithme.

Afin d'éviter d'insérer un même élement plusieurs fois dans la structure, nous associons à chaque noeud i, une variable permettant de savoir si le noeud considéré est contenu ou a déjà été contenu dans la structure. Le tableau $elementOfQ[\cdot]$ permet de conserver l'information en question. Pour un noeud i quelconque, nous avons que

$$elementOfQ[i] = \begin{cases} 1, si\ i \in \mathcal{Q}; \\ -1, si\ i\ a\ déjà\ été\ dans\ \mathcal{Q}; \\ 0, sinon. \end{cases}$$

[13]Les détails concernant l'implantation de cette structure de données sont présentés dans la section 2.2.

Le code *C++* correspondant à la méthode *sp (const long id, int** a, int** b, int** c)* est le suivant:

```
int TDLTP::sp (const long id, int** arrivalTimes,
               int** nextNodes, int** nextTimes)
{
  // On verifie s'il y a des liens entrant a
  // la destination
  int size = incomingLinks[id].size();
  if (!size)
    return 0;

  // Initialisation
  for (int t = 0; t < nbPeriods; t++)
    for (int i = 0; i < nbNodes; i++) {
      nextNodes[t][i] = -1;
      nextTimes[t][i] = -1;
      if (i == destination)
        arrivalTimes[t][i] = 0;
      else
        arrivalTimes[t][i] = INFINITY;
    }

  for (int i = 0; i < nbNodes; i++)
    elementOfQ[i] = 0;

  // Les noeuds adjacents a la destination sont
  // introduits dans la deque
  long link;
  long candidat;
  for (i = 0; i < size; i++) {
    link = incomingLinks[id][i];
    candadat = _fromNode[link];
    for (int t = 0; t < nbPeriods; t++) {
      arrivalTimes[t][candidat] = t+
      travelTimes[link][t];
      nextNodes[t][candidat] = id;
      if (travelTimes[link][t]+t >= nbPeriods)
        nextTimes[t][candidat] = nbPeriods-1;
      else
        nextTimes[t][candidat] = t+
        travelTimes[link][t];
    }

    deque->insertAtEnd (candidat);
    elementOfQ[candidat] = 1;
  }

  // Iteration principale
  while (!deque->empty()) {

    long current = _deque->remove();
    elementOfQ[current] = -1;

    // On traite les noeuds adjacents au noeud
    // courant
    size = incomingLinks[current].size();
    for (i = 0; i < size; i++) {

      link = incomingLinks[current][i];
      node = fromNode[link];
      int modified = 0;
      int tt;

      for (int t = 0; t < nbPeriods; t++) {

        tt = t+_travelTimes[link][t];
        if (tt >= nbPeriods-1)
          tt = nbPeriods-1;

        if (arrivalTimes[t][candidat] >
            travelTimes[link][t]+
            arrivalTimes[tt][current]) {

          arrivalTimes[t][candidat] =
          travelTimes[link][t]+
          arrivalTimes[tt][current];
          modified = 1;
          nextNodes[t][candidat] = current;
          nextTimes[t][candidat] = tt;
        }
      }

      // Le noeud candidat est insere dans
      // la deque si au moins une de ses
      // etiquettes est modifiees
      if (modified) {

        if (elementOfQ[candidat] == 1) continue;

        if (elementOfQ[candidat] == -1) {
          deque->insertAtFront (candidat);
          elementOfQ[candidat] = 1;
        }
        else {
```

```
        deque->insertAtEnd (candidat);
        elementOfQ[candidat] = 1;
      }
    }
  }
```

```
  }

  return 1;
}
```

Implantation de l'algorithme DOT

L'implantation de l'algorithme proposé par Chabini requiert l'utilisation d'un outil de calcul des plus courts chemins statiques effectuant une recherche par parcours arrière. Les résultats obtenus dans la section 2.2 nous suggèrent d'utiliser l'outil *BSP2QUEUE*. Ce dernier semble en effet le plus efficace parmi les outils diponibles présentement. Le code *C++* correspondant à la méthode *sp (const long id, int** a, int** b, int** c)* est le suivant:

```
int DOT::sp (const long id, int** arrivalTimes,
             int** nextNodes, int** nextTimes)
{
  // On verifie s'il y a des liens entrant a
  // la destination
  int size = incomingLinks[destination].size();
  if (!size)
    return 0;

  // Initialisation
  for (int t = 0; t < nbPeriods; t++)
    for (int i = 0; i < nbNodes; i++) {
      nextNodes[t][i] = -1;
      nextTimes[t][i] = -1;
      if (i == destination)
        arrivalTimes[t][i] = 0;
      else
        arrivalTimes[t][i] = INFINITY;
    }

  // Un outil de pcc statique est utilise
  // pour la derniere periode de temps
  ssp->setCostLink (travelTimes[t]);
  ssp->sp (destination);
  long* arrTimes = ssp->bestTrip();
  long* next = ssp->nextNode();

  for (int i = 0; i < nbNodes; i++) {
    arrivalTimes[t][i] = arrTimes[i];
    nextNodes[t][i] = next[i];
    nextTimes[t][i] = t;
  }

  // Iteration principale
  int from, to;
  int tt;
  for (t = nbPeriods-2; t >= 0; t--)
    for (int i = 0; i < nbLinks; i++) {
      from = fromNode[i];
      to = toNode[i];

      tt = t+travelTimes[t][i];
      if (tt >= nbPeriods-1)
        tt = nbPeriods-1;

      if (arrivalTimes[t][from] >
          travelTimes[t][i]+
          arrivalTimes[tt][to]) {

        arrivalTimes[t][from] =
        travelTimes[t][i]+
        arrivalTimes[tt][to];
        nextNodes[t][from] = to;
        nextTimes[t][from] = tt;
      }
    }

  return 1;
}
```

Implantation de l'algorithme CHRONOSPT

Dans un premier temps, la *liste de compartiments* suggérée par Pallottino et Scutellà doit être implantée. La figure 2.12 permet de voir une représentation de cette structure. Chaque

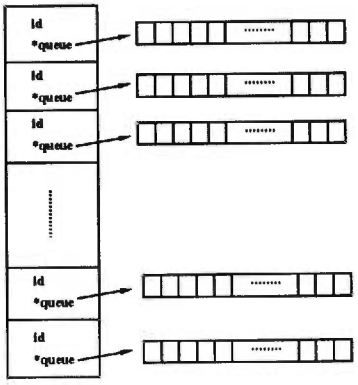

Figure 2.12: Liste de compartiments.

compartiment contient un identificateur (**id**) et un pointeur (**queue**) vers une *queue*[14]. Pour ajouter un élément à la structure, l'usager doit fournir l'identificateur du compartiment (*id* $\in [0, C-1]$ où C est le nombre de compartiments) et la valeur qui sera contenue dans la *queue* associée à ce compartiment. Afin d'implanter cette structure de données, la classe **BUCKET** a été développée.

```
struct BUCKETCEL {

   int id;
   CRTQueue<int>* queue;
};

class BUCKET {

 public:
   ...

   void setSize (long size1, long size2)
   {
     buckets = new BucketCel[size1];
     top = buckets;
     end = buckets+size1-1;
     for (int i = 0; i < size1; i++) {
       buckets[i].queue = new CRTQueue<int> (size2);
       buckets[i].id = i;
     }

     current = end;
   }

   void add (int value, int bck)
   {
     buckets[bck].queue->insert (value);
   }

   int remove ()
   {
      return *current.queue->remove();
   }

   int bucketNumber ()
   {
```

[14]Les détails concernant l'implantation de cette structure de données sont présentés dans la section 2.2.

```
    return *current.id;
  }

  int empty ()
  {
    while (*current.queue->empty())
      if (current == top)
        return 1;
      current--;
    return 0;
  }

private:

  BucketCel* buckets;
  BucketCel* current;
  BucketCel* top;
  BucketCel* end;
  ...
};
```

Étant donné s, la taille de chaque *queue* associée à chaque compartiment, un objet de la classe $BUCKET$ nécessite un espace mémoire de $16+C(8+28+4s) = 16+4C(9+s)$ *bytes.*

Afin d'éviter d'insérer un même élement plusieurs fois dans la liste de compartiments, notre implantation associe pour chaque noeud i et pour chaque période de temps t une variable permettant de savoir si un noeud i est contenu ou a déjà été contenu dans la structure au temps t. Le tableau $elementOfQ[\cdot][\cdot]$ permet de conserver l'information en question. Pour un noeud i au temps t, nous avons que

$$elementOfQ[t][i] = \begin{cases} 1, si\ i \in \mathcal{Q}\ au\ temps\ \mathrm{t}; \\ -1, si\ i\ a\ déjà\ été\ dans\ \mathcal{Q}\ au\ temps\ \mathrm{t}; \\ 0, sinon. \end{cases}$$

Le code *C++* correspondant à la méthode *sp (const long id, int** a, int** b, int** c)* est le suivant:

```
int CHRONOSPT::sp (const long id, int** arrivalTimes,
                   int** nextNodes, int** nextTimes)
{
  // On verifie s'il y a des liens entrant a
  // la destination
  int size = incomingLinks[destination].size();
  if (!size)
    return 0;

  // Initialisation
  for (int t = 0; t < nbPeriods; t++)
    for (int i = 0; i < nbNodes; i++) {
      nextNode[t][i] = -1;
      nextTime[t][i] = -1;
      if (i == destination)
        arrivalTimes[t][i] = 0;
      else
        arrivalTimes[t][i] = INFINITY;
    }

  for (int t = 0; t < nbPeriods; t++)
    buckets->add (destination, t);

  // Iteration principale
  while (!buckets->empty()) {

    int curNode = buckets->remove();
    int curTime = buckets->bucketNumber();
    int link, candidat;
    int tt;

    // On traite les noeuds adjacents au noeud
    // courant
    size = incomingLinks[curNode].size();
    for (int i = 0; i < size; i++) {
```

```
link = incomingLinks[curNode][i];
candidat = fromNode[link];

tt = curTime+travelTimes[curTime][link];
if (tt > nbPeriods-1) continue;

if (arrivalTimes[curTime][candidat] >
    travelTimes[curTime][link]+
    arrivalTimes[tt][curNode]) {

  arrivalTimes[curTime][candidat] =
  travelTimes[curTime][link]+
  arrivalTimes[tt][curNode];
  nextNodes[curTime][candidat] = curNode;
  nextTimes[curTime][candidat] = tt;

  if (!elementOfQ[curTime][candidat]) {
    buckets->add (candidat, curTime);
    elementOfQ[curTime][candidat] = 1;
  }
 }
}
}

return 1;
}
```

2.3.3 Expérimentations et analyse des résultats

Cette partie s'intéresse à l'analyse des performances des implantations séquentielles développées dans la présente section. Afin de tester ces implantations, nous avons généré, pour les réseaux urbains des villes de Winnipeg, Ottawa et Montréal[15], des temps de parcours $d_{ij}(t) \in [1,5]$ entiers pour des intervalles variant de 30, 60, 90 à 120 périodes de temps.

Le tableau 2.XII contient les temps d'exécution obtenus sur une station de travail *SUN SPARC Ultra 1/140*, pour chacun des trois algorithmes appliqués sur les différents exemplaires disponibles[16]. Dans chaque cas, on retrouve le temps de calcul global correspondant au traitement de toutes les destinations et, entre parenthèses, le temps de calcul moyen par destination. Le temps mesuré est le temps réel requis par l'application et il est exprimé en secondes. Dans le cas du réseau de Montréal, seuls les résultats pour des intervalles de 30 et 60 périodes de temps sont disponibles en raison de l'espace mémoire limité sur ce type de machine. Étant donné que les résultats ne sont pas complets, nous n'utiliserons pas ces derniers pour analyser en profondeur le comportement des trois algorithmes. On peut toutefois remarquer que l'algorithme *TDLTP* est le moins efficace des trois et que les algorithmes *DOT* et *CHRONOSPT* ont des comportements très semblables pour le réseau de Winnipeg. Dans le cas des réseaux d'Ottawa et de Montréal, l'algorithme *DOT* est le plus performant.

Le tableau 2.XIII contient les temps d'exécution (temps réel en secondes) obtenus sur un *SUN SPARC Server 1000* pour chacun des trois algorithmes appliqués sur les différents exemplaires disponibles. Les 256 *Mb* de mémoire vive de cette machine, nous permettent cette fois d'obtenir des résultats pour le réseau pour des intervalles de 90 et 120 périodes de

[15] Les caractéristiques de ces réseaux sont fournies à la section 2.1.

[16] Les temps d'exécution obtenus sur cette station de travail seront particulièrement utiles lors de l'évaluation des performances d'une implantation parallèle développée au chapitre 3.

Réseau/$\|\mathcal{M}\|$	TDLTP	DOT	CHRONOSPT
Winnipeg/30	14.01 (0.091)	3.95 (0.026)	2.93 (0.019)
Winnipeg/60	63.28 (0.411)	8.13 (0.053)	8.33 (0.054)
Winnipeg/90	109.71 (0.712)	11.99 (0.078)	12.99 (0.084)
Winnipeg/120	162.67 (1.056)	15.97 (0.104)	16.72 (0.107)
Ottawa/30	143.64 (0.557)	19.41 (0.075)	30.30 (0.117)
Ottawa/60	371.64 (1.440)	37.87 (0.147)	59.44 (0.230)
Ottawa/90	615.12 (2.384)	55.62 (0.216)	90.23 (0.350)
Ottawa/120	884.46 (3.427)	72.588 (0.281)	119.76 (0.464)
Montréal/30	1032.66 (1.477)	133.92 (0.192)	214.49 (0.307)
Montréal/60	2670.33 (3.820)	265.18 (0.379)	451.43 (0.646)

Tableau 2.XII: Temps d'exécution sur un *SUN SPARC Ultra 1/140*.

temps. Comme nous l'avions déjà remarqué dans les résultats du tableau 2.XII, l'algorithme *TDLTP* est le moins efficace des trois algorithmes. Cependant, contrairement aux résultats obtenus sur la station de travail *SUN SPARC Ultra 1/140*, nous remarquons que l'algorithme *CHRONOSPT* devient parfois plus efficace que l'algorithme *DOT*, contrairement aux résultats précédents où il performe moins bien dans tous les cas. Ce comportement s'explique par les différences entre les mémoires *caches* des deux architectures. La *cache* [48] est un espace mémoire où se retrouvent généralement certaines intructions qui sont exécutées fréquemment, ou certaines données qui sont accédées lors de l'exécution. Toutefois, cet espace mémoire est limité et ne peut contenir tout le code exécutable ou toutes les données à traiter. Lorsque deux architectures possèdent une *cache* très différente, l'exécution d'un même programme sur chacune d'elles peut mener à des comportements très différents. Dans le cas présent, une étude des caractéristiques des deux architectures utilisées, nous montre que le *SUN SPARC Ultra 1/140* possède une *cache* de 32 *Kb*[17] tandis que le *SUN SPARC Server 1000* peut compter sur une *cache* de 1/2 *Mb* (500 *Kb*) pour chacun des 8 processeurs. On peut alors conclure que l'algorithme *CHRONOSPT*, étant donné le type d'opérations qu'il effectue, semble prendre avantage de la dimension de la *cache* sur le *SUN SPARC Server 1000*[18].

Afin d'analyser plus en profondeur les résultats du tableau 2.XIII, nous présentons dans les figures 2.13 et 2.14 l'influence de la modification de certaines caractéristiques des réseaux. Tout d'abord, les trois graphiques de la figure 2.13 nous montrent l'influence de la largeur de l'intervalle sur le temps d'exécution des trois algorithmes. Ces graphiques nous permettent de voir que l'algorithme *TDLTP* est très sensible à la largeur de l'intervalle tandis que ce facteur n'a que peu d'influence sur les algorithmes *DOT* et *CHRONOSPT*.

[17] Il s'agit en fait d'une *cache* de 16*Kb* pour les données et de 16 *Kb* pour les instructions.

[18] Pour plus de détails sur le concept de *cache*, consulter Patterson et Hennessy [48].

Réseau/$\|\mathcal{M}\|$	TDLTP	DOT	CHRONOSPT
Winnipeg/30	41.62 (0.270)	8.94 (0.058)	8.96 (0.058)
Winnipeg/60	148.09 (0.962)	21.44 (0.139)	23.82 (0.155)
Winnipeg/90	277.79 (1.804)	35.71 (0.232)	38.28 (0.249)
Winnipeg/120	452.86 (2.941)	48.96 (0.318)	52.53 (0.341)
Ottawa/30	272.88 (1.058)	53.88 (0.209)	47.14 (0.183)
Ottawa/60	755.02 (2.926)	117.76 (0.456)	105.60 (0.409)
Ottawa/90	1488.04 (5.767)	182.91 (0.709)	176.40 (0.684)
Ottawa/120	2116.00 (8.202)	232.55 (0.901)	236.77 (0.900)
Montréal/30	2468.31 (3.531)	462.37 (0.661)	382.75 (0.548)
Montréal/60	6898.26 (9.869)	888.11 (1.271)	798.23 (1.142)
Montréal/90	11670.90 (16.697)	1408.82 (2.015)	1392.63 (1.992)
Montréal/120	19257.30 (27.550)	1833.18 (2.623)	2041.77 (2.921)

Tableau 2.XIII: Temps d'exécution sur un *SUN SPARC Server 1000.*

En fait, ces résultats expérimentaux réflètent les complexités de calcul respectives des trois algorithmes.

En effet, nous avons déjà vu au chapitre 2, que la complexité de calcul des algorithmes *DOT* et *CHRONOSPT* s'exprime linéairement en fonction de $\mathcal{M}$ contrairement à celle de l'algorithme *TDLTP* ($\mathcal{O}(n^3\mathcal{M}^2)$) qui varie quadratiquement en fonction de $\mathcal{M}$.

L'influence de la taille du réseau sur le temps d'exécution des trois algorithmes est également évaluée par l'entremise du graphique de la figure 2.14. Les courbes sont obtenues en considérant les réseaux de Winnipeg, Ottawa et Montréal pour $|\mathcal{M}| = 60$. On remarque que, pour l'algorithme *TDLTP*, les caractéristiques physiques (nombre de liens, nombre de noeuds) ont également un impact significatif sur le temps de calcul requis. L'expression de la complexité de calcul de l'algorithme *TDLTP* justifie encore une fois l'allure de la courbe obtenue.

L'analyse des temps d'exécution obtenus nous montre donc clairement l'inefficacité de l'algorithme *TDLTP* proposé par Ziliaskopoulos et Mahmassani [57] comparativement aux deux autres approches proposées par Chabini [12] (*DOT*) et Pallottino et Scutellà [46] (*CHRONOSPT*). Les graphiques des figures 2.13 et 2.14 permettent de constater la forte influence qu'ont les caractéristiques (dimension et largeur de l'intervalle considéré) des réseaux utilisées sur la performance de l'algorithme *TDLTP* contrairement aux algorithmes *DOT* et *CHRONOSPT*.

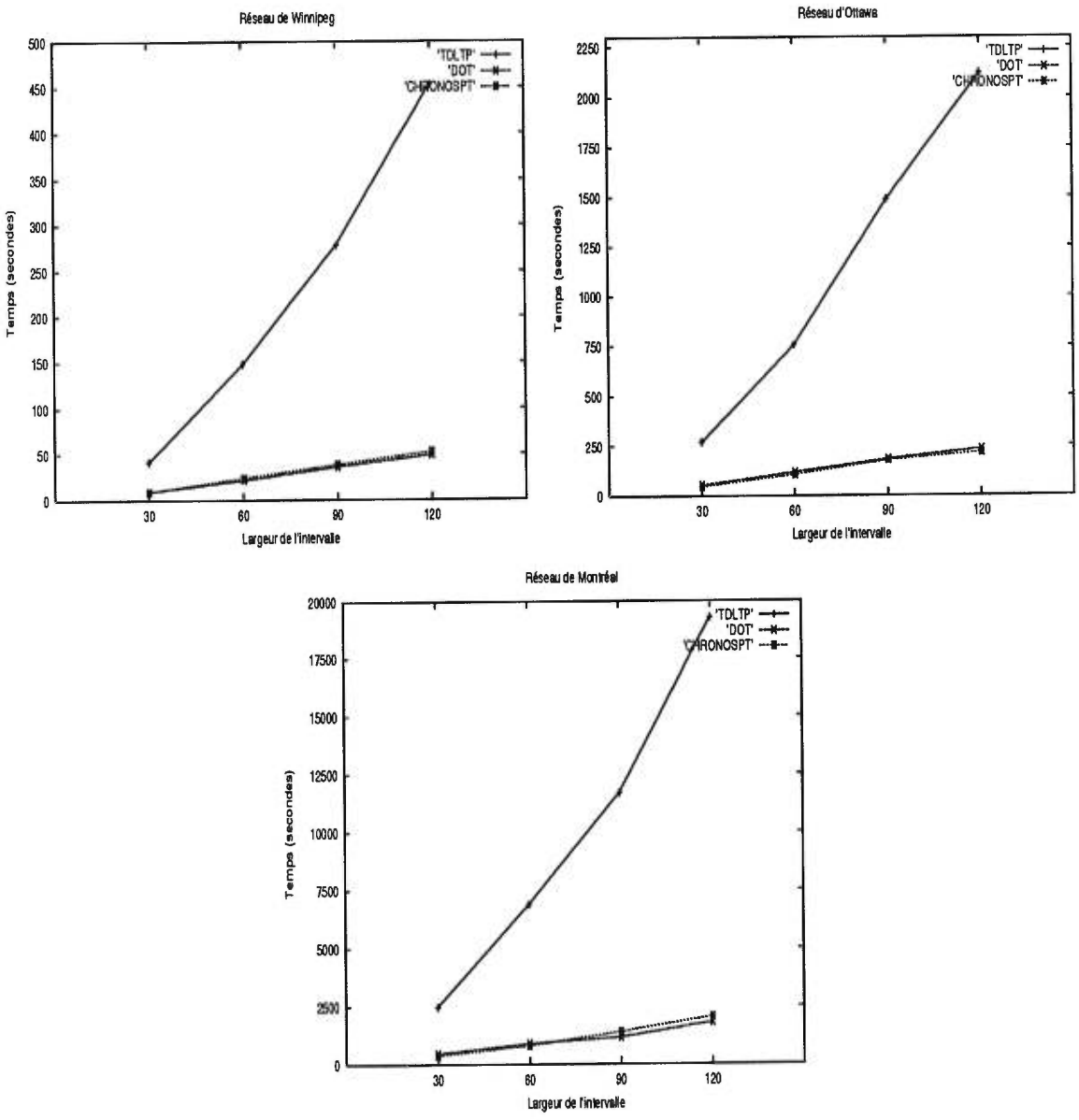

Figure 2.13: Influence de $|\mathcal{M}|$ sur le temps d'exécution.

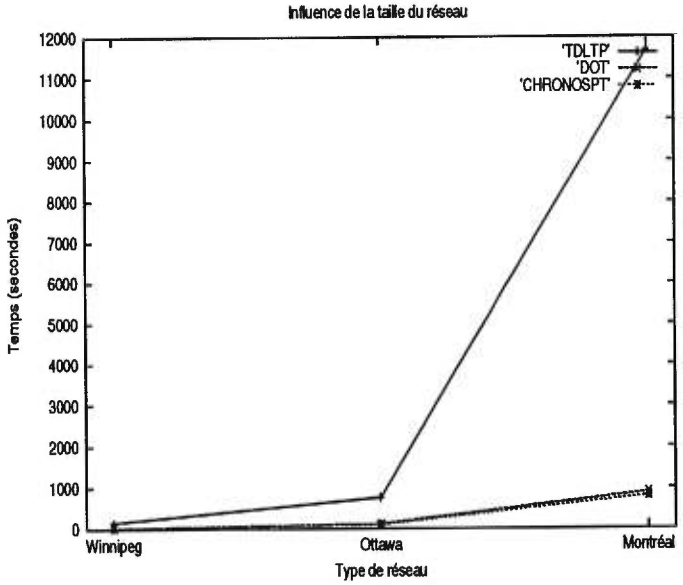

Figure 2.14: Influence de la taille du réseau sur le temps d'exécution.

Chapitre 3

Implantations parallèles d'algorithmes de calcul des plus courts chemins temporels

Plusieurs champs d'étude en sciences, incluant la recherche opérationnelle et la programmation mathématique, comportent des modèles qui requièrent des calculs de plus en plus intensifs. Bien que la puissance des processeurs augmente sans cesse, certains modèles exigent maintenant encore un effort de calcul trop grand pour espérer obtenir une solution en un temps raisonnable. Le calcul parallèle ([3, 7, 55]) représente donc aujourd'hui, une solution permettant de résoudre à moindre coût (temps) les problèmes auxquels nous sommes confrontés.

Quelques travaux ([10, 25, 59]) ont déjà contribué à étudier l'impact du calcul parallèle sur certains modèles de recherche opérationnelle et d'optimisation, notamment les algorithmes de calcul des plus courts chemins. Dans le but de résoudre le problème d'équilibre dans un réseau à demande fixe [24], Florian, Chabini et Le Saux [25] utilisent entre autre l'environnement **PVM** (*Parallel Virtual Machine*) pour le calcul distribué sur un réseau de 16 stations de travail *SUN SPARC Ultra 1*. Ils développent également une implantation parallèle **multithreads**[1] pour le calcul sur une *machine multiprocesseurs à mémoire partagée*[2] (*SUN SPARC Center 1000* à 8 processeurs). Leurs implantations utilisent le modèle de décomposition *maître-esclave*[3]. Les résultats qu'ils obtiennent leur permettent de conclure,

[1] Voir section 3.1.
[2] Voir section 3.1.
[3] Voir section 3.1.

entre autre, que l'implantation *multithreads* performe mieux que l'implantation utilisant l'environnement *PVM*. Un autre travail réalisé par Ziliaskopoulos, Mahmassani et Kotzinos [59], traite de l'implantation parallèle de l'algorithme *TDLTP* développé par Ziliaskopoulos et Mahmassani[4]. Dans ce travail, ils conçoivent deux algorithmes utilisant le concept de *mémoire partagée* et un algorithme utilisant une stratégie d'*échanges de messages*. Les deux designs à mémoire partagée sont implantés sur un *CRAY Y-MP/8* tandis que l'algorithme procédant par échanges de messages est implanté sur un *CRAY T3D* à 32 processeurs et utilise l'environnement *PVM*. Les résultats de Ziliaskopoulos, Mahmassani et Kotzinos, leur permettent de conclure que la parallélisation du calcul d'un plus court chemin conventionnel n'est pas efficace. Il s'avère en fait beaucoup plus avantageux d'affecter directement une destination à chaque processeur disponible. Ce résultat est également remarqué par Florian, Chabini et Le Saux.

Dans ce chapitre, nous nous intéressons à l'implantation parallèle des algorithmes de calcul des plus courts chemins temporels traités au chapitre 2[5]. Plus spécifiquement, nous utilisons, d'une part, l'environnement *PVM* pour étudier le comportement du calcul distribué sur un réseau de stations de travail et, d'autre part, la programmation *multithreads* pour évaluer l'impact du calcul sur une *machine à mémoire partagée*. Ce chapitre permet de procéder à une analyse sévère des performances des deux environnements de parallélisation dans le cadre du calcul des plus courts chemins temporels.

La première section de ce chapitre présente les deux types d'environnements choisis pour effectuer nos implantations parallèles, et introduit quelques concepts relatifs au traitement parallèle. Dans la deuxième section, nous présentons en détail les implantations parallèles qui ont été développées et nous effectuons une analyse des performances des deux environnements de parallélisation utilisés.

3.1 Traitement parallèle

Dans cette section, nous introduisons en premier lieu certains concepts reliés au traitement parallèle. Ensuite, nous présentons les deux environnements utilisés pour développer nos implantations parallèles. Pour terminer, nous définissons les mesures de performances utilisées pour analyser l'efficacité des implantations parallèles.

[4] Voir chapitre 1, section 1.2.3.

[5] La parallélisation des algorithmes de calcul des plus courts chemins statiques présentés au chapitre 2 est laissée de côté puisque déjà traitée par Florian, Chabini et Le Saux [25].

3.1.1 Quelques concepts de base reliés au traitement parallèle

L'utilisation du traitement parallèle nécessite une connaissance de certains concepts et introduit une nouvelle terminologie. Plusieurs des concepts de base relatifs au traitement parallèle sont définis par Chabini [11]. Nous reprenons ici ceux qui nous intéressent particulièrement.

La décomposition

La décomposition définit la division d'une tâche en sous-tâches pouvant être exécutées par plusieurs processeurs. Elle comprend les concepts suivants:

Le partionnement: Consiste à déterminer la taille des sous-tâches. On retrouve, en général, deux degrés de parallélisme découlant de ce concept:

- Parallélisme à grains fins: Chaque sous-tâche correspond (en général) à une instruction machine.
- Parallélisme à gros grains: Chaque sous-tâche correspond (en général) à une procédure comprenant un ensemble d'instructions machine.

L'allocation: Consiste à planifier l'exécution des sous-tâches en tenant compte des contraintes de précédence et, également, de l'architecture des processeurs.

Le balancement de charge: Vise à obtenir une meilleure efficacité en uniformisant les temps d'exécution des sous-tâches sur chacun des processeurs.

Les communications: Certaines sous-tâches doivent communiquer entre elles. Ces échanges d'information peuvent avoir des impacts sur la performance. Il est donc important de les minimiser.

La synchronisation: Ce concept fait référence aux deux modes selon lesquels un processeur peut évoluer: **synchrone** et **asynchrone**. Dans le mode synchrone, un processeur doit attendre à un point prédéterminé jusqu'à l'arrivée d'une donnée ou jusqu'à ce qu'un certain nombre d'étapes soient accomplies par d'autres processeurs. Le mécanisme choisi pour assurer la synchronisation peut avoir un impact non négligeable sur la performance. En mode asynchrone, il n'y a pas d'attente en aucun point de l'exécution. Cependant, il est plus difficile dans ce cas d'assurer la validité de l'algorithme.

Les architectures multiprocesseurs à mémoire partagée
(Shared-Memory Multiprocessors)

Dans ce modèle, chaque processeur a accès à une mémoire commune divisée en modules. Deux processeurs, source et récepteur, communiquent entre eux de la façon suivante: la source écrit l'information en mémoire commune de manière à ce que le récepteur puisse ensuite la lire. Même si ce modèle procure un moyen plus efficace de communiquer, il engendre un problème d'accès simultanés par plusieurs processeurs à l'information contenue dans une cellule mémoire. Les conflits, qui sont gérés par le réseau d'interconnections, retardent l'accès et, par conséquent, diminuent la vitesse des communications. Les communications demeurent toutefois peu coûteuses comparativement à d'autres types d'architectures, ce qui favorise un parallélisme à grains fins.

Les architectures multiprocesseurs à échanges de messages
(Message-Passing Multiprocessors)

Dans ce modèle, chaque processeur possède sa propre mémoire qu'il peut accéder directement. Il n'y a donc pas de conflits possibles, mais ce type d'architecture implique que la même information est répliquée autant de fois qu'il y a de processeurs. Par conséquent, l'espace mémoire total requis peut rapidement devenir problématique. Les communications avec les autres processeurs, qui se font par échanges de messages, doivent se faire à travers les liens physiques formant le réseau d'interconnections. Les communications sont donc plus lentes et le développement d'un algorithme pour ce type d'architecture est souvent plus difficile que pour une architecture à mémoire partagée. En raison du coût des communications, ce type d'architecture favorise davantage le parallélisme à gros grains.

3.1.2 L'environnement PVM (*Parallel Virtual Machine*)

L'environnement *PVM* (*Parallel Virtual Machine*) (Geist et *al.* [29]) utilise le modèle d'échanges de messages afin de permettre le calcul distribué à travers un ensemble hétérogène de machines. On parle de calcul distribué lorsqu'un ensemble de machines connectées par un lien physique (câble *Ethernet*) est utilisé dans le but de résoudre un problème qui est généralement de grande taille. L'objectif du calcul distribué est bien entendu de surpasser la puissance de calcul fournie par une seule machine. Le concept clé de l'environnement *PVM*, tel que décrit pas Geist et *al.* [29], est que l'ensemble des machines utilisées apparaissent en réalité comme une seule *machine virtuelle.*

L'environnement *PVM* est composé de deux parties. La première partie est un *daemon* (appelé *pvmd3*), résidant sur chacune des machines et formant ainsi la machine virtuelle. En fait, lorsqu'un usager désire utiliser l'environnement *PVM*, il doit en premier lieu créer une machine virtuelle en démarrant un *daemon* sur un ensemble de machines choisies. La seconde partie de l'environnement consiste en une librairie de fonctions composant l'interface *PVM*. Les fonctions contenues dans cette librairie permettent les échanges de messages, la création de processus, la coordination des tâches et la modification de la machine virtuelle.

Le modèle de calcul de l'environnement *PVM*[6] est basé sur la notion qu'une application est composée de plusieurs tâches, chaque tâche étant responsable d'une partie de la charge de travail totale. Cet environnement supporte deux types de méthode de parallélisation: le parallélisme fonctionnel et le modèle **SIMD** (*Single-Instruction Multiple-Data*). Dans le parallélisme fonctionnel, chaque tâche consiste à exécuter une fonction. Dans le parallélisme basé sur le modèle *SIMD*, toutes les tâches sont les mêmes. Cependant, chacune d'elles ne travaille que sur une partie (généralement petite) des données.

3.1.3 La programmation multithreads

Les *machines multiprocesseurs à mémoire partagée* procurent une solution peu coûteuse pour le calcul parallèle. En utilisant la programmation *multithreads* combinée à l'exécution sur ce type de machine, il est possible d'obtenir un rendement assez satisfaisant.

Avant de discuter des concepts concernant la programmation *multithreads*, il est nécessaire de définir le concept de *thread* lui-même. Ce concept est clairement défini par Berg et Lewis [6]. Comme les systèmes d'exploitation multitâches (par exemple, **UNIX** et **Windows NT**) permettent de réaliser plusieurs opérations simultanément en exécutant plus d'un processus, un processus peut faire de même en utilisant plusieurs *threads* [6]. Chaque *thread* est en fait une séquence d'instructions pouvant exécuter ses instructions indépendamment des autres *threads*, permettant ainsi à un processus *multithreads* d'exécuter un certain nombre de tâches en concurrence. La distinction entre un processus et un *thread* n'est pas évidente. Pour mieux la comprendre, Berg et Lewis ajoutent qu'un processus est une entité appartenant au noyau du système d'exploitation (*kernel-level entity*) tandis qu'un *thread* est une entité se situant au niveau utilisateur (*user-level entity*). Comme dans le cas de fonctions se situant au niveau utilisateur, la structure du *thread* peut être accédée directement en utilisant la librairie *multithreads* associée au système d'exploitation utilisé.

[6]Pour plus de détails concernant l'environnement *PVM*, consulter Geist et *al.* [29].

Il est important de noter que les registres (pointeur de pile (*stack pointer*), marqueur d'instructions (*program counter*)) font partie du *thread* et que chaque *thread* a sa propre pile d'exécution. Cependant, la structure du *thread* n'inclut pas le code exécutable. Ce dernier est plutôt global et peut être exécuté simultanément par d'autres *threads*. La figure 3.1 montre un exemple où deux *threads* (T1 et T3) exécutent la même fonction. On peut y voir

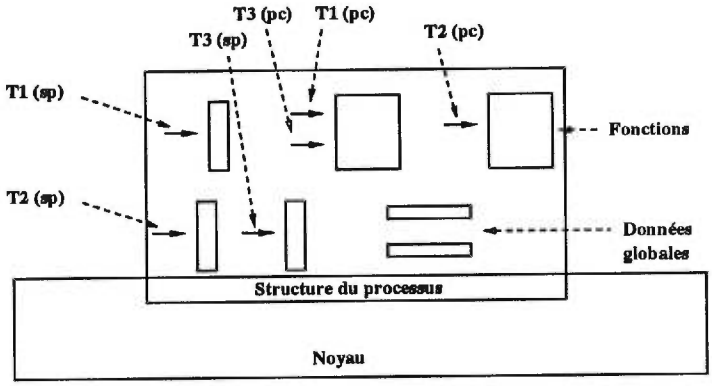

Figure 3.1: Relation entre un processus et les threads y étant associés.

les pointeurs de pile (sp) et les marqueurs d'instructions (pc) associés à chacun des trois *threads* présents.

Tous les *threads* d'un processus partagent le même état que ce processus. Ils résident dans le même espace mémoire, voient les mêmes fonctions et les mêmes données. Quand un *thread* modifie une variable propre au processus, alors tous les autres *threads* verront la modification lorsqu'ils accèderont à cette variable. Le prix à payer pour ce partage des ressources est que les données qui sont modifiées par un ou plusieurs *threads* doivent être protégées. Observons la suite d'instructions suivante où les *threads* T1 et T2 modifient la variable $D = 0$:

```
1. T1 lit D: T1.registre <- D=0
2. T2 lit D: T2.registre <- D=0
3. T1 incremente D de 1: T1.registre = T1.registre+1
4. T2 incremente D de 1: T2.registre = T2.registre+1
5. T1 reecrit dans D: T1.registre=1 -> D
6. T2 reecrit dans D: T2.registre=1 -> D
```

À la fin de cette suite d'instructions, comprenant deux incrémentations asynchrones, nous avons toujours $D = 1$. Afin d'obtenir $D = 2$, nous devons associer une clé à la variable D et prendre possession de cette dernière chaque fois que nous désirons modifier ou lire la variable:

```
1. T1 acquiere D.cle
2. T2 essai d'acquerir D.cle mais doit attendre que T1 la redonne
3. T1 lit D: T1.registre <- D=0
4. T1 incremente D de 1: T1.registre = T1.registre+1
5. T1 reecrit dans D: T1.registre=1 -> D
6. T1 redonne D.cle
7. T2 est reveille et acquiere D.cle
8. T2 lit D: T2.registre <- D=1
9. T2 incremente D de 1: T2.registre = T2.registre+1
10. T2 reecrit dans D: T2.registre=2 -> D
11. T2 redonne D.cle
```

Maintenant, nous avons que $D = 2$. Cependant, la suite d'instructions est devenue séquentielle et, de plus, les *threads* ont perdu un certain temps pour demander, acquérir et redonner la clé. Dans ce cas, il aurait été préférable d'avoir un seul *thread* exécutant la tâche. Il faut donc être prudent dans la conception d'un algorithme utilisant le concept *multithreads*.

Les problèmes de synchronisation que l'on peut rencontrer sont dûs principalement aux structures de données partagées (**SDP**). La présence de *SDP* dans une implantation *multithreads* a donc un impact sur les performances de cette dernière:

- Si aucune *SDP* n'est présente, alors aucun mécanisme de synchronisation n'est nécessaire. Dans ce cas, on doit s'attendre à de très bonnes performances.
- S'il y a une *SDP* qui est accédée en lecture seulement, alors il n'est également pas nécessaire d'introduire de mécanismes de synchronisation. Les performances devraient aussi être excellentes.
- S'il y a une *SDP* dont les données n'ont pas à être mises à jour immédiatement, il peut être avantageux pour chaque *thread* de conserver ses résultats intermédiaires dans l'espace mémoire lui étant réservé et de retarder la mise à jour de la *SDP* aussi longtemps que possible. De cette façon, les temps d'attente sont réduits à une petite portion du temps de calcul global et la performance reste satisfaisante.
- S'il y a une *SDP* qui doit être mise à jour immédiatement, alors les temps d'attente doivent être réduits au maximum. Ceci est possible si les *threads* acquièrent les clés associées à chacune des données protégées, pour de très courtes périodes de temps par rapport au temps de calcul global.

Il est maintenant évident que la programmation *multithreads* est soumise à certaines limites engendrées en partie par les *SDP*. Cependant, il est souvent possible de concevoir des algorithmes évitant ou limitant au maximum la présence de mécanismes de synchronisation.

3.1.4 Mesures de performances des implantations parallèles

La performance d'une implantation parallèle peut être mesurée de plusieurs façons ([11, 13, 14]). La mesure de cette performance dépend en quelque sorte du critère d'efficacité que l'on juge important. Dans notre cas, ce critère d'efficacité est le **temps d'exécution**. À partir de ce critère, nous pouvons reporter un certain nombre de mesures permettant d'évaluer la performance de nos implantations parallèles. Les mesures qui nous intéressent principalement dans ce mémoire sont l'**accélération** et la **charge relative**. Afin de définir ces deux mesures, dénotons par

- p: le nombre de processeurs;
- n: la taille du problème;
- $T_s(n)$: le temps d'exécution de l'implantation séquentielle, sur un des processeurs de la machine parallèle, pour résoudre le problème de taille n;
- $T(n,p)$: le temps d'exécution de l'implantation parallèle sur p processeurs pour résoudre le problème de taille n.

L'*accélération* $a(n,p)$ associée à l'exécution de l'implantation parallèle sur p processeurs est alors définie par

$$a(n,p) = \frac{T_s(n)}{T(n,p)}. \tag{3.1}$$

Bien qu'idéalement, $a(n,p) = p$, nous avons généralement que $a(n,p) < p$. À noter que nous utilisons, pour calculer l'accélération, le meilleur temps d'exécution séquentiel $T_s(n)$ et non le temps d'exécution de l'implantation parallèle sur 1 processeur $T(n,1)$. En effet, puisque nous avons que $T(n,1) \geq T_s(n)$ (en raison du coût supplémentaire attribuable aux communications ou aux mécanismes de synchronisation), l'accélération obtenue en utilisant le temps d'exécution de l'implantation parallèle sur 1 processeur est normalement non représentative de la réalité.

La *charge relative* est une nouvelle mesure de performance introduite par Chabini [11]. Idéalement, l'exécution parallèle dure $T_s(n)/p$. Désignons d'abord par la *charge des processeurs*, le temps supplémentaire (par rapport au temps idéal) que ceux-ci prennent pour finir le traitement. Nous incluons dans cette quantité toutes les formes de retard que peut causer une implantation parallèle:

- les délais de communications entre les processeurs;

- le temps d'attente des processeurs;
- le temps d'exécution des parties non parallélisables de l'algorithme;
- le temps supplémentaire causé par les instructions exclusives au code parallèle par rapport à l'implantation séquentielle;
- la variation dans le temps de calcul, due au fait qu'un algorithme parallèle peut emprunter un chemin qui diffère de celui de la version séquentielle.

Alors, la charge relative $b(n,p)$ est obtenue lorsque l'on divise la charge des processeurs par le temps d'exécution de l'implantation séquentielle. En pratique, cette mesure s'évalue de la façon suivante:

$$b(n,p) = \frac{T(n,p)}{T_s(n)} - \frac{1}{p}. \tag{3.2}$$

L'utilisation de la charge relative comme mesure de performance s'avère entre autre essentielle dans les cas où les courbes d'accélérations des implantations parallèles considérées sont difficilement distinguables. En effet, à chacune de ces courbes peut correspondre des charges relatives dont les comportements diffèrent, permettant ainsi de porter un jugement sur l'efficacité des implantations. Un autre avantage que procure l'utilisation de cette mesure repose sur la possibilité d'estimer la meilleure accélération que l'on pourrait obtenir en utilisant un grand nombre de processeurs p. En effet, puisque nous avons que

$$a(n,p) = \frac{1}{1/p + b(n,p)}, \tag{3.3}$$

nous obtenons, lorsque $p \to +\infty$,

$$a(n,p) \approx \frac{1}{b(n,p)}. \tag{3.4}$$

La charge relative nous offre donc la possibilité d'effectuer une analyse plus juste des performances de notre implantation parallèle. De plus, il est possible de faire des prévisions sur le comportement de l'implantation pour un nombre de processeurs hypothétiques.

3.2 Implantations parallèles des algorithmes *DOT*, *CHRONOSPT* et *TDLTP*

Cette section traite du développement d'implantations parallèles pour les trois algorithmes de calcul des plus courts chemins temporels traités au chapitre 2, à savoir, les algorithmes *DOT*, *CHRONOSPT* et *TDLTP*. L'environnement *PVM* et la programmation *multithreads* sont utilisés pour développer ces implantations parallèles.

Dans une premier temps, nous développons une implantation parallèle utilisant l'environnement *PVM*. L'objectif est d'effectuer un calcul distribué sur un réseau de stations de travail. Par la suite, à l'aide de la programmation *multithreads*, nous concevons une implantation parallèle destinée au calcul sur une machine multiprocesseurs à mémoire partagée. Les résultats obtenus par les deux types d'implantations sont analysés et comparés.

3.2.1 Implantation utilisant l'environnement *PVM*

L'environnement *PVM* (*Parallel Virtual Machine*), tel que décrit dans la section 3.1, est un modèle d'échanges de messages permettant le calcul distribué à travers un ensemble hétérogène de machines. Cet environnement supporte deux types de modèles de parallélisation: le modèle *fonctionnel* et le modèle *SIMD* (*Single-Instruction Multiple-Data*). Dans ce travail, nous nous restreignons au modèle de parallélisation *SIMD*[7].

Description de l'implantation

L'implantation développée, utilise le schéma *maître-esclave* présenté dans la figure 3.2. Ce schéma est essentiellement composé d'un programme *maître* et d'un certain nombre de copies d'un programme *esclave*. La tâche du programme *maître* englobe la répartition du travail, la collecte des résultats ainsi que la coordination de ces deux actions. Il gère en quelque sorte l'ensemble du traitement parallèle. D'un autre côté, la tâche du programme *esclave* est relativement simple. Elle consiste à exécuter le travail assigné par le programme *maître* et à lui communiquer les résultats obtenus. Les échanges de messages se font donc uniquement entre le *maître* et les *esclaves*.

La stratégie utilisée pour paralléliser les trois algorithmes de calcul des plus courts chemins temporels est très simple. Étant donné un nombre de destinations, notre stratégie consiste à diviser le travail en assignant une ou plusieurs destinations à chaque *esclave*. (Cette stratégie a entre autre été utilisée par Florian, Chabini et Le Saux [25].) L'objectif

[7] Voir section 3.1.

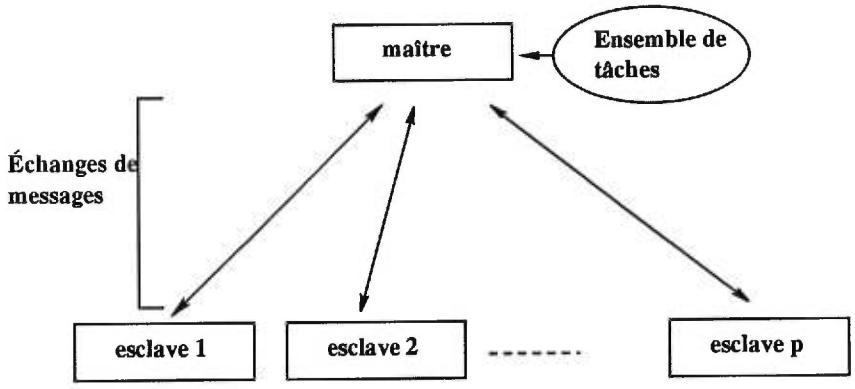

Figure 3.2: Schéma maître-esclave.

n'est donc pas de paralléliser le calcul lui-même, c'est-à-dire l'algorithme séquentiel, mais plutôt de considérer l'ensemble des destinations à traiter comme une tâche globale que l'on peut fragmenter en sous-tâches. Les opérations effectuées par les programmes *maître* et *esclave* peuvent se résumer comme suit:

Programme *maître*

1. Lit le fichier *NFF*.
2. Crée p copies du programme esclave.
3. Divise l'ensemble des destinations en K sous-ensembles.
4. Envoie, à chaque programme *esclave*, un sous-ensemble $k \in K$.
5. Tant qu'il y a un programme *esclave* actif,
   - 5a. reçoit les résultats du programme *esclave* i;
   - 5b. si $K \neq \emptyset$, envoie un nouveau sous-ensemble $k \in K$ au programme *esclave* i, sinon, désactive le programme *esclave* i.

Programme *esclave*

1. Lit le fichier *NFF*.
2. Crée un outil de calcul des plus courts chemins temporels.
3. Tant qu'il y a un sous-ensemble de destinations à traiter,
   - 3a. reçoit du programme maître un sous-ensemble de destinations k ou l'ordre de se désactiver;
   - 3b. pour chacune des destinations dans k, calcule les plus courts chemins temporels;
   - 3c. retourne les résultats au programme maître.

L'utilisation d'un environnement à échanges de messages sous-entend que chaque esclave doit posséder une copie des données nécessaires au traitement des opérations comprises dans le code qu'il exécute. Par conséquent, les programmes *maître* et *esclave* doivent tous deux lire le fichier *NFF* afin d'avoir en leur possession une copie du réseau traité. Ce

type d'implantation nous force donc à répliquer l'information en fonction du nombre de processeurs utilisés.

Le nombre de copies du programme *esclave*, démarrées par le programme *maître*, correspond généralement au nombre de processeurs disponibles lors de l'exécution. Afin de distribuer l'ensemble des destinations $|\mathcal{D}|$ parmi les p processeurs disponibles, nous utilisons une répartition définie par

$$\alpha \cdot \frac{|\mathcal{D}|}{p}, \ \alpha \in (0, 1].$$

Cette expression nous donne le nombre de sous-ensembles de destinations à traiter. Le paramètre α offre la possibilité d'effectuer un balancement dynamique de la charge de travail totale. Lorsque $\alpha = 1$ (ce qui correspond à la valeur utilisée dans nos expérimentations), nous obtenons un nombre de sous-ensembles de destinations, égal au nombre de processeurs. Cela signifie en fait que le nombre de destinations traitées par chaque processeur est fixé à l'avance, puisque chaque processeur sera responsable d'un et un seul sous-ensemble de destinations. Il s'agit donc d'un balancement statique de la charge de travail globale. Cependant, une valeur de $\alpha < 1$ résulte en un nombre K de sous-ensembles de destinations plus grand que le nombre de processeurs. Par conséquent, il est possible de gérer ces K sous-ensembles de destinations en assignant un sous-ensemble $k \in K$ au premier processeur libre, favorisant ainsi l'utilisation des processeurs les plus efficaces. Dans ce cas, il s'agit alors d'un balancement dynamique de la charge de travail globale. Un tel balancement de charge peut entre autre s'avérer avantageux si certains processeurs utilisés consacrent déjà, lors du calcul, une partie de leurs ressources à une autre application. Le choix de α dépend donc du degré d'utilisation des processeurs utilisés.

Le code *C++* présenté dans ce qui suit, correspond à l'implantation utilisant l'algorithme *DOT* (les implantations utilisant les algorithmes *CHRONOSPT* et *TDLPT* sont identiques). Afin de faciliter la compréhension du code, nous laissons de côté les appels aux procédures *PVM*. Les opérations correspondantes sont toutefois décrites et précédées du symbole ⇒.

Programme *maître*

```
...
main (int argc, char* argv[])
{
  int nbProcs;
  int k;
  if (argc == 2)
    nbProcs = 1;
    k = 1;
  else if (argc == 3)
    nbProcs = atoi(argv[2]);
    k = 1;
  else if (argc == 4) {
    nbProcs = atoi(argv[2]);
    k = atoi(argv[3]);
```

```
}
else {
  cerr << "usage: " << argv[0]
       << " <NFF network> [nb. procs] [k]\n";
  exit(1);
}

NFF* network = new NFF;
ifstream inputFile (argv[1]);
fichierNFF >> *network;

-> Cree p programmes esclave
-> Envoie le nom du fichier a traiter
   a chaque programme esclave

struct Task {
  int first;
  int end;
};

// Divise les destinations
Task* t;
int charge = k*nbCentroids/nbProcs;
int nbTask;
if (k != 1)
  nbTask = nbCentroids/charge;
else
  nbTask = nbProcs;
int r = nbCentroids%charge;

t = new Task[nbTasks];

int pos = 0;
for (int i = 0; i < r; i++) {
  t[i].first = pos;
  t[i].end = pos+charge+1;
  pos += charge+1;
}

for (i = r; i < nbTasks; i++) {
  t[i].first = pos;
  t[i].end = pos+charge;
  pos += charge;
}

for (i = 0; i < nbProcs; i++)
  -> Envoie un message de type 10 au
     processeur i
     comprenant t[i].first et t[i].end
pos = i;
procs = nbProcs;
while (procs) {

  -> Recoit un message du proceseurs j
     de type T

  while (T == 99) {

    if (pos < nbTasks) {
      -> Envoie un message de type 10 au
         processeur j
         comprenant t[pos].first et t[pos].end
      pos++;
    }
    else {
      -> Envoie un message de type 99
      procs--;
    }
  }

  if (T == 10)
    -> Resultats du processeurs j
 }
}
```

Programme *esclave*

```
#include <DOT.h>
...
void main ()
{
  -> Recoit du maitre le nom du fichier a
     traiter

  NFF* network = new NFF;
  ifstream inputFile (''fichier NFF'');
  inputFile >> *network;

  // Les attributs necessaires a la creation de
  // l'outil sont ajoutes
  ...

  // Cree un outil de calcul des plus courts
  // chemins temporels
  NFFTDOT toolDOT (*network);

  int nbPeriods = tool.numberOfPeriods();
  int nbNodes = tool.numberOfNodes();

  struct Path {
```

```
  int** arrivalTimes;
  int** nextNodes;
  int** nextTimes;
};

Path* p;

// Alloue l'espace memoire necessaire
// pour conserver l'information resultante
// du calcul des plus courts chemins

for (;;) {

  -> Recoit du maitre un message de type T

  if (T == 99)
    break;
  else
    -> Recoit deux entiers: first et end

  for (int i = first; i < end; i++)
    toolDOT.sp (nodeCentroid[i], arrivalTimes,
                nextNodes, nextTimes);

  -> Envoie au maitre un message de type 99
     (fin des calculs)

  -> Envoie les resultats au maitre
     (message de type 10)
  }
}
```

On remarque dans le programme *esclave* que les résultats sont communiqués seulement lorsque les calculs sont terminés. Cette approche diminue le nombre de messages à transmettre mais en augmente toutefois la taille. De plus, on augmente considérablement l'espace mémoire utilisé par le programme *esclave* (on doit conserver les résultats pour chaque destination traitée). Certains modèles peuvent exiger que les résultats soient disponibles le plus tôt possible afin d'effectuer une autre opération. Dans ce cas, une approche dans laquelle les résultats du calcul des plus courts chemins pour une destination seraient communiqués immédiatement, pourrait être utilisée.

Expérimentations et analyse des résultats

Nous nous attardons, dans cette partie, à l'analyse des performances de l'implantation parallèle présentée précédemment. Le réseau de 16 stations de travail *SUN SPARC Ultra 1/140* décrit à la section 2.1 est utilisé lors de nos expérimentations. Afin de tester notre implantation parallèle, nous utilisons les réseaux de Winnipeg et d'Ottawa pour des intervalles comprenant 30, 60, 90 et 120 périodes de temps ainsi que le réseau de Montréal pour des intervalles comprenant 30 et 60 périodes de temps. (On ne peut utiliser le réseau de Montréal pour des intervalles de 90 et 120 périodes de temps en raison de l'espace mémoire requis par ces exemplaires.)

Nous présentons dans les tableaux 3.I, 3.II et 3.III les temps d'exécution obtenus en fonction du type de réseau considéré (Winnipeg/30 périodes de temps, Winnipeg/60 périodes de temps, *etc.*) et du nombre de processeurs utilisés. Le temps d'exécution est mesuré en secondes et correspond au temps réel requis par l'application. Ces résultats sont obtenus en fixant le paramètre de balancement α à 1, ce qui correspond à un balancement de

charge statique. En fait, les expérimentations ont été réalisées dans des conditions idéales, c'est-à-dire que chaque processeur utilisé était totalement dédié à notre application. Il a été vérifié que dans ces conditions, un balancement de charge dynamique ($\alpha < 1$) est inutile. En effet, puisqu'un balancement de charge dynamique augmente le nombre des communications, il faut se retrouver dans une situation où il devient avantageux d'adopter cette stratégie. Or, cette augmentation du nombre des communications, produite par une fragmentation plus fine de la tâche globale, n'est bénéfique que si certains processeurs utilisés sont significativement moins efficaces à exécuter les tâches prescrites.

Nos expérimentations préliminaires nous ont permis de constater l'inefficacité de l'implantation lorsque l'on tient compte de la tansmission des résultats dans le temps de calcul total. En effet, les résultats que chaque *esclave* devrait normalement communiquer au *maître* sont contenus dans trois matrices[8] de dimensions $\mathcal{D}_k \times 12n|\mathcal{M}|$ *bytes*, où $\mathcal{D}_k$ représente le nombre de destinations contenues dans le sous-ensemble k. On voit clairement que le nombre de *bytes* que l'*esclave* doit transmettre au *maître* via le réseau, devient rapidement très important. Dans ce contexte, et même en ne transmettant qu'une seule matrice (par exemple, les temps d'arrivée à la destination), il est difficile d'obtenir une accélération supérieure à 2 sur 15 processeurs. Par conséquent, les résultats présentés dans les tableaux 3.I, 3.II et 3.III ne tiennent pas compte des temps de communication. Ces résultats servent donc à mesurer l'efficacité du calcul distribué sur un réseau de stations de travail.

À partir des résultats des tableaux 3.I, 3.II et 3.III, nous sommes en mesure d'évaluer et de représenter graphiquement l'**accélération** et la **charge relative**[9]. Ces deux mesures nous permettent de mieux évaluer l'impact de l'utilisation de l'environnement *PVM* pour le calcul distribué sur un réseau de stations de travail. Les figures 3.4 à 3.13 nous montrent, pour chacune des implantations parallèles des trois algorithmes (*DOT*, *CHRONOSPT* et *TDLTP*), les courbes associées à l'accélération et à la charge relative pour un réseau donné (Winnipeg/30 périodes de temps, Winnipeg/60 périodes de temps, *etc.*). Les graphiques obtenus nous permettent de constater que cette implantation parallèle est efficace. En effet, nous remarquons que les courbes d'accélération sont quasi linéaires, quelque soit le type de réseau traité. Nous remarquons également que nous atteignons dans la plupart des cas une accélération de 13 sur 15 processeurs. De plus, nous mesurons des charges relatives inférieures à 0.09, et nous observons que les courbes correspondantes sont décroissantes lorsque le nombre de processeurs augmente. Ce qui implique que le calcul des plus courts

[8]Voir chapitre 2, section 2.3.2.
[9]Voir section 3.1.

Réseau/$\|\mathcal{M}\|$	Nombre de processeurs														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Winnipeg/30	4.06	2.07	1.37	1.06	0.83	0.70	0.63	0.54	0.48	0.43	0.39	0.36	0.35	0.34	0.34
Winnipeg/60	8.84	4.48	3.02	2.29	1.81	1.52	1.29	1.17	1.04	0.94	0.82	0.78	0.66	0.65	0.58
Winnipeg/90	13.15	6.61	4.46	3.36	2.67	2.23	1.89	1.72	1.54	1.39	1.22	1.13	0.96	0.95	0.95
Winnipeg/120	16.15	8.20	5.55	4.16	3.31	2.78	2.35	2.13	1.92	1.72	1.52	1.41	1.29	1.20	1.18
Ottawa/30	20.95	10.50	7.00	5.49	4.37	3.70	3.19	2.75	2.49	2.24	2.08	1.89	1.72	1.64	1.48
Ottawa/60	44.87	22.57	15.08	11.36	9.18	7.54	6.49	5.79	5.09	4.58	4.21	3.86	3.50	3.32	3.15
Ottawa/90	69.04	31.38	21.21	17.32	14.00	11.63	9.94	8.99	7.82	6.99	6.46	5.93	5.37	5.13	4.86
Ottawa/120	73.24	36.72	24.49	18.92	14.90	12.29	10.61	9.81	8.32	7.46	7.05	6.32	5.84	5.62	5.61
Montréal/30	149.51	74.94	49.31	37.10	30.01	25.21	21.21	18.93	16.45	14.88	13.55	12.50	11.46	10.63	10.31
Montréal/60	290.14	147.34	97.43	74.08	58.75	49.93	42.77	37.31	33.02	29.83	26.19	24.10	23.29	21.53	20.29

Tableau 3.I: Implantation *PVM*: temps d'exécution de l'algorithme *DOT*.

Réseau/$\|\mathcal{M}\|$	Nombre de processeurs														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Winnipeg/30	2.96	1.49	0.99	0.73	0.60	0.49	0.43	0.38	0.34	0.30	0.28	0.27	0.25	0.24	0.23
Winnipeg/60	9.01	4.52	3.22	2.29	1.81	1.53	1.30	1.19	1.06	0.96	0.79	0.73	0.74	0.66	0.62
Winnipeg/90	13.62	6.84	4.64	3.50	2.79	2.36	2.03	1.82	1.65	1.48	1.37	1.27	1.20	1.14	1.09
Winnipeg/120	16.71	8.50	5.70	4.37	3.47	2.95	2.52	2.32	2.05	1.90	1.71	1.60	1.42	1.39	1.39
Ottawa/30	30.90	15.64	10.43	7.89	6.32	5.23	4.51	4.02	3.53	3.18	2.92	2.68	2.44	2.32	2.07
Ottawa/60	62.36	31.48	21.48	16.25	13.02	10.78	9.28	8.28	7.27	6.52	5.60	5.15	4.69	4.46	4.19
Ottawa/90	99.13	47.33	30.75	22.92	18.64	15.77	13.53	11.77	10.59	9.56	8.57	7.97	7.33	6.95	6.62
Ottawa/120	119.73	60.93	41.51	30.78	24.55	20.93	17.72	15.84	14.79	13.19	11.60	10.75	9.78	9.30	8.87
Montréal/30	226.20	113.95	75.52	57.38	45.76	37.85	32.57	28.68	25.44	22.80	21.13	19.30	17.64	16.76	14.78
Montréal/60	464.60	233.35	155.99	118.82	94.24	78.90	66.84	59.64	53.86	47.46	43.38	40.13	34.98	32.63	30.68

Tableau 3.II: Implantation *PVM*: temps d'exécution de l'algorithme *CHRONOSPT*.

Réseau/$\|\mathcal{M}\|$	Nombre de processeurs														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Winnipeg/30	14.33	7.27	4.82	3.62	2.91	2.46	2.09	1.87	1.69	1.51	1.36	1.26	1.12	1.07	1.02
Winnipeg/60	64.17	33.36	22.38	16.45	13.21	11.03	9.49	8.50	7.42	7.22	6.29	5.62	5.35	4.83	4.77
Winnipeg/90	110.26	55.91	37.03	28.59	22.56	18.86	16.08	14.08	12.48	12.29	10.45	9.57	9.09	8.08	7.95
Winnipeg/120	163.19	81.72	55.26	41.64	33.24	28.04	23.99	21.23	18.63	17.60	15.53	14.29	13.37	12.12	12.04
Ottawa/30	145.70	72.89	48.52	36.78	29.57	24.40	21.43	18.86	16.69	14.94	13.66	12.66	11.37	10.85	10.10
Ottawa/60	370.99	190.43	126.87	95.04	76.55	63.67	55.09	48.03	43.24	38.62	35.51	33.20	30.56	28.17	26.40
Ottawa/90	626.73	313.76	209.84	158.73	126.87	106.24	92.36	81.38	71.08	64.06	58.75	53.90	49.26	46.55	44.12
Ottawa/120	899.51	451.35	301.60	226.70	181.82	151.01	129.96	116.05	102.09	91.37	84.41	77.67	72.48	67.27	63.54
Montréal/30	1040.53	527.97	350.99	263.11	213.04	177.64	152.92	136.26	119.96	107.44	99.02	90.02	83.17	77.74	73.82
Montréal/60	2766.46	1363.65	911.71	685.97	566.31	472.40	403.41	350.91	314.38	280.44	253.81	244.51	225.41	198.30	201.55

Tableau 3.III: Implantation *PVM*: temps d'exécution de l'algorithme *TDLTP*.

chemins temporels représente donc suffisamment de travail (en ne tenant pas compte du temps requis pour la transmission des résultats) pour justifier l'utilisation de ce genre d'implantation parallèle. En utilisant les concepts que nous avons introduits dans la section 3.1, cela revient à dire que la **granularité** des sous-tâches obtenues pour les différents problèmes testés est suffisante et favorise l'utilisation de plusieurs processeurs.

Dans chacune des figures 3.4 à 3.13, il est intéressant de noter que l'on peut facilement distinguer les courbes de charges relatives contrairement à celles correspondant aux accélérations. On remarque en effet que pour chaque graphique, la courbe de la charge relative associée à l'algorithme *TDLTP* se trouve sous celle associée à l'algorithme *CHRONOSPT* qui, elle-même, se situe généralement sous la courbe correspondant à l'algorithme *DOT*. En fait, ce comportement est relié aux performances des implantations séquentielles de chacun des trois algorithmes (voir tableau 2.XII). Mis à part ce fait, la charge relative nous permet, comme nous l'avons vu dans la section 3.1, d'estimer la meilleure accélération que l'on pourrait obtenir si nous disposions d'un nombre illimité de processeurs. Évidemment, la qualité d'une telle estimation est étroitement reliée à la stabilité des courbes d'accélération et de charge relative. Il est donc important d'analyser le comportement de ces courbes avant de tirer des conclusions de ces estimés. Le tableau 3.IV présente, pour chacun des réseaux utilisés (Winnipeg/30 (W/30), *etc.*) et pour chacun des algorithmes considérés, les estimations obtenues en utilisant la relation 3.4. Afin d'obtenir ces estimés, nous utilisons la

Algorithme	W/30	W/60	W/90	W/120	O/30	O/60	O/90	O/120	M/30	M/60
DOT	50.0	166.7	142.9	142.9	100.0	58.8	47.6	90.9	100.0	100.0
CHRONOSPT	83.3	125.0	58.8	62.5	200.0	250.0	142.9	142.9	142.9	*1000.0*
TDLTP	166.7	111.1	166.7	142.9	250.0	250.0	200.0	200.0	200.0	142.0

Tableau 3.IV: Estimation de l'accélération potentielle.

dernière valeur de charge relative observée, c'est-à-dire celle correspondant à l'utilisation de 15 processeurs[10]. (On remarque que l'estimation de l'accélération de l'implantation parallèle de l'algorithme *CHRONOSPT* pour le réseau M/60 est de *1000.* Cette estimation optimiste est due à un rendement inespéré obtenu en utilisant 15 processeurs. L'analyse de la courbe d'accélération s'avère donc très importante dans ce cas précis.)

La figure 3.3 nous montre, pour le réseau de Montréal/30 et pour chacun des algorithmes considérés, une prévision du comportement des courbes d'accélérations que l'on pourrait obtenir si l'on disposait de plus de 15 processeurs. Ce réseau est choisi puisque

[10]Compte tenu de l'instabilité de certaines courbes, des estimations obtenues en fonction des cinq dernières valeurs de charge relative pourraient, dans certains cas, être plus précises.

les courbes d'accélération et de charge relative obtenues sont relativement stables. Les accélérations prévues sont calculées à l'aide de la relation 3.3, où $b(n,p)$ correspond toujours à la charge relative obtenue pour 15 processeurs. Les courbes obtenues permettent de confirmer que l'implantation *PVM* pourra également être efficace lorsqu'un nombre plus élevé de processeurs seront disponibles.

En résumé, pour justifier l'utilisation de l'environnement *PVM* pour le calcul distribué sur un réseau de stations de travail, il faut que le problème considéré puisse être fragmenté en sous-tâches de granularités suffisantes. De plus, il faut que les communications entre les processeurs soit considérablement négligeables comparées au temps de calcul total. Dans notre cas, le calcul des plus courts chemins temporels implique un effort de calcul suffisamment important pour favoriser l'utilisation de l'environnement *PVM*. Les accélérations ainsi que les charges relatives observées nous le confirment. Cependant, la masse de résultats à transmettre, pendant ou après les calculs, rend l'utilisation de l'implantation *PVM* inefficace, voire même inutile, pour tous les exemples considérés. Cette faiblesse de *PVM* dans le processus d'échanges de messages, déjà remarquée par Florian, Chabini et Le Saux [25], est particulièrement néfaste dans notre cas. Il est donc possible d'envisager d'utiliser l'environnement *PVM* pour le calcul des plus courts chemins temporels dans un contexte ne nécessitant pas une collecte instantanée des résultats. On peut imaginer, par exemple, une application où les résultats sont conservés sur disques et utilisés ultérieurement. On pourrait également imaginer un modèle dans lequel chaque *esclave* utilise le résultat de ses calculs pour exécuter une seconde tâche menant à un résultat final facilement communicable au programme *maître*.

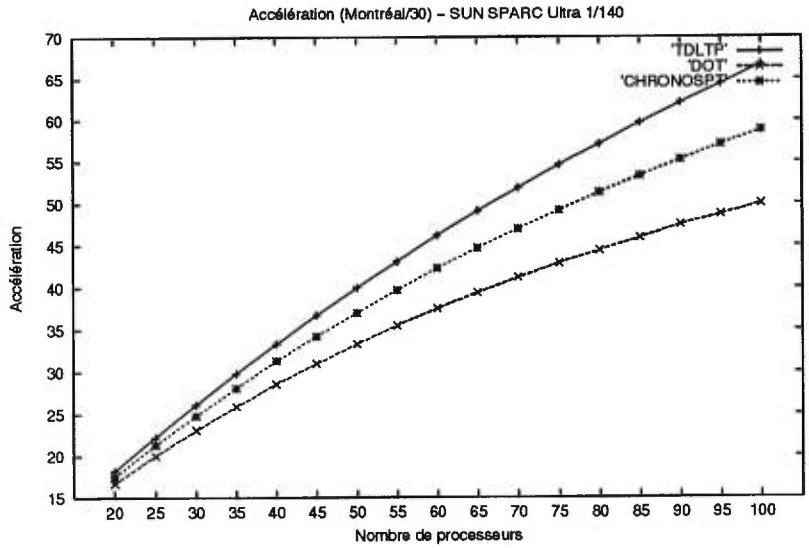

Figure 3.3: Accélération prévue sur plus de 15 processeurs.

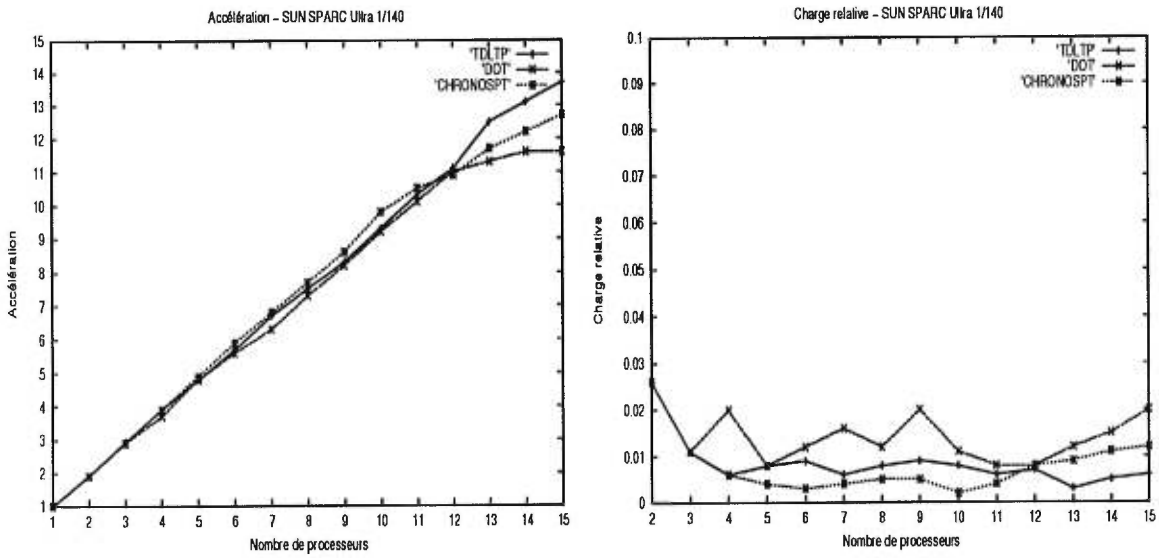

Figure 3.4: Accélération et charge relative - Winnipeg, 30 périodes de temps.

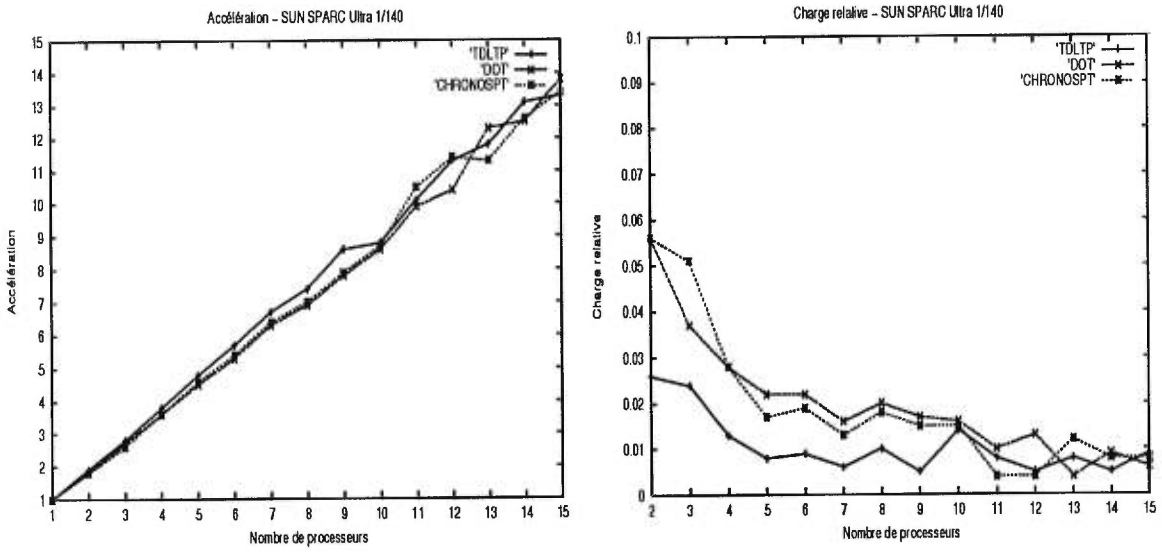

Figure 3.5: Accélération et charge relative - Winnipeg, 60 périodes de temps.

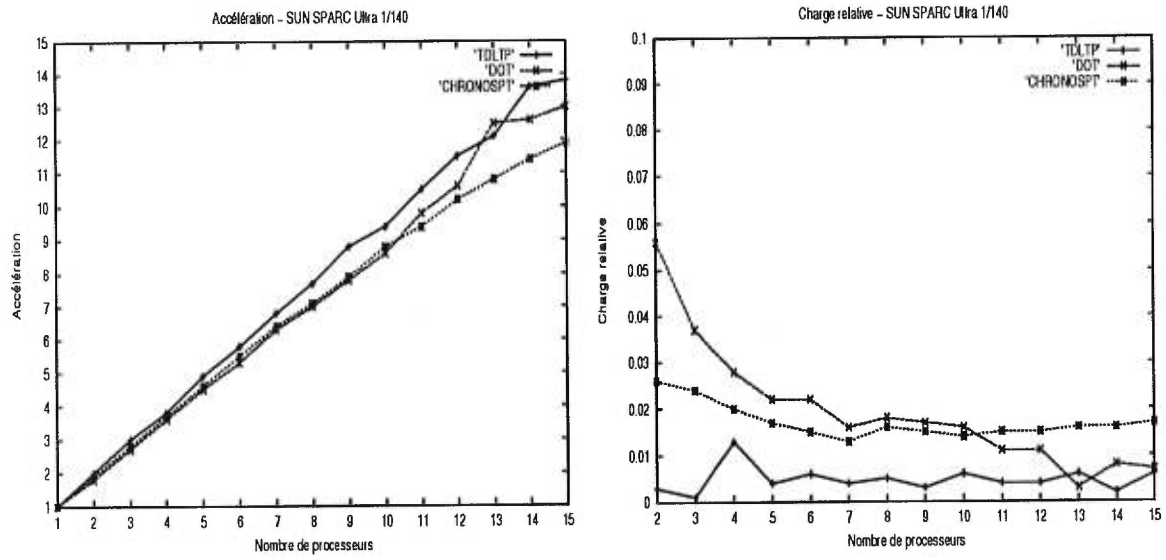

Figure 3.6: Accélération et charge relative - Winnipeg, 90 périodes de temps.

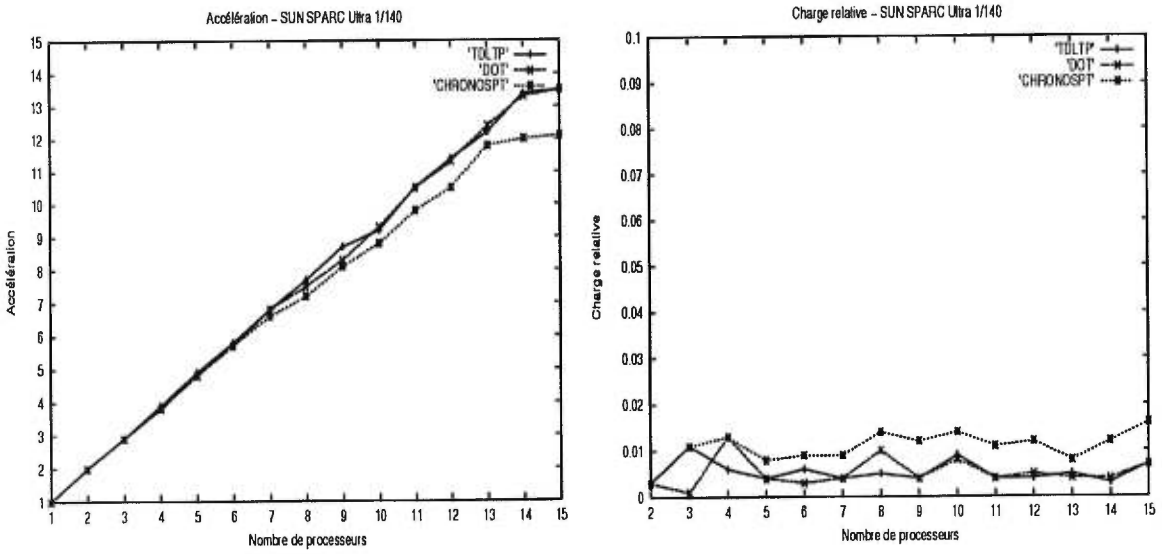

Figure 3.7: Accélération et charge relative - Winnipeg, 120 périodes de temps.

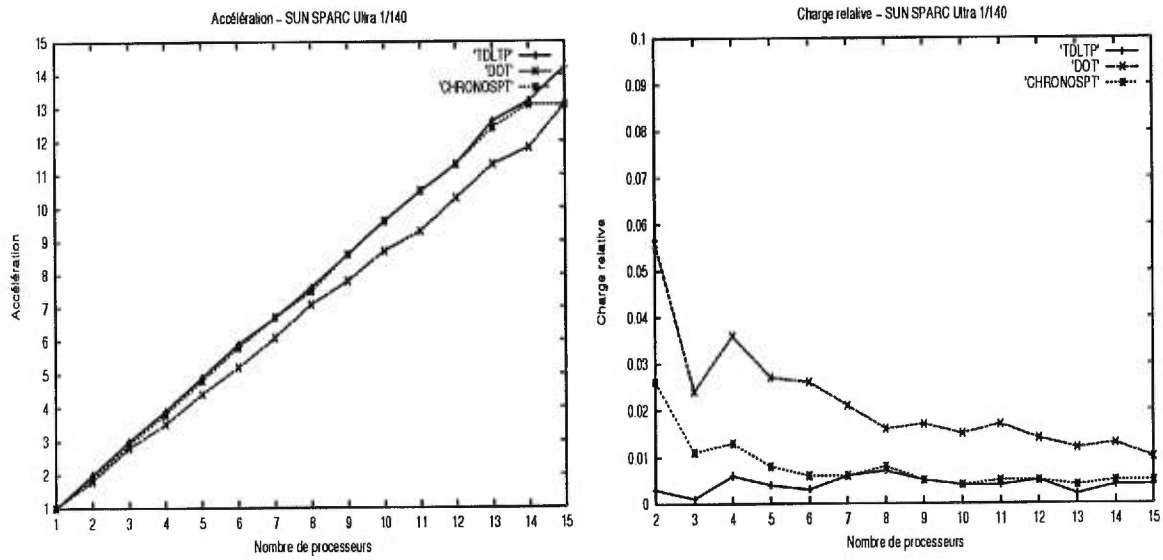

Figure 3.8: Accélération et charge relative - Ottawa, 30 périodes de temps.

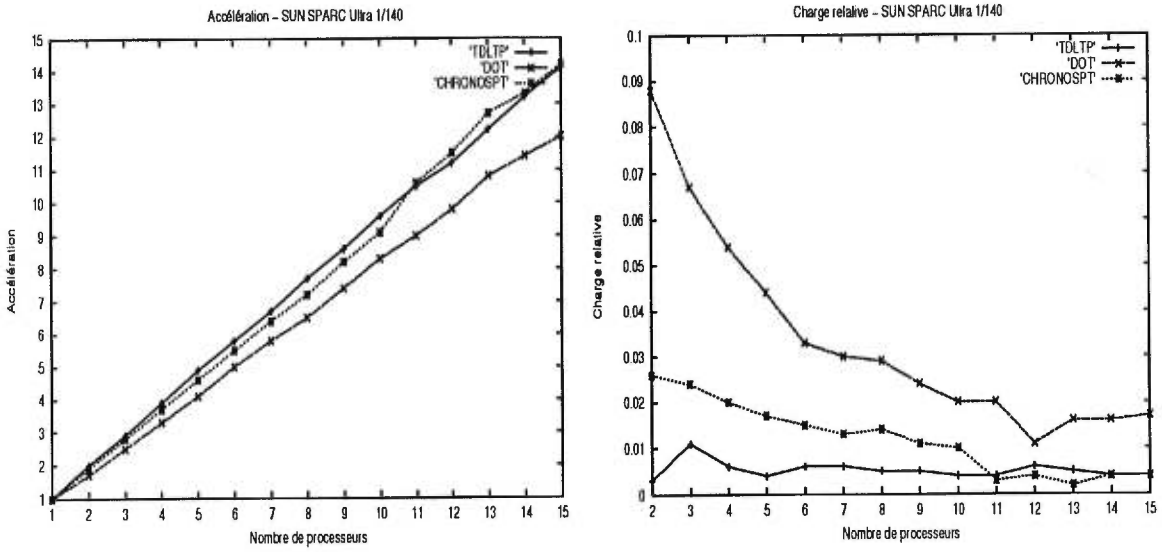

Figure 3.9: Accélération et charge relative - Ottawa, 60 périodes de temps.

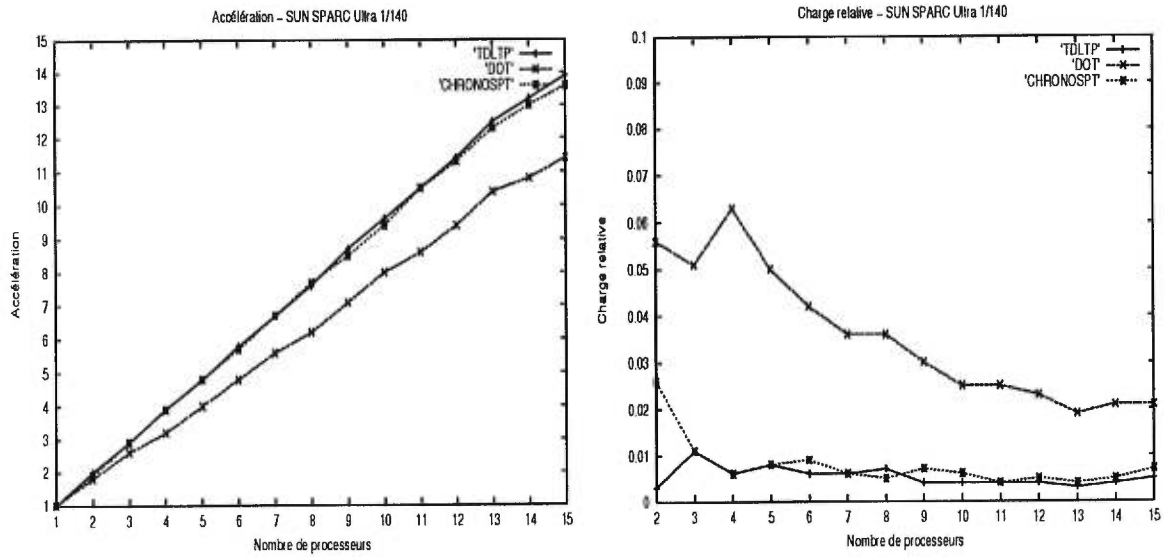

Figure 3.10: Accélération et charge relative - Ottawa, 90 périodes de temps.

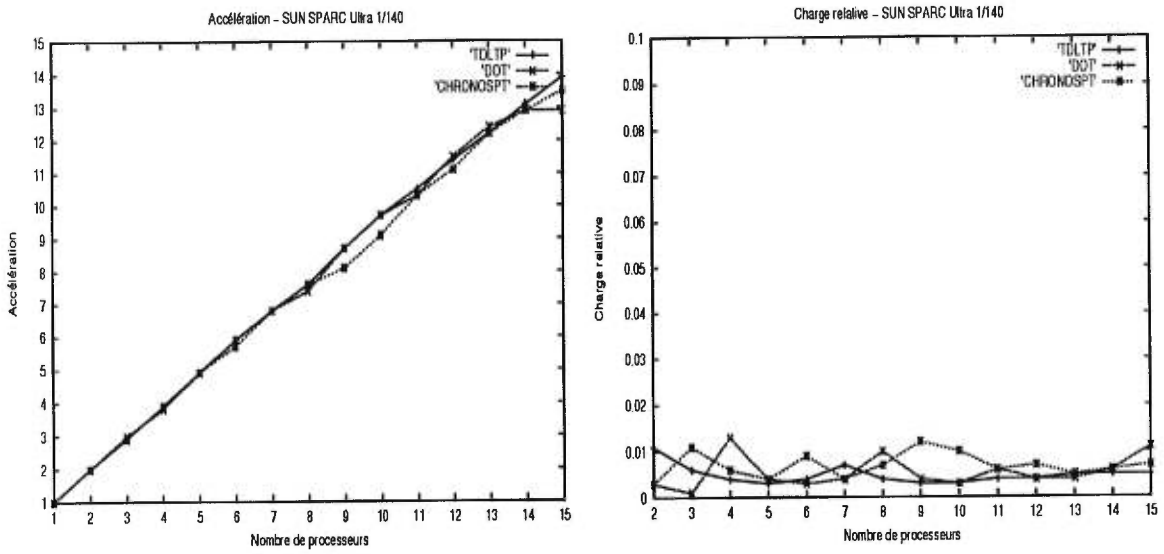

Figure 3.11: Accélération et charge relative - Ottawa, 120 périodes de temps.

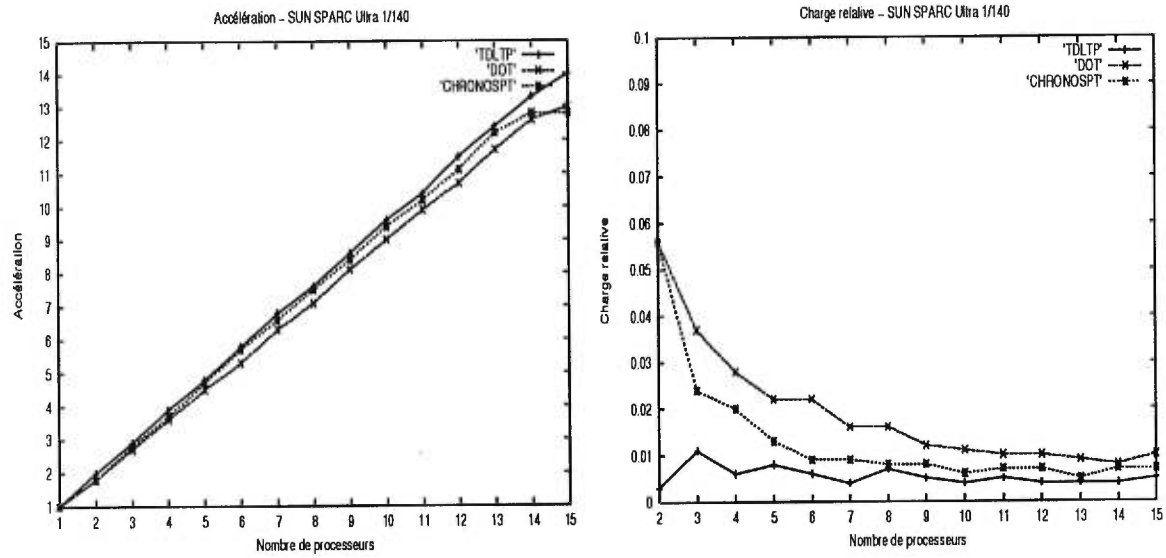

Figure 3.12: Accélération et charge relative - Montréal, 30 périodes de temps.

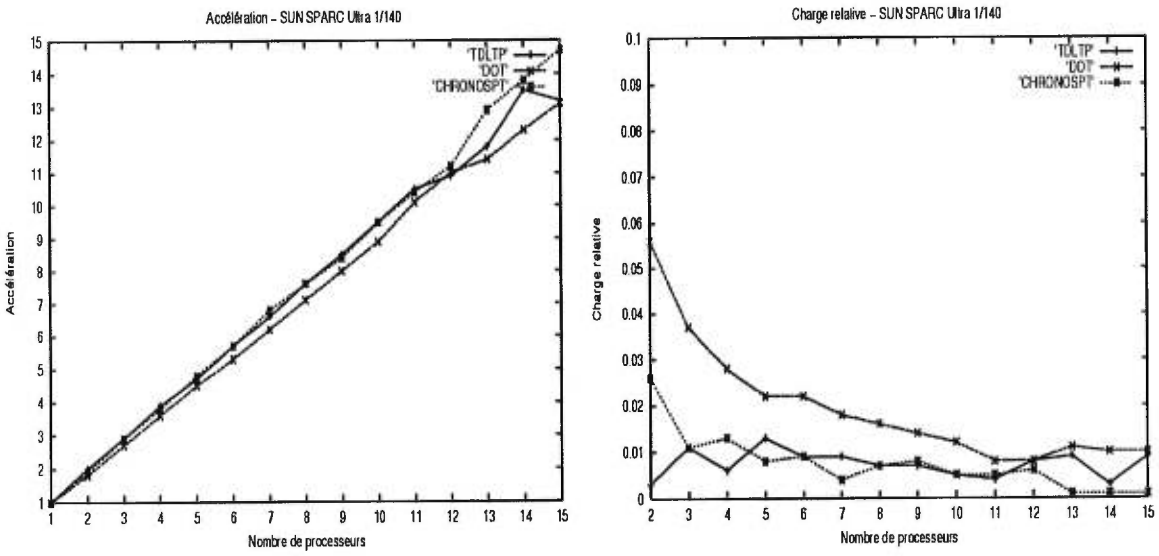

Figure 3.13: Accélération et charge relative - Montréal, 60 périodes de temps.

3.2.2 Implantation *multithreads*

L'utilisation de la programmation *multithreads* permet d'obtenir une implantation pouvant exploiter les caractéristiques d'une machine à mémoire partagée. En effet, dans ce type d'implantation, les communications (qui correspondent en réalité à des opérations de lecture et d'écriture) se font à travers la mémoire partagée. Donc, contrairement à l'implantation utilisant l'environnement *PVM*, aucune information n'est échangée puisqu'elle est commune. Cependant, l'accès à cette information doit être coordonné afin d'éviter les conflits, ce qui peut parfois diminuer l'efficacité de l'implantation parallèle.

L'implantation développée utilise également le schéma *maître-esclave* (voir figure 3.2). Chaque *esclave* (*thread*) est responsable du calcul des plus courts chemins pour un certain nombre de destinations. Toutefois, contrairement à l'implantation *PVM*, la distribution des tâches se fait au moyen d'un balancement totalement dynamique de la charge de travail globale. C'est-à-dire que chaque *thread* pige, aussitôt son calcul terminé, une nouvelle destination parmi l'ensemble des destinations non traitées.

Description de l'implantation

Contrairement à une implantation utilisant l'environnement *PVM*, où le programme *maître* et le programme *esclave* correspondent à deux fichiers exécutables distincts, l'utilisation de la programmation *multithreads* résulte en un seul code exécutable. Ainsi, à un point quelconque de l'exécution du programme *multithreads*, un certain nombre de *threads* sont créés (ou réactivés) afin d'exécuter, en parallèle, une tâche prescrite.

Bien que nous soyons en présence d'un seul code exécutable, le programme *multithreads* qui a été développé, permet de distinguer une partie *maître* et une partie *esclave*. Les opérations exécutées par chacune de ces parties sont les suivantes:

Partie *maître*

1. Lit le fichier *NFF*.
2. Crée p outils de calcul.
3. Crée p *threads* et attend la fin des calculs.

Partie *esclave*

1. Sélectionne un outil de calcul des plus courts chemins.
2. Tant qu'il y a des destinations à traiter,
   - 2a. sélectionne une destination parmi l'ensemble des destinations non traités;
   - 2b. calcule les plus courts chemins pour la destination choisie.

La partie *maître* comprend essentiellement trois étapes. La première étape consiste à lire le fichier *NFF*. Les opérations visant à fournir un objet *NFF* complet au constructeur de l'outil de calcul des plus courts chemins, sont comprises dans cette étape. Dans la deuxième étape, un nombre d'outils correspondant au nombre de *threads* désirés est créé. En effet, chaque *thread* doit posséder son propre outil afin de pouvoir exécuter ses calculs indépendamment des autres *threads*. Enfin, la troisième étape consiste à créer le nombre de *threads* désirés et à attendre la fin des calculs. D'un autre côté, les opérations effectuées par chaque *thread* se résument en deux étapes. La première étape consiste à choisir un outil de calcul des plus courts chemins, tandis que la seconde implique le choix et le traitement d'une destination tant que toutes les destinations n'ont pas été traitées.

Comme nous l'avons déjà vu dans la section 3.1, la programmation *multithreads* nécessite souvent l'usage d'un mécanisme de synchronisation lors de l'accès aux données partagées par les *threads*. Ce mécanisme de synchronisation s'avère nécessaire dans notre implantation afin de gérer, d'une part, la sélection d'un outil de calcul et, d'autre part, le processus de sélection des destinations. Le mécanisme de synchronisation utilisé dans notre implantation est le mécanisme de clé par exclusion mutuelle (*mutual exclusion lock*), présent dans la librairie *multithreads*. Ce mécanisme permet à un et un seul *thread* à la fois d'exécuter une certaine portion du code[11].

Le code *C++* présenté correspond à l'implantation *multithreads* de l'algorithme *DOT* (les implantations correspondant aux algorithmes *CHRONOSPT* et *TDLTP* sont identiques). Afin de faciliter la compréhension du code, nous laissons de côté les détails associés aux procédures de la librairie *multithreads*. Les opérations en question sont toutefois décrites mais précédées par le symbole ⇒.

```
#include <DOT.h>
...

// Variables globales
NFFAttributeRef _nodeCentroid;
NFFTDOT** _toolDOT;
int _pos;
int _threads;

struct Path {
  int** arrivalTimes;
  int** nextNodes;
  int** nextTimes;
};

Path* _p;

void spDOT ()
{
  -> aquiere la cle
  int t = _threads++;
  -> redonne la cle

  for(;;) {

    -> aquiere la cle
    if (!_pos) {
      -> redonne la cle
      break;
    }
```

[11] Pour plus de détails, voir Berg et Lewis [6].

```
    int i = --_pos;
    -> redonne la cle

    _toolDOT[t]->sp (_nodeCentroid[i],
    _p[t].arrivalTimes, _p[t].nextNodes,
    _p[t].nextTimes);
  }
}

main (int argc, char* argv[])
{
  int nbThreads;
  if (argc == 2)
    nbThreads = sysconf(_SC_NPROCESSORS_ONLN);
  else if (argc == 3)
    nbThreads = atoi(argv[2]);
  else {
    cerr << "usage: " << argv[0]
         << " <NFF network> [nb. threads]\n";
    exit(1);
  }

  NFF* network = new NFF;
  ifstream fichierNFF (argv[1]);
  fichierNFF >> *network;

  // Les attributs necessaires a la creation de
  // l'outil sont ajoutes
  ...

  int nbPeriods = tool.numberOfPeriods();
  int nbNodes = tool.numberOfNodes();

  _toolDOT = new NFFTDOT*[nbThreads];
  for (int i = 0; i < nbThreads; i++)
     _toolDOT[i] = new NFFTDOT (*network);

  // Pour conserver les resultats
  _p = new Path[nbThreads]

  for (int k = 0; k < nbThreads; k++) {

    _p[k].arrivalTimes = new int[nbPeriods];
    for (int i = 0; i < nbPeriods; i++)
      _p[k].arrivalTimes[i] = new int[nbNodes];

    _p[k].nextNodes = new int[nbPeriods];
    for (i = 0; i < nbPeriods; i++)
      _p[k].nextNodes[i] = new int[nbNodes];

    _p[k].nextTimes = new int[nbPeriods];
    for (i = 0; i < nbPeriods; i++)
      _p[k].nextTimes[i] = new int[nbNodes];
  }

  _threads = 0;
  _pos = centroids.size();

  -> cree nbThreads qui executent la fonction
     spDOT
  -> attend que les nbThreads aient termine
     leurs calculs
}
```

La fonction *spDOT()* correspond au code exécuté simultanément par les p *threads* créés. Comme on peut le constater, le mécanisme de synchronisation s'assure en premier lieu que chaque *thread* sélectionne un outil *DOT* distinct et, par la suite, voit à ce qu'une destination ne soit pas traitée par plusieurs *threads* à la fois. Dès qu'un *thread* termine de traiter une destination, il essaie immédiatement d'acquérir la clé dans le but de choisir la prochaine destination (s'il y en a une) dans la liste.

Le programme *multithreads* développé peut être exécuté sur une machine composée d'un ou de plusieurs processeurs. Le nombre de *threads* démarrés correspond, par défaut, au nombre de processeurs présents sur la machine. Par conséquent, notre implantation s'assure d'utiliser au maximum les ressources (processeurs) disponibles afin de fournir l'exécution la plus efficace possible. L'exécution du programme *multithreads* sur une machine à un processeur se comporte donc «exactement» comme la version séquentielle correspondante.

Toutefois, un coût d'exécution supplémentaire, dû à la présence d'un mécanisme de synchronisation, est évidemment observé.

Expérimentations et analyse des résultats

Nous nous attardons, dans cette partie, à l'analyse des performances de l'implantation *multithreads* présentée précédemment. Le *SUN SPARC Server 1000* à 8 processeurs à mémoire partagée décrit dans la section 2.1, est l'environnement parallèle utilisé pour tester cette implantation. Les réseaux de Winnipeg, Ottawa et Montréal pour des intervalles comprenant 30, 60, 90 et 120 périodes de temps, sont utilisés lors des expérimentations.

Nous présentons dans les tableaux 3.V, 3.VI et 3.VII, les temps d'exécution obtenus en fonction du type de réseau considéré (Winnipeg/30 périodes de temps, Winnipeg/60 périodes de temps, *etc.*) et du nombre de *threads* utilisés. Le temps d'exécution est mesuré en secondes et correspond au temps réel requis par l'application. Puisque nous disposons de 8 processeurs, nous utilisons un maximum de 8 *threads* lors des expérimentations.

Réseau/$\|\mathcal{M}\|$	Nombre de *threads*							
	1	2	3	4	5	6	7	8
Winnipeg/30	9.02	4.90	3.44	2.31	2.00	1.82	1.36	1.31
Winnipeg/60	22.66	12.03	7.49	5.46	4.40	3.75	3.11	2.70
Winnipeg/90	39.13	18.90	12.39	9.11	7.48	6.13	5.24	4.57
Winnipeg/120	51.30	26.48	17.20	12.88	10.26	8.47	7.13	6.34
Ottawa/30	55.40	27.72	17.77	13.74	11.40	9.24	7.78	6.82
Ottawa/60	120.50	64.07	40.72	30.65	23.90	19.85	17.01	14.82
Ottawa/90	196.18	79.09	54.30	41.10	39.70	30.31	26.13	22.74
Ottawa/120	259.42	132.63	80.21	60.71	48.51	40.03	34.02	30.25
Montréal/30	487.07	239.70	156.03	120.26	94.82	78.20	67.10	58.76
Montréal/60	957.32	471.01	316.42	233.89	186.85	153.36	132.43	116.39
Montréal/90	1457.92	735.55	469.62	361.01	285.97	241.75	202.71	179.23
Montréal/120	1866.33	901.75	615.44	455.48	370.97	311.84	279.32	241.30

Tableau 3.V: Implantation *multithreads*: temps d'exécution de l'algorithme *DOT*

À partir des résultats des tableaux 3.V, 3.VI et 3.VII, nous sommes en mesure d'évaluer et de représenter graphiquement l'**accélération** et la **charge relative**. Les figures 3.15 à 3.26 nous montrent, pour chacune des implantations *multithreads* des trois algorithmes (*DOT*, *CHRONOSPT* et *TDLTP*), les courbes associées à l'accélération et à la charge relative pour un réseau donné (Winnipeg/30 périodes de temps, Winnipeg/60 périodes de temps, *etc.*). Les courbes obtenues réflètent bien l'efficacité de ce type d'implantation parallèle. Comme dans le cas de l'implantation utilisant l'environnement *PVM*, nous constatons que les courbes d'accélération sont quasi linéaires et nous atteignons, en

Réseau/$\|\mathcal{M}\|$	Nombre de *threads*							
	1	2	3	4	5	6	7	8
Winnipeg/30	9.07	4.90	3.43	2.31	2.23	1.69	1.51	1.33
Winnipeg/60	25.00	12.17	8.09	6.16	5.19	4.10	3.55	3.08
Winnipeg/90	40.91	20.48	12.82	9.75	7.86	6.90	6.03	5.21
Winnipeg/120	56.53	29.d185	20.27	14.51	11.28	10.06	8.56	7.68
Ottawa/30	47.55	23.58	16.43	12.60	9.73	8.14	7.15	6.18
Ottawa/60	111.77	58.11	37.50	27.80	24.50	18.23	15.85	14.56
Ottawa/90	180.28	90.61	59.25	44.47	38.05	30.26	27.32	22.20
Ottawa/120	237.36	119.75	87.47	59.97	52.53	41.24	37.20	32.57
Montréal/30	383.43	193.25	129.32	97.79	80.95	78.84	58.06	50.52
Montréal/60	804.08	403.04	271.92	228.69	170.91	146.80	126.84	105.73
Montréal/90	1447.06	735.74	491.95	334.73	295.36	234.18	201.07	177.88
Montréal/120	2108.87	1157.77	698.82	512.19	410.18	359.31	330.32	323.91

Tableau 3.VI: Implantation *multithreads*: temps d'exécution de l'algorithme *CHRONOSPT*.

Réseau/$\|\mathcal{M}\|$	Nombre de *threads*							
	1	2	3	4	5	6	7	8
Winnipeg/30	42.48	26.19	20.61	11.61	9.65	10.20	6.90	5.98
Winnipeg/60	152.70	84.12	50.93	37.94	29.62	25.69	21.82	18.72
Winnipeg/90	284.17	141.36	93.70	70.94	56.49	46.57	40.05	35.49
Winnipeg/120	468.11	242.56	158.16	115.27	91.06	75.98	65.07	56.76
Ottawa/30	279.23	140.97	94.79	71.18	56.69	46.15	39.30	34.57
Ottawa/60	875.52	429.21	283.55	214.23	164.99	138.03	117.56	101.63
Ottawa/90	1528.86	766.68	510.08	379.60	300.69	248.61	220.84	187.90
Ottawa/120	2310.87	1142.58	760.91	571.56	441.80	366.84	313.55	273.27
Montréal/30	2554.18	1280.91	897.11	648.16	522.86	433.05	366.70	321.54
Montréal/60	7015.45	3512.11	2374.84	1756.78	1385.41	1153.34	996.37	900.35
Montréal/90	12649.70	6206.59	4214.83	3084.31	2459.92	2043.63	1727.24	1545.25
Montréal/120	19418.70	10105.30	6441.43	4841.89	3933.71	3287.49	2802.25	2414.32

Tableau 3.VII: Implantation *multithreads*: temps d'exécution de l'algorithme *TDLTP*.

moyenne, une accélération de 7.5 lorsque 8 *threads* sont utilisés. Dans le cas des charges relatives nous observons, à l'exception du réseau de Winnipeg pour 30 périodes de temps, des valeurs inférieures à 0.06. De plus, les courbes de charges relatives nous permettent de constater, en général, une certaine décroissance. Ce comportement, que nous avons également observé dans l'analyse des résultats de l'implantation *PVM*, laisse entrevoir que l'utilisation d'un nombre plus élevé de *threads* pourrait améliorer davantage les temps de calcul. En supposant que nous disposons d'un nombre illimité de *threads*, il nous est possible en utilisant la relation 3.4, d'obtenir une estimation de la meilleure accélération que nous pourrions atteindre. Comme nous l'avons mentionné dans l'analyse des résultats de l'implantation *PVM*, la qualité d'une telle estimation est reliée à la stabilité des courbes d'accélération et de charge relative. L'analyse du comportement de ces courbes est donc essentielle avant de

pouvoir tirer une conclusion de l'estimation en question. Le tableau 3.VIII présente, pour chacun des réseaux utilisés (Winnipeg/30, *etc.*) et pour chacun des algorithmes considérés, les estimations obtenues. Afin d'obtenir ces estimés, nous utilisons la dernière valeur de

Algorithme	W/30	W/60	W/90	W/120	O/30	O/60
DOT	45.5	500.0	333.3	200.0	500.0	500.0
CHRONOSPT	41.5	200.0	83.3	71.4	142.9	83.3
TDLTP	55.6	500.0	333.3	*5000.0*	500.0	100.0
Algorithme	O/90	O/120	M/30	M/60	M/90	M/120
DOT	*5000.0*	200.0	500.0	142.9	500.0	142.9
CHRONOSPT	500.0	83.3	142.9	142.9	333.3	29.4
TDLTP	500.0	200.0	200.0	200.0	142.9	*5000.0*

Tableau 3.VIII: Estimation de l'accélération potentielle.

charge relative observée, c'est-à-dire celle correspondant à l'utilisation de 8 *threads*[12]

La figure 3.14 nous montre, pour le réseau de Montréal/30 et pour chacun des algorithmes considérés, une prévision du comportement des courbes d'accélérations que l'on pourrait obtenir s'il était possible d'utiliser plus de 8 *threads*. Les accélérations prévues sont calculées à l'aide de la relation 3.3, où $b(n,p)$ correspond toujours à la charge relative obtenue pour 8 *threads*. Les courbes obtenues nous montrent clairement que le temps de calcul des plus courts chemins peut, pour chacun des algorithmes, être amélioré en augmentant le nombre de *threads*. On remarque également que la courbe associée à l'algorithme *DOT*, laisse entrevoir un taux de croissance relativement constant pour un nombre de *threads* plus grand que 100. Ce comportement est caractéristique de l'implantation *multithreads* puisque ce type de parallélisme n'est que très peu influencé par la diminution de la granularité de la tâche à accomplir.

En résumé, l'utilisation de la programmation *multithreads* s'avère, tout comme l'utilisation de l'environnement *PVM*, très efficace pour le calcul des plus courts chemins temporels. Cependant, cette stratégie de parallélisation s'avère globalement plus efficace que l'utilisation de l'environnement *PVM*, compte tenu de l'absence de communications de résultats. En effet, la mémoire partagée nous évite d'une part à transmettre l'information nécessaire au calcul des plus courts chemins et d'autre part, à transmettre les résultats obtenus suite au calcul des plus courts chemins.

L'ensemble des résultats présentés nous a permis de constater l'efficacité des deux approches de parallélisation pour le calcul des plus courts chemins temporels, et l'avantage que procure la programmation *multithreads* pour ce type de problème. Toutefois, bien que

[12]Compte tenu de l'instabilité de certaines courbes, des estimations obtenues en fonction des cinq dernières valeurs de charge relative pourraient, dans certains cas, être plus précises.

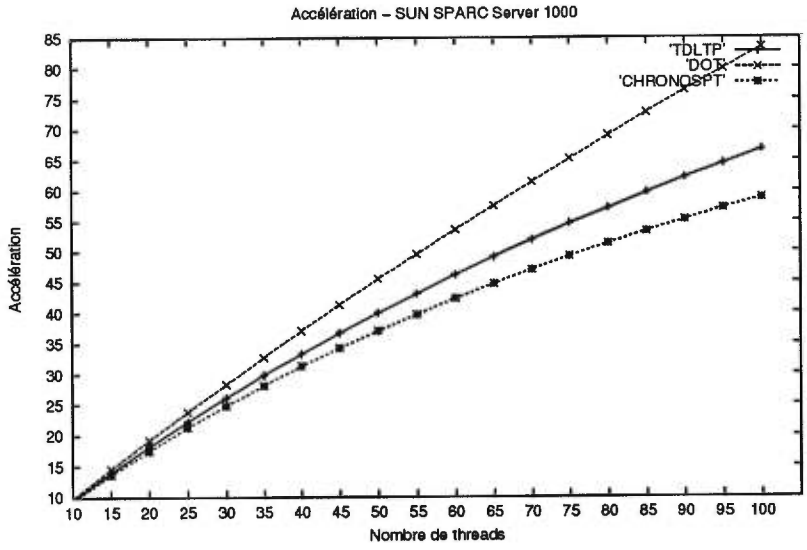

Figure 3.14: Accélération prévue sur plus de 8 *threads*.

notre analyse des résultats, basée principalement sur les mesures d'accélération et de charge relative, nous a mené vers cette conclusion, le temps réel de calcul demeure toujours un facteur déterminant dans le choix d'une stratégie de parallélisation. En ce sens, si nous avons à notre disposition 16 *SUN SPARC Ultra 1/140* connectés par un réseau de 100 *Mbits/sec*, les résultats des tableaux 3.I, 3.II et 3.III nous montrent que l'utilisation de l'environnement *PVM* nous procure des temps de calcul totaux bien inférieurs à ceux obtenus par le calcul sur le *SUN SPARC Server 1000*. Il nous reste cependant à trouver un moyen efficace de communiquer les résultats sans trop hypothéquer les temps de calcul bruts.

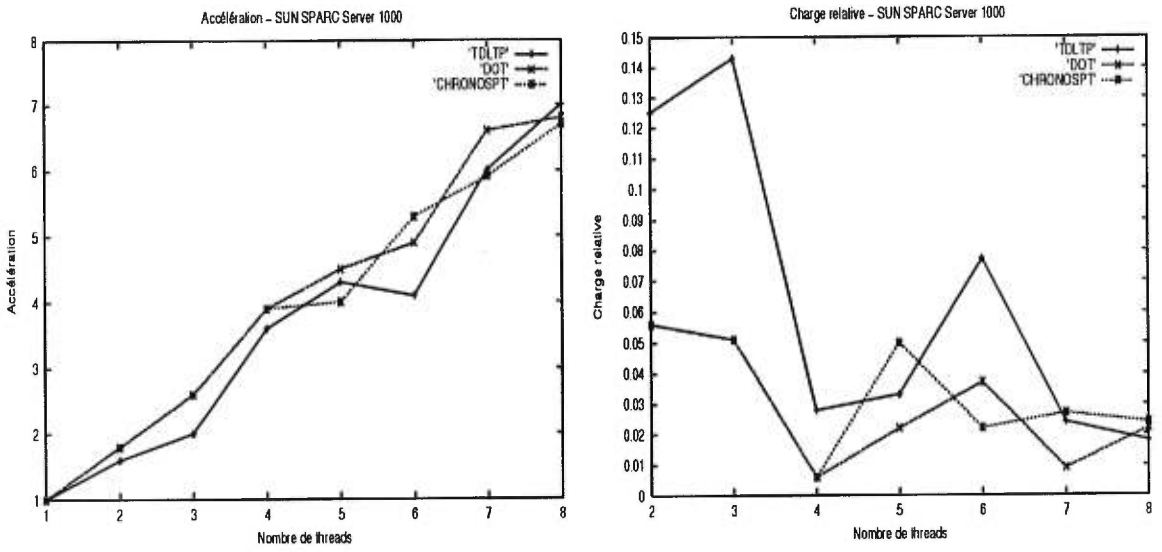

Figure 3.15: Accélération et charge relative - Winnipeg, 30 périodes de temps.

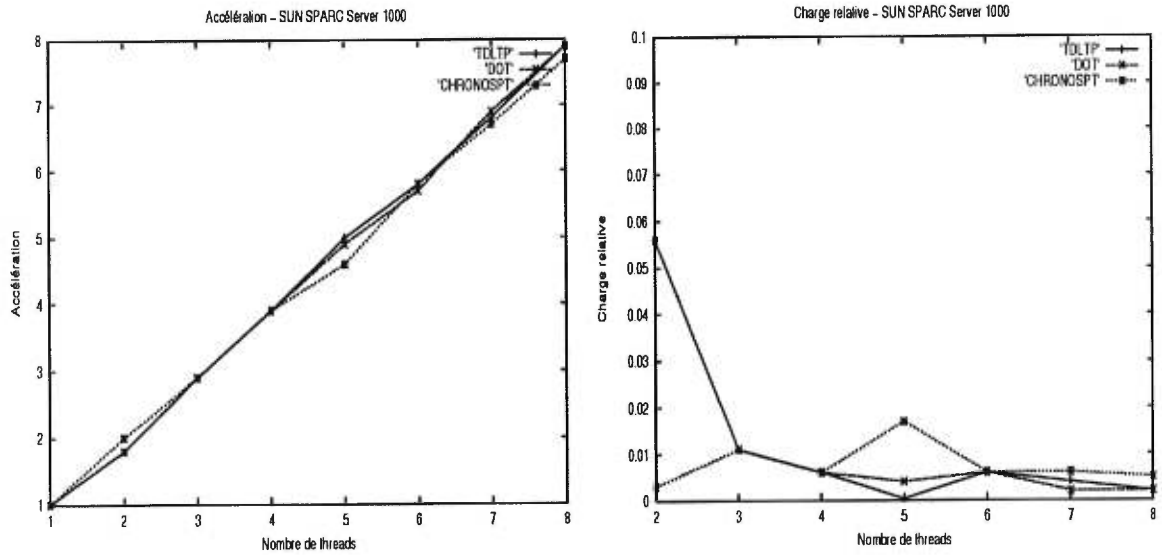

Figure 3.16: Accélération et charge relative - Winnipeg, 60 périodes de temps.

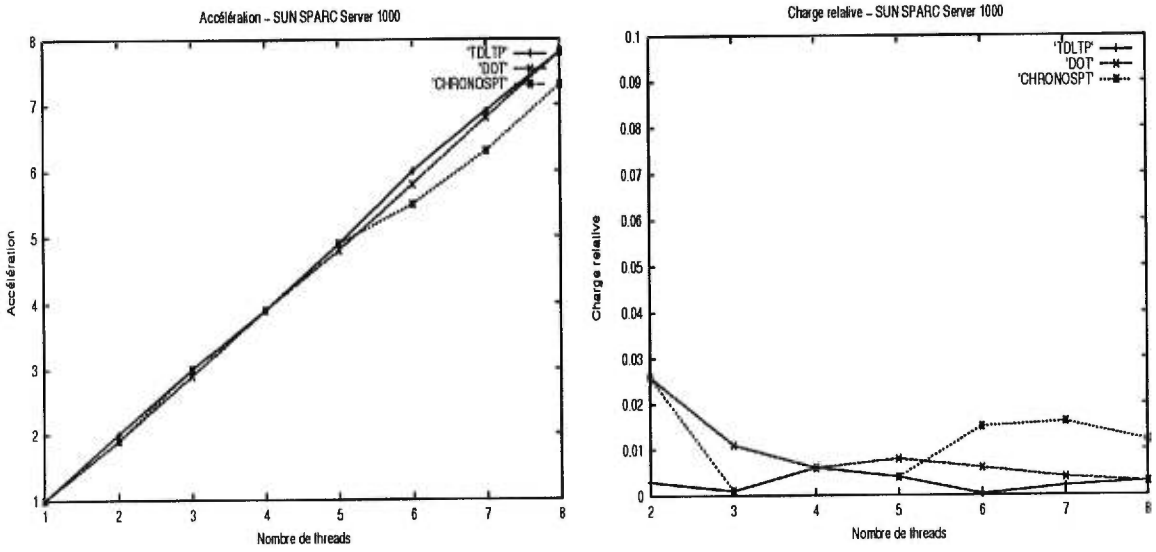

Figure 3.17: Accélération et charge relative - Winnipeg, 90 périodes de temps.

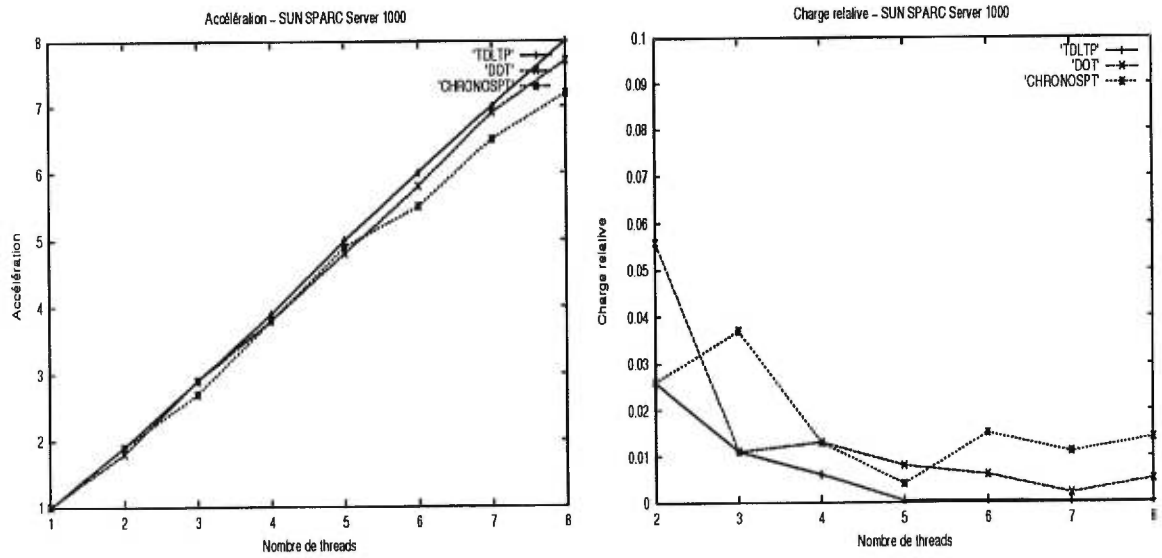

Figure 3.18: Accélération et charge relative - Winnipeg, 120 périodes de temps.

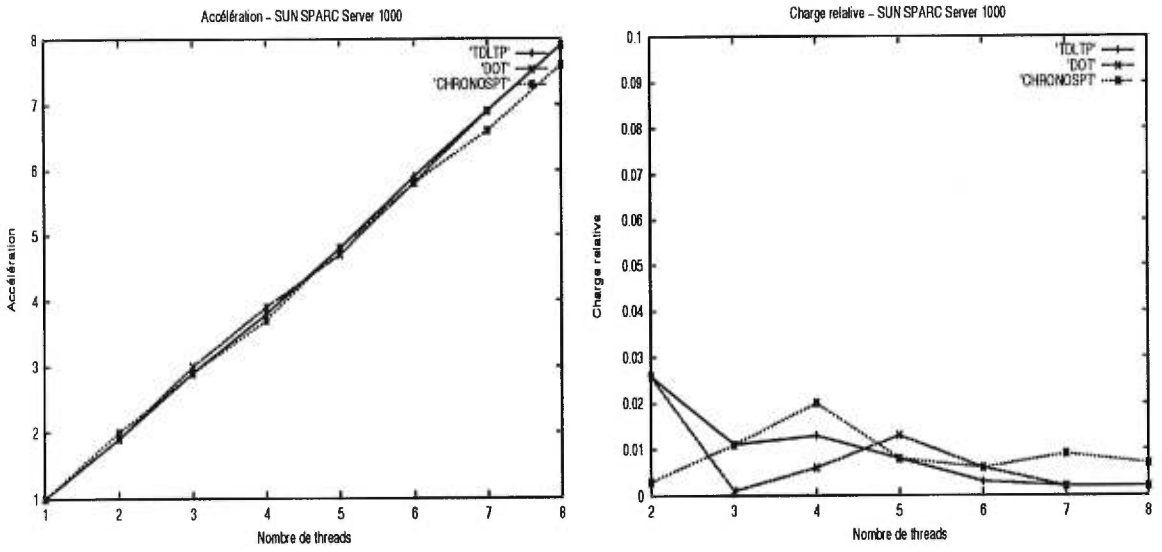

Figure 3.19: Accélération et charge relative - Ottawa, 30 périodes de temps.

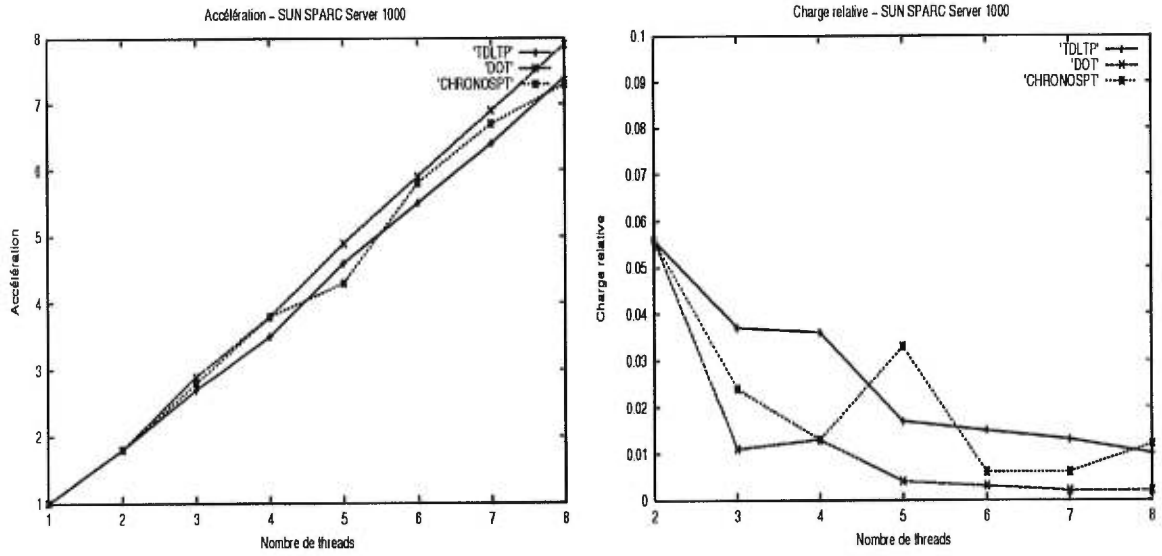

Figure 3.20: Accélération et charge relative - Ottawa, 60 périodes de temps.

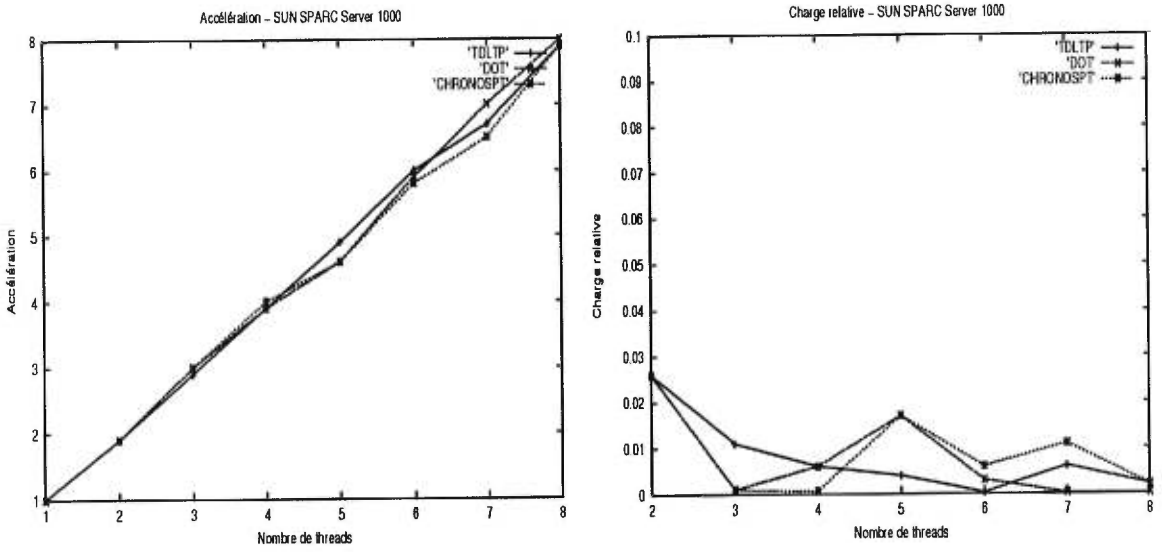

Figure 3.21: Accélération et charge relative - Ottawa, 90 périodes de temps.

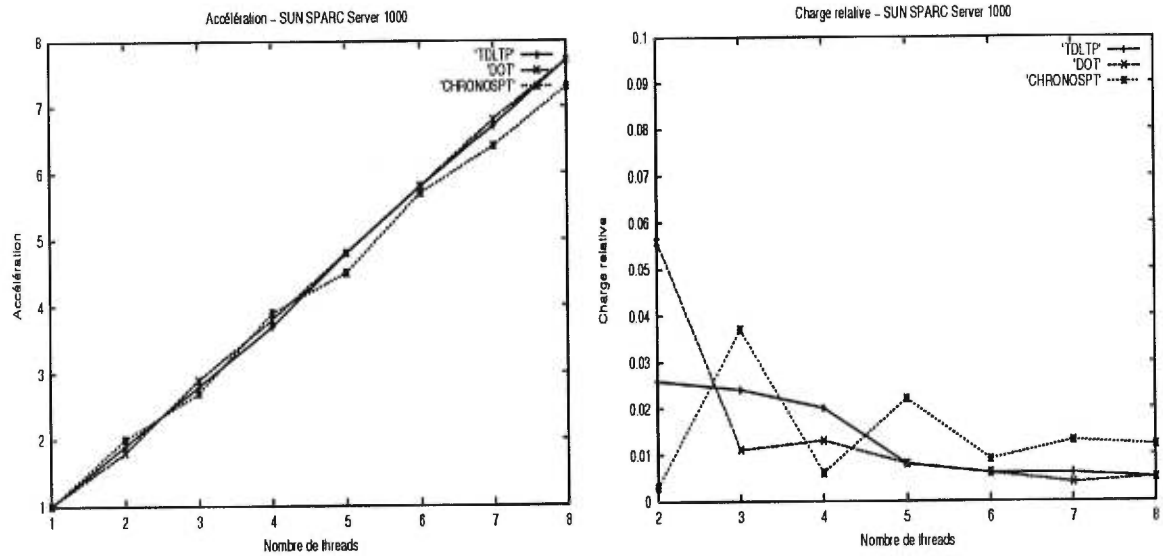

Figure 3.22: Accélération et charge relative - Ottawa, 120 périodes de temps.

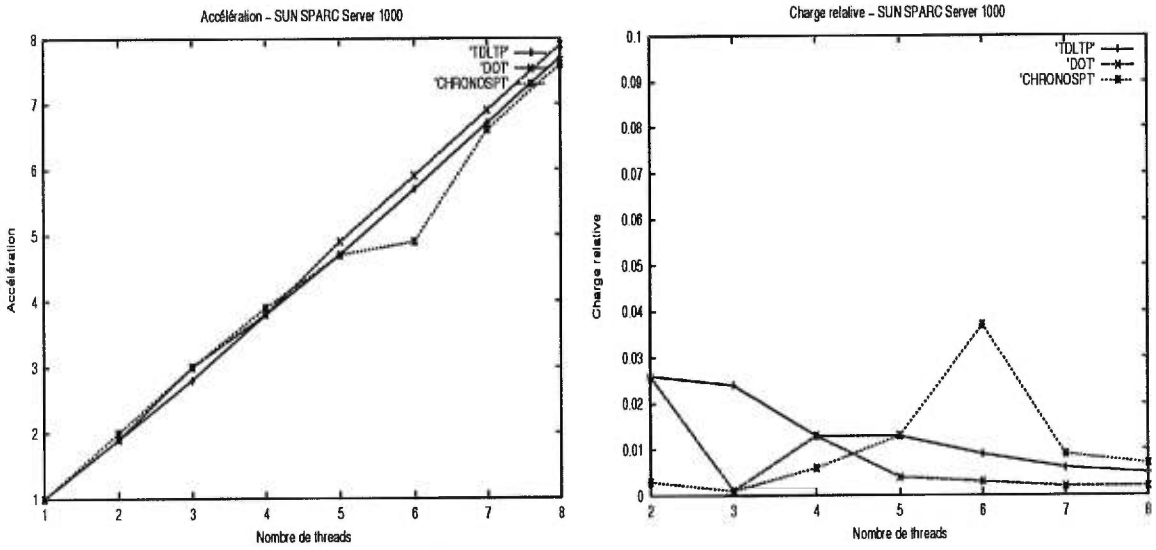

Figure 3.23: Accélération et charge relative - Montréal 30 périodes de temps.

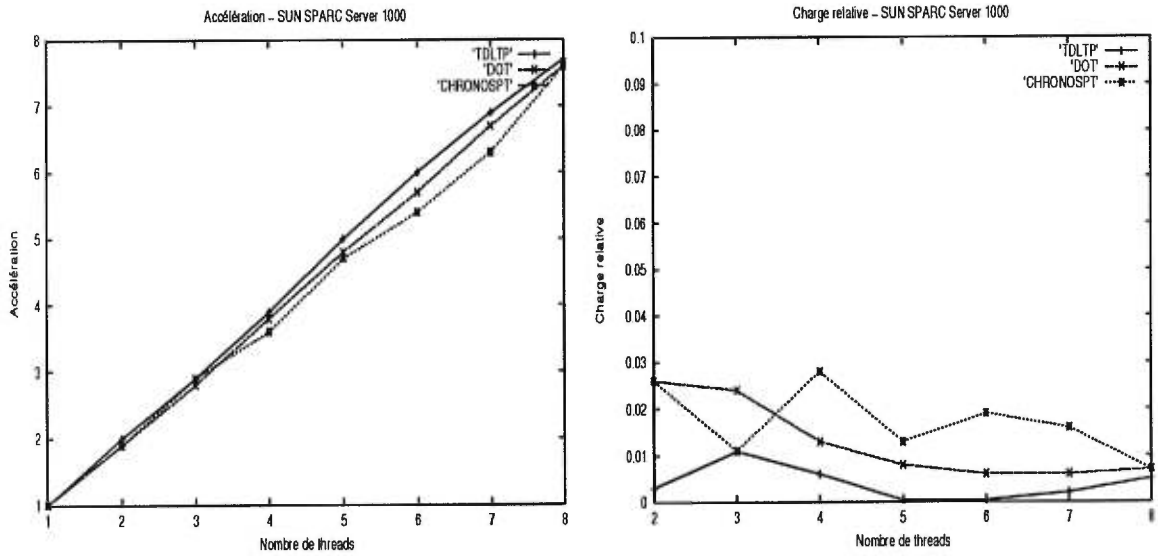

Figure 3.24: Accélération et charge relative - Montréal, 60 périodes de temps.

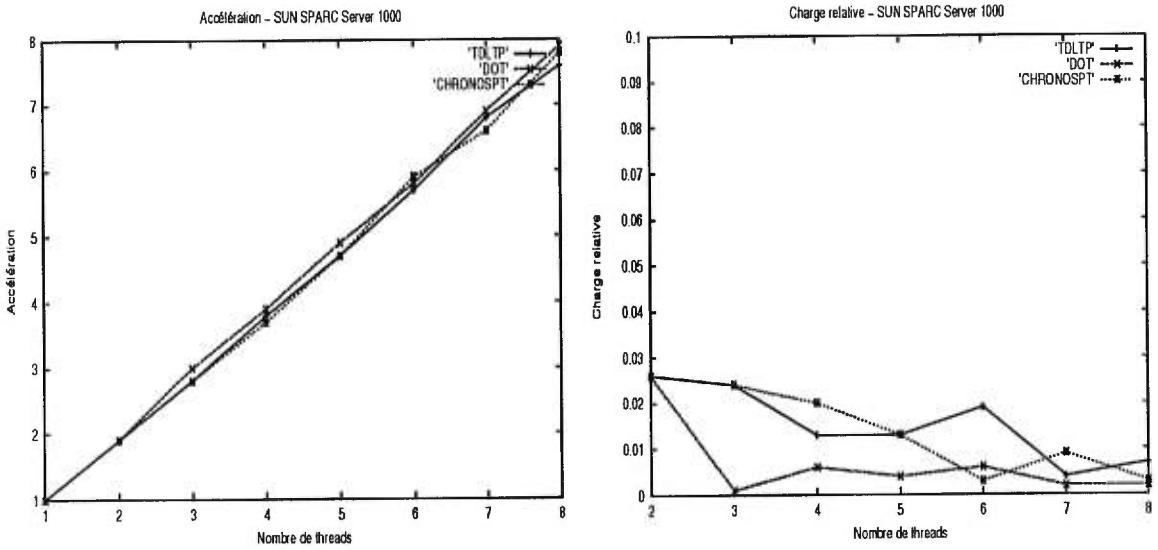

Figure 3.25: Accélération et charge relative - Montréal, 90 périodes de temps.

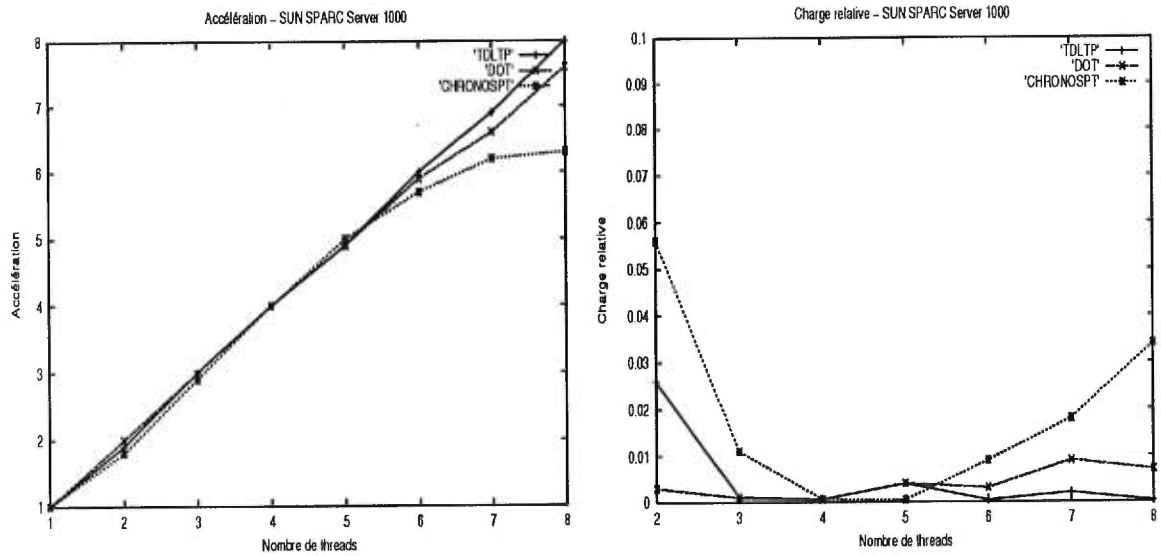

Figure 3.26: Accélération et charge relative - Montréal, 120 périodes de temps.

Conclusion

Dans ce mémoire, nous nous sommes intéressés au *calcul des plus courts chemins statiques et temporels* et, plus particulèrement, au développement d'implantations séquentielles et parallèles dans un contexte de **planification des transports**. L'objectif principal de cette étude était de fournir des outils de calcul des plus courts chemins statiques et temporels pouvant répondre aux besoins variés qu'exigent les différents modèles connus de planification des transports (modèle d'affectation dynamique, modèle de simulation, *etc.*).

Préalablement au développement des implantations séquentielles et parallèles, nous avons affectué une synthèse des différents résultats et algorithmes de calcul des plus courts chemins statiques et temporels. Cette synthèse nous a permis d'approfondir notre connaissance du problème et, par conséquent, de mieux diriger notre travail lors du développement des implantations. Cette revue de la littérature nous a également donné l'occasion de bâtir une bibliographie du sujet qui pourra être utile dans l'avenir.

Le développement des implantations séquentielles nous a permis de créer un ensemble d'outils de calcul des plus courts chemins statiques et temporels efficaces et facilement utilisables. Nos implantations ont été développées en langage *C++* et elles sont caractérisées par l'utilisation du format de fichier *NFF* choisi pour représenter nos réseaux. Dans le cas du calcul des plus courts chemins statiques, nous avons tenté de développer des outils pouvant répondre à différents besoins. Par exemple, nous avons développé des implantations effectuant la recherche des chemins *par parcours avant* (*forward star*), ainsi que *par parcours arrière* (*backward star*). De plus, des implantations traitant les délais et les interdictions sur les mouvements aux intersections ont été développées. Bien que plusieurs études ont déjà traité des performances reliées aux différentes implantations connues, nous avons effectué un certain nombre d'expérimentations, visant à mesurer les performances de nos implantations. Les résultats obtenus s'avèrent en fait similaires à ceux qui ont été observés auparavant. Ceci nous permet de conclure que l'utilisation du langage *C++* et du format de fichier *NFF* n'ont pas d'effets négatifs sur l'efficacité des calculs. En ce qui concerne le calcul des

plus courts chemins temporels, l'état actuel de la recherche nous a conduit à l'implantation de trois différentes approches de résolution. Ces approches de résolution sont celles proposées respectivement par Ziliaskopoulos et Mahmassani [57] (*TDLTP*), Chabini [12] (*DOT*) et Pallotino [46] (*CHRONOSPT*). Les résultats que nous avons obtenus, nous montrent que l'algorithme *TDLTP* s'avère très inefficace comparativement aux algorithmes *DOT* et *CHRONOSPT*. Nous constatons également que les algorithmes *DOT* et *CHRONOSPT* présentent des comportements relativement semblables. Une distinction peut toutefois être faite entre ces deux algorithmes puisque le calcul des plus courts chemins temporels effectué à l'aide de l'algorithme *CHRONOSPT* ne prend pas en considération les temps d'arrivée qui excèdent la dernière période de temps. En ce sens, nous pouvons conclure que l'algorithme *DOT* représente l'approche la plus efficace parmi les trois que nous avons étudiées.

Dans le but d'analyser les possibilités et les performances de deux types de traitements parallèles, nous avons développé deux implantations parallèles des algorithmes *TDLTP*, *DOT* et *CHRONOSPT*. Dans un premier temps, l'environnement **PVM** (*Parallel Vitual Machine*) a été utilisé dans le but d'effectuer un calcul distribué sur un réseau de stations de travail *SUN SPARC Ultra 1/140*. Les résultats obtenus nous ont montré que la granularité de ce type de problème est suffisante pour justifier l'utilisation de cette stratégie de parallélisation. Cependant, le problème de transmission des résultats demeure un point noir puisque l'efficacité de cette approche de parallélisation diminue drastiquement lorsque l'on cherche à communiquer les résultats à un moment ou à un autre de l'exécution. Dans un deuxième temps, nous avons procédé au développement d'une implantation parallèle utilisant la **programmation multithreads** dans le but d'exploiter les caractéristiques d'une machine multiprocesseurs à mémoire partagée. Nos expérimentations sur le *SUN SPARC Server 1000* comportant 8 processeurs, nous ont permis de constater que cette implantation parallèle s'avère aussi efficace que celle utilisant l'environnement *PVM*. Toutefois, l'utilisation de cette stratégie de parallélisation ne requiert pas de transmission des résultats. De plus, l'espace mémoire global nécessaire est de beaucoup inférieur à l'implantation utilisant l'environnement *PVM*.

Ainsi, à la lumière des résultats obtenus, nous sommes en mesure de conclure que, pour le problème considéré, l'utilisation de la *programmation multithreads* constitue la meilleure option de parallélisation. Cependant, si le problème traité ne nécessite que très peu d'échanges de messages (comparativement au temps de calcul total), alors l'utilisation de l'environnement *PVM* doit être prise en considération.

Suite à ce travail, nous avons déjà noté quelques avenues de recherche qu'il serait intéressant d'explorer. Tout d'abord, dans les deux types d'implantations parallèles

développées, nous utilisons une stratégie de parallélisation *maître-esclave*. Il serait intéressant, par exemple, de développer une implantation parallèle permettant entre autre les communications entre les *esclaves*. Ce type d'approche pourrait diminuer l'impact des échanges de messages lorsque ceux-ci sont coûteux.

Une autre approche de parallélisation pourrait également être étudiée dans le cas de l'algorithme *DOT*. En effet, la structure de cet algorithme se prête bien à une *parallélisation topologique*. Toutefois, cette approche implique l'apparition de conflits naturels dans le traitement des parties situées aux frontières de la division topologique. Compte tenu que cette stratégie implique un partage d'information relativement complexe, nous pouvons supposer qu'une implantation parallèle utilisant la *programmation multithreads* serait plus appropriée.

Notons enfin que l'ajout de nouveaux outils de calculs des plus courts chemins, ou le raffinement de ceux déjà en place, demeure une tâche continue que nous envisageons de poursuivre.

Bibliographie

[1] Ahuja, R.K., Magnanti, T.L., Orlin, J.B., «Network Flows: Theory, Algorithms and Applications», Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[2] Ahuja, R.K., Mehlorn, K., Orlin, J.B. et Tarjan, R.E., «Faster Algorithms for the Shortest Path Problem», Technical Report No. 193, Operations Research Center, M.I.T., 1988.

[3] Akl, S.G., «The Design and Analysis of Parallel Algorithms», Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[4] Barceló, J., Ferrer, J.L. et Montero, L., «Advanced Interactive Microscopic Simulator for Urban Networks. Vol. I: System Description, and Vol. II: User's Manual», Departamento de Estadísca e Investigación Operativa, Facultad de Informaticá, Universidad Politécnica de Cataluña, 1989.

[5] Bellman, R., «On a Routing Problem», *Quartely Applied Mathematics*, Vol. 16, pp. 87-90, 1958.

[6] Berg, D.J., Lewis, B., «A Guide to Multithreaded Programming», SunSoft Press, Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[7] Bersekas, D.P., Tsitsiklis, J.N., «Parallel and Distributed Computation, Numerical Methods», Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[8] Bertsekas, D.P., «A Simple and Fast Label-Algorithm for Shortest Paths», *Networks*, Vol. 23, pp. 703-709, 1993.

[9] Caldwell, T., «On Finding Minimal Routes in a Network with Turning Penalities», *Communications of the ACM*, Vol. 4, pp. 107-108, 1961.

[10] Chabini, I., «Des implantations parrallèles de l'algorithme d'approximation linéaire pour la résolution du problème d'affectation du trafic», Publication No. 382, Centre de recherche sur les transports, Université de Montréal, 1992.

[11] Chabini, I., «Nouvelles méthodes séquentielles et parallèles pour l'optimisation de réseaux à coûts linéaires et convexes», Publication No. 986, Centre de recherche sur les transports, Université de Montréal, 1994.

[12] Chabini, I., «A New Algorithm for Shortest Paths in Discrete Dynamic Network», 8th IFAC Symposium on Transportation Systems, Chinia, Greece, 1997.

[13] Chabini, I., Florian, M., «On Performance Measures of Parallel Algorithms», Publication No. CRT-95-14, Centre de recherche sur les transports, Université de Montréal, 1995.

[14] Chabini, I., Gendron, B., «Parallel Performance Measures Revisited», High Performance Computing Symposium'95, pp. 381-392, Canada's Ninth Annual High Performance Computing Symposium and Exhibition, Montréal, Canada, July 10-12, 1995.

[15] Cherkassky, B.V., Goldberg, A.V. et Radzik, T., «Shortest Paths Algorithms: Theory and Experimental Evaluation», *Mathematical Programming*, Vol. 73, pp. 129-174, 1996.

[16] Cooke, K.L. et Halsey, E., «The Shortest Route Through a Network with Time-Dependent Intermodal Transit Times», *Journal of Math. Anal. Appl.*, Vol. 14, pp. 492-498, 1966.

[17] Della Valle, G., Tartaro, D., «Ricerca dei percorsi di minimo costo mediante un algoritmo ibrido in presenza di penalità di svolta», *Ricerca Operativa*, Vol. 24, pp. 5-38, 1994.

[18] Denardo, E.V., Fox, B.L., «Shortest-Route Methods: 1.Reaching, Pruning, and Buckets», *Operations Research*, Vol. 27, pp. 161-186, 1979.

[19] Dial, R.B., «Algorithm 360: Shortest Path Forest with Topological Odering», *Communications of the ACM*, Vol. 12, pp. 632-633, 1969.

[20] Dijkstra, E.W., «A Note on Two Problems in Connexion With Graphs», *Numerische Mathematik*, Vol. 1, pp. 269-271, 1959.

[21] Dreyfus, S.E., «An Appraisal of Some Shortest-Path Algorithms», *Operations Research*, Vol. 17, pp. 395-412, 1969.

[22] Drissi-Kaïtouni, O., «A Shortest Path Algorithm for Networks with Time Dependent Link Travel Times», Publication No. 715, Centre de recherche sur les transports, Université de Montréal, 1990.

[23] Easa, S.M., «Shortest Route Algorithm with Movement Prohibitions», *Transportation Research*, Vol. 19B, pp. 197-208, 1985.

[24] Florian, M., Hearn, D., «Network Equilibrium Models and Algorithms», *Handbooks in Operations Research and Management Science: Network Routing*, Vol. 8, Chap. 6, 1986.

[25] Florian, M., Chabini, I., Le Saux, E. «Parallel and Distributed Computation of Shortest Routes and Network Equilibrium Models», 8th IFAC Symposium on Transportation Systems, Chania, Greece, 1997.

[26] Ford, L.R., «Network Flow Theory», Rand Corporation Report No., 1956.

[27] Gallo, G., et Pallottino, S., «Shortest Path Methods: a Unifying Approach», *Mathematical Programming Study*, Vol. 26, pp. 38-64, 1986.

[28] Gallo, G., et Pallottino, S., «Shortest Path Algorithms», *Annals of Operations Research*, Vol. 13, pp. 3-79, 1988.

[29] Geist, A. et al., «PVM: Parallel Virtual Machine, a Users' Guide and Tutorial for Networked Parallel computing», The MIT Press, Cambridge, Massachusetts, 1994.

[30] Glover, F., Klingman, D., Philips, N.V. et Shneider, R.F., «New Polynomial Shortest Path Algorithms and their Computational Attributes», *Management Science*, Vol. 31, pp. 1106-1128, 1985.

[31] Goldberg, A.V. et Silverstein, C., «Implementations of Dijkstra's Algorithms Based on Multi-Level Buckets», in *Network Optimization*, Hearn, D.W. et Pardolos, P. (Eds), Springer Verlag Lecture Note Series in Economics and Mathematical Systems, Vol. 450, pp. 292-327, 1997.

[32] INRO Consultants Inc., «Emme/2 User's Manual, Software Release 8.0», 789 pages, juin 1996, Montreal, Quebec, Canada.

[33] Johnson, D.B., «A Note on Dijkstra's Shortest Path Algorithm», *Journal of the ACM*, Vol. 20, pp. 385-388, 1973.

[34] Johnson, E.L., «On Shortest Path and Sorting», *In Proceedings of the 25th A.C.M. Annual Conference*, pp. 510-517, 1972.

[35] Kaufman, D.E. et Smith, R.L., «The Fastest Path in Time-Dependent Network for Intelligent Vehicle/Highway Systems Application», *IVHS Journal*, Vol. 1, pp. 1-11, 1993.

[36] Kershenbaum, A., «A Note on Finding Shortest Path Trees», *Networks*, Vol. 11, pp. 399-400, 1981.

[37] Kirby R.F. et Potts, R.B., «The Minimum Route Problem for Networks with Turn Penalities and Prohibitions», *Transportation Research*, Vol. 3, pp. 397-408, 1969.

[38] Mahmassani, H.S., Peeta, S., Hu, T.Y. et Ziliaskopoulos,A.K., «Dynamic Traffic Assignment and Simulation Procedures for ADIS/ATMS Applications», Technical Report DTFH61-90-R-00074-FINAL, Center for Transportation Research, The University of Texas at Austin, 1993.

[39] Mondou, J.F., «Mise au point d'un système interactif-graphique pour la planification tactique du transport des marchandises», Publication No. 136, Centre de recherche sur les transports, Université de Montréal, 1989.

[40] Mondou, J.F., Crainic, G.T. et Nguyen, S., «Shortest Path Algorithms: a Computational Study With the C Programming Language», *Computers and Operations Research*, Vol. 18, pp. 767-786, 1991.

[41] Moore, E.F., «The Shortest Path Through a Maze», *Proc. Int. Symp. on Theory of Switching*, Part. 2, Harvard University Press, pp. 285-292, 1959.

[42] Orda, A. et Rom, R., «Shortest-Path and Minimum-Delay Algorithms in Network with Time-Dependent Edge-Length», *Journal of the ACM*, Vol. 37, pp. 607-625, 1990.

[43] Orda, A. et Rom, R., «Minimum-Weight Paths in Time-Dependent Networks», *Networks*, Vol. 21, pp. 295-319, 1991.

[44] Pallottino, S., «Adaptation de l'algorithme de D'Esopo et Pape pour la détermination de tous les chemins les plus courts: améliorations et simplifications», Publication No. 136, Centre de recherche sur les transports, Université de Montréal, 1979.

[45] Pallottino, S., «Shortest Path Methods: Complexity, Interrelations and New Propositions», *Networks*, Vol 14, pp. 257-267, 1984.

[46] Pallottino, S., Scutellà, M.G., «Shortest Path Algorithms in Transportation Models: Classical and Innovative Aspects», Présentation au Colloque du C.R.T. sur les modèles d'équilibre, Octobre 1996, (à paraître).

[47] Pape, U., «Implementation and Efficiency of Moore-Algorithms for Shortest Route Problem», *Mathematical Programming*, Vol. 7, pp. 212-222, 1974.

[48] Patterson, D.A., Hennessy, J.L., «Computer Organization & Design: The Hardware/Software Interface», Morgan Kaufmann, San Francisco, California, 1994.

[49] Shier, D.R., et Witzgall, C., «Properties of Labeling Methods for Determining Shortest Path Trees», *J. Res. Nath. Bureau Stand.*, Vol. 86, pp. 317-330, 1981.

[50] Solaris, «Multithreaded Programming Guide», SunSoft Press, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[51] Spiess, H., «Contributions à la théorie et aux outils de planification des réseaux de transport urbain», Publication No. 382, Centre de recherche sur les transports, Université de Montréal, 1985.

[52] Standish, T.A., «Data Structures, Algorithms, and Software Principles», Addison-Wesley, 1994.

[53] Stroustrup, B., «The C++ Programming Language: Second Edition», Addison-Welsey, 1994.

[54] Yagar, S., «Dynamic Traffic Assignment by Individual Path Minimization and Queueing», *Transportation Research*, Vol. 5, pp. 179-196, 1971.

[55] Zenios, S.A., «Parallel Numerical Optimization: Current State and an Annoted Bibliography», ORSA Journal on Computing, Vol. 1, pp 20-43, 1989.

[56] Ziliaskopoulos, A.K. et Mahmassani, H.S., «Design and Implementation of a Shortest Path Algorithm with Time-Dependent Arc Costs», *Proc. of 5th Advanced Technology Conference*, Washington, D.C., pp. 1072-1093, 1992.

[57] Ziliaskopoulos, A.K. et Mahmassani, H.S., «A Time-Dependent Shortest-Path Algorithm for Real-Time Intelligent Vehicle/Highway Systems Applications», *Transportation Research Record 1408*, pp. 94-100, 1993.

[58] Ziliaskopoulos, A.K. et Mahmassani, H.S., «A Note on Least Time Path Computation Considering Delays and Prohibitions for Intersection Movements», *Transportation Research B*, Vol. 30, pp. 359-367, 1996.

[59] Ziliaskopoulos, A.K. et Mahmassani, H.S., Kotzinos, D. «Design and Implementation of Parallel Time-Dependent Least Time Algorithms for Intelligent Transportation Systems Applications», *Transportation Research C*, Vol. 5, pp. 95-107, 1997.