2m11.3145.9

Université de Montréal

Le calcul parallèle des plus courts chemins temporels

par
Jean-Nicolas Pépin

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

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.) en informatique

novembre, 2003

©Jean-Nicolas Pépin, 2003

QA
76
U54
2004
V.013

Université de Montréal

Direction des bibliothèques

AVIS

L'auteur a autorisé l'Université de Montréal à reproduire et diffuser, en totalité ou en partie, par quelque moyen que ce soit et sur quelque support que ce soit, et exclusivement à des fins non lucratives d'enseignement et de recherche, des copies de ce mémoire ou de cette thèse.

L'auteur et les coauteurs le cas échéant conservent la propriété du droit d'auteur et des droits moraux qui protègent ce document. Ni la thèse ou le mémoire, ni des extraits substantiels de ce document, ne doivent être imprimés ou autrement reproduits sans l'autorisation de l'auteur.

Afin de se conformer à la Loi canadienne sur la protection des renseignements personnels, quelques formulaires secondaires, coordonnées ou signatures intégrées au texte ont pu être enlevés de ce document. Bien que cela ait pu affecter la pagination, il n'y a aucun contenu manquant.

NOTICE

The author of this thesis or dissertation has granted a nonexclusive license allowing Université de Montréal to reproduce and publish the document, in part or in whole, and in any format, solely for noncommercial educational and research purposes.

The author and co-authors if applicable retain copyright ownership and moral rights in this document. Neither the whole thesis or dissertation, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms, contact information or signatures may have been removed from the document. While this may affect the document page count, it does not represent any loss of content from the document.

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Le calcul parallèle des plus courts chemins temporels

présenté par :

Jean-Nicolas Pépin

a été évalué par un jury composé des personnes suivantes :

Bernard Gendron
président-rapporteur

Michael Florian
directeur de recherche

Jean-Yves Potvin
membre du jury

RÉSUMÉ

Ce mémoire traite du calcul des *plus courts chemins* dans un graphe orienté. Nous nous intéressons plus particulièrement à l'implantation *parallèle* d'algorithmes de calcul des plus courts chemins *temporels* dans un contexte de **planification des transports**.

Bien que les objectifs principaux de ce mémoire se situent au niveau de l'implantation parallèle d'algorithmes de calcul des plus courts chemins temporels, nous pensons qu'il est aussi important de bien retracer l'évolution des recherches sur le calcul des plus courts chemins *statiques* et *temporels*. C'est pourquoi nous procédons à une revue de la littérature la plus complète possible, tant pour les plus courts chemins statiques que pour les plus courts chemins temporels.

Nous réalisons des implantations des diverses structures de données qui nous seront utiles par la suite ainsi que d'un outil de calcul des plus courts chemins statiques. Nous développons aussi des implantations séquentielles et parallèles de trois algorithmes proposés dans la littérature[1]. Notre façon de modéliser les réseaux et de calculer les plus courts chemins dans nos implantations sont influencées par le contexte pratique de la planification des transports. Les implantations séquentielles effectuées sont analysées pour bien en étudier les comportements. Les réseaux de transport des villes canadiennes de Winnipeg, Ottawa et Montréal sont utilisés afin d'évaluer les performances des différentes implantations.

Nous réalisons aussi des implantations parallèles des trois algorithmes de calcul des plus courts chemins temporels traités dans le cas séquentiel. Nous utilisons la **programmation *multithreads*** pour évaluer les performances du calcul sur une machine à 64 processeurs à mémoire partagée (SUNW Ultra-Enterprise 10 000). L'impact du calcul parallèle sur la calcul des plus courts chemins temporels est mesuré en présentant les résultats obtenus pour les réseaux de transport des trois villes canadiennes citées

---

[1] Il s'agit des algorithmes proposés par Pallottino et Scutellà [47], Ziliaskopoulos et Mahmassani [60] et Chabini [10].

précédemment. Nous comparons ensuite les résultats réels obtenus avec ceux prédits à l'aide de la mesure de *charge relative*. Nous pouvons remarquer des différences significatives dues à l'imprécision de la mesure de charge relative.

Mots clés : plus courts chemins, statique, temporel, calcul parallèle, programmation *multithreads*, charge relative, planification des transports.

SUMMARY

This thesis is about *shortest paths* in a directed graph. We are particularly interested in *parallel* implementations of *temporal* shortest paths algorithms for **transportation applications**.

Even though this thesis is mainly about parallel implementations of temporal shortest paths, we also provide a detailed literature survey on *static* and *temporal* shortest paths problems.

We present some useful data structures for efficient implementation of a computation tool for static shortest paths. These serve for the sequential and parallel implementations of three dynamic shortest path algorithms proposed in the literature[1]. The models and computational tools are influenced by the practical context of transportation applications. We use the transportation networks of three Canadian cities (Winnipeg, Ottawa and Montreal) to analyze the performances of the sequential implementations.

We also describe parallel implementations of the three aformentioned algorithms. We use multithread programming to evaluate their performances on a shared-memory 64 processors computer (SUNW Ultra-Enterprise 10 000). We use the transportation networks of the same three Canadian cities to show the impact of parallel computation on the performance of temporal shortest paths algorithms. We then compare our results with predictions obtained by using the *relative burden* performance measure. We observe significant differences between our results and the predictions, due to the imprecise nature of the *relative burden*. The notion of *relative burden* does not yield reliable results.

Key words: shortest paths, static, temporal, parallel computation, multithread programming, relative burden, transportation applications.

---

[1] Those algorithms were proposed by Pallottino and Scutellà [47], by Ziliaskopoulos and Mahmassani [60] and by Chabini [10].

TABLE DES MATIÈRES

Résumé iii

Summary v

Remerciements xi

Introduction 1

1. Revue de littérature sur les algorithmes de calcul des plus courts chemins statiques et temporels 3
   - 1.1 Algorithmes de calcul des plus courts chemins statiques 3
     - 1.1.1 Terminologie et notation 4
     - 1.1.2 Algorithme générique et classes d'algorithmes 4
     - 1.1.3 Méthodes d'ajustements progressifs (*label-correcting methods*) 7
     - 1.1.4 Méthodes d'extensions sélectives (*label-setting methods*) 13
   - 1.2 Algorithmes de calcul des plus courts chemins temporels 20
     - 1.2.1 Notation, formulations et propriétés 21
     - 1.2.2 Algorithmes de recherche *origine-vers-nœuds* pour un temps de départ fixe 22
     - 1.2.3 Algorithmes de recherche *nœuds-vers-destination* pour tous les temps de départ 28
2. Énoncé des algorithmes 36
   - 2.1 Rappel 36
   - 2.2 ChronoSPT (Pallottino et Scutellà) 37
   - 2.3 TDLTP (Mahmassani et Ziliaskopoulos) 38
   - 2.4 DOT (Chabini) 39
3. Implantations préalables de structures de données et autres outils essentiels 40
   - 3.1 Langage de programmation 40
   - 3.2 Structure *QUEUE* 41
   - 3.3 Structure *DEQUE* 44
   - 3.4 Structure *2QUEUE* 46
   - 3.5 Outils de calcul des plus courts chemins statiques *BSP2QUEUE* 48
4. Implantation séquentielle des algorithmes de calcul des plus courts chemins temporels 52

4.1 Modélisation des données et environnement de travail 52
4.1.1 Modélisation de données 52
4.1.2 Environnement de travail 55
4.2 Implantation séquentielle des algorithmes de calcul des plus courts chemins temporels 56
4.2.1 Implantation de l'algorithme ChronoSPT 59
4.2.2 Implantation de l'algorithme TDLTP 61
4.2.3 Implantation de l'algorithme DOT 62
4.3 Expérimentations et analyse des résultats 64
5. Implantation parallèle des algorithmes de calcul des plus courts chemins temporels 69
5.1 Traitement parallèle 70
5.1.1 Quelques concepts de bases reliés au traitement parallèle 70
5.1.2 La programmation multithreads 72
5.1.3 Mesures de performances des implantations parallèles 75
5.2 Implantation parallèle des algorithmes DOT, ChronoSPT et TDLTP 77
5.3 Expérimentations et analyse des résultats 82
Conclusion 88
Bibliographie 90
Annexe I xii

LISTE DES TABLEAUX

3.I Interface de la classe QUEUE 43
3.II Interface de la classe DEQUE 45
3.III Interface de la classe 2QUEUE 47
3.IV Interface partagée par les outils de calculs par parcours arrière 48
4.I Caractéristiques des réseaux de transport utilisés 55
4.II Spécifications techniques de la machine utilisée 55
4.III Interface des classes CHRONOSPT, TDLTP, DOT 57
4.IV Définitions des matrices de la méthode sp 57
4.V Temps d'exécution séquentielle 64
5.I Implantation parallèle : temps d'exécution pour l'algorithme Chrono-SPT 83
5.II Implantation parallèle : temps d'exécution pour l'algorithme DOT 83
5.III Implantation parallèle : temps d'exécution pour l'algorithme TDLTP 84
5.IV Estimation de l'accélération potentielle sur 60 processeurs 86

LISTE DES FIGURES

1.1 Cas pathologique pour l'implantation avec *deque* 10
3.1 Structure de données *queue* 41
3.2 Certains états possibles de la structure de données *queue* 42
3.3 Structure de données *deque* 44
3.4 Structure de données *2queue* 46
3.5 Structure de données *2queue* 46
4.1 Exemple de réseau 53
4.2 Exemple de fichier NFF (correspondant au réseau de la figure 4.1) 54
4.3 Hiérarchie des classes 56
4.4 Liste de compartiments 59
4.5 Influence de la période de temps sur le temps d'exécution 67
4.6 Influence de la taille du réseau sur le temps d'exécution 68
5.1 Relation entre un processus et les threads y étant associés 73
5.2 Schéma maître-esclave 78
5.3 Accélération – Winnipeg – 30 périodes de temps xii
5.4 Charge relative – Winnipeg – 30 périodes de temps xii
5.5 Accélération – Winnipeg – 60 périodes de temps xiii
5.6 Charge relative – Winnipeg – 60 périodes de temps xiii
5.7 Accélération – Winnipeg – 90 périodes de temps xiv
5.8 Charge relative – Winnipeg – 90 périodes de temps xiv
5.9 Accélération – Winnipeg – 120 périodes de temps xv
5.10 Charge relative – Winnipeg – 120 périodes de temps xv
5.11 Accélération – Ottawa – 30 périodes de temps xvi
5.12 Charge relative – Ottawa – 30 périodes de temps xvi
5.13 Accélération – Ottawa – 60 périodes de temps xvii
5.14 Charge relative – Ottawa – 60 périodes de temps xvii
5.15 Accélération – Ottawa – 90 périodes de temps xviii
5.16 Charge relative – Ottawa – 90 périodes de temps xviii
5.17 Accélération – Ottawa – 120 périodes de temps xix

5.18 Charge relative – Ottawa – 120 périodes de temps xix
5.19 Accélération – Montréal – 30 périodes de temps xx
5.20 Charge relative – Montréal – 30 périodes de temps xx
5.21 Accélération – Montréal – 60 périodes de temps xxi
5.22 Charge relative – Montréal – 60 périodes de temps xxi
5.23 Accélération – Montréal – 90 périodes de temps xxii
5.24 Charge relative – Montréal – 90 périodes de temps xxii
5.25 Accélération – Montréal – 120 périodes de temps xxiii
5.26 Charge relative – Montréal – 120 périodes de temps xxiii

REMERCIEMENTS

J'aimerais maintenant prendre le temps de remercier toutes les personnes qui ont permis la réalisation de ce mémoire.

Premièrement, un merci tout particulier à mon directeur de recherche, monsieur Michael Florian. Tout au long de la réalisation de ce mémoire, il s'est assuré que j'aie tout le support nécessaire, tant sur le plan technique que sur le plan financier. Il a su m'orienter dans la bonne direction.

Je veux également remercier Nicolas Tremblay qui, en me donnant accès à ses implantations, a permis de réaliser ce mémoire dans des délais raisonnables. Il a su m'expliquer les subtilités que je n'avais pas bien saisies. Son aide aura été plus que grandement appréciée.

Aussi, je me dois de dire un merci particulier à François Guertin, qui a su m'encourager dans mes premiers mois de recherche à temps plein. Son aide au niveau du calcul parallèle aura été très précieuse.

Un merci aussi au Centre de recherche sur les transports, pour la qualité de son personnel au niveau administratif et au niveau du support technique.

Un merci à Jean-François ainsi qu'à mon père pour leurs conseils judicieux au niveau de la qualité du français.

Un merci à mes parents pour leur soutien, leurs encouragements et leur réconfort tout au long de cette longue aventure…

INTRODUCTION

Le calcul des plus courts chemins est, depuis plusieurs années, un problème fondamental en optimisation de réseaux, ce qui a mené à de nombreux ouvrages sur le sujet. Notons, parmis ceux-ci, les ouvrages de Deo et Pang [21], Cherkassky, Goldberg et Radzik [16], Denardo et Fox [20], Gallo et Pallottino [28, 29] qui permettent de constater l'importance accordée dans la littérature au calcul des plus courts chemins. L'importance du calcul des plus courts chemins vient du fait que plusieurs modèles provenant de différents domaines d'application nécessitent un algorithme de calcul des plus courts chemins. Un de ces domaines, qui est celui sur lequel nous nous concentrons, est celui de la **planification des transports**.

Dans ce domaine, il existe plusieurs variantes du problème de calcul des plus courts chemins. Dans certains cas, on aura besoin de tenir compte de l'aspect temporel (par exemple, dans le cas d'un modèle de simulation) et dans d'autres, on sera plutôt intéressé à connaître les plus courts chemins à partir d'une origine vers plusieurs destinations (par exemple, dans le cas du problème d'affectation statique du traffic). Il se peut aussi que notre modèle doive permettre les mouvements aux intersections (délais et interdictions associés à chaque intersection). Comme on peut le constater, selon le type de problème traité, les besoins de calcul d'un plus court chemin peuvent être très différents, si bien que les algorithmes utilisés doivent être bien adaptés à chaque problème.

La première étape de ce mémoire consiste à développer des implantations séquentielles d'outils de calcul des plus courts chemins temporels. Ces outils pourront être réutilisés facilement par la suite. La seconde étape consiste à développer des implantations parallèles de ces mêmes algorithmes de calcul des plus courts chemins temporels. Nous nous attardons plus particulièrement aux algorithmes proposés par Pallottino et Scutellà [47], Ziliaskopoulos et Mahmassani [60] ainsi que celui proposé par Chabini [10]. Les expérimentations sont réalisées sur une machine à 64 processeurs à mémoire partagée (SUNW Ultra-Entreprise 10 000) en utilisant la **programmation *multithreads***.

L'organisation du mémoire est faite 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. Cette revue de la littérature se veut la plus complète possible, mais étant donné l'étendue des domaines d'application du calcul des plus courts chemins, il est certain qu'elle n'est pas exhaustive. Dans le **chapitre 2**, nous présentons de plus près les trois algorithmes de calcul des plus courts chemins temporels qui seront utilisés par la suite. Le **chapitre 3** présente des implantations de différents outils essentiels à l'implantation de nos algorithmes. Le **chapitre 4**, quant à lui, traite des implantations séquentielles des trois algorithmes. On verra que ces implantations sont nécessaires pour les implantations parallèles présentées au **chapitre 5**. Enfin, la conclusion résume les principaux résultats obtenus et fournit quelques avenues de recherche possibles pour le futur.

1. REVUE DE LITTÉRATURE SUR 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 n'inclut toutefois pas les algorithmes de calcul des plus courts chemins considérant les délais et les interdictions aux intersections. Nous nous limitons au problème classique, puisque la partie importante de ce mémoire se situe au niveau du calcul des plus courts chemins temporels. Cette synthèse approfondie sert de base théorique pour le développement effectué dans les chapitres suivants. De plus, elle se veut une mise à jour récente des recherches dans le domaine du calcul des plus courts chemins. La structure et une partie du contenu de notre revue de littérature est inspirée par celle réalisée par Nicolas Tremblay en 1998 [56].

Dans la section 1.1, nous traitons du calcul des plus courts chemins statiques. Nous tentons de résumer ceux qui nous apparaissent comme les plus importants. Nous laissons aussi de côté les derniers travaux les plus récents effectués dans ce domaine. Ce champ d'étude comportant beaucoup de travaux, nous ne prétendons pas faire ici une revue exhaustive de ce qui a été fait dans la littérature. Le calcul des plus courts chemins temporels est le sujet abordé dans la section 1.2. On retrouve, dans cette section, les algorithmes récemment proposés, ainsi que ceux qui ont servi de base à leur développement. Ce type de problème est particulièrement intéressant dans un contexte de planification en transport. De plus, on ajoute à la fin de cette section les récents développements d'algorithmes parallèles de calcul des plus courts chemins temporels.

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 et la notation qui sera utilisée. La deuxième partie présente un algorithme

générique de calcul des plus courts chemins statique et expose aussi les différentes 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

Soit $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ un graphe connexe orienté avec $\mathcal{N}$ et $\mathcal{A}$ représentant respectivement l'ensemble des nœuds 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) $d : \mathcal{A} \to \Re$. 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 $\mathcal{P} = i_1, i_2, \ldots, i_k$ 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 nœud 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 nœuds $i \neq r$. 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 du nœud i (*forward structure*) et l'ensemble des arcs entrant (*backward structure*) au nœud 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 nœud r (origine), l'arborescence $\mathcal{T}(r)$ des plus courts chemins vers tous les autres nœuds $i \in \mathcal{N}$ (destinations), $i \neq r$.

1.1.2 ALGORITHME GÉNÉRIQUE ET CLASSES D'ALGORITHMES

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 r à chaque nœud $k \in \mathcal{P}$. Les longueurs Π_k représentent bien les longueurs des plus courts chemins de

r à k. En effet, 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 à 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, il nous faut choisir le nœud 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 Π_k des plus courts chemins doivent nécessairement satisfaire le système d'équations suivant :

$$\left| \begin{array}{l} \Pi_r = 0 \\ \Pi_j = \min_{i:(i,j) \in \mathcal{A}\text{-}j} \{\Pi_i + d_{ij}\}, \ \forall j \neq r \end{array} \right.$$

Ce sont les *équations de Bellman* (Bellman [4]). Bellman a montré que, si le graphe $\mathcal{G}$ est tel qu'il n'existe pas de circuits de longueurs négatives et qu'il existe un chemin de r à j, pour chaque nœud j, alors les longueurs des plus courts chemins sont entièrement décrites par ces équations.

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

Conditions d'optimalité de Bellman :

Si nous avons un vecteur $\Pi = (\Pi_1, \Pi_2, \ldots, \Pi_n)$ *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 $\Pi = (\Pi_1, \Pi_2, \ldots, \Pi_n)$ représente les longueurs des plus courts chemins d'un nœud origine r vers tous les autres nœuds $j \in \mathcal{N}$.

Voici maintenant une description d'un algorithme générique de calcul des plus courts chemins statiques. Quel que soit celui que l'on considère, on peut dire que les

algorithmes de calcul des plus courts chemins cherchent tous, pour la plupart, à exécuter les opérations suivantes [28, 29] :

1. Initialiser une arborescence $\mathcal{T}(r)$ et, pour chaque sommet $i \in \mathcal{N}, i \neq r$, poser 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 l'étape 2 jusqu'à ce que les conditions d'optimalité soient satisfaites. C'est-à-dire :

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

Un des facteurs importants qui affecte l'efficacité d'un algorithme exécutant ces opérations est la façon dont les arcs violant la condition d'optimalité seront sélectionnés. Comme il arrive souvent que $n << m$, il peut être intéressant de sélectionner les nœuds plutôt que les arcs [28, 29]. On peut ainsi explorer l'ensemble des arcs sortant d'un nœud et, ensuite, continuer la suite des opérations.

Si on incorpore ces opérations dans un algorithme générique [28, 29] effectuant le calcul des plus courts chemins statiques d'une origine r vers tous les nœuds (*one-to-all*) et basé sur l'exploration des listes d'arcs sortants, on bâtit l'algorithme suivant. Associons, tout d'abord, à chaque nœud i un prédécesseur p_i (le plus récent nœud à partir duquel le nœud i a vu son étiquette être modifiée) et considérons un ensemble de candidats Q où les opérations de sélection et d'insertion d'éléments sont permises. L'algorithme générique est alors :

$\Pi_r = 0$; $p_r = -1$; $\Pi_i = +\infty$, $\forall i \in \mathcal{N} - r$; $Q = \{r\}$;
tant que $Q \neq \emptyset$ faire
  choisir $i \in Q$; $Q = Q - \{i\}$;
  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 $j \notin Q$ alors $Q = Q + \{j\}$;

La procédure générique que l'on vient de présenter ne spécifie aucunement la façon dont les nœuds sont sélectionnés. Chaque règle de sélection affecte la façon dont le graphe $\mathcal{G}$ est exploré et, par conséquent, implique une stratégie particulière de recherche. Ces différentes stratégies de recherche peuvent être classées en trois groupes qui représentent les trois approches les plus utilisées [28, 29]. Ces stratégies sont la recherche en largeur (*breath-first-search*), la recherche en profondeur (*depth-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 est beaucoup plus répandue que celle impliquant les stratégies de recherche et sera donc celle adoptée dans la suite du texte. Toutefois, Gallo et Pallottino [28, 29] préfèrent la première classification au lieu de la seconde puisqu'elle fait référence à la structure des algorithmes plutôt qu'à leurs comportements. En fait, certains algorithmes peuvent appartenir à l'une ou l'autre des classes selon le type de données considéré. Johnson [37] 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 Q sous forme de liste. Dans la plupart des ouvrages, on représente une liste sous forme de queue ou de pile [54]. 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 en début de liste et elle permet l'implantation de la recherche en profondeur.[2]

La littérature ne contient aucun exemple d'algorithme utilisant une pile comme structure de données. L'utilisation de la recherche en profondeur pour le calcul des plus courts chemins ne semble pas naturelle. En effet, une stratégie qui explore le graphe autour de l'origine, sans trop s'en éloigner, comme c'est le cas avec la recherche en largeur, est plus avantageuse. Gallo et Pallottino ont d'ailleurs vérifié numériquement que l'utilisation d'une pile donne une procédure inefficace comparativement à l'utilisation d'une queue [28]. Kershenbaum [40], quant à lui, 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[3]

L'utilisation d'une queue permet une implantation efficace du premier algorithme par *ajustements progressifs*, qui a été crédité à Bellman [4], Ford [27] et Moore [42]. Cet algorithme consiste à maintenir à chaque itération k, une étiquette $\Pi^{(k)}_i$ qui représente la plus courte distance de l'origine r au nœud 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 nœud i, sont élémentaires et contiennent au plus $n - 1$ arcs. On est alors assuré que $\Pi^{(k+1)}_i = \Pi_i$, $\forall\, i$. La procédure est la suivante. On pose que $\Pi^{(1)}_r = 0$, $\Pi^{(1)}_j = d_{rj}$, $\forall\,(r, j) \in \mathcal{A}$ et $\Pi^{(1)}_j = +\infty$, $\forall\,(r, j) \notin \mathcal{A}$. Ensuite, on a que :

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

Cette procédure itérative se base sur les équations d'optimalité développées par Bellman [4] et vues un peu plus tôt à la section 1.1.2.

La complexité de l'algorithme de *Bellman-Ford-Moore* est $O(nm)$. Chaque nœud peut être inséré dans la queue au plus $(n - 1)$ fois puisque son étiquette sera modifiée lors du

---

[2] Plus détails concernant les structures de données seront donnés un peu plus loin. Au besoin, consulter Shaffer [51] ou Standish [54].

[3] Plus de détails sur cette structure de données à la section 3.1

traitement des $(n - 1)$ autres nœuds, dans le pire des cas. Aussi, pour chaque nœud i sélectionné, on doit vérifier, pour $(i, j) \in \mathcal{A}$, si les conditions d'optimalité sont respectées. Donc, on doit faire $\Sigma_{i \in \mathcal{N}} |\mathcal{A}^+_i| = m$ opérations et on obtient la complexité mentionnée si haut, $\mathcal{O}((n-1)m) = \mathcal{O}(nm)$.

Implantation avec *deque*[4]

D'Esopo et Pape [49] proposent une nouvelle façon d'insérer les candidats dans la liste. Cette nouvelle stratégie peut être appliquée à l'aide d'une structure de données, la *deque*, qui combine les propriétés de la queue et de la pile. Une *deque* 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 nœud i se fait en fin de liste;
- -les insertions subséquentes d'un même nœud i se font en début de liste;
- -les suppressions se font en début de liste.

En utilisant cette politique d'insertion, D'Esopo et Pape combinent en fait les stratégies de recherche en largeur et en profondeur. En effet, lorsqu'un nœud i redevient candidat, procéder à l'insertion de ce nœud en début de liste signifie que l'on effectue une recherche en profondeur sur ce nœud. Concrètement, on essaye immédiatement de diminuer les étiquettes des nœuds successeurs d'un certain nœud i (recherche en profondeur). On réussit ainsi à diminuer considérablement le nombre de mises à jour d'étiquettes qui seraient normalement effectuées par un algorithme utilisant une recherche en largeur « pure ».

La complexité d'un tel algorithme est $\mathcal{O}(n2^n)$. C'est Kershenbaum [40] qui a permis d'établir que cette implantation a un temps d'exécution exponentiel en pire cas, en utilisant le réseau de la figure 1.1. Supposons que la liste d'adjacence du nœud origine 1 soit ordonnée selon l'ordre croissant des numéros de nœuds et que, pour les autres nœuds, cette liste d'adjacence soit ordonnée selon l'ordre décroissant des numéros de nœuds. Alors, à chaque fois qu'un nœud est visité, le nœud 2 obtient une nouvelle étiquette et il

---

[4] Plus de détails sur cette structure de données à la section 3.2

est ensuite immédiatement revisité puisque, dans ce cas. il est inséré au début de la liste. On remarque que l'étiquette du nœud 2 prendra respectivement toutes les valeurs entre 20 et 5 et que, pour chaque nouvelle étiquette, ce nœud sera revisité. Donc, le nœud numéro 2 sera traité 16 fois. Ceci correspond à $2^{n-2} = 2^4 = 16$, puisque dans cet exemple n = 6.

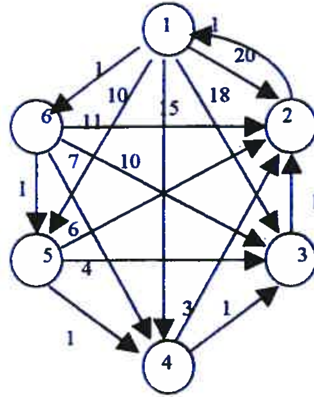


Figure 1.1 : Cas pathologique pour l'implantation avec *deque*.

Kershenbaum donne en plus une procédure qui permet de construire des réseaux de n nœuds possédant les caractéristiques de l'exemple précédent. Le résultat général auquel il arrive est que l'étiquette du nœud 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 chacun des n nœuds peut, dans le pire des cas, être visité 2^{n-2} fois, on obtient un temps d'exécution dans l'ordre de $O(n2^n)$.

Cependant, malgré son temps d'exécution exponentiel en pire cas, cette implantation s'avère très efficace en pratique. Son efficacité a été vérifiée, entre autres, par Gallo et Pallottino [29] et par Mondou, Crainic et Nguyen [41]. Les caractéristiques du réseau pathologique décrit par Kershenbaum (ordonnancement des listes d'adjacence des nœuds et coûts sur les arcs) sont rarement présentes simultanément. Il s'agit plutôt d'une construction purement théorique qui ne se retrouve que très rarement dans la pratique. En conséquence, la plupart du temps, cette implantation n'atteindra pas sa borne exponentielle.

Implantation avec *2queue*[5]

Contrairement à la combinaison d'une queue et d'une pile dans le cas d'une *deque*, la structure de données *2queue* est plutôt une combinaison, comme son nom l'indique, de deux queues. Pallottino [45, 46] suggère l'utilisation de cette structure de données pour implanter l'idée de D'Esopo et Pape. L'objectif est ici d'éliminer la possibilité d'une exécution exponentielle en pire cas. Comme ce mauvais comportement est dû, tel que 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 Q qui est modifiée. Pour cette implantation, les insertions subséquentes d'un nœud i se font à la fin de la première queue (qui est suivie d'une autre). De cette manière, lorsque toutes les étiquettes ont été modifiées au moins une fois, l'algorithme se comporte par la suite comme une implantation avec queue. Donc, puisque dans une implantation avec queue chaque nœud voit son étiquette modifiée au plus n fois, l'utilisation d'une *2queue* nous donnera $O(n^2)$ modifications d'étiquettes pour chaque nœud. Les opérations d'insertion et de sélection se font bien sûr en temps constant, tandis que l'initialisation nécessite $O(n)$ opérations. De plus, pour chaque nœud i sélectionné, on doit toujours vérifier, pour $(i, j) \in \mathcal{A}$, si les conditions d'optimalité sont respectées. Ce qui résulte en $\Sigma_{i \in \mathcal{N}} |\mathcal{A}^+_i| = m$ opérations. La complexité de cette implantation est donc $O(mn^2)$.

Les résultats obtenus par Gallo et Pallottino [29] et par Mondou, Crainic et Nguyen [41] montrent qu'une implantation utilisant la structure *2queue* est aussi performante que celle utilisant la structure *deque*. Ceci suggère fortement d'implanter l'idée de D'Esopo et Pape avec une *2queue*, pour ainsi 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. [31]. L'idée est d'effectuer une partition de l'ensemble des nœuds candidats Q en deux ensembles disjoints Q' et Q''. Ces

[5] Plus de détails sur cette structure de données à la section 3.3

deux ensembles sont représentés sous forme de queue. L'insertion des nœuds candidats dans un de ces deux ensembles se fait selon un certain critère. On définit d'abord une valeur s, que l'on appelle le seuil, avec laquelle on pourra comparer les étiquettes des nœuds à insérer dans Q. On insère dans la queue Q' les nœuds dont l'étiquette est inférieure ou égale à s et on insère les autres nœuds dans la queue Q''. La suppression des nœuds se fait au début de Q'. Lorsque la queue Q' est vide, on augmente le seuil et les éléments de Q'' qui ont une valeur plus petite ou égale à ce nouveau seuil sont transférés dans Q'. Évidemment, lorsque Q' et Q'' sont vides, l'algorithme se termine.

La complexité de cette implantation est $O(n^2m)$. L'opération de sélection d'un nœud nécessite $O(n)$ opérations en pire cas puisque la queue Q'' doit être balayée lorsque 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 nœud i dont l'étiquette est telle que $d_i \leq s$ n'entrera plus dans Q''. Donc, le nombre de fois que la queue Q'' est balayée lorsque 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. Lorsqu'il n'y a pas de mise à jour des listes, l'algorithme se comporte comme une implantation avec queue et ainsi, un nœud ne peut être retiré de Q' qu'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 $O(n^3)$. Le coût de l'opération d'insertion est clairement $O(n^2m)$ puisque pour chaque nœud i sélectionné, on doit parcourir la liste d'adjacence $\mathcal{A}^+_i$. On peut donc conclure que la complexité de cette implantation est $O(n^2m)$.

Le principal avantage d'utiliser une implantation avec seuil est d'augmenter la probabilité de choisir l'étiquette de coût minimum, et donc, possiblement diminuer les modifications supplémentaires d'étiquettes, tout en profitant de la simplicité des opérations fournies par une queue.

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

Bertsekas [7] propose une nouvelle façon d'ordonner les nœuds candidats dans la queue pour tenter de traiter les nœuds ayant les plus petites étiquettes le plus tôt possible. L'objectif est en fait de réussir à simuler l'*algorithme de Dijkstra*, que nous verrons à la

prochaine section, mais à un plus petit coût. L'heuristique qu'il utilise consiste à comparer l'étiquette Π_j d'un nœud *j* qui doit entrer dans la queue $\mathcal{Q}$ avec l'étiquette Π_i du nœud *i* à la tête de $\mathcal{Q}$. Si $\Pi_j \leq \Pi_i$, alors le nœud *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 ont été proposées dans la littérature. On peut, par exemple, combiner cette stratégie 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 réalisés par Gallo et Pallottino [29]. Les résultats qu'il a obtenus 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 plus performant que LTHRESH.

La complexité d'un algorithme utilisant la stratégie de la *plus petite étiquette d'abord* n'a pas été démontrée à notre connaissance. Bien qu'il soit 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 $\mathcal{O}(nm)$.

1.1.4 MÉTHODES D'EXTENSIONS SÉLECTIVES (*label-setting methods*)

Les algorithmes qui utilisent cette approche sont des implantations particulières de la méthode originale proposée par Dijkstra [23]. La propriété de base de ces algorithmes est que si $d_{ij} \geq 0,\ \forall (i, j) \in \mathcal{A}$, **alors chaque nœud est retiré de** $\mathcal{Q}$ **exactement une fois**. Les

implantations basées sur cette stratégie font une partition des nœuds en deux ensembles : l'ensemble des nœuds ayant une étiquette permanente (c'est-à-dire ceux qui qui ne sont plus dans $\mathcal{Q}$) et l'ensemble des nœuds 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 $\mathcal{Q} \neq \varnothing$ 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 $\mathcal{Q} = \mathcal{Q} + \{j\}$;

Dans la littérature, la plupart des implantations efficaces de l'*algorithme de Dijkstra* utilisent une queue de priorité pour représenter l'ensemble des candidats $\mathcal{Q}$. Une queue de priorité est un ensemble d'éléments auxquels on associe une valeur réelle ou étiquette. L'opération de suppression d'un élément permet de retirer l'élément d'étiquette minimale. Plusieurs structures de données peuvent être utilisées pour implanter la queue de priorité. À chaque type d'implantation 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. Évidemment, 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 nœud. L'opération de sélection d'un nœud i est exécutée n fois. À chaque fois, on doit choisir respectivement parmi n, $(n - 1)$, $(n - 2)$, ..., 2, 1 nœuds. Ce qui implique que cette opération est dans l'ordre de $\mathcal{O}(n^2)$. Ensuite, pour chaque nœud i, on doit explorer la liste des arcs sortant $\mathcal{A}^+{}_i$, ce qui se fait en au plus m opérations. L'insertion d'un nœud 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, qui se fait maintenant en temps constant, au détriment, par contre, 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 nœud *i* sont dans l'ordre de $O(n)$. Ainsi, le coût total de ces opérations est dans l'ordre de $O(n^2)$ puisque chaque nœud est visité au plus une fois. Cette implantation a également une complexité de $O(n^2)$. Les résultats de Gallo et Pallottino [29] démontrent clairement que ces implantations sont inefficaces.

Implantation avec liste de compartiments[6]

Une façon plus efficace de représenter la queue de priorité est de conserver les nœuds candidats dans une liste de compartiments (Dial [22], Denardo et Fox [20]). Dial a été le premier 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 vient du fait que l'*algorithme de Dijkstra* donne une étiquette permanente au nœud *i* ayant la plus petite étiquette Π_i. De plus, lors de la visite des arcs $(i, j) \in \mathcal{A}^+_i$, l'algorithme ne décroît pas les étiquettes Π_j lors de leur mise à jour, 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. Si on note $L = \max \{d_{ij}\}$, on a que $0 \leq \Pi_i \leq nL$ (un plus court chemin ne peut évidemment contenir plus de $(n - 1)$ arcs). Le compartiment *k* contient alors tous les nœuds dont l'étiquette temporaire est $\Pi_i = k$. L'opération de sélection consiste alors à parcourir la table jusqu'à ce que l'on identifie le premier compartiment non vide *k*. Les nœuds *i* de ce compartiment sont ensuite sélectionnés un par un et leurs étiquettes deviennent alors permanentes. Les nœuds *j* tels que $(i, j) \in \mathcal{A}$ et dont l'étiquette Π_j est modifiée lors du traitement d'un des nœuds *i* du compartiment *k* sont déplacés dans un

---

[6] Plus de détails sur ce type d'implantation à la section 4.2.1.

autre compartiment k' ($k' > k$) dans la table. Ceci implique que les compartiments 0, 1, 2, …, k seront toujours vides lors des prochaines itérations.

Cette structure de données permet de réaliser la suppression et l'ajout d'un élément en un temps constant. On peut aussi vérifier en temps constant si un compartiment contient des éléments. Par conséquent, la mise à jour d'une étiquette se fait en temps constant. Puisqu'il nous faut effectuer au plus m opérations de mise à jour des étiquettes et que ces opérations s'effectuent en temps constant, la mise à jour de toutes les étiquettes est $O(m)$. Toutefois, l'utilisation de cette structure amène un coût supplémentaire qui provient du parcours des $nL + 1$ compartiments de la table lors de la phase de sélection des nœuds. La complexité de cette méthode est donc $O(m + nL + 1) = O(m + nL)$.

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 nœud $j \in Q$, $\Pi_j \leq \Pi_i + L$ où i est un nœud avec une étiquette minimale dans 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]. Malgré ceci, ce type d'implantation obtient généralement de bons résultats en pratique. Gallo et Pallottino [29] et Mondou, Crainic et Nguyen [41] montrent que l'implantation de l'*algorithme de Dijkstra* avec une 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 [20] 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 nœuds pour lesquels l'étiquette Π_i se situe dans l'intervalle $[\lfloor k \rfloor, \lfloor k \rfloor+1)$. Ils recommandent, dans le but encore de réduire l'espace mémoire utilisé par cette structure de données, d'utiliser plusieurs niveaux de compartiments, idée clairement décrite par Goldberg et Silverstein [32]. Prenons l'exemple d'une liste de compartiments à deux niveaux. Dans ce cas nous avons 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 compartiment du niveau inférieur conserve les étiquettes correspondant à une seule distance (comme dans l'implantation 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 nœud est modifiée, le nœud est déplacé (si nécessaire) dans le 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. À ce moment-là, lorsque l'étiquette d'un nœud est modifiée, le nœud est alors déplacé (si nécessaire) vers un nouveau compartiment supérieur. Si 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 de placer les nœuds de ce compartiment dans les compartiments inférieurs correspondants. L'espace qui était occupé par les compartiments associés au compartiment supérieur précédent peut alors être réutilisé par les nouveaux compartiments inférieurs du niveau supérieur courant.

On remarque facilement que cette façon de procéder implique l'utilisation de seulement $2\sqrt{L+1}$ compartiments. La complexité de cette implantation avec deux niveaux de compartiments est de l'ordre de $O(m + n(1 + \sqrt{L}))$ et Goldberg et Silverstein montrent qu'en généralisant cette structure afin d'utiliser k niveaux de compartiments, on obtient une complexité dans l'ordre de $O(m + n(k + L^{1/k}))$.

Implantation avec arbre partiellement ordonné

Un arbre complet partiellement ordonné [54] est une structure de données dans laquelle chaque élément de la queue de priorité correspond à un nœud de l'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 $O(\log_b n)$ opérations et requiert un temps dans l'ordre de $O(b \log_b n)$ pour effectuer la sélection de l'élément minimal. Par conséquent, la complexité d'un algorithme utilisant ce type de structure est $O(m \log_b n + nb \log_b n)$. La valeur optimale de b est donnée par $b = \max \{2, \lceil m/n \rceil\}$ [1]. Ceci nous donne alors un temps d'exécution $O(m \log_b n)$. Johnson [38] a toutefois démontré, à l'aide de plusieurs expériences, que le choix de $b = 2$ donne l'implantation la plus efficace en pratique. Une des explications possibles à ce résultat est que les ordinateurs conventionnels fonctionnent en mode binaire.

Dans ce cas précis où $b = 2$, on appelle le b-arbre partiellement ordonné un monceau. Dans un monceau, chaque élément de la queue de priorité est associé à un nœud d'un arbre binaire. Cette structure de données permet toutes les opérations nécessaires, telles la modification d'une étiquette ou l'insertion ou la suppression d'un élément, en un temps $O(\log n)$ où n est le nombre d'éléments du monceau [54]. On doit retirer l'élément à la racine de l'arbre au plus $(n - 2)$ fois, ce qui résulte en $O(n \log n)$ opérations. Le nombre de modifications d'étiquettes est au plus m puisque pour chaque nœud i sélectionné, il est nécessaire de parcourir la liste d'adjacence de ce nœud. Puisque la modification d'une étiquette se fait en un temps $O(\log n)$, alors l'algorithme effectuera $O(m \log n)$ modifications d'étiquettes. Donc, la complexité de cette implantation est dans $O((m + n) \log n) = O(m \log n)$. Si le réseau est dense, c'est-à-dire que $m \approx n^2$, la complexité de l'algorithme devient $O(n^2 \log n)$. Évidemment, si le réseau est tel que $m \approx n$, alors la complexité est plutôt $O(n \log n)$. On remarque donc que l'utilisation d'un monceau est plus efficace, comparativement à une implantation $O(n^2)$, lorsque le réseau n'est pas complet. On peut aussi voir que si le nombre d'arcs est tel que $m = O(n^2/\log n)$, alors ce type d'implantation est équivalent à 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 par Gallo et Pallottino [29] et Mondou, Crainic et Nguyen [41] ont démontré l'efficacité de ce type d'implantation sur tous les types de réseaux. Il faut toutefois noter que sur les réseaux incomplets, certaines implantations utilisant une technique d'ajustements progressifs sont parfois plus efficaces.

Autres implantations

Il existe plusieurs autres types d'implantations qui ont été développées au cours des dernières années. Le but recherché est toujours le même : trouver une implantation avec une complexité de calcul inférieure aux implantations déjà connues. 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. Il arrive même souvent qu'aucun résultat numérique associé à ces implantations ne soit présenté.

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 $O(1)$, sauf pour la sélection de l'élément minimal qui requiert plutôt un temps $O(\log n)$. Cette implantation a donc une complexité de $O(m + n \log n)$, 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 [22] et une implantation de l'*algorithme de Dijkstra* avec queue où les nœuds d'étiquettes temporaires sont conservés dans une seule et même queue. La stratégie proposée par Dial utilise d'une certaine façon $L + 1$ queues. Chacune de ces queues contient les nœuds dont l'étiquette temporaire est égale à une certaine valeur k. L'idée de cet algorithme hybride consiste plutôt à conserver dans la k^e queue les nœuds dont l'étiquette est comprise dans un certain intervalle $[pk, pk + p + 1]$ où $p \in$ N. On réussit ainsi à réduire le nombre de compartiments à $1+\lceil \log L \rceil$. La complexité de ce type d'implantation est donc $O(m+n\log(nL))$. Les résultats obtenus par Mondou, Crainic et Nguyen montrent que si les réseaux utilisés sont complets ($m \approx n^2$),

cette implantation a une efficacité comparable aux méthodes les plus rapides lorsque la taille des problèmes augmente. Cependant, malgré l'attrait de sa meilleure borne théorique, on constate que 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 combinée à l'utilisation d'arbres de Fibonacci a un temps d'exécution $O(m + n\sqrt{\log L})$.

1.2 ALGORITHMES DE CALCUL DES PLUS COURTS CHEMINS TEMPORELS

Dans la section 1.1, on a pu constater l'importance accordée au calcul des plus courts chemins statiques dans la littérature. Dans les dernières années, les réseaux routiers sont devenus de plus en plus complexes et congestionnés. Ceci a mené au développement d'un nouveau champ d'expertise, la *planification en transport*, dont un des buts est de profiter de technologies avancées pour permettre une circulation plus sécuritaire et efficace sur les routes. Dans plusieurs modèles de planification en transport, le facteur temps doit absolument être considéré. L'introduction de cette nouvelle dimension a suffi à renouveler l'intérêt accordé au calcul des plus courts chemins.

Dans la présente section, nous présentons donc une synthèse des travaux traitant du calcul des plus courts chemins temporels. Dans la première partie de cette section, nous modifions la notation de la section précédente pour y introduire 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-nœuds (*one-to-all*) pour un temps de départ fixe et des algorithmes de recherche nœuds-vers-destination (*all-to-one*) pour tous les temps de départ. À la fin de la 3e partie, on inclut aussi certains travaux récents effectués au niveau du calcul parallèle des plus courts chemins temporels. En effet, la demande grandissante pour des applications fournissant des réponses en temps réel a poussé les chercheurs à améliorer les performances des algorithmes à l'aide du calcul parallèle.

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 nœud i au nœud j. Étant donné un temps de départ t du nœud i, nous avons alors que $t + d_{ij}(t)$ est le temps d'arrivée au nœud 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 nœud i au nœud j. On a soit que $d_{ij}(t)$ et $c_{ij}(t)$ sont définis pour t appartenant à un intervalle de temps discret S = $\{t_0, t_1, \ldots, t_M\}$, soit ils appartiennent à un intervalle de temps continu. Le modèle discret, c'est-à-dire où t est défini pour un intervalle de temps discret S, est d'un intérêt particulier puisque dans le domaine du transport, le temps est généralement divisé en un intervalle de temps discret. Comme le contexte de ce mémoire est la planification en transport, nous mettrons donc l'accent sur l'aspect discret du problème. Nous porterons tout de même une certaine attention au cas continu.

On peut classer les algorithmes de calcul des plus courts chemins temporels en deux types, selon la façon utilisée pour traiter les coûts sur les liens. Dans le cas où le coût $c_{ij}(t)$ est ignoré, cela revient à déterminer, étant donné un nœud origine r et un temps de départ t, les chemins minimisant les temps d'arrivée vers tous les autres nœuds $i \neq r$ (*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 nœud origine r et un temps de départ t, les chemins de coûts minimums vers tous les autres nœuds $i \neq r$ (*time-dependent least-cost path*). Il y a aussi plusieurs autres critères qui permettent de classer les algorithmes de calcul des plus courts chemins temporels, comme la définition des temps de parcours (discret ou continu), la possibilité d'avoir des temps d'attente aux nœuds ou non ou le choix des temps de départ (pour un temps de départ fixe, pour tous les temps de départ, pour certains temps de départ). Chabini [14] propose d'ailleurs une classification semblable à celle que l'on vient d'énoncer. Il propose quatre critères : l'objectif (temps ou coût), l'attente (permise partout, permise partiellement, non permise), selon les choix d'origines et de destinations (origine-vers-nœuds, nœuds-vers-destination) ou selon les différentes propriétés du modèles et des données.

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

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

En général, on dira qu'un réseau est FIFO si tous les liens du réseau satisfont cette propriété. Si elle 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 nœud i avant d'emprunter le lien (i, j). Bien sûr, ceci implique que les temps d'attente sont permis. Une propriété similaire peut aussi être imposée aux coûts sur les liens. On dit qu'un lien (i, j) est **CC** (*Cost-Consistent*) si « quitter le nœud i en empruntant le lien (i, j) plus tôt ne coûte pas plus cher que quitter plus tard ». La définition formelle de cette propriété variant parfois d'un auteur à l'autre, nous avons décidé de présenter ici celle suggérée par Pallottino et Scutellà [47]. Définissons $w_i(t)$ comme le coût d'attendre au nœud 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 Pallottino 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-nœuds pour un temps de départ fixe t_0.

Dreyfus [24] propose, de manière heuristique, de généraliser l'*algorithme de Dijkstra* (voir section 1.1.4) afin d'être en mesure de calculer les plus courts chemins temporels avec temps d'attente interdits pour tous les nœuds. Il est le premier à proposer cette généralisation qui sera reprise et prouvée plus tard par plusieurs auteurs [10, 39, 47]. Dans son approche, on suppose que l'on quitte le nœud origine r au temps t_0 et on définit, pour chaque nœud i, une étiquette Π_i qui représente une borne supérieure sur le plus court temps d'arrivée au nœud i. On tente alors de rendre ces étiquettes permanentes afin d'obtenir le temps de parcours minimal du nœud r vers chacun des autres nœuds i sachant que le temps de départ du nœud r est t_0. À une itération quelconque, on cherche à satisfaire la relation suivante :

$$\Pi_j = \min_{i:(i,j) \in \mathcal{A}\text{-}j} \{\Pi_j, \Pi_i + d_{ij}(\Pi_i)\}.$$

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

$$\Pi_j = \begin{cases} \min_{i:(i,j) \in \mathcal{A}\text{-}j} \{ \Pi_i + d_{ij}(\Pi_i)\}, \ \forall j \neq r; \\ t_0, j = r. \end{cases}$$

L'apport important de cette généralisation est qu'elle montre 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 $O(n^2)$. La validité de la généralisation de l'*algorithme de Dijkstra* proposée par Dreyfus est formellement prouvée par Kaufman et Smith [39]. Kaufman et Smith démontrent en plus que cette dernière n'est valide que si tous les liens du réseau possèdent la propriété FIFO. Sans cette condition, ils notent que cette méthode ne réussit pas à identifier les plus courts chemins! Kaufman et Smith donnent, entre autre, un contre-exemple où la procédure de Dreyfus échoue.

Orda et Rom [43, 44] sont les premiers à proposer une approche qui n'est pas restreinte par la condition FIFO. Toutefois, il faut noter que leur approche est conçue pour le modèle continu. On peut les considérer comme les pionniers dans l'étude de ce modèle. Dans leurs travaux, ils étudient trois différentes politiques de temps d'attente aux nœuds du réseau :

-temps d'attente illimité pour tous les nœuds;

-temps d'attente interdit pour tous les nœuds;

-temps d'attente illimité au nœud d'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 nœuds visités. Étant donné un nœud origine r et un temps de départ t_0, on cherche à déterminer les plus courts chemins de r vers tous les autres nœuds. Supposons que l'on atteint le nœud i au temps t, que l'on attend pendant un certain temps Δt, et que l'on quitte ensuite le nœud i en direction du nœud j. Le temps d'arrivée au nœud j sera alors $t + [\Delta t + d_{ij}(t + \Delta t)]$. Soit $\overline{d_{ij}}(t, \Delta t) \equiv \Delta t + d_{ij}(t + \Delta t)$, 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 $\overline{d_{ij}}(t, \Delta t^*) \leq \overline{d_{ij}}(t, \Delta t)$, $\forall \Delta t \geq 0$. À l'aide de cette approche, il est donc possible de déterminer les temps d'attente optimaux pour chaque nœud i ou encore le temps d'attente optimal au nœud origine si les pauses sont interdites pour chacun des autres nœuds 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 nœud. Cette opération est, dans ce cas, fonction de la valeur $\overline{d_{ij}}(t)$. Nous avons que

$$\Pi_j = \min_{i:(i,j)\in \mathcal{A}\text{-}j} \{\Pi_j, \Pi_i + \overline{d_{ij}}(\Pi_i)\}$$

où Π_i représente bien sûr le temps de parcours minimal du nœud origine r au nœud 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, ceci ne permet pas de trouver efficacement les chemins lorsque les temps d'attente sont interdits dans tout le réseau.

Orda et Rom [44] sont aussi les premiers à introduire le problème de recherche des chemins de coûts minimums dans un réseau temporel (*time-dependent least-cost path*) et à proposer un algorithme pour le solutionner. Leur approche calcule les chemins de coûts minimums entre un nœud origine r et tous les autres nœuds avec temps de départ t_0 au nœud origine. Ils proposent un algorithme qui permet de calculer les chemins lorsque les temps d'attente sont permis aux nœuds. Ils montrent également que lorsque les temps d'attente sont interdits à chaque nœud du réseau, le problème de recherche des chemins sans circuits est *NP-difficile*. Par contre, ils ne fournissent aucune implantation de leur

algorithme et, par conséquent, il n'y a pas de résultats qui permettent d'en analyser les performances. Bien qu'ils aient été les premiers à s'attarder à ce problème. il a fallu attendre quelques années pour voir quelqu'un fournir des résultats expérimentaux pour le problème des chemins de coûts minimums dans un réseau temporel.

Drissi-Kaïtouni [25] montre quant à lui qu'il est possible d'utiliser l'*algorithme de Dijkstra* dans un réseau qui n'est pas nécessairement FIFO. Soit

$$\overline{d_{ij}}(\Pi_i) = \min\{d_{ij}(t) + t - \Pi_i : t \geq \Pi_i\} \quad \forall (i,j) \in \mathcal{A},\ \forall \Pi_i \in \mathcal{S}.$$

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

Au milieu des années 1990, les travaux traitant du calcul des plus courts chemins temporels ont donné lieu à de nouveaux résultats et à de nouveaux algorithmes. Chabini [10] 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 $\mathcal{S} = \{0, 1, 2, \ldots, \mathcal{M} - 1\}$. 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 nœud j, que les chemins qui visitent le nœud 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 minimums est équivalente à l'équation fonctionnelle (1.2) :*

$$\Pi_j = \begin{cases} \min_{i:(i,j)\in \mathcal{A}\text{-}j} \min_{t \geq \Pi_i} \{t + d_{ij}(t)\},\ \forall j \neq r; \\ 0,\ j = r. \end{cases} \quad (1.1)$$

$$\Pi_j = \begin{cases} \min_{i:(i,j)\in \mathcal{A}\text{-}j} \{ \Pi_i + d_{ij}(\Pi_i)\}, \ \forall j \neq r; \\ 0, j = r. \end{cases} \qquad (1.2)$$

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 qu'il est le premier à énoncer 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 nœuds 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.*

Si la condition FIFO n'est pas satisfaite, il 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 nœuds.

Indépendamment à Chabini, Pallottino et Scutellà [47] proposent eux aussi 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 de temps

discret $\mathcal{S} = \{t_0, t_1, t_2, \ldots, t_{\mathcal{M}}\}$, alors le réseau espace-temps $\mathcal{R} = (\mathcal{V}, \mathcal{E})$ est défini de la façon suivante :

$$\mathcal{V} = \{i_h : i \in \mathcal{N}, 1 \leq h \leq \mathcal{M}\};$$

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

$\mathcal{R}$ est un réseau avec $|\mathcal{V}| = n\mathcal{M}$ nœuds et $|\mathcal{E}| = (m + n)\mathcal{M}$ liens et il ne contient pas de circuits. En particulier, chaque visite chronologique des nœuds de $\mathcal{R}$, c'est-à-dire dans laquelle les nœuds sont visités selon un ordre non-décroissant de leurs indices de temps, procure une visite topologique de $\mathcal{R}$.

Pallottino et Scutellà notent, tout comme Chabini, 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'ils proposent, et qu'ils nomment **Chrono-SPT,** effectue une sélection des nœuds de $\mathcal{R}$ selon l'ordre chronologique (on considère en premier lieu les nœuds correspondant au temps t_0, ensuite ceux correspondant au temps t_1 et ainsi de suite). Cette opération est effectuée au moyen d'une liste de compartiments $L = \{L_0, L_1, \ldots, L_{\mathcal{M}}\}$ où L_h dénote le compartiment contenant les nœuds à visiter au temps t_h. L'algorithme proposé permet de calculer les chemins temporels de coûts minimums avec ou sans temps d'attente pour chaque nœud i du réseau. Soit $\Pi_i(t)$ l'étiquette représentant le coût du chemin de l'origine au nœud i_t. Alors, l'algorithme est :

1.Initialisation

$L_0 = \{r\}$; $L_h = \varnothing$, $0 < h < \mathcal{M}$;
$\Pi_i(t) = \infty$, $\forall\, t, i \neq r$;
$\Pi_q(t) = 0$, $\forall\, t$;

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 nœud *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}).

Cet algorithme se modifie facilement pour s'adapter au cas du calcul des chemins minimisant les temps de parcours. Il suffit en effet dans ce cas d'ignorer les coûts $c_{ij}(t)$ sur les liens et de ne considérer que les temps de parcours $d_{ij}(t)$. Leur approche parcourt 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 $O(m\mathcal{M})$ en pire cas.

Les différentes approches vues dans cette partie permettent de calculer les plus courts chemins temporels pour une origine r vers tous les autres nœuds et ce, pour un temps de départ fixe. Toutefois, les modèles de planification en transport les plus utilisés présentement nécessitent plutôt de connaître les plus courts chemins à partir de tous les nœuds vers une destination donnée et ce, pour tous les temps de départ possibles. Bien sûr, il est possible d'utiliser les algorithmes que nous venons de présenter, en appliquant l'algorithme de son choix pour chaque nœud et pour chaque temps de départ possible, mais à un coût qui n'est clairement pas minimal.

1.2.3 ALGORITHMES DE RECHERCHE NŒUDS-VERS-DESTINATION POUR TOUS LES TEMPS DE DÉPART

Cooke et Halsey [17] sont les premiers à développer un algorithme calculant les plus courts chemins temporels de chaque nœud vers une destination donnée. En se basant sur le principe d'optimalité développé par Bellman [4], ils proposent une fonction itérative qui donne les chemins minimisant les temps de parcours à partir de chaque nœud i vers la destination q et ce, pour chaque période de temps t. Ils font deux hypothèses : $\mathcal{S} = \{t_0, t_0 + 1, \ldots, t_0 + \mathcal{M}\}$ est un ensemble discret d'intervalles de temps et les temps de parcours $d_{ij}(t)$ sont entiers pour $t \in \mathcal{S}$. L'entier $\mathcal{M}$ est choisi pour que les temps de parcours soient bien 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 nœud i à la destination q où le temps

de départ du nœud 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 maintient, à chaque itération k, un ensemble $\Pi^{(k)}{}_i(t)$ 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 nœud i est t. Au début de l'algorithme, les temps de parcours $d_{ij}(t)$ sont modifiés de la façon suivante :

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

On pose alors, comme conditions de départ,

$$\Pi^{(0)}{}_q(t) = 0, \text{ et } \Pi^{(0)}{}_i(t) = \bar{d}_{iq}(t),\ \forall\, i \neq q,$$

où $\Pi^{(0)}{}_i(t)$ représente le temps de parcours minimal reliant le nœud i et la destination q et qui est constitué de seulement un lien. De manière générale, on a que $\Pi^{(k)}{}_q(t) = 0$, et pour $i \neq q$ et $k = 1, 2, \ldots, n-2$, on a que

$$\Pi^{(k)}{}_i(t) = \begin{cases} \textit{temps minimum pour un chemin dans } \Pi^{(k)}{}_i(t), \textit{ si } \Pi^{(k)}{}_i(t) \neq \varnothing; \\ \infty, \textit{ si } \Pi^{(k)}{}_i(t) = \varnothing. \end{cases}$$

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

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

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

Ziliaskopoulos et Mahmassani [60] ont également travaillé sur cette variante du problème. L'algorithme qu'ils proposent peut calculer les plus courts chemins sans nécessairement permettre les temps d'attente aux nœuds, ce que Orda et Rom [43, 44] n'avaient pas réussi à faire antérieurement. Ils considèrent $\mathcal{S} = \{t_0, t_0 + \delta, t_0 + 2\delta, \ldots, t_0 +$

$(\mathcal{M}-1)\delta\}$ 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 désirée. Les temps de parcours $d_{ij}(t)$ prennent des valeurs réelles non-négatives et ils sont définis sur $\mathcal{S}=\{t_0, t_0+\delta, t_0+2\delta, \ldots, t_0+(\mathcal{M}-1)\delta\}$. Ils supposent que $d_{ij}(\bar{t})=d_{ij}(t_0+k\delta)$ pour $\bar{t}$ dans l'intervalle $t_0+k\delta<\bar{t}<t_0+(k+1)\delta$. 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 (ou période d'intérêt). On définit $\Pi_i(t)$, le temps de parcours minimal du chemin reliant le nœud i à la destination q quand le temps de départ du nœud i est t. Enfin, définissons $\Pi_i=[\Pi_i(t_0), \Pi_i(t_0+\delta), \ldots, \Pi_i(t_0+(\mathcal{M}-1)\delta)]$ comme étant un vecteur associé au nœud 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 [17] :

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

Plutôt que de visiter tous les nœuds à chaque itération, une liste des nœuds ayant le potentiel d'améliorer au moins une étiquette d'un autre nœud est maintenue. Alors, à partir de la destination, l'algorithme tente de résoudre récursivement l'équation d'optimalité en visitant tous les nœuds de cette liste. Plus précisément, lorsqu'un nœud i de la liste est visité, les étiquettes $\Pi_j(t)$ $(t\in\mathcal{S})$ de tous les nœuds j tels que $(i, j)\in\mathcal{A}$ sont, s'il y a lieu, mises à jour. Si au moins un des nœuds voit une de ses étiquettes modifiée, alors il est inséré dans la liste. Cet algorithme procède donc par ajustements progressifs. Si $\mathcal{Q}$ est la liste des nœuds à visiter, alors, initialement, la liste $\mathcal{Q}$ contient seulement la destination q. La première itération consiste à mettre à jour les étiquettes des nœuds pouvant rejoindre directement la destination q et à 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 nœud $j\in\mathcal{Q}$, 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 nœud i est inséré dans Q. Cette dernière étape est répétée jusqu'à ce que Q soit vide. À la fin de l'algorithme. les vecteurs Π_i de chaque nœud i contiennent les temps de parcours minimums, pour chaque période de temps $t \in S$, des plus courts chemins les reliant à la destination. Les étapes de l'algorithme sont les suivantes :

1. Initialisation
$\Pi_q = (0, 0, \ldots, 0)$
$\Pi_i = (+\infty, +\infty, \ldots, +\infty)$
2. Étape principale
si $Q = \varnothing$, alors aller à 4.
sélectionner le premier nœud j de la queue Q
pour chaque nœud $i : (i, j) \in \mathcal{A}_j$
pour chaque période de temps $t \in S$
si $\Pi_i(t) > d_{ij}(t) + \Pi_j(t + d_{ij}(t))$, alors remplacer $\Pi_i(t)$ par la nouvelle valeur $d_{ij}(t) + \Pi_j(t + d_{ij}(t))$.
si au moins une des étiquettes $\Pi_i(t)$ a été modifiée, alors le nœud i est inséré dans la queue Q.
3. Répéter l'étape 2.
4. L'algorithme est terminé.

Si on utilise une queue pour implanter la liste Q, alors la complexité de l'algorithme est dans l'ordre de $O(n^3\mathcal{M}^2)$. Initialement, la queue Q contient un seul élément, la destination q. Pendant la première itération, au plus $(n - 1)$ nœuds sont insérés dans Q. En exécutant $(n - 1)$ répétitions de l'étape (2), un des nœuds verra une de ses étiquettes devenir permanente. Nous avons que chacun des $(n - 1)$ nœuds peut entrer au plus $\mathcal{M}$ fois dans la queue Q, puisque chaque nœud 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. Comme l'étape (2) nécessite clairement $O(\mathcal{M}(n - 1))$ opérations, la complexité totale de l'algorithme est $O(n^3\mathcal{M}^2)$ en pire cas.

De plus, Ziliaskopoulos et Mahmassani proposent un algorithme pour résoudre le problème de recherche des chemins de coûts minimums 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 nœud i au temps t et $\Pi_i = [\Pi_i(t_0), \Pi_i(t_0 + \delta), \ldots, \Pi_i(t_0 + (\mathcal{M} - 1)\delta)]$ comme étant un vecteur associé au nœud 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 :

$$\Pi_i(t) = \min_{j:(i,j) \in \mathcal{A}+i} \{c_{ij}(t) + \Pi_j(t + d_{ij}(t))\}, \ \forall\, t \in \mathcal{S},\ \forall\, i \in \mathcal{N} \backslash q;$$

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

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

1. Initialisation
$\Pi_q = (0, 0, \ldots, 0)$
$\Pi_i = (+\infty, +\infty, \ldots, +\infty)$
2. Étape principale
si $Q = \varnothing$, alors aller à 4.
sélectionner le premier nœud j de la queue Q
pour chaque nœud $i : (i, j) \in \mathcal{A}_j$
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 nœud i est inséré dans la queue Q.
3. Répéter l'étape 2.
4. L'algorithme est terminé.

On peut voir que cet algorithme est simplement 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 minimisant les temps de parcours (*time-dependent least-time path*).

Pallottino et Scutellà [47] proposent d'utiliser une modification de leur algorithme *origine-vers-nœuds* calculant les plus courts chemins temporels pour un temps de départ fixe au nœud origine. En effectuant une visite inversée de la structure de compartiments L, ils utilisent la propriété acyclique du réseau espace-temps pour effectuer une visite chronologique inverse du réseau espace-temps. Une fois leur 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 nœud vers la destination q. La complexité de calcul de cet algorithme est $\Theta(\mathcal{M} + |\mathcal{E}^*|)$ et $O(m\mathcal{M})$ en pire cas.

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 nœud 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 $L_k = L_k + \{j\}$

Une autre approche utilisant la propriété acyclique du réseau espace-temps est également développée par Chabini [10]. 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 aux temps plus grands que t. On peut vérifier cette affirmation en observant le forme fonctionnelle donnée par l'équation (1.3) vue un peu plus tôt.

Dans son algorithme, Chabini suppose que, pour les temps de départ plus grands ou égaux à la dernière période de temps ($\mathcal{M} - 1$), le calcul des plus courts chemins temporels est équivalent au cas statique. Cette hypothèse est tout à fait raisonnable puisque l'on peut supposer que les temps de parcours sont constants après un certain intervalle de temps (après la période de pointe, par exemple). 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 $\Pi_q(t) = d_{ij}(t) + \Pi_j(t + d_{ij}(t))$

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 un temps dans $O(m\mathcal{M})$. Donc, globalement, nous obtenons une complexité de calcul qui est $\Omega(SSP + n\mathcal{M} + m\mathcal{M})$ pour cet algorithme.

Nous prenons quelques instants pour faire ressortir une conclusion concernant la variante du problème *nœuds-vers-destination* où les temps d'attente sont interdits pour tous les nœuds 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. Les trois algorithmes développés respectivement par Ziliaskopoulos et Mahmassani, Pallottino et Scutellà, et Chabini permettent de calculer les plus courts chemins dans ces conditions. Toutefois il se peut que ces chemins contiennent des boucles. En effet, lorsque les temps d'attente sont interdits, il est souvent nécessaire d'utiliser soit un chemin alternatif, soit une boucle, afin d'atteindre le nœud destination dans un temps au mieux égal à celui qu'on obtiendrait si les temps d'attente étaient permis. Ziliaskopoulos et Mahmassani [60] 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.

Nous voulons maintenant faire remarquer que déjà certains travaux ont été réalisés concernant le calcul parallèle des plus courts chemins temporels. Chabini [14, 15 entre autres] a réalisé diverses implantations parallèles de son algorithme DOT, que nous verrons en détails à la section 2.4. Il propose de tester ses implantations tout d'abord sur une machine à mémoire partagée[7], puis en utilisant une machine à échange de messages[8] dans un environnement PVM.[9] De plus, il suggère deux décompositions différentes : tout d'abord en décomposant par destination, puis en le faisant de façon topologique. Cette dernière est tout à fait naturelle puisque son algorithme n'oblige pas les arcs à être traités dans un ordre précis. Aussi, il compare ses résultats à ceux obtenus par différentes implantations parallèles d'algorithmes de calcul des plus courts chemins temporels connus. Chabini [15] propose également une façon de décomposer par période de temps. Il explique qu'on peut aussi voir cette décomposition comme une décomposition par sous-réseaux du réseau espace-temps.

---

[7] Plus de détails sur une architecture à mémoire partagée à la section 5.1.1 et en consultant Berg et Lewis [5].
[8] Plus de détails à la section 5.1.1 et en consultant Berg et Lewis [5].
[9] Pour plus de détails concernant l'environnement PVM, consulter Geist et al. [63].

Ziliaskopoulos et Mahamassani [61], puis Ziliaskopoulos et Kotzinos [62] développent quant à eux, une implantation de l'algorithme de calcul des plus courts chemins temporels TDLTP (voir section 2.3). Ils utilisent une stratégie à échange de messages dans un environnement PVM. Ils obtiennent des résultats très satisfaisants qui motivent à poursuivre les recherches dans l'implantation parallèle de tels algorithmes.

Bien que cette revue de littérature ne soit pas exhaustive, elle se veut la plus complète possible. Elle fournit un outil de travail récent pour quiconque pense poursuivre des recherches dans le domaine du calcul des plus courts chemins, que ce soit au niveau statique ou temporel.

2. ÉNONCÉS DES ALGORITHMES IMPLANTÉS

Nous avons vu dans notre revue de littérature qu'il y a différents modèles de calcul des plus courts chemins qui ont été définis par le passé. Nous allons ici nous concentrer sur un problème spécifique, celui de trouver les plus courts chemins dans des réseaux dynamiques à temps discrets. De plus, nous nous consacrons aux algorithmes qui cherchent à minimiser les temps de parcours des chemins reliant chaque nœud i à une destination d. Nous avons choisi de nous concentrer sur cette variante du problème des plus courts chemins parce que dans le domaine de la planification des transports, les réseaux sont très souvent dynamiques à temps discrets. Aussi, en ayant les plus courts chemins joignant chaque nœud à une destination donnée, on a accès rapidement à l'information voulue. Dans cette section, nous présentons donc les trois algorithmes utilisés dans le cadre de cette recherche. L'intérêt d'utiliser trois algorithmes différents n'est pas ici de démontrer la supériorité de l'un par rapport à l'autre, mais tout simplement de voir comment ils réagissent au traitement parallèle ainsi qu'aux différents réseaux. Les trois algorithmes retenus sont ceux proposés par Mahmassani et Ziliaskopoulos [60], Chabini [10] et, Pallottino et Scutellà [47][10].

2.1 RAPPEL

Faisons tout d'abord un bref rappel de la notation qui sera utilisée pour présenter les prochains algorithmes. Soit $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ un réseau qui associe à chaque lien $(i, j) \in \mathcal{A}$, un temps de parcours $d_{ij}(t)$ qui représente le temps nécessaire pour passer du nœud i au noeud j au temps t. Dans le cas qui nous intéresse, t appartient à un ensemble discret d'intervalles de temps $\tau = \{t_0, t_1, \ldots, t_{\mathcal{M}-1}\}$, où t_0 est le plus petit temps de départ possible pour chaque noeud du réseau, et $\mathcal{M}$ est un entier choisi assez grand pour représenter la période de temps qui nous intéresse. On définit d'abord les vecteurs $\Pi_i = [\Pi_i(t_0), \Pi_i(t_1), \ldots, \Pi_i(t_{\mathcal{M}-1})]$ associés à chaque nœud i, où $\Pi_i(t)$, est l'étiquette associée au nœud i au temps t (ie i_t). Ces étiquettes représentent en fait le temps de parcours minimal pour atteindre la destination d à partir du nœud i. On définit également $S_i(t)$, le nœud suivant le nœud i au temps t, dans le chemin le plus court courant menant de i à la destination d. Le

[10] Voir section 1.2.3.

problème est donc de trouver, pour chaque période de temps. les plus courts chemins reliant chaque nœud i à la destination d.

2.2. CHRONOSPT (Pallottino et Scutellà)

Dans un article paru en 1998, Pallottino et Scutellà [47] proposent un algorithme qui exploite la propriété acyclique du réseau espace-temps, c'est-à-dire que seule la partie non-redondante de ce réseau doit être considérée, permettant ainsi de réduire considérablement la taille du réseau à générer implicitement. Leur suggestion est de sélectionner les nœuds dans un ordre chronologique (on considère tout d'abord les nœuds correspondant au temps t_1, puis ceux correspondant au temps t_2, etc). L'algorithme, qu'ils nomment **Chrono-SPT**, parcourt donc une liste de compartiments B, où chaque compartiment B_{ti} contient les nœuds à visiter au temps t_i, pour implanter de façon efficace cette visite chronologique des nœuds du réseau espace-temps. Voici maintenant la description de l'algorithme:

1.Initialisation

$\Pi_i(t) = +\infty, \forall(t \in \tau, i \neq d)$;
$\Pi_d(t) = 0, \forall t \in \tau$;
$B_{ti} = \{d\}, \forall t \in \tau$;
$t_i = \mathcal{M} - 1$;

2.Itération principale

tant que $B_{t1} \neq \varnothing$ faire
  choisir j de B_{t1};
  $B_{t1} = B_{t1} - \{j\}$;
  pour chaque i tel que $(i, j) \in \mathcal{A}_j$ 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
      $\Pi_i(t_1) = d_{ij}(t_1) + \Pi_j(t_2)$;
      $S_i(t) = j$;
      si $i \notin B_{t1}$, alors $B_{t1} = B_{t1} + \{i\}$;

3.Mise à jour

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

4.Fin de l'algorithme.

Lorsque l'algorithme termine, le vecteur Π_i, associé à chaque nœud *i* du graphe, contient les étiquettes $\Pi_i(t)$ pour chaque période de temps $t \in \tau$. À partir du tableau $S_i(t)$, qui contient le successeur de chaque nœud *i* au temps *t* dans le chemin le plus court menant de *i* à la destination *d*, on peut facilement construire les chemins les plus courts à partir de chaque nœud *i* du graphe vers la destination *d*. Cet algorithme ne considère pas les temps d'arrivée plus grands (ou égaux) à la dernière période de temps $\mathcal{M} - 1$, si bien que certaines étiquettes garderont leur valeur initiale infinie si on ne peut pas rejoindre la destination avant la dernière période de temps.

2.3. TDLTP (Mahmassani et Ziliaskopoulos)

L'algorithme qui est proposé par Ziliaskopoulos et Mahmassani [60] est en fait une extension d'une approche classique de calcul de plus courts chemins statiques, puiqu'ils utilisent une technique d'*ajustements progressifs*. On peut donc l'implanter en utilisant une structure de données simple. Supposons que Q est la liste de nœuds à visiter. Voici la description de l'algorithme qu'ils ont appelés **TDLTP** (*time-dependent least-time path*), nom qui décrit en fait le type de problème traité par cet algorithme :

1.Initialisation

$\Pi_d = 0, \ \forall t \in \tau;$

$\Pi_i = +\infty, \forall (t \in \tau, i \neq d);$

2.Itération principale

si $Q = \varnothing$, alors aller à 4

choisir le premier nœud j de la queue Q et faire $Q = Q - \{j\}$

pour chaque nœud i tel que $(i, j) \in \mathcal{A}_j$ faire

pour chaque période de temps $t \in S$ faire

si $\Pi_i(t) > d_{ij}(t) + \Pi_j(t + d_{if}(t))$, 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 $Q = Q + \{i\}$

3.Répéter l'étape 2.

4.Fin de l'algorithme.

Lorsque l'algorithme termine, le vecteur Π_i, associé à chaque nœud $i \in N$, contient les étiquettes $\Pi_i(t)$ pour chaque période de temps $t \in \tau$. Dans cet algorithme, on considère $\tau = \{t_0, t_0 + \delta, t_0 + 2\delta, \ldots, t_0 + (\mathcal{M} - 1)\delta\}$, 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 qui nous intéresse. Les temps de parcours $d_{ij}(t)$ sont des réels non-négatifs et ils sont définis sur τ. Ils supposent que $d_{ij}(t^*) = d_{ij}(t_0 + k\delta)$ pour t^* dans l'intervalle $t_0 + k\delta < t^* < t_0 + (k+1)\delta$. De plus, on suppose également que $d_{ij}(t)$, pour des valeurs de t supérieures à $t_0 + (\mathcal{M} - 1)\delta$, est constant et égal à $d_{ij}(t_0 + (\mathcal{M} - 1)\delta)$.

2.4. DOT (Chabini)

L'algorithme de Chabini [10], quant à lui, ne requiert l'utilisation d'aucune structure de données (on n'a pas besoin de la liste de nœuds à visiter). Par contre, l'algorithme suppose que les temps de parcours sont entiers et strictement positifs et définis dans $\tau = \{0, 1, \ldots, \mathcal{M} - 1\}$. Cette hypothèse est à la base du raisonnement derrière cet algorithme, qu'il nomme **DOT** (*Decreasing Order of Time*). Il utilise, tout comme Pallottino et Scutellà, la propriété acyclique du réseau espace-temps associé à l'expansion dans le temps du réseau temporel original. Supposons qu'on a déjà un outil de calcul des plus courts chemins statiques (SSP, *Static Shortest Path*). Voici donc l'algorithme proposé par Chabini :

1.Initialisation

$\Pi_i(t) = +\infty, \forall(t < M - 1, i \neq d);$

$\Pi_d(t) = 0, \ \forall t < M - 1;$

$\Pi_i(\mathcal{M} - 1) = \text{SSP}(d_{ij}(\mathcal{M} - 1), d);$

Note : $\Pi_i(t) = \Pi_i(\mathcal{M} - 1), \ \forall(t \geq M - 1, i \neq d);$

2.Itération principale

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

pour chaque arc $(i, j) \in A$, 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 $b_i(t) = j$

Encore une fois, le vecteur Π_i associé à chaque nœud i du graphe contient les étiquettes $\Pi_i(t)$ pour chaque période de temps $t \in \tau$ à la fin de l'algorithme. On peut constater que dans cet algorithme le calcul des plus courts chemins pour des temps de départ supérieurs ou égaux à $\mathcal{M} - 1$ est équivalent au calcul des plus courts chemins statiques.

On verra en détails, dans les sections suivantes, comment les implantations séquentielles et parallèles de ces trois algorithmes ont été réalisées.

3. IMPLANTATION PRÉALABLE DE STRUCTURES DE DONNÉES ET AUTRES OUTILS ESSENTIELS

Pour réaliser les implantations séquentielle et parallèle de nos trois algorithmes, nous avons bien sûr utilisé certaines structures de données et nous avons aussi développé certains outils fort utiles. Nous commencerons tout d'abord par tenter de justifier notre choix de langage de programmation. Ensuite, nous présenterons ces structures de données, en expliquant leur fonctionnement et en décrivant leur implantation. Nous discuterons donc des implantations des structures *queue*, *deque* et *2queue*. De plus, nous décrirons l'implantation d'un outil de calcul des plus courts chemins statiques nécessaire à l'algorithme DOT.

3.1 LANGAGE DE PROGRAMMATION

Avant de commencer à réaliser une implantation, il faut d'abord décider quel langage de programmation sera utilisé. Ce choix pourra s'avérer fort important, puisque la qualité de l'implantion (sa portabilité, ses performances, etc) dépend en bonne partie de ce choix. Le choix du langage de programmation est donc le point de départ de notre développement. On doit se questionner pour savoir ce qu'on attend exactement de notre application. Est-ce que la réutilisation des outils est un aspect important? Est-ce qu'on recherche, avant tout, l'implantation la plus rapide et efficace possible? Est-ce qu'on veut plutôt un code qui sera peut-être un peu lourd, mais plus facilement compréhensible pour un utlisateur futur? Avec les réponses à ces questions et à bien d'autres, on peut ainsi établir nos priorités quant aux qualités requises par notre langage de programmation.

Dans cette recherche, le langage choisi a été le langage orienté objet C++, car ce langage permet de développer des applications performantes. Aussi, la hiérarchie de classes, disponible dans les langages orientés objet, permet d'obtenir un code plus facilement lisible pour des utilisateurs futurs, plus facile à maintenir et à réutiliser. Le but n'est pas ici de vanter les mérites du langage C++, mais plutôt de simplement justifier son utilisation pour le développement de nos implantations. Comme nous voulions

développer des outils de base pouvant être réutilisés, le souci de clareté était très important, d'où l'utilisation d'un langage orienté objet. De plus, comme un des buts importants de cette recherche est de voir à quel point des implantations parallèles peuvent être efficaces dans notre cas précis, alors le souci de performance était aussi important. Le langage C++ est largement utilisé depuis déjà plusieurs années, alors son choix ne devrait pas surprendre. Dans les sections qui vont suivre, nous allons présenter les codes C++ des implantations.

Dans le but de permettre à toute personne ne connaissant pas de façon approfondie ce langage, nous définissons ici certains concepts essentiels à la compréhension des implantations présentées dans la suite de ce mémoire :

- * : utilisé pour définir un pointeur sur un type quelconque;
- . ou → : utilisé pour accéder à une méthode ou à un champ d'une classe;
- new : utilisé pour allouer l'espace mémoire nécessaire à la création d'un objet d'un certain type défini.

De plus, pour alléger le code pour une meilleure compréhension, certains détails techniques seront omis et d'autres remplacés par une version pseudo-code.

3.2 STRUCTURE *QUEUE*

Une queue est une structure de données qui permet les opérations d'insertion à la fin de la structure et de suppression au début de la structure, comme il est montré à la figure 3.1.

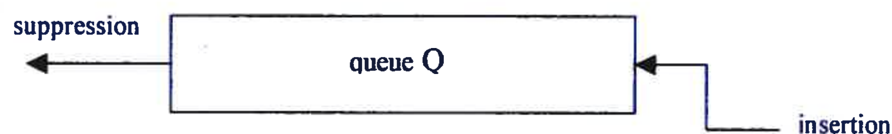


Figure 3.1 : Structure de données queue

Notre implantation de cette structure utilise quatre pointeurs pour bien gérer les différentes opérations. Les pointeurs **top** et **eoq** (*end-of-queue*) sont fixes puisqu'ils représentent le début et la fin de la queue. Ils servent à bien délimiter l'espace qu'occupe

la queue. Le pointeur *top* pointe en fait sur la première case mémoire libre après la structure de données. Les deux autres pointeurs utilisés sont **first** et **end**. Le pointeur *first* pointe sur le premier élément, tandis que le pointeur *end* pointe sur la première case mémoire venant après celle occupée par le dernier élément de la queue. Chaque nouvel élément est ajouté dans la case pointée par *end*. Un déplacement d'une case mémoire vers la droite (incrémentation) du pointeur *end* est effectuée à chaque fois qu'un élément est ajouté à la queue et, une fois le pointeur *eoq* atteint, une mise à jour vers le pointeur *top* est effectuée. Le pointeur *first* se comporte de façon similaire au pointeur *end*. Chaque opération de sélection d'un élément provoque une incrémentation du pointeur *first* tant que le pointeur *eoq* n'est pas atteint. Comme pour le pointeur *end*, une mise à jour vers le pointeur *top* est effectuée lorsque *first* = *eoq*, permettant ainsi de poursuivre le processus. Évidemment, la queue est **vide** lorsque *first* = *end*. La prochaine figure (3.2) montre trois états possibles de la structure de données queue.

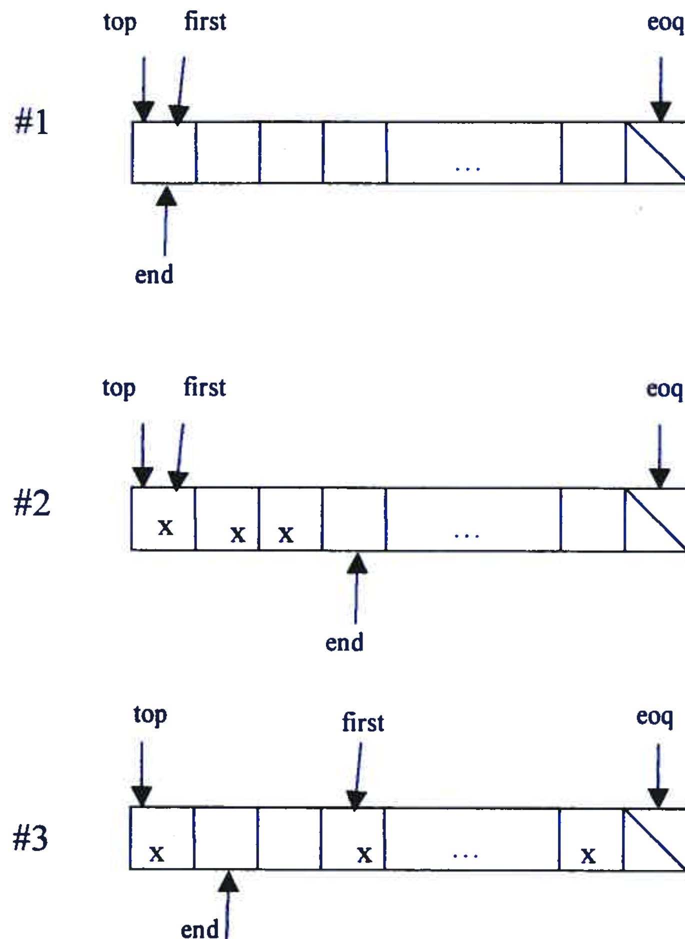


Figure 3.2 : Certains états possibles de la structure de données queue.

Dans le premier état, on représente l'état initial : la queue est vide et les pointeurs sont en place. Dans le second état, il y a trois éléments au début de la queue. Le pointeur *end* pointe donc sur la quatrième case mémoire. Dans le troisième état, on a atteint précédemment le pointeur *eoq*, une mise à jour a donc été faite pour que le pointeur *end* retourne au début de la liste. Puis, un autre élément a été inséré et le pointeur *end* pointe maintenant la deuxième case mémoire.

On a développé la classe paramétrée QUEUE<T>, dont l'interface est présentée au tableau 3.I, afin d'implanter cette structure de données. On utilise ici le concept de ***template*** propre au langage C++. Ceci nous permet de créer une **classe conteneur** dont le type des objets qui y sont contenus doit être préalablement défini par l'utilisateur. Dans notre cas, nous devrons donc définir le type d'éléments qui iront dans notre queue (*int, long, float, double, char,* etc). Voici maintenant le code C++ relié à l'implantation de cette classe.

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 3.I : 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*). Cette structure de données sera utilisée lors de l'implantation de l'algorithme de Pallottino et Scutellà, ChronoSPT.

3.3 STRUCTURE *DEQUE*

Une *deque* est une structure de données légèrement différente et aussi un peu plus complexe qu'une queue. Cette structure est en fait une combinaison d'une pile et d'une queue (voir la figure 3.3). Elle permet les insertions au début et à la fin de la liste et les suppressions au début de la liste.

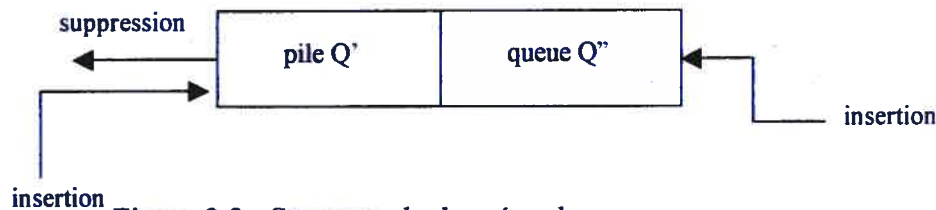


Figure 3.3 : Structure de données *deque*.

Bien que plus complexe, cette structure de données s'implante de façon semblable à la classe paramétrée QUEUE<T>. En effet, en pratique, l'implantation se fait sans distinction entre la pile et la queue. Ce n'est que le jeu des pointeurs, qui sont les quatre mêmes que pour la structure précédente, qui permettra, en plus des opérations de la structure queue, d'insérer des éléments au début de la *deque*. Pour cette opération, il suffit de déplacer le pointeur *first* d'une case mémoire vers la gauche (décrémentation) et, si le pointeur *top* est rencontré, on doit faire une mise à jour vers le pointeur *eoq*. Le nouvel élément sera inséré dans la case mémoire pointée par le pointeur *first*. La figure 3.2 qui montrait trois états possibles d'une queue pourrait tout aussi bien montrer trois états possibles d'une *deque*.

Pour l'implantation de la structure de données *deque*, nous avons encore utilisé le concept de *template* pour développer la classe paramétrée DEQUE<T> dont l'interface est présentée dans le tableau 3.II. On peut remarquer, qu'outre la différence de noms des méthodes d'insertion, cette interface est la même que dans le cas de la classe QUEUE<T>. Voici maintenant le code C++ relié à l'implantation de cette classe.

```
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; }
}
```

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 3.II : Interface de la classe *DEQUE*.

Comme pour la classe QUEUE<T>, la création d'un objet de la classe DEQUE<T> requiert 28 + *st* bytes (où *s* et *t* sont définis de la même façon que pour la classe QUEUE<T>). Cette structure de données sera utilisée lors de l'implantation de l'algorithme de Ziliaskopoulos et Mahmassani, TDLPT. Ils avaient d'ailleurs eux-mêmes utilisé cette structure de données [60].

3.4 STRUCTURE *2QUEUE*

Le structure de données *2queue* est elle aussi, tout comme la structure *deque* une combinaison de deux sous-structures. Comme le montre la figure 3.4, une *2queue* en en fait tout simplement la juxtaposition de deux queues.

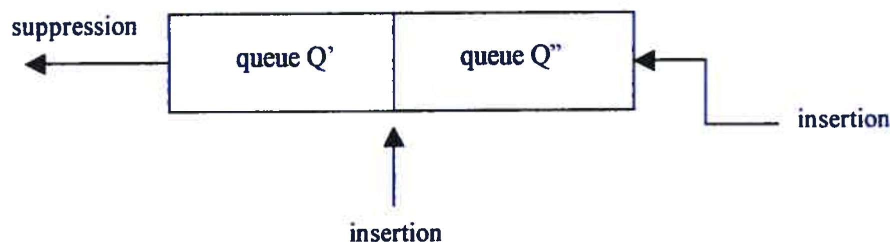


Figure 3.4 : Structure de données *2queue*.

Cette structure permet les mêmes opérations qu'une queue, mais elle permet aussi une insertion supplémentaire à la fin de la première queue (Q' sur la figure 3.4). Bien qu'elle soit un peu plus complexe qu'une simple queue, son implantation devient relativement peu compliquée maintenant que la classe QUEUE<T> a été implantée. La figure 3.5 permet de mieux voir cette structure et, surtout, les pointeurs utilisés pour la gérer.

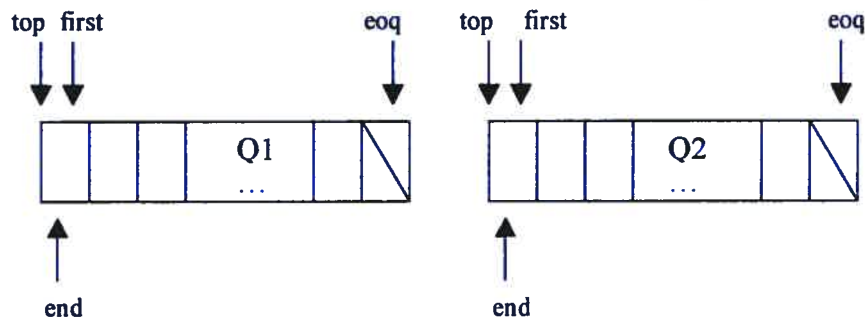


Figure 3.5 : Structure de données *2queue*.

Les pointeurs associés aux deux queues (Q1 et Q2 sur la figure 3.5) se comportent exactement de la même façon que sur la figure 3.2. Nous avons, cette fois aussi, développé une classe paramétrée 2QUEUE<T>, qui utilise directement deux objets de la classe QUEUE<T>. Voici maintenant le code C++ relié à l'implantation de cette classe, suivi de l'interface de cette classe dans le tableau 3.III.

```
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);
        _queue-2>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();
    }
}
```

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 dans Q2).
long size() const;	Permet de connaître la taille de la structure.
int empty() const;	Permet de savoir si la structure est vide.

Tableau 3.III : Interface de la classe *2QUEUE*.

Encore une fois, on peut remarquer qu'à l'exception des méthodes d'insertion, l'interface de cette classe est la même que celles des classes QUEUE<T> et DEQUE<T>. Si t et s sont définis comme précédemment, alors le nombre de bytes nécessaires à la création d'un objet de cette classe est $8 + 2(28 + st)$. Cette structure de données sera utilisée dans

l'implantation d'un outil de calcul des plus courts chemins statiques nécessaire à l'algorithme de Chabini, DOT. C'est d'ailleurs cet outil de calcul que nous présentons à l'instant.

3.5 OUTIL DE CALCUL DES PLUS COURTS CHEMINS STATIQUES BSP2QUEUE

Comme nous l'avons vu à la section 2.4, l'algorithme DOT de Chabini nécessite un outil de calcul des plus courts chemins statiques. Nicolas Tremblay [56], dans son mémoire, en avait développé plusieurs, mais il avait retenu BSP2QUEUE (*Backward Shortest Path* utilisant une *2queue*) puisqu'il avait obtenu les meilleurs résultats. Nous reprenons donc à notre tour ce même outil de calcul. Cet outil de calcul dérive d'une classe supérieure, BACKWARDSP, dont nous présentons l'interface au tableau 3.IV.

Méthodes***	Fonctionnalités
BSP?? (NFF &); BSP?? ();	Constructeurs.
void setNetwork (NFF &);	Permet de sélectionner un réseau.
void sp (const long id, float* a, int* b);	Calcul les plus courts chemins statiques pour un nœud destination à partir de tous les autres nœuds (toutes les origines).
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 respectivement aux nombre de liens, au nombre de nœuds et au nombre de centroïdes du réseau.

*** Les caractères ?? sont utilisés pour éviter de nommer toutes les classes d'outils de calcul concernées. Dans le cas de l'outil qui nous intéresse, on aurait donc *2QUEUE* qui remplacerait ces deux caractères.

Tableau 3.IV : Interface partagée par les outils de calculs par parcours arrière.

La méthode sp(-) du tableau 3.IV accepte trois paramètres : l'identificateur du nœud destination à considérer ainsi que deux vecteurs de dimension n pour conserver les résultats du calcul. Le premier vecteur (a[i]) gardera en mémoire le coût du chemin reliant i à la destination tandis que le deuxième vecteur (b[i]) gardera en mémoire le nœud suivant le nœud i sur le plus court chemin le reliant à la destination. Compte tenu que les entiers et les réels à simple précision requièrent un espace mémoire de 4 bytes sur la machine utilisée, l'espace mémoire total requis pour conserver les résultats est $2*4*n = 8n$ bytes.

L'utilisation de l'outil de calcul est très simple. Le court programme C++ qui suit montre comment utiliser l'outil de calcul BSP2QUEUE pour calculer les plus courts chemins pour chaque destination possible.

```
#include <BSP2QUEUE.h>
...
void main() {
    ..
    ifstream inputFile ("winnipeg.nff");

    NFF network;
    inputFile >> network;
    ...
    BSP2QUEUE tool (network);
    int nbNodes = tool.numberOfNodes();

    float* bestTripCost = new float[nbNodes];
    int* nextNode = new int[nbNodes];

    for (int i = 0; i < nbNodes; i++) {
        tool.sp (i, bestTripCost, nextNode);
        ...
    }
}
```

À l'aide des informations données par la méthode *numberOfNodes()*, nous sommes en mesure de construire les vecteurs qui nous permettent d'obtenir les résultats du calcul des plus courts chemins. Ensuite, pour chaque nœud origine, nous utilisons la méthode sp(-) pour déterminer ces plus courts chemins vers la destination. On peut remarquer que ce programme ne conserve pas les résultats des calculs, mais c'est qu'on suppose ici qu'une certaine opération est effectuée avec les résultats obtenus pour une origine avant d'en choisir une autre.

Nous allons maintenant présenter l'implantation de l'outil de calcul BSP2QUEUE. Cette implantation utilise un objet *2queue* de type 2QUEUE<*float*>. On fixe la taille de la queue à *n*/2. Les expérimentations déjà réalisées par Tremblay [56] permettent de conclure que cette borne est raisonnable, étant donné que le nombre d'éléments présents dans la *2queue* n'est jamais très élevé par rapport au nombre total de nœuds. Cette valeur nous assure qu'il n'y aura pas de débordements lors de l'exécution. L'objet *2queue* utilise

un espace mémoire de 8 + 2(28 + 2n) = 64 + 4n bytes. Pour éviter qu'un même élément soit inséré plusieurs fois dans la *2queue*, l'implantation associe à chaque nœud i une variable permettant de savoir si ce nœud i est dans la structure ou s'il l'a déjà été. Ceci nous permet de déterminer dans laquelle des deux queues de la *2queue* le nœud i doit être inséré. On définit donc la variable *elementOfQ[i]* ainsi :

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

Nous présentons maintenant le code C++ relié à l'implantation de la méthode sp(-) de l'outil de calcul des plus courts chemins statiques BSP2QUEUE.

```
void BSP2QUEUE::sp (const long
destination,
      float* bestTripCost, int* nextNode) {

      // initialisation
      for (int i = 0; i < nbNodes; i++ ) {
            nextNode[i] = -1;
            bestTripCost[i] = INFINITY;
           _elementOfQ[i] = 0;
      }

      bestTripCost[destination] = 0;

      // on ajoute les noeuds qui mènent à la
      // destination dans la 2queue
      long link;
      long
size=(*_incomingLinks)[destination].size();
      for (int i = 0; i < size; i++) {
            link =
(*_incomingLinks)[destination][i];
            int node = (*_fromNode)[link];
            bestTripCost[node] =
_costLink[link];
            nextNode[node] = destination;
            _elementOfQ[node] = 1;
            _2queue->insertInQ2 (node);
      }

      // Étape principale

      while (!_2queue->empty()) {
            long selectedNode = _2queue->remove();
            _elementOfQ[selectedNode] = -1;
            float bestCost =bestTripCost[selectedNode];

            long previousLink;
            float newCost;

            int size =
               (*_incomingLinks)[selectedNode].size();
            for (int i = 0; i < size; i++) {
                  previousLink =
                     (*_incomingLinks)[selectedNode][i];
                  int candidat =
                     (*_fromNode)[previousLink];
                  newCost =
                     bestCost+_costLink[previousLink];

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

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

Dans le prochain chapitre, nous allons maintenant présenter les implantations séquentielles des trois algorithmes vus au chapitre 2. Nous verrons que nous y utilisons tous les outils et structures présentés dans le chapitre courant. L'algorithme de Pallottino et Scutellà, ChronoSPT, utilise une queue, l'algorithme de Zilaskopoulos et Mahmassani, TDLTP utilise une *deque* tandis que l'algorithme proposé par Chabini. DOT, utilise l'outil de calcul des plus courts chemins statiques BSP2QUEUE. Cet outil utilise la structure de données *2queue* vue précédemment.

4. IMPLANTATION SÉQUENTIELLE DES ALGORITHMES DE CALCUL DES PLUS COURTS CHEMINS TEMPORELS

Dans ce chapitre, nous allons voir en détails comment a été faite l'implantation séquentielle des trois algorithmes de calcul des plus courts chemins temporels présentés au chapitre 2. L'objectif est ici de fournir des outils de base (calcul des plus courts chemins temporels) qui pourront ensuite être utilisés par des modèles de planification des transports plus complexes. On commencera par expliquer comment la modélisation des données a été réalisée et bien décrire l'environnement de travail qui a servi à réaliser l'implantation des algorithmes. Ensuite, on décrira en détails les implantations séquentielles des 3 algorithmes.

4.1 MODÉLISATION DES DONNÉES ET ENVIRONNEMENT DE TRAVAIL

On a vu au chapitre précédent que le choix du langage de programmation était une des décisions importantes à prendre avant d'entreprendre le développement de nos implantations. Bien sûr, ce n'est pas le seul critère important; il y a aussi la façon choisie de représenter les données, pour que l'information soit accessible de façon efficace. C'est sur ce critère que porte la prochaine section. Par la suite, nous décrirons sommairement l'environnement de travail utilisé.

4.1.1 MODÉLISATION DES DONNÉES

Dans cette sous-section, nous allons expliquer comment les données, modélisées en réseaux de transports, ont été représentées. C'est une partie très importante de notre phase de développement, car elle influence beaucoup les performances de notre application. Nous reverrons tout d'abord quelles sont les composantes qui caractérisent un réseau de transport. Ensuite, les détails sur la représentation de ces réseaux seront donnés. Finalement, un tableau permettra de bien visualiser les différences entre les divers réseaux utilisés pour tester nos implantations.

Le terminologie utilisée dans le cadre de ce mémoire est celle qui est le plus souvent utilisée dans le domaine des transports. Considérons le réseau présenté à la figure ci-dessous (figure 4.1, tirée de Tremblay [56]). Ce réseau est composé de quatre **nœuds** (*nodes* en anglais) et six **liens** (*links*). On dénote par **centroïde** (*centroid*), une zone origine et/ou destination. On représente un centroïde comme un « nœud virtuel » connecté à un ou plusieurs nœuds et par lequel il est impossible de transiter. Si on revient à la figure 4.1, les nœuds 1 et 4 pourraient être considérés comme centroïdes. Les chiffres sur chaque arc représentent le numéro de l'arc, suivi du coût de l'arc.

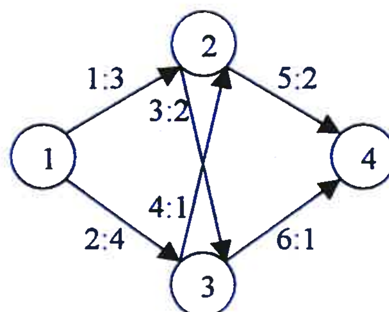


Figure 4.1 : Exemple de réseau.

Nous ne considérons pas, dans ce mémoire, les réseaux avec **virages** (*turns*) pénalisés, avec interdictions ou avec délais permis. Nous nous concentrons plutôt sur des réseaux temporels de base.

Nous allons maintenant expliquer comment ces simples réseaux seront représentés dans nos implantations. La représentation choisie devra être simple et permettre de traiter les données de façon efficace. De plus, cette représentation devra être suffisamment générale pour faciliter les échanges de données entre les 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 deux exigences ci-haut mentionnées. Il s'agit du format de fichier **NFF** (*Network File Format*)[11], développé par Éric Le Saux, Mike

[11] La documentation détaillée du format de fichier NFF est disponible en ligne à l'adresse suivante : http://www.crt.umontreal.ca/~lab_sit/DOC-NFF/.

Florian, Nicolas Tremblay et Ismaïl Chabini en 1996. Bien que conçu pour représenter différents types de données, le format de fichier NFF est particulièrement bien adapté à la représentation des réseaux. Pour donner un exemple, nous présentons à la figure 4.2 le fichier résultant de la représentation du réseau de la figure 4.1 à l'aide du format de fichier NFF. Chaque **section** (*Nodes, Centroids, Links*) comporte certains attributs. Par exemple, la section *Links* a quatre attributs : un identificateur (*ID*), un point de départ (*from*), un point d'arrivée (*to*) et un coût pour voyager sur l'arc (*Cost*).

```
<NFF 3.6.1>
<title>Exemple</title>
<section Nodes>
 <structure>
  long ID;
  float X;
  float Y;
 </structure>
 <data>
  1, 12.45, 40.35;
  2, 34.67, 54.12;
  3, 34.24, 26.90;
  4, 55.68, 39.56;
 </data>
</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>
```

Figure 4.2: Exemple de fichier NFF (correspondant au réseau de la figure 4.1).

L'utilisation du format de fichier NFF est d'autant plus naturelle, puisqu'il y a une librairie de support C++ qui lui est associée. Cette librairie permet le traitement efficace des données. On peut, entre autres, accéder facilement et rapidement aux liens incidents à chaque nœud. C'est une opération qui se produit très fréquemment dans les algorithmes de calcul de plus courts chemins dans des réseaux, d'où l'importance d'accéder à cette information rapidement, pour obtenir des applications performantes. La librairie de support associée au format de fichier NFF permet de générer pratiquement toute l'information nécessaire (nombre de liens entrants dans un nœud, par exemple) à partir des informations de bases contenues dans l'exemple de la figure 4.2. On a donc que le format de fichier NFF est simple et efficace, puisqu'on peut ainsi accéder à toute l'information voulue très rapidement.

Pour comparer les performances des trois algorithmes, nous avons utilisé trois jeux de données réelles correspondant aux réseaux de transport des villes canadiennes de Winnipeg, Ottawa et Montréal. Ces réseaux proviennent de la banque de données du

logiciel de planification en transport *emme/2* [36]. Les caractéristiques de ces réseaux sont assez différentes, comme le montre le tableau 4.I. On pourra voir si les caractéristiques des différents réseaux influent sur le comportement des algorithmes testés.

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

Tableau 4.I : Caractéristiques des réseaux de transport utilisés

4.1.2 ENVIRONNEMENT DE TRAVAIL

Nous allons maintenant décrire l'environnement de travail utilisé pour réaliser les implantations séquentielle et parallèle des algorithmes et ensuite effectuer les diverses expérimentations. La machine utilisée est le *SUNW Ultra-Enterprise 10 000* , une machine à 64[12] processeurs à mémoire partagée[13]. Les détails techniques de la machine sont présentés au tableau 4.II. Comme l'a démontré Nicolas Tremblay dans son mémoire [56], cet environnement de travail où la mémoire est partagée est plus utile au traitement parallèle qu'un réseau qui regrouperait plusieurs machines, entre autres parce que nous avons accès à une mémoire vive très importante (~60 Go). Cette quantité plutôt importante de mémoire permet de traiter de très gros réseaux. Les expérimentations auraient probablement pu être réalisées sur des réseaux de taille plus importante que celle du réseau de Montréal. Toutes les implantations ont été compilées par le compilateur SUN C++ disponible avec SOLARIS 5.8.[14]

Nom	Nombre de processeurs	Mémoire vive	Cache L2 par proc.	Système d'exploitation (OS)	Fréquence de l'horloge
SUNW Ultra-Enterprise 10000	60	61440 Mb	8 Mb	SOLARIS 5.8.	400 MHz

Tableau 4.II : Spécifications techniques de la machine utilisée

[12] À l'origine, la machine comportait 64 processeurs, mais seulement 60 étaient opérationnels lors de la réalisation des expérimentations.
[13] On définira plus en détails ce type d'architecture au prochain chapitre.
[14] CC : SUN Workshop 6 update 2 C++ 5.3.

4.2 IMPLANTATION SÉQUENTIELLE DES ALGORITHMES DE CALCUL DES PLUS COURTS CHEMINS TEMPORELS

Dans cette portion du mémoire, nous nous intéressons de plus près à l'implantation séquentielle de nos trois algorithmes de calcul des plus courts chemins temporels, déjà décrits au chapitre 2. Tel que mentionné précédemment, nous nous concentrons sur la variante du problème de calcul des plus courts chemins temporels en temps discret et cherchant les chemins qui minimisent les temps de parcours (*time-dependent least-time path*). Nous utiliserons, par la suite, pour alléger le texte, l'expression « plus courts chemins » pour désigner les chemins minimisant les temps de parcours. Nous nous concentrons aussi sur la variante du problème qui cherche les plus courts chemins de chaque nœud vers une destination donnée (*all-to-one*) et ce, pour chaque période de temps. Nous verrons maintenant le code C++ des trois algorithmes présentés à la section précédente.

Afin d'implanter ces trois algorithmes, les classes CHRONOSPT, TDLTP et DOT ont été développées. De plus, pour éviter la duplication d'information et tirer avantage du principe d'héritage, ces trois classes dérivent d'une classe supérieure, TDSP (*Time-Dependent Shortest-Path*). La figure 4.3 permet de bien visualiser les liens entre ces quatre classes. Pour faciliter les mises à jour et rendre le code un peu plus clair, les trois classes CHRONOSPT, TDLTP et DOT partagent une interface unique. Le tableau 4.III décrit les différentes parties de l'interface. La méthode sp(-) est la principale fonctionnalité. C'est la méthode qui permet de calculer les plus courts chemins. Elle accepte quatre paramètres : l'identificateur du nœud destination à considérer, suivi de trois matrices qui vont garder en mémoire différentes informations essentielles (voir tableau 4.IV).

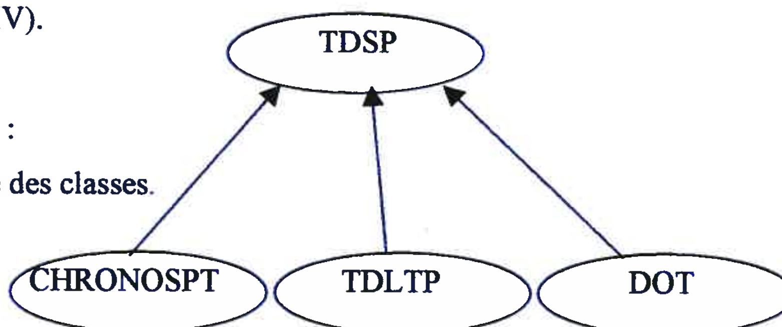


Figure 4.3 :
Hiérarchie des classes.

Méthodes[a]	Fonctionnalités.
?? (NFF& network) ?? ();	Constructeurs.
void setNetwork (NFF& network)	Permet de choisir 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 a été fait 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 conernées.

Tableau 4.III : Interface des classes CHRONOSPT, TDLTP, DOT.

Matrices	Définitions
a[t][i]	Temps d'arrivée à la destination à partir du nœud i, 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.

Tableau 4.IV : Définitions des matrices de la méthode sp.

Puisque sur la machine utilisée 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 est de $3 * 4 * n * |\mathcal{M}| = 12n|\mathcal{M}|$ bytes.

L'utilisation d'un outil se fait facilement. Si, par exemple, nous voulons 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.ff");
		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 j = 0; j < nbPeriods; j++)
		arrivalTimes[j] = new int[nbNodes];
	int** nextNodes = new int[nbPeriods];
	for (int j = 0; j < nbPeriods; j++)
		nextNodes[j] = new int[nbNodes];
	int** nextTimes = new int[nbPriods];
	for (int j = 0; j < nbPeriods; j++)
		nextTimes[j] = new int[nbNodes];
	for (int j = 0; j < nbCentroids; j++) {
		tool.sp (nodeCentroid[j], arrivalTimes, nextNodes, nextTimes);
		...
	}
	...
}
```

Pour nous permettre de construire les matrices nécessaires pour garder en mémoire les résultats du calcul des plus courts chemins, nous avons besoin de connaître le nombre de périodes de temps, le nombre de nœuds dans le réseau et aussi le nombre de centroïdes. Pour ce faire, dans cet exemple, les méthodes numberOfPeriods(), numberOfNodes() et numberOfCentroids() sont utilisées. Ensuite, pour chaque centroïde j, nous calculons, grâce à la méthode sp(-), les plus courts chemins reliant chaque nœud au centroïde traité. Il est important de noter que le premier paramètre de la méthode sp(-) doit correspondre à l'identificateur d'un nœud.

On remarque que ce programme ne conserve pas les résultats du calcul des plus courts chemins. On suppose ici qu'une opération est effectuée avec les résultats obtenus pour un certain centroïde avant d'en traiter un autre. Il pourrait parfois, dans le cadre de certains modèles de planification en transport, être fort utile, et même essentiel, de pouvoir conserver les résultats des calculs pour chacun des centroïdes considérés. Dans ce cas, il faudrait avoir une structure plus coûteuse en espace mémoire pour conserver les résultats des calculs des plus courts chemins pour chaque centroïde. Un espace mémoire de $|\mathcal{C}|$ * $12n|\mathcal{M}|$ bytes (où $|\mathcal{C}|$ est le nombre de centroïdes dans le réseau) serait nécessaire. Si on prend le réseau de Montréal 120 (le plus gros utilisé dans cette recherche), on aurait besoin d'environ 7 Go d'espace mémoire, ce qui est quand même considérable. Compte tenu de la quantité de mémoire disponible sur l'ordinateur utilisé, on aurait pu être tenté de conserver les résultats des calculs, mais ils n'étaient pas utiles pour cette recherche.

Nous allons maintenant décrire les implantations des algorithmes de calcul des plus courts chemins temporels associées à chacune des trois classes. Pour chacun des trois algorithmes utilisés dans ce mémoire, nous présentons les détails relatifs à l'implantation de l'algorithme, ainsi que le code C++ correspondant. Pour permettre de mieux comprendre les codes C++, nous définissons ici quelques variables importantes :

-travelTimes[t][i] : temps de parcours au temps t associé au lien i;

-fromNode[i] : nœud origine du lien i;

-toNode[i] : noeud terminal du lien i;

-incomingLinks[i] : liste des liens entrant au nœud i.

4.2.1 IMPLANTATION DE L'ALGORITHME CHRONOSPT

Il faut tout d'abord implanter la liste de compartiments suggérée par Pallotino et Scutellà [47] (on peut aussi consulter la section 2.2 au besoin). La figure 4.4 permet de voir une représentation de cette liste où chaque compartiment contient un identificateur (id) et un pointeur (queue) vers une queue (l'implantation d'une queue a été vue au chapitre précédent).

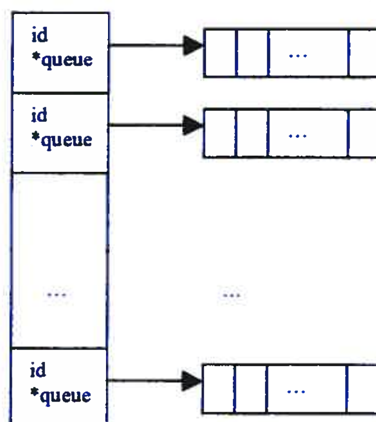


Figure 4.4 : Liste de compartiments.

Si on veut ajouter un élément à la structure, il suffit de fournir l'identificateur (*id*) du compartiment 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 j = 0; j < size1; j++) {
			buckets[j].queue = new CRTQueue<int>
(size2);
			buckets[j].id = j;
		}
		current = end;
	}
	void add (int value, int bck) {
		buckets[bck].queue → insert (value);
	}
	int remove () {
		return *current.queue → remove ();
	}
	int bucketNumber () {
		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;
	...
};
```

Supposons que chaque queue associée à chacun des C compartiments ait une taille q. Alors, un objet de la classe BUCKET occupe un espace mémoire de $16 + C(8 + 28 + 4q) = 16 + 4C(9 + q)$ bytes. Pour ne pas insérer un même élément plusieurs fois dans la liste de compartiments, nous avons associé à chaque doublet (nœud *i*, période de temps *t*) une variable permettant de savoir si un nœud *i* est déjà contenu ou a déjà été contenu dans la structure au temps *t*. Le tableau elementOfQ[-][-] permet de conserver cette information. Pour un nœud *i* au temps *t*, nous avons que

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

Voyons maintenant le code C++ correspondant à la méthode sp(-) pour cette classe :

```
int CHRONOSPT :: sp (const long id, int**
        arrivalTimes, int** nextNodes, int**
        nextTimes) {

    // On vérifie s'il y a des liens entrant à 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);

    // Itération 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;
}
```

4.2.2 IMPLANTATION DE L'ALGORITHME TDLTP

Pour l'implantation de l'algorithme de Ziliaskopoulos et Mahmassani [60] (voir la section 2.3 au besoin), nous avons choisi une *deque* comme structure de données. L'implantation de cette structure a été vue au chapitre précédent. Cette structure de données avait également été utilisée par Ziliaskopoulos et Mahmassani dans leur implantation de l'algorithme TDLTP [60]. Cette fois aussi, pour ne pas insérer un élément plusieurs fois dans la liste, nous associons à chaque nœud i une variable permettant de savoir si ce nœud est déjà contenu ou s'il a déjà été contenu dans la structure. Le tableau elementOfQ[-] permet de conserver cette information. Pour un nœud i, nous avons donc que

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

Le code C++ correspondant à la méthode sp (-) est le suivant :

```
int TDLTP :: sp (const long id, int** arrivalTimes, int** nextNodes, int** nextTimes) {

    // On vérifie s'il y a des liens entrant à la destination
    int size = incomingLinks[id].size();
    if (!size)
        return 0;

    // Initialisation
    for (int t = 0; t < nbPeriods; t++)
        for (int i = 0; i < nb Nodes; 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 à la destination sont introduits //dans la deque
    long link, candidat;
    for (int i = 0; i < size; i++) {
        link = incomingLinks[id][i];
        candidat = 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
```

```
                        nexTimes[t][candidat] = t +
                        travelTimes[link][t];
                }

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

// Itération principale
        while (!deque->empty()) {
                long current = deque->remove();
                elementOfQ[current] = -1;

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

                        for (int t = 0; t < nb Periods; 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 nœud candidat est inséré dans la deque // si au moins une de ses étiquettes est //modifiée
                        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;
}
```

4.2.3 IMPLANTATION DE L'ALGORITHME DOT

L'implantation de l'algorithme proposé par Chabini [10] (voir la section 2.4 au besoin) est relativement différente des deux autres puisque cet algorithme requiert l'utilisation d'un outil de calcul des plus courts chemins statiques effectuant une recherche par

parcours arrière. Les résultats obtenus par Nicolas Tremblay [56] suggèrent d'utiliser l'outil BSP2QUEUE (dont nous avons présenté l'implantation à la section 3.5). Ce dernier, selon lui, était l'outil le plus efficace parmis ceux qu'il avait implanté. C'est donc cet outil que nous réutilisons à notre tour. Le code C++ correspondant à la méthode sp (-) est le suivant :

```
int DOT :: sp (const long id, int** arrivalTimes, int** nextNodes, int** nextTimes) {

    // On vérifie s'il y a des liens entrant à 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 calcul des plus courts chemins //statiques est utilisé pour la dernière période 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;
        }

    // Itération principale
    int from, to, tt;
    for (int 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[t][to]) {

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

                nextNodes[t][from] = to;
                nextTimes[t][from] = tt;
            }
        }
    return 1;
}
```

4.3 EXPÉRIMENTATIONS ET ANALYSE DES RÉSULTATS

Dans cette partie, nous nous intéressons à l'analyse des performances des implantations séquentielles développées dans le présent chapitre. Afin de tester ces implantations, nous avons généré des temps de parcours $d_{ij}(t) \in [1,5]$ entiers pour les réseaux urbains des villes canadiennes de Winnipeg, Ottawa et Montréal (les caractéristiques de ces réseaux ont été données à la section 4.1.2). Cette opération a été répétée pour des intervalles variant de 30, 60, 90 à 120 périodes de temps.

Le tableau 4.V présente les temps d'exécution obtenus sur une machine SUNW Ultra-Enterprise 10000, pour chacun des trois algorithmes, pour chacun des douze cas disponibles (les réseaux des villes de Winnipeg, Ottawa et Montréal pour 30, 60, 90 et 120 périodes de temps). Dans chaque cas, on retrouve le temps de calcul global correspondant au traitement de toutes les destinations. Le temps mesuré est le temps réel requis par l'application et il est exprimé en secondes. Il s'agit ici de la moyenne de cinq exécutions des algorithmes où on a enlevé les quelques données jugées aberrantes (écart de plus de 15% par rapport à la moyenne).[15]

Réseau / $|\mathcal{M}|$	CHRONOSPT	TDLTP	DOT
Winnipeg / 30	1.571075	1.87116	1.886985
Winnipeg / 60	3.00629	4.536865	4.002515
Winnipeg / 90	4.487665	7.595955	6.312253
Winnipeg / 120	6.896795	13.20103	12.946
Ottawa / 30	11.21288	8.95793	8.87991
Ottawa / 60	21.6749	20.71733	18.78563
Ottawa / 90	33.02885	35.096	34.88785
Ottawa / 120	42.66903	53.35335	53.30793
Montréal / 30	81.37175	65.15995	83.1965
Montréal / 60	154.3853	186.1265	305.6063
Montréal / 90	247.3215	301.776	407.6738
Montréal / 120	360.6995	525.6568	591.535

Tableau 4.V : Temps d'exécution séquentielle

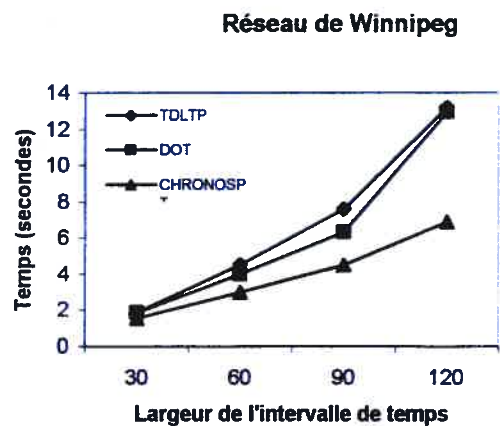
[15] Bien que la queue de priorité mise en place sur la machine pour traiter les demandes des usagers réserve tous les processeurs pour l'algorithme, certains usagers peuvent tout de même se connecter et lancer certaines tâches sans passer par la queue de priorité. De plus, il se peut très bien que ces résultats aberrants soient dûs à des copies de sécurité effectuées par le système sans avertissement, pendant la nuit (période où ont été effectués les tests).

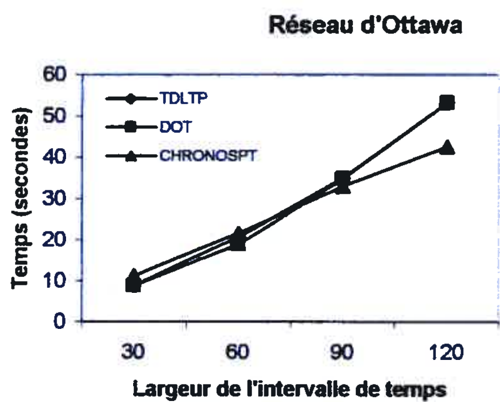
On ne peut pas vraiment comparer les résultats d'un algorithme par rapport à un autre parce que certaines améliorations de performances ont été appliquées à l'algorithme TDLTP, alors que certaines améliorations au niveau de la stabilité ont coûté en efficacité pour l'algorithme DOT. En effet, les calculs effectués par l'algorithme DOT étaient faussés à l'occasion par un mauvais appel de constructeur. De plus, un certain pointeur était parfois détruit par erreur. Ces deux modifications ont grandement modifié les résultats de l'algorithme DOT. Il est maintenant plus lent, mais plus stable. Nos résultats concordent donc plus ou moins avec ceux dans la littérature (Chabini [14], Tremblay [56]). Théoriquement, l'algorithme TDLTP devrait être moins efficace que les deux autres, mais une amélioration importante au niveau de l'implantation a résulté en un gain important de performance. En effet, une technique vue dans SunTune [53] a été utilisée. Les matrices de la méthode sp(-) ont été implantées en inversant les lignes et les colonnes. Ceci a eu pour effet d'utiliser davantage la mémoire cache, réduisant ainsi les accès à la mémoire virtuelle (accès beaucoup plus lents) (voir Patterson et Hennessy [50] ou Heuring et Jordan [34] au besoin). Cette même technique s'est toutefois avérée inefficace dans le cas des deux autres algorithmes, si bien qu'elle n'a pas été utilisée. D'autres techniques vues dans SunTune ont été testées sans grands résultats concluants.

Afin d'analyser plus en profondeur les résultats obtenus, nous présentons les figures 4.5 et 4.6 qui permettent de bien visualiser l'influence des différentes caractéristiques des réseaux sur le comportement des algorithmes. Tout d'abord, les trois graphiques de la figure 4.5 nous montrent l'influence de la largeur de l'intervalle de temps sur le temps d'exécution des trois algorithmes. On peut y voir que les trois algorithmes se comportent étonnamment de façon similaire[16]. En effet, de par la complexité de calcul des trois algorithmes, on aurait pu s'attendre à ce que l'algorithme TDLTP réagisse davantage à l'augmentation de la période de temps, puisque sa complexité varie quadratiquement avec $|\mathcal{M}|$, contrairement aux deux autres, dont la complexité varie linéairement avec $|\mathcal{M}|$. Pourtant, ceci n'est pas apparent dans les résultats obtenus. Ensuite, dans la figure 4.6, on montre l'influence des caractéristiques du réseau sur chaque algorithme. Les résultats présentés dans cette figure sont ceux pour une largeur d'intervalle de 60 périodes de

[16] Garder en mémoire la remarque du paragraphe précédent.

temps. Encore une fois, les résultats obtenus ne concordent pas avec la complexité de calcul des algorithmes, puisque l'on remarque que DOT réagit fortement à la taille du réseau, contrairement aux deux autres, où les changements sont moins importants. Encore là, la performance de TDLTP est beaucoup mieux que prévue, due aux améliorations apportées à l'algorithme.





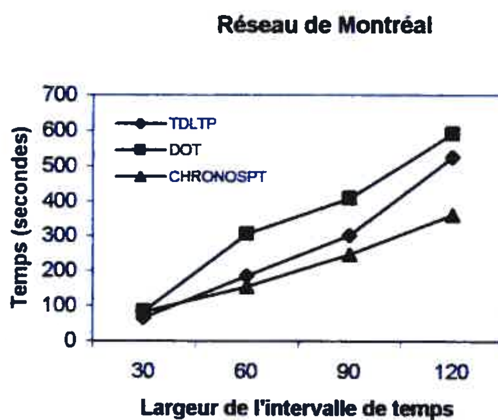


Figure 4.5 : Influence de la période de temps sur le temps d'exécution

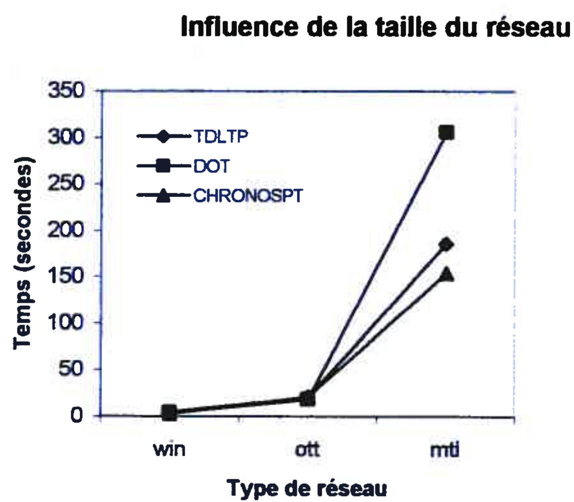


Figure 4.6 : Influence de la taille du réseau sur le temps d'exécution

5. TRAITEMENT PARALLÈLE DES ALGORITHMES DE CALCUL DES PLUS COURTS CHEMINS DANS DES RÉSEAUX DYNAMIQUES

Au fur et à mesure que la recherche opérationnelle avance et progresse, plusieurs modèles requièrent des calculs toujours plus exigeants. Même si les processeurs sont de plus en plus rapides, certains modèles exigent encore un effort de calcul trop grand pour permettre d'obtenir des solutions en temps raisonnable. Même que de plus en plus d'applications dans le domaine de la planification en transports requièrent des réponses en temps réel. Le calcul parallèle [3, 6, 58] est une des approches qui permettent de résoudre certains de ces problèmes trop exigeants ou pour lesquels nous avons besoin d'une réponse en temps réel, puisqu'il nous permet de les résoudre en une fraction du temps.

Certains chercheurs ont déjà travaillé sur l'implantation parallèle de certains modèles de recherche opérationnelle [8, 26, 61], en particulier, les algorithmes de calcul des plus courts chemins. Florian, Chabini et Le Saux [26] ont utilisé l'environnement parallèle PVM (*Parallel Virtual Machine*) pour tenter de résoudre le problème d'équilibre à demande fixe. Ils ont en fait travaillé sur un réseau de 16 stations de travail, dont ils ont comparé les résultats avec ceux obtenus par une implantation *multithreads* utilisant le modèle de décomposition maître-esclave sur une machine multiprocesseurs à mémoire partagée. Ils ont pu en conclure que l'implantation *multithreads* performe mieux que l'implantation utilisant l'environnement PVM. C'est pourquoi nous nous concentrons seulement sur l'implantation parallèle *multithreads* dans ce mémoire. Ziliaskopoulos, Mahmassani et Kotzinos [61] ont eux aussi comparé diverses implantations parallèles. Ils ont conçu deux implantations de l'algorithme TDLTP (vu au chapitre 2) utilisant le concept de mémoire partagée et aussi une utilisant une stratégie d'échanges de messages sur un environnement PVM. Ils en concluent que la parallélisation d'un algorithme de calcul des plus courts chemins conventionnels n'est pas efficace. Il est en effet beaucoup plus utile d'affecter directement une destination à chaque processeur disponible, résultat aussi remarqué par Florian, Chabini et Le Saux.

Dans ce chapitre, nous présentons l'implantation parallèle des algorithmes de calcul des plus courts chemins temporels vus au chapitre 2. Plus spécifiquement, nous utilisons la programmation *multithreads* pour évaluer l'impact du calcul de ces plus courts chemins sur une machine à mémoire partagée. La première section de ce chapitre présente l'environnement choisi pour effectuer les implantations parallèles et introduit quelques concepts relatifs au traitement parallèle. Dans la deuxième section, nous présentons les codes C++ des implantations parallèles qui ont été développées.

5.1 TRAITEMENT PARALLÈLE

Dans cette section, nous verrons tout d'abord certains concepts reliés au traitement parallèle. Ensuite, nous présentons l'environnement *multithreads* utilisé pour développer nos implantations. Pour terminer, nous définissons les mesures de performances utilisées pour analyser l'efficacité de nos implantations parallèles.

5.1.1 QUELQUES CONCEPTS DE BASE RELIÉS AU TRAITEMENT PARALLÈLE

Avant de se lancer dans le calcul parallèle, il est nécessaire de connaître certains concepts essentiels à une bonne compréhension. Il faut, de plus, se familiariser avec une nouvelle terminologie spécifique au calcul parallèle. Plusieurs des concepts de base relatifs au traitement parallèle sont définis par Chabini [9].

Un des concepts importants est la décomposition. La décomposition est la division d'une tâche en plusieurs sous-tâches qui pourront être exécutées par différents processeurs. Il faut tout d'abord déterminer la taille des sous-tâches (ce qu'on appelle le partionnement). On peut utiliser un parallélisme à gros grains, principe selon lequel chaque sous-tâche correspond, en général, à une procédure comprenant plusieurs instructions machine. On peut aussi utiliser un parallélisme à grains fins, où, en général, chaque sous-tâche correspond à une instruction machine. Ensuite, il faut planifier l'exécution des sous-tâches, en tenant compte des contraintes de précédence et de l'architecture des

processeurs (ce qu'on appelle l'allocation). Autre étape importante de la décomposition, le balancement de charge, consiste à uniformiser les temps d'exécution des sous-tâches sur chacun des processeurs pour obtenir une meilleure efficacité. Aussi, un autre facteur dont il faut tenir compte lors de la décomposition, est la communication entre les sous-tâches. Puisque certaines d'entre elles devront s'échanger de l'information, et ceci peut grandement affecter la performance de l'application, il est important de tenter de minimiser ces échanges d'informations et choisissant bien nos sous-tâches. Le dernier concept relié à la décomposition est la synchronisation. Un processeur peut évoluer de façon synchrone ou asynchrone. On dira qu'un processeur fonctionne de façon synchrone s'il doit attendre à un certain point précis de l'exécution jusqu'à l'arrivée d'une donnée ou jusqu'à ce qu'un certain nombre d'étapes aient été accomplies par d'autres processeurs. On dira plutôt qu'un processeur fonctionne de façon asynchrone s'il n'y a pas d'attente en aucun point de l'exécution. Dans ce cas, il devient plus difficile de garantir la validité de l'algorithme. Le mécanisme de synchronisation qui sera utilisé peut avoir un impact important sur la performance de l'application. Il faudra donc faire ce choix judicieusement.

Il existe deux types d'architecture multiprocesseurs : à mémoire partagée (*shared-memory multiprocessors*) ou à échanges de messages (*message-passing multiprocessors*). Dans le premier cas, chaque processeur a accès à une mémoire commune divisée en modules. Deux processeurs, appelés source et récepteur, procurent un moyen de communication efficace. En effet, la source écrit l'information en mémoire commune et le récepteur vient ensuite la lire. Bien qu'efficace, ce moyen de communication amène un problème d'accès simultanés par plusieurs processeurs à la même information (à la même cellule mémoire). Ces conflits retardent l'accès et diminuent ainsi la vitesse des communications. Celles-ci demeurent toutefois relativement peu coûteuses, nous amenant à favoriser un parallélisme à grains fins. Dans le second cas, chaque processeur a sa propre mémoire qu'il peut accéder directement. Il n'y a donc pas de conflits possibles, mais cela amène bien évidemment un autre problème. En effet, pour que chaque processeur ait accès à l'information voulue, il faut qu'elle soit dupliquée autant de fois qu'il y a de processeurs. L'espace mémoire total peut évidemment devenir rapidement

trop important. Les communications entre processeurs sont aussi beaucoup plus lentes que dans le cas de mémoire partagée, puisque qu'elles doivent se faire à travers les liens physiques formant le réseau d'interconnections. Puisque les communications sont très coûteuses, on favorise souvent un parallélisme à gros grains pour ce type d'architecture.

5.1.2 LA PROGRAMMATION *MULTITHREADS*

On recherche toujours à obtenir des exécutions de plus en plus rapides pour toutes nos applications. Les machines multiprocesseurs à mémoire partagée nous permettent d'obtenir ces exécutions plus rapides, et donc peu coûteuses, pour le calcul parallèle. En utilisant la programmation *multithreads* sur ce type de machine, on peut ainsi obtenir un rendement plutôt satisfaisant.

Premièrement, avant d'aborder le sujet de programmation *multithreads*, il nous apparaît nécessaire de bien définir le concept de *thread*. Lewis et Berg [5], dans leur excellent livre pour quiconque souhaite s'introduire à la programmation multihreads, définissent très bien ce qu'est un *thread*. Comme les systèmes d'exploitation multitâches permettent de réaliser plusieurs opérations simultanément en exécutant plus d'un processus, un processus peut faire de même en utilisant plusieurs *threads* [5]. Chaque *thread* est une suite d'instructions qui pourront être exécutées indépendamment des autres *threads*. Ainsi, un processus *multithreads* peut exécuter un certain nombre de tâches en concurrence, chaque *thread* étant associé à une tâche. La distinction entre un processus et un *thread* n'est donc pas évidente. Pour mieux la comprendre, Lewis et Berg 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. Il est important de noter que les registres (pointeur de pile (*stack*)) et le 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 global et peut être exécuté simultanément par d'autres

threads. La figure 5.1 montre un exemple où deux *threads* T1 et T3 exécutent la même fonction. On peut y voir les pointeurs de pile (sp) et les marqueurs d'instructions (pc) associés à chacun des trois *threads* présents. On peut bien y voir que la structure du processus se situe dans le noyau du système d'exploitation tandis que les *threads* se situent plutôt au niveau utilisateur.

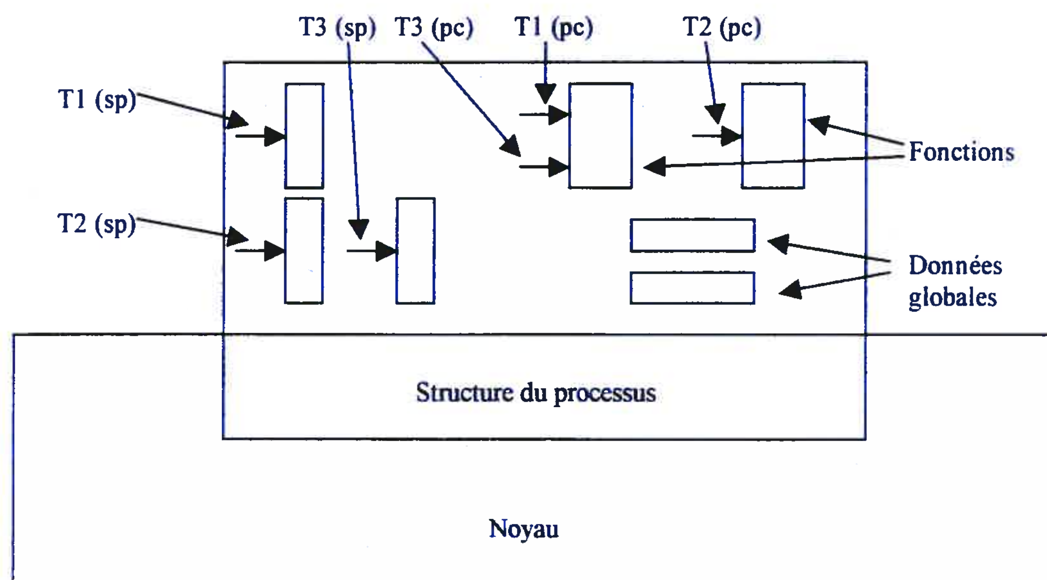


Figure 5.1 : Relation entre un processus et les *threads* y étant associés.

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 suivantes 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 incrémente D de 1 : T1.registre = T1.registre+1
4. T2 incrémente D de 1 : T2.registre = T2.registre+1
5. T1 réécrit dans D : T1.registre=1 -> D
6. T2 réécrit 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 acquière D.cle
2. T2 essaye d'acquérir D.cle, mais doit attendre que T1 la redonne
3. T1 lit D : T1.registre <- D=0
4. T1 incrémente D de 1 : T1.registre = T1.registre+1
5. T1 réécrit dans D : T1.registre=1 -> D
6. T1 redonne D.cle
7. T2 est réveillé et acquière D.cle
8. T2 lit D : T2.registre <- D=1
9. T2 incrémente D de 1 : T2.registre = T2.registre+1
10. T2 réécrit dans D : T2.registre=2 -> D
11. T2 redonne D.cle

Maintenant, nous avons que D = 2. Par contre, la suite d'instructions est maintenant 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*.

Tremblay explique bien les problèmes de synchronisation que l'on peut rencontrer [56]. Ils sont dûs principalement aux structures de données partagées (SDP). Leur présence 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.

Comme la présence de SDP influe beaucoup sur les performances de nos applications, à cause des mécanismes de synchronisation, il est important de concevoir nos algorithmes en ayant en tête de limiter ou d'éliminer leur présence.

5.1.3 MESURES DE PERFORMANCE DES IMPLANTATIONS PARALLÈLES

La performance d'une implantation parallèle peut être mesurée de plusieurs façons [9, 11, 12, 30] et la mesure de cette performance dépend du critère d'efficacité jugé important. Pour notre part, le critère d'efficacité choisi est le temps d'exécution. Concernant ce critère, il y a plusieurs mesures différentes 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

$$a(n,p) = \frac{T_s(n)}{T(n,p)} \tag{5.1}$$

Bien qu'idéalement a(n, p) = p, nous avons généralement que a(n, p) < p. Notons 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.[17]

La **charge relative** est une mesure de performance introduite par Chabini [9]. Idéalement, l'exécution parallèle a une durée de $T_s(n)/p$. La charge relative des processeurs est une mesure qui tient compte du temps supplémentaire (par rapport au temps idéal) que les processeurs prennent pour finir le traitement. Cette mesure inclut toutes les causes de retards possibles que peut causer une implantation parallèle, notamment les délais de communications entre les processeurs, une des causes importantes de perte de temps lors de l'exécution. Aussi, mentionnons le temps d'attente des processeurs, le temps d'exécution des parties non parallélisables de l'algorithme, le temps supplémentaire requis par les instructions exclusives au code parallèle par rapport à l'implantation séquentielle et la variation dans le temps de calcul due au fait qu'un algorithme parallèle peut emprunter un chemin différent de celui emprunté par la version séquentielle. La charge relative b(n, p) est obtenue en divisant la charge des processeurs par le temps d'exécution de l'implantation séquentielle[18]. La charge relative s'évalue en pratique de la façon suivante :

$$b(n,p) = \frac{T(n,p)}{T_s(n)} - \frac{1}{p}. \tag{5.2}$$

Cette mesure de charge relative peut être particulièrement utile si l'accélération ne permet pas de vraiment différencier le comportement de deux implantations parallèles. En effet, à chacune des courbes d'accélération peut correspondre des charges relatives très différentes. Nous pouvons donc porter un meilleur jugement sur l'efficacité de ces

---

[17] Toutefois, nous verrons que nous avons quand même dû utiliser l'implantation parallèle sur 1 processeur dans nos calculs. Les explications seront données à la prochaine section.

[18] Nous avons dû remplacer le temps d'exécution séquentiel par celui de l'exécution parallèle 1 processeur. Les détails seront donnés à la prochaine section.

implantations. Un autre avantage théorique que procure l'utilisation de cette mesure est la possibilité d'estimer la meilleure accélération possible en utilisant un grand nombre de processeurs p. Puisque nous avons que

$$a(n,p) = \frac{1}{1/p + b(n,p)} , \tag{5.3}$$

nous obtenons alors, lorsque p $\rightarrow +\infty$,

$$a(n,p) \approx \frac{1}{b(n,p)} . \tag{5.4}$$

La charge relative nous offre donc la possibilité d'effectuer une analyse plus juste des performances de notre implantation parallèle. De plus, il est théoriquement possible de faire des prévisions sur le comportement de l'implantation pour un nombre de processeurs hypothétiques.

5.2 IMPLANTATIONS PARALLÈLES DES ALGORITHMES DOT, CHRONOSPT ET TDLTP

Dans cette section, nous présentons en détails les implantations parallèles des trois algorithmes de calcul des plus courts chemins temporels présentés au chapitre 2, c'est-à-dire, les algorithmes DOT, CHRONOSPT et TDLTP. Nous avons utilisé la programmation *multithreads* pour développer ces implantations parallèles destinées au calcul sur une machine multiprocesseurs à mémoire partagée. Nous terminerons cette section avec une analyse des résultats obtenus.

L'utilisation de la programmation *multithreads* permet d'obtenir une implantation pouvant exploiter les caractéristiques d'une machine à mémoire partagée. Le principal avantage d'un telle implantation est que les communications (opérations de lecture et d'écriture) se font à travers la mémoire partagée. L'information étant commune à tous les threads, il n'y a donc pas d'échange d'information à faire. Cependant, l'accès à cette information doit être contrôlé afin d'éviter les conflits, ce qui aurait pour effet de diminuer l'efficacité de l'implantation parallèle. Nous devons donc être particulièrement attentifs aux mécanismes de synchronisation utilisés pour régir l'accès aux données

partagées (variables globales, structure de données partagées, ...). Ce point sera explicité un peu plus loin dans cette section.

L'implantation développée utilise le schéma maître-esclave que nous présentons à la figure 5.2. Ce schéma est essentiellement composé d'un programme maître et d'un certain nombre de copies d'un programme esclave. Le programme maître, comme son nom l'indique, est celui qui est le maître des tâches à accomplir. Il a les tâches de bien répartir le travail, de rassembler les résultats et aussi de bien coordonner ces deux tâches. Il gère en quelque sorte l'ensemble du traitement parallèle. La tâche du programme esclave quant à elle est relativement simple. Elle consiste à exécuter le travail assigné par la programme maître et à lui communiquer les résultats obtenus. L'objectif ici n'est pas de tenter de paralléliser l'algorithme séquentiel, mais plutôt de considérer l'ensemble des destinations à traiter comme la tâche globale que l'on veut fragmenter en sous-tâches. Chaque esclave (thread) est responsable du calcul des plus courts chemins pour un certain nombre de destinations. 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 prend, aussitôt son calcul terminé, une nouvelle destination parmi l'ensemble des destinations non traitées.

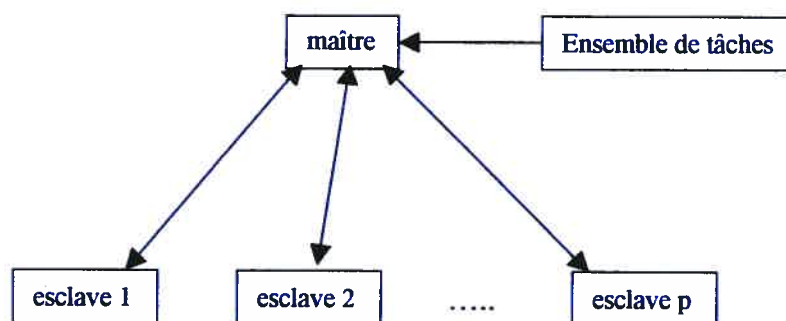


Figure 5.2 : Schéma maître-esclave

L'utilisation de la programmation *multithreads* résulte en un seul code exécutable pour le programme maître et le programme esclave. Ainsi, en 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 précise. Bien que nous sommes en présence d'un seul

code exécutable, le programme *multithreads* qui a été développé permet tout de même 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 traitées;
   2b. calcule les plus courts chemins pour la destination choisie.

La partie maître comprend trois étapes importantes. Dans la première étape, la programme maître doit lire le fichier NFF. Il doit entre autres s'assurer de fournir un objet NFF complet au constructeur de l'outil de calcul des plus courts chemins. Dans la deuxième étape, le programme maître créé un nombre d'outils de calcul correspondant au nombre de threads désirés, car chaque thread doit posséder son propre outil afin de pouvoir exécuter ses calculs indépendamment des autres threads. Enfin, dans la troisième étape, le nombre de threads désirés est crée et le programme maître attend la fin des calculs. Les opérations effectuées par chaque thread, quant à elles, se résument en deux étapes. La première étape consiste à choisir un outil de calcul des plus courts chemins. Pour le seconde, le programme esclave vérifie tout d'abord s'il y a encore des destinations qui n'ont pas été traitées. Si oui, il en choisit une parmi celles-ci et effectue le calcul des plus courts chemins pour cette destination.

Avant de passer au code C++ des implantations, nous voulons expliciter un peu l'usage des mécanismes de synchronisation[19]. Ces mécanismes sont essentiels en programmation *multithreads* pour bien gérer l'accès aux données partagées par les threads. Dans notre cas, il faut surtout bien coordonner la sélection d'un outil de calcul (chaque thread doit avoir son propre outil) et aussi la sélection des destinations à traiter (une destination ne

[19] Nous référons le lecteur au chapitre 5 du livre de Lewis et Berg pour plus de détails [5].

doit pas être traitée par deux threads en même temps, ni être traitée plus d'une fois). Le mécanisme de synchronisation utilisé dans notre implantation est le mécanisme de clé par exclusion mutuelle (*mutual exclusion lock*, ou simplement *mutex*), présent dans la librairie *multithreads*. Il a été choisi pour sa grande simplicité d'utilisation. Il permet à un et un seul thread à la fois d'exécuter une certaine portion du code lorsqu'il acquiert la clé. Tout autre thread qui tentera d'acquérir le clé sera « endormi » (*put to sleep*) et ne se réveillera que si la clé devient disponible.

Le code C++ présenté correspond à l'implantation *multithreads* de l'algorithme DOT. Nous ne présenterons pas les implantations correspondant aux algorithmes CHRONOSPT et TDLTP puisqu'elles sont identiques à celle de l'algorithme DOT. Pour alléger le code et ainsi permettre une meilleure compréhension, nous laissons de côté les détails associés aux procédures de la librairie *multithreads*. Les opérations en question sont toutefois décrites en pseudo-code, mais précédées par le symbole $\Rightarrow$.

La fonction spDOT() correspond au code exécuté simultanément par les *p* threads créés. Cette fonction est la fonction de calcul des plus courts chemins. Comme on peut le constater, le mécanisme de synchronisation s'assure d'abord 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 a fini de traiter une destination, il essaye immédiatement d'acquérir la clé dans le but de choisir la prochaine destination (s'il y en a une) dans la liste.

```
#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() {
    ⇒ acquérir la clé
    int t = _threads++;
    ⇒ redonner la clé

    for(;;) {
        ⇒ acquérir la clé
        if (!_pos) {
            ⇒ redonner la clé
            break;
        }
        int i = --_pos;
```

```
            ⇒ redonner la clé

            toolDOT[t] -> sp (nodeCentroid[i],
                _p[t].arrivalTimes,
_p.nextNodes,
                _p[t].nextTimes);
        }
}

main (int argc, char* argv[]) {
        int nbThreads;
        if (argc == 2)
            nbThreads = sysconf
                (_SC_NPROCESSORS_ONL
                N);
        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 nécessaires à la création
        de l'outil // sont ajoutés
        ...

        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 résultats
        _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 (int i = 0; i < nbPeriods; i++)
                _p[k].nextNodes[i] = new
int[nbNodes];
            _p[k].nextTimes = new int[nbPeriods];
            for (int i = 0; i < nbPeriods; i++)
                _p[k].nextTimes[i] = new int[nbNodes];
        }

        _threads = 0;
        _pos = centroids.size();

        ⇒ crée nbThreads qui exécutent la fonction
        spDOT
        ⇒ attend que les nbThreads aient terminé leurs
        calculs
}
```

Le programme *multithreads* développé n'a pas besoin d'être obligatoirement exécuté sur une machine multiprocesseurs, puisque le nombre de threads créés 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 comme la version séquentielle correspondante, mais on devrait remarquer un coût d'exécution supplémentaire, dû à la présence d'un mécanisme de synchronisation.

5.3 EXPÉRIMENTATIONS ET ANALYSE DES RÉSULTATS

Cette partie porte sur l'analyse des performances de l'implantation *multithreads* présentée précédemment. L'environnement parallèle utilisé est le *SUNW Enterprise 10 000* décrit à section 4.1.2. Tel que mentionné plus tôt, seulement 60 des 64 processeurs de cette machine à mémoire partagée étaient disponibles lors des expérimentations. Les réseaux routiers des villes canadiennes de Winnipeg, Ottawa et Montréal[20], pour des intervalles de 30, 60, 90 et 120 périodes de temps, sont utilisés pour effectuer les tests.

Nous présentons donc, aux tableaux 5.I, 5.II et 5.III, les temps d'exécution obtenus en fonction du nombre de threads utilisés et du type de réseau considéré (ville, nombre de périodes de temps). Le temps d'exécution est mesuré en secondes et correspond au temps réel requis par l'application. Les temps présentés représentent le temps moyen de 5 exécutions dont on a enlevé les quelques résultats jugés aberrants (écart de plus de 15% par rapport à la moyenne).[21] Nous avons utilisé un maximum de 60 threads pour réaliser les expérimentations. À partir des résultats de ces trois tableaux, nous avons évalué l'accélération et la charge relative. Nous avons représenté graphiquement les courbes associées à ces deux mesures pour les trois algorithmes implantés. Les figures 5.3 à 5.26 (présentées à l'annexe I) montrent ces graphiques pour chacun des réseaux considérés (ville, nombre de périodes de temps).

---

[20] Les caractéristiques de ces réseaux ont été présentées au tableau 4.I.

[21] Voir la note en bas de page à la page 64.

Réseau / période de temps	Nombre de threads							
	1	2	4	8	16	32	48	60
Winnipeg/30	1.29046	0.64315	0.340209	0.195576	0.099469	0.05841	0.050886	0.053901
Winnipeg/60	3.13471	1.502108	0.793676	0.471002	0.23878	0.245178	0.097593	0.090917
Winnipeg/90	4.566215	2.308957	1.162048	0.663041	0.373767	0.205961	0.157195	0.14216
Winnipeg/120	7.125095	3.083202	1.61083	0.916314	0.526614	0.294991	0.233082	0.204531
Ottawa/30	10.6909	5.408573	2.744162	1.446628	0.761734	0.394005	0.275371	0.232615
Ottawa/60	23.16755	11.04982	5.755258	2.926215	1.512007	0.881519	0.593029	0.571662
Ottawa/90	33.14782	16.80617	8.393893	4.409558	2.308693	1.228822	1.441175	1.74973
Ottawa/120	43.63848	2.72047	11.56938	5.899003	3.116027	2.226215	2.680638	3.093878
Montréal/30	79.74237	39.05837	20.17963	10.24172	5.406233	3.22486	.3.845662	3.42369
Montréal/60	155.0138	85.79592	40.22033	20.69402	10.98442	10.59335	8.917682	10.12742
Montréal/90	256.8147	126.8887	63.63567	32.20265	17.06528	17.85018	22.7309	25.53058
Montréal/120	359.3778	180.2335	88.44777	44.7979	34.70137	36.3394	44.98995	48.83392

Tableau 5.I : Implantation parallèle : temps d'exécution de l'algorithme Chrono-SPT.

Réseau / période de temps	Nombre de threads							
	1	2	4	8	16	32	48	60
Winnipeg/30	1.876792	0.9625	0.486259	0.303611	0.151359	0.08638	0.070729	0.076512
Winnipeg/60	3.919115	1.982547	1.01297	0.574	0.309858	0.248309	0.227988	0.155949
Winnipeg/90	6.3525	3.203835	1.979818	1.386337	0.691752	0.299282	0.284427	0.192812
Winnipeg/120	13.21652	6.014675	2.2397	1.316267	0.863121	0.44423	0.354745	0.240488
Ottawa/30	8.97346	4.437945	2.769165	2.989688	1.434273	0.48558	0.26649	0.242339
Ottawa/60	18.6167	12.26255	7.46748	3.941027	1.462622	0.996897	1.100492	0.571784
Ottawa/90	34.21835	21.85472	8.09377	4.759368	2.1798	1.582697	1.169507	1.222892
Ottawa/120	53.2155	30.2534	17.70872	5.864777	4.256245	2.051938	1.764998	1.785152
Montréal/30	82.08957	40.4287	23.36637	9.302305	11.19132	3.947843	2.587583	2.246897
Montréal/60	298.1663	211.1797	93.32425	49.19763	26.29455	15.44512	11.37255	10.20137
Montréal/90	400.9755	315.5882	148.1517	64.47273	33.5572	15.51242	12.92595	11.61395
Montréal/120	588.6295	333.0688	170.4852	75.84368	42.21722	32.25388	25.19247	21.56018

Tableau 5.II : Implantation parallèle : temps d'exécution de l'algorithme DOT.

Réseau / période de temps	Nombre de threads							
	1	2	4	8	16	32	48	60
Winnipeg/30	1.892535	1.055092	0.527447	0.305938	0.227493	0.09225	0.074193	0.070623
Winnipeg/60	4.492682	2.371602	1.24976	0.677311	0.372509	0.232642	0.150759	0.125423
Winnipeg/90	7.620335	4.050433	2.128522	1.25591	0.658013	0.33259	0.233434	0.196166
Winnipeg/120	13.07177	5.905553	2.895233	1.600218	0.837937	0.513518	0.328963	0.269949
Ottawa/30	8.99986	4.988485	2.610022	1.504743	0.753749	0.388256	0.271996	0.244265
Ottawa/60	20.90835	11.38852	6.140178	3.092807	1.57552	0.979596	0.638984	0.512107
Ottawa/90	34.70138	20.74228	9.1709857	4.80372	2.604203	1.530435	1.178897	0.85886
Ottawa/120	53.16632	28.00507	14.50577	6.653773	4.021337	2.331665	1.475808	1.249518
Montréal/30	64.9157	36.84158	20.53122	9.595303	5.374848	2.79527	2.089677	1.619178
Montréal/60	187.9192	97.62212	47.02768	23.61443	12.08952	6.521097	4.558182	3.83791
Montréal/90	310.7052	161.2692	82.35458	41.11302	20.8536	11.29483	8.065007	6.772932
Montréal/120	536.2063	235.897	130.6853	64.28457	33.08838	18.06543	13.18148	11.25267

Tableau 5.III : Implantation parallèle : temps d'exécution de l'algorithme TDLTP.

Regardons maintenant ces résultats de plus près. On peut premièrement constater que pour les trois algorithmes, les temps séquentiels sont semblables aux temps parallèles sur un processeur pour les réseaux de Winnipeg et d'Ottawa. Par contre, pour le réseau de la ville de Montréal, qui est nettement plus gros tant au niveau du nombre de nœuds qu'au niveau du nombre de liens ou de centroïdes, l'implantation parallèle sur un processeur fournit de meilleurs résultats que l'implantation séquentielle! Ceci est bien évidemment anormal, puisque la version parallèle devrait être ralentie par certains mécanismes de synchronisation et de communication. Malgré de nombreuses heures de recherche, ces résultats demeurent toujours inexpliqués. L'explication la plus probable concerne les effets dûs à la mémoire cache. Les implantations séquentielle et parallèle n'utilisent possiblement pas de la même façon la mémoire cache. **Le manque d'outils nous permettant de savoir exactement ce qui se passe en mémoire cache nous empêche de vérifier cette supposition.** Nous avons donc utilisé les résultats parallèles sur un processeur au lieu des résultats séquentiels dans le calcul des charges relatives.

Si on regarde bien les graphiques portant sur l'accélération, on peut voir que, en général, l'implantation parallèle offre une accélération quasi-linéaire jusqu'à 16 threads. Par la suite, il y a certaines différences, selon l'algorithme et selon le réseau (ville et nombre de périodes de temps). Contrairement à ce que l'on espérait, on ne réussit pas à obtenir de très bonnes accélérations lorsque 60 threads sont utilisés. Bien sûr, l'accélération augmente avec le nombre de threads (sauf en de rares exceptions, où l'accélération va même jusqu'à diminuer!), ce qui continue de prouver l'utilité du calcul parallèle comme façon d'améliorer les performances de nos applications, mais les gains en temps sont moins importants que ce que l'on prévoyait. Probablement qu'avec un outil de calcul par thread ainsi que les matrices pour conserver des résultats intermédiaires, on utilise un espace mémoire suffisamment grand pour que les accès mémoire ne se fassent pas souvent dans la mémoire cache lorsque 60 threads sont utilisés. Ceci expliquerait que le rapport accélération/nb.threads semble diminuer lorsque le nombre de threads devient plus grand que 16. **Encore une fois, le manque d'outils adéquats pour vérifier l'utilisation de la mémoire cache nous empêche de valider cette supposition.**

On peut aussi remarquer que l'algorithme TDLTP obtient, en général, de meilleures accélérations que les deux autres, ceci étant dû principalement à l'amélioration des performances de l'algorithme. On ne s'attardera pas plus longtemps à des comparaisons entre les performances des trois algorithmes puisque, tel que mentionné précédemment, certaines modifications ont eu pour effet d'améliorer grandement les temps d'exécution de l'algorithme TDLTP et d'autres ont provoqué un ralentissement de l'exécution de l'algorithme DOT.[22]

Du côté des graphiques de charge relative, on peut facilement remarquer un manque de stabilité. Bien que dans plusieurs cas, les courbes observées nous permettent de constater une certaine décroissance (ou à tout le moins, une non-croissance), dans d'autres cas, les courbes ne présentent aucune caractéristique évidente. Prenons, par exemple, la figure 5.20. Lorsqu'on utilise 32 threads et plus, les courbes semblent se stabiliser un peu, mais

[22] Voir la page 65 pour les détails des modifications effectuées. Ces modifications sont les mêmes pour les cas séquentiel et parallèle.

avec la forme de ces courbes, pour 2, 4, 8 ou 16 threads, on ne peut supposer que la courbe demeurera stable avec plus de 60 threads. Aussi, sur certains graphiques, on remarque des valeurs de charge relative négatives. Ceci est plutôt anormal. Une raison à laquelle on peut penser est que les temps parallèles sur un processeur sont trop lents, si bien que l'accélération observée sur 4 ou 8 processeurs, par exemple, est anormalement grande dans certains cas. La principale conclusion que l'on doit tirer de ces graphiques est que la mesure de charge relative est trop instable lorsque l'on utilise plus de 16 threads pour se permettre de l'utiliser dans le but de faire des prédictions sur les comportements futurs de nos applications. D'ailleurs, si on utilise la valeur de charge relative observée pour 16 threads pour prédire les performances sur 60 processeurs (ceci suppose que la charge relative serait constante même si on passe de 16 à 60 threads), on remarque que les prédictions sont très loin de la réalité. Nous présentons au tableau 5.4 l'estimation de l'accélération potentielle en utilisant la relation 5.3 vue à la section 5.1.3.

		Algorithme utilisé		
		Chrono-SPT	DOT	TDLTP
Réseau utilisé et nombre de périodes de temps	Win/30	32.00308	28.72382	24.87375
	Win/60	32.96024	30.09343	26.96775
	Win/90	27.76124	15.85763	24.68142
	Win/120	35.61708	51.35337	54.73614
	Ott/30	39.34317	8.771799	26.37284
	Ott/60	51.46507	30.55142	33.875
	Ott/90	41.99017	55.96181	34.23161
	Ott/120	39.10505	29.28432	33.553
	Mtl/30	45.5313	11.05006	27.05333
	Mtl/60	39.95597	23.61042	54.05327
	Mtl/90	48.50492	26.41619	46.98438
	Mtl/120	19.71367	38.62813	62.99224

Tableau 5.IV: Estimation de l'accélération potentielle sur 60 processeurs.

Bien évidemment, puisque les courbes de la charge relative ne sont pas suffisamment stables, une telle estimation s'avère complètement fausse. Peut-être qu'en utilisant les 5 ou 10 dernières valeurs de la charge relative, on réussirait à obtenir une estimation correcte.

CONCLUSION

Dans ce mémoire, nous nous sommes intéressés au calcul des plus courts chemins statiques et temporels. L'objectif principal de ce mémoire était de développer des outils de calcul des plus courts chemins temporels, puis de réaliser une implantation parallèle de ces outils, utiles dans un contexte de planification des transports.

Tout d'abord, une revue de la littérature sur le calcul des plus courts chemins a été réalisée, en portant une attention particulière au calcul des plus courts chemins temporels. Cette synthèse a permis de bien comprendre les différentes variantes du problème et aussi de mieux voir l'intérêt du calcul des plus courts chemins temporels. Sans être exhaustive, cette revue de littérature couvre tout de même une période allant de 1958 à 2002.

Les implantations séquentielles et parallèles ont été réalisées à l'aide du langage de programmation C++, en utilisant le format de fichier NFF pour représenter les réseaux. Nous nous sommes attardés à l'implantation de trois algorithmes proposés dans la littérature, soit celui de Pallottino et Scutellà [47] (Chrono-SPT), celui de Chabini [10] (DOT) et celui de Ziliaskopoulos et Mahmassani [60] (TDLTP). Puisque nous avons réussi à améliorer les performances de l'algorithme TDLTP et que par souci de stabilité, l'algorithme DOT a été ralenti, nous ne nous sommes pas attardés à les comparer entre eux. De toutes façons, certains travaux ont déjà été faits à ce sujet (Tremblay [56], Chabini [14], entre autres).

L'implantation parallèle a été réalisée en utilisant la programmation *multithreads* pour bien exploiter les caractéristiques de la machine à mémoire partagée mise à notre disposition (SUNW Enterprise 10 000). Les résultats obtenus montrent que, en général, lorsque plus de 16 threads sont utilisés, le rapport accélération/nb.threads diminue. Pour moins de 16 threads, les gains en accélération sont pourtant quasi-linéaires. Une des raisons possibles à ce comportement est que plus il y a de threads utilisés, plus l'espace nécessaire devient grand et les accès mémoire se font en bonne partie en-dehors de la mémoire cache. Les accélérations obtenues, bien que moins importantes que celles

espérées, permettent tout de même d'obtenir de meilleurs temps d'exécution, confirmant l'intérêt pour le calcul parallèle lorsque vient le temps de développer des applications performantes. De plus, nous avons vu que la mesure de charge relative est relativement instable, si bien qu'elle ne peut pas vraiment être utilisée pour établir des prévisions fiables pour le futur.

Avec de plus en plus d'applications dans le domaine de la planification en transports qui nécessitent des solutions en temps réel, le calcul parallèle utilisant la programmation *multithreads* est appelé à être de plus en plus utilisé. Même si cela fait déjà plusieurs années que les machines multiprocesseurs à mémoire partagée existent, elles seront perfectionnées, tout comme les techniques de calcul parallèle, si bien que nous réussirons à traiter, de façon plus rapide, des problèmes toujours plus gros.

BIBLIOGRAPHIE

Note : Certaines références ne sont pas citées dans le texte de ce mémoire, mais nous jugeons qu'elles ont été suffisamment utiles à la réalisation de ce mémoire pour apparaître dans cette bibliographie. Certains ouvrages ont aidé à bien comprendre le calcul parallèle et d'autres ont permis de traiter certains problèmes de programmation.

[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] Bellman, R., On a Routing Problem, *Quartely Applied Mathematics*, Vol. 16, pp. 87-90, 1958.

[5] Berg, D.J., Lewis, B., Threads Primer: A guide to Multithreaded Programming, SunSoft Press, Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[6] Bertsekas, D.P., Tsitsiklis, J.N., Parallel and Distributed Computation, Numerical Methods, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[7] Bertsekas, D.P., A Simple and Fast Label-Algorithm for Shortest Paths, *Networks*, Vol. 23, pp. 703-709, 1993.

[8] Chabini, I., Des implantations parallèles de l'algorithme d'approximation linéaire pour la résolution du problème d'affectation du traffic, Publication No. 382, Centre de recherche sur les transports, Université de Montréal, 1992.

[9] 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.

[10] Chabini, I., A new Algorithm for Shortest Paths in Discrete Dynamic Network, 8th IFAC Symposium on Transportation Systems, Chinia, Greece, 1997.

[11] 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.

[12] Chabini, I., Gendron, B., Parallel Performance Measures Revesited, High Performance Computing Symposium '95, pp. 381-392, Canada's Ninth Annual High Performance Computing Symposium and Exhibition, Montréal, Canada, July 10-12, 1995.

[13] Chabini, I., Discrete Dynamic Shortest Path Problems in Transportation Applications, *Transportation Research Record* 1645, pp. 170-175, 1998.

[14] Chabini, I., Algorithms and high performance computing for dynamic shortest paths and analytical dynamic traffic assignment models, Technical Report, Operations Research Center, M.I.T., 2000.

[15] Chabini, I., Ganugapati, S., Parallel Algorithms for Dynamic Shortest Path Problems, *International Transactions in Operational Research*, Vol. 9, No. 3, pp. 279-302, 2002.

[16] Cherkassky, B.V., Goldberg, A.V. et Radzik, T., Shortest Paths Algorithms: Theory and Experimental Evaluation, *Mathematical Programming*, Vol. 73, pp. 129-174, 1996.

[17] 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.

[18] Cormen, T.H., Leiserson, C.E. et Rivest R.L., Introduction to Algorithms, M.I.T. Press, McGraw-Hill Book Company, 1990.

[19] Delannoy, C, Programmer en langage C++, Eyrolles, 5e édition mise à jour, 2000.

[20] Denardo, E.V., Fox, B.L., Shortest-Route Methods: 1.Reaching, Pruning and Buckets, *Operations Research*, Vol. 27, pp. 161-186, 1979.

[21] Deo, N. et Pang, C.-Y., Shortest Path Algorithms : Taxonomy and Annotation, *Networks*, Vol. 14, pp. 275-323, 1984.

[22] Dial, R.B., Algorithm 360 : Shortest Path Forest with Topological Ordering, *Communications of the* ACM, Vol. 12, pp. 632-633, 1969.

[23] Dijkstra, E.W., A Note on Two Problems in Connexion With Graphs, *Numerische Mathematik*, Vol. 1, pp. 269-271, 1959.

[24] Dreyfus, S.E., An Appraisal of Some Shortest-Path Algorithms, *Operations Research*, Vol. 17, pp. 395-412, 1969.

[25] 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.

[26] Florian, M., Chabini, I. et Le Saux, E., Parallel and Distributed Computation of Shortest Routes and Network Equilibrium Models, 8th IFAC Symposium onTransportation Systems, Chania, Greece, 1997.

[27] Ford, L.R., Network Flow Theory, Rand Corporation Report No., 1956.

[28] Gallo, G. et Pallottino, S., Shortest Path Methods : a Unifying Approach, *Mathematical Programming Study*, Vol. 26, pp. 38-64, 1986.

[29] Gallo, G. et Pallottino, S., Shortest Path Algorithms, *Annals of Operations Research*, Vol. 13, pp. 3-79, 1988.

[30] Gelenbe, E., Multiprocessor Performance, John Wiley & Sons, 1989.

[31] Glover, F., Klingman, D., Philips, N.V. et Schneider, R.F., New Polynomial Shortest Path Algorithms and their Computational Attributes, *Management Science*, Vol. 31, pp. 1106-1128, 1985.

[32] 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.

[33] Gottlieb, A. et Almasi, G.S., Highly Parallel Computing, The Benjamin/Cummings Publishing Company, 2e édition, 1994.

[34] Heuring, V.P. et Jordan, H.F., Computer Systems Design and Architecture, Addison-Wesley, 1997.

[35] Horn, M.E.T., Efficient Modeling of Travel in Networks with Time-Varying Link Speeds, CSIRO Mathematical and Information Sciences Technical Report CMIS 99/97.

[36] INRO Consultants Inc., Emme/2 User's Manual Software Release 8.0, Montréal, Québec, Canada, juin 1996.

[37] Johnson, D.B., A Note on Dijkstra's Shortest Path Algorithm, *Journal of the ACM*, Vol. 20, pp. 385-388, 1973.

[38] Johnson, E.L., On Shortest Path and Sorting, *In Proceedings of the 25th ACM Annual Conference*, pp. 510-517, 1972.

[39] 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.

[40] Kershenbaum, A., A Note on Finding Shortest Path Trees, *Networks*, Vol. 11, pp. 399-400, 1981.

[41] 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.

[42] 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.

[43] 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.

[44] Orda, A. et Rom, R., Minimum-Weight Paths in Time-Dependent Networks, *Networks*, Vol. 21, pp. 295-319, 1991.

[45] 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 rechercher sur les transports, Université de Montréal, 1979.

[46] Pallottino, S., Shortest Path Methods : Complexity, Interrelations and New Propositions, *Networks*, Vol. 14, pp. 257-267, 1984.

[47] Pallottino, S. et Scutellà, M.G., Shortest Path Algorithms in Transportation Models : Classical and Innovative Aspects, in (P. Marcotte et S. Nguyen (Eds)) *Equilibrium and Advanced Transportation Modelling*, Kluwer, pp. 245-281, 1998.

[48] Pallottino, S., Scutellà, M.G., Orlin, J.B. et Ahuja, R.K., Dynamic Shortest Paths: Minimizing Travel Times and Costs, Technical Report TR-01-23, Dipartimento di informatica, Università di Pisa, 2001.

[49] Pape, U., Implementation and Efficiency of Moore-Algorithms for Shortest Route Problem, *Mathematical Programming*, Vol. 7, pp. 212-222, 1974.

[50] Patterson, D.A. et Hennessy, J.L., Computer Organization & Design : The Hardware/Software Interface, Morgan Kaufmann, San Francisco, California, 1994.

[51] Shaffer, C.A., A Practical Introduction to Data Structures and Algorithm Analysis, Prentice Hall, 1998.

[52] Solaris, Multithreaded Programming Guide, Sun Microsystems, 1997.

[53] Sun Microsystems, SunTune: Application Performance Optimization on Sun Systems, Sun Microsystems, 2001.

[54] Standish, T.A., Data Structures, Algorithms, and Software Principles, Addison-Wesley, 1994.

[55] Stroustrup, B., The C++ Programing Language, Special Edition, Addison-Wesley, 2000.

[56] Tremblay, N., La calcul des plus courts chemins statiques et temporels: synthèse, implantations séquentielles et parallèles, mémoire d'étudiant, Publication CRT-98-30, Centre de recherche sur les transports, Université de Montréal, 1998.

[57] Xavier, C. et Iyengas, S.S., Introduction to Parallel Algorithms, John Wiley & Sons, 1998.

[58] Zenios, S.A., Parallel Numerical Optimization : Current State and an Annoted Bibliography, *ORSA Journal on Computing*, Vol. 1, pp. 20-43, 1989.

[59] 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.

[60] 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.

[61] Ziliaskopoulos, A.K., Mahmassani, H.S. et 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.

[62] Ziliazkopoulos, A.K. et Kotzinos, D., Design and Implementation of Massively Parallel Time-Dependent Shortest Path Algorithms, *Computer-Aided Civil and Infrastructure Engineering*, Vol. 16, pp. 337-346, 2001.

Ajout de dernière minute:

[63] Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek et Vaidy Sunderam, PVM : Parallel Virtual Machine, a User's Guide and Tutorial for Networked Parallel Computing, The M.I.T. Press, Cambridge, Massachusetts, 1994.

ANNEXE I

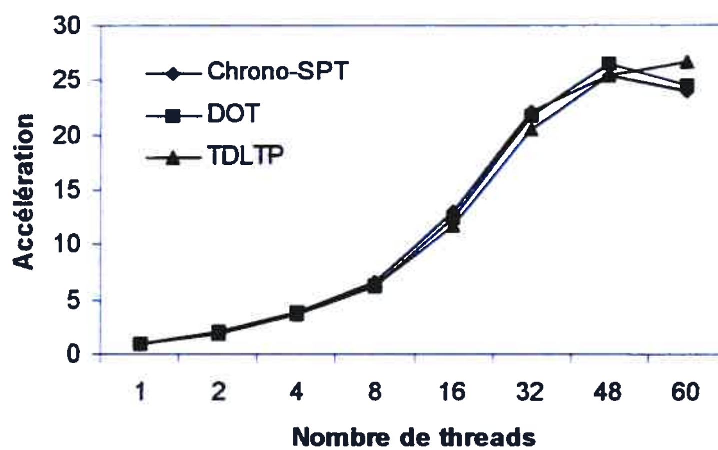


Figure 5.3 : Accélération – Winnipeg – 30 périodes de temps.

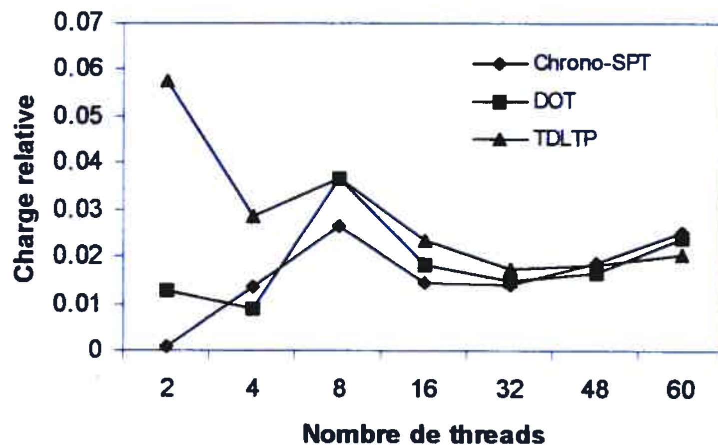


Figure 5.4 : Charge relative – Winnipeg – 30 périodes de temps.

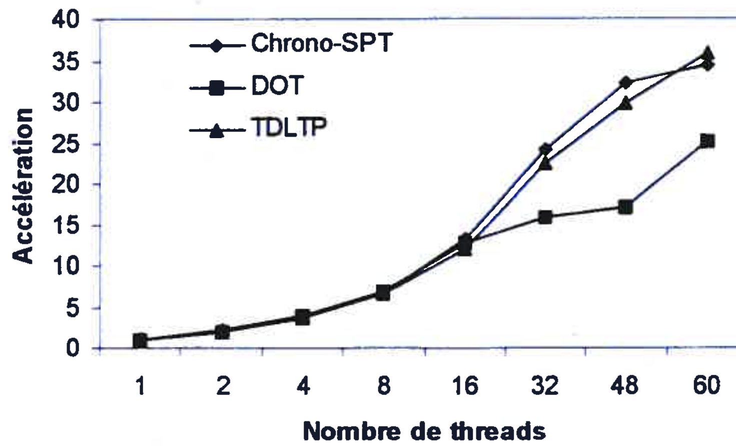


Figure 5.5: Accélération – Winnipeg – 60 périodes de temps.

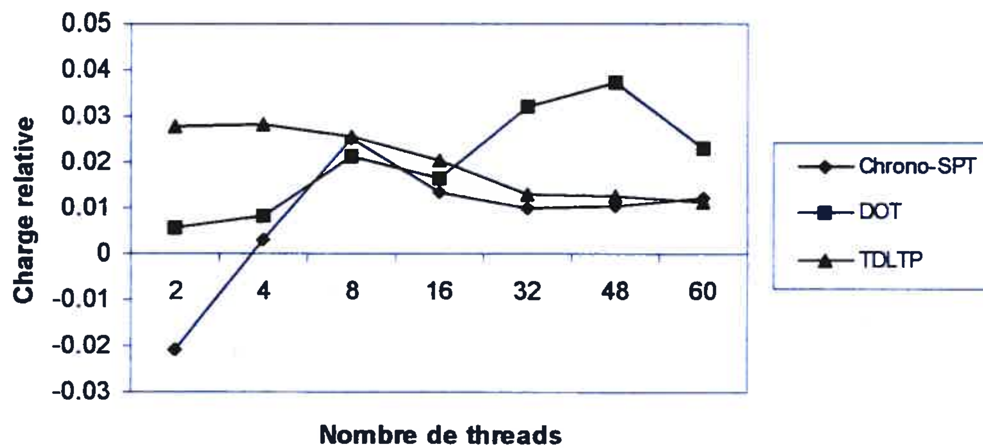


Figure 5.6: Charge relative – Winnipeg – 60 périodes de temps.

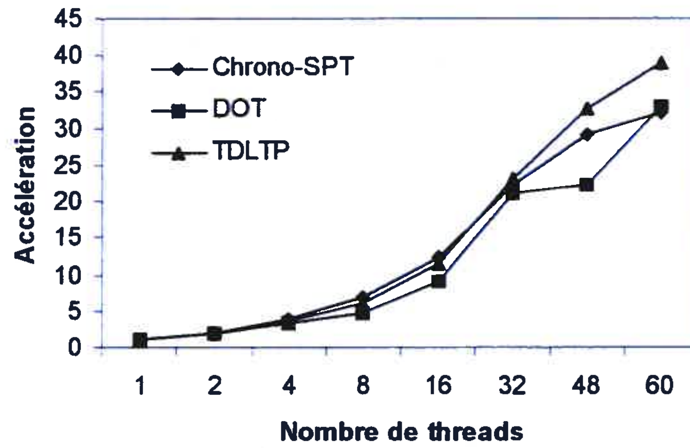


Figure 5.7: Accélération – Winnipeg – 90 périodes de temps.

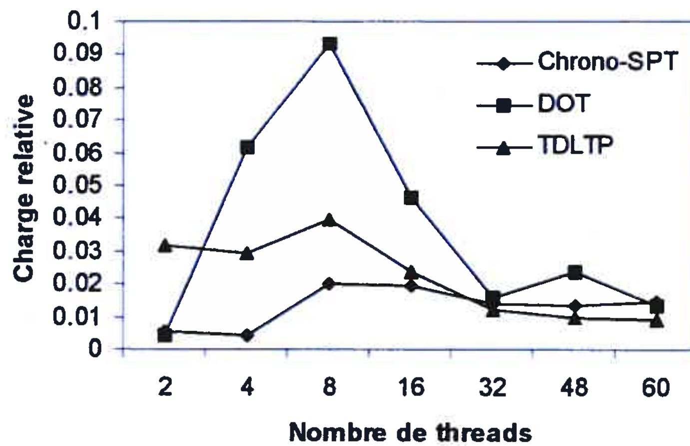


Figure 5.8: Charge relative – Winnipeg – 90 périodes de temps.

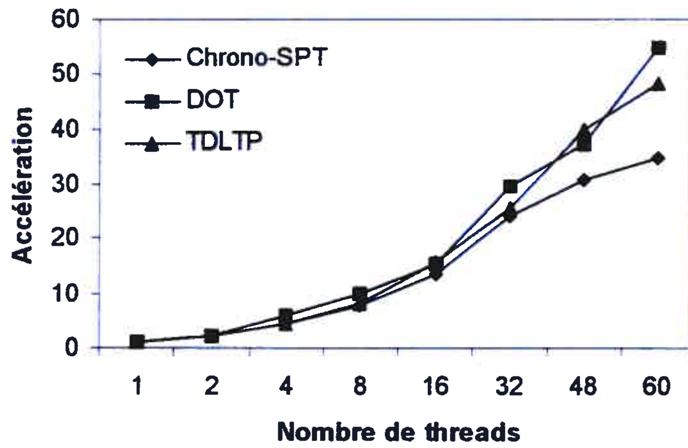


Figure 5.9: Accélération – Winnipeg – 120 périodes de temps.

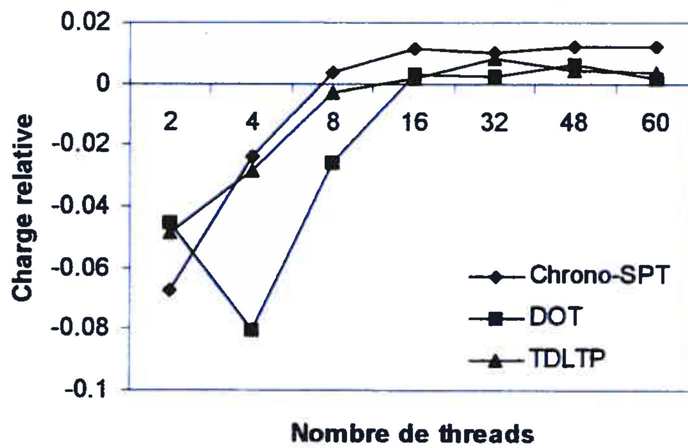


Figure 5.10: Charge relative – Winnipeg – 120 périodes de temps.

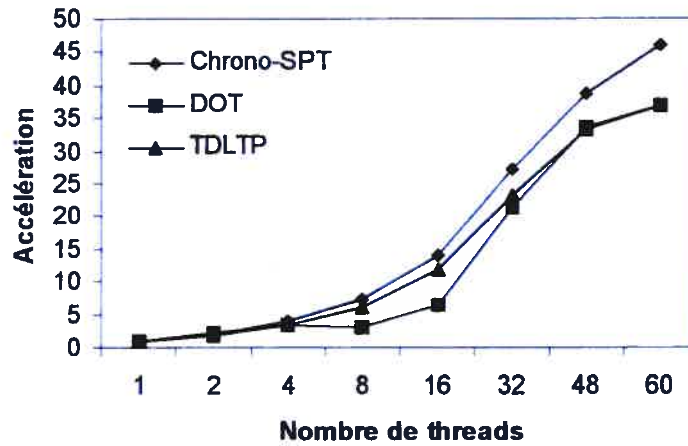


Figure 5.11: Accélération – Ottawa – 30 périodes de temps.

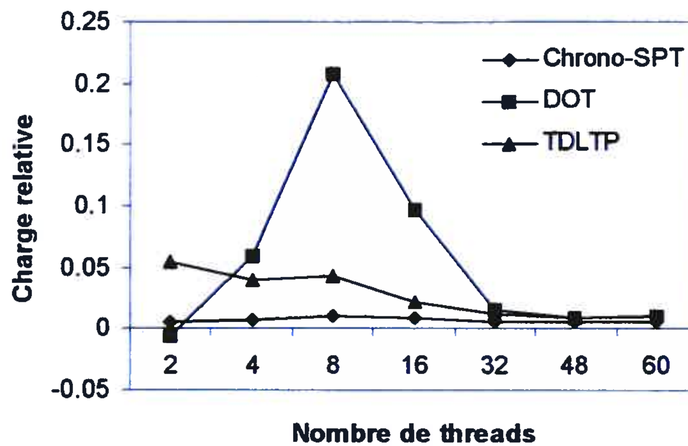


Figure 5.12: Charge relative – Ottawa – 30 périodes de temps.

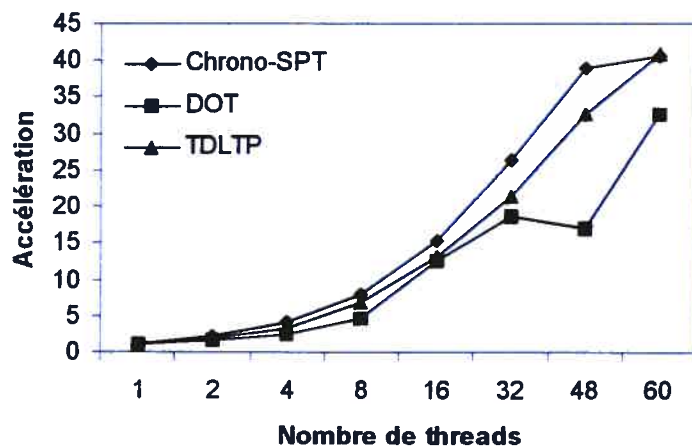


Figure 5.13: Accélération – Ottawa – 60 périodes de temps.

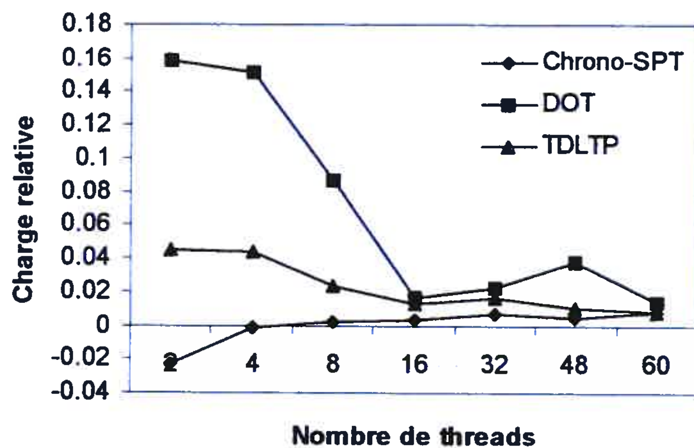


Figure 5.14: Charge relative – Ottawa – 60 périodes de temps.

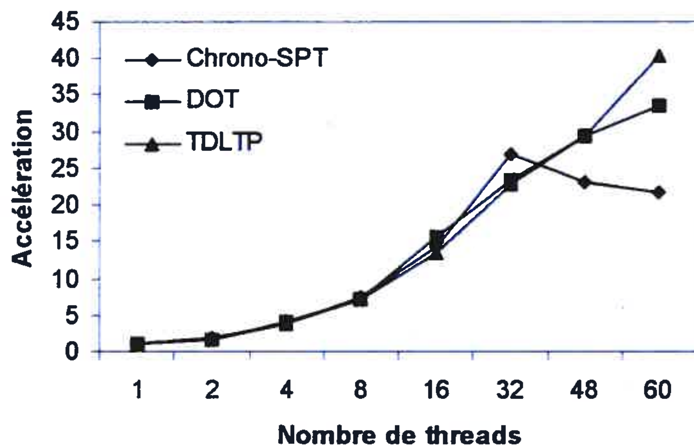


Figure 5.15: Accélération – Ottawa – 90 périodes de temps.

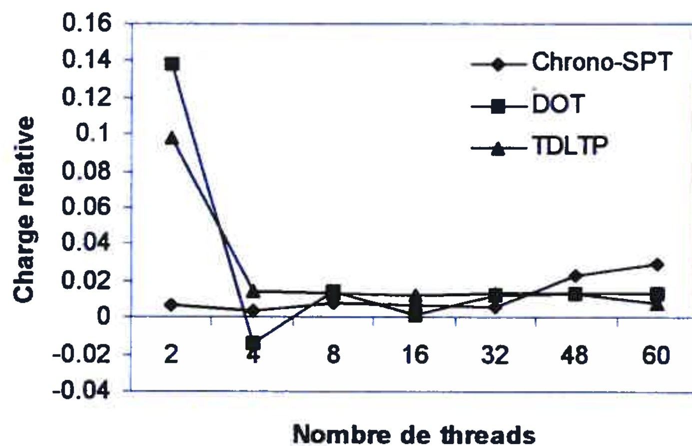


Figure 5.16: Charge relative – Ottawa – 90 périodes de temps.

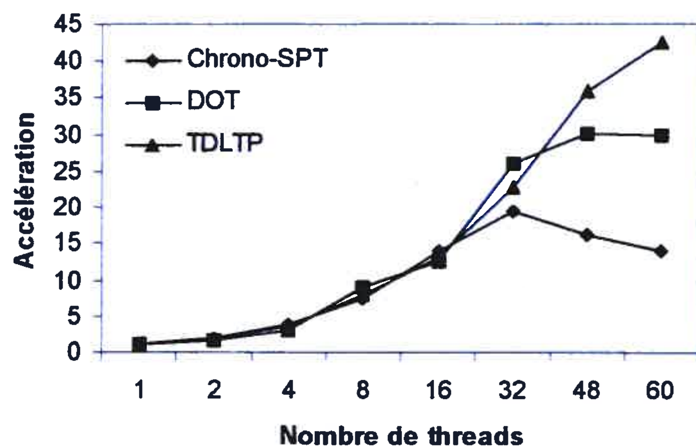


Figure 5.17: Accélération – Ottawa – 120 périodes de temps.

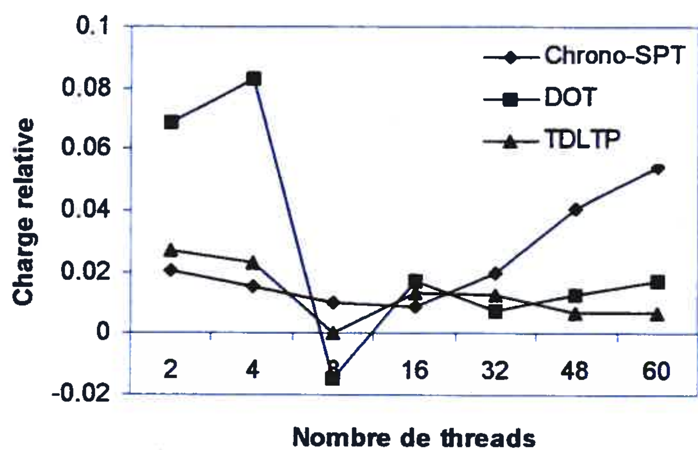


Figure 5.18: Charge relative – Ottawa – 120 périodes de temps.

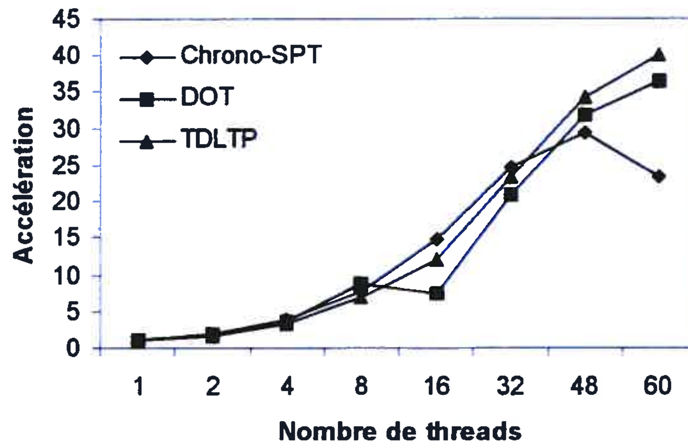


Figure 5.19: Accélération – Montréal – 30 périodes de temps.

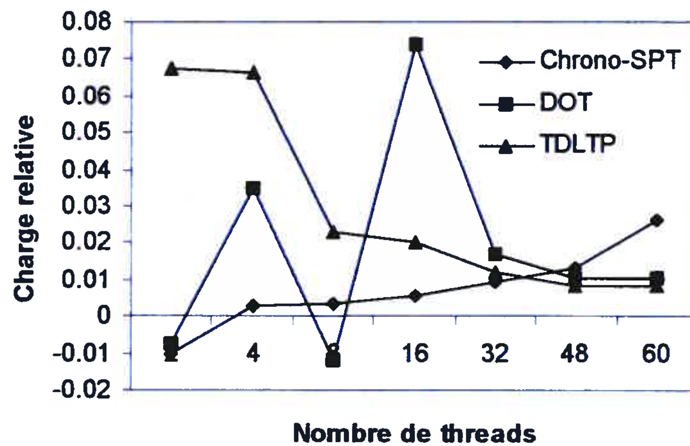


Figure 5.20: Charge relative – Montréal – 30 périodes de temps.

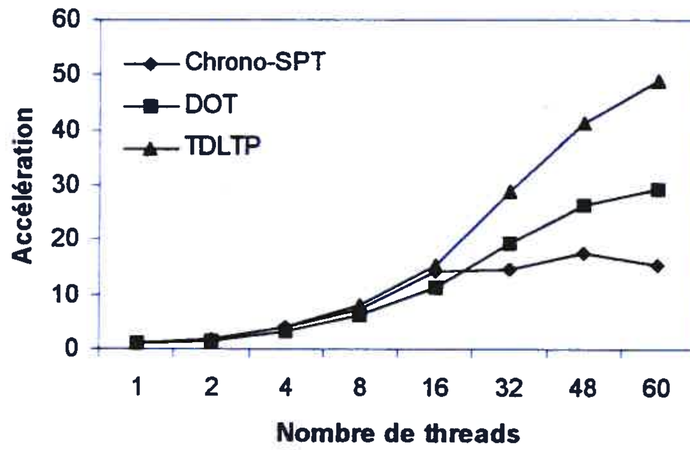


Figure 5.21: Accélération – Montréal – 60 périodes de temps.

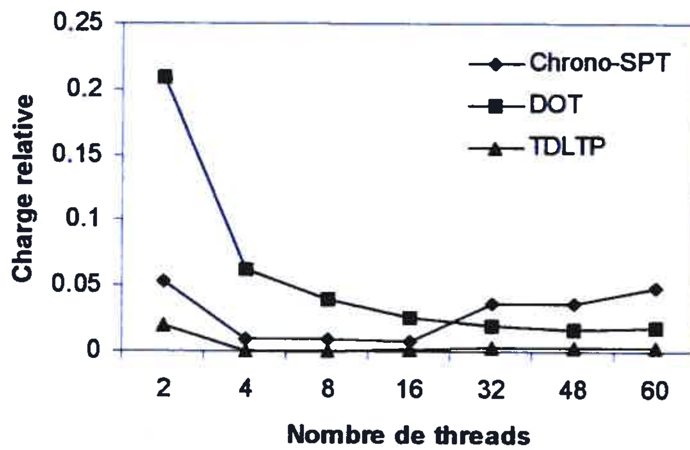


Figure 5.22: Charge relative – Montréal – 60 périodes de temps.

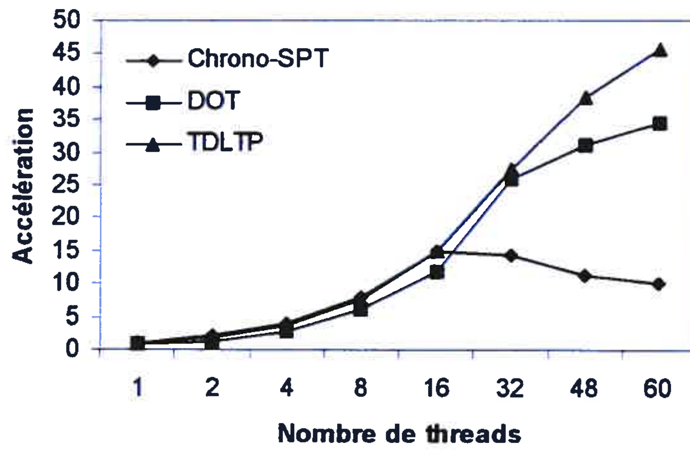


Figure 5.23: Accélération – Montréal – 90 périodes de temps.

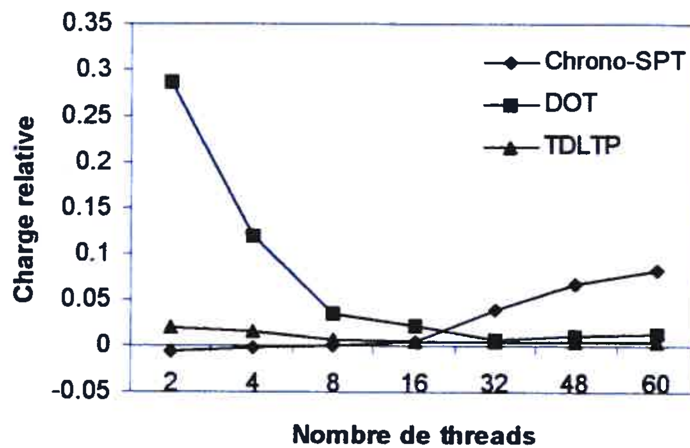


Figure 5.24: Charge relative – Montréal – 90 périodes de temps.

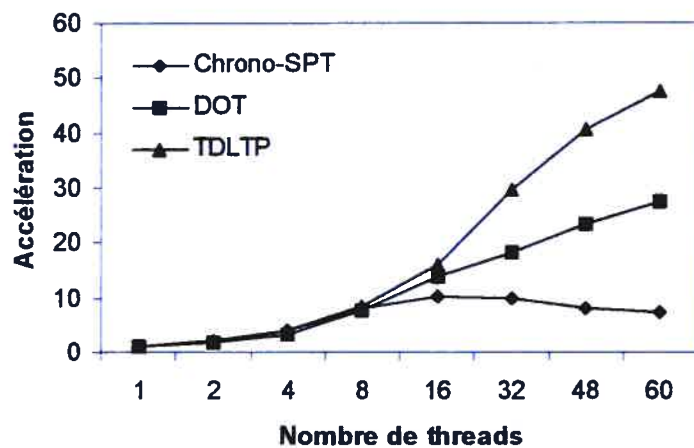


Figure 5.25: Accélération – Montréal – 120 périodes de temps.

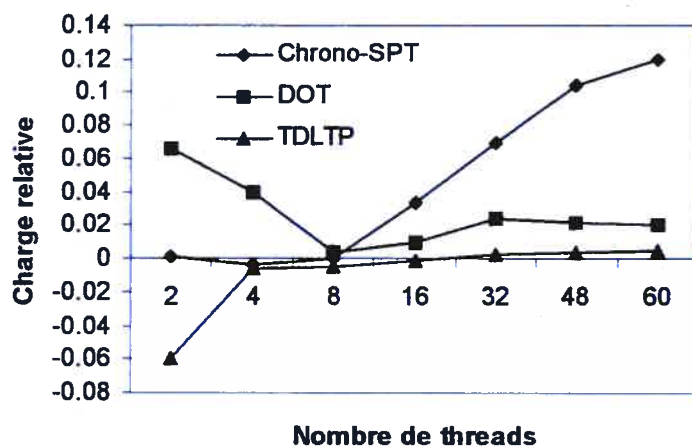


Figure 5.26: Charge relative – Montréal – 120 périodes de temps.