Université de Montréal

Etude numérique d'algorithmes d'affectation d'équilibre de réseaux :
modèles statiques à coûts symétriques avec demandes fixes dans l'espace des chemins.

par

Naïma Abbes

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
Maîtrise (M.Sc)
en informatique et recherche opérationnelle.

Novembre 2006

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Etude numérique d'algorithmes d'affectation d'équilibre de réseaux :

modèles statiques à coûts symétriques avec demandes fixes dans l'espace des chemins.

Présenté par :

Naïma Abbes

a été évalué par un Jury composé des personnes suivantes :

Michel Gendreau

Président rapporteur

Michael Florian.

Directeur de recherche

Jacques Ferland

Membre du Jury.

Mémoire accepté le :

22 décembre 2006

SOMMAIRE

La recherche opérationnelle a joué un rôle important dans le développement des modèles d'équilibre de réseau ainsi que dans l'élaboration d'algorithmes pour l'obtention de solutions pratiques.

Le but de ce projet est l'étude de méthodes d'affectation d'équilibre statique de réseaux à coûts symétriques avec demandes fixes dans l'espace des chemins. Cette affectation est calculée en tenant compte du phénomène de congestion sur le réseau. Le problème consiste à déterminer les choix des chemins des usagers se déplaçant dans un réseau de transport. Un chemin désigne un itinéaire reliant une origine et une destination.

Pour résoudre ce problème, qui est devenu un classique en recherche opérationnelle, certains se sont basés sur la formulation du problème comme un problème d'optimisation non linéaire où la fonction objectif est convexe et d'autres comme une inégalité variationnelle. Dans notre cas, nous considérons le modèle comme un problème d'optimisation convexe et non linéaire. Les algorithmes étudiés sont l'algorithme du convexe simplexe, l'algorithme d'affectation basé sur la méthode du gradient réduit et celui basé sur la méthode du gradient projeté. Ces trois algorithmes sont jumelés à une approche de décomposotion par paire O-D, ils se déroulent dans l'espace de flots sur les chemins et proposent des solutions où tous les chemins ont le même coût pour une même paire O-D, ce dernier étant le coût minimum pour cette paire O-D.

L'objectif est de comparer l'efficacité de ces variantes du point de vue rapidité de convergence, qualité des résultats obtenus et temps de calcul . Notre travail consiste donc en l'implantation de ces méthodes, la réalisation des tests et l'analyse des résultats. Les tests

ont été effectués sur les données relatives aux réseaux des villes de Winnipeg, Ottawa et Montréal afin d'évaluer les performances des implantations sur un petit, moyen et grand réseau. Les implantations sont analysées afin de vérifier certains résultats déjà connus et d'en étudier les comportements. Elles ont été effectuées avec le langage de programmation C++ pour assurer une meilleure performance aux algorithmes. Les résultats ont été comparés à ceux obtenus avec le programme EMME/2, et s'avèrent très satisfaisants.

Mots clés : Réseau routier, Affectation d'équilibre, modèle statique, espace des chemins, optimisation convexe, non linéaire, demandes fixes, coûts symétriques.

RESUME

The traffic assignment problem aims to predict the route flows and travel times that result from the way in which travellers choose routes from their origins to their destinations. There is a large variety of traffic assignment models, but in this thesis, we consider only the symmetric network equilibrium model with fixed travel demands.

This model may be formulated as a nonlinear cost network optimization model or a variational inequality. In this project, we consider the model as a nonlinear cost network optimization problem. The studied algorithms are : convex simplex, algorithms based on the reduced gradient method and the projected gradient method in the space of path flows. In these algorithms, the network equilibrium model is decomposed by O-D pair and a sequence of problems for each O-D pair is solved by storing the paths implicitly. The solution is obtained when all the used paths are of equal minimal cost for each O-D pair.

The aim of this project is to compare the performance of these algorithms from the point of view of empirical speed of convergence, the quality of the results and the computing time.Our work consists in the implementation of these algorithms, the execution of tests and the analysis of results. We have evaluated the numerical performance of the algorithms on small, average and large networks that originate from the cities of Winnipeg, Ottawa and Montreal. The results were compared to those obtained with the EMME/2 code and proved to be satisfactory.

Key Words : transportation networks, symmetric equilibrium assignment, static models, space of path flows, fixed demands, optimization formulation.

TABLE DES MATIÈRES

SOMMAIRE v

RESUME vii

TABLE DES MATIÈRES viii

LISTE DES TABLEAUX xi

LISTE DES FIGURES xii

INTRODUCTION 2

CHAPITRE 1 : FORMULATIONS DU PROBLÈME D'AFFECTATION D'ÉQUILIBRE AVEC DEMANDES FIXES 6

1.1 : Formulation mathématique 6

1.2 : Formulation d'inégalité variationnelle 8

1.3 : Formulation de complémentarité non linéaire 9

1.4 : Formulation de point fixe 9

1.5 : Formulation d'optimisation 10

1.6 : Existence et unicité de la solution 11

CHAPITRE 2 : REVUE DE LITTÉRATURE 12

2.1 : Adaptation de l'algorithme d'approximation linéaire de Frank-Wolfe au problème d'affectation d'équilibre 12

2.2 : Algorithmes de décomposition simpliciale 15

CHAPITRE 3 : ALGORITHMES D'ÉQUILIBRE DE CHEMINS ÉTUDIÉS . 18

3.1 : Algorithme du simplexe convexe 19

3.2 : Adaptation de l'algorithme du gradient réduit au problème d'affectation d'équilibre 21

3.3 : Adaptation de l'algorithme du gradient projeté au problème d'affectation d'équilibre 25

CHAPITRE 4 : IMPLANTATION DES ALGORITHMES ÉTUDIÉS 30

4.1 : Les données du réseau 30

4.2 : Les structures de données reliées aux chemins du réseau. 32

4.3 : L'algorithme des plus courts chemins 35

4.4 : La recherche unidimensionnelle 38

4.5 : Le développement des implantations 40

CHAPITRE 5 : PRÉSENTATION ET ANALYSE DES RÉSULTATS 42

5.1 : **Réseau de Winnipeg** 44

5.2 : **Réseau d'Ottawa** 49

5.3 : **Réseau de Montréal** 54

CONCLUSION 58

BIBLIOGRAPHIE **59**

ANNEXES **66**

LISTE DES TABLEAUX

Tableau 4.1 : Performance de l'algorithme de Spiess 38

Tableau 5.1 : Caractéristiques des réseaux 42

Tableau 5.2 : Valeurs de la fonction économique-Ville de Winnipeg 45

Tableau 5.3 : Ecarts relatifs -ville de Winnipeg- 46

Tableau 5.4 : Valeurs de la fonction économique avec différentes valeurs de ϵ -ville de Winnipeg- 47

Tableau 5.5 : Valeurs de la fonction économique-Ville d'Ottawa 50

Tableau 5.6 : Ecarts relatifs -ville d'Ottawa- 50

Tableau 5.7 : Valeurs de la fonction économique-Ville de Montréal 54

Tableau 5.8 : Ecarts relatifs -ville de Montréal- 55

LISTE DES FIGURES

Figure 4.1 : Réseau de Braess 31

Figure 4.2 : Représentation de la structure des chemins 34

Figure 4.3 : Hiérarchie des classes développées 41

Figure 5.1 : graphe F.Eco/Itération -Réseau de Winnipeg 48

Figure 5.2 : graphe F.Eco/Itération -Réseau d'Ottawa 52

Figure 5.3 : graphe F.Eco/Itération -Réseau de Montréal 57

A ma mère.

REMERCIEMENTS

Je tiens tout d'abord à remercier mon directeur de recherche, M. Michael Florian, pour la confiance qu'il m'a accordée, sa disponibilité ainsi que son soutien moral et financier tout au long de ce travail.

Je remercie également les personnes qui travaillent au centre de recherche sur les transports (CRT) pour leur aide et les moyens mis à notre disposition. Je les remercie aussi de m'avoir permis d'accéder aux données relatives aux réseaux des villes de Winnipeg, Ottawa et Montréal.

Je tiens à exprimer ma gratitude à mon mari qui m'a encouragée tout au long de mes études.

Merci aussi à toutes les personnes qui ont contribué de près ou de loin à la réalisation de ce travail.

INTRODUCTION

Les problèmes de planification du transport ont intéressé plusieurs auteurs depuis les dernières décennies. Parmi ces problèmes, on trouve celui de l'affectation du trafic routier suivant un modèle d'équilibre. Ces modèles permettent d'établir des prévisions sur les structures de réseau et d'effectuer des évaluations économiques et des évaluations sur la qualité de service rendu aux usagers et ainsi appréhender les conséquences potentielles de l'évolution de divers paramètres à court et moyen termes. Ils sont utilisés entre autres dans de nombreuses applications telles que les réseaux électriques, les réseaux de conduite d'eau et les problèmes d'équilibre de prix spatial.

Dans ce travail, nous considérons le problème d'affectation statique à demande fixe sur un réseau routier à coûts symétriques caractérisé par une matrice de demandes origine-destination. Il consiste à déterminer les choix des chemins des usagers ayant à se déplacer dans un réseau de transport en tenant compte du phénomène de congestion.

Un modèle d'affectation d'équilibre est dit statique, si nous analysons le système avec une demande moyenne durant une période de temps limitée. Si, par contre, nous tenons compte de la variation du niveau de la demande et des flots des arcs tout au long d'une période de temps, le modèle est dit dynamique. Si, en plus, nous considérons que la demande entre chaque paire origine-destination est indépendante du flot, alors le modèle est à demande fixe, dans le cas contraire, nous obtenons un modèle à demande variable. Dans la pratique, les modèles statiques sont plus utiles et plus performants que les modèles dynamiques.

Le comportement des usagers en fonction des coûts des itinéraires repose sur deux principes de Wardrop (1952). Le principe dit descriptif et le principe normatif. Le premier

principe correspond à la situation dans laquelle chaque usager tend à minimiser son coût de transport sans se préoccuper des autres usagers (comportement égoïste), tous les chemins utilisés entre un noeud origine et un noeud destination ont le même coût de parcours alors que les chemins non utilisés ont des coûts plus élevés. Le second principe, quant à lui cherche à minimiser le coût total du voyage (comportement coopératif). Le présent travail porte sur le problème d'équilibre caractérisé par le principe descriptif.

Nous dirons que la fonction coût est symétrique si son Jacobien est symétrique et c'est le cas quand le coût dépend uniquement du flot sur l'arc.

Compte tenu de la grande importance du problème dans la pratique d'une part et sa taille dans la réalité d'autre part, plusieurs méthodes ont été développées incluant les méthodes de linéarisation, de décomposition cyclique et des approches duales. Certaines de ces méthodes se déroulent dans l'espace des liens et d'autres dans l'espace des chemins. Avant les dernières décennies, les algorithmes se déroulant dans l'espace des chemins n'étaient particuliérement utilisés que dans des réseaux possédant un nombre de paires O-D restreint tels que les réseaux aériens et ferroviaires. Pour les réseaux routiers, les algorithmes se déroulant dans l'espace des flots sur les arcs étaient plus répandus. Les travaux effectués par Chen et Lee (1999) ont cependant démontré que les algorithmes se déroulant dans l'espace des chemins sont aussi performants compte tenu des capacités des machines actuelles du point de vue stockage et rapidité de calcul.

Parmi les méthodes de linéarisation les plus connues, nous citons la méthode de Frank-Wolfe, adaptée au problème d'affectation d'équilibre par Bruynooghe, Gibert et Sakarovitch en 1968, et plusieurs de ses variantes, les méthodes de décomposition simpliciale reliées à la région réalisable et reposant sur la théorie polyédrale. Ces dernières ont leur origine dans les travaux effectués par Holloway (1974) et Hohenbalken (1977). Bertsekas et Gafni (1982), Pang et Yu (1984) entre autres ont développé des algorithmes de décomposition simpliciale. Hearn et al (1987) ont proposé la méthode de décomposition simpliciale

restreinte qui permet de limiter le nombre de points extrêmes à sauvegarder lors de la résolution du sous problème linéaire Frank-Wolfe. Larsson et Patrikson (1992) ont étendu la méthode. Aussi, nous citons les travaux de Leventhal et al. (1973) qui combinent une approche de restriction avec une stratégie de génération de colonnes .

Quant aux méthodes de décomposition cyclique, elles sont classées selon la manière dont le problème est décomposé, soit par origine, soit par paire O-D. Les algorithmes décomposés par origine reviennent à Weintraub (1974), Dembo et Klincewicz (1981), Dembo (1987) et Dembo et Tulowitzki (1988), ils utilisent les flots sur les arcs comme variables explicites. Quant aux algorithmes décomposés par paire O-D, ils utilisent les flots sur les chemins comme variables explicites

La forme duale a été étudiée entre autres par Fukushima (1984), Carey (1985) et Hearn et Lawphongpanich (1990). Larsson et al (1997b) ont proposé une approche duale basée sur le dualité lagrangienne et la méthode du sous gradient.

Les algorithmes que nous proposons d'étudier sont décomposés par paire O-D ; l'approche de décomposition retenue est celle suggérée dans Florian (1986). Ces algorithmes sont :

- l'algorithme du simplexe convexe, dont l'origine provient des travaux effectués par Gibert en 1968 pour les fonctions coûts non linéaires et ceux effectués par Dafermos et Sparrow en 1969 sur les réseaux quadratiques. Néanmoins, la méthode fut plus répandue grâce aux recherches de Nguyen (1974).
- l'algorithme d'affectation basé sur la méthode du gradient réduit : développée initialement par Wolfe en 1967 et s'appuyant sur les fondements de la méthode du simplexe, elle s'adapte parfaitement au problème d'affectation d'équilibre.
- l'algorithme d'affectation basé sur la méthode du gradient projeté qui est issu d'une idée de Soumis (1978).

Il y a quelques années, ces algorithmes qui se déroulent dans l'espace des chemins étaient

considérés impraticables. Avec le développement important de la technologie des ordinateurs et l'augmentation de la taille des réseaux, l'implantation de ces méthodes n'est pas nécessairement valide actuellement. L'intérêt de ce projet est donc de tester ces algorithmes avec les moyens matériels actuels sur des réseaux de différentes tailles tels que les réseaux des villes de Winnipeg, Ottawa et Montréal, de les implanter avec un langage de programmation des plus performants tel que le C++, de comparer la rapidité de convergence, la qualité des résultats obtenus et de vérifier si ces algorithmes surpassent les meilleures méthodes existantes.

Ce mémoire est organisé de la façon suivante. Au premier chapitre, nous présentons des formulations du problème d'affectation d'équilibre. Au chapitre 2, et bien que le mémoire soit axé davantage sur l'aspect développement, nous présentons une revue de littérature concernant les différents algorithmes d'affectation d'équilibre de réseau existants. Les deux chapitres suivants constituent le coeur de ce mémoire, dans le premier, nous exposons les détails des algorithmes étudiés et dans le second leur implantation . L'analyse et la présentation des résultats sont décrites dans le chapitre 5 et nous terminons par une conclusion qui présente différentes avenues de recherches possibles. Nous fournissons aussi en annexe B les différents modules programmés en C++ pour le développement des algorithmes étudiés.

Chapitre 1
Formulations du problème d'affectation d'équilibre avec demandes fixes

Dans ce chapitre, nous présentons une formulation mathématique du problème d'affectation d'équilibre, suivie de plusieurs autres formulations équivalentes telles que la formulation d'inégalité variationnelle, de complémentarité non linéaire, de point fixe et d'optimisation. Ces formulations permettent d'établir les résultats d'existence et d'unicité du problème.

1.1 Formulation mathématique

La première formulation mathématique du problème d'affectation d'équilibre revient à Beckman, McGuire et Winsten en 1956. D'autres formulations ont été proposées par la suite comme celle de Rosenthal (1973) qui introduit une formulation en nombres entiers. La formulation présentée dans ce chapitre est celle proposée par Dafermos et Sparrow (1969) ; elle est définie dans l'espace des chemins et repose sur les deux principes de Wardrop (1952).

Nous considérons le réseau de transport $G = (N, A)$ où :

N est l'ensemble des noeuds du réseau représentant les centroïdes(origines et/ou destinations), les intersections dont au moins un virage est pénalisé et des noeuds réguliers où tous les mouvements sont permis sans délai.

A est l'ensemble des arcs orientés du réseau représentant les rues d'une zone urbaine.

$v_a, a \in A$ est le vecteur de flot sur l'arc a.

$s_a(v)$ est la fonction coût associée à l'arc a , où v est le vecteur $(v_a)_{a \in A}$.

g_p est la demande fixe associée à la p-ième paire O-D. Elle correspond au volume de flot voulant voyager de l'origine O_p vers la destination D_p.

h_k est le vecteur de flot sur le chemin k.

P est l'ensemble des paires O-D.

K_p est l'ensemble des chemins reliant l'origine à la destination de la p-ième paire O-D.

K est ensemble de tous les chemins du réseau, $K = \cup_{p \in P} K_p$.

Le coût du chemin $s_k (= s_k(h))$ est égal à la somme des coûts de tous les arcs que le chemin traverse :

$$s_k = \sum_{a \in A} \delta_{ak} s_a(v), \quad k \in K_p, \quad p \in P, \tag{1.1}$$

où δ_{ak} est défini comme suit :

$$\delta_{ak} = \begin{cases} 1, & \text{si } a \in k, \\ 0, & \text{sinon.} \end{cases} \tag{1.2}$$

Le flot sur les chemins satisfait les contraintes de conservation de flots et non négativité :

$$\sum_{k \in K_p} h_k = g_p, \qquad p \in P, \tag{1.3}$$

$$h_k \geq 0, \qquad k \in K. \tag{1.4}$$

Les flots sur les liens sont donnés par :

$$v_a = \sum_{p \in P} \sum_{k \in K_p} \delta_{ak} h_k. \tag{1.5}$$

En posant

$$u_p^* = min_{k \in K_p} s_k, \qquad \forall p \in P, \tag{1.6}$$

le principe d'équilibre de Wardrop s'énonce comme suit :

$$s_k^*(v) - u_p^*(v) = \begin{cases} 0, & \text{si } h_k^* > 0 \\ \geq 0, & \text{si } h_k^* = 0. \end{cases} \qquad k \in K_p, p \in P, \tag{1.7}$$

Une variable de flot ou de coût à l'équilibre est désignée en ajoutant *.

1.2 Formulation d'inégalité variationnelle

Smith (1979) identifie le modèle d'affectation d'équilibre comme un problème d'inégalité variationnelle :

$$\sum_{a \in A} s_a(v^*)(v_a - v_a^*) \geq 0, \qquad \forall v \in \Omega_v, \tag{1.8}$$

où Ω_v est le domaine réalisable défini par les contraintes (1-3)-(1-5) dans l'espace des flots sur les arcs et v^* le flot à l'équilibre.

Proposition 1 : Si la demande g_p est non négative pour tout $p \in P$ et $s_k(h) > 0$ pour tout $k \in K_p$, $p \in P$, alors la formulation d'inégalité variationnelle et le modèle d'équilibre de réseau sont équivalents.

Une preuve de la proposition ci-dessus est fournie dans Florian et Hearn (1996) et Patriksson (1994).

1.3 Formulation de complémentarité non linéaire

La formulation de complémentarité non linéaire du modèle d'équilibre de réseau s'écrit comme suit :

$$F_i(x)x_i = 0, \tag{1.9}$$

$$F_i(x) \geq 0, \tag{1.10}$$

$$x_i \geq 0. \tag{1.11}$$

$i = 1, 2, .., n = |K| + |P|$, où $x = (h, u)$ et F est composé de n éléments où les $|K|$ premiers sont de la forme $s_k(h) - u_p$, $k \in K_p$, $p \in P$ et les $|P|$ derniers de la forme $\sum_{k \in K_p} h_k - g_p, p \in P$.

Proposition 2 : Si la demande g_p est non négative pour tout $p \in P$ et $s_k(h) > 0$ pour tout $k \in K_p$, $p \in P$, alors la formulation de complémentarité non linéaire et le modèle d'équilibre de réseau sont équivalents.

Cette formulation revient à Aashtiani et Magnanti (1981) où une preuve détaillée de la proposition ci-dessus est donnée.

1.4 Formulation de point fixe

Un problème de point fixe permet de trouver une solution x^* tel que :

$$x^* = H(x^*) = proj_{R^n_+}(x - F(x)) \tag{1.12}$$

où x et F sont définis comme précédemment.

La première formulation en point fixe revient à Sender et Netter (1970) pour les modèles à demandes fixes et variables et les réseaux multimodals.

Proposition 3 : x^* résoud le problème de point fixe si et seulement si x^* résoud le problème d'inégalité variationnelle.

Une preuve de la proposition est exprimée dans Florian et Hearn (1996).

1.5 Formulation d'optimisation

Sous l'hypothèse que la fonction coût est séparable par arc, c'est-à-dire que :

$$s_a(v) = s_a(v_a), \qquad \forall a \in A, \tag{1.13}$$

et que son jacobien est symétrique et défini positif, alors la forme variationnelle (1-8) est identique à un problème d'optimisation convexe. Une preuve d'équivalence est donnée dans Kinderlehrer et Stampacchia (1980).

Le problème devient :

$$\min F = \sum_{a \in A} \int_0^{v_a} s_a(t)dt \tag{1.14}$$

s.à

$$\sum_{k \in K_p} h_k = g_p, \quad p \in P, \tag{1.15}$$

$$h_k \geq 0, \qquad k \in K, \tag{1.16}$$

$$v_a = \sum_{p \in P} \sum_{k \in K_p} \delta_{ak} h_k, \qquad a \in A. \tag{1.17}$$

1.6 Existence et unicité de la solution

Ces formulations équivalentes au problème d'équilibre de réseau ont permis d'une part d'obtenir des conditions d'existence et d'unicité des solutions d'équilibre et d'autre part de développer les premières méthodes de résolution du problème.

Les propositions d'existence et d'unicité suivantes sont énoncées et démontrées dans Florian et Hearn (1996) et Patriksson (1994).

Nous prenons d'une manière générale le cas où la demande g_p n'est pas fixe mais donnée par une fonction $g_p(u)$, où u est le vecteur des coûts minimums sur l'ensemble des paires O-D.

$g_p = g_p(u), \quad \forall p \in P,$

$u = (u_p)_{p \in P}$.

Proposition 4 :

Si les fonctions de coûts s_k sont positives et continues et les fonctions de demandes g_p sont non négatives, continues et bornées pour chaque $k \in K$ et chaque $p \in P$, alors le problème d'affectation d'équilibre du réseau possède une solution.

Proposition 5 :

Si s_k et $-g_p$ sont strictement monotones, alors les flots sur les arcs v^* et les coûts minimums u^* des chemins sont uniques.

Chapitre 2
Revue de littérature

La formulation du problème d'affectation d'équilibre sous forme d'un problème d'optimisation convexe a permis de développer une grande variété d'algorithmes pour la résolution du problème. Dans ce chapitre, nous présentons une sélection des principales méthodes traitant du sujet. Nous décrirons les algorithmes de linéarisation suivants : Frank-Wolfe et décomposition simpliciale basée sur les points extrêmes du domaine réalisable. Quant aux algorithmes basés sur la décomposition cyclique par paire O-D, ils seront présentés dans le chapitre 3.

2.1 Adaptation de l'algorithme d'approximation linéaire de Frank-Wolfe au problème d'affectation d'équilibre

L'un des algorithmes les plus simples pour la minimisation d'une fonction convexe sous contraintes linéaires est la méthode d'approximation linéaire.
Soit le problème suivant :

$$min\ f(x) \tag{2.1}$$

s.à

$$Ax = b, \tag{2.2}$$

$$x \geq 0. \tag{2.3}$$

l'algorithme prend, d'une manière générale, la forme suivante :

- **Etape 0** : Trouver une solution intiale réalisable x^1, $l = 1$.
- **Etape 1** : Trouver une direction de descente d^l à partir d'un point extrêmal y^l. Ce point est identifié en résolvant le sous problème linéaire suivant :

$$min \bigtriangledown f(x^l)y \tag{2.4}$$

$$Ay = b, \tag{2.5}$$

$$y \geq 0. \tag{2.6}$$

  $d^l = y^l - x^l$.
- **Etape 2** : Si $|\frac{\bigtriangledown f(x^l)(y^l - x^l)}{f(x^l)}| < \epsilon$ s'arrêter.
- **Etape 3** : Trouver le pas optimal $\lambda^l = argmin_{0 \leq \lambda \leq 1} f(x^l + \lambda d^l)$, $x^{l+1} = x^l + \lambda^l d^l$, mettre à jour $l = l + 1$, et retourner à l'étape 1.

Proposée par Bruynooghe, Gibert et Sakarovitch en 1969 pour la résolution du problème d'équilibre de réseau, la méthode est devenue populaire grâce aux travaux de LeBlanc et al. (1975) et Nguyen et Florian (1976). Elle est implantée dans plusieurs logiciels de planification de transport tels que les logiciels TRAFFIC et EMME/2.

La traduction de l'algorithme dans le cadre d'affectation d'équilibre de réseau concerne l'identification du point extrêmal en résolvant le sous problème linéaire suivant :

$$min \sum_{a \in A} s_a(v_a^l) y_a \tag{2.7}$$

s.à

$$\sum_{k \in K_p} y_k = g_p, \qquad \forall p \in P, \tag{2.8}$$

$$y_k \geq 0, \qquad \forall k \in K, \tag{2.9}$$

$$y_a = \sum_{p \in P} \sum_{k \in K_p} \delta_{ak} y_k, \qquad \forall a \in A. \tag{2.10}$$

$$\sum_{a\in A} s_a(v_a^l)y_a = \sum_{a\in A} s_a(v_a^l)\sum_{p\in P}\sum_{k\in K_p}\delta_{ak}y_k \quad (2.11)$$

$$= \sum_{p\in P}\sum_{k\in K_p} y_k \sum_{a\in A}\delta_{ak}s_a(v_a^l) \quad (2.12)$$

$$= \sum_{p\in P}\sum_{k\in K_p} y_k s_k, \quad (2.13)$$

où $s_k = \sum_{a\in A}\delta_{ak}s_a(v_a^l)$ est le coût du chemin k, étant donné v^l.

Le problème (2.7)-(2.10) revient à :

$$min \sum_{p\in P}\sum_{k\in K_p} s_k^l y_k \quad (2.14)$$

s.à

$$\sum_{k\in K_p} y_k = g_p, \qquad p \in P, \quad (2.15)$$

$$y_k \geq 0, \qquad k \in K. \quad (2.16)$$

La solution optimale est calculée en affectant toute la demande sur le chemin le plus court. Une telle affectation est connue sous le nom d'affectation "tout ou rien".

L'adaptation de la méthode d'approximation linéaire au problème d'affectation d'équilibre produit l'algorithme suivant :

- **Etape 0** : Trouver une solution initiale v^l; $s^l = s(v^l)$; $l = 1$.
- **Etape 1** : Calculer le vecteur de flot sur l'arc y^l en réalisant une affectation tout ou rien basée sur les coûts actuels $s(v^l)$. La direction de descente est alors $d^l = y^l - v^l$.
- **Etape 2** : Si un critère d'arrêt est satisfait, alors s'arrêter. Sinon continuer.
- **Etape 3** : Trouver $\lambda^{(l)}$ en résolvant $\sum_{a\in A} s_a(v_a^l + \lambda d^l)d^l = 0, 0 \leq \lambda \leq 1$.

- **Etape 4** : Mettre à jour les flots sur l'arc $v^{l+1} = v^l + \lambda^l d^l$ et les coûts sur l'arc $s^{l+1} = s(v^{l+1})$, $l = l + 1$, et retourner à l'étape 1.

Plusieurs critères d'arrêts peuvent être prédéfinis se basant sur le nombre d'itérations, la fonction d'écart (*gap function*) ou la meilleure borne inférieure. Mais, près de la solution optimale, la méthode de Frank-Wolfe converge mal (Luenberger (1984)), les solutions ont tendance à osciller. En raison de ce problème, de nombreuses variantes ont été développées qui tentent d'améliorer le taux de convergence. Parmi elles, la méthode PARTAN (*Parallel Tangent*). Cette dernière alterne une itération ordinaire de l'approximation linéaire avec une itération PARTAN durant laquelle une nouvelle direction de recherche est générée à l'aide des deux derniers points trouvés. LeBlanc et al. (1985) ont été les premiers à appliquer cette méthode au problème d'affectation d'équilibre du réseau.

Guélat (1983) compare trois variantes de Frank-Wolfe : heuristique, PARTAN, restriction qui consiste à minimiser la fonction économique sur un domaine restreint avant de poursuivre Frank-Wolfe, et une méthode d'approximation quadratique qui génère des directions de descente à l'aide d'une approximation quadratique de la fonction économique et les adapte au programme TRAFFIC.

2.2 Algorithmes de décomposition simpliciale

La méthode de décomposition simpliciale est appliquée à des fonctions objectifs non linéaires, continues et différentiables avec contraintes linéaires sur des domaines réalisables convexes. C'est un cas particulier du principe de décomposition de Dantzig-Wolfe ; elle est reliée à la région réalisable et fondée sur le théorème de Carathéodory pour lequel un polyèdre fermé peut être exprimé en termes de ses points extrêmes.

Adaptée au problème d'affectation d'équilibre de réseau avec demande fixe, elle consiste à résoudre alternativement des sous problèmes de plus courts chemins afin de générer des points extrêmes du polyèdre réalisable et des problèmes maîtres dans l'enveloppe convexe des points extrêmes générés. L'efficacité de l'algorithme repose sur la résolution du problème maître.

La méthode a été développée grâce aux travaux de Holloway (1974) et Hohenbalken (1977). Holloway a développé une extension de l'algorithme de Frank-Wolfe, il a utilisé une représentation interne du domaine réalisable suivie d'une restriction afin d'améliorer le taux de convergence. Quant aux travaux de Hohenbalken, ils ont démontré qu'il est possible d'éliminer les colonnes ayant un poids nul de la solution du problème maître.

L'algorithme décrit ci-dessous est l'algorithme de décomposition simpliciale restreint appliqué au problème d'affectation d'équilibre. Il revient à Hearn et al. (1987). Il est dit restreint, car il permet de contrôler le nombre de points extrêmes générés à chaque itération pour éviter que ce dernier ne devienne immense.

Le problème à résoudre est donc le suivant :

$$min\ S(v) = \sum_{a \in A} \int_0^{v_a} s_a(x)dx \tag{2.17}$$

s.à

$$\sum_{k \in K_p} h_k = g_p, \qquad p \in P, \tag{2.18}$$

$$h_k \geq 0, \qquad k \in K, \tag{2.19}$$

$$v_a = \sum_{k \in K} \delta_{ak} h_k, \qquad \forall a \in A. \tag{2.20}$$

On suppose que le domaine réalisable est borné et qu'il existe un nombre fini de points extrêmes du domaine réalisable Θ. Chaque point de Θ peut être alors écrit comme une

combinaison convexe de ces points extrêmes.

$\Theta_y^{(l)}$: Ensemble des points extrêmes de Θ retenus à l'itération l.

$\Theta_v^{(l)}$: Ensemble contenant les points extrêmes de l'itération l.

$\Theta^{(l)}$: Ensemble de points retenus pour la résolution du problème maître à l'itération l.

$q \geq 1$: Paramètre qui contrôle le nombre des points extrêmes à chaque itération.

L'algorithme se présente comme suit :

- **Etape 0** : Soit $v^{(0)}$ un point extrême réalisable. $\Theta_y^{(0)} = \emptyset$, $\Theta_v^{(0)} = \{v^{(0)}\}$ et $l = 0$.
- **Etape 1** : Résoudre le sous problème suivant :

$$min \sum_{a \in A} s_a(v_a^{(l)}) y_a \tag{2.21}$$

s.à

$$\sum_{k \in K_p} h_k = g_p, \qquad \forall p \in P, \tag{2.22}$$

$$h_k \geq 0, \qquad \forall k \in K, \tag{2.23}$$

$$y_a = \sum_{k \in K} \delta_{ak} h_k, \qquad \forall a \in A. \tag{2.24}$$

Cette étape présente le calcul habituel des chemins les plus courts comme à l'étape 1 de l'algorithme d'approximation linéaire de Frank-Wolfe. Soit $y^{(l)}$ la solution du programme linéaire. Si $s(v^{(l)})(y^{(l)} - v^{(l)}) \geq 0$, alors $v^{(l)}$ est la solution et s'arrêter. Sinon, si $|\Theta_y^{(l)}| < q$, alors $\Theta_y^{(l+1)} = \Theta_y^{(l)} \cup \{y^{(l)}\}$ et $\Theta_v^{(l+1)} = \Theta_v^{(l)}$. Si $|\Theta_y^{(l)}| = q$, remplacer l'élément de $\Theta_y^{(l)}$ qui a le plus faible poids dans l'expression de $v^{(l)}$ par $y^{(l)}$ afin d'obtenir $\Theta_y^{(l+1)}$, et poser $\Theta_v^{(l+1)} = \{v^{(l)}\}$. Poser $\Theta^{(l+1)} = \Theta_y^{(l+1)} \cup \Theta_v^{(l+1)}$.

- **Etape 2** (problème maître) : $v^{(l+1)} \in argmin\{S(v) : v \in co(\Theta^{(l+1)})\}$, $v^{(l+1)} = \sum_{i=1}^{m} \lambda_i z_i$ où $m = |\Theta^{(l+1)}|$ et $z_i \in \Theta^{(l+1)}$. Enlever tous les éléments z_i pour lesquels $\lambda_i = 0$ de $\Theta_y^{(l+1)}$ et de $\Theta_v^{(l+1)}$. Faire $l = l + 1$ et retourner à l'étape 1.

Chapitre 3
Algorithmes d'équilibre de chemins étudiés

Nous exposons dans ce chapitre les algorithmes d'équilibre de chemins étudiés dans ce projet. Il s'agit des algorithmes du simplexe convexe, des gradients réduit et projeté adaptés au problème d'affectation d'équilibre.

Ces trois algorithmes se déroulent dans l'espace des chemins et proposent des solutions où tous les coûts des chemins d'une même paire O-D sont égaux. Comme le nombre de chemins croit exponentiellement avec la taille du réseau, les méthodes sont implantées avec une approche de décomposition par paire O-D, ce qui simplifie le processus de résolution. Cette décomposition cyclique, identique à la décomposition (ou relaxation) de Gauss-Seidel, résoud pour chaque paire O-D un seul problème durant un cycle de l'algorithme. Les flots pour les autres paires demeurent fixes. Nous adoptons une approche de décomposition décrite par Florian (1986). Elle consiste à itérer selon le schéma suivant :

- **Etape 0** : $p = 0, p' = 0$.
- **Etape 1** : Si $p' = |P|$, le nombre total des paires O-D, s'arrêter. Sinon $p = p \bmod |P| + 1$.
- **Etape 2** : Si la solution courante est optimale pour le sous problème p (3.1-3.5), poser $p' = p' + 1$, et aller à l'étape 1, sinon résoudre le sous problème p. Mettre à jour les flots courants, poser $p' = 0$ et retourner à l'étape 1.

Nous rappelons ci-dessous l'énoncé du sous-problème à résoudre :

$$\min \sum_{a \in A} \int_0^{v_a^p + \bar{v_a}} s_a(t)dt \tag{3.1}$$

s.à

$$\sum_{k \in K_p} h_k = g_p, \quad p \in P, \tag{3.2}$$

$$h_k \geq 0, \qquad k \in K, \tag{3.3}$$

$$\bar{v_a} = \sum_{p' \neq p} \sum_{k \in K_{p'}} \delta_{ak} h_k, \tag{3.4}$$

$$v_a^p = \sum_{k \in K_p} \delta_{ak} h_k, \qquad \forall a \in A. \tag{3.5}$$

La convergence de cette méthode est assurée, car la fonction objectif est strictement convexe et elle décroit à la résolution de chaque sous problème. Une preuve est fournie dans Auslender (1976).

Les algorithmes décrits ci-dessous s'appliquent à la résolution d'un sous-problème p pour une paire O-D fixée.

3.1 Algorithme du simplexe convexe

L'algorithme du simplexe convexe fut appliqué au modèle d'équilibre de réseau avec une stratégie de décomposition par Nguyen (1973, 1974) qui appliqua la méthode au réseau de Winnipeg et obtint des résultats encourageants. L'algorithme est utilisé pour résoudre (3.1)-(3.5). Il est défini dans l'espace des chemins ; une solution est obtenue quand tous les chemins utilisés ont même coût pour chaque paire O-D. Il procède de la manière suivante : trouver le chemin le moins coûteux et le chemin le plus coûteux et transférer les flots entre

les deux chemins de telle manière à égaliser leur coût. Cette procédure est réalisée grâce à une recherche linéaire sur la variable λ.

Considérons $K_p^+ = \{k \in K_p, h_k > 0\}$, l'ensemble des chemins avec flot positif. Un chemin appartenant à K_p^+ est dit actif.

L'algorithme alors comprend les étapes suivantes :

- **Etape 0** : Trouver une solution initiale v_a^p, $s_a = s_a(v_a^p + \bar{v_a})$ et K_p^+ pour p fixé.
- **Etape 1** : Calculer les coûts des chemins courants utilisés s_k, $k \in K_p^+$, $s_k = \sum_{a \in A} \delta_{ak} s_a(v_a)$ et trouver k_1 tel que $s_{k_1} = min_{k \in K_p^+} s_k$ et k_2 tel que $s_{k_2} = max_{k \in K_p^+} s_k$. Si $(s_{k_2} - s_{k_1}) \leq \epsilon$ aller à l'étape 4, sinon définir des directions de descente $d_{k_1} = s_{k_2} - s_{k_1}$ pour le chemin k_1 et $d_{k_2} = s_{k_1} - s_{k_2}$ pour le chemin k_2.
- **Etape 2** : Trouver le pas optimal λ qui redistribue le flot $h_{k_1} + h_{k_2}$ entre les chemins k_1 et k_2 de tel manière que leurs coûts soient égaux. Pour cela, il faut résoudre

$$\min_{0 \leq \lambda \leq \frac{-h_{k_2}}{d_{k_2}}} \sum_{a \in A} \int_0^{v_a^p + \lambda y_a + \bar{v_a}} s_a(x) dx$$

  où $y_a = \delta_{ak_1} d_{k_1} + \delta_{ak_2} d_{k_2}$.
- **Etape 3** : Mettre à jour les variables de flot ainsi que les coûts sur les arcs et sur les chemins.

  $h_k = h_k + \lambda d_k, k = k_1,\ k_2; v_a^p = v_a^p + \lambda y_a; s_a = s_a(v_a^p + \bar{v_a})$
- **Etape 4** : Identifier le plus court chemin $\tilde{k}$ de coût $s_{\tilde{k}} = min_{k \in K_p} s_k$. Si $s_{\tilde{k}} < min_{k \in K_p^+} s_k$, alors ajouter $\tilde{k}$ à l'ensemble des chemins actifs K_p^+, $K_p^+ = K_p^+ \cup \tilde{k}$ et aller à l'étape 1. Sinon s'arrêter.

  $\bar{v_a}$ et v_a^p sont donnés par (3.4) et (3.5).

3.2 Adaptation de l'algorithme du gradient réduit au problème d'affectation d'équilibre

La méthode du gradient réduit revient à Wolfe (1967). Elle s'apparente à l'algorithme du simplexe, car elle utilise les notions de variables de base et hors base, d'une façon similaire à ce dernier. Elle est destinée à résoudre un problème d'optimisation à contraintes linéaires. La présentation qui suit s'inspire de Minoux (1983) et Luenberger (1984).

$$min\ f(x) \tag{3.6}$$

$$Ax = b, \tag{3.7}$$

$$x \geq 0. \tag{3.8}$$

où A est une matrice $(m \times n)$, $x \in R^n$ et $b = (b_1, b_2, ...b_m)^T, n > m$, f est continûment différentiable.

Nous supposons que chaque ensemble de m colonnes de A est linéairement indépendant. Soit B une base c'est à dire une sous matrice carrée $(m \times m)$ inversible. Suivant cette hypothèse $A = [B, N], x = [x_B, x_N]$ où x_B sont les variables de base (variables dépendantes), $x_B > 0$, et x_N les variables hors base (variables indépendantes), $x_N \geq 0$.

Le problème d'optimisation (3.6)-(3.8) devient :

$$\{\ min\ f(x_B, x_N) |\qquad Bx_B + Nx_N = b, \qquad x_B > 0, \qquad x_N \geq 0.\}$$

Les contraintes (3.7) sont alors équivalentes à :

$$x_B = B^{-1}b - B^{-1}Nx_N. \tag{3.9}$$

L'idée du gradient réduit est de considérer à chaque étape le problème seulement en terme des variables indépendantes (c'est un gradient dit réduit parce qu'il est fonction du vecteur x_N seulement).

Nous devons construire une direction de descente d, telle que $\bigtriangledown f(x)^T d < 0$ et $Ad = 0$.

La condition $Ad = 0$ est facile à établir :

$$A(x + \lambda d) = b \Longrightarrow Ax + \lambda Ad = b \Longrightarrow \lambda Ad = 0 \Longrightarrow Ad = 0.$$

En décomposant d comme suit $d = [d_B, d_N]$, nous avons alors :

$$\bigtriangledown f(x)^T d = \bigtriangledown_B f(x)^T d_B + \bigtriangledown_N f(x)^T d_N, \tag{3.10}$$

où

$$d_B = -B^{-1} N d_N, \tag{3.11}$$

et donc

$$\bigtriangledown f(x)^T d = (\bigtriangledown_N f(x)^T - \bigtriangledown_B f(x)^T B^{-1} N) d_N, \tag{3.12}$$

$$\bigtriangledown f(x)^T d = r_N^T d_N. \tag{3.13}$$

Par définition, $r^T = (r_B^T, r_N^T) = (0, \bigtriangledown_N f(x)^T - \bigtriangledown_B f(x)^T B^{-1} N)$ est appelé gradient réduit.

La direction du gradient réduit est donc calculée comme suit, pour tous les indices j correspondant à des variables hors base :

$$d_j = \begin{cases} -r_j, & \text{si } r_j \geq 0, \\ -x_j r_j, & \text{si } r_j < 0, \end{cases} \tag{3.14}$$

et pour les variables de base : $d_B = -B^{-1} N d_N$.

Si $d_j = 0, \forall j \in N$, la solution est optimale.

Il faut s'assurer que $x = [x_B, x_N] \geq 0$. Pour cela, il faut déterminer $\alpha_1, \alpha_2, \alpha_3$ tels que :

$$\alpha_1 = max\{\alpha | x_B + \alpha d_B \geq 0\}$$
$$\alpha_2 = max\{\alpha | x_N + \alpha d_N \geq 0\}$$
$$\alpha_3 \in argmin\{f(x + \alpha \bigtriangledown x) | 0 \leq \alpha \leq \alpha_1, 0 \leq \alpha \leq \alpha_2\}.$$

En appliquant le gradient réduit au problème d'affectation d'équilibre, on doit définir un chemin qui constitue la variable de base alors que les autres chemins de la paire O-D constituent les variables hors base. Florian (1986) définit la variable de base comme étant le chemin avec le plus grand flot.

Les matrices à définir donc sont les suivantes :

B=[1], il ya un chemin dans la base soit k_1 ce chemin.

N=[1,1,1,....1] où la dimension de N est 1 x (nombre des chemins de la paire - 1).

Nous calculons le gradient réduit de la fonction objectif du sous problème de la paire O-D p à résoudre par rapport à h_{k_1}

$$\frac{\partial f(h)}{\partial h_{k_1}} = \sum_{a \in A} \delta_{ak_1} s_a(v_a^p + \bar{v_a}) = s'_{k_1}(v), \tag{3.15}$$

et par rapport à h_k où k est un chemin hors base.

$$\frac{\partial f(h)}{\partial h_k} = \sum_{a \in A} \delta_{ak} s_a(v_a^p + \bar{v_a}) = s'_k(v), \tag{3.16}$$

où, rappelons-le,

$$\begin{aligned} \bar{v_a} &= \sum_{p' \neq p} \sum_{k \in K_{p'}} \delta_{ak} h_k, \\ v_a^p &= \sum_{k \in K_p} \delta_{ak} h_k, \\ B^{-1}N &= [1, 1, 1, ..., 1]. \end{aligned}$$

Sachant que l'expression du gradient réduit est $r^T = [0, \bigtriangledown_N f(x)^T - \bigtriangledown_B f(x)^T B^{-1} N]$ alors

$$r_k = s'_k(v) - s'_{k_1}(v). \tag{3.17}$$

L'adaptation de la méthode du gradient réduit au problème d'affectation d'équilibre fournit l'algorithme suivant :

- **Etape 0** : Calculer une solution initiale v_a^p, K_p^+.
- **Etape 1** : Choisir une variable de base h_{k_1}, où $k_1 \in argmax_{k \in K_p^+}\{h_k\}$,

  $K_p^+ = \{k |\ h_k > 0\}$.

  Calculer les coûts des chemins $s'_k = \sum_{a \in A} \delta_{ak} s_a(v_a), k \in K_p^+$.

  Calculer le gradient réduit : $r_k = s'_k(v) - s'_{k_1}(v), k \in K_p^+$.

  Calculer la direction de descente selon la règle suivante :

$$d_k = \begin{cases} -r_k & \text{si } r_k \geq 0 \ ou \ (h_k = 0 \ et \ r_k < 0), \\ -h_k r_k & \text{si } r_k < 0 \ et \ h_k > 0, \\ -\sum_{k \neq k_1} d_k & \text{si } k = k_1. \end{cases}$$

  Si $max_{k \in K_p^+}\{abs(d_k)\} < \epsilon$, aller à l'étape 4.
- **Etape 2** : Trouver le pas optimal λ, solution du problème suivant :

$$min \sum_a \int_0^{v_a^p + \bar{v_a} + \lambda y_a^p} s_a(x) dx \tag{3.18}$$

  s.à

$$0 \leq \lambda \leq min(\frac{-h_k}{d_k} | d_k < 0), \tag{3.19}$$

  où $y_a^p = \sum_{k \in K_p^+} \delta_{ak} d_k$.
- **Etape 3** : Mettre à jour les variables de flot sur les arcs et sur les chemins.

  $h_k = h_k + \lambda d_k, k \in K_p^+; v_a^p = v_a^p + \lambda y_a^p, a \in A$.
- **Etape 4** : Identifier le plus court chemin $\tilde{k}$ de coût $s'_{\tilde{k}}(v) = min_{k \in K_p} s'_k(v)$. Si $s'_{\tilde{k}}(v) < min_{k \in K_p^+} s'_k(v)$, alors ajouter le chemin $\tilde{k}$ à l'ensemble des chemins actifs K_p^+, $K_p^+ = K_p^+ \cup \tilde{k}$ et aller à l'étape 1. Sinon s'arrêter.

3.3 Adaptation de l'algorithme du gradient projeté au problème d'affectation d'équilibre

La méthode du gradient projeté proposée par Rosen (1960) est une méthode essentiellement intéressante dans le cas des contraintes linéaires.

Considérons le problème suivant :

$$min\ f(x) \tag{3.20}$$

$$A_i^T x \leq b_i, i \in I_1, \tag{3.21}$$

$$A_i^T x = b_i, i \in I_2. \tag{3.22}$$

Par définition, une contrainte $A_i^T \leq b_i$ est active, si elle est satisfaite à égalité. Soient $I(x)$ l'ensemble des indices des contraintes actives $I(x) \supset I_2$ et A_q la matrice correspondant aux contraintes actives.

C'est une méthode itérative qui consiste à projeter la direction opposée au gradient sur le sous espace défini par l'ensemble des contraintes actives. Si l'opération de projection est suffisamment simple, la méthode devient très opérationnelle.

Nous allons chercher une direction de déplacement d qui permet de diminuer le plus possible $f(x)$. Pour cela, nous allons projeter la direction opposée au gradient g sur le sous espace défini par l'ensemble des contraintes actives A_q. Ce qui conduit au problème suivant :

$$min\ \frac{1}{2}|-g-d|^2 \tag{3.23}$$

s.à

$$A_q^T d = 0, \tag{3.24}$$

où A_q est la matrice des contraintes actives. En supposant que les q contraintes qui la composent sont linéairement indépendantes, alors la matrice A_q est de rang q. La direction projetée doit satisfaire $A_q^T d = 0$ afin de s'assurer que toutes les contraintes actives demeurent actives.

Calculons la direction d. Pour cela, associons les multiplicateurs λ aux contraintes $A_q^T d = 0$ et appliquons les conditions d'optimalité de Karush-Kuhn-Tucker (Luenberger 1984).

$$(-g - d)(-1) + A_q^T \lambda = 0 \tag{3.25}$$

$$d = -g - A_q^T \lambda \tag{3.26}$$

Sous la condition que $A_q d = 0$, nous calculons λ comme suit :

$$A_q d = 0 = -A_q g - A_q A_q^T \lambda \tag{3.27}$$

$$A_q A_q^T \lambda = -A_q g \tag{3.28}$$

$$\lambda = -(A_q A_q^T)^{-1} A_q g \tag{3.29}$$

A_q étant de rang plein, la matrice $(A_q A_q^T)$ est inversible. D'où

$$d = -g + A_q^T((A_q A_q^T)^{-1} A_q g) \tag{3.30}$$

$$d = -g(I - A_q^T(A_q A_q^T)^{-1} A_q) \tag{3.31}$$

$$d = -gP \tag{3.32}$$

où $P = I - A_q^T(A_q A_q^T)^{-1} A_q$ est la matrice de projection.

Soumis (1978) fut l'un des premiers à appliquer la méthode du gradient projeté au problème d'affectation d'équilibre dans l'espace des chemins.

Dans ce cas, nous avons une seule contrainte active $\sum_{k \in K_p} h_k = g_p$ et $k \in K_p^+ (h_k > 0)$.

Supposons que nous avons n chemins dans K_p^+

$$\begin{aligned}
\mathcal{A}_q &= \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \\
\mathcal{A}_q A_q^T &= \begin{bmatrix} 1 & 1 & 1 \cdots & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \cdots \\ 1 \end{bmatrix} = n \\
(A_q A_q^T)^{-1} &= \frac{1}{n} \\
\mathcal{A}_q^T (A_q A_q^T)^{-1} A_q &= \begin{bmatrix} 1 \\ 1 \\ \cdots \\ 1 \end{bmatrix} [\frac{1}{n}] \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{n} & \cdots & \cdots & \frac{1}{n} \\ \frac{1}{n} & \cdots & \cdots & \frac{1}{n} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{1}{n} & \cdots & \cdots & \frac{1}{n} \end{bmatrix} \\
\mathcal{I} - A_q^T (A_q A_q^T)^{-1} A_q &= \begin{bmatrix} \frac{n-1}{n} & -\frac{1}{n} & \cdots & -\frac{1}{n} \\ -\frac{1}{n} & \frac{n-1}{n} & \cdots & -\frac{1}{n} \\ -\frac{1}{n} & \cdots & \frac{n-1}{n} & \cdots \\ -\frac{1}{n} & \cdots & \cdots & \frac{n-1}{n} \end{bmatrix}
\end{aligned}$$

Les gradients des chemins sont s_k, $k = 1, 2, ..., n$.

Le gradient projeté dans le cas $k = 1$ est $\frac{n-1}{n} s_1 - \frac{1}{n}(s_2 + ... + s_n) = s_1 - \frac{1}{n}(\sum_{i=1}^{n} s_k) = s_1 - \bar{s}$, où $\bar{s}$ est le coût moyen des chemins.

En général, le gradient projeté est $(s_k - \bar{s}), k = 1, 2, ..., n$.

L'adaptation de la méthode du gradient projeté au problème d'affectation d'équilibre fournit l'algorithme suivant :

- **Etape 0** : Calculer une solution initiale v_a^p, K_p^+.
- **Etape 1** : Calculer la direction de descente telle que : $d_k = \bar{s_k} - s_k, k \in K_p^+$, où $\bar{s_k}$ est la moyenne des coûts des chemins $k \in K_p^+$.

  Si $max_{k \in K_p^+}\{abs(d_k)\} < \epsilon$, aller à l'étape 4.

- **Etape 2** : Trouver le pas optimal λ qui résoud le problème suivant :

$$min \sum_a \int_0^{v_a^p + \bar{v}_a + \lambda y_a^p} s_a(x)dx \tag{3.33}$$

s.à

$$0 \leq \lambda \leq min(\frac{-h_k}{d_k} | d_k < 0), \tag{3.34}$$

où $y_a^p = \sum_{k \in K_p^+} \delta_{ak} d_k$.

- **Etape 3** : Mettre à jour les variables de flot sur les arcs et sur les chemins. $h_k = h_k + \lambda d_k, k \in K_p^+; v_a^p = v_a^p + \lambda y_a^p, a \in A$.
- **Etape 4** : Identifier le plus court chemin $\tilde{k}$ de coût $s'_{\tilde{k}}(v) = min_{k \in K_p} s'_k(v)$, si $s'_{\tilde{k}}(v) < min_{k \in K_p^+} s'_k(v)$ alors ajouter le chemin $\tilde{k}$ à l'ensemble des chemins actifs K_p^+, $K_p^+ = K_p^+ \cup \tilde{k}$ et aller à l'étape 1. sinon s'arrêter.

Pour chacun des algorithmes étudiés, la solution initiale est obtenue en affectant toute la demande de la paire O-D sur le chemin le plus court de cette paire suivant la stratégie "tout ou rien". Le plus court chemin est obtenu à partir d'un flot nul sur le réseau.

Ces algorithmes peuvent être résolus selon deux approches : l'approche Gauss-Seidel et l'approche Jacobi. Selon la première approche, les informations de flots et de coûts mises à jour lors d'un traitement d'une paire O-D sont considérés dans le traitement des paires O-D prochaines. Elle est caractérisée par le calcul d'un plus court chemin au court du traitement d'une paire O-D. Le fait que l'information la plus récente soit considérée fait que ce type d'approche converge plus rapidement. Pour la version Jacobi, les mises à jour ne sont considérées dans les prochaines paires O-D qu'une fois toutes les paires O-D sont traitées, c'est-à-dire qu'à l'itération suivante. Un arbre de plus courts chemins est généré pour toutes les paires O-D ayant une origine commune. Les paires O-D qui ont une origine

commune sont traitées dans l'ordre. Une approche alternative du genre bloc Gauss-Seidel par origine est décrite par Nguyen (1973) ; elle exploite la structure d'arbres des bases rendue possible par la formulation sommet-arc. Un sous-problème est associé à chaque origine et non plus à chacune des paires O-D.

Chapitre 4
Implantation des algorithmes étudiés

Dans ce chapitre, nous nous intéressons à l'implantation des algorithmes d'affectation d'équilibre présentés dans le chapitre précédent. Les deux premières sections discutent respectivement de la présentation des données du réseau et des structures de données utilisées. Les deux sections suivantes s'attardent sur les algorithmes de plus courts chemins et de recherche unidimensionnelle adoptés. Quant à la dernière section, elle présente la hiérarchie des classes développées.

4.1 Les données du réseau

Le format adopté pour l'exploitation du réseau est le format NFF (*Network File Format*) (Tremblay 1998). L'utilisation de ce format se justifie par la facilité de représenter les données relatives à un réseau et également par la présence d'une librairie de support en C++. Cette dernière permet de générer de nouvelles informations à partir des données minimales fournies au départ. A titre d'exemple, nous présentons le réseau de Braess à l'aide du format de fichier NFF, dans lequel chaque section (Nodes, Centroids, Links) contient un certain nombre d'attributs. Les coûts des arcs sont $s_1(v_1) = 10v_1$, $s_2(v_2) = v_2 + 50$, $s_3(v_3) = v_3 + 50$, $s_4(v_4) = 10v_4$. Il y a une paire O-D dont la demande est 6.

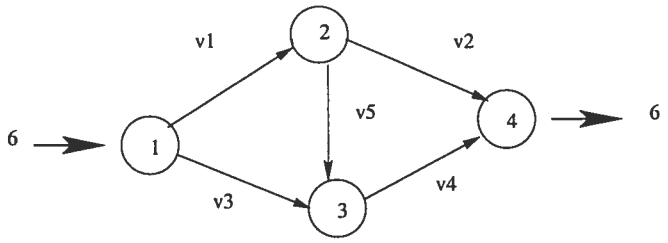

Figure 4.1 – Réseau de Braess

```
<nff 3.0>
<title>reseau de Braess</title>
<functions VolumeDelayLink "volad">
1, "10*volad";
2, "volad+50";
3, "volad+10";
</functions>
<section Nodes>
<structure>
long ID;
float x;
float y;
</structure>
<data>
1,23.45,34.45;
2,44.67,44.12;
3,12.34,66.90;
4,43.68,44.22;
</data>
</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;
function VolumeDelayLink LinkFunction;
</structure>
<data>
1,1,2,1;
2,2,4,2;
3,1,3,2;
4,3,4,1;
5,2,3,3;
</data>
</section>
<matrix demands double>
   <index>
      Centroids From;
      Centroids To;
   </index>
   <data>
      1,2:6;
      </data>
</matrix>
</nff>
```

Compte tenu de la nature du problème d'affectation d'équilibre dans un réseau de transport, les sections que nous avons représentées prennent en considération les délais et les interdictions associés à chaque intersection. Elles sont définies comme suit :

- Noeuds : comportant les attributs identificateur, une référence au centroïde correspondant au noeud (si le noeud n'est pas un centroïde, elle vaut -1), une liste des arcs entrant et sortant du noeud et une liste des virages présents au noeud.
- Centroïdes : comportant les attributs identificateur et une référence au noeud associé au centroïde.
- Arcs : les attributs sont un identificateur, une référence au noeud origine de l'arc, une référence au noeud terminal de l'arc, la longueur, le nombre de voies en circulation, la liste des virages sortant de l'arc et un identifiant de la fonction coût.
- Virages : les attributs sont un identificateur, une référence au noeud comportant le virage, une référence au lien origine du virage, une référence au lien destination du virage et un identifiant de la fonction coût.
- Fonctions coûts des arcs : elles sont fonction de la longueur de l'arc, du flot et du nombre de voies.
- Fonctions coûts des virages.
- Matrice comportant pour chaque paire origine-destination une demande positive.

4.2 Les structures de données reliées aux chemins du réseau.

Les algorithmes développés évoluent dans l'espace des chemins, ils sauvegardent tous les chemins avec du flot positif alors une partie de la mémoire est utilisée particulièrement

pour les réseaux de moyenne et grande taille comme dans le cas des villes d'Ottawa et Montréal. Au fil des itérations, plusieurs chemins sont crées et mémorisés, d'où l'importance de définir une structure de données qui reflète la qualité de l'implantation.

Les trois algorithmes sont relativement voisins à certains égards, les structures de données adoptées sont par conséquent identiques dans les trois cas.

Deux structures de données sont définies pour la gestion des chemins : une structure de données dynamique et une structure de données statique.

- La structure de données dynamique permet la gestion des chemins de la même paire O-D, elle réserve ou supprime l'espace à un chemin au besoin. Elle se présente sous forme de listes linéaires chaînées, où un chemin est composé de plusieurs maillons de la liste. Chaque maillon comporte les informations suivantes :

  - Le numéro du chemin,
  - Le numéro d'arc,
  - Le flot sur le chemin,
  - Le numéro de virage, (si le chemin ne contient pas de virage, il vaut -1),
  - Le pointeur qui pointe sur l'arc suivant dans le chemin.

Dès que le numéro du chemin change, on passe à un chemin différent.

Dans le réseau de Braess, les flots sur les chemins h^* et les coûts u^* satisfaisant les conditions d'équilibre sont :

Paire O-D 1 :

$h^*_{(1,2,4)} = 2\,;\, s_{(1,2,4)} = 92$

$h^*_{(1,3,4)} = 2\,;\, s_{(1,3,4)} = 92$

$h^*_{(1,2,3,4)} = 2\,;\, s_{(1,2,3,4)} = 92\,;\, u^* = 92$

Le vecteur de flots v^* correpondant est :

$v^*_1 = 4\,;\, v^*_2 = 2\,;\, v^*_3 = 2\,;\, v^*_4 = 4\,;\, v^*_5 = 2\,;$

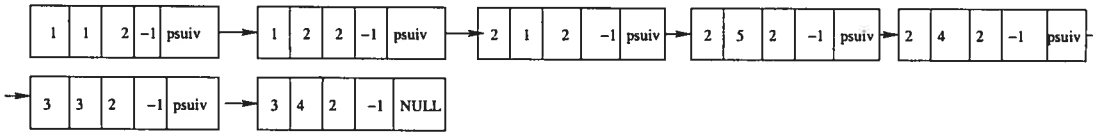

Figure 4.2 – Représentation de la structure des chemins

La figure (4.2) fournit la représentation des chemins obtenus selon la structure précédemment définie.

Pour calculer le flot sur chaque arc, il suffit de regrouper les flots sur tous les maillons identifiés avec ce numéro d'arc.

Cette structure est intéressante, car toutes les informations relatives à un chemin s'y trouvent. Lorsqu'un chemin se retrouve avec un flot nul lors du calcul de la solution, il est éliminé automatiquement de la liste. Une suppression d'un chemin entraine la suppression de tous les maillons qui le composent.

- La structure de données statique sauvegarde l'adresse du premier chemin d'une paire O-D. C'est un tableau dont la taille est égale au nombre total des paires O-D calculé à partir de la matrice des demandes.

Chaque chemin généré est un chemin le plus court de la paire O-D impliquée basé sur les coûts courants ; quand celui-ci est identifié, il est comparé aux chemins existants. On commence par comparer le nombre d'arcs du nouveau chemin avec ceux déjà connus. Si le nombre d'arcs du chemin le plus court est différent du nombre d'arcs de tous les autres chemins, alors c'est un nouveau chemin. Dans le cas contraire, on compare ce chemin arc par arc à chacun des chemins connus. Si pour un chemin donné tous les arcs sont égaux respectivement, on interrompt la comparaison, le chemin existe déjà et aucune modification n'est apportée à l'information reliée à ce chemin. Sinon c'est un nouveau chemin, il est ajouté à la fin de la liste et son numéro est incrémenté automatiquement.

4.3 L'algorithme des plus courts chemins

Dans les trois méthodes testées sur les réseaux provenant des trois villes canadiennes suivantes : Winnipeg, Ottawa et Montréal, la phase de recherche du plus court chemin occupe plus de 50% du temps total d'exécution. Dans le programme EMME/2 basé sur l'algorithme de Frank-Wolfe, ce taux est de 82% pour le réseau de Winnipeg et de 74% pour les réseaux d'Ottawa et Montréal, d'où l'importance de choisir un algorithme de plus court chemin performant et rapide.

La recherche du plus court chemin est un problème qui a été largement étudié, plusieurs algorithmes ont été développés, adaptés à plusieurs domaines de recherche. Dans le cadre des tests effectués, nous avons adopté l'approche Gauss-Seidel pour le développement de nos algorithmes. Cette méthode converge plus rapidement que la variante Jacobi. Dans cette approche, le problème consiste à identifier un plus court chemin de la paire O-D courante. Il est alors approprié d'utiliser un algorithme utilisant une technique d'extension selective *(label setting)* qui permet d'arrêter le calcul dès que la destination est rencontrée. Compte tenu également de la nature du problème qui doit prendre en considération les délais et les interdictions aux intersections, notre choix s'est porté sur l'algorithme de Spiess (Tremblay 1998) conçu initialement pour résoudre le problème d'affectation d'équilibre dans un réseau urbain à demande fixe avec pénalités de virages sur les noeuds d'intersection (Spiess 1984). Il est également utilisé dans le logiciel de planification de transport EMME/2.

Contrairement aux méthodes conventionnelles qui assignent des étiquettes aux sommets, l'approche développée par Spiess assigne des étiquettes aux liens et suggére que les mouvements avec pénalités ne se trouvent que dans un sous ensemble de noeuds. Dans tous les

autres noeuds, tous les mouvements sont permis sans délai. L'algorithme s'assure également de ne pas considérer les chemins contenant des virages en U dans le cas où le noeud terminal du lien traité est un noeud régulier. Avant de décrire l'algorithme, définissons la notation utilisée :

Soient $G = (N, A)$ un réseau où pour chaque lien $a = (i, j) \in A$ est associée une fonction coût $s : A \rightarrow R$, $A_i^+ = \{(i, j) \in A\}$ et $A_i^- = \{(j, i) \in A\}$ représentant respectivement l'ensemble des liens sortant et l'ensemble des liens entrant au noeud i. D est l'ensemble des noeuds destinations et $\bar{N} \subseteq N$ un sous ensemble des noeuds pour lesquels il existe des mouvements pénalisés ou interdits. s_{ijk} est le délai associé au mouvement de (i, j) vers (j, k). Soient également π_a (π_q), le coût d'un plus court chemin de l'origine r jusqu'à la fin du lien a (la destination q) et b_a (b_q), le lien précédent le lien a (la destination q). Soit Q l'ensemble des liens qui ont été rejoints et ϕ_i , $i \in N$, qui prend la valeur 1 si le chemin de r à i ne contient pas de noeuds $k \in \bar{N}$ et 0 sinon.

L'algorithme s'écrit alors comme suit :

Etape 1 : *Initialisation*

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

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

$\pi_a = \infty, b_a = -1, \forall a \in A - A_r^+$;

$\pi_a = s_a, \forall a \in A_r^+$;

$Q = A_r^+$.

Etape 2 : *Sélection des liens à traiter*

tant que $Q \neq \emptyset$ faire

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

si $j \in D$, alors aller à l'étape 3 ;

si $j \in \bar{N}$, alors aller à l'étape 4 ;

autrement aller à l'étape 5 ;

$Q = Q - \{\bar{a}\}$.

Etape 3 : *Le noeud j est une destination.*

$\pi_q = \pi_{\bar{a}}, b_q = \bar{a}$.

Etape 4 : *Le noeud j est une intersection.*

pour $a = (j,k) \in A_j^+$ tels que $\phi_k = 0$ faire

si $\pi_{\bar{a}} + s_{ijk} + s_a < \pi_a$, alors

$\pi_a = \pi_{\bar{a}} + s_{ijk} + s_a$;

$b_a = \bar{a}$;

$Q = Q + \{a\}$.

Etape 5 : *Le noeud j est un noeud régulier*

pour $a = (j,k) \in A_i^+$ tels que $k \neq i$ faire

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

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

$b_a = \bar{a}$;

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

$\phi_j = \phi_i$.

L'algorithme a été implanté avec une stratégie de tri des étiquettes de type monceau (heap).

Le tableau (4.1) résume la performance de l'algorithme de Spiess testé sur les réseaux des trois villes canadiennes suivantes : Winnipeg, Ottawa et Montréal.

Ville	Temps CPU(sec)
Winnipeg	0,11
Ottawa	0,54
Montréal	3,68

Tableau 4.1 – Performance de l'algorithme de Spiess

4.4 La recherche unidimensionnelle

Chacun des trois algorithmes développés comprend une étape de recherche du pas optimal λ^*. Le sous-problème de minimisation à résoudre est le suivant :

$$\min_{0 \leq \lambda \leq \Delta} \sum_{a \in A} \int_0^{v_a^p + \lambda y_a + \bar{v_a}} s_a(x)dx \tag{4.1}$$

Le pas optimal λ^* est celui qui annule la dérivée de la fonction objectif du sous-problème.
$f' = \frac{\partial f_\lambda(v)}{\partial \lambda} = \sum_{a \in A} s_a(v_a^p + \lambda y_a + \bar{v_a} y_a) = 0$

f étant convexe, f' est monotone croissante. La connaissance analytique de la dérivée permet de transformer le problème de minimisation en un problème de recherche du zéro d'une fonction monotone croissante.

Beaucoup de méthodes traditionnelles existent pour résoudre ce problème, certaines utilisent un calcul de dérivées et d'autres non. Parmi elles figurent la méthode de Newton-Raphson, la section dorée, la bissection, la sécante, et l'interpolation quadratique.

Nous avons comparé deux différentes méthodes de recherche linéaire : une méthode combinant la bissection et une variante de la sécante qui revient à Guélat(1983), et la méthode

de l'interpolation quadratique.

Méthode de Guélat

Cette approche combine le faible intervalle d'incertitude de la méthode de la bissection au taux de convergence rapide de la sécante. L'algorithme s'énonce comme suit :

- **Etape 0** : $\lambda_a = 0, \lambda_b = \Delta$.
- **Etape 1** : Si $f'(\lambda_a)$ et $f'(\lambda_b)$ ont le même signe, alors

  si $f(\lambda_a) \leq f(\lambda_b)$, alors $\lambda^* = \lambda_a$,

  sinon $\lambda^* = \lambda_b$, s'arrêter.
- **Etape 2** : La bissection

  Calculer $\lambda = \frac{\lambda_a + \lambda_b}{2}$;

  Si $f'(\lambda_a) et f'(\lambda)$ ont le même signe, alors

  $\lambda_a = \lambda$, sinon $\lambda_b = \lambda$.
- **Etape 3** : La fausse position

  Calculer $\lambda = \lambda_b - \frac{\lambda_b - \lambda_a}{f'(\lambda_b) - f'(\lambda_a)} f'(\lambda_b)$.

  Si $|f'(\lambda)| < \epsilon$, alors $\lambda^* = \lambda$, s'arrêter.

  Si $f'(\lambda) et f'(\lambda_a)$ ont le même signe, alors

  $\lambda_a = \lambda$, sinon $\lambda_b = \lambda$;

  Aller à l'étape 2.

Méthode d'interpolation quadratique

L'idée est d'approximer la fonction à optimiser par une parabole $ax^2 + bx + c$, ce qui a l'avantage d'assurer le passage de la courbe par trois points a, b, c. Nous approximons donc la fonction sur l'intervalle $[a, c]$ par la quadratique ayant les mêmes valeurs que cette dernière aux points a, b, c. La description de l'algorithme suit ci-dessous :

- **Etape 0** : $\lambda_a = 0, \lambda_b = \Delta$.
- **Etape 1** : Si $f'(\lambda_b) < 0$, alors $\lambda^* = \lambda_b$ et s'arrêter.

  Si $f'(\lambda_a) > 0$, alors $\lambda^* = \lambda_a$ et s'arrêter.

- **Etape 2** : Calculer $\lambda_1 = (\lambda_a + \lambda_b)/2$;

  $f_{23} = (f'(\lambda_1) - f'(\lambda_b))/(\lambda_1 - \lambda_b)$;

  $f_{13} = (f'(\lambda_a) - f'(\lambda_b))/(\lambda_a - \lambda_b)$;

  $a = (f_{13} - f_{23})/(\lambda_a - \lambda_1)$;

  $b = f_{23} - a(\lambda_1 + \lambda_b)$;

  $c = f'(\lambda_1) - \lambda_1(b + a\lambda_1)$;

  $discr = bb - 4ac$.

- **Etape 3** : Si $(f_{23} \neq 0)$, alors $\lambda^* = \lambda_b - f'(\lambda_b)/f_{23}$;

  si $(discr \geq 0)$, alors $\lambda^* = -2c/(b + \sqrt{discr})$;

  si $(\lambda^* > \Delta,)$ alors $\lambda^* = \Delta$;

  si $(\lambda^* < 0)$, alors $\lambda^* = 0$;

  si $(|f'(\lambda^*)| <= 0.01)$, alors s'arrêter, sinon

  si $f'(\lambda^*) < 0$, alors $\lambda_a = \lambda^*$,

  sinon $\lambda_b = \lambda^*$, aller à l'étape 2.

Les résultats des deux méthodes s'avèrent très rapprochés. Néanmoins, un léger avantage est attribué à la méthode d'interpolation quadratique en terme de précision. Notre choix s'est porté donc sur cette dernière, car en plus elle ne nécessite à aucun moment un calcul de la fonction objectif du sous problème.

4.5 Le développement des implantations

Le développement des implantations a été effectué à l'aide du langage C++. Afin d'éviter la duplication des informations, nous avons utilisé le concept d'héritage.

La figure (4-3) permet de montrer la hiérarchie des classes développées. Au plus haut ni-

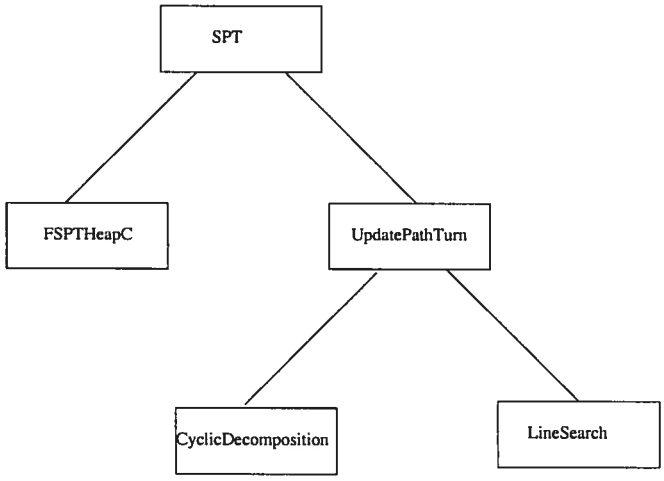

Figure 4.3 – Hiérarchie des classes développées

veau, nous retrouvons la classe SPT. Cette classe regroupe les outils de gestion du réseau. Dérivant de cette classe, nous retrouvons la classe FSPTHeapC qui comprend les outils de calcul du plus courts chemins considérant les délais et les interdictions qui peuvent exister aux intersections. La structure de données utilisée est le monceau (*heap*). L'intérêt d'utiliser cette implantation appliquant une technique d'extension sélective (*label setting*) est d'arrêter le calcul dès que le noeud destination est atteint. Au même niveau, se trouve la classe UpdatePathTurn qui regroupe les outils de gestion des chemins actifs et des virages (création, suppression, mise à jour du volume...). Au niveau inférieur existent deux classes : CyclicDecomposition et LineSearch. La première classe comporte toutes les méthodes permettant le déroulement de la stratégie de décomposition cyclique par paire O-D retenue et la deuxième classe regroupe les outils de calcul de la recherche unidimensionnelle qui a pour but d'identifier le pas optimal λ^* le long d'une direction de descente. Le détail des méthodes des différentes classes est fourni en annexe B.

Chapitre 5
Présentation et analyse des résultats

Dans ce chapitre, nous présentons les résultats numériques visant à valider les performances de nos algorithmes. Nous appliquons les algorithmes sur les réseaux de transport routier des villes canadiennes suivantes : Winnipeg, Ottawa et Montréal dont les caractéristiques sont les suivantes :

Ville	Nb. noeuds	Nb. centroïdes	Nb. liens	Nb. virages	Nb. paires O-D
Winnipeg	1067	154	2534	338	4344
Ottawa	9087	258	6963	703	6527
Montréal	9987	699	19304	490	34922

Tableau 5.1 – Caractéristiques des réseaux

Nous avons programmé les trois algorithmes en C++ et la mise au point est faite sur des stations de travail SUN cadencée à 1200 MHZ et possédant 16G de RAM fonctionnant sous UNIX. Notons que pour l'ensemble des tests, le code exécutable a été compilé à l'aide de l'option d'optimisation du compilateur CC, ce qui procure une réduction du temps de calcul de l'ordre de 50%.

Le critère d'arrêt des algorithmes testés est tel que l'ensemble des paires O-D soit résolu à l'optimalité, c'est-à-dire que les coûts des chemins avec flot positif de chaque paire sont tous égaux et qu'aucun chemin plus court ne peut être ajouté à la paire. Néanmoins une tolérance d'erreurs sur les coûts des chemins est admise ; plus cette tolérance est faible, plus la qualité des affectations d'équilibre calculées est bonne. Nous avons choisi une tolérance d'erreurs de l'ordre de $0,5\%$ pour les réseaux de Winnipeg et Ottawa et de 2% pour le

grand réseau de Montréal. La tolérance d'erreurs est la différence relative entre le chemin le plus coûteux s_{k_2} et le chemin le moins coûteux s_{k_1} ; elle est calculée selon la formule suivante : $(s_{k_2} - s_{k_1})/s_{k_1}$.

Compte tenu de la taille des réseaux traités, nous interrompons le calcul des affectations avant que le critère d'arrêt ne soit satisfait. Un calcul plus précis nécessiterait un plus grand temps d'exécution. Nous nous limitons à un nombre maximal d'itérations préalablement défini. Lorsque nous parlons du nombre d'itérations, nous entendons le nombre de fois que toutes les paires O-D sont traitées.

Pour mesurer la qualité des solutions trouvées , nous avons considéré deux mesures d'écart relatif. Il faut se rappeler que l'écart relatif est plus petit ou égal à 1 et qu'il tend vers zéro à mesure que nous nous approchons de la solution optimale.

L'écart relatif 1 :

$$\frac{\sum_{a\in A} s_a(v_a).v_a - \sum_{p\in P} g_p u_p}{\sum_{a\in A} s_a(v_a).v_a} \tag{5.1}$$

L'écart relatif 2 :

$$\frac{\sum_{a\in A} \int_0^v s_a(x)dx - BLB}{\sum_{a\in A} \int_0^v s_a(x)dx} \tag{5.2}$$

où

u_p est le coût du plus court chemin de la paire O-D p ;

g_p est la demande de la paire O-D p ;

BLB est la meilleure borne inférieure obtenue jusqu'à l'itération l :

$$BLB = max_{i=1,2,\ldots,l} \sum_{a\in A} \int_0^v s_a(x)dx + s(v^{(i)})(y^{(i)} - v^{(i)}), \tag{5.3}$$

où y est le vecteur de flot obtenu en affectant toute la demande sur le plus court chemin de la paire O-D.

Il faut noter que le calcul de l'écart relatif 2 nécessite le calcul des plus courts chemins pour toutes les paires O-D à la fin de chaque itération.

5.1 Réseau de Winnipeg

Nous exposons d'abord les résultats obtenus avec le réseau de Winnipeg, et nous analysons ensuite ceux d'Ottawa et de Montréal.

Les meilleures bornes inférieures pour la fonction économique obtenues au cours des essais grâce à l'équation (5.3) sont :

BLB =769 411 pour l'algorithme du simplexe convexe,

BLB =769 691 pour l'algorithme du gradient réduit,

BLB =769 676 pour l'algorithme du gradient projeté.

Nous pouvons les considérer comme solution optimale pour chacun des problèmes.

Le tableau (5.2) fournit les valeurs de la fonction économique en fonction du nombre d'itérations. A la lumière de ce tableau, nous remarquons que les temps d'exécution des trois méthodes sont proches avec un léger avantage pour la méthode du gradient réduit. Le temps d'exécution inclut le temps nécessaire pour la lecture du fichier NFF qui est de l'ordre de 0,43 s pour le réseau de Winnipeg. Le temps consommé par une itération est différent selon que nous sommes au début ou à la fin du traitement. Les premières itérations prennent plus de temps que les dernières compte tenu qu'au début toutes les paires O-D

Itération	Simp Convexe		Grad Réduit		Grad Projeté	
	F. Econ	CPU (sec)	F. Econ	CPU (sec)	F. Econ	CPU(sec)
1	796 833	4,72	796 538	4,75	796 873	4,78
5	771 082	21,37	771 066	19,47	771 078	20,75
10	770 224	36,65	770 219	32,75	770 191	36,02
20	770 190	65,16	770 172	57,08	770 093	62,53
30	770 190	93,56	770 168	79,57	770 050	88,28
40	770 190	121,89	770 168	99,58	770 006	113,14
50	770 190	150,33	770 167	121,31	769 993	138,14
60	770 190	178,66	770 167	143,95	769 990	161,01
70	770 190	206,85	770 167	163,23	769 983	185,26
80	770 190	235,23	770 167	184,98	769 983	210,10
Temps Moyen	2,94		2,31		2,63	

Tableau 5.2 – Valeurs de la fonction économique-Ville de Winnipeg

sont traitées, mais au fur et à mesure de l'augmenation du nombre d'itérations certaines satisfont les conditions d'optimalité et ne sont plus considérées dans le traitement.

Dans les trois algorithmes testés, la phase de recherche de plus court chemin consomme entre 50% à 65% du temps total d'exécution, contre 3% à 15% pour la phase de recherche unidimensionnelle. Le reste du temps est partagé entre les tâches telles que l'évaluation des coûts sur les arcs, le calcul des coûts des chemins et du gradient de la fonction objectif.

Au cours des premières itérations, la vitesse de convergence est plus rapide ; elle s'avère très lente près de l'optimum. Après un certain nombre d'itérations, la diminution de la fonction objectif n'est plus sensible. La méthode du simplexe convexe a convergé plus rapidement suivie de l'algorithme du gradient réduit, puis de l'algorithme du gradient projeté. Néanmoins, ce dernier fournit la valeur de la fonction économique la plus proche de l'optimum.

En ce qui concerne le calcul des écarts relatifs, nous présentons les résultats dans le tableau (5.3).

Itération	Simp Convexe		Grad Réduit		Grad Projeté	
	Ecart Rel1	Ecart Rel2	Ecart Rel1	Ecart Rel2	Ecart Rel1	Ecart Rel2
10	0,0009586	0,0011166	0,0008323	0,0010119	0,0007792	0,0009464
20	0,0008317	0,0010111	0,0005320	0,0006420	0,0004761	0,0005836
30	0,0008317	0,0010111	0,0005102	0,0006196	0,0004629	0,0004941
40	0,0008317	0,0010111	0,0005086	0,0006192	0,0003617	0,0004367
50	0,0008317	0,0010111	0,0005129	0,0006171	0,0003457	0,0004111
60	0,0008317	0,0010111	0,0005133	0,0006170	0,0003564	0,0004076
70	0,0008317	0,0010111	0,0005133	0,0006170	0,0003444	0,0003978
80	0,0008046	0,0009707	0,0005133	0,0006170	0,0003413	0,0003977

Tableau 5.3 – Ecarts relatifs -ville de Winnipeg-

De ce dernier tableau, nous retenons que l'écart est une bonne mesure de la décroissance de la fonction économique puisque l'algorithme du gradient projeté fournit les meilleures résultats en ce qui concerne la valeur de la fonction économique et au niveau de l'écart aussi.

La représentation de la fonction économique en fonction du nombre d'itérations (figure 5.1) fournit une idée de la qualité des directions de descente générées par les différents algorithmes. Sur ce point, les trois techniques s'équivalent avec un avantage pour la méthode du gradient projeté qui s'avère être plus efficace proche de l'optimun . Avec ce réseau, la méthode du gradient projeté a fournit un meilleur comportement, suivie de la méthode du gradient réduit et enfin le simplexe convexe.

Comme nous l'avons indiqué précédemment, la tolérance d'erreurs sur les coûts des chemins ϵ influe sur la qualité des solutions obtenues. Nous confirmons ce résultat avec le tableau (5.4) où nous comparons les valeurs de la fonction économique avec $\epsilon = 5\%$, $\epsilon = 0,5\%$ et $\epsilon = 0,05\%$ en utilisant la méthode du gradient projeté sachant qu'elle est la plus efficace.

Itération	F. Econ		
	$\epsilon = 5\%$	$\epsilon = 0,5\%$	$\epsilon = 0,05\%$
10	770 696	770 191	770 146
20	770 643	770 093	769 945
30	770 632	770 050	769 791
40	770 631	770 006	769 640
50	770 630	769 993	769 506
60	770 630	769 990	769 386
70	770 630	769 983	769 256
80	770 630	769 983	769 134

Tableau 5.4 – Valeurs de la fonction économique avec différentes valeurs de ϵ -ville de Winnipeg-

En outre, nous avons constaté que les approches du gradient réduit et projeté distribuent le flot sur un plus grand nombre de chemins. Au bout de 80 itérations, l'algorithme du simplexe convexe a généré 4 589 chemins, alors que les méthodes du gradient réduit et projeté ont généré resprectivement 6 454 et 6 368 chemins.

Puisque nous disposons aussi du programme EMME/2 que nous présentons brièvement ci-dessous, nous avons comparé les résultats obtenus dans notre travail avec ceux calculés avec EMME/2.

EMME/2(Multimodal Equilibrium) est un logiciel interactif-graphique servant à la planification des transports urbains et régionaux. C'est un ensemble d'outils intégrés permettant de définir, de valider et d'évaluer des scénarios de réseaux de transport multimodal. Il dispose de modèles d'affectation permettant de prévoir le nombre de passagers sur les liens,

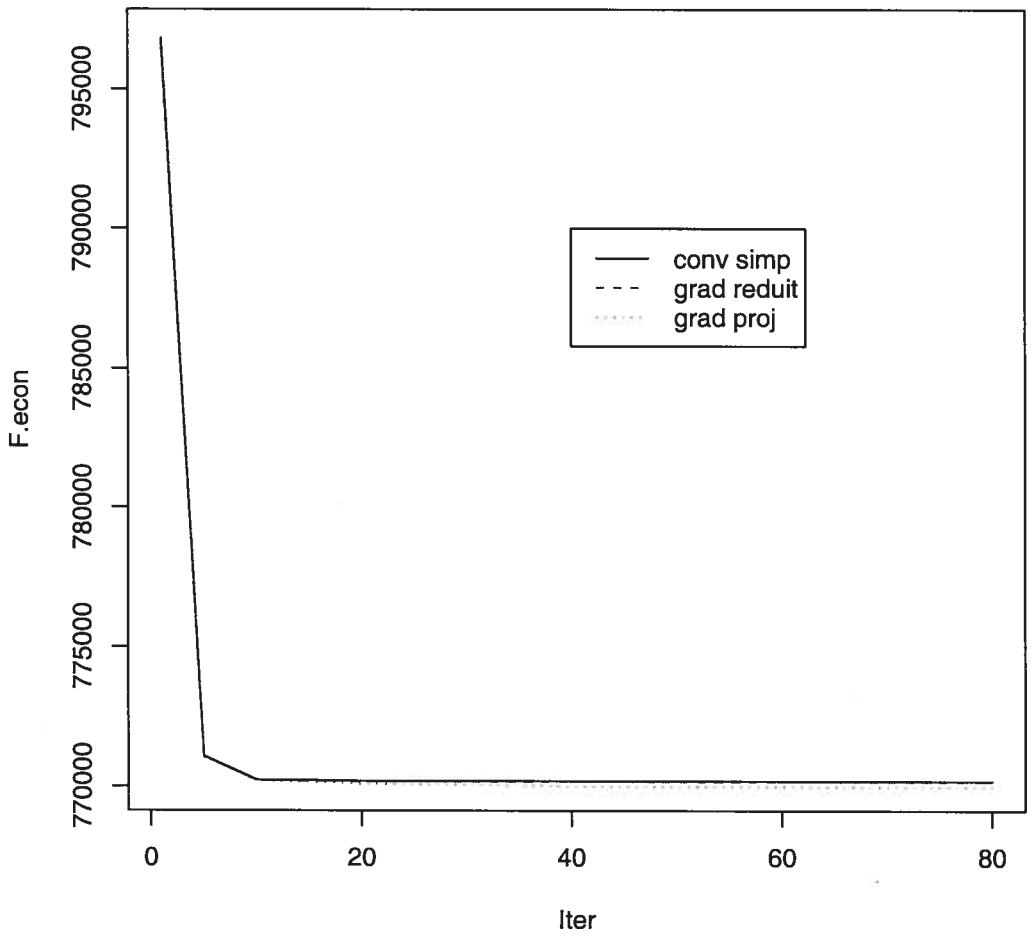

Figure 5.1 – graphe F.Eco/Itération -Réseau de Winnipeg

lignes de transport en commun...etc. Pour ces modèles, il comprend des procédures d'affectation d'équilibre du réseau routier à demande fixe ou variable avec pénalités de virage aux noeuds d'intersection. La définition et la validation des scénarios et l'analyse des résultats se font au moyen d'outils interactifs-graphiques. Il a été programmé en Fortran au Centre de recherche sur les transports de l'université de Montréal (CRT) et c'est l'un des logiciels les plus répandus dans le monde.

Nous avons constaté que nos méthodes résolvent le problème d'une façon sensiblement plus efficace. Avec EMME/2, nous avons obtenu une valeur de la fonction objectif égale à 770 288 après 112,7 secondes et au bout de 200 itérations, alors qu'avec les trois méthodes testées, nous avons obtenu des valeurs plus petites soient 770 231 pour la méthode du simplexe convexe, 770 219 pour la méthode basée sur le gradient réduit et 770 191 pour la méthode basée sur le gradient projeté après moins de 40 secondes et au bout de 10 itérations seulement.

5.2 Réseau d'Ottawa

Nous nous proposons maitenant d'examiner le comportement des trois méthodes lors de la résolution du problème d'équilibre sur le réseau d'Ottawa.

Les meilleures bornes inférieures calculées lors des essais sont :

BLB=2,01957e06, obtenue grâce à la méthode du simplexe convexe,

BLB=2,02012e06, obtenue grâce à la méthode du gradient réduit,

BLB=2,01999e06, obtenue grâce à la méthode du gradient projeté.

Nous les considérons comme solution optimale du problème.

Les tableaux (5.5) et (5.6) fournissent respectivement les valeurs de la fonction économique et des écarts relatifs en fonction du nombre d'itérations.

Itération	Simp Convexe		Grad Réduit		Grad Projeté	
	F. Econ	CPU (sec)	F. Econ	CPU (sec)	F. Econ	CPU(sec)
1	2,05083e06	18,46	2,05054e06	19,95	2,05075e06	17,59
5	2,02301e06	83,57	2,02302e06	80,58	2,02277e06	71,85
10	2,02098e06	143,34	2,02086e06	130,78	2,02074e06	126,87
20	2,02072e06	249,89	2,02055e06	230,17	2,02043e06	223,16
30	2,02066e06	355,53	2,02048e06	297,00	2,02038e06	331,79
40	2,02064e06	460,00	2,02046e06	379,46	2,02036e06	433,34
50	2,02063e06	565,86	2,02046e06	454,02	2,02035e06	533,84
60	2,02062e06	670,26	2,02045e06	528,56	2,02035e06	635,76
70	2,02061e06	774,37	2,02044e06	606,31	2,02034e06	733,22
80	2,02060e06	880,64	2,02044e06	682,88	2,02034e06	826,98
Temps Moyen	11,01		8,54		10,34	

Tableau 5.5 – Valeurs de la fonction économique-Ville d'Ottawa

Itération	Simp Convexe		Grad Réduit		Grad Projeté	
	écart Rel1	écart Rel2	écart Rel1	écart Rel2	écart Rel1	écart Rel2
10	0,0008179	0,0009821	0,0007263	0,0008686	0,0006476	0,0007740
20	0,0006594	0,0007460	0,0003220	0,0003872	0,0003900	0,0005294
30	0,0005756	0,0006879	0,0002766	0,0003342	0,0002965	0,0003475
40	0,0006172	0,0006591	0,0002508	0,0002725	0,0002663	0,0003172
50	0,0005957	0,0006525	0,0001816	0,0002181	0,0002860	0,0003069
60	0,0005265	0,0006101	0,0001415	0,0001719	0,0001702	0,0001745
70	0,0004994	0,0005865	0,0001319	0,0001588	0,0001640	0,0001728
80	0,0004904	0,0005799	0,0001361	0,0001575	0,0001460	0,0001726

Tableau 5.6 – Ecarts relatifs -ville d'Ottawa-

Sur la base de ces deux tableaux, nous remarquons qu'en ce qui concerne le temps de calcul, c'est la méthode du gradient réduit qui offre généralement les meilleurs résultats suivi

de près par la méthode du gradient projeté . Ce temps inclut la lecture du fichier NFF qui est de 1.18s pour le réseau d'Ottawa. La répartition du temps de calcul s'effectue d'une manière identique à celle du réseau de Winnipeg.

La méthode du gradient projeté a fournit la valeur de la fonction économique la plus proche de l'optimum. Les trois méthodes convergent pratiquement au même temps. Contrairement au réseau de Winnipeg, nous constatons que l'écart ne reflète pas une bonne mesure de la décroissance de la fonction économique puisque l'algorithme du gradient projeté fournit les meilleurs résultats en ce qui concerne la valeur de la fonction économique, mais au niveau de l'écart, c'est la méthode du gradient réduit qui offre un meilleur résultat.

Selon les valeurs des écarts obtenus, les trois méthodes ont donné un meilleur résultat avec le réseau d'Ottawa qu'avec le réseau de Winnipeg.

Nous pouvons également constaté sur la figure (5.2) que les trois méthodes ne se distinguent pas entre elles, les taux de convergence fournis par chacune d'elles sont pratiquement identiques, mais qu'au voisinage de l'optimum, les méthodes du gradient réduit et du gradient projeté ont un meilleur comportement.

Au bout de 80 itérations, l'algorithme du simplexe convexe a généré 6 978 chemins, les algorithmes du gradient réduit et du gradient projeté ont généré respectivement 10 145 et 7 712 chemins.

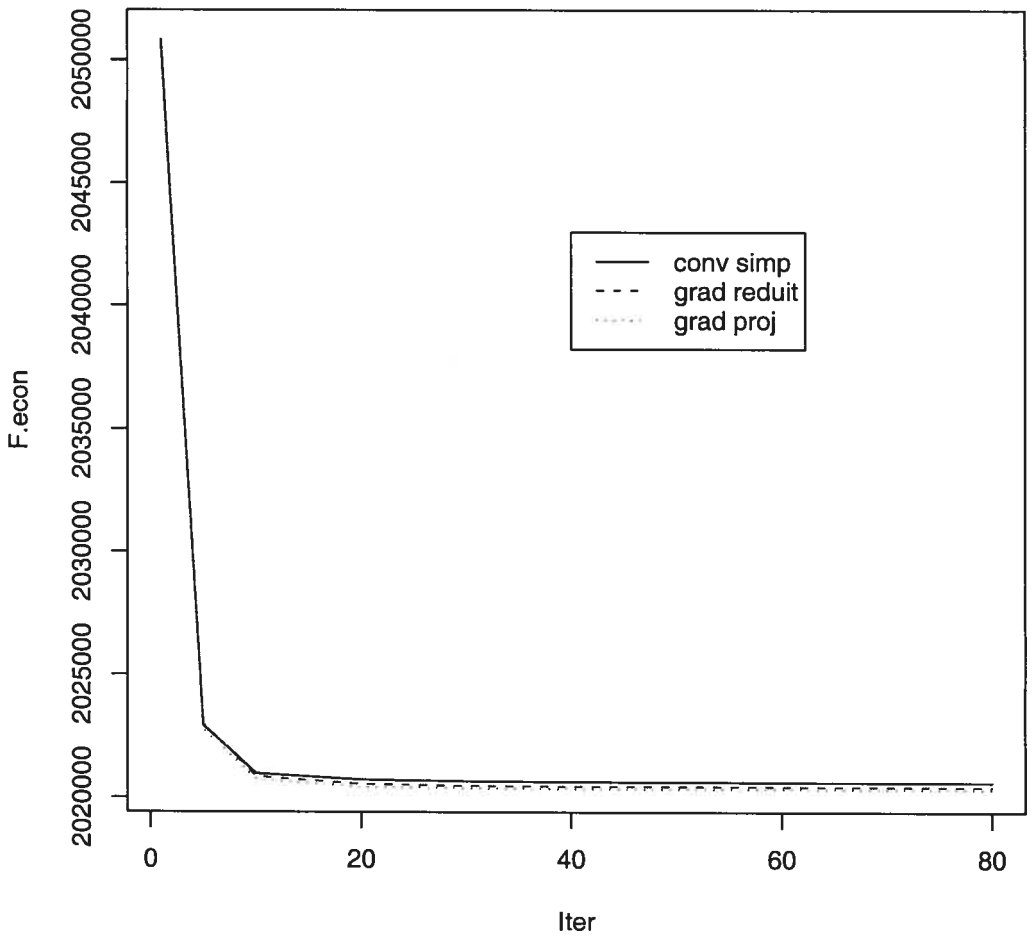

Figure 5.2 – graphe F.Eco/Itération -Réseau d'Ottawa

Quant à la comparaison avec le programme EMME/2, nous obtenons également un meilleur résultat pour ce réseau. Avec EMME/2, la valeur de la fonction économique atteinte au bout de 200 itérations et après 288 secondes est égale à 2,02127e06. Avec les trois méthodes testées, les valeurs des fonctions économiques sont respectivement 2,02109e06 pour l'algorithme du simplexe convexe au bout de huit itérations et après 132,2 secondes, 2,02122e06 pour l'algorithme du gradient réduit au bout de neuf itérations et après 114,56 secondes et 2,02109e06 pour l'algorithme du gradient projeté au bout de huit itérations et après 115,96 secondes seulement.

5.3 Réseau de Montréal

Pour réduire le temps d'exécution pour l'ensemble des tests réalisés sur la ville de Montréal, nous avons adopté la stratégie suivante :

- Commencer les premières itérations avec une tolérance d'erreur de 20% sur le coût des chemins.
- Diminuer cette tolérance à 10%, 5% et enfin 2% au fur et à mesure que les itérations augmentent.

Les meilleures bornes inférieures pour la fonction économique obtenues au cours des essais grâce à l'équation (5.3) sont :

BLB =3,25298e07 pour l'algorithme du simplexe convexe,

BLB =3,25924e07 pour l'algorithme du gradient réduit,

BLB =3,25873e07 pour l'algorithme du gradient projeté.

Itération	Simp Convexe		Grad Réduit		Grad Projeté	
	F. Econ	CPU (sec)	F. Econ	CPU (sec)	F. Econ	CPU(sec)
1	3,42544e07	214,63	3,42195e07	310,30	3,42172e07	276,30
5	3,29164e07	1000,84	3,27275e07	1235,26	3,27168e07	1342,41
10	3,26852e07	1829,92	3,26353e07	2112,31	3,26178e07	2373,78
15	3,26798e07	2688,46	3,26268e07	2863,56	3,26018e07	3287,45
20	3,26792e07	3536,57	3,26246e07	3586,06	3,25919e07	4140,85
Temps Moyen	176,83		179,30		207,04	

Tableau 5.7 – Valeurs de la fonction économique-Ville de Montréal

Le tableau (5.7) fournit les valeurs de la fonction économique ainsi que le temps CPU nécessaire pour y parvenir à différentes itérations.

La remarque faite quant à la convergence de la fonction économique tient toujours, puisque la méthode du gradient projeté fournit toujours la meilleure valeur de la fonction économique.

Contrairement aux réseaux précédents, l'algorithme du simplexe convexe est plus rapide en temps de calcul suivi de l'algorithme du gradient réduit et enfin l'algorithme du gradient projeté. Ce temps inclut également la lecture du fichier NFF qui est de 2,86 s pour le réseau de Montréal.

Le temps de calcul est réparti de la manière suivante : 40% à 50% pour le calcul des plus courts chemins, 10% à 16% pour le calcul du λ optimal. L'évaluation des coûts sur les arcs peut atteindre jusqu'à 10%.

Itération	Simp Convexe		Grad Réduit		Grad Projeté	
	écart Rel1	écart Rel2	écart Rel1	écart Rel2	écart Rel1	écart Rel2
5	0,0126749	0,016951	0,0056258	0,007510	0,0052936	0,0070652
10	0,0038772	0,005160	0,0013591	0,001806	0,0010558	0,0014051
15	0,0034255	0,004555	0,0008713	0,001156	0,0004080	0,0005478
20	0,0034035	0,004527	0,0007681	0,000989	0,0001063	0,0001408

Tableau 5.8 – Ecarts relatifs -ville de Montréal-

Sur le tableau (5.8) et la figure (5.3), nous pouvons constater que les méthodes du gradient réduit et projeté se détachent par rapport à la méthode du simplexe convexe sur la qualité des solutions et des directions de descentes générées quoique la méthode du gradient projeté demeure la meilleure.

Au bout de 20 itérations, l'algorithme du simplexe convexe a généré 35 603 chemins avec flots positifs sur l'ensemble des paires O-D, ceux du gradient réduit et projeté ont généré respectivement 81 437 et 64 572 chemins.

Comme la méthode du gradient projeté génére les meilleurs résultats, nous la retenons

pour la comparer avec EMME/2. Sur l'ensemble des tests effectués, nous avons obtenu le meilleur résultat en fixant le paramètre de tolérance ϵ à $0,5\%$ au lieu de 2% et en exécutant l'algorithme sur 50 itérations. Avec EMME/2, nous avons obtenu une valeur de la fonction économique équivalente à 3,24425e07 au bout de 2437,7 secondes et après 200 itérations. Avec l'algorithme du gradient projeté, nous avons obtenu 3,24171e07, mais au bout de deux heures de calcul.

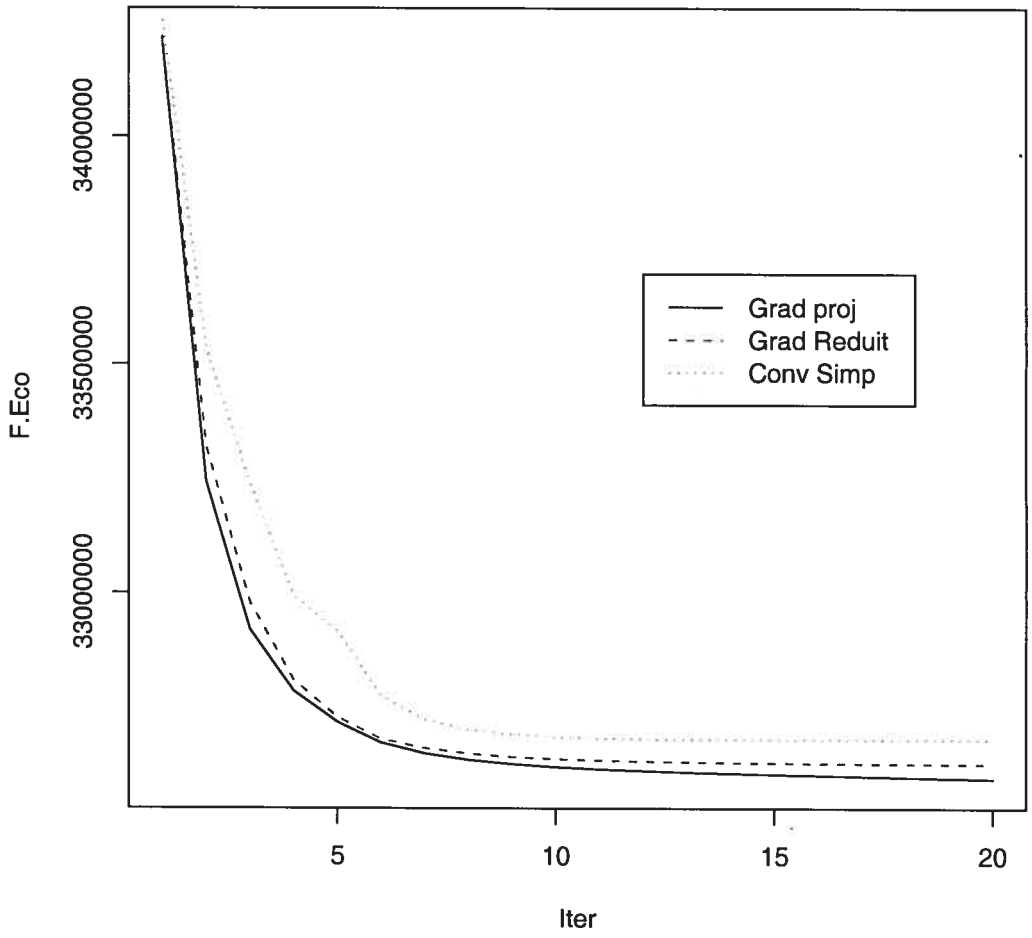

Figure 5.3 – graphe F.Eco/Itération -Réseau de Montréal

CONCLUSION

Dans ce mémoire, nous avons étudié trois méthodes d'affectation d'équilibre statique de réseaux à coûts symétriques avec demandes fixes dans l'espace des chemins. Elles sont basées sur une formulation d'optimisation et et sur les principes d'équilibre de Wardrop (1952).

Il s'agit de l'algorithme du simplexe convexe et des algorithmes des gradients réduit et projeté adaptés au problème d'affectation d'équilibre. Ils ont été développés dans le but de les comparer du point de vue vitesse de convergence, qualité d'affectation et temps de calcul avec les moyens informatiques actuels et sur des réseaux de tailles différentes. Nous avons implanté des algorithmes de plus courts chemins et de recherche linéaire. Pour le calcul du plus court chemin, nous avons utilisé l'algorithme de Spiess (1984) conçu initialement pour résoudre le problème d'affectation d'équilibre dans un réseau urbain à demande fixe avec pénalités de virages sur les noeuds d'intersection en utilisant une implantation avec monceau (heap). L'intérêt d'utiliser une implantation impliquant une technique d'extensions sélectives vient du fait qu'il est possible d'arrêter le calcul dès qu'une destination donnée est atteinte. Quant à la recherche linéaire, nous avons choisi la méthode d'interpolation quadratique par rapport à la méthode proposée par Guélat (1983) à cause de sa meilleure précision dans le calcul du pas optimal λ^* . Dans le cadre de de cette démarche, nous avons opté pour l'approche Gauss-Seidel. Elle converge plus rapidement que la variante Jacobi.

Nous avons programmé ces algorithmes en C++ sur des stations SUN, nous avons mesuré leurs performances numériques sur les réseaux des villes de Winnipeg, Ottawa et Montréal

et nous avons procédé à l'analyse des résultats en les comparant aux solutions calculées sur les mêmes réseaux par le programme EMME/2.

Nous avons utilisé le format NFF (Network File Format) pour exploiter le réseau. Ce format s'est avéré très facile pour présenter les différents éléments du réseau. Cependant, concernant le temps de calcul, nous avons constaté une lenteur dans l'exécution de certaines tâches telles que l'évaluation des coûts sur les arcs particuliérement pour un réseau de grande taille comme celui de Montréal.

Nous avons retenu les remarques suivantes :

- Pour un petit réseau de la taille de la ville de Winnipeg, les trois algorithmes ont des performances presque identiques. Cependant, la méthode du gradient projeté fournit un meilleur comportement, suivie de la méthode du gradient réduit et enfin la méthode du simplexe convexe.
- Pour des réseaux moyens et grands de la taille des villes d'Ottawa et Montréal, les algorithmes basés sur les gradients réduit et projeté génèrent des solutions et des directions de descentes de meilleure qualité que la méthode du simplexe convexe. La méthode du gradient projeté est plus performante près de l'optimum, mais son temps de calcul est un peu plus grand.

Quant à la comparaison avec EMME/2, nous avons remarqué que le comportement de nos méthodes est meilleur en termes de qualité des solutions et temps de calcul particuliérement pour les petits et moyens réseaux, tels que les réseaux des villes de Winnipeg et Ottawa. Avec un grand réseau tel que le réseau de la ville de Montréal, la qualité des solutions calculées est meilleure avec nos méthodes, cependant le temps de calcul exigé est plus important que pour EMME/2.

Comme suite à ce travail, il serait intéressant de considérer des fonctions coûts non séparables et asymétriques lors de la résolution du problème.

BIBLIOGRAPHIE

[1] Aashtiani H.Z., Magnanti T.L. (1981), "Equilibria on a congested transportation network", *SIAM Journal on Algebraic and Discrete methods*, 2, pp. 213-226.

[2] Auslender A. (1976), "Optimisation, méthodes numériques", Masson, Paris.

[3] Bertsekas D.P., Gafni E. (1982), "Projection methods for variational inequalities with application to the traffic assignment", *Mathematical Programming Study*, 17, pp. 139-159.

[4] Carey M. (1985), "The dual of the traffic assignment problem with elastic demands", *Transportation Research, B*,19, pp. 227-237.

[5] Chen A., Lee D.H. (1999), "Path-Based Algorithms for Large Scale Traffic Equilibrium Problems : a Comparison between DSD and GP", Paper presented at the 78th Transportation Research Board Annual Meeting, Washington DC.

[6] Dafermos S., Sparrow F.T. (1969), "The traffic assignment problem for a general network", *Journal of Research of the National Bureau of Standards, Series B, 73B(2),* pp. 91-118.

[7] Dafermos S. (1980), "Traffic equilibrium and variational Inequalities", *Transportation Science*, 14, pp. 42-54.

[8] Dafermos S. (2001), "An extended traffic assignment model with applications to two-way traffic", Department of computer science, Cornell University, Ithaca, New York.

[9] Denault L. (1994), "Etude de deux méthodes d'ajustement de matrices origine-destination à partir de flots des véhicules observés", Centre de recherche sur les transports, Université de Montréal, Publication CRT-1994-991.

[10] Dembo R.S (1987), "A primal truncated Newton algorithm with application to large-scale nonlinear network optimization", *Mathematical Programming Study*, 31, pp. 43-71.

[11] Dembo R.S., Klincewicz J.G. (1981), "A scaled reduced gradient algorithm for network flow problems with convex separable costs", *Mathematical Programming Study*, 15, pp. 125-147.

[12] Dembo R.S, Tulowitzki U. (1988), "Computing equilibria on large multicommodity networks : an application of truncated quadratic programming algorithms", *Networks*, 18, pp. 273-284.

[13] Florian M., Nguyen S. (1974), "A method for computing network equilibrium with elastic demands", *Transportation Science*, 8, pp. 321-332.

[14] Florian M., Nguyen S. (1976), "An application and validation of Equilibrium trip assignment method", *Transportation Science*, 10, pp. 374-390.

[15] Florian M. (1986), "Non linear cost network models in transportation analysis", Mathematical Programming Study, 26, pp.167-196.

[16] Florian M., Hearn D. (1996), "Network Equilibrium Models and Algorithms", *Handbooks in Operations Research and Management Science*, 8, pp. 485-550.

[17] Florian M., Grainic Theodor G. (2005), "Modèles d'optimisation pour la planification des systèmes de transport", Centre de recherche sur les transports, Université de Montréal, Publication CRT-2005-12.

[18] Fukushima M. (1984), "On the dual approach to the traffic assignment problem", *Transportation Research, B,* 18, pp. 235-245.

[19] Guelat J. (1983), "Algorithmes pour le problème d'affectation du trafic d'équilibre avec demandes fixes-Comparaisons", Centre de recherche sur les transports, Université de Montréal, Publication CRT-1983-299.

[20] Hearn D.W., Lawphongpanich S., Ventura J.A. (1987), "Restricted simplicial decomposition : computations et extensions", *Mathematical Programming*, 31, pp. 99-118.

[21] Hearn D.W., Lawphongpanich S. (1990), "A dual ascent algorithm for traffic assignment problems", *Transportation Research, B,* 24, pp. 423-430.

[22] Hillel B. (2002), "Origin Based Algorithm for the traffic assignment problem", *Transportation Science*, 36, pp. 398-417.

[23] Hohenbalken B.D. (1977), "Simplicial decomposition in nonlinear programming algorithms", *Mathematical Programming*, 13, pp. 49-68.

[24] Holloway C.A. (1974), "An extension of the Frank and Wolfe method of feasible directions", *Mathematical Programming*, 6, pp. 14-27.

[25] Kinderlehrer D., Stampacchia G. (1980), "An introduction to variational inequalities and applications", Academic Press, New York.

[26] Larsson T., Patriksson M. (1992), "Simplicial decomposition with disaggregated representation for the traffic assignment problem", *Transportation Science*, 26, pp. 4-17.

[27] Larsson T., Patriksson M., Rydergren C. (1997), "Applications of simplicial decomposition with non linear columm generation to non linear network flows", *Network optimization Economics and mathematical systems*, 450, pp. 346-373.

[28] Larsson T., Liu Zhuangwei., Patriksson M. (1997), "A Dual scheme for traffic assignment problems", Département de mathématique, Linkoping, Université de Suède.

[29] Larsson T., Patriksson M. (1999), "Side contrained traffic equilibrium models :analysis, computation and applications", *Transportation Research, B*, 33, pp. 233-264.

[30] LeBlanc L.J., Morlok E.K., Pierskalla W.P. (1975), " An efficient approach to solving the road network equilibrium traffic assignment problem", *Transportation Science*, 19, pp. 445-462.

[31] LeBlanc L.J., Helgason R.V., Boyce D.E. (1985), " Improved Efficiency of the Frank-Wolfe Algorithm for Convex Network Programs", *Transportation Research*, 9, pp. 309-318.

[32] Leventhal T., Nemhauser G., Trotter L. (1973), "A column generation algorithm for optimal traffic assignment", *Transportation Science*, 7, pp. 168-176.

[33] Luenberger G.D. (1984), "Linear and non linear programming", Addison-Wesley Publishing Compagny, second edition,

[34] Marcotte P. (1997), "Inéquations variationnelles : motivation, algorithmes de résolution et quelques app.lications.", Centre de recherche sur les transports, Université de Montréal, Publication CRT-1997-02.

[35] Minoux M. (1983), "Programmation mathématique théorie et algorithme-Tome 1". Dunod.

[36] Nguyen S. (1973), "Une approche unifiée des méthodes d'équilibre pour l'affectation du trafic", Département d'informatique, Université de Montréal, Publication 54.

[37] Nguyen S. (1974), "An Algorithm for the traffic assignment problem" , *Transportation Science 8*, pp. 203-216.

[38] Pang J.S., Yu C.S. (1984), "Linearized simplicial decomposition methods for computing traffic equilibria on networks ", *Networks, 14, pp. 427-438*

[39] Patriksson M. (1994), "The traffic assignment problem-Models and Methods", VSP, Utrecht.

[40] Rockafellar R.T. (1984), "Network flows and monotropic optimization", John Wiley, New York.

[41] Rosen J.B. (1960), "The gradient projection method for non linear programming, part I :linear contraints", *Journal of the Society of Industrial and Applied Mathematics*, 8, pp. 181-217

[42] Rosenthal R.W. (1973), "The network equilibrium problem in integers", *Networks*, 3 , pp. 53-59

[43] Rubio-Ardanaz J.M. (1995), "Un nouvel algorithme de décomposition simpliciale pour la résolution du problème d'équilibre de réseau à coûts symétriques et asymétriques", Centre de recherche sur les transports, Université de Montréal, Publication CRT-1995-20.

[44] Sender J.G., Netter M. (1970), "Equilibre offre-demande et tarification sur un réseau de transport", *Institut de Recherche des Transports*, Arcueil, France.

[45] Smith M.J. (1979), "The Existence Uniqueness and Stability of Traffic Equilibria", *Transportation Research, 13B*, pp. 295-304,

[46] Soumis F. (1978), " Planification d'une flotte d'avions", Centre de recherche sur les transports, Université de Montréal, Publication CRT-1978-133.

[47] Spiess H. (1984), "Contribution à la théorie et aux outils de planification des réseaux de transport urbain ", Centre de recherche sur les transports, Université de Montréal, Publication CRT-1984-382.

[48] Strustrup B. (1994), "The C++ programming language : Second Edition", Addison Wesley.

[49] Tremblay N. (1998), "Le calcul des plus courts chemins statiques et temporels : synthèse, implantations séquentielles et parallèles", Centre de recherche sur les transports, Université de Montréal, Publication CRT-1998-30.

[50] Wardrop J.G. (1952), "Some theoretical aspects of road traffic research", *Proc. Institute of Civil Engineers, Part II* 1, pp. 325-378.

[51] Weintraub A. (1974), "A primal algorithm to solve network flow problems with convex costs", *Management Science*, 21, pp. 87-97.

[52] Wolfe P. (1967), "Methods of Nonlinear Programming", Chapitre 6 de "Nonlinear Programming", J. Abadie, Interscience, John Wiley, New York, pp. 97-131.

Annexe A

Nous présentons la forme générale des fonctions de coût des réseaux de Winnipeg, Ottawa et Montréal. Ces fonctions sont séparables, continûment différentiables et strictement monotones.

Forme générale des fonctions de coût de la ville de Winnipeg :

$s_a(v_a) = D_a(\alpha_a + (\gamma_a \frac{v_a}{V_a})^{\beta}_a)$

où α_a, γ_a et β_a sont des coefficients constants.

D_a et V_a désignent repectivement la longueur de l'arc a et le nombre de voies de circulation sur l'arc a.

Forme générale des fonctions de coût de la ville d'Ottawa :

$s_a(v_a) = \sigma_a + 60\frac{D_a}{\chi_a}(1.0 + 0.2(\frac{v_a}{0.75V_a v_a})^{4.5})$

où σ_a, χ_a sont des coefficients constants.

D_a et V_a désignent repectivement la longueur de l'arc a et le nombre de voies de circulation sur l'arc a.

Forme générale des fonctions de coût de la ville de Montréal :

$s_a(v_a) = D_a\alpha_a(\beta_a+2+\sqrt{\gamma_a^2(1-(0.15^{\mu}_a va)/(\lambda_a V_a))^2+\beta_a^2}-\chi_a-\sigma_a(1-(0.15^{\mu}_a v_a)/(\lambda_a V_a))^2)+$
$D_a 0.0001 v_a/V_a$

où α_a, γ_a, β_a, χ_a, μ_a, λ_a, σ_a sont des coefficients constants.

D_a et V_a désignent repectivement la longueur de l'arc a et le nombre de voies de circulation sur l'arc a.

Annexe B
Code de l'agorithme du simplexe convexe

```
// SPT.h
#ifndef _SPT_H_
#define _SPT_H_

#include <NFF/NFF.h>
#define INFINITY 999999999
// ============================================================================
class SPTError: public CRTError {
 public:

   SPTError (const char* msg)
      : CRTError ("SPTError:")
   {
      _errorMessage += msg;
   }
};
// ----------------------------------------------------------------------------
class NFFTSPT {
 public:
   // Constructeur
   NFFTSPT ();
   // Destructeur
   ~NFFTSPT ();
   // affichage des resultats
   double* pathCost () const { return _pathCost; }
   double* bestTripCost () const { return _bestTripCost; }
   long* firstLink () const { return _firstLink; }
   long* nextLink () const { return _nextLink; }
   // Retour d'un  pointeur sur le reseau
   NFF* network () const { return _network; }
   // Caracteristiques du reseau
   int numberOfLinks () const { return _nbLinks; }
   int numberOfNodes () const { return _nbNodes; }
   int numberOfTurns () const { return _nbTurns; }
   int numberOfCentroids () const { return _nbCentroids; }
   // affectation des couts aux liens et virages
   void costLink (double* v);
   void costTurn ();
 protected:
   NFF* _network;
   NFFAttributeLong*     _idCentroid;
   NFFAttributeRef*      _nodeCentroid;
   NFFAttributeLong*     _idNode;
   NFFAttributeRef*      _centroid;
   NFFAttributeLong*     _idLink;
   NFFAttributeRef*      _fromNode;
```

```
    NFFAttributeRef*      _toNode;
    NFFAttributeRefList*  _outgoingLinks;
    NFFAttributeRefList*  _incomingLinks;
    NFFAttributeRefList*  _turnsAtNode;
    NFFAttributeFloat*    _length;
    NFFAttributeFloat*    _lanes;
    NFFAttributeFunction*  fct;
    NFFAttributeFunction* fct1;
    NFFAttributeRef*     _toLink;
    NFFAttributeRefList* _outgoingTurns;
    NFFEvalContext ctxt;
    // -------------------------------------------------------------
    static double* _costLink;
    static double* _costTurn;
    double* _bestTripCost;
    long*  _firstLink;
    long*  _nextLink;
    double* _pathCost;
    int*   _elementOfS;
    double* vps;
    int _nbLinks;
    int _nbCentroids;
    int _nbNodes;
    int _nbTurns;
    // -------------------------------------------------------------
    // initialisation du reseau
    void _setNetwork (NFF& network);
  private:
    // initialisation des attributs du reseau
    NFF* _setAttributes (NFF& network);
    // -------------------------------------------------------------
    // Copy constructor
    NFFTSPT (const NFFTSPT& tool);
    // Assignment
    NFFTSPT& operator= (const NFFTSPT& tool);
};
#endif
// ============================================================================
// FSPTHeapC.h
#include "SPT.h"
#include "Heap.h"
// ============================================================================
class NFFTFSPTHeapC: public  NFFTSPT  {
  public:
    // Contructeurs
    NFFTFSPTHeapC ();
    NFFTFSPTHeapC (NFF& network);
    // Destructeurs
    ~NFFTFSPTHeapC ();
    // initialisation du reseau
    void setNetwork (NFF& network);
    // calcul du plus court chemin
    void sp (const long centroidNumber,const long centroiddest);
  private:
    CRTHeap* _heap;
  // -------------------------------------------------------------
  // Copy constructor
    NFFTFSPTHeapC (const NFFTFSPTHeapC& tool);
  // Assignment
```

```
   NFFTFSPTHeapC& operator= (const NFFTFSPTHeapC& tool);
};
// ===========================================================================
//UpdatePathTurn.h
# include "SPT.h"
 struct path {
   long numPath;
   double hk;
   long arc;
   long virage;
   struct path* psuivant;
 };
 extern struct path** topPaths;
 class UpdatePathTurn: public NFFTSPT {
public :
  // Constucteur
     UpdatePathTurn();
     UpdatePathTurn(NFF& network);
  // Destructeurr
    ~UpdatePathTurn();
  // Initialisation des vecteurs de travail
     void setVects() ;
  // ajout d'un chemin actif a Kp+
     void setPath(long num,double  hk, long arc,long virage);
  // calcul  des couts des chemins.
     double costPath(struct path *tete);
  // calcul du chemin le moins couteux et le plus couteux de la paire O-D.
     void minMaxPath(struct path *tete);
  // Insertion du numero du virage dans la structure du chemin.
     void setTurn(long num,struct path* tete);
  // Mise a jour du volume dans les virages
     void updateVolumeTurn(struct path* tete,double hk1, double hk2);
  // calcul du chemin minimum.
     void minPath(struct path *tete);
  // calcul du nombre de chemins dans la paire traitÃ©e.
     long numberPath(struct path* tete);
  // retrait du chemin avec flot nul de l'ensemble des chemins actifs.
     void deletePathNul(struct path* top,long l);
//---------------------------------------------------------------------------
protected :
  // vp: vecteur de flots sur la paire(O-D) p.
     static double* vp;
  // vpp:vecteur de flots sur toutes les paires a l'exception de p
     static double* vpp;
     double* vv;
     static double* ya;
     double* y;
     double* yv;
     long* sigmak1;
     long* sigmak2;
     static double* v;
     double sk1,sk2;
     long double dk1;
     static long double dk2;
     struct path* pk1;
     struct path* pk2;
     struct path* tete;
     struct path* nouv;
     struct path* pcourant;
```

```
    struct path* ppcourant;
    struct path* deb;
 // ------------------------------------------------------------
 private:
  // Copy constructor
    UpdatePathTurn (const UpdatePathTurn& tool);
  // Assignment
    UpdatePathTurn& operator= (const UpdatePathTurn& tool);
};
//============================================================================
  // LineSearch.h
# include "UpdatePathTurn.h"
  class LineSearch: public UpdatePathTurn {
public :
  // Constucteur
    LineSearch();
    LineSearch(NFF& network);
  // Destructeur
    ~LineSearch();
  // calcul de la derivee de la fonction objectif pour un lambda donne
  // en fonction de la decomposition du probleme par pair O-D
    float fprim(float lambda,struct path *pk1, struct path *pk2 );
  // recherche du pas optimal.
    double   ls(struct path *pk1,struct path *pk2);
 // ------------------------------------------------------------
 private:
  // Copy constructor
    LineSearch (const LineSearch& tool);
  // Assignment
    LineSearch& operator= (const LineSearch& tool);
};
//============================================================================
//Cyclicdecomposition.h

#include <fstream.h>
#include "FSPTHeapC.h"
#include "LineSearch.h"
#include <CRT/Time.h>
extern CRTCPUTime chronometer;
extern long org,dest, _nbLinks,_nbTurn,numberPairs;
extern double d;
extern float opt;
extern int  nbTopPath;
extern struct path** topPaths;

class Cyclicdecomposition :public UpdatePathTurn {
public :
  // constructeur
    Cyclicdecomposition();
  // destructeur
    ~Cyclicdecomposition();
  // lecture du fichier NFF
    void readFileNFF();
  // initialisation des tableaux de travail.
    void setVectPath();
  // initialisation de la matrice des demandes .
    void setMatrixDemands();
  // rÃ©cupÃ©ration de la demande pour la paire O-D l.
    void demandOd(long l);
```

```
  // verification si la solution courante est optimale pour le sous probleme
     bool verifSubProblem(struct path* top,long l);
  // resolution du sous probleme l.
     void solveSubProblem(struct path* top,long double d,long l);
  // rÃ©cupÃ©ration du pointeur qui pointe sur le 1er chemin de la paire O-
     struct path* gettete();
  // mise a jour des vecteurs de flots
     void updatevp();
  // calcul de la fonction objectif.
     float  fonctObjectif();
  // calcul du cout total.
     float coutTotal();
  // calcul des gaps relatifs.
     float gapRelatif1();
     float gapRelatif2();
  // calcul de la somme des plus courts chemins des paires O-D par la demand
     float sumSpDemand();
  // calcul de la meilleure borne inferieure.
     void BLBC();
//-------------------------------------------------------------------------
  long topI,topJ;
  long* tab;
  double* valFonctObjectif;
  double* BLB;
  double* costLink;
  double* costTurn;
  double* best;
  long* first;
  long* next;
  NFFMatrixDouble* m;
  NFFTFSPTHeapC *ssp;
  LineSearch *ls1;
  NFF network;
 private:
  // Copy constructor
     Cyclicdecomposition (const Cyclicdecomposition& tool);
  // Assignment
     Cyclicdecomposition& operator= (const Cyclicdecomposition& tool);
};
//=========================================================================
// SPT.cc
#include "SPT.h"
double *NFFTSPT::_costTurn = 0;
double *NFFTSPT::_costLink = 0;
// Constructeur
NFFTSPT::NFFTSPT ()
   : _bestTripCost (0),
     _firstLink (0),
     _nextLink (0),
     _pathCost (0),
     _elementOfS (0)
{
}
// =========================================================================
// Destructeur
NFFTSPT::~NFFTSPT ()
{
   // libÃ©ration de la mÃ©moire.
   delete[] _costLink;
```

```
   delete[] _costTurn;
   delete[] _bestTripCost;
   delete[] _firstLink;
   delete[] _nextLink;
   delete[] _pathCost;
   delete[] _elementOfS;
   delete[] vps;
}
// ============================================================================
// _setAttributes
NFF* NFFTSPT::_setAttributes (NFF& network)
{
   // ------------------------------------------------------------
   NFFSection &centroids = network.sectionRef ("Centroids");
   _idCentroid = &(centroids.longAttribute("ID"));
   _nodeCentroid = &(centroids.refAttribute ("Node"));
   // ------------------------------------------------------------
   NFFSection &nodes = network.sectionRef ("Nodes");
   _idNode = &(nodes.longAttribute ("ID"));
   _centroid = &(nodes.refAttribute ("Centroid"));
   _incomingLinks = &(nodes.refListAttribute ("IncomingLinks"));
   _outgoingLinks = &(nodes.refListAttribute ("OutgoingLinks"));
   _turnsAtNode = &(nodes.refListAttribute ("TurnsAtNode"));
   // ------------------------------------------------------------
   NFFSection &links = network.sectionRef ("Links");
   _idLink = &(links.longAttribute ("ID"));
   _fromNode = &(links.refAttribute ("From"));
   _toNode = &(links.refAttribute ("To"));
   _length=&(links.floatAttribute("Length"));
   _lanes=&(links.floatAttribute("Lanes"));
    _outgoingTurns = &(links.refListAttribute ("OutgoingTurns"));
    fct=&(links.functionAttribute("LinkFunction"));
   // ------------------------------------------------------------
   NFFSection &turns = network.sectionRef ("Turns");
    _toLink = &(turns.refAttribute ("To"));
   fct1=&(turns.functionAttribute("TurnFunction"));
    // ------------------------------------------------------------
   _nbLinks = links.size();
   _nbNodes = nodes.size();
   _nbCentroids = centroids.size();
   _nbTurns = turns.size();
   return &network;
}
// ============================================================================
// _setNetwork
void NFFTSPT::_setNetwork (NFF& network)
{
   _network = _setAttributes (network);
   delete[] _costLink;
   delete[] _costTurn;
   delete[] _bestTripCost;
   delete[] _firstLink;
   delete[] _nextLink;
   delete[] _pathCost;
   delete[] _elementOfS;
   delete[] vps;
   _costLink = new double[_nbLinks];
   if (!_costLink)
      throw SPTError ("Cannot allocate _costLink memory");
```

```
    _firstLink = new long[_nbCentroids];
    if (!_firstLink)
       throw SPTError ("Cannot allocate _firstLink memory");
    _bestTripCost = new double[_nbCentroids];
    if (!_bestTripCost)
       throw SPTError ("Cannot allocate _bestTripCost memory");
    _nextLink = new long[_nbLinks];
    if (!_nextLink)
       throw SPTError ("Cannot allocate _nextLink memory");
    _pathCost = new double[_nbLinks];
    if (!_pathCost)
       throw SPTError ("Cannot allocate _pathCost memory");
    _elementOfS = new int[_nbLinks];
    if ( !_elementOfS )
       throw SPTError ("Cannot allocate _elementOfS memory");
   _costTurn = new double[_nbTurns];
    if (!_costTurn)
       throw SPTError ("Cannot allocate _costTurn memory");
    vps= new double[_nbLinks];
    if ( !vps )
       throw SPTError ("Cannot allocate vps memory");
    for (int i = 0; i < _nbLinks; i++)
      _costLink[i] = 0;
    for (int i = 0; i < _nbTurns; i++)
      _costTurn[i] = 0;
    for (int i = 0; i < _nbLinks; i++)
      vps[i] = -1;
}
// ===========================================================================
void   NFFTSPT::costTurn()
{
  for (int i = 0; i < _nbTurns; i++)
   _costTurn[i]=(*fct1)[i](ctxt);
 }
//============================================================================
void  NFFTSPT::costLink(double * v)
{
  for (int i = 0; i < _nbLinks; i++) {
   if (vps[i]!=v[i])
   _costLink[i]=(*fct)[i](ctxt,(*_length)[i],v[i],(*_lanes)[i]);
   vps[i]=v[i];
  }
}
//============================================================================
// FSPTHeapC.cpp
#include "FSPTHeapC.h"
// ===========================================================================
// Constructeurs
NFFTFSPTHeapC::NFFTFSPTHeapC ()
    :  NFFTSPT (),
       _heap (0)
{
}
// ---------------------------------------------------------------------------
NFFTFSPTHeapC::NFFTFSPTHeapC (NFF& network)
    :  NFFTSPT (),
       _heap (0)
{
  setNetwork (network);
```

```
}
// ============================================================================
// Destructeur
NFFTFSPTHeapC::~NFFTFSPTHeapC ()
{
 delete _heap;
}
// ============================================================================
// setNetwork
void NFFTFSPTHeapC::setNetwork (NFF& network)
{
   NFFTSPT::_setNetwork (network);
   delete _heap;
   _heap = new CRTHeap (_nbLinks);
   if (!_heap)
     throw SPTError ("Cannot create object _heap");
}
// calcul  du plus court chemin d'une origine vers une destination. .
void NFFTFSPTHeapC::sp (const long originNumber,const long centroiddest)
{
   // initialisation
  long origin = (*_nodeCentroid)[originNumber];
  for (register int  i = 0; i < _nbLinks; i++) {
      _elementOfS[i] = 0;
      _pathCost[i] = INFINITY;
   }

   for (register int  i = 0; i < _nbCentroids; i++ )
      _firstLink[i] = -1;
   int size = (*_incomingLinks)[origin].size();
   for (register int i = 0; i < size; i++)
      _elementOfS[(*_incomingLinks)[origin][i]] = 1;
   _bestTripCost[(*_centroid)[origin]] = 0;
   // insertion des liens sortants dans le monceau
   long link;
   size = (*_outgoingLinks)[origin].size();
   for (register int i = 0; i < size; i++) {
      link = (*_outgoingLinks)[origin][i];
      _pathCost[link] = _costLink[link];
      _nextLink[link] = -1;
      _heap->add (link, _pathCost[link]);
   }

   int nbCentroids = _nbCentroids-1;
   while (!_heap->empty()) {
    long linkMin = _heap->top();
     if (_elementOfS[linkMin])
      continue;
     double costMin = _pathCost[linkMin];
     long endLink = (*_toNode)[linkMin];
     if ((*_centroid)[endLink] != -1) {
        _bestTripCost[(*_centroid)[endLink]] = costMin;
        _firstLink[(*_centroid)[endLink]] = linkMin;
      if (centroiddest==-1) {
   if (!(--nbCentroids)) {
           if (_heap->size())
     _heap->flush();
      break;
    }
```

```
  }
  else if ((*_nodeCentroid)[centroiddest]==endLink) {
        if (_heap->size())
 _heap->flush();
  break;
}
 int size = (*_incomingLinks)[endLink].size();
 for (register int i = 0; i < size; i++)
    _elementOfS[(*_incomingLinks)[endLink][i]] = 1;
 }
 else {
   long nextLink;
   double  newCost;
    //  noeud regulier
      if ((*_turnsAtNode)[endLink].empty()) {
       int reverseLink = 0;
       int size = (*_outgoingLinks)[endLink].size();
       for (register int i = 0; i < size; i++) {
          nextLink = (*_outgoingLinks)[endLink][i];
       if (_elementOfS[nextLink])
 continue;
    // virage en U non autorise
       if ((*_toNode)[nextLink] == (*_fromNode)[linkMin]) {
          reverseLink = 1;
          continue;
       }
       newCost = costMin+_costLink[nextLink];
       if (newCost < _pathCost[nextLink]) {
           _pathCost[nextLink] = newCost;
           _nextLink[nextLink] = linkMin;
           _heap->add (nextLink, newCost);
           long linkMin = _heap->minValue();
        }
      }
        if (!reverseLink) {
           size = (*_incomingLinks)[endLink].size();
           for (int j = 0; j < size; j++)
              _elementOfS[(*_incomingLinks)[endLink][j]] = 1;
        }
     }
     else {
      // noeud avec virage
        long outgoingTurn;
        int size = (*_outgoingTurns)[linkMin].size();
        for (register int i = 0; i < size; i++) {
          outgoingTurn = (*_outgoingTurns)[linkMin][i];
          nextLink = (*_toLink)[outgoingTurn];
          if (_elementOfS[nextLink])
           continue;
           newCost = costMin+_costTurn[outgoingTurn]+
           _costLink[nextLink];
           if (newCost < _pathCost[nextLink]) {
              _pathCost[nextLink] = newCost;
              _nextLink[nextLink] = linkMin;
              _heap->add (nextLink, newCost);
           }
        }
     }
  }
```

```
      _elementOfS[linkMin] = 1;
   }

  }

//=============================================================================
// UpdatePathTurn.cpp
# include "UpdatePathTurn.h"
NFF* net;
long double UpdatePathTurn::dk2=0;
double* UpdatePathTurn::vp=0;
double* UpdatePathTurn::vpp=0;
double* UpdatePathTurn::ya=0;
double* UpdatePathTurn::v=0;
// constructeur
UpdatePathTurn::UpdatePathTurn (NFF& network)
   : NFFTSPT ()
{
     NFFTSPT ::_setNetwork (network);
}
UpdatePathTurn::UpdatePathTurn()
   : NFFTSPT ()
{
}
//destucteur
UpdatePathTurn::~UpdatePathTurn()
{
   // libÃ©ration de la mÃ©moire
  delete[] vp;
  delete[] vpp;
  delete[] v;
  delete[] vv;
  delete[] ya;
  delete[] y;
  delete[] yv;
  delete[] sigmak1;
  delete[] sigmak2;
}
// initialisation des vecteurs de travail
void UpdatePathTurn::setVects()
{     vp=new double[_nbLinks];
      vpp=new double[_nbLinks];
      v=new double[_nbLinks];
      ya= new double[_nbLinks];
      y=new double[_nbLinks];
      vv= new double[_nbTurns];
      yv= new double[_nbTurns];
      sigmak1= new long[_nbLinks];
      sigmak2= new long[_nbLinks];
      if (!vp)
throw SPTError("Cannot allocate vp Memory");
      if (!vpp)
     throw SPTError("Cannot allocate vpp Memory");
      if (!v)
     throw SPTError("Cannot allocate v Memory");
      if (!vv)
     throw SPTError("Cannot allocate vv Memory");
      if (!ya)
```

```
throw SPTError("Cannot allocate ya Memory");
      if (!y)
throw SPTError("Cannot allocate y Memory");
      if (!yv)
      throw SPTError("Cannot allocate yv Memory");
      if (!sigmak1)
throw SPTError("Cannot allocate sigmak1 Memory");
      if (!sigmak2)
throw SPTError("Cannot allocate sigmak2 Memory");
      for (register int i=0;i<_nbLinks;i++) {
        vp[i]=0;
vpp[i]=0;
        ya[i]=0;
        y[i]=0;
      }
      for (register int i=0;i<_nbTurns;i++){
        vv[i]=0;
        yv[i]=0;
      }
}
//---------------------------------------------------------------------------
void UpdatePathTurn::setPath(long num, double hk, long arc,long virage)
 {
      nouv=(path*)malloc(sizeof(struct path));
      nouv->psuivant=NULL;
      nouv->arc=arc;
      nouv->hk=hk;
      nouv->numPath=num;
      nouv->virage=virage;
 }
//---------------------------------------------------------------------------
 double UpdatePathTurn::costPath(struct path *tete)
{
  if (tete!=NULL) {
   ppcourant=tete;
   long num;
   double sk=0,costT=0;
   num=ppcourant->numPath;
   while (num==ppcourant->numPath) {
    // calcul du cout sur le virage .
    if (ppcourant->virage!=-1)
     costT=costT+_costTurn[ppcourant->virage];
    // calcul du cout sur les liens.
    sk=sk+_costLink[ppcourant->arc];
    ppcourant=ppcourant->psuivant;
    if (ppcourant==NULL) break;
   }
  return sk+costT;
  }
}
//---------------------------------------------------------------------------
void UpdatePathTurn::minMaxPath(struct path *tete)
{
   struct path* psauv;
   sk1=sk2=costPath(tete);
   pk1=pk2=tete;
   while(ppcourant!=NULL) {
     psauv=ppcourant;
     double sk=costPath(ppcourant);
```

```
      if (sk<sk1) {
       sk1=sk;
       pk1=psauv;
      }
      else if (sk>sk2) {
             sk2=sk;
             pk2=psauv;
      }
    }
  }
//-------------------------------------------------------------------------
void UpdatePathTurn::setTurn(long num,struct path* tete)
{
  struct path* ppcourant=tete;
  long numero=ppcourant->numPath;
  while (num!=numero) {
    ppcourant=ppcourant->psuivant;
    numero=ppcourant->numPath;
  }
  while (num==ppcourant->numPath){
   // verification de l'existence d'un virage sur le chemin
    long noeud=(*_toNode)[ppcourant->arc];
    if (!(* _turnsAtNode)[noeud].empty()){
     long size=(* _outgoingTurns)[ppcourant->arc].size();
     bool trouv=false;
     int i=0;
     while ((i<size)&&(!trouv)) {
       long virageSortant=(* _outgoingTurns)[ppcourant->arc][i];
       struct path* next=ppcourant->psuivant;
       if (next==NULL) break;
       if ((* _toLink)[virageSortant]==next->arc) {
          ppcourant->virage=virageSortant;
          trouv=true;
       }
       i=i+1;
      }
    }
    ppcourant=ppcourant->psuivant;
    if (ppcourant==NULL) break;
  }
 }
//-------------------------------------------------------------------------
void UpdatePathTurn::updateVolumeTurn(struct path *tete,double anchk, double
{
   struct path* ppcourant=tete;
   long num=ppcourant->numPath;
   while (num==ppcourant->numPath){
     if (ppcourant->virage!=-1)
       vv[ppcourant->virage]=vv[ppcourant->virage]-anchk+nouvhk;
     ppcourant=ppcourant->psuivant;
     if (ppcourant==NULL) break;
   }
}
//-------------------------------------------------------------------------
void UpdatePathTurn::minPath(struct path *tete)
{
   sk1=costPath(tete);
   while(ppcourant!=NULL){
     double sk=costPath(ppcourant);
```

```
     if (sk<sk1)
      sk1=sk;
   }

}
//-------------------------------------------------------------------------
 long UpdatePathTurn::numberPath(struct path *tete)
{
  long nb=0;
  struct path* pcourant=tete;
  while (pcourant!=NULL) {
    nb++;
    long num=pcourant->numPath;
    while (num==pcourant->numPath) {
      pcourant=pcourant->psuivant;
      if (pcourant==NULL) break;
    }
  }
  return nb;
}
//-------------------------------------------------------------------------
void UpdatePathTurn::deletePathNul(struct path* top,long l)
{
  pcourant=top;
  struct path* prec=pcourant;
  struct path* precedent;
  while (pcourant!=NULL) {
    if (pcourant->hk<=0)  {
     if (pcourant==top)  {
     top=pcourant->psuivant;
     topPaths[l]=top;
    }
    else
      prec->psuivant=pcourant->psuivant;
     precedent=pcourant;
     pcourant=pcourant->psuivant;
     free(precedent);
     }
    else  {
     prec=pcourant;
     pcourant=pcourant->psuivant;
   }
  }
}

//=========================================================================
// LineSearch.cpp
# include "LineSearch.h"

// constructeur
LineSearch::LineSearch()
   : UpdatePathTurn ()
{
}
 LineSearch::LineSearch(NFF& network)
   : UpdatePathTurn ()
{
 NFFTSPT::_setNetwork (network);
}
```

```
// destructeur
LineSearch::~LineSearch()
{
}
//-------------------------------------------------------------------------
float LineSearch::fprim(float lambda,struct path *pk1, struct path *pk2)
{
  float sumL=0,sumT=0;
  struct path *pcourant=pk1;
  long numero=pcourant->numPath;
  struct path *ppcourant;
  while (numero==pcourant->numPath) {
   v[pcourant->arc]=vpp[pcourant->arc]+vp[pcourant->arc]+lambda*ya[pcourant-
   if (v[pcourant->arc]<0)
    v[pcourant->arc]=0;
    sumL=sumL+((*fct)[pcourant->arc](ctxt,(*_length)[pcourant->arc],
    v[pcourant->arc],(*_lanes)[pcourant->arc]))*ya[pcourant->arc];
   if  (pcourant->virage!=-1)
    sumT=sumT+_costTurn[pcourant->virage];
   pcourant=pcourant->psuivant;
   if (pcourant==NULL) break;
  }
  pcourant=pk2;
  numero=pcourant->numPath;
  while (numero==pcourant->numPath){
   bool trouv=false, trouv2=false;
   ppcourant=pk1;
   long num=ppcourant->numPath;
   while ((!trouv)&&(num==ppcourant->numPath)){
     if (ppcourant->arc==pcourant->arc)
      trouv=true;
     ppcourant=ppcourant->psuivant;
     if (ppcourant==NULL) break;
   }
   ppcourant=pk1;
   while ((!trouv2)&&(num==ppcourant->numPath)){
    if ((ppcourant->virage!=-1)&& (pcourant->virage!=-1))
     if (ppcourant->virage==pcourant->virage)
      trouv2=true;
     ppcourant=ppcourant->psuivant;
    if (ppcourant==NULL) break;
   }
   if (!trouv) {
    v[pcourant->arc]=vpp[pcourant->arc]+vp[pcourant->arc]+lambda*ya[pcourant
    if (v[pcourant->arc]<0)
     v[pcourant->arc]=0;
     sumL=sumL+((*fct)[pcourant->arc](ctxt,(*_length)[pcourant->arc],
     v[pcourant->arc],(*_lanes)[pcourant->arc]))*ya[pcourant->arc];
    }
    if (!trouv2) {
     if  (pcourant->virage!=-1)
     sumT=sumT+_costTurn[pcourant->virage];
    }
    pcourant=pcourant->psuivant;
    if (pcourant==NULL) break;
   }
   return sumL+sumT;
}
//-------------------------------------------------------------------------
```

```
double  LineSearch:: ls(struct path *tete,struct path* pk2)
{
  double lambdaa=0,lambda1,opt;
  double lambdab=-pk2->hk/dk2, optmax= lambdab;
  int nbFit=0;
  if (fprim(lambdab,tete)<0)
   return lambdab;
  if (fprim(lambdaa,tete)>0)
   return 0;
  fit :
   lambda1=(lambdaa+lambdab)/2;
   nbFit++;
   double f23=(fprim(lambda1,tete)-fprim(lambdab,tete))/(lambda1-lambdab);
   double f13=(fprim(lambdaa,tete)-fprim(lambdab,tete))/(lambdaa-lambdab);
   double a=(f13-f23)/(lambdaa-lambda1);
   double b=f23-a*(lambda1+lambdab);
   double c= fprim(lambda1,tete)-lambda1*(b+a*lambda1);
   double discr=b*b-4*a*c;
   if (f23!=0)
     opt=lambdab-fprim(lambdab,tete)/f23;
   if ( discr>=0)
     opt=-2*c/(b+sqrt(discr));
   if (opt> optmax )
     opt=optmax;
   if  (opt< 0 )
     opt=0;
   if (fabs(fprim(opt,tete))<=0.01)
     return opt;
   else {
     if (fprim(opt,tete)<0) {
     lambdaa=opt;
     goto fit;
   }
   else {
     lambdab=opt;
     goto fit;
   }
  }
 }
// ========================================================================
// Cyclidecomposition.cpp
#include "Cyclicdecomposition.h"
CRTCPUTime chronometer;
long org,dest,numberPairs,_nbLinks,_nbTurns,sauvl=-1;
double d;
float opt;
int  nbTopPath,calcule=0;
struct path** topPaths;
double**  matriceDemands;
extern int nbIter;
extern int IterMaj;
extern double epsilon;
extern int  nbfois;
// constructeur de la classe cyclicDecomposition.
Cyclicdecomposition:: Cyclicdecomposition()
{
  readFileNFF();
  _setNetwork(network);
  ssp = new NFFTFSPTHeapC(network);
```

```
  ls1  = new LineSearch(network);
  ssp->costTurn();
}
//=============================================================================
//destructeur
Cyclicdecomposition::~Cyclicdecomposition()
{
  delete ssp;
  delete ls;
  for (register int j=0;j<numberPairs;j++){
   struct path* pcourant=topPaths[j];
   while (pcourant!=NULL) {
     struct path* prec=pcourant;
     pcourant=pcourant->psuivant;
     free(prec);
   }
  }
  delete[]tab;
  delete[] topPaths;
  delete[] valFonctObjectif;
  delete[] BLB;
  for ( int  i=0 ;i<numberPairs;i++)
   delete matriceDemands[i];
  delete[] matriceDemands;
}
//=============================================================================
 void Cyclicdecomposition::readFileNFF()
{
   ifstream fichierNFF (argv[1]);
   if (!fichierNFF) {
   cerr << "Impossible d'ouvrir le fichier "<< argv[1];
   exit(1);
}
   chronometer.start();
   fichierNFF >> network;
   chronometer.stop();
   cout << "Temps de lecture: " << chronometer.totalTime() << " sec" << endl

 // mise a jour des elements du reseau.
    NFFSection& nodes = network.sectionRef ("Nodes");
    NFFAttributeRef &centroid =nodes.createInternalRefs("Centroid","Centroic
    NFFSection& links = network.sectionRef ("Links");
    NFFSection& turns = network.sectionRef ("Turns");
    NFFSection& centroids = network.sectionRef ("Centroids");
    links.createInternalRefLists ("OutgoingTurns", "Turns", "From");
    nodes.createInternalRefLists ("OutgoingLinks", "Links", "From");
    nodes.createInternalRefLists ("IncomingLinks", "Links", "To");
    nodes.createInternalRefLists ("TurnsAtNode", "Turns", "At");
 // caracteristiques de la matrice des demandes.
    m = &(network.doubleMatrix ("demands"));
    _nbLinks=links.size();
    _nbTurns=turns.size();
    topI = m->indexes()[0]->size();
    topJ = m->indexes()[1]->size();
    numberPairs=m->size();
 }
//=============================================================================
//  lecture de la matrice des demandes a partir du fichier NFF.
 void Cyclicdecomposition::setMatrixDemands()
```

```
{
    double val;
    long k=0,indiceI=0,indiceJ;
    while (indiceI<=topI) {
      indiceJ=0;
      while (indiceJ<=topJ) {
       if ((val = m->get(indiceI,indiceJ)) != m->undefined()) {
        matriceDemands[k][0]=val;
        matriceDemands[k][1]=indiceI;
        matriceDemands[k][2]=indiceJ;
        k=k+1;
       }
      indiceJ++;
      }
      indiceI++;
    }
}
//=============================================================================
void Cyclicdecomposition::setVectPath()
{
   topPaths=new struct path* [numberPairs];
   if (!topPaths)
     throw SPTError("Cannot allocate topPath Memory");
   for (int i = 0; i <numberPairs; i++)
     topPaths[i]=NULL;
   matriceDemands= new double* [numberPairs];
   if (!matriceDemands)
     throw SPTError("Cannot allocate MatriceDemands Memory");
   for ( int  i=0 ;i<numberPairs;i++)
     matriceDemands[i]=new double [3];
   tab= new long[_nbLinks];
   if (!tab)
     throw SPTError("Cannot allocate tab Memory");
   valFonctObjectif= new double[nbfois];
   if (!valFonctObjectif)
     throw SPTError("Cannot allocate valFonctObjectif Memory");
   BLB= new double[nbfois];
   if (!BLB)
     throw SPTError("Cannot allocate BLB Memory");
   for (int i = 0; i <nbfois; i++){
     valFonctObjectif[i]=0.0;
     BLB[i]=0.0;
   }
 }
//-----------------------------------------------------------------------------
void Cyclicdecomposition::demandOd(long l)
{
      d=matriceDemands[l][0];
      org=matriceDemands[l][1];
      dest=matriceDemands[l][2];
}
//-----------------------------------------------------------------------------
struct path*  Cyclicdecomposition:: gettete()
{
  return tete;
}
//-----------------------------------------------------------------------------
void  Cyclicdecomposition:: updatevp()
{
```

```
  for (register int j=0;j<_nbLinks;j++)
    vpp[j]=vpp[j]+vp[j];
}
//-------------------------------------------------------------------------
 bool Cyclicdecomposition::verifSubProblem (struct path* top, long l)
{
   bool solve=false;
   if (top!=NULL) {
     pcourant=top;
     // recuperation du vecteur de flot vp pour la paire (O-D)p
     for (register int j=0;j<_nbLinks;j++)
       vp[j]=0;
     while (pcourant!=NULL) {
       vp[pcourant->arc]=pcourant->hk+vp[pcourant->arc];
       pcourant=pcourant->psuivant;
     }
     for (register int j=0;j<_nbLinks;j++)
       if (vpp[j]-vp[j]<10e-5)
        vpp[j]=0;
       else vpp[j]=vpp[j]-vp[j];

    // comparaison des couts des chemins
    pcourant=top;
    int nb=numberPath(top);
    if (nb>1){
      while ((pcourant!=NULL) && (!solve)) {
        double sk=costPath(pcourant);
        while ((ppcourant!=NULL) && (!solve)){
          double skc=costPath(ppcourant);
          if (skc<sk){
             if ((sk-skc)/skc>epsilon)
              solve=true;
          }
          else {
             if ((skc-sk)/sk>epsilon)
              solve=true;
          }
        }
        long num=pcourant->numPath;
        while (num==pcourant->numPath) {
          pcourant=pcourant->psuivant;
          if (pcourant==NULL) break;
        }
      }
    }
  // verification de l'existance d'un chemin plus court .
    if (!solve) {
      ssp->sp(org,dest);
      best=ssp->bestTripCost();
      minPath(top);
      int bes=(int)(best[dest]*1000.0 + 0.5);
      double bess=bes/1000.0;
      int bes1=(int)(sk1 *1000.0 + 0.5);
      double bess1=bes1/1000.0;
      if (bess < bess1) {
        solve=true;
        calcule=1;
        sauvl=l;
      }
```

```
     }
    }
  return solve;
}
//=============================================================================
void Cyclicdecomposition::solveSubProblem(struct path* top,long double d
,long l)
{
  //  calcul de la solution initiale.
  int cpt=-1;
  if (top==NULL) {
    for (register int j=0;j<_nbLinks;j++){
     vp[j]=0;
     tab[j]=0;
    }
    ssp->costLink(vpp);
    ssp->sp(org,dest);
    first= ssp->firstLink();
    best=ssp->bestTripCost();
    next=ssp->nextLink();
 // initialisation du vecteur de flot vp
    vp[first[dest]]=d;
    long a1=first[dest];
    while (next[a1]!=-1) {
      vp[next[a1]]=d;
      cpt=cpt+1;
      tab[cpt]=next[a1];
      a1=next[a1];
    }
// ajout du chemin a l'ensemble des chemins actifs.
   int j=0;
   if (cpt>=0) {
    while (cpt>=0) {
      setPath(1,d,tab[cpt],-1);
      if (j==0)
       tete=nouv;
      else
       pcourant->psuivant=nouv;
     pcourant=nouv;
     j=j+1;
     cpt=cpt-1;
    }
    setPath(1,d,first[dest],-1);
    pcourant->psuivant=nouv;
   }
   else  {
    setPath(1,d,first[dest],-1);
    tete=nouv;
   }
// mise a jour du flot dans les virages.
   setTurn(1,tete);
   updateVolumeTurn(tete,0,d);
   }
   else
    tete=top;
//=============================================================================
  // calcul des couts des chemins
 step1:
 minMaxPath(tete);
```

```
 if ((sk2-sk1)/sk1>epsilon) {
  dk1=sk2-sk1;
  dk2=sk1-sk2;
  // initialisation de sigmak1, sigmak2 et ya
  struct path* pcourant=pk1;
  long num=pcourant->numPath;
  for (register int i=0;i<_nbLinks;i++) {
    sigmak1[i]=0;
    sigmak2[i]=0;
  }
  while (num==pcourant->numPath) {
   sigmak1[pcourant->arc]=1;
   pcourant=pcourant->psuivant;
   if (pcourant==NULL) break;
  }
  pcourant=pk2;
  num=pcourant->numPath;
  while (num==pcourant->numPath) {
    sigmak2[pcourant->arc]=1;
    pcourant=pcourant->psuivant;
    if (pcourant==NULL) break;
  }
  for (register int j=0;j<_nbLinks;j++)
    ya[j]= sigmak1[j]*dk1+(sigmak2[j]*dk2);

// calcul du pas optimal
  double lambdaOpt=ls1->ls(pk1,pk2);
  pcourant=pk1;
  num=pcourant->numPath;
  double anchk,nouvhk;
  anchk=pcourant->hk;
  while (num==pcourant->numPath) {
    if (pcourant->hk+lambdaOpt*dk1<10e-5)
      pcourant->hk=0;
    else
      pcourant->hk=pcourant->hk+lambdaOpt*dk1;
    nouvhk=pcourant->hk;
    pcourant=pcourant->psuivant;
    if (pcourant==NULL) break;
  }
  updateVolumeTurn(pk1,anchk,nouvhk);
  pcourant=pk2;
  anchk=pcourant->hk;
  num=pcourant->numPath;
  while (num==pcourant->numPath) {
    if (pcourant->hk+lambdaOpt*dk2<10e-5)
      pcourant->hk=0;
    else
      pcourant->hk=pcourant->hk+lambdaOpt*dk2;
    nouvhk=pcourant->hk;
    pcourant=pcourant->psuivant;
    if (pcourant==NULL) break;
  }
  updateVolumeTurn(pk2,anchk,nouvhk);
  for (register int j=0;j<_nbLinks;j++)
    if (vp[j]+lambdaOpt*ya[j]<10e-5)
       vp[j]=0;
    else
       vp[j]=vp[j]+lambdaOpt*ya[j];
```

```
  }
//=============================================================================
// calcul du plus court chemin dans kp
   if ( (calcule!=1)|| (sauvl!=1))  {
      for (register int j=0;j<_nbLinks;j++)
        v[j]=vpp[j]+vp[j];
    ssp->costLink(v);
    ssp->sp(org,dest);
    best=ssp->bestTripCost();
   }
  first= ssp->firstLink();
  next=ssp->nextLink();
  for (register int j=0;j<_nbLinks;j++)
    tab[j]=0;
  calcule=0;
  long a1=first[dest];
  cpt=-1;
  while (next[a1]!=-1) {
    cpt=cpt+1;
    tab[cpt]=next[a1];
    a1=next[a1];
  }
//=============================================================================
//  ajout du chemin a kp+ s'il n'existe pas.
   minPath(tete);
   int bes=(int)(best[dest]*1000.0 + 0.5);
   double bess=bes/1000.0;
   int bes1=(int)(sk1 *1000.0 + 0.5);
   double bess1=bes1/1000.0;
   if (bess < bess1) {
  // verification si le chemin n'existe pas dans Kp+
     long num;
     bool trouv=false;
     pcourant=tete;
     while ((pcourant!=NULL) && (!trouv)) {
     int j=0, nb=cpt;
     num=pcourant->numPath;
     deb=pcourant;
     while (num==pcourant->numPath) {
       j=j+1;
       pcourant=pcourant->psuivant;
       if (pcourant==NULL) break;
     }
     if (j==cpt+2) {
        while ((tab[nb]==deb->arc) && (nb>=0)) {
          deb=deb->psuivant;
          nb=nb-1;
        }
        if ((nb<0)&& (deb->arc==first[dest]))
          trouv=true;
      }
     }
 // ajout du chemin a la fin de la structure.
    if (!trouv) {
      pcourant=tete;
      while (pcourant->psuivant!=NULL) {
        num=pcourant->numPath;
        pcourant=pcourant->psuivant;
      }
```

```
      num=num+1;
      while (cpt>=0) {
         setPath(num,0,tab[cpt],-1);
         pcourant->psuivant=nouv;
         pcourant=nouv;
         cpt=cpt-1;
      }
      setPath(num,0,first[dest],-1);
      pcourant->psuivant=nouv;
      setTurn(num,tete);
     }
     goto step1;
   }
    for (register int j=0;j<_nbLinks;j++)
      vpp[j]=vpp[j]+vp[j];
 }
//=============================================================================
  float  Cyclicdecomposition::coutTotal()
{
  float sumL=0,sumT=0;
  for (register int j=0;j<_nbLinks;j++)
   v[j]=0;
  for (register int j=0;j<numberPairs;j++){
   pcourant=topPaths[j];
   while (pcourant!=NULL) {
    v[pcourant->arc]=pcourant->hk+v[pcourant->arc];
    pcourant=pcourant->psuivant;
  }
   }
  for (int i=0;i<_nbLinks;i++)
   sumL=sumL+((*fct)[i](ctxt,(*_length)[i],v[i],(*_lanes)[i]))*v[i];
  for (int i=0;i<_nbTurns;i++)
   sumT=sumT+_costTurn[i](ctxt)*vv[i];
  return sumL+sumT;
}
//=============================================================================
// calcul de la fonction objectif avec la methode de Simpson
float  Cyclicdecomposition::fonctObjectif()
{
  float sumL=0,sumT=0;
  int m=4;
  float SumEven,SumOdd;
  for (int i=0; i<_nbLinks; i++) {
   SumEven=0;
  for (int k=1; k<=m-1;k++)  {
   v[i]=((vpp[i])/(2*m))*(2*k);
   SumEven=SumEven+(*fct)[i](ctxt,(*_length)[i],v[i],(*_lanes)[i]);
   }
   SumOdd=0;
   for (int k=1; k<=m;k++) {
   v[i]=((vpp[i])/(2*m))*(2*k-1);
   SumOdd=SumOdd+(*fct)[i](ctxt,(*_length)[i],v[i],(*_lanes)[i]);
 }
   v[i]=vpp[i];
   sumL=sumL+(v[i]/(6*m)*((*fct)[i](ctxt,(*_length)[i],0,(*_lanes)[i])+
   _costLink[i]+2*SumEven+4*SumOdd));
   }

  for ( int i=0; i<_nbTurns; i++) {
```

```
   SumEven=0;
  for (int k=1; k<=m-1;k++)
   SumEven=SumEven+(*fct1)[i](ctxt);
  SumOdd=0;
  for (int k=1; k<=m;k++)
   SumOdd=SumOdd+(*fct1)[i](ctxt);
  sumT=sumT+vv[i]/(6*m)*(2*_costTurn[i]+2*SumEven+4*SumOdd);
 }

  return sumL+sumT;
}

//=============================================================================
float Cyclicdecomposition::sumSpDemand()
{
  float som=0;
  ssp->costLink(v);
  long ancorg=-1;
  for (int i=0;i<numberPairs;i++){
   if (ancorg!=matriceDemands[i][1]) {
       ssp->sp(matriceDemands[i][1],-1);
       best=ssp->bestTripCost();
       ancorg=matriceDemands[i][1];
   }
   som=som+(matriceDemands[i][0]*best[(int)matriceDemands[i][2]]);
 }
 return som;
}

//-----------------------------------------------------------------------------
float Cyclicdecomposition::gapRelatif1()
{
     return (coutTotal()-sumSpDemand())/coutTotal();
}
//-----------------------------------------------------------------------------
float Cyclicdecomposition::gapRelatif2()
{
  float max=-1;
  for (int i=0;i<IterMaj;i++){
   if (BLB[i]>max)
   max=BLB[i];
  }
  return ( valFonctObjectif[IterMaj-1]-max)/valFonctObjectif[IterMaj-1];

}
//-----------------------------------------------------------------------------
void Cyclicdecomposition::BLBC()
{

  long ancorg=-1;
  for (register int j=0;j<_nbLinks;j++)
   y[j]=0;
  for (register int j=0;j<_nbTurns;j++)
   yv[j]=0;
  for (int i=0;i<numberPairs;i++) {
   if (ancorg!=matriceDemands[i][1]) {
    ssp->sp(matriceDemands[i][1],-1);
    best=ssp->bestTripCost();
    first= ssp->firstLink();
```

```
    next=ssp->nextLink();
    ancorg=matriceDemands[i][1];
   }
   int cpt=0;
   long a1=first[(int)matriceDemands[i][2]];
   y[a1]=matriceDemands[i][0]+y[a1];
   tab[cpt]=a1;
   while (next[a1]!=-1) {
    y[next[a1]]=matriceDemands[i][0]+y[next[a1]];
    cpt=cpt+1;
    tab[cpt]=next[a1];
    a1=next[a1];
   }
  while (cpt>=0) {
   long arc=tab[cpt];
   long noeud=(*_toNode)[arc];
   if (!(* _turnsAtNode)[noeud].empty())  {
     long size=(* _outgoingTurns)[arc].size();
     bool trouv=false;
     int i=0;
     while ((i<size)&&(!trouv)) {
     long virageSortant=(* _outgoingTurns)[arc][i];
     long next=cpt-1;
     if (cpt<0) break;
     if ((* _toLink)[virageSortant]==tab[next]) {
      yv[virageSortant]=yv[virageSortant]+d;
      trouv=true;
   }
   i=i+1;
  }
  }
   cpt=cpt-1;
 }
 }

 float sumL=0, sumT=0;
 for (int i=0;i<_nbLinks;i++)
 sumL=sumL+((*fct)[i](ctxt,(*_length)[i],vpp[i],(*_lanes)[i]))*(y[i]-vpp[i])
 for (int i=0;i<_nbTurns;i++)
  sumT=sumT+_costTurn[i]*(yv[i]-vv[i]);
 BLB[IterMaj-1]=  valFonctObjectif[IterMaj-1]+sumL+sumT;
}

//=============================================================================

# include "Cyclicdecomposition.h"
#include <CRT/Time.h>
int  nbfois,IterMaj;
double epsilon;

  main (int argc, char* argv[] )
{
 try {
  cout<<" Nombre d'iterations majeures :";
  cin>>nbfois;
  cout<< " Erreur admise sur les couts des chemins :";
  cin>>epsilon;
  nbTopPath=-1;
  long l=0, ll=0 ;
```

```
IterMaj=0;
Cyclicdecomposition cycle;
// Mise a jour des vecteurs de travail
cycle.setVectPath();
cycle.setVects();
// Mise a jour de la matrice des demandes
cycle.setMatrixDemands();

while ((ll!=numberPairs) && (IterMaj!=nbfois)) {
 l=l % (numberPairs)+1;
 struct path* top=topPaths[l-1];
 cycle.demandOd(l-1);
 bool solve=cycle.verifSubProblem(top,l);
 if ((!solve) && (top!=NULL)){
   ll++;
   cycle.updatevp();
 }
 else {
 // resolution du sous-probleme.
  cycle.solveSubProblem(top,d,l);

 // mise a jour du tableau topPaths
  if (topPaths[l-1]==NULL)
  topPaths[l-1]=cycle.gettete();
 // suppression des chemins avec du flot nul
  cycle.deletePathNul(cycle.gettete(),l-1);
  ll=0;
 }
 if (l==numberPairs) {
  // calcul de la fonction objectif et borne inferieure.
  IterMaj++ ;
  cycle.valFonctObjectif[IterMaj-1]=cycle.fonctObjectif();
  cycle.BLBC();
 }
}
}
  catch (CRTError& err)
  {
   cerr << err << endl;
   exit(1);
  }

}
//=============================================================================
```