2m11.3507.7

Université de Montréal

Simulation dynamique du trafic routier urbain et optimisation des contrôles

par

Lefong Hua

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

M. Sc.

en informatique

décembre, 2006

© Lefong Hua, 2006

QA
76
U54
2007
V.023

Direction des bibliothèques

AVIS

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

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

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

NOTICE

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

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

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

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Simulation dynamique du trafic routier urbain et optimisation des contrôles

Présenté par :

Lefong Hua

A été évalué par un Jury composé des personnes suivantes :

Jean-Yves Potvin
président-rapporteur

Michael Florian
directeur de recherche

Bernard Gendron
membre du jury

Mémoire accepté le 11 avril 2007

Sommaire

Le trafic routier est un problème qui prend de plus en plus d'ampleur au fil du temps et le besoin d'améliorer la fluidité de la circulation est devenu essentiel dans bien des villes. L'intérêt de ce travail est d'utiliser une nouvelle méthode d'interface pour fournir des données d'entrée à un logiciel d'optimisation des contrôles routiers. Dans le cadre de ce travail, nous nous sommes intéressés à Synchro et Dynameq.

Synchro est un outil commun d'optimisation des contrôles sur un réseau routier. Il peut ainsi produire des nouveaux plans pour réduire la congestion originale. Dynameq est un nouvel outil pour la modélisation temporelle d'un réseau routier. Il a comme rôle principal de simuler la congestion engendrée par la demande des véhicules sur un réseau.

Synchro et Dynameq sont de bons compléments. Synchro peut se servir des résultats d'une simulation de Dynameq pour optimiser les contrôles. Dynameq peut à son tour utiliser les nouveaux plans générés par Synchro pour lancer une nouvelle simulation. Nous pouvons ensuite comparer les résultats des deux différentes simulations de Dynameq et vérifier si Synchro trouve des bons plans pour diminuer la congestion originale. Le passage du réseau entre Dynameq et Synchro peut ainsi se faire sur plusieurs itérations afin d'en arriver à une solution optimale. Nous avons conçu un programme avec le langage de programmation Java, qui peut jouer le rôle d'interface entre les deux logiciels. Ce dernier permet ainsi de transférer un réseau de Dynameq à Synchro et de transférer des plans de Synchro à Dynameq.

En somme, le but de ce travail a été d'utiliser une approche itérative pour optimiser les plans d'un réseau. Nous avons testé cette dernière sur deux réseaux différents et avons réussi à diminuer la congestion de 24% dans le réseau de Calgary.

Mots clés : affectation dynamique du trafic, approche itérative, Dynameq, interface, intersection, mouvements de virage, optimisation des contrôles, plans, simulation dynamique, Synchro

Abstract

Road traffic is an ever growing problem and many cities have felt the need for improving the fluidity of circulation. The main objective of this study is to use a new method of interface to provide data to software that optimizes traffic lights timing. In this study, two different software were used: Synchro and Dynameq.

Synchro is a common tool for optimizing traffic lights on a road network. New traffic control plans can be produced to reduce an existing state of congestion. On the other hand, Dynameq is a new tool for the temporal modeling traffic on a road network. Its main application is to simulate the time-dependant flows and corresponding queues generated by the demand on the network.

Synchro and Dynameq complement each other. Synchro can use the results of a Dynameq simulation to optimize the traffic light. On the other hand, Dynameq can use the new plans generated by Synchro in a new simulation. The results of these two different Dynameq simulations can then be compared, to verify if Synchro has found the better plan that diminishes the initial state of congestion.

The road network can be transferred from Dynameq to Synchro on several iterations in order to obtain an optimal solution. Our work has been to realize a code, written in the Java language, which acts as is an interface between the two software. This interface allows the transfer of a network from Dynameq to Synchro and the transfer of the traffic control plans from Synchro to Dynameq.

In the end, the goal of this study is to use an iterative approach to optimize the plans of a network. We tested this iterative approach on two different networks, and managed to reduce the congestion by 24% in the Calgary network.

Key words: Dynameq, dynamic simulation, dynamic traffic assignment, iterative approach, interface, intersetcion, movement counts, optimisation of traffic lights, plans, Synchro.

Table des matières

Table des matières 1

Liste des figures 3

Liste des tableaux 6

Remerciements 8

Introduction 9

Chapitre 1 - Dynameq et ses fonctionnalités 13

1.1 Le réseau 13
1.2 Le plan de trafic 18
1.3 La demande 20
1.4 Modèle ADT 21
1.5 L'algorithme de solution 24
1.6 Résultats d'une simulation 27
1.7 Importation et exportation des fichiers d'entrée et de sortie 28

Chapitre 2 - Synchro et ses fonctionnalités 29

2.1 Le réseau 29
2.2 Les voies 30
2.3 Les volumes 31
2.4 Le temps 32
2.5 Les phases 34
2.6 Optimisation des contrôles 35
2.7 Lecture et écriture des fichiers d'entrée et de sortie 37
2.8 Différences majeures entre Synchro et Dynameq 38

Chapitre 3 - Interface entre Synchro et Dynameq 39

3.1 Généralités 39
3.2 Conversion des fichiers de Dynameq à ceux de Synchro 42
3.3 Conversion des fichiers de Synchro à celui de Dynameq 51
3.4 Exemple d'exécution 53
3.5 Limites du programme et améliorations futures 54

Chapitre 4 - Essais numériques 55

4.1 Réseau simple 55
4.2 Réseau de Calgary 67

Conclusion 83

Bibliographie 85

Annexe 1 – Les classes de l'interface entre Synchro et Dynameq 87

Annexe 2 – Des exemples de fichiers d'entrée et de sortie 122

Liste des figures

Figure 1.1 : Un réseau dans Dynameq..14

Figure 1.2 : Une fenêtre définissant les propriétés d'un lien dans Dynameq...........15

Figure 1.3 : Une fenêtre définissant les propriétés d'un nœud dans Dynameq........16

Figure 1.4 : Une fenêtre définissant les propriétés d'un véhicule dans Dynameq.....17

Figure 1.5 : Une fenêtre définissant les propriétés d'un plan de trafic dans Dynameq...19

Figure 1.6 : Une fenêtre d'une matrice de demande..20

Figure 1.7 : Le modèle ADT de Dynameq..26

Figure 1.8 : Les résultats d'une simulation de Dynameq...................................27

Figure 2.1 : Un réseau dans Synchro...29

Figure 2.2 : Une fenêtre des voies dans Synchro..31

Figure 2.3 : Une fenêtre des volumes dans Synchro..32

Figure 2.4 : Une fenêtre du temps dans Synchro...33

Figure 2.5 : Une fenêtre des phases dans Synchro..34

Figure 2.6 : Optimisation de la longueur d'un cycle dans Synchro.......................35

Figure 2.7 : Optimisation du décalage et du temps maximal dans Synchro............36

Figure 2.8 : Lecture et écriture des fichiers d'entrée et de sortie dans Synchro.......37

Figure 3.1 : Les angles d'un nœud dans Synchro...44

Figure 3.2 : Le numéro des voies à une intersection......................................46

Figure 4.1 : Réseau simple dans Dynameq...55

Figure 4.2 : Plans du réseau simple dans Dynameq......................................56

Figure 4.3 : Les mouvements des 4 intersections du réseau simple dans Dynameq..57

Figure 4.4 : Réseau simple dans Synchro, après une première simulation dans Dynameq..58

Figure 4.5 : Temps total de déplacement sous trois optimisations dans le réseau simple de Dynameq...63

Figure 4.6 : Temps total de déplacement sous les optimisations Cycle 1 à Cycle 4 dans le réseau simple de Dynameq..64

Figure 4.7 : Temps total de déplacement sous les optimisations Intuition 1 et 2 dans le réseau simple de Dynameq..65

Figure 4.8 : Temps total de déplacement sous toutes les optimisations dans le réseau simple de Dynameq...66

Figure 4.9 : Réseau de Calgary dans Dynameq..67

Figure 4.10 : Centre du réseau Calgary, dans Dynameq.................................68

Figure 4.11 : Les 4 phases de l'intersection 3412 du réseau de Calgary dans Dynameq..69

Figure 4.12 : Les 5 phases de l'intersection 3413 du réseau de Calgary dans Dynameq..........70

Figure 4.13 : Réseau de Calgary dans Synchro..........71

Figure 4.14 : Temps total de déplacement sous trois optimisations dans le réseau de Calgary de Dynameq..........76

Figure 4.15 : Temps total de déplacement sous les optimisations Feux à Feux 4 dans le réseau de Calgary de Dynameq..........77

Figure 4.16 : Temps total de déplacement sous les optimisations Feux à Feux 4 dans le réseau de Calgary de Dynameq..........78

Figure 4.17 : Temps total de déplacement des voitures sous toutes les optimisations dans le réseau de Calgary de Dynameq..........79

Figure 4.18 : Comparaison de la congestion de deux simulations à l'intersection 3412 du réseau de Calgary..........80

Figure 4.19 : Comparaison de la congestion de deux simulations à l'intersection 3413 du réseau de Calgary..........81

Liste des tableaux

Tableau 4.1 : Les 4 plans des intersections du réseau simple dans Dynameq.........57

Tableau 4.2 : Plan de trafic de la simulation Original dans le réseau simple de Dynameq.........59

Tableau 4.3 : Plan de trafic de la simulation Délai dans le réseau simple de Dynameq.........60

Tableau 4.4 : Plan de trafic de la simulation Feux dans le réseau simple de Dynameq.........60

Tableau 4.5 : Plan de trafic de la simulation Cycle 1 dans le réseau simple de Dynameq.........60

Tableau 4.6 : Plan de trafic de la simulation Cycle 2 dans le réseau simple de Dynameq.........61

Tableau 4.7 : Plan de trafic de la simulation Cycle 3 dans le réseau simple de Dynameq.........61

Tableau 4.8 : Plan de trafic de la simulation Cycle 4 dans le réseau simple de Dynameq.........61

Tableau 4.9 : Plan de trafic de la simulation Intuition 1 dans le réseau simple de Dynameq.........62

Tableau 4.10 : Plan de trafic de la simulation Intuition 2 dans le réseau simple de Dynameq.........62

Tableau 4.11 : Temps total de déplacement des voitures dans le réseau simple de Dynameq.........63

Tableau 4.12 : Le plan de l'intersection 3412 du réseau de Calgary dans Dynameq..69

Tableau 4.13 : Le plan de l'intersection 3413 du réseau de Calgary dans Dynameq..70

Tableau 4.14 : Les plans de Année Base...72

Tableau 4.15 : Les plans de Délai...72

Tableau 4.16 : Les plans de Feux...73

Tableau 4.17 : Les plans de Feux 1...73

Tableau 4.18 : Les plans de Feux 2...73

Tableau 4.19 : Les plans de Feux 3...74

Tableau 4.20 : Les plans deCycle...74

Tableau 4.21 : Les plans de Cycle 1...74

Tableau 4.22 : Les plans de Cycle 2...75

Tableau 4.23 : Temps total de déplacement des voitures dans le réseau de Calgary de Dynameq..76

Tableau 4.24 : Les plans de l'intersection 3412 dans Année Base et Feux 2..........80

Tableau 4.25 : Les plans de l'intersection 3413 dans Année Base et Feux 2..........81

Remerciements

Je tiens d'abord à exprimer ma reconnaissance à mon directeur de recherche, monsieur Michael Florian, pour m'avoir dirigé tout au long de mon travail. Ses précieux conseils m'ont initié aux problèmes d'optimisation de réseaux routiers et aux outils informatiques tels que Synchro et Dynameq. Je tiens également à le remercier pour son soutien financier.

Je tiens également à remercier Michael Mahut et Nicolas Tremblay, deux personnes qui m'ont aidé à me familiariser avec le logiciel Dynameq et ses subtilités.

Finalement, j'aimerais remercier mon collaborateur M. Pierre-Étienne Genest pour sa contribution à une partie de la programmation de ce travail, ainsi que pour sa documentation que j'ai traduite en français.

Introduction

Le trafic routier est un problème qui prend de plus en plus d'ampleur au fil du temps. Avec le coût élevé des infrastructures ainsi que la croissance continue du nombre de véhicules qui circulent dans les grandes régions urbaines, le besoin d'améliorer la fluidité de la circulation est devenu essentiel dans bien des villes.

Pour ce faire, il existe deux types d'interventions. Le premier est l'investissement dans les infrastructures. À Montréal, par exemple, il y a eu deux projets de ce type dans les dernières années, soient ceux à l'échangeur Rockland ainsi qu'à l'intersection de l'avenue des Pins et de l'avenue du Parc. Le deuxième type d'intervention est l'amélioration des contrôles routiers. Selon LCN (2006), il y a avait au mois d'août 2006 un projet en cours de 20 millions de dollars pour synchroniser des feux de circulation sur 804 intersections de la ville de Montréal. Les feux de circulation ont actuellement une seule programmation, mais après les modifications, ils auront plusieurs programmations, variant selon l'heure de la journée. Ceci a pour but de rendre la circulation beaucoup plus fluide et ce genre de projet est déjà présent dans bien d'autres villes en Amérique du Nord. C'est ce type d'intervention qui nous intéresse dans ce travail.

Depuis des années, différentes méthodes de simulation sur les choix de routes dans le but de modéliser la congestion sur un réseau ont été appliquées. Ces méthodes se divisent en deux grandes catégories : l'approche macroscopique et l'approche microscopique.

Selon Florian et Hearn (1995), l'approche macroscopique utilise des modèles statiques. Pour ce faire, on combine des modèles statiques avec des outils d'optimisation sur les choix de paramètres pour les intersections signalisées. On essaie donc de comprendre les effets du trafic sur le temps total de déplacement des voitures dans un réseau routier. L'approche macroscopique se base sur la demande de véhicules pour évaluer la congestion. Les véhicules sont regroupés et modélisés comme un ensemble et les résultats représentent une moyenne dans le temps. Dans le réseau, l'emphase est mise sur les flots de véhicules qui circulent sur des liens (une

rue qui relie deux intersections). Les premiers essais utilisant cette approche ont été réalisés dans les années 1970. Malheureusement, les résultats obtenus n'étaient pas très satisfaisants. Le problème principal est qu'on ne peut expliquer avec précision les causes de la congestion. La vitesse des véhicules est une fonction décroissante du flot seulement, sans considérer d'autres composantes. Une des composantes importantes de la congestion est la formation et la dissipation des files d'attentes dans un réseau, mais elles ne sont pas modélisées dans l'approche macroscopique. Dans le cas d'un réseau hyper congestionné, on pourrait donc retrouver des flots sur les liens du réseau qui dépassent la capacité pratique de ces derniers. Les mouvements de virage prédits sont alors irréalistes, ce qui met en doute la fiabilité de cette approche.

Selon Florian et Hearn (1999), l'approche microscopique utilise des modèles dynamiques. Avec les années, cette approche est devenue beaucoup plus populaire. Durant une simulation dynamique, chaque véhicule est modélisé individuellement et fait son propre choix de route, ce qui fait que la simulation est en temps réel. L'emphase est mise sur les intersections plutôt que sur les liens, ce qui permet de se concentrer sur les feux de circulation. Scherr, Adams et Bauer (2003) expliquent que le modèle dynamique peut fournir des prévisions sur les flots sur les liens ainsi que sur les virages aux intersections. Puisque les flots respectent strictement les capacités pratiques des liens du réseau, le modèle dynamique est plus approprié que le modèle statique. Aussi, le modèle dynamique permet d'expliquer les causes de la congestion avec plus de précision, car on a accès à des informations importantes comme les files d'attentes et le débordement arrière des voitures. Aujourd'hui, les modèles dynamiques les plus populaires sont représentés par des modèles de micro-simulation, tels que :

- AIMSUN2 (http://www.tss-bcn.com/)
- CORSIM (http://mctrans.ce.ufl.edu/featured/TSIS/Version5/corsim.htm)
- DRACULA (http://www.its.leeds.ac.uk/software/dracula/)
- MITSIM (http://mit.edu/its/mitsimlab.html/)
- PARAMICS (http://www.paramics-online.com/home/home.htm)
- VISSIM (http://www.vissim.com/)

L'intérêt de ce travail est d'utiliser une nouvelle méthode d'interface pour fournir des données d'entrée à un logiciel d'optimisation des contrôles routiers. De nombreux logiciels ont été conçus pour modéliser les effets temporels du trafic, les fils d'attente, etc... Dans le cadre de ce travail, nous nous sommes intéressés à Synchro et Dynameq.

Synchro est un outil commun d'optimisation des contrôles sur un réseau routier. Il est très répandu et représente un quasi standard dans la pratique de l'ingénierie du trafic. Il est utilisé partout en Amérique du Nord, incluant le service de circulation de la ville de Montréal. Pour optimiser les contrôles, il doit se baser sur une multitude d'informations sur un réseau. Ensuite, il peut produire des nouveaux plans pour réduire la congestion originale. En pratique, Synchro ne produit ces nouveaux plans qu'une seule fois, mais dans notre travail, nous vérifions l'optimalité de ces derniers.

Dynameq est un nouvel outil pour la modélisation temporelle d'un réseau routier. Il a comme rôle principal de simuler la congestion engendrée par la demande des véhicules sur un réseau. Après une simulation dans Dynameq, nous avons accès à de nombreuses informations sur la congestion du réseau, telles que les flots sur les liens, la longueur des files d'attentes aux intersections, le nombre de véhicules par mouvement de virage à chaque intersection, etc.

Chaque logiciel a des fonctions différentes qui peuvent être appliquées sur un même réseau. Les rôles joués par Sychro et Dynameq sont complémentaires. Synchro peut se servir des résultats d'une simulation par Dynameq pour optimiser les contrôles. Dynameq peut à son tour utiliser les nouveaux plans générés par Synchro pour faire une nouvelle simulation. On peut ensuite comparer les résultats de deux simulations de Dynameq et vérifier si Synchro trouve des bons plans pour diminuer la congestion originale. Le passage du réseau entre Dynameq et Synchro peut ainsi se faire sur plusieurs itérations afin d'en arriver à une solution qui réduit la congestion originale. Puisque les tailles des réseaux sont relativement grandes, il est impensable de convertir à la main un réseau entier conçu dans Dynameq pour Synchro, surtout pour un nombre élevé d'itérations.

Les deux logiciels sont munis de fonctions pour lire différents fichiers d'entrées sur les réseaux ainsi que sur les plans de ces derniers. Ils sont aussi munis de fonctions pour produire des fichiers de sorties sur les réseaux ainsi que les plans de ces derniers. Malheureusement, les fichiers de Dynameq et Synchro ne sont pas compatibles. Nous avons conçu un programme avec le langage de programmation Java, qui peut jouer le rôle d'interface entre les deux logiciels. En convertissant les fichiers de sorties d'un logiciel à des fichiers d'entrées de l'autre, ce dernier permet ainsi de transférer un réseau de Dynameq à Synchro ainsi que de transférer des plans de Synchro à Dynameq. De cette façon, les échanges d'informations d'un même réseau entre les deux logiciels peuvent être faits très rapidement.

En somme, le but de ce travail a été d'utiliser une approche itérative pour optimiser les plans d'un réseau. Nous avons testé cette dernière sur deux réseaux différents et avons réussi à diminuer la congestion de 24% dans le réseau de Calgary.

Au chapitre 1, nous présentons le logiciel Dynameq et ses principales fonctionnalités qui sont nécessaires à notre travail. Au chapitre 2, nous faisons de même avec le logiciel Synchro. Puis, il est question du cœur de notre travail au chapitre 3, où l'interface entre Synchro et Dynameq est expliquée en détail. Par la suite, nous analysons au chapitre 4 les essais numériques sur deux réseaux ainsi que leurs résultats. Finalement, nous discutons en conclusion la validité de l'approche itérative utilisée dans ce travail.

Chapitre 1 - Dynameq et ses fonctionnalités

Au fil des années, la modélisation de la congestion est devenue une priorité pour la planification du transport, avec le coût élevé des infrastructures ainsi que la croissance continue du nombre de véhicules circulant dans les grandes régions urbaines. L'affectation dynamique du trafic (ADT) peut simuler l'évolution temporelle du trafic sur un réseau et Dynameq est un outil de choix pour l'appliquer. Faisons maintenant un survol des éléments nécessaires à la simulation.

1.1 Le réseau

Le réseau de Dynameq est assez complexe. Il est constitué de liens (rues), de nœuds (intersections), de centroïdes et des véhicules qui y circulent. Faisons une analogie avec un graphe. Les nœuds sont donc des sommets et les liens des arcs orientés. Les centroïdes sont des nœuds qui ont la particularité d'être un point d'origine ou de destination d'un véhicule. Sur ce réseau, on y retrouve les feux de circulation, les mouvements causant des conflits aux intersections, la capacité du flot, les changements de voie et les véhicules multi-classes. Voici maintenant un exemple de réseau.

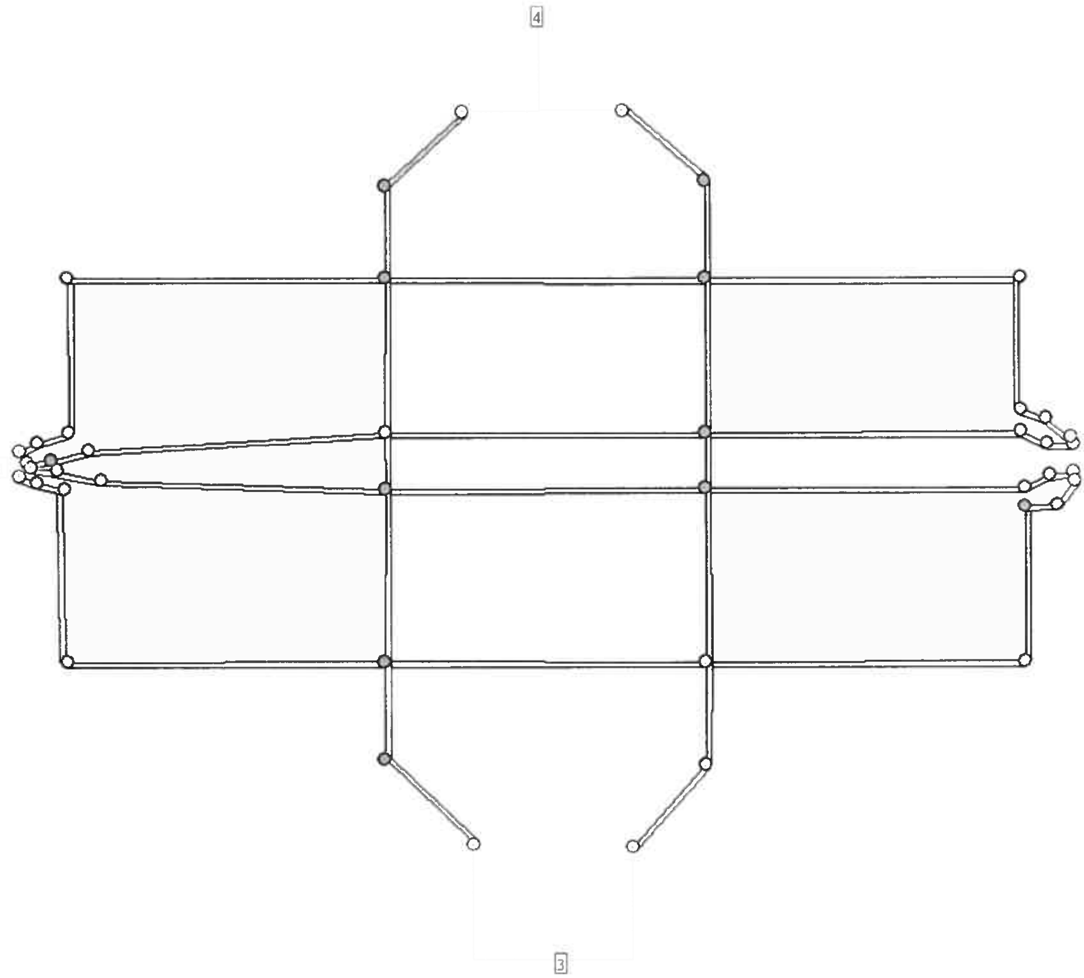


Figure 1.1 : Un réseau dans Dynameq

Sur la figure 1.1, il y a des liens, représentés par des arcs reliant deux points. Ensuite, il y a des nœuds, représentés par des points. Finalement, il y a des centroïdes, qui sont représentés par des points blancs.

1.1.1 Les liens

Un lien est une rue qui relie deux nœuds ou intersections. Ce lien a une seule direction et par conséquent, il peut y avoir un nombre maximum de deux liens qui relient deux nœuds. Puisque Dynameq utilise un modèle d'ADT, les voitures se déplacent individuellement sur les liens. Chaque lien est donc défini avec beaucoup d'informations utiles à la simulation : longueur du lien, nombre de voies, limite de vitesse permise, etc… De plus, il est possible de définir pour chaque voie les classes de véhicules pouvant y circuler. Dans le cadre de ce travail, cette dernière information n'est pas utile, car on se limite à une seule classe de véhicules par réseau. Avec toutes ces informations, il existe une relation entre le flot de véhicules sur un lien et la densité de voitures, en termes de véhicule/km/voie. Cette relation a un impact direct sur la congestion. Voici maintenant la fenêtre d'un lien de Dynameq.

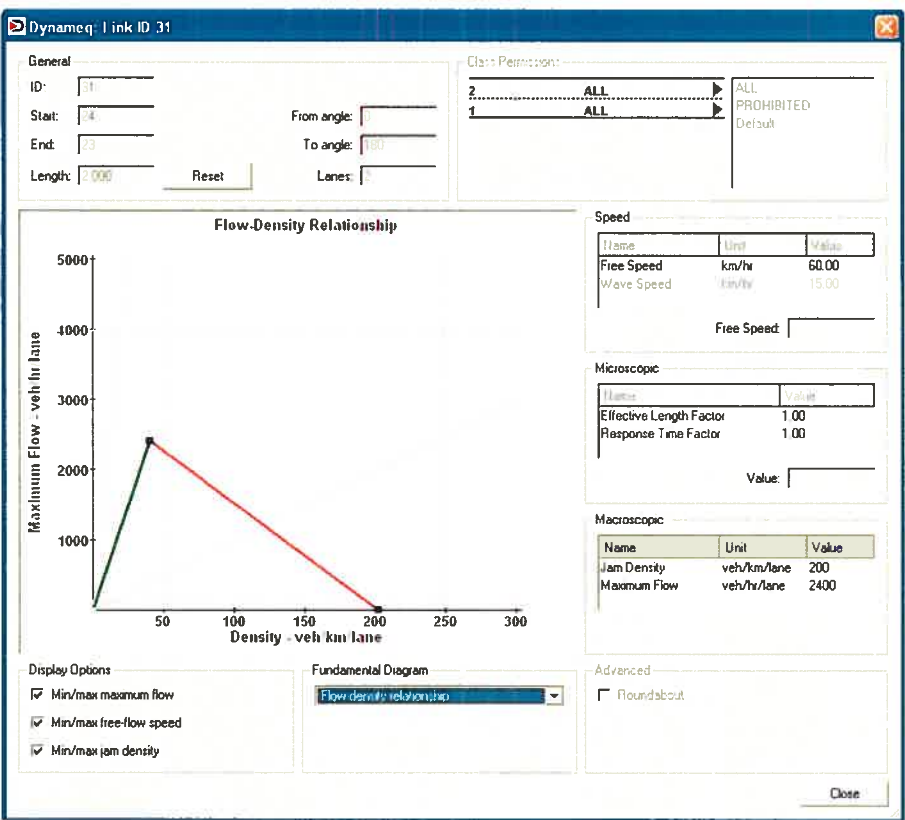


Figure 1.2 : Une fenêtre définissant les propriétés d'un lien dans Dynameq

1.1.2 Les nœuds

Un nœud est une intersection où différents liens se croisent. À un nœud donné, il existe des liens entrants (les véhicules arrivent à cette intersection) et des liens sortants (les véhicules quittent cette intersection). Un véhicule arrivant à une intersection exécute un mouvement. Ce dernier part d'un lien entrant et se dirige vers un lien sortant. Dans le cadre de ce travail, il existe trois mouvements: avancer tout droit, tourner à gauche et tourner à droite. Les intersections de plus de quatre liens entrants ou sortants sont ignorées et aucune optimisation ne sera faite sur le plan de trafic. Voici maintenant la fenêtre d'un nœud de Dynameq.

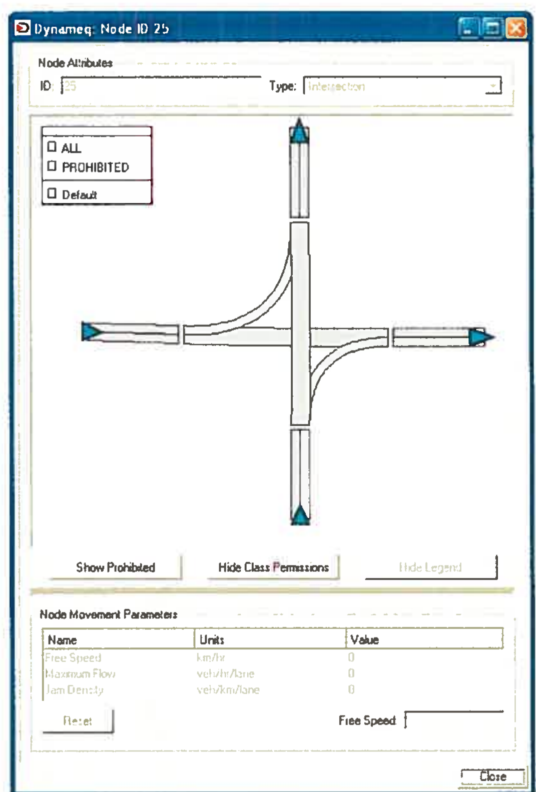


Figure 1.3 : Une fenêtre définissant les propriétés d'un nœud dans Dynameq

Sur la figure 1.3, il y a deux liens entrants et deux liens sortants. Les mouvements permis sont bien représentés, avec le nombre de voies réservées à chacun d'eux.

1.1.3 Les véhicules

Les véhicules circulant sur le réseau de Dynameq sont modélisés individuellement. Il est donc bien important d'avoir des informations précises sur ces derniers. Les paramètres des véhicules sont divisés en deux catégories : les paramètres physiques et les paramètres de la route. Les paramètres physiques sont la longueur et le temps de réponse du véhicule (temps de réaction du conducteur). Ces derniers définissent la densité de la congestion ainsi que la vitesse de vague négative (vitesse à laquelle la file d'attente se propage) associée à chaque classe de véhicule. Les paramètres de la route incluent l'identificateur de la classe de véhicule (non utile dans le cadre de ce travail), ce qui détermine pour chaque classe les voies et les virages qui peuvent être utilisés. Voici maintenant un exemple de fenêtre définissant les paramètres du véhicule.

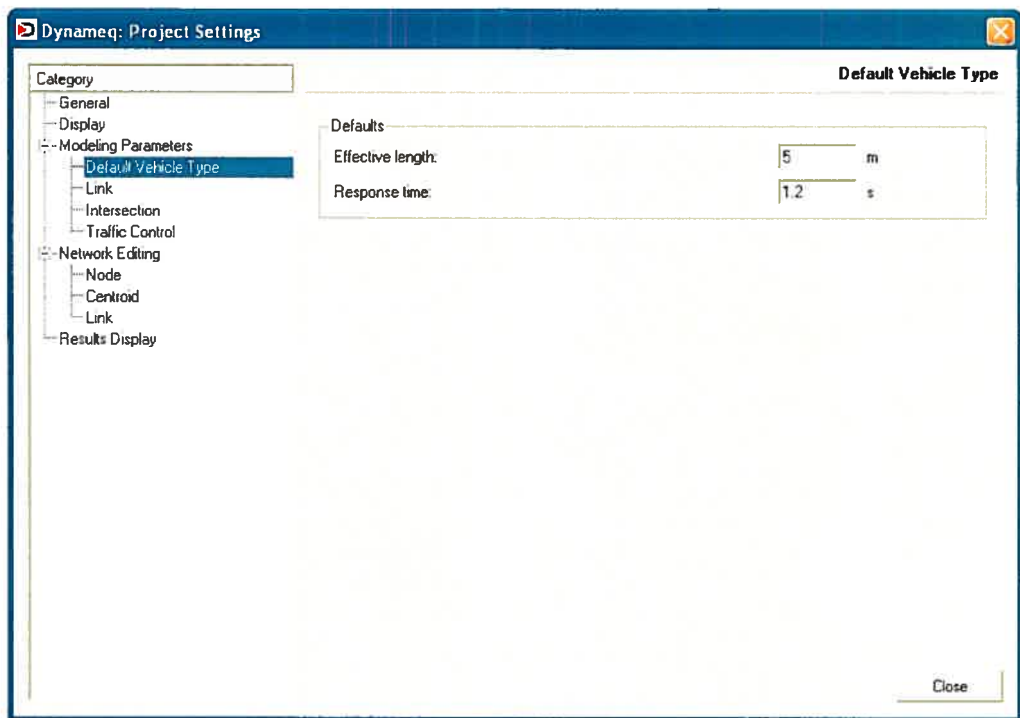


Figure 1.4 : Une fenêtre définissant les propriétés d'un véhicule dans Dynameq

Dans la partie droite de la figure 1.4, on peut modifier les paramètres physiques du véhicule : la longueur et le temps de réponse.

1.2 Le plan de trafic

À une intersection donnée, des véhicules provenant de différentes directions voulant exécuter des mouvements de virages peuvent entrer en conflit. C'est à cet endroit que les feux de circulation jouent un grand rôle en donnant à tour de rôle une priorité de mouvement aux véhicules. C'est le plan de trafic qui correspond à la boîte noire derrière les feux de circulation. Il est à noter qu'un nœud n'ayant pas de plan de trafic défini a tout de même un plan de trafic par défaut. Dans ce cas, un arrêt d'une petite durée est imposé à chaque véhicule.

Deux valeurs de temps sont associées au plan de trafic, soient la longueur du cycle et le délai. La longueur du cycle correspond à la durée en secondes du plan. Le délai correspond au nombre de secondes à partir du début de la simulation avant que le plan se mette en branle.

Le plan de trafic contient aussi des phases. Une phase contient un ensemble de mouvements de virage permis et une durée de temps leur est allouée. Les véhicules exécutant ces mouvements ont donc priorité sur les autres, qui attendent leur tour. À chaque phase est attribué un temps de feu vert où les mouvements sont permis, de feu jaune qui correspond aux dernières secondes où les mouvements sont permis et de feu rouge où tous les mouvements à cette intersection sont interdits. Les informations nécessaires à chaque phase incluent le nombre de voies associées à chaque mouvement de virage, ainsi que les voies utilisées pour exécuter ce virage.

Voici maintenant un exemple d'une fenêtre définissant les propriétés d'un plan.

Figure 1.5 : Une fenêtre définissant les propriétés d'un plan de trafic dans Dynameq

Sur la figure 1.5, les mouvements permis sont illustrés dans la partie de droite. Dans la partie médiane gauche, on y trouve les informations sur le plan de trafic, comme la longueur du cycle ainsi que le délai. Dans la partie inférieure gauche, on y trouve les différentes phases ainsi que la durée de temps de feu vert, jaune et rouge pour chacune d'elles.

1.3 La demande

La demande correspond à l'ensemble des véhicules partant de points d'origine allant vers des points de destination. Une matrice de demande contient le flot de véhicules pour chaque paire de centroïdes origine-destination (O-D). Aussi, elle détermine à quelle heure commence et termine la demande. Il est possible de fixer la demande par intervalle de temps. Dans ce cas, le taux de flot de véhicules varie. Voici maintenant un exemple d'une fenêtre d'une matrice de demande.

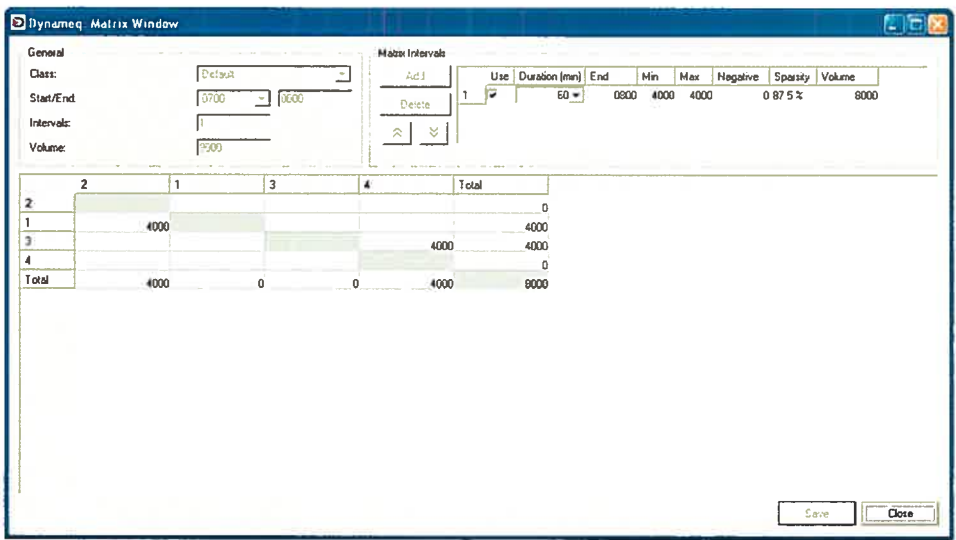


Figure 1.6 : Une fenêtre d'une matrice de demande

Sur la figure 1.6, il y a la fenêtre qui permet de modifier les paramètres de la demande. Dans la partie supérieure gauche, on peut fixer l'heure de début et de fin, ainsi que le nombre d'intervalles. Dans la partie du bas, on peut fixer le nombre de véhicules pour chaque paire O-D.

1.4 Modèle ADT

Les 2 points suivants (1.4 et 1.5) s'inspirent grandement du travail de Florian, Mahut et Tremblay (2005).

Dynameq est un modèle ADT basé sur le principe d'équilibre dynamique descriptif. Durant la simulation, l'objectif est de minimiser le temps de déplacement de chaque véhicule. Cet objectif est atteint lorsque pour chaque paire O-D, tous les véhicules partant d'une origine donnée et allant à une destination donnée prennent environ le même temps pour parcourir ce trajet. Le modèle de chargement du réseau utilisé dans Dynameq a été conçu par M. Mahut (2000).

Pour ce faire, Dynameq utilise une méthode itérative. Chaque itération consiste en une exécution d'un modèle choix de route et en une exécution d'une simulation du trafic. Un choix de route est donc assigné à chaque véhicule et après chaque itération, les choix de routes peuvent être modifiés pour obtenir une meilleure solution. Le processus se répète jusqu'à ce que le tout converge vers l'équilibre dynamique descriptif.

Avant d'entrer dans les détails du modèle, faisons une brève analogie afin de bien comprendre ce qui se passe. Considérons une simple itération comme une journée. Lors de la première journée, les conducteurs prennent connaissance des composantes du réseau (les nœuds, les liens et les contrôles). Par contre, ils ne connaissent pas la demande, ni la congestion qui sera engendrée par cette demande. Ils font donc un choix de chemin et subissent ensuite la congestion en cours de route. Ensuite, pour les journées suivantes, chaque conducteur considère la possibilité de choisir un chemin différent, en tenant compte de la congestion qu'il a subie. Après un certain nombre de jours, lorsqu'il y a convergence, les conducteurs arrêtent tous de considérer un nouveau chemin et gardent le leur pour les journées à venir.

Deux approches différentes sont généralement employées pour évaluer la façon dont les conducteurs choisissent leur chemin: l'affectation dynamique en route et l'affectation dynamique d'équilibre. Cette dernière s'assure que pour chaque période de temps, les chemins utilisés soient environ de la même longueur. Dans ce qui suit,

l'approche adoptée est de chercher une solution approximative aux conditions d'équilibre dynamique.

Dans le problème d'affectation dynamique d'équilibre, un conducteur fait un choix de chemin avant son départ et ne change pas son choix en cours de route. Chaque conducteur cherche donc à minimiser son temps de déplacement avec son choix de chemin. Les choix de chemin sont donc modélisés comme une variable de décision. Avant de faire leurs choix, les conducteurs ont accès à l'information du réseau, qui inclut les temps de déplacement sur tous les chemins, qu'ils soient utilisés ou non. Pour converger à ces conditions d'optimalité, l'algorithme de résolution prend donc la forme d'un procédé itératif.

L'approche adoptée pour résoudre le modèle dynamique d'équilibre (1)-(3) est basée sur une discrétisation temporelle en périodes de temps $\tau = 1, 2, \ldots, \left|\frac{T_d}{\Delta t}\right|$, où Δt est la durée choisie d'un intervalle de temps. Ces résultats se retrouvent dans un modèle de temps discret.

La formulation du modèle mathématique du problème dynamique d'équilibre est dans l'espace des flots des chemins $h_k(t)$, pour tous les chemins k appartenant à l'ensemble K_i pour une paire O-D i, $i \in I$, au temps t. Les taux de demandes variants avec le temps sont notés $g_i(t)$. Les taux des flots des chemins dans la région réalisable Ω satisfont les contraintes de conservation de flot et de non négativité pour $t \in T_d$, où $(0, T_d)$ est la période durant laquelle la demande temporelle est définie. Ceci est

$$\Omega = \left\{ h(t) : \sum_{k \in K_i} h_k(t) = g_i(t), i \in I; h_k(t) \geq 0 \right\} \qquad \forall t \in T_d, \qquad (1)$$

et la version temporelle du choix de route optimal de Wardop (1952) sont dans le modèle :

$$h_k \in \Omega, u_i(t) = \min_{k \in K_i} \{ s_k(t) \}$$

$s_k^{\tau} = u_i^{\tau}$ si $h_k^{\tau} > 0$

$$s_k^{\tau} \geq u_i^{\tau} \text{ si } h_k^{\tau} = 0 \qquad \forall\ k \in k_i,\ i \in I, \tau = 1, 2, \ldots, \left|\frac{T_d}{\Delta t}\right| \tag{2}$$

Friesz et al (1993) ont montré que ces conditions sont équivalentes au problème d'inégalités, qui est de trouver $h^* \in \Omega$ tel que

$$\sum_{\tau} \sum_{k \in K} \left(S(h^*), h - h^* \right) \geq 0, \forall h \in \Omega \tag{3}$$

où $K = \bigcup_{i \in I} k_i$ où h^{τ} est le vecteur des chemins des flots h_k^{τ}, $\forall$ k, $\forall$ τ.

Dans ce modèle, l'existence d'une solution unique dépend des propriétés de l'application *s(h[g])*. Cette dernière représente la relation de dépendance entre le coût des liens *s* et l'affectation de la demande *g* qui génère les flots sur les chemins *h*. Les propriétés de cette application ne sont pas facilement vérifiables, dû au fait que c'est le résultat d'une simulation et non d'une fonction analytique. Par conséquent, les propriétés théoriques sont difficiles à établir rigoureusement, et des résultats empiriques confirment les propriétés du modèle.

1.5 L'algorithme de solution

L'algorithme utilisé consiste en deux composantes principales autres que le calcul des plus courts chemins temporels. La première est une méthode pour déterminer un nouvel ensemble d'entrée de flots des chemins, à partir des temps de déplacements sur les chemins de la dernière itération. La deuxième est une méthode pour déterminer les flots actuels sur les liens et les temps de déplacement, qui sont les résultats d'un ensemble donné de taux de flots sur les chemins. De plus, l'algorithme nécéssite un ensemble de flots sur les chemins.

Les flots d'entrée sur les chemins $h_k^\tau, k \in K$ sont déterminés par une variante de la méthode des moyennes successives, qui est appliqué à chaque paire O-D I et à chaque intervalle de temps τ. Une solution réalisable initiale est trouvée en affectant la demande pour chaque période de temps à un ensemble de plus courts chemins dynamiques successifs. Commençant à la deuxième itération et ce jusqu'à un nombre prédéterminé d'itérations N, le temps de déplacement sur les liens est utilisé pour déterminer un nouvel ensemble de plus courts chemins dynamiques qui sont ajoutés à l'ensemble courant des chemins.

À l'itération n, $n \leq N$, le volume considéré comme flot d'entrée à chaque chemin dans l'ensemble est g_i^τ / n, $i \in I$, $\forall$ τ. Ensuite, pour l'itération m, $m \geq N$, le chemin le plus court parmi ceux utilisés est identifié et les taux des flots d'entrées sur les chemins sont redistribués sur les chemins connus.

Si le flot d'un chemin donné décroît en dessous d'une petite valeur prédéterminée, alors le chemin est laissé de côté et son flot restant est distribué aux autres chemins utilisés.

Voici le résumé de l'algorithme.

- *Étape 0 Initialisation ($l=1$):*

  Calculer les plus courts chemins temporels basés sur l'écoulement libre du temps de déplacement;

  Charger les demandes pour obtenir une solution initiale; $l=l+1$

- *Étape 1 Reallocation des flots d'entrées sur les chemins:*

  Étape 1.1 Si $l \leq N$

  Trouver un nouveau chemin plus court dynamique;

  Assigner à chaque chemin k le flot d'entrée $\frac{g_i^{\tau}}{l}$

  Étape 1.2 Si $l > N$

  Identifier le plus court chemin parmi les chemins utilisés;

  Redistribuer les flots comme suit:

$$h^l_k(\tau) = h^{l-1}_k(\tau)\left(\frac{l-1}{l}\right) + \frac{g_i(\tau)}{l} \qquad \text{si } s^l_k(\tau) = u^l_k(\tau)$$

$$h^l_k(\tau) = h^{l-1}_k(\tau)\left(\frac{l-1}{l}\right) \qquad \text{sinon}$$

- *Étape 2 Critère d'arrêt:*

  Si $l \leq L \; or \; RGap \leq \varepsilon \Rightarrow STOP$;

  sinon retour à l'étape 1

Aucune preuve formelle de convergence ne peut être donnée pour cet algorithme. Toutefois, une mesure du gap inspirée du modèle d'équilibre statique de réseau peut être utilisée pour qualifier une solution donnée. Le gap correspond à la différence entre le temps total de déplacement observé et le temps total de déplacement qui aurait été observé si tous les véhicules avaient le temps de déplacement sur chaque intervalle τ égal au temps de déplacement du plus court chemin courant.

Le RGap est le gap relatif pour chaque temps de départ τ, qui peut être calculé comme suit :

$$R\,Gap^{\tau}(n) = \frac{\sum_{i\in I}\sum_{k\in k_i} h_k^{\tau}(n) s_k^{\tau}(n) - \sum_{i\in I} g_i^{\tau}\, u_i^{\tau}(n)}{\sum_{i\in I} g_i^{\tau}\, u_i^{\tau}(n)}$$

La structure générale de l'algorithme est illustrée sur la figure 1.7.

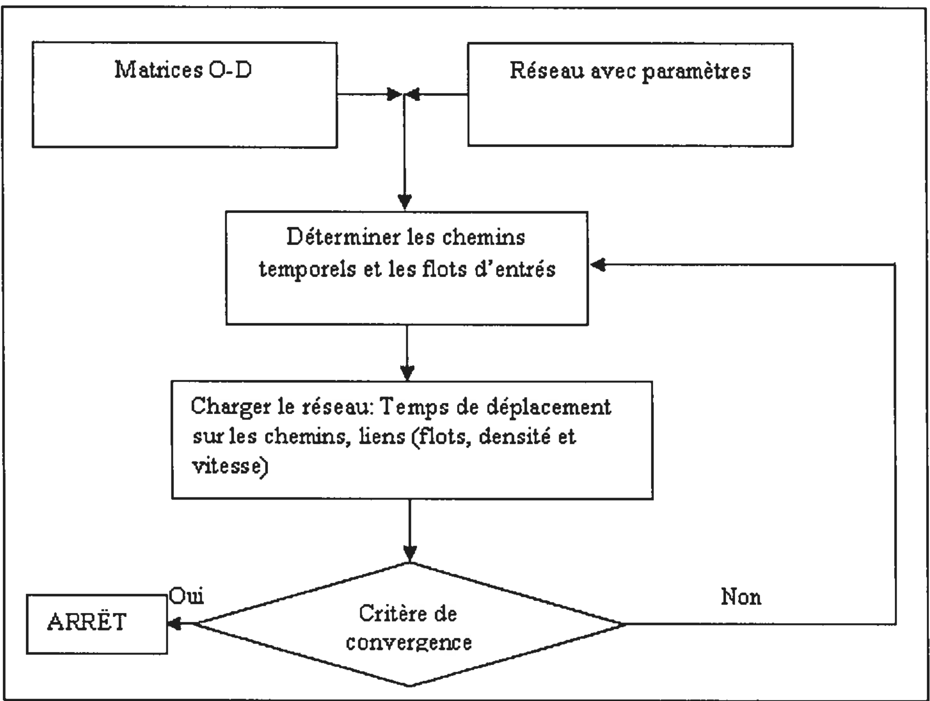


Figure 1.7 Le modèle ADT de Dynameq

1.6 Résultats d'une simulation

Après une simulation, Dynameq est un logiciel qui illustre très bien les résultats au niveau graphique. Il est possible d'avoir accès à une multitude d'informations, mais dans le cadre de ce travail, on va se limiter à décrire en détail les deux résultats qui sont utiles.

Le premier résultat est le taux horaire du nombre de véhicules par mouvement de virage à chaque intersection. Ce résultat peut servir de paramètre d'entrée dans le réseau de Synchro et nous en discuterons plus loin. Le deuxième résultat est la matrice des temps moyens de déplacement. Elle donne le temps moyen de déplacement d'un véhicule pour chaque paire O-D sur des intervalles donnés. Pour obtenir le temps moyen global de déplacement d'un véhicule, on fait une moyenne des intervalles. Ensuite, ce dernier résultat est multiplié par la demande, ce qui donne le temps total de déplacement des véhicules d'une paire O-D. En faisant la somme des temps totaux de déplacement pour chaque paire O-D, on obtient le temps total de déplacement des véhicules dans le réseau. C'est sur ce dernier résultat qu'on se base pour comparer le niveau de congestion de deux simulations. Voici maintenant un exemple de fenêtre qui illustre les résultats d'une simulation dans Dynameq.

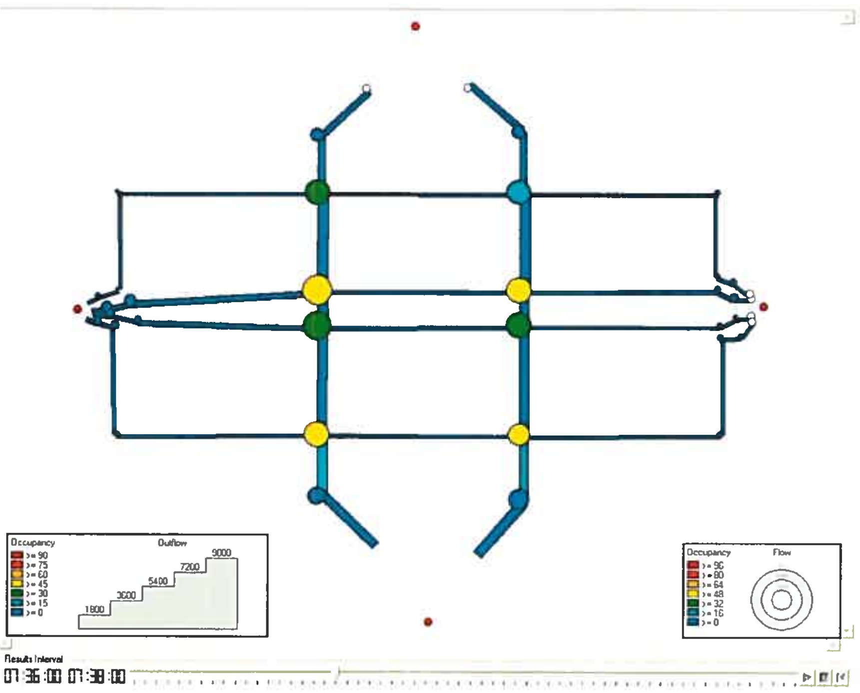


Figure 1.8 : Les résultats d'une simulation de Dynameq

1.7 Importation et exportation des fichiers d'entrée et de sortie

Dynameq permet d'importer et exporter une multitude de fichiers en format ascii. Nous ne donnerons ici qu'une description des fichiers utiles dans le cadre de ce travail. À noter que le deuxième fichier est le seul qu'on importe ou exporte, tandis que tous les autres servent seulement à exporter. Lorsqu'on importe un fichier, on n'a besoin que du plan de trafic, car c'est la seule composante du réseau qu'on change.

1) Un fichier contenant les informations sur le réseau
2) Un fichier contenant les informations sur le plan de trafic (**à importer ou exporter**)
3) Un fichier contenant les informations sur les mouvements de virages, après une simulation. Dynameq lui attribue le nom "movementcount.out"
4) Un fichier contenant les informations sur la matrice des temps moyens de déplacement, après une simulation.

Chapitre 2 - Synchro et ses fonctionnalités

Tout comme Dynameq, Synchro est un logiciel permettant de modéliser un réseau. En plus de cette fonction, il a comme rôle d'optimiser les contrôles. C'est pour cette raison que Synchro est un très bon complément à Dynameq.

2.1 Le réseau

Le concept de réseau est très semblable à celui de Dynameq. Les éléments qui le définissent sont les liens (rues) et les nœuds (intersections). Les centroïdes n'existent pas car, comme nous le verrons plus loin, les voitures ne partent pas d'une origine pour aller vers une destination. Un lien a deux directions, ce qui fait qu'un seul lien relie deux intersections. C'est une différence majeure avec le réseau de Dynameq où un lien n'a qu'une seule direction. Dans Synchro, les intersections se créent automatiquement lorsque deux liens se croisent. Une intersection est soit signalisée ou non signalisée. Voici maintenant un exemple de réseau dans Synchro.

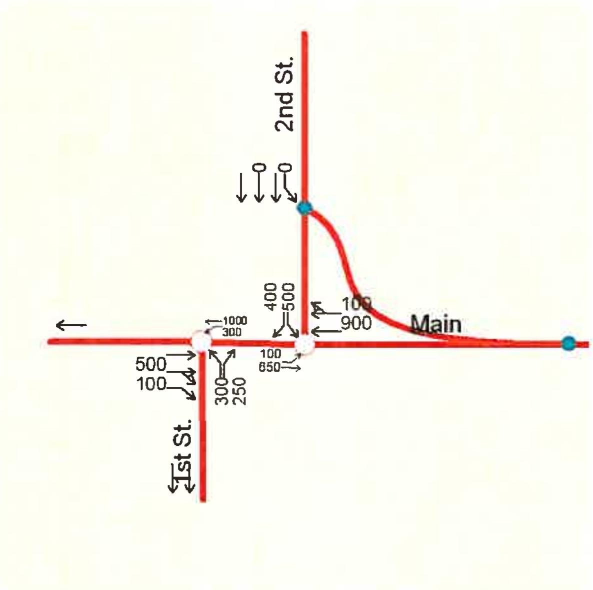


Figure 2.1 : Un réseau dans Synchro

Sur la figure 2.1, on y trouve un réseau dans Synchro. Les nœuds sont représentés par des points blancs (intersection signalisées) et bleus (intersections non signalisées) et les liens sont représentés par des lignes rouges.

2.2 Les voies

Les voies consistent principalement aux mouvements de virage à une intersection donnée. À chaque intersection, une fenêtre des voies est réservée. Dans cette dernière, on y retrouve les mouvements permis avec le nombre de voies réservées à ces derniers.

Il existe un système de points cardinaux pour définir les mouvements. Il existe une multitude de points cardinaux, mais dans le cadre de ce travail, nous nous limitons aux quatre suivants : nord, sud, est, ouest (on laisse tomber les combinaisons comme Nord-Est, Sud-Ouest, etc.). Le mouvement est donc défini comme suit : le point cardinal vers lequel le véhicule se dirige (EB pour Est, WB pour Ouest, NB pour Nord et SB pour Sud) avec le virage qu'il effectue (L pour gauche, T pour tout droit et R pour droite). En combinant ces deux derniers éléments, il y a donc 12 mouvements possibles.

Pour un mouvement donné, il y a deux types de voie. Le premier type est une voie standard et elle permet d'aller dans une seule direction. Ce type de voie se trouve dans tous les mouvements. Le deuxième type de voie est une voie partagée et elle donne le choix d'aller tout droit, ou de tourner. Ce type de voie se trouve uniquement dans la section du mouvement d'aller tout droit. Dans cette dernière, il y a alors 3 types de voie : les voies pour aller strictement tout droit, les voies partagées qui permettent de tourner à gauche ou d'aller tout droit et les voies partagées qui permettent de tourner à droite ou d'aller tout droit.

Plusieurs autres paramètres sont présents dans cette fenêtre, mais puisqu'ils ne sont pas nécessaires dans le cadre de ce travail, nous laissons Synchro définir des valeurs par défaut.

Voici maintenant un exemple de fenêtre des voies dans Synchro.

LANE WINDOW	EBL	EBT	WBT	WBR	SBL	SBR
Lanes and Sharing (#RL)	↰	↑	↑↱		↰	↱
Ideal Satd. Flow (vphpl)	1900	1900	1900	1900	1900	1900
Lane Width (ft)	12	12	12	12	12	12
Grade (%)	—	0	0	—	0	—
Area Type	—	Other	Other	—	Other	—
Storage Length (ft)	0	—	—	0	0	0
Storage Lanes (#)	—	—	—	—	—	—
Total Lost Time (s)	4.0	4.0	4.0	4.0	4.0	4.0
Leading Detector (ft)	50	50	50	—	50	50
Trailing Detector (ft)	0	0	0	—	0	0
Turning Speed (mph)	15	—	—	9	15	9
Right Turn Channelized	—	None	—	None	—	None
Curb Radius (ft)	—	—	—	—	—	—
Add Lanes (#)	—	—	—	—	—	—
Lane Utilization Factor	1.00	1.00	0.95	—	1.00	1.00
Right Turn Factor	1.000	1.000	0.985	—	1.000	0.850
Left Turn Factor (prot)	0.950	1.000	1.000	—	0.950	1.000
Saturated Flow Rate (prot)	1770	1863	3486	—	1770	1583
Left Turn Factor (perm)	0.154	1.000	1.000	—	0.950	1.000
Right Ped Bike Factor	1.000	1.000	1.000	—	1.000	1.000
Left Ped Factor	1.000	1.000	1.000	—	1.000	1.000
Saturated Flow Rate (perm)	287	1863	3486	—	1770	1583
Right Turn on Red	—	—	—	Yes	—	Yes
Saturated Flow Rate (RTOR)	0	0	16	—	0	26
Headway Factor	1.00	1.00	1.00	1.00	1.00	1.00

Figure 2.2 : Une fenêtre des voies dans Synchro

Sur la figure 2.2, on y trouve six mouvements possibles à l'intersection. On remarque qu'il y a 5 voies standards et une voie partagée. Cette dernière se trouve au mouvement WBT, qui se dirige vers l'ouest, en allant tout droit.

2.3 Les volumes

Les volumes correspondent au taux horaire de véhicules effectuant un mouvement de virage à une intersection donnée. À chaque intersection, une fenêtre des volumes est réservée. Cette dernière est divisée en sections selon le nombre de mouvements permis (12 au maximum). On y trouve donc pour chaque section le nombre de véhicules effectuant le mouvement en question. Dans le cadre de ce travail, ce paramètre est directement importé à partir du résultat d'une simulation de Dynameq, qui nous donne le compte des mouvements de virage.

Encore une fois, plusieurs autres paramètres sont présents dans cette fenêtre, mais ne sont pas nécessaires dans le cadre du travail. On laisse alors Synchro définir des valeurs par défaut. Voici maintenant un exemple d'une fenêtre des volumes.

VOLUME WINDOW	EBL	EBT	WBT	WBR	SBL	SBR
Traffic Volume (vph)	100	650	900	100	500	400
Conflicting Peds. (#/hr)	0	—	—	0	0	0
Conflicting Bikes (#/hr)	—	—	—	0	—	0
Peak Hour Factor	1.00	1.00	1.00	1.00	1.00	1.00
Growth Factor	1.00	1.00	1.00	1.00	1.00	1.00
Heavy Vehicles (%)	2	2	2	2	2	2
Bus Blockages (#/hr)	0	0	0	0	0	0
Adj. Parking Lane?	No	No	No	No	No	No
Parking Maneuvers (#/hr)	—	—	—	—	—	—
Traffic from mid-block (%)	—	0	0	—	0	—
Link OD Volumes	—			—		—
Adjusted Flow (vph)	100	650	900	100	500	400
Lane Group Flow (vph)	100	650	1000	0	500	400

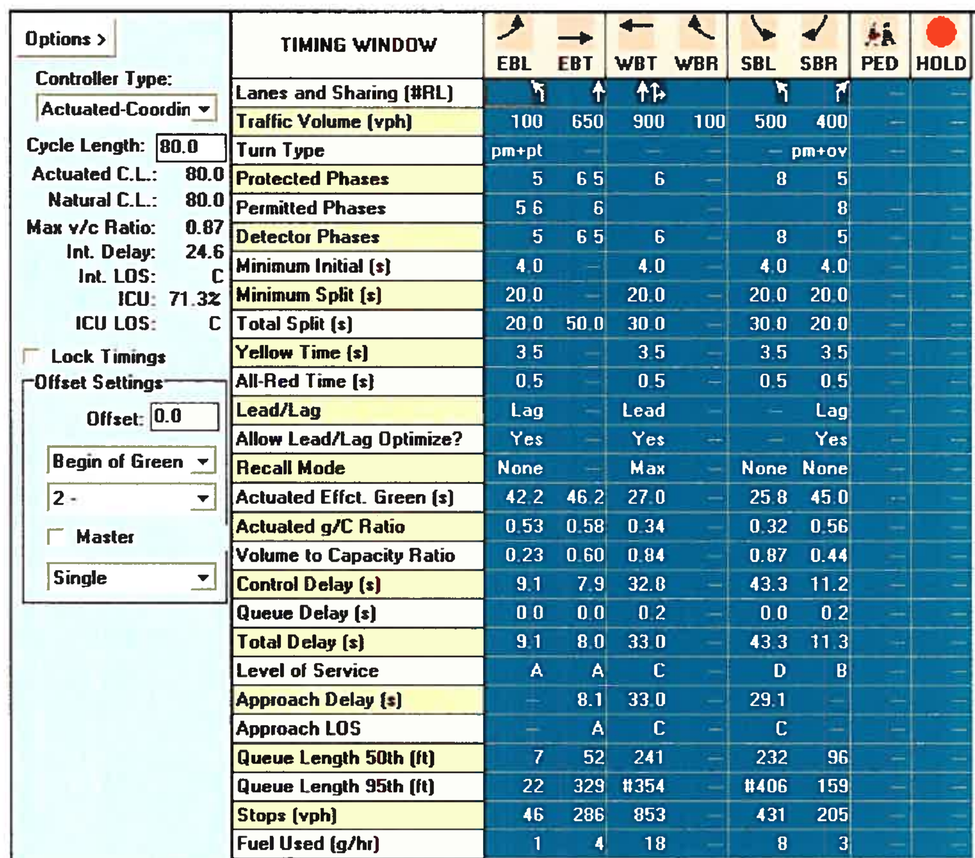
Figure 2.3 : Une fenêtre des volumes dans Synchro

2.4 Le temps

Dans Synchro, la version du plan de trafic de Dynameq est divisée en deux composantes, le temps et les phases. Examinons d'abord celle du temps. Celle des phases sera décrite à la section suivante.

Le temps représente presque toutes les informations du plan de trafic. À chaque intersection, une fenêtre du temps est réservée. D'abord, on y retrouve les informations globales sur le plan telles que la durée du cycle ainsi que le délai. Ensuite, on y retrouve les informations sur les mouvements. Deux types de phases peuvent être attribués à chacun des mouvements. Le premier type est une phase protégée, lorsque le mouvement a priorité sur un autre mouvement causant un conflit. Le deuxième type est une phase permise, lorsque le mouvement est permis, mais doit céder le passage à un autre mouvement causant un conflit.

Encore une fois, plusieurs autres informations sont présentes dans cette fenêtre, mais dans le cadre de ce travail, nous n'en tenons pas compte et laissons Synchro définir des valeurs par défaut. Voici maintenant un exemple de fenêtre du temps.



TIMING WINDOW	EBL	EBT	WBT	WBR	SBL	SBR	PED	HOLD
Lanes and Sharing (#RL)				–			–	–
Traffic Volume (vph)	100	650	900	100	500	400	–	–
Turn Type	pm+pt	–	–	–	–	pm+ov	–	–
Protected Phases	5	6 5	6	–	8	5	–	–
Permitted Phases	5 6	6		–		8	–	–
Detector Phases	5	6 5	6	–	8	5	–	–
Minimum Initial (s)	4.0	–	4.0	–	4.0	4.0	–	–
Minimum Split (s)	20.0	–	20.0	–	20.0	20.0	–	–
Total Split (s)	20.0	50.0	30.0	–	30.0	20.0	–	–
Yellow Time (s)	3.5	–	3.5	–	3.5	3.5	–	–
All-Red Time (s)	0.5	–	0.5	–	0.5	0.5	–	–
Lead/Lag	Lag	–	Lead		–	Lag	–	–
Allow Lead/Lag Optimize?	Yes	–	Yes	–	–	Yes	–	–
Recall Mode	None	–	Max	–	None	None	–	–
Actuated Effct. Green (s)	42.2	46.2	27.0	–	25.8	45.0	–	–
Actuated g/C Ratio	0.53	0.58	0.34	–	0.32	0.56	–	–
Volume to Capacity Ratio	0.23	0.60	0.84	–	0.87	0.44	–	–
Control Delay (s)	9.1	7.9	32.8	–	43.3	11.2	–	–
Queue Delay (s)	0.0	0.0	0.2	–	0.0	0.2	–	–
Total Delay (s)	9.1	8.0	33.0	–	43.3	11.3	–	–
Level of Service	A	A	C	–	D	B	–	–
Approach Delay (s)	–	8.1	33.0	–	29.1	–	–	–
Approach LOS	–	A	C	–	C	–	–	–
Queue Length 50th (ft)	7	52	241	–	232	96	–	–
Queue Length 95th (ft)	22	329	#354	–	#406	159	–	–
Stops (vph)	46	286	853	–	431	205	–	–
Fuel Used (g/hr)	1	4	18	–	8	3	–	–

Figure 2.4 : Une fenêtre du temps dans Synchro

Sur la figure 2.4, on y trouve dans la partie de gauche les informations globales sur le plan. Dans la partie de droite, on y trouve les informations sur les mouvements.

2.5 Les phases

Les phases sont la deuxième composante de la version du plan de trafic de Dynameq. À chaque intersection signalisée, une fenêtre du temps est réservée. Dans cette dernière, on y retrouve des informations sur chacune des phases. La plus importante est le nombre de temps alloué en secondes. Elles se divisent sur le temps maximal, le temps de feu jaune et le temps de feu rouge. Le temps de feu vert s'obtient en soustrayant les temps de feu jaune et rouge au temps maximal.

Encore une fois, plusieurs autres informations sont présentes dans cette fenêtre, mais dans le cadre du travail, nous n'en tenons pas compte et laissons Synchro définir des valeurs par défaut. Voici maintenant un exemple de fenêtre des phases.

Options >

Controller Type: Actuated-Coordin

Cycle Length: 80.0

Actuated Cycles
90th %: 80.0
70th %: 80.0
50th %: 80.0
30th %: 80.0
10th %: 80.0

Quick Reports:
Green Times
Starts
Details

PHASING WINDOW	1-WBTL	2-EBWB	4-NBL	5-EBTL	6-EBWB	8-SBL
Minimum Initial (s)	4.0	4.0	4.0	4.0	4.0	4.0
Minimum Split (s)	10.0	20.0	20.0	20.0	20.0	20.0
Maximum Split (s)	10.0	44.0	26.0	20.0	30.0	30.0
Yellow Time (s)	3.5	3.5	3.5	3.5	3.5	3.5
All-Red Time (s)	0.5	0.5	0.5	0.5	0.5	0.5
Lead/Lag	–	Lead	Lag	Lag	Lead	–
Allow Lead/Lag Optimize?	–	Yes	Yes	Yes	Yes	–
Vehicle Extension (s)	3.0	3.0	3.0	3.0	3.0	3.0
Minimum Gap (s)	3.0	3.0	3.0	3.0	3.0	3.0
Time Before Reduce (s)	0.0	0.0	0.0	0.0	0.0	0.0
Time To Reduce (s)	0.0	0.0	0.0	0.0	0.0	0.0
Recall Mode	None	C-Max	None	None	Max	None
Pedestrian Phase	No	Yes	Yes	No	Yes	Yes
Walk Time (s)	–	5.0	5.0	–	5.0	5.0
Flash Dont Walk (s)	–	11.0	11.0	–	11.0	11.0
Pedestrian Calls (#/hr)	–	0	0	–	0	0
Dual Entry?	No	Yes	Yes	No	Yes	Yes
Inhibit Max?	Yes	Yes	Yes	Yes	Yes	Yes
90th %ile Green Time (s)	6 mx	40 cd	22 mx	16 mx	26 cd	26 mx
70th %ile Green Time (s)	7 mx	40 cd	21 gp	16 mx	26 cd	26 mx
50th %ile Green Time (s)	10 mx	40 cd	18 gp	16 mx	26 cd	26 mx
30th %ile Green Time (s)	13 mx	40 cd	16 gp	16 gp	26 cd	26 hd
10th %ile Green Time (s)	12 gp	45 cd	12 gp	12 gp	31 cd	25 hd

Figure 2.5 : Une fenêtre des phases dans Synchro

2.6 Optimisation des contrôles

Pour faire son optimisation, Synchro se base sur toutes les composantes mentionnées précédemment et ne les applique que sur les intersections signalisées. Il existe 2 types d'optimisation. Le premier type est l'optimisation de la longueur d'un cycle, en fixant une durée de cycle indentique à toutes les intersections.

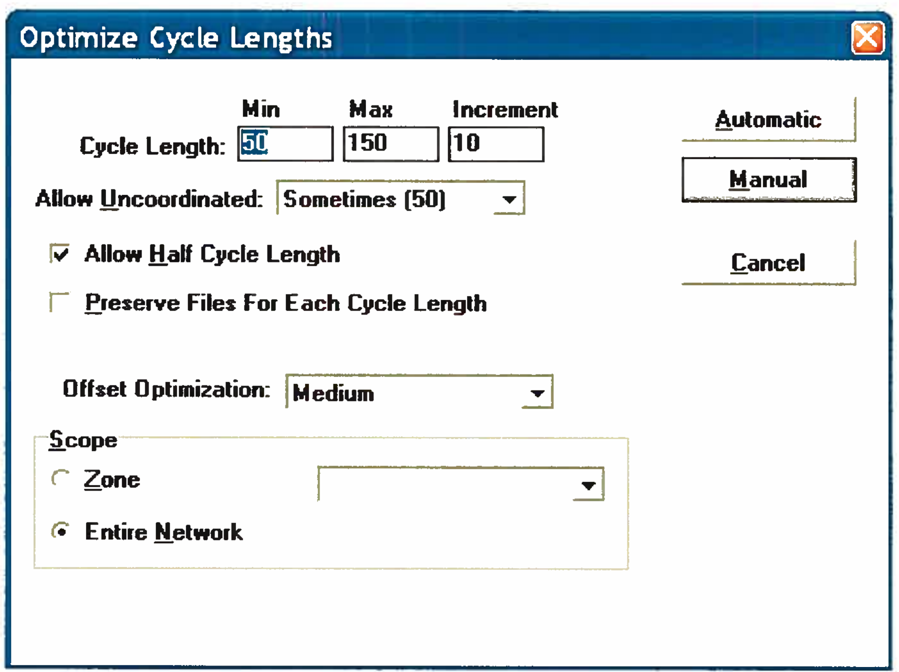


Figure 2.6 : Optimisation de la longueur d'un cycle dans Synchro

Il est à noter qu'on peut raffiner la recherche en forçant Synchro à trouver une longueur de cycle dans un intervalle donné ainsi qu'une incrémentation désirée.

Le deuxième type est l'optimisation du délai. Synchro peut aussi optimiser le temps maximal, mais ceci est optionnel. Pour chaque intersection, Synchro trouve une valeur optimale de délai. Ensuite, Synchro fixe une valeur de temps maximal à chaque phase. Il est à noter que les temps de feu jaune et de feu rouge ne sont pas optimisés et par conséquent, restent les mêmes.

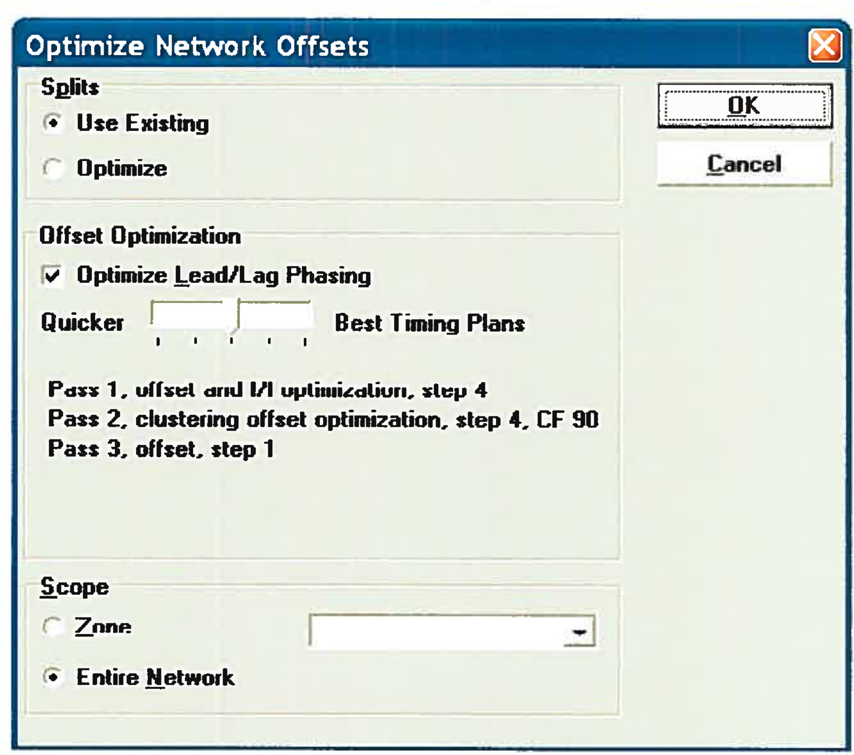


Figure 2.7 : Optimisation du décalage et du temps maximal dans Synchro

Après l'optimisation, des nouveaux plans sont donc produits et nous verrons en détail au prochain chapitre comment se servir de ces plans pour les appliquer dans le même réseau de Dynameq.

Nous ne connaissons malheureusement pas l'algorithme qui se cache derrière Synchro pour optimiser les plans. Dans le cadre de ce travail, nous allons supposer que cet algorithme résout correctement les problèmes posés à Synchro.

2.7 Lecture et écriture des fichiers d'entrée et de sortie

La fonction de lecture et d'écriture de fichiers d'entrée et sortie est très facile d'accès. Une fenêtre permet de choisir la tâche désirée, comme le montre la figure 2.8.

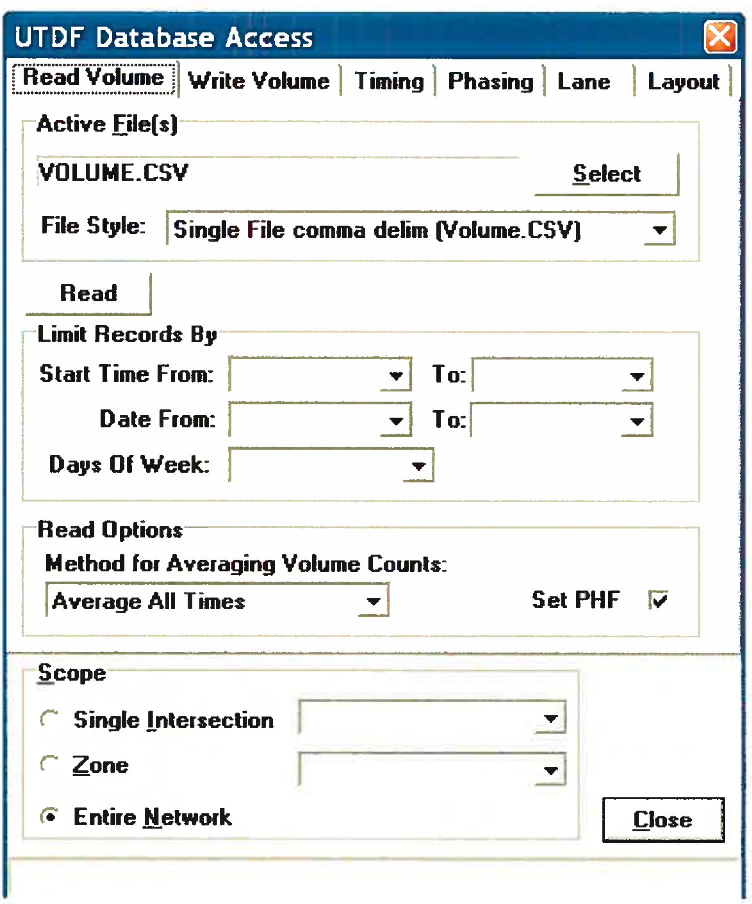


Figure 2.8 : Lecture et écriture des fichiers d'entrée et de sortie dans Synchro

Les onglets dans le haut de la fenêtre permettent de lire ou écrire les différents fichiers de format CSV. Dans le cadre de ce travail, les fichiers d'entrée à lire sont ceux du réseau, des voies, du temps, des phases ainsi que des volumes. Les fichiers de sortie à écrire ne sont seulement que ceux du temps et des phases. Dynameq n'a besoin que de ces deux derniers, car ce ne sont que les plans qui changent lors du passage du réseau de Synchro à Dynameq (voir chapitre suivant pour plus de détails).

2.8 Différences majeures entre Synchro et Dynameq

Comme nous avons pu constater précédemment, il y a des différences majeures entre Dynameq et Synchro. Voici un résumé en détail de ces différences.

1) Dans Dynameq, les flots sur liens sont une composante importante, alors que dans Synchro, on ne tient compte que des volumes. L'emphase est mise principalement sur l'intersection.
2) Dans Synchro, il n'y a pas de concept de demande. On ignore le point d'origine et de destination des véhicules, ce qui est importe, étant le nombre de véhicules aux intersections.
3) Dans Dynameq, les liens n'ont qu'une seule direction alors que dans Synchro, les liens ont deux directions.

Chapitre 3 - Interface entre Synchro et Dynameq

3.1 Généralités

Nous voici maintenant au cœur du travail. Nous avons conçu un programme d'interface entre Synchro et Dynameq qui a deux rôles principaux:

1) Transférer un réseau de Dynameq à Synchro (Ntw_S et Vol_S). Les fichiers de Dynameq sont convertis en fichiers de Synchro afin de permettre à ce dernier d'optimiser les contrôles. Cette partie est converte en détail à la section 3.2.

2) Transférer des plans de Synchro à Dynameq (Ntw_D). Les fichiers de Synchro sont convertis en un fichier de Dynameq afin de permettre à ce dernier de faire une simulation avec les plans. Cette partie est couverte en détail à la section 3.3.

L'interface consiste en 3 programmes et d'une librairie de classes écrites en Java. Voici d'abord une desciption des 3 programmes :

Ntw_S

Fonctionnalité : Convertit 2 fichiers de Dynameq et produit 4 fichiers CSV pour Synchro.

Classes requises : ListElement, ListElementNodes, MyIn

Fichiers en entrée : reseau.dat, plan

Fichiers en sortie : LAY.CSV, LAN.CSV, TIM.CSV, PHA.CSV, corTable.txt

Exécution : java Ntw_S reseau.dat plan où

1) reseau.dat est le fichier ascii exporté de Dynameq représentant le réseau
2) plan est le fichier ascii exporté de Dynameq représentant le plan de trafic

Vol_S

Fonctionnalité : Convertit 1 fichier de Dynameq et produit 1 fichier CSV pour Synchro.

Classes requises : ListElement, MyIn, NodeVolumes

Fichiers en entrée : movementcout.out (représente le compte des mouvements de virage)

Fichiers en sortie : VOL.CSV

Exécution : java Vol_S movementcount.out

Ntw_D

Fonctionnalité : Convertit 2 fichiers de Synchro et produit 1 fichier ascii pour Dynameq.

Classes requises : CSVReader, CSVWriter

Fichiers en entrée : TIM.CSV, PHA.CSV

Fichiers en sortie : trafficControl.dat

Exécution : java Ntw_D plan où

1) TIM.CSV et PHA.CSV sont les fichiers exportés de Synchro représentant le plan de trafic
2) plan est le fichier original ascii exporté de Dynameq représentant le plan de trafic

Voici maintenant une desciption des autres classes de la librairie :

CSVReader: Lit un fichier de format CSV de Microsoft Excel. Le fichier In est nécessaire au fonctionnement de celui-ci.

CSVWriter: Écrit un fichier de format CSV de Microsoft Excel.

In: Lit des différents types de données à partir de: stdin, fichier, URL.

ListElement: Représente une liste chaînée de liens. Chaque lien de la liste a différentes propriétés, dont un pointeur vers le prochain lien de la liste.

ListElementNodes: Représente une liste chaînée de noeuds. Chaque nœud de la liste a différentes propriétés dont un pointeur vers le prochain nœud de la liste.

MyIn: Fichier codé à partir de StdIn.java, programmé par des professeurs non-identifiés de l'Université de Princeton. Construit un fichier qui pourra être lu mot par mot.

NodeVolumes : Classe utilisée par Vol_S. Représente un nœud avec le compte des mouvements de virages à ce dernier.

Il est très important de s'assurer que les classes ont déjà été compilées avec succès avant l'exécution d'un des trois programmes principaux. Dans ce qui suit, il y aura une description en détail de ces trois derniers. D'abord, il y aura une description des programmes Ntw_S et Vol_S qui convertissent les fichiers de Dynameq à ceux Synchro. Ensuite, il y aura une description du programme Ntw_D qui convertit les fichiers de Synchro à celui de Dynameq.

3.2 Conversion des fichiers de Dynameq à ceux de Synchro

3.2.1 Conversion du fichier du réseau et du plan de trafic (Ntw_S)

Ntw_S est le programme qui convertit des fichiers ascii exportés de Dynameq aux fichiers de format CSV (LAY, LAN, TIM et PHA) de Synchro. Pour exécuter, il faut entrer la commande suivante : java Ntw_S reseau.dat plan. Il y a deux fichiers passés en argument lors de l'exécution de Ntw_S. Le premier, reseau.dat, est le fichier ascii représentant le réseau, tandis que le deuxième, plan, est le fichier ascii représentant le plan de trafic du réseau.

Voici maintenant une description de la structure du programme Ntw_S. Les points ont été divisés selon les sections (en anglais) du code.

Obtention d'informations (section "Get info")

Le programme parcourt les fichiers de sorties de Dynameq, prend les informations et les met dans les listes chaînées ListElement et ListElementNodes. Ces dernières contiennent respectivement les informations sur les liens, les mouvements et les nœuds du réseau.

Modification des liens (section "Modify links")

Cette partie est seulement utile pour corriger une erreur qui arrive parfois lors du changement des liens de Dynameq à Synchro. Dans le réseau de Dynameq, l'emphase est mise sur le nœud individuel. Dans celui de Synchro, les liens ne sont qu'une façon de rejoindre des nœuds qui sont adjacents. À une intersection donnée, les informations sur les liens sont requises pour exécuter l'optimisation, qui est dépendante du numéro ainsi que de l'orientation des liens. Un lien a deux directions, tandis que dans le réseau de Dynameq, un lien n'a qu'une seule direction. Toujours dans Synchro, à chaque lien qui est connecté à une intersection est assignée une direction (nord, est, sud-ouest, etc.). Puisque nous ne connaissons pas l'algorithme que Synchro utilise pour assigner une direction à un lien, nous devons faire des suppositions. Nous allons fixer les directions cardinales Nord, Sud, Est, Ouest (et

non NO, NE, etc.) comme les seules possibles. Le raison est que nous ne pouvons traiter les intersections de cinq branches ou plus et ceci est une limite du programme. Ce qui est réglé ici est le cas dans lequel une intersection de Dynameq a huit liens (un lien a une seule direction dans ce cas) reliés à une intersection, mais pas nécessairement en paires. Par exemple, les deux liens partant et arrivant à une intersection donnée peuvent provenir de deux nœuds différents. Cet exemple est présent dans le réseau de Calgary, au nœud 3413. À cette intersection, il y a cinq nœuds qui sont considérés comme adjacents dans Synchro et cinq liens sont donc créés.

Cette partie du code règle le problème. Dans Dynameq deux liens entrant et sortant d'une intersection ayant les mêmes directions mais qui sont reliés à deux nœuds différents sont modifiés pour être reliés à un nœud adjacent commun à l'intersection. Après ce changement, l'intersection a donc quatre liens dans Synchro et à chacun de ces liens est assignée une direction (nord, sud, est, ouest).

D'autres modifications et ajustements sont possibles, mais ils n'ont pas encore été implémentés puisqu'ils ne changeraient pas l'optimisation faite par Synchro. L'ajustement le plus important à apporter serait celui de régler les problèmes de directions et de liens. C'est le cas de bien des intersections dans le réseau de Calgary. Une fois convertis à Synchro, les liens n'ont pas de directions cardinales précises. Ceci est à cause des angles bizarres ainsi que des courbes qui ne sont pas représentées. Une façon de régler ce problème serait de repérer les intersections dans cette partie du programme et d'ajouter des nœuds dans Synchro dans le but de réaligner les liens. Pour l'instant, cet ajustement n'est pas nécessaire, puisque dans le réseau de Calgary, aucune intersection signalisée n'a ce problème.

Écriture des fichiers de sorties (section “Write Output Files”)

Cette section permet d’écrire les trois fichiers suivants : réseau, voies et table de correspondance. Voyons en détail comment sont écrits chacun des trois fichiers.

Réseau (layout)

Un fichier du réseau est créé pour Synchro, qui est nommé par défaut LAYOUT.CSV. Le programme le nomme LAY.CSV. Ce fichier contient seulement les informations sur les nœuds, ce qui est suffisant pour obtenir un réseau complet sauf pour les informations sur les voies. À chaque nœud est assigné un numéro, un nom, des coordonnées, un type signalisé ou non signalisé et le numéro des nœuds qui lui sont adjacents. Avec cette information, Synchro crée automatiquement des liens entre les nœuds adjacents.

Voici maintenant des détails sur la méthode de programmation. Pour chaque nœud, son numéro est gardé lors du passage du fichier de Dynameq au fichier de Synchro. Les noms sont simples, comme “Intersection 123” pour un nœud avec numéro 123 et de type signalisé. Pour déterminer ce type de nœud, le programme trouve les nœuds qui lui sont adjacents. Ils sont mis temporairement dans une matrice (adjacentNodesTemp[]). Ensuite, dépendamment du nombre de nœuds adjacents, le nom et le type du nœud sont écrits dans le fichier.

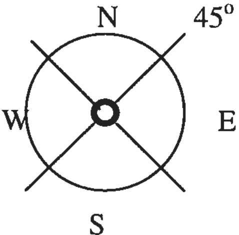


Fig 3.1 : Les angles d’un nœud dans Synchro

La prochaine étape est de trouver l’angle entre chaque nœud adjacent et le nœud courant. Dépendamment de l’angle (voir la figure 3.2), une direction cardinale est attribuée à chaque nœud adjacent et est mise dans la matrice adjacentNodes[][]. Le

premier indice utilise l'ordre dans lequel les nœuds apparaissent dans la liste chaînée des nœuds. Le deuxième indice correspond aux quatre points cardinaux dans l'ordre suivant : nord, sud, est, ouest. Ensuite, dépendamment de sa direction à partir du nœud, les nœuds adjacents sont écrits dans le bon ordre.

Voies (lanes)

Synchro prend les informations du fichier des voies (nommé LANES.CSV par défaut, mais LAN.CSV pour le programme) pour obtenir un réseau plus détaillé que celui obtenu avec le fichier du réseau. À chaque intersection, on y retrouve les informations sur le nombre de voies pour chaque mouvement. On y retrouve aussi les voies partagées pour chaque mouvement. Le fichier des voies contient deux rangées d'informations pour chaque intersection. La première rangée désigne le nombre de voies utilisées pour chacun des 12 mouvements possibles. La deuxième rangée indique si une voie utilisée par un mouvement d'aller tout droit est partagée entre ce dernier et un mouvement de virage à gauche et/où à droite.

Les données dans Dynameq sont suffisantes pour y retirer toute l'information sur les voies, mais n'est pas directement accessible. En plus du fichier sur le réseau de Dynameq, on a besoin du fichier sur son plan de trafic. À l'aide de ce dernier, le programme parcourt les phases de chaque nœud pour avoir les informations sur les mouvements spécifiques de chaque voie. Les mouvements sont facilement identifiables grâce à la matrice des nœuds adjacents. Le programme écrit par la suite l'information dans le fichier. Pour ce faire, il commence par aller au premier mot "NODE" et lit le numéro du nœud. Ensuite, il trouve l'indice du nœud dans la liste chaînée des nœuds pour accéder à ses nœuds adjacents dans adjacentNodes[][].

Tout le long du programme, une correspondance entre les mouvements et un chiffre est utilisé comme l'illustre la figure 3.3. Les tableaux appelés lanesByMouvement, laneIDbyMovement et sharedLanesByMovement dans Ntw_S.java donnent l'information sur chacun des 12 mouvements.

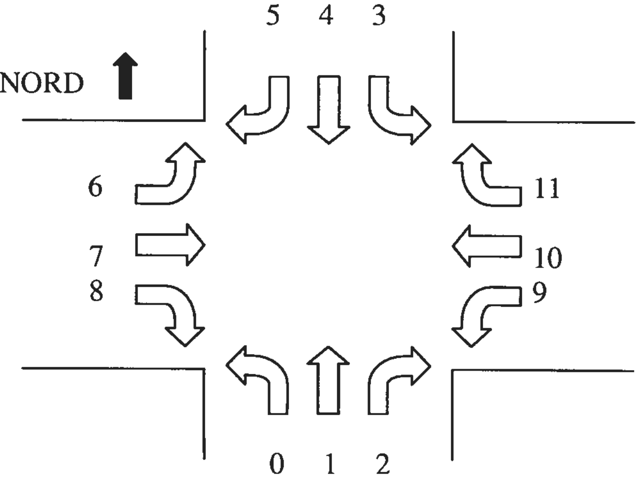


Fig 3.2 : Le numéro des voies à une intersection

À la prochaine étape, le programme lit l'information sur chaque mouvement dans la section "STATE". Pour chaque mouvement, les informations suivantes sont prises en note : le numéro du lien qui précède le mouvement, le numéro du lien qui suit le mouvement, le nombre de voies attribuées au mouvement et le "in lane", numéro de la voie la plus à gauche. Utilisant les numéros des liens, le programme fait une représentation objet de ces derniers. À partir des données de ces objets, il repère ensuite les nœuds précédant et suivant le mouvement. Ensuite, il accède aux directions des nœuds adjacents. Il est ensuite facile d'identifier le type de mouvement. Les informations sont finalement écrites dans la table de correspondance ainsi que dans la liste chaînée des mouvements.

Table de correspondance (correspondance table)

L'information contenue dans la table de correspondance est sur les voies et les liens. Le but de cette dernière est de faire un pont entre les liens de Dynameq et les directions des nœuds dans Synchro. Son format ressemble à celui dans Dynameq, et elle contient trois sections. La première section contient des informations sur les nœuds de remplacement. La seconde section identifie les types de mouvements (format de Synchro) à une intersection, avec les numéros des liens (format de Dynameq). Finalement, il y a la section des liens avec leurs propriétés.

Conversion du fichier du plan de trafic

Cette section convertit le fichier ascii du plan de trafic et ajoute de l'information au fichier LANES.CSV déjà créé. Il produit aussi les fichiers TIM.CSV et PHA.CSV de Synchro. Voici les méthodes expliquées en détail.

void writeTraffic(String ControlDataFileName)

Voici la méthode principale qui va ajouter des lignes au fichier LANES.CSV. Pour ce faire, elle parcourt mot par mot le fichier du plan de trafic généré par Dynameq (ControlDataFileName).

Chaque fois que le mot "NODE" est rencontré, le numéro de ce noeud est gardé en mémoire. Ensuite, pour chaque nœud, chaque fois que le mot "STATE" est rencontré, le numéro de cette phase est aussi gardé en mémoire. Pour chaque phase, on trouve les mouvements permis grâce à la méthode *findDirection(int ID, int startID, int endID)*. On ajoute alors les mouvements permis de cette phase dans un vecteur. Une fois qu'on a parcouru toutes les phases du nœud courant, les vecteurs sont ajoutés dans un vecteur principal appelé "states".

Ensuite, si on rencontre le mot "PLAN", c'est signe qu'il y a un plan de trafic, donc quelque chose à faire dans LANES.CSV. On ajoute tous les vecteurs "states" au vecteur "plan". Ensuite, il reste à convertir le vecteur plan en un format imprimable dans LANES.CSV grâce à la méthode *makeOutput(int ID, Vector plan).*

String findDirection(int ID, int startID, int endID)

Cette méthode reçoit en paramètres le numéro du noeud courant, le startID et le endID. Elle parcourt la liste de mouvements (moves) et cherche le mouvement avec les paramètres donnés. Elle retourne "extra", qui équivaut à une String qui représente la direction du movement (NBE, NBL, etc…).

void makeOutput(int ID, Vector plan)

Cette méthode reçoit en paramètres le numéro du nœud courant et le vecteur "plan" qui contient tous les states. Elle prépare deux vecteurs (outputPhases et outputPermPhases) qui serviront à l'impression des lignes dans le fichier LANES.CSV. En parcourant les différents states dans le vecteur "plan", cette méthode s'assure de remplir correctement les deux vecteurs grâce aux méthodes *conflict(Vector p)* et *fillOutputs(int a, int i, Vector p, Vector output)*.

boolean conflict(Vector p):

Cette méthode reçoit en paramètre un vecteur contenant les différents mouvements permis lors d'une phase et vérifie s'il y a conflit entre deux mouvements. Pour ce faire, elle vérifie, pour chaque mouvement tout droit, si les mouvements dans les trois autres directions qui entrent en conflit avec ce mouvement est présent dans le vecteur. Si oui, la méthode retourne true, false sinon.

Vector keepXBT(Vector p):

Cette méthode reçoit en paramètre un vecteur contenant les différents mouvements permis lors d'une phase et enlève de ce vecteur tous les mouvements sauf ceux allant tout droit. Elle retourne une copie de ce vecteur modifié.

Vector fillOutputs(int a, int i, Vector p, Vector output)

Cette méthode reçoit en paramètre un entier *a*, qui équivaut à 1 si on travaille sur le vecteur outputPhases et 2 si on travaille sur le vecteur outputPermPhase (output, le 4e paramètre, étant une copie du vecteur). Elle reçoit également en paramètre un entier *i* qui indique la phase courante sur laquelle on travaille. Finalement, elle reçoit en paramètre le vecteur *p*, la phase courante. La méthode parcourt les mouvements permis du vecteur *p* et va remplir, avec le chiffre *i* indiquant la phase courante, les bons endroits dans des tableaux. Ces derniers représentent chacun une ligne dans le fichier LANES.CSV. Ces tableaux sont ensuite stockés dans le vecteur output (ce que la méthode retourne). Il est à noter que si *a*=2, cela signifie qu'il y a un conflit

entre les mouvements dans la phase courante. Les mouvements tout droits ne seront pas ajoutés au vecteur “output” (qui représente outputPermPhase dans ce cas) et seront ajoutés dans le vecteur outputPhase.

int[] newTable(int size, int EMPTYVALUE)

Cette méthode crée simplement un tableau de taille équivalente au paramètre “size”, avec comme valeur par défaut “EMPTYVALUE”.

void printOutput(int a, int ID, Vector output)

C’est cette méthode qui va écrire directement dans le fichier LANES.CSV. Elle prend en paramètre l’entier a pour savoir s’il faut imprimer la ligne Phase (si $a=1$) ou PermPhase (si $a=2$). Ensuite, en ayant en paramètres le numéro du nœud courant ainsi que le vecteur output, elle fait simplement imprimer chaque élément (tableau) du vecteur output.

3.2.2 Conversion du fichier des volumes (Vol_S)

Après avoir fait une simulation dans Dynameq, un fichier ascii movementcount.out contenant les volumes est exporté. À partir de ce dernier, ce programme génère le fichier d’entrée de Synchro, VOL.CSV.

À l’aide de la table de correspondance, ce programme prend le fichier des volumes à toutes les intersections et le convertit en un fichier VOL.CSV. Il est requis que le fichier des volumes soit produit à partir d’une simulation sur une heure, car les volumes représentent un nombre de véhicules exécutant les mouvements sur une heure.

Voici maintenant la structure générale du programme. Les points ont été divisés selon les sections en anglais du code.

Écriture des sections de remplacement et de mouvements dans la table de correspondance

Le programme lit la section de remplacement et place le numéro du nœud dans une table de hachage nommé “replace”. Les clés sont des nombres représentant les numéros des nœuds qui doivent être remplacés et la clé de correspondance est un nombre entier avec le numéro du nœud avec lequel il est remplacé. Ensuite, il lit la section des mouvements exactement comme dans Ntw_D (voir la section 3.2).

Obtention des volumes à partir des comptes

Ce fichier de comptes est formaté de façon à ce que chaque ligne contienne le numéro de l’intersection, suivi du numéro du nœud à partir duquel proviennent les véhicules et ensuite, le numéro du nœud vers lequel vont les véhicules. Le reste de la ligne contient les comptes du nombre de voitures effectuant le virage. Ces comptes sont basés sur une simulation d’une heure.

Le programme écrit d’abord les entêtes et initialise les variables. Pour chaque ligne d’entrée, il vérifie si le nœud est différent du précédant. Si tel est le cas, il met le compte dans une table de hachage de volumes sous la clé qui est le numéro de l’ancien nœud. Ensuite, il réinitialise le tableau des comptes et définit le nouveau numéro du nœud pour le reste de l’itération. Puis, il lit les numéros des nœuds entrants et sortants du mouvement et les remplace dans le cas où ils apparaissent dans la table de hachage de remplacement. Par la suite, il fait la somme des comptes sur le reste de la ligne et trouve le type de mouvement à partir du numéro du nœud. Finalement, si le mouvement est permis, il met le compte dans sa position, dans le tableau des volumes de l’intersection courante.

Écriture des volumes dans le fichier

Le programme trouve tous les numéros des nœuds dans la table de hachage. Ensuite, il les parcourt et prend note de la valeur des comptes. Ensuite, il écrit les volumes pour chaque mouvement dans le fichier VOL.CSV.

3.3 Conversion des fichiers de Synchro à celui de Dynameq

Après avoir fait une optimisation des contrôles du réseau de Synchro, on peut exporter des fichiers de sortie de format CSV, représentant un plan de trafic (TIMING et PHASING). Le but de ce programme est de convertir les deux fichiers au fichier ascii trafficControl.dat de Dynameq.

Revenons maintenant sur les fichiers en entrée pour ce programme. Pour exécuter, il faut entrer la commande suivante : java Ntw_D control. Le fichier control est le fichier ascii original exporté de Dynameq du plan de trafic. Après optimisation dans Synchro, les phases sont toutes restées les mêmes, mais les plans ont changé. L'idée est donc de faire une copie du fichier control au fichier trafficControl.dat, mais en modifiant les plans lors du passage d'un fichier à l'autre.

Voici maintenant la structure générale du programme. Les méthodes sont expliquées en détail.

void loadOffsets()

Cette méthode va lire dans le fichier Timing.CSV les délais de chaque intersection ayant un plan de trafic et les met dans un vecteur auquel on aura accès plus tard.

void writeTraffic(String ControlDataFileName)

Cette méthode reçoit en paramètre le fichier "ControlDataFileName", qui représente le fichier ascii original exporté de Dynameq du plan de trafic. Elle parcourt ce fichier mot à mot et les copie dans un fichier output trafficControl.dat. Lorsqu'elle rencontre le mot "NODE", elle garde en mémoire le numéro du nœud courant. Lorsqu'elle

rencontre le mot “PLAN”, c’est là qu’il y a un traitement à faire. D’abord, la valeur du délai du nœud courant est prise à partir du vecteur et copiée sur la première ligne du plan. Ensuite, elle lit les lignes de ce plan et laisse la méthode **printTrafficPlan(String nodeID, int stateNumber)** s’occuper d’imprimer ces lignes changées. Elle continue de faire ainsi jusqu’à la fin du fichier. Le résultat obtenu est une copie conforme du fichier “ControlDataFileName”, à l’exception des plans qui ont été changés.

void printTrafficPlan(String nodeID, int stateNumber)

Cette méthode reçoit en paramètres le numéro du nœud courant, ainsi qu’un index relatif du numéro de la phase courante. Elle cherche d’abord dans le fichier PHASING.CSV la première ligne contenant le numéro du nœud courant. Elle cherche ensuite la première colonne contenant la phase courante, grâce à l’index relatif de la phase courante. Elle lit dans le fichier au bon endroit pour obtenir les temps de “green”, “yellow” et “red” pour chaque phase. Ensuite, elle imprime tout ceci dans le fichier de sortie tafficControl.dat.

3.4 Exemple d'exécution

Voici un exemple de séquence d'instructions à exécuter, pour une itération du passage de Dynameq vers Synchro et ensuite de Synchro vers Dynameq. Il faut préalablement avoir lancé une simulation dans Dynameq.

1) Dans Dynameq, exporter les fichiers *reseau.dat*, *control* en format ascii.
2) Convertir le réseau de Dynameq ainsi que le plan des contrôles en exécutant, sur la ligne de commande : **java Ntw_S reseau.dat control**. Ceci produit quatre fichiers en sortie : *LAY.CSV, LAN.CSV, TIM.CSV, PHA.CSV.*
3) Après une simulation sur une heure dans Dynameq, exporter le fichier des volumes *movementcount.out* en format ascii.
4) Convertir les volumes de Dynameq en format CSV en exécutant, sur la ligne de commande : **java Vol_S mvoementcount.out**. Ceci produit 1 fichier en sortie : *VOL.CSV.*
5) Dans Synchro, charger les cinq fichiers en format CSV pour obtenir le réseau, ses contrôles, ainsi que ses volumes.
6) Optimiser les contrôles dans Synchro. Ensuite, écrire les fichiers *PHASING.CSV* ainsi que *TIMING.CSV.*
7) Convertir les contrôles de Synchro en exécutant, sur la ligne de commande : **java Ntw_D control**. Ceci produit 1 fichier en sortie : *trafficControl.dat.*
8) Charger le fichier *trafficControl.dat* dans Dynameq. On a maintenant de nouveaux contrôles, définis par Synchro.
9) Pour faire une nouvelle itération, lancer une simulation avec les nouveaux contrôles et répéter les étapes 3 à 8. Il n'est pas nécessaire de refaire les étapes 1 et 2, puisque le réseau est déjà chargé dans Synchro.

3.5 Limites du programme et améliorations futures

L'interface contient certaines limites sur les rôles qu'elle peut jouer. Les limites suivantes sont réglables en changeant manuellement l'information sur le réseau ainsi que sur le plan de trafic dans Synchro. Certaines d'entres elles peuvent causer des erreurs durant l'exécution d'un des programmes. Par contre, aucune n'affecte l'exécution du programme sur les deux réseaux testés au chapitre 4.

1) Les mouvements de virage de demi-tour sont ignorés et on suppose qu'ils ne sont pas supposés se produire
2) Les intersections de plus de quatre liens entrants ou de plus de quatre liens sortants ne sont pas supportés et il faudrait beaucoup de changements pour pouvoir les inclure. Ils causent un message d'avertissement et il est ensuite possible de les ajuster manuellement dans Synchro.
3) Les angles des liens reliés à une intersection signalisée sont essentiels pour une exécution correcte du programme. Si deux liens ont la même direction cardinale, les informations sur les voies seront erronées.
4) Les courbes ne sont pas supportées. Il est à noter que cette limite est la principale cause d'erreurs potentielles de la limite décrite en 3). Il n'existe pas vraiment de façon de convertir les courbes de Dynameq à Synchro, car Synchro ne les supporte pas.

Chapitre 4 - Essais numériques

Voici des essais numériques qui ont été faits sur deux réseaux. Le premier est un réseau simple avec seulement quelques liens et quelques nœuds. Le deuxième est un réseau représentant une partie du sud de la ville de Calgary.

4.1 Réseau simple

4.1.1 Description du réseau

Commençons les essais numériques par un réseau très simple. Le but de travailler sur un tel réseau est de pouvoir comparer les différentes optimisations de Synchro ainsi qu'une solution intuitive qu'on peut se faire à partir de ce dernier.

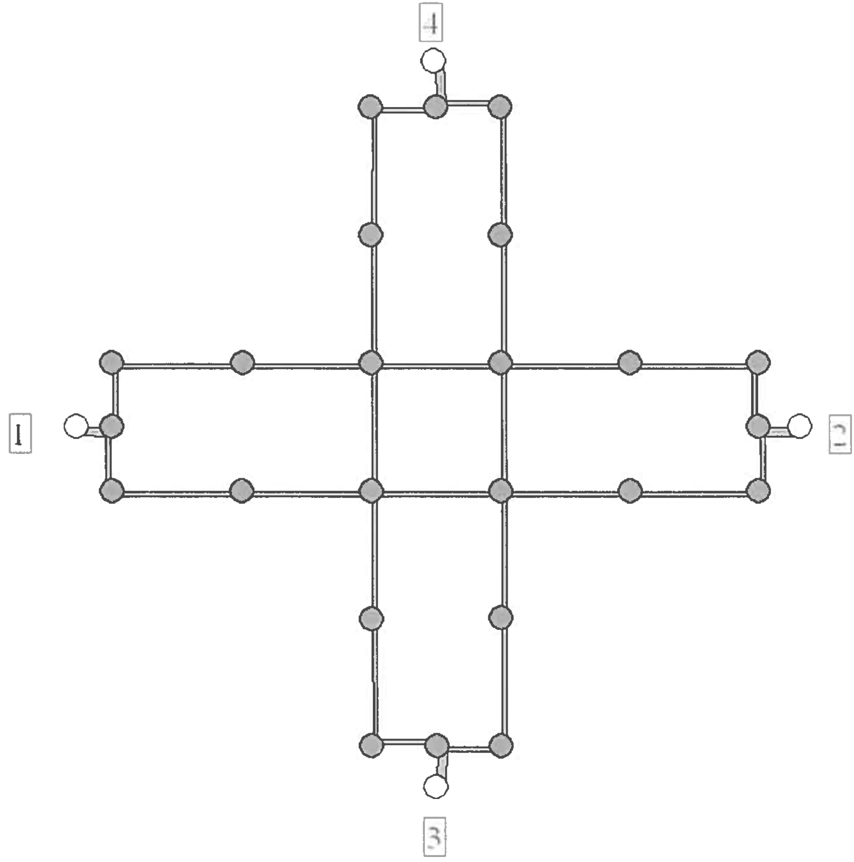


Figure 4.1 : Réseau simple dans Dynameq

Le réseau est constitué de seulement quatre intersections signalisées et elles sont situées au centre. Il y a deux paires O-D, soit du centroïde 1 à 2 et du centroïde 3 à 4. Il y a donc un flot partant de la gauche allant vers la droite et un flot partant du bas allant vers le haut. Aux intersections centrales, il n'est donc possible que d'aller de l'avant et les virages sont interdits. Les liens sont tous de longueur égale de 100 mètres et il n'y a qu'une seule voie pour y circuler. La demande pour les deux paires O-D est égale et est fixée à 1000. Comme on peut le remarquer, nous nous sommes organisés pour que les distances et les demandes entre les deux paires O-D soient égales. De cette façon, une solution intuitive de trouver des plans qui minimisent la congestion peut être donnée, soit celle de donner un temps égal de feu vert aux quatre intersections.

4.1.2 Les plans initiaux

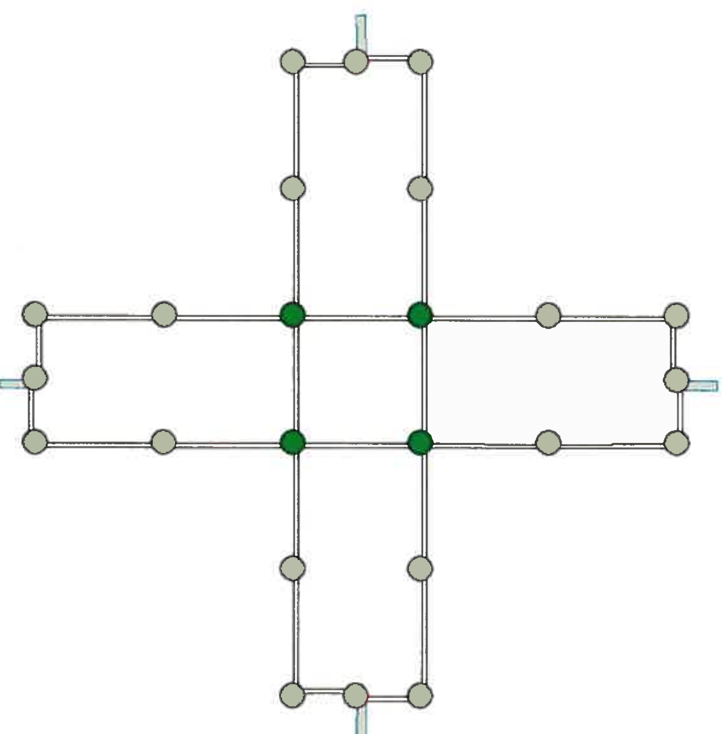


Figure 4.2 : Plans du réseau simple dans Dynameq

Sur la figure 4.2, les points verts correspondent aux intersections signalisées. Ces dernières contiennent toutes un plan identique. Pour chacun des plans, il existe deux

phases. Durant la première le seul mouvement permis est d'aller de gauche à droite et de bas en haut. Les mouvements sont illustrés sur la figure 4.3.

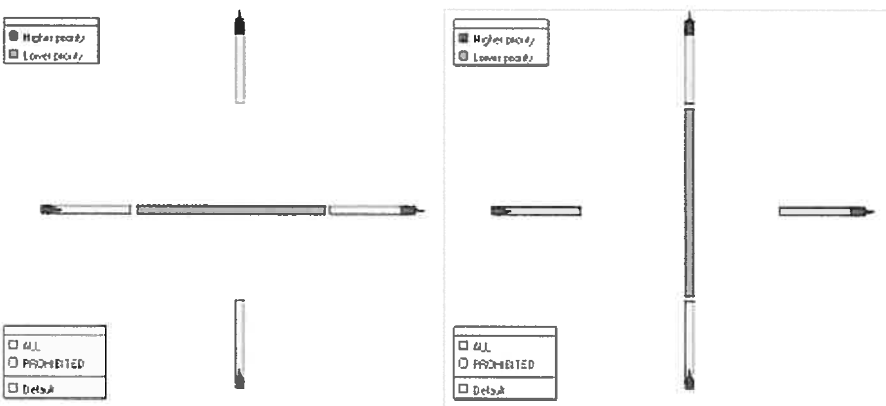


Figure 4.3 : Les mouvements des 4 intersections du réseau simple dans Dynameq

Pour chaque phase, il y a quatre secondes de feu jaune ainsi qu'une seconde où tous les feux sont rouges. Les plans de base pour chaque intersection sont les mêmes. En phase 1, il y a 40 secondes de feu vert pour aller de la gauche vers la droite. En phase 2, il y a 20 secondes de feu vert pour aller de bas en haut. La longueur du cycle est donc de 70 secondes pour chacune des 4 intersections. Nous n'avons pas mis de délai aux plans et fixé sa valeur à 0. Dans le tableau 4.1, on y trouve les temps en secondes de feu vert alloués à chaque phase ainsi que la longueur du cycle en secondes aux quatre intersections du réseau

Original	Phase 1	Phase 2	Cycle
Int 1	40	20	70
Int 2	40	20	70
Int 3	40	20	70
Int 4	40	20	70

Tableau 4.1 : Les 4 plans des intersections du réseau simple dans Dynameq

4.1.3 Les itérations

Après avoir lancé une simulation dans Dynameq avec les données originales mentionnées ci-haut, il y a deux résultats qui nous intéressent : les mouvements de

virage à chacune des quatre intersections et la matrice des temps moyens de déplacement de chaque voiture.

Le premier résultat est directement importé dans Synchro, une fois converti en fichier d'entrée de volumes. À l'aide du deuxième résultat, on obtient le temps total de déplacement des véhicules dans le réseau. C'est ce dernier résultat qu'on va comparer d'une simulation à l'autre dans Dynameq.

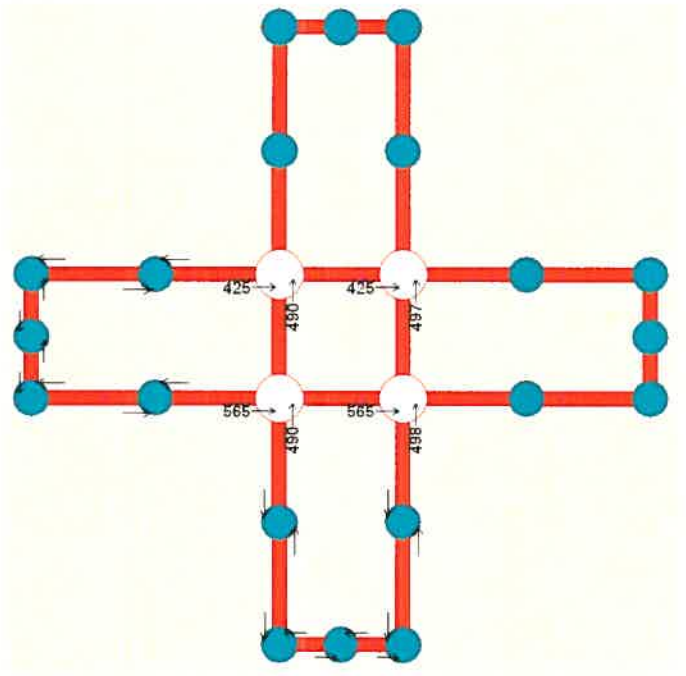


Figure 4.4 : Réseau simple dans Synchro, après une première simulation dans Dynameq

Plusieurs plans doivent être optimisés et nous avons besoin de Synchro pour le faire. Pour obtenir ces derniers, on doit donner à Synchro toutes les informations nécessaires : le réseau, le plan original et les mouvements de virage. Une fois le tout chargé dans Synchro, on peut optimiser les plans de quatre façons :

1) **Optimiser les délais** : Pour chaque intersection, seuls les délais sont optimisés.
2) **Optimiser les contrôles et les délais** : Les temps de feu vert ainsi que les délais sont optimisés pour chaque intersection.
3) **Optimiser d'abord la longueur du cycle pour toutes les intersections et optimiser les feux et les délais** : Synchro trouve une longueur de cycle

optimale pour toutes les intersections et va ensuite optimiser les feux ainsi que les délais.

4) **Solution intuitive** : Une solution intuitive décrite en 4.1.1 est donnée et nous supposons par hypothèse qu'elle est optimale. Nous avons d'abord laissé Synchro trouver la longueur du cycle optimal et par la suite, nous avons attribué à chaque intersection un temps égal de feu vert pour chaque phase.

Lorsque Synchro a terminé l'optimisation, on utilise l'interface pour générer le fichier du plan de trafic dont Dynameq a besoin pour charger les nouveaux plans. Avec ce nouveau plan, on lance à nouveau une simulation, ce qui nous permet d'obtenir les résultats de la même façon qu'avec la première simulation.

En tout, neuf simulations sont lancées dans Dynameq. Dans ce qui suit, une description des paramètres de chaque simulation est donnée. En dessous des colonnes des phases, on y retrouve le nombre de secondes de feu vert qui leur sont allouées. Les temps de feu jaune et rouge sont ignorés, puisqu'ils ne sont pas changés. Ils restent donc à leurs valeurs de base, soient 4 secondes et 1 seconde respectivement.

Orginal : Une simulation est lancée dans Dynameq avec des plans de base déjà définis plus haut dans ce document.

Original	Phase 1	Phase 2	Délai	Cycle
Int 1	40	20	0	70
Int 2	40	20	0	70
Int 3	40	20	0	70
Int 4	40	20	0	70

Tableau 4.2 : Plan de trafic de la simulation Original dans le réseau simple de Dynameq

Délai : À partir des résultats de la simulation originale, Synchro optimise seulement les délais. Une simulation est lancée dans Dynameq avec les nouveaux plans.

Délai	Phase 1	Phase 2	Délai	Cycle
Int 1	40	20	7	70
Int 2	40	20	7	70
Int 3	40	20	0	70
Int 4	40	20	0	70

Tableau 4.3 : Plan de trafic de la simulation Délai dans le réseau simple de Dynameq

Feux : À partir des résultats de la simulation originale, Synchro optimise les délais et les feux. Une simulation est lancée dans Dynameq avec les nouveaux plans.

Feux	Phase 1	Phase 2	Délai	Cycle
Int 1	29	31	2	70
Int 2	28	32	9	70
Int 3	32	28	78	70
Int 4	31	29	4	70

Tableau 4.4 : Plan de trafic de la simulation Feux dans le réseau simple de Dynameq

Cycle 1 (45) : À partir des résultats de la simulation originale, Synchro trouve d'abord une longueur de cycle optimale (45 secondes) et optimise ensuite les délais et les feux. Une simulation est lancée dans Dynameq avec les nouveaux plans.

Cycle 1	Phase 1	Phase 2	Délai	Cycle
Int 1	16	19	54	45
Int 2	16	19	6	45
Int 3	18	17	48	45
Int 4	18	17	0	45

Tableau 4.5 : Plan de trafic de la simulation Cycle 1 dans le réseau simple de Dynameq

Cycle 2 (48) : À partir des résultats de la simulation Cycle 1, Synchro trouve d'abord une longueur de cycle optimale (48 secondes) et optimise ensuite les délais et les feux. Une simulation est lancée dans Dynameq avec les nouveaux plans.

Cycle 2	Phase 1	Phase 2	Délai	Cycle
Int 1	16	22	2	48
Int 2	19	19	10	48
Int 3	19	19	54	48
Int 4	21	17	5	48

Tableau 4.6 : Plan de trafic de la simulation Cycle 2 dans le réseau simple de Dynameq

Cycle 3 (44) : À partir des résultats de la simulation Cycle 2, Synchro trouve d'abord une longueur de cycle optimale (44 secondes) et optimise ensuite les délais et les feux. Une simulation est lancée dans Dynameq avec les nouveaux plans.

Cycle 3	Phase 1	Phase 2	Délai	Cycle
Int 1	19	15	0	44
Int 2	13	21	3	44
Int 3	21	13	48	44
Int 4	16	18	52	44

Tableau 4.7 : Plan de trafic de la simulation Cycle 3 dans le réseau simple de Dynameq

Cycle 4 (46) : À partir des résultats de la simulation Cycle 3, Synchro trouve d'abord une longueur de cycle optimale (46 secondes) et optimise ensuite les délais et les feux. Une simulation est lancée dans Dynameq avec les nouveaux plans.

Cycle 4	Phase 1	Phase 2	Délai	Cycle
Int 1	26	19	64	55
Int 2	14	31	9	55
Int 3	32	13	56	55
Int 4	20	25	0	55

Tableau 4.8 : Plan de trafic de la simulation Cycle 4 dans le réseau simple de Dynameq

Intuition 1 : Les résultats de Cycle 2 donnant le plus petit temps total de déplacement des voitures jusqu'à maintenant (1936 secondes), nous fixons la longueur du cycle à 48 secondes, de mettre des temps de feu vert égaux aux 4 intersections et de fixer les délais à 0 seconde. Une simulation est lancée dans Dynameq avec ces plans.

Intuition	Phase 1	Phase 2	Délai	Cycle
Int 1	19	19	0	48
Int 2	19	19	0	48
Int 3	19	19	0	48
Int 4	19	19	0	48

Tableau 4.9 : Plan de trafic de la simulation Intuition 1 dans le réseau simple de Dynameq

Intuition 2 : À partir des résultats de la simulation Intuition 1, Synchro optimise seulement les délais. Une simulation est lancée dans Dynameq avec les nouveaux plans.

Intuition 1	Phase 1	Phase 2	Délai	Cycle
Int 1	19	19	0	48
Int 2	19	19	8	48
Int 3	19	19	51	48
Int 4	19	19	1	48

Tableau 4.10 : Plan de trafic de la simulation Intuition 2 dans le réseau simple de Dynameq

4.1.4 Les résultats

Pour chaque simulation de Dynameq, à l'aide de la matrice des temps de déplacement moyen des voitures, nous avons calculé le temps total de déplacement des voitures. Comparons maintenant ces différentes valeurs pour chaque simulation. Les résultats sont bien illustrés dans le tableau 4.11. Les nombres sont tous en secondes.

Dem: 1000	(1, 2)	(3, 4)	Total
Original	886.79	1274.85	2161.64
Délai	889.73	1143.89	2033.63
Feux	988.14	1015.44	2003.57
Cycle 1	1042.21	900.89	1943.10
Cycle 2	961.66	974.36	1936.02
Cycle 3	937.41	1093.16	2030.57
Cycle 4	1057.41	920.76	1978.17
Intuition 1	962.78	961.80	1924.58
Intuition 2	973.78	948.69	1922.48

Tableau 4.11 : Temps total de déplacement des voitures dans le réseau simple de Dynameq

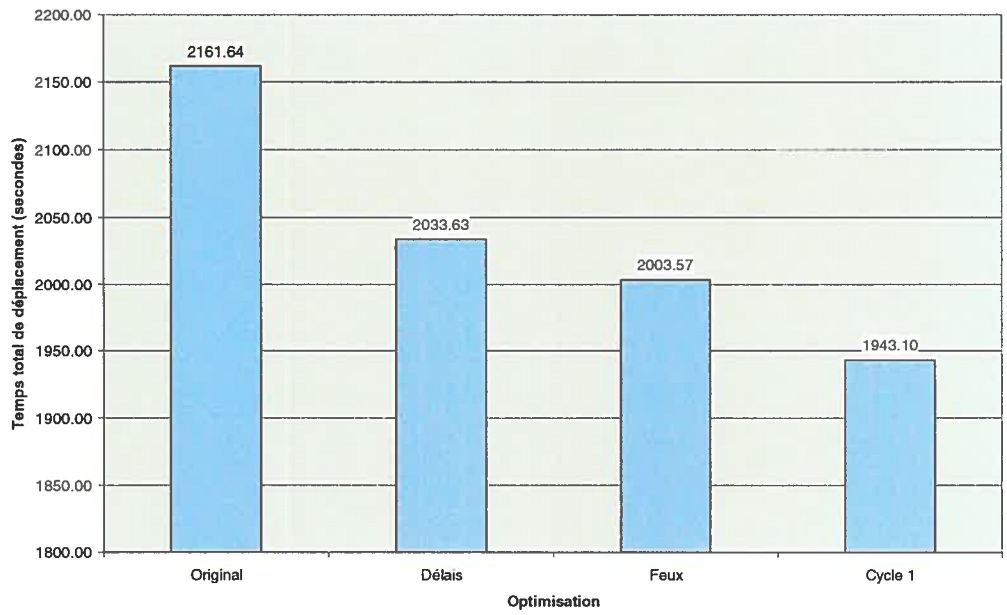


Figure 4.5 : Temps total de déplacement sous trois optimisations dans le réseau simple de Dynameq

Les histogrammes de la figure 4.5 représentent les temps de déplacement sous différentes optimisations. Le premier résultat, celui de la simulation Original, donne un temps de 2161.64 secondes. On peut constater que les trois sortes d'optimisations de Synchro (Délais, Feux et Cycle 1) donnent des meilleurs résultats. Parmi ces dernières, la meilleure est de loin Cycle 1 (1943.10 secondes), où on optimise d'abord la longueur du cycle et ensuite les délais ainsi que les feux.

Comparaison des temps de déplacement sous les optimisaions Cycle 1 à Cycle 4

Figure 4.6 : Temps total de déplacement sous les optimisations Cycle 1 à Cycle 4 dans le réseau simple de Dynameq

Cycle 1 donnant le plus petit temps total de déplacement des voitures, nous avons donc tenté d'améliorer cette solution en faisant trois itérations supplémentaires (Cycle 2, Cycle 3, Cycle 4). Comme nous pouvons le constater sur la figure 4.6, c'est en Cycle 2 que nous trouvons la meilleure solution (1936.02 secondes).

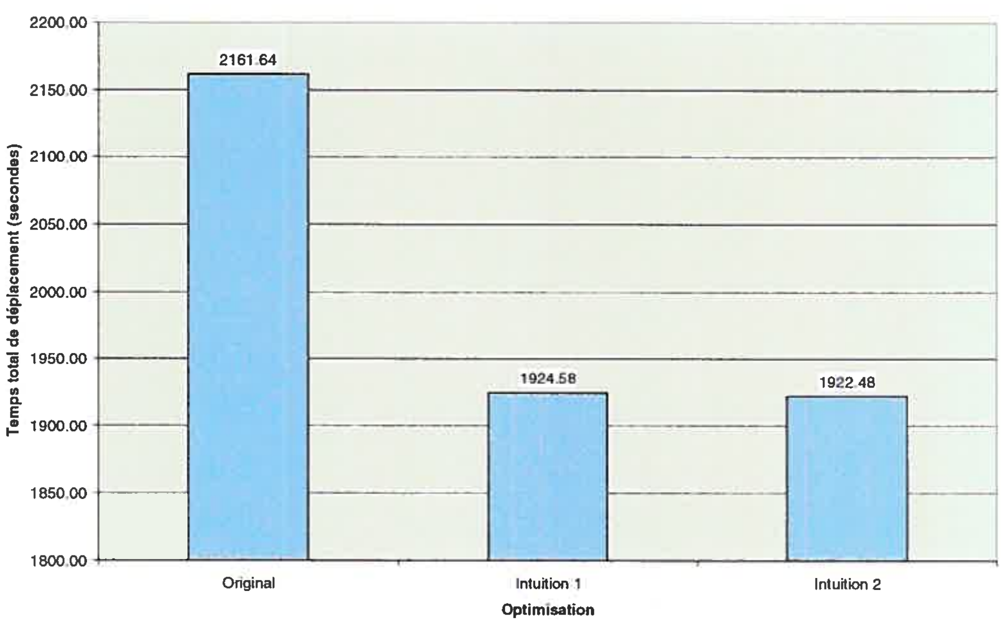


Figure 4.7 : Temps total de déplacement sous les optimisations Intuition 1 et 2 dans le réseau simple de Dynameq

Pour comparer les solutions trouvées par Synchro à la solution intuitive que nous pouvons construire sur un réseau simple, nous avons lancé deux autres simulations, illustrées sur la figure 4.7. D'abord, on se base sur la longueur de cycle optimale trouvée par Synchro et on fixe un temps égal de feu vert aux quatre intersections. Cette simulation, Intuition 1, nous donne un temps de 1924.58 secondes, légèrement mieux que Cycle 2. Une itération supplémentaire a été faite à partir de Intuition 1 et nous avons en Intuition 2 un temps de 1922.48 secondes, le meilleur temps parmi les neuf simulations.

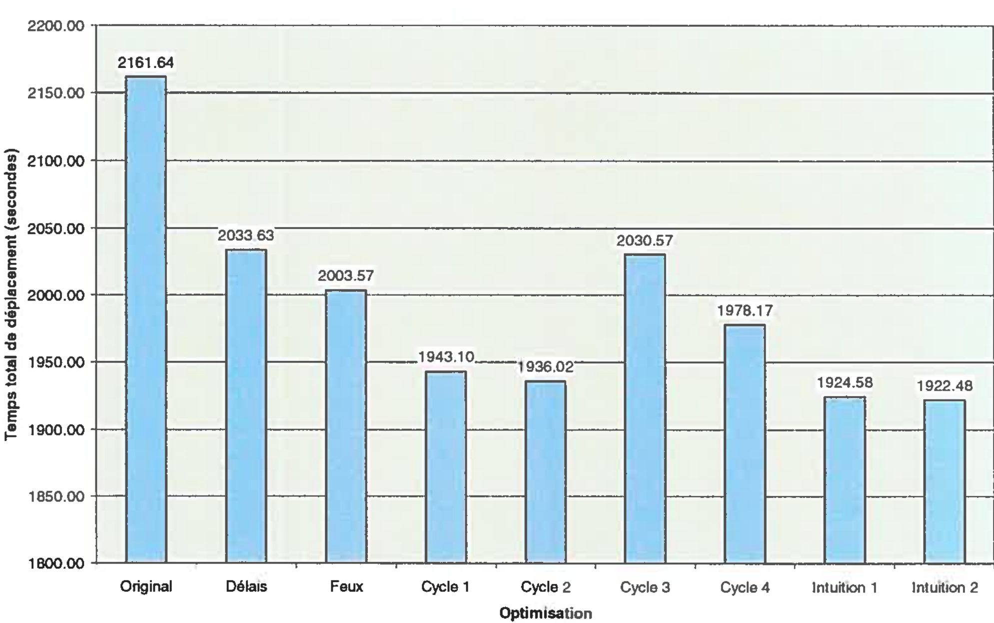


Figure 4.8 : Temps total de déplacement sous toutes les optimisations dans le réseau simple de Dynameq

La figure 4.8 donne le temps total de déplacement sous toutes les optimisations. En résumé, Synchro nous permet de trouver une très bonne solution en modifiant les plans du réseau de la façon suivante : optimiser d'abord la longueur du cycle et par la suite les délais, ainsi que les feux. La solution intuitive mène toutefois à un plus petit temps total de déplacement, ce qui nous pousse à croire que celle-ci est la solution optimale pour ce réseau simple. Même si la meilleure solution trouvée par Synchro n'est pas la solution optimale, nous pouvons dire que la procédure d'itérations entre Synchro et Dynameq est un succès pour le réseau simple, car la congestion a été diminuée de beaucoup avec cette dernière.

4.2 Réseau de Calgary

4.2.1 Description du réseau

Voici le réseau de Calgary, qui représente une partie se retrouvant au sud de cette ville.

Figure 4.9 : Réseau de Calgary dans Dynameq

Ce réseau est plus complexe que le dernier étudié. Il est constitué de 221 nœuds et de 734 liens. Il y a 1329 paires O-D sur un total de 77 centroïdes, illustrés par des points rouges sur la figure 4.6. Il y a en tout 18 intersections signalisées. Une longue rue principale traverse le réseau du nord au sud et les voitures y circulent dans les deux directions. La congestion se retrouve principalement sur cette rue et les rues transversales.

4.2.2 Les plans initiaux

Figure 4.10 : Centre du réseau Calgary, dans Dynameq

La figure 1.2 montre la partie du réseau où il y a le plus de congestion, soit celle du centre. Les deux intersections majeures sont celles du nœud 3412 (en bas et au centre) et du nœud 3413 (en haut au centre).

Les points verts correspondent aux intersections signalisées et comme on l'a déjà mentionné, il y en a 18 en tout. Contrairement au réseau simple, les plans contiennent des mouvements différents d'une intersection à l'autre et le nombre de phases est variable. Il est inutile d'illustrer toutes les phases pour chacune des 18 intersections ayant des plans. Cependant, contentons-nous d'étudier en détail les plans des intersections 3412 et 3413 et nous tenterons de tirer des conclusions à partir de ceux-ci, au fur et à mesure que Synchro optimise les plans.

Voici les phases de l'intersection 3412. De gauche à droite, on a les phases 1 à 4.

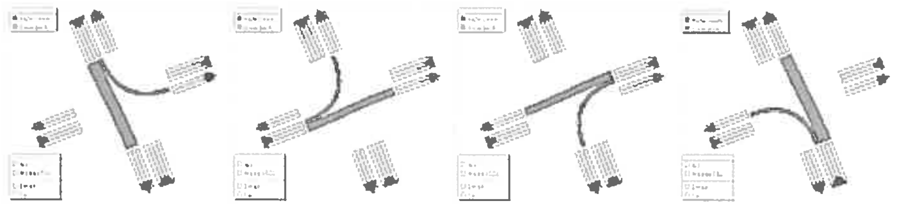


Figure 4.11 : Les 4 phases de l'intersection 3412 du réseau de Calgary dans Dynameq

Durant chaque phase, il y a deux mouvements permis : aller tout droit et tourner à gauche. Pour les voitures venant de l'est et de l'ouest, trois voies sont réservées pour aller tout droit, tandis que pour les voitures venant du nord et du sud, deux voies sont réservées pour aller tout droit. Une voie est réservée aux voitures voulant tourner à gauche. Le mouvement de tourner à droite est interdit à cette intersection, puisqu'il est possible d'emprunter une rampe qui donne le même résultat avant d'arriver à cette intersection. Le plan de l'intersection 3412 est illustré sur la figure 4.12, et les temps sont en secondes.

Phase	1	2	3	4
Vert	27	35.5	15.5	43
Jaune	4	4	4	4
Rouge	2	2	2	2

Tableau 4.12 Le plan de l'intersection 3412 du réseau de Calgary dans Dynameq

Si on regarde le temps de feu vert alloué à chaque phase, ils sont sensiblement équilibrés, sauf la phase 3 qui en a un peu moins.

Voici les phases de l'intersection 3413. De gauche à droite, on a les phases 1 à 5.

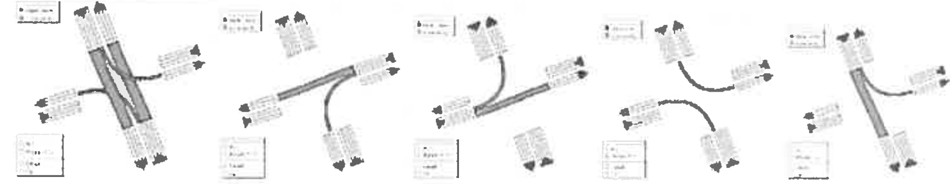


Figure 4.12 : Les 5 phases de l'intersection 3413 du réseau de Calgary dans Dynameq

Comme à l'intersection 3412, les mouvements permis à chaque phase sont d'aller tout droit et de tourner à gauche, sauf à la phase 4 où seuls les tournants à gauche sont permis. 3 voies sont réservées pour aller tout droit pour les voitures provenant du nord et du sud et 2 voies sont réservées pour tourner à gauche pour les voitures provenant de l'est et de l'ouest. Pour la même raison qu'à l'intersection 3412, le mouvement de tourner à droite est interdit à cette intersection.

Phase	1	2	3	4	5
Vert	54.5	11	38	8.5	2
Jaune	4	4	4	4	4
Rouge	2	2	2	2	2

Tableau 4.13 Le plan de l'intersection 3413 du réseau de Calgary dans Dynameq

Si nous regardons le temps (en secondes) de feu vert alloué à chaque phase, on remarque qu'il y a un désiquilibre et que le plus grand temps est donné à la phase 1, tandis que le plus plus petit temps est donné à la phase 5.

4.2.3 Les itérations

Tout comme dans le cas du réseau simple, il y a deux résultats qui nous intéressent après une simulation dans Dynameq, soit les mouvements de virage à chacune des 18 intersections ayant un plan et la matrice des temps moyen de déplacement de chaque paire O-D.

Il faut maintenant laisser place à Synchro pour optimiser les plans des trois façons différentes, expliquées en détail dans le cas du réseau simple (délais, feux et délais, cycle et feux et délais).

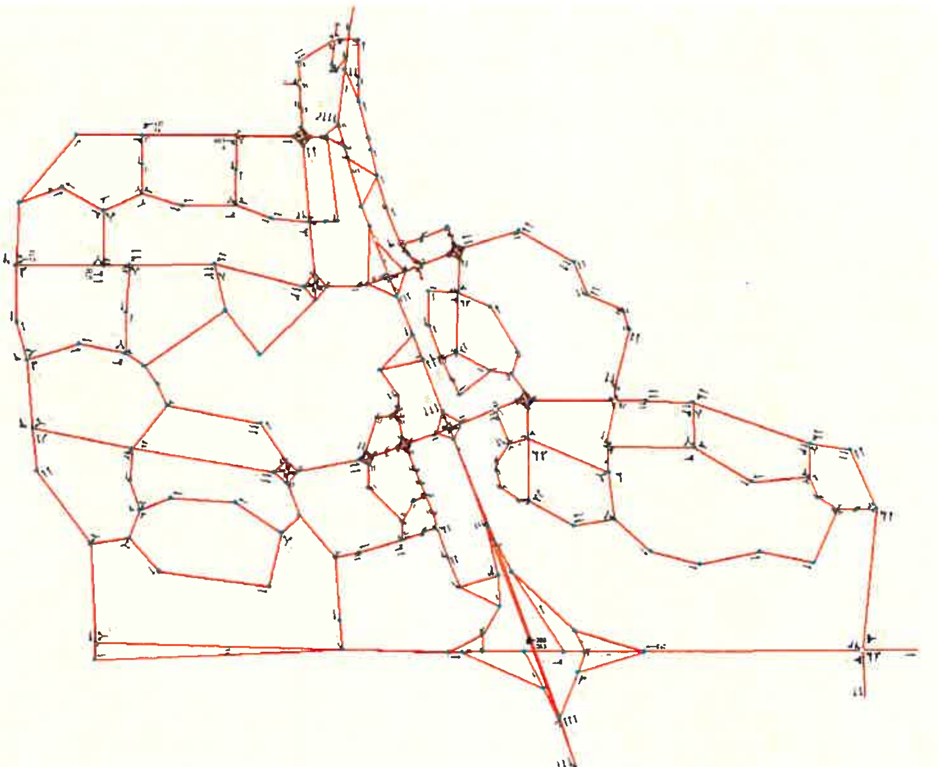


Figure 4.13 : Réseau de Calgary dans Synchro

Tout comme dans le dernier exemple du réseau simple, neuf simulations sont lancées dans Dynameq. Dans ce qui suit, une description des paramètres de chaque simulation est donnée. Tel que mentionné auparavant, nous nous sommes concentrés sur deux intersections majeures, soit les intersections 3412 et 3413.

Année Base : Une simulation est lancée dans Dynameq avec les plans de base déjà définis plus haut dans ce document. Année Base est la simulation de base qui vient avec le réseau de Calgary.

Intersection 3412						Intersection 3413					
Phase	1	2	3	4		Phase	1	2	3	4	5
Vert	27	35.5	15.5	43		Vert	54.5	11	38	8.5	2
Jaune	4	4	4	4		Jaune	4	4	4	4	4
Rouge	2	2	2	2		Rouge	2	2	2	2	2

Tableau 4.14 : Les plans de Année Base

Délai : À partir des résultats de la simulation Année Base, Synchro optimise seulement les délais. Comme on peut le remarquer, Synchro juge que la solution optimale est de ne pas avoir de délai, donc aucun changement aux plans de base. Une simulation non nécessaire (puisque les résultats seront les mêmes) est lancée dans Dynameq avec cette optimisation.

Intersection 3412						Intersection 3413					
Phase	1	2	3	4		Phase	1	2	3	4	5
Vert	27	35.5	15.5	43		Vert	54.5	11	38	8.5	2
Jaune	4	4	4	4		Jaune	4	4	4	4	4
Rouge	2	2	2	2		Rouge	2	2	2	2	2

Tableau 4.15 : Les plans de Délai

Feux : À partir des résultats de la simulation Année Base, Synchro optimise les délais et les feux. Une simulation est lancée dans Dynameq avec cette optimisation.

Intersection 3412					Intersection 3413					
Phase	1	2	3	4	Phase	1	2	3	4	5
Vert	23	46	5	42	Vert	56	3	37	16	2
Jaune	4	4	4	4	Jaune	4	4	4	4	4
Rouge	2	2	2	2	Rouge	2	2	2	2	2

Tableau 4.16 : Les plans de Feux

Feux 1 : À partir des résultats de la simulation Feux, Synchro optimise les délais et les feux. Une simulation est lancée dans Dynameq avec cette optimisation.

Intersection 3412					Intersection 3413					
Phase	1	2	3	4	Phase	1	2	3	4	5
Vert	11	63	2	40	Vert	52	3	37	19	2
Jaune	4	4	4	4	Jaune	4	4	4	4	4
Rouge	2	2	2	2	Rouge	2	2	2	2	2

Tableau 4.17 : Les plans de Feux 1

Feux 2 : À partir des résultats de la simulation Feux 1, Synchro optimise les délais et les feux. Une simulation est lancée dans Dynameq avec cette optimisation.

Intersection 3412					Intersection 3413					
Phase	1	2	3	4	Phase	1	2	3	4	5
Vert	9	66	2	39	Vert	50	4	38	20	2
Jaune	4	4	4	4	Jaune	4	4	4	4	4
Rouge	2	2	2	2	Rouge	2	2	2	2	2

Tableau 4.18 : Les plans de Feux 2

Feux 3 : À partir des résultats de la simulation Feux 2, Synchro optimise les délais et les feux. Une simulation est lancée dans Dynameq avec cette optimisation.

Intersection 3412					Intersection 3413					
Phase	1	2	3	4	Phase	1	2	3	4	5
Vert	8	67	2	39	Vert	48	4	39	21	2
Jaune	4	4	4	4	Jaune	4	4	4	4	4
Rouge	2	2	2	2	Rouge	2	2	2	2	2

Tableau 4.19 : Les plans de Feux 3

Cycle : À partir des résultats de la simulation Année Base, Synchro trouve d'abord une longueur de cycle optimale (74 et 68 pour les intersections 3412 et 3413) et optimise ensuite les délais et les feux. Une simulation est lancée dans Dynameq avec cette optimisation.

Intersection 3412					Intersection 3413					
Phase	1	2	3	4	Phase	1	2	3	4	5
Vert	8	21	2	19	Vert	19	3	12	2	2
Jaune	4	4	4	4	Jaune	4	4	4	4	4
Rouge	2	2	2	2	Rouge	2	2	2	2	2

Tableau 4.20 : Les plans deCycle

Cycle 1: À partir des résultats de la simulation Cycle, Synchro trouve d'abord une longueur de cycle optimale (57 et 51 pour les intersections 3412 et 3413) et optimise ensuite les délais et les feux. Une simulation est lancée dans Dynameq avec cette optimisation.

Intersection 3412					Intersection 3413					
Phase	1	2	3	4	Phase	1	2	3	4	5
Vert	2	18	2	11	Vert	9	3	5	2	2
Jaune	4	4	4	4	Jaune	4	4	4	4	4
Rouge	2	2	2	2	Rouge	2	2	2	2	2

Tableau 4.21 : Les plans de Cycle 1

Cycle 2: À partir des résultats de la simulation Cycle 1, Synchro trouve d'abord une longueur de cycle optimale (47 et 41 pour les intersections 3412 et 3413) et optimise ensuite les délais et les feux. Une simulation est lancée dans Dynameq avec cette optimisation.

Intersection 3412

Phase	1	2	3	4
Vert	2	13	2	6
Jaune	4	4	4	4
Rouge	2	2	2	2

Intersection 3413

Phase	1	2	3	4	5
Vert	3	2	2	2	2
Jaune	4	4	4	4	4
Rouge	2	2	2	2	2

Tableau 4.22 : Les plans de Cycle 2

4.2.4 Les résultats

Pour chaque simulation de Dynameq, à l'aide de la matrice des temps de déplacement moyen des voitures, nous avons calculé le temps total de déplacement des voitures. Comparons maintenant ces différentes valeurs pour chaque simulation. Les résultats sont bien illustrés dans le tableau 4.21 ainsi que sur la figure 4.11.

Optimisation	Temps
Année Base	73250.8444
Délai	73250.8444
Feux	60338.7104
Feux 1	57783.6133
Feux 2	55954.0236
Feux 3	56314.8908
Cycle	63500.3681
Cycle 1	70633.2407
Cycle 2	81847.6275

Tableau 4.23 : Temps total de déplacement des voitures dans le réseau de Calgary de Dynameq

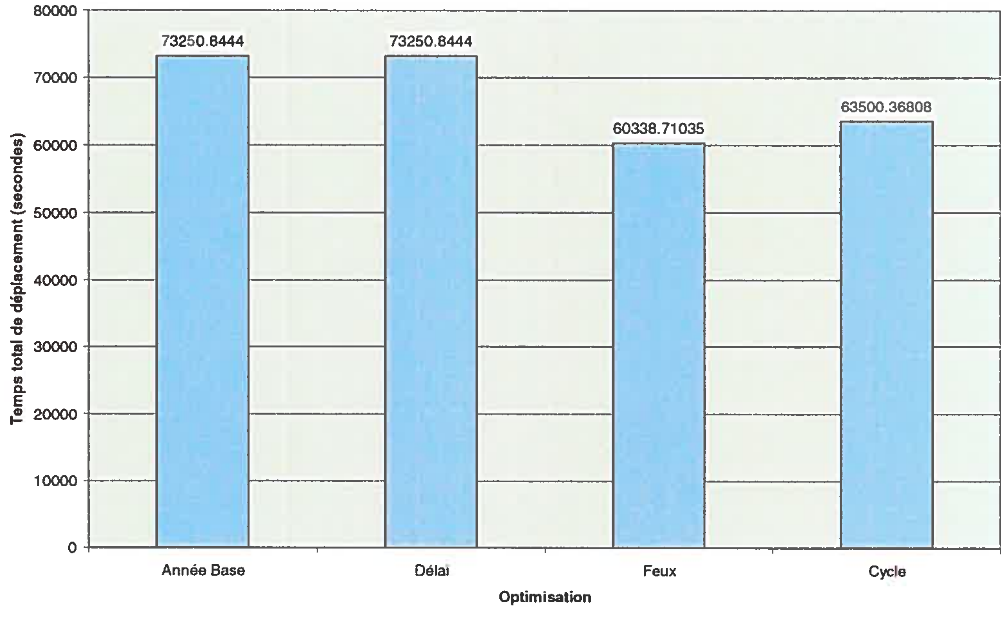


Figure 4.14 : Temps total de déplacement sous trois optimisations dans le réseau de Calgary de Dynameq

Les histogrammes de la figure 4.14 représentent le temps total de déplacement sous trois optimisations dans le réseau de Calgary. Le premier résultat, celui de la simulation Année Base donne un temps de 73251. Les résultats des trois façons d'optimiser par Synchro sont illustrés dans les simulations Délai, Feux et Cycle. Tout d'abord, notons que lorsque Synchro tente d'optimiser les plans au niveau des délais seulement, il juge que la solution optimale est de laisser tous les délais à 0, donc de garder les plans intacts. Il est donc trivial que les résultats de Année Base et Délai soient égaux, puisque c'est une simulation avec des paramètres identiques. Dans les deux autres simulations, Feux et Cycle, le temps total de déplacement des voitures (60339 et 63500) est meilleur que celui de Année Base (73251). Trois itérations supplémentaires ont été lancées à partir de Feux et deux itérations supplémentaires ont été lancées à partir de Cycle afin de déterminer si une des deux approches permet d'arriver à une solution optimale.

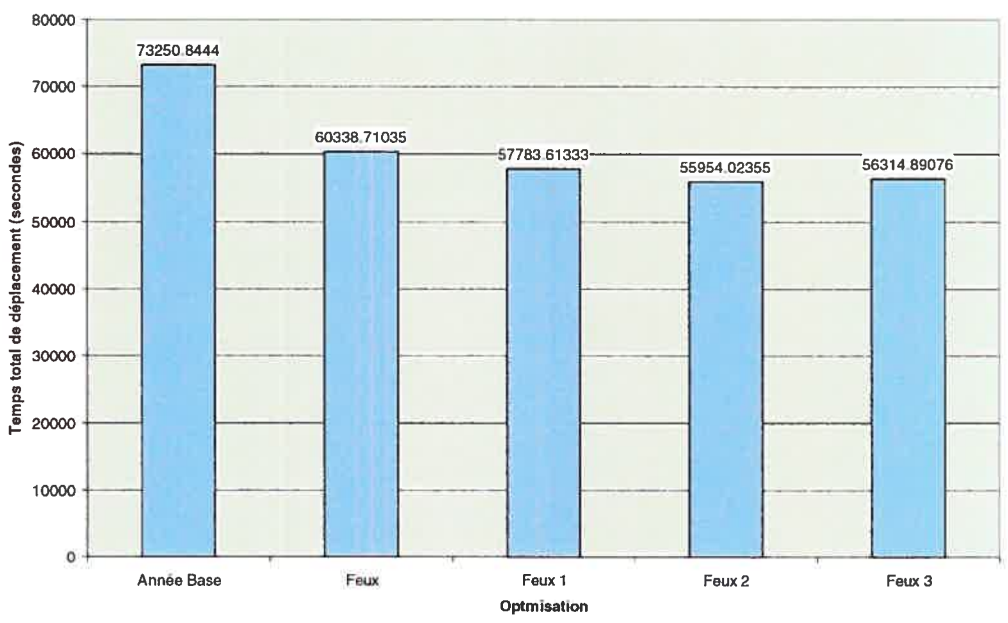


Figure 4.15 : Temps total de déplacement sous les optimisations Feux à Feux 4 dans le réseau de Calgary de Dynameq

Dans la séquence d'itérations lancées à partir de Feux, le temps total de déplacement diminue d'une itération à l'autre et atteint son minimum en Feux 2 (55954), comme nous pouvons le constater sur la figure 4.15. Une itération supplémentaire fait

augmenter la valeur de la solution, ce qui fait que la meilleure solution provient jusqu'à maintenant de la simulation Feux 2.

Figure 4.16 : Temps total de déplacement sous les optimisations Feux à Feux 4 dans le réseau de Calgary de Dynameq

Dans la séquence d'itérations lancées à partir de Cycle, le temps total de déplacement augmente d'une itération à l'autre, ce qui ne fait que créer plus de congestion, comme nous pouvons le constater sur la figure 4.16. Après seulement deux itérations supplémentaires, la valeur de la solution est encore pire que celle de la simulation Année Base.

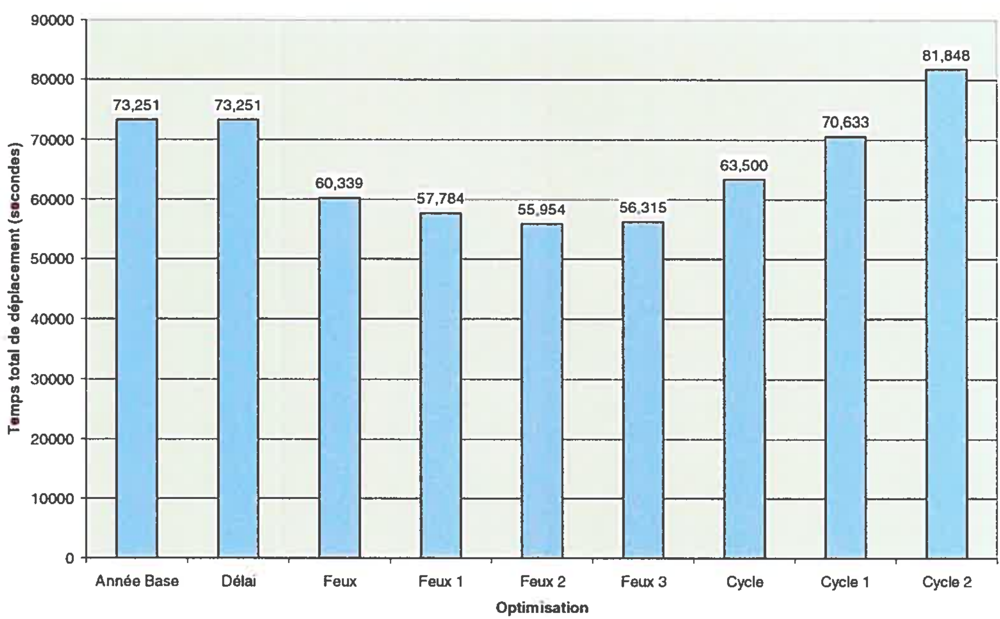


Figure 4.17 : Temps total de déplacement des voitures sous toutes les optimisations dans le réseau de Calgary de Dynameq

La figure 4.17 donne le temps total de déplacement sous toutes les optimisations. Optimiser les cycles n'est pas la bonne approche pour le réseau de Calgary et optimiser seulement les feux et les délais est une approche de loin supérieure. La solution optimale est donc d'optimiser les feux et les délais sur trois itérations et on obtient un temps total de déplacement de 55954, une diminution d'environ 24% par rapport à la solution originale.

Faisons maintenant des comparaisons entre les files d'attentes aux intersections 3412 et 3413, entre les simulations Année Base et Feux 2. Sur la figure 4.12, l'intersection de gauche représente celle de Année Base et celle de droite représente celle de Feux 2.

Intersection 3412

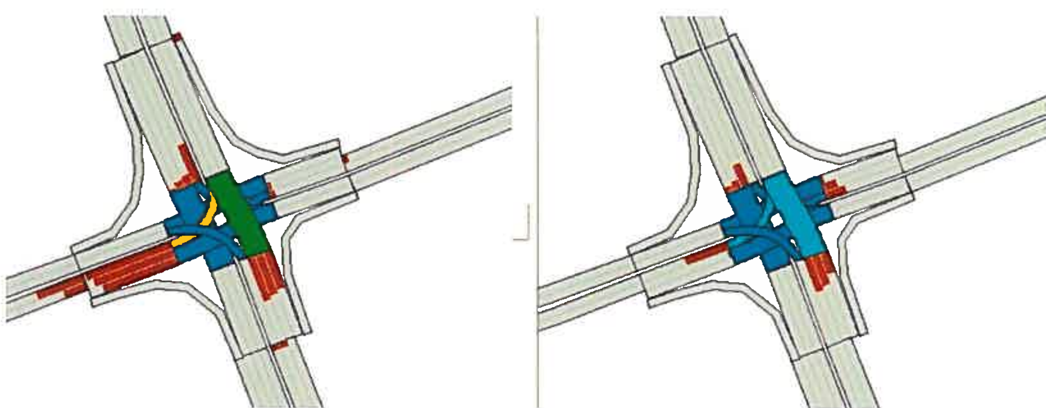


Figure 4.18 : Comparaison de la congestion de deux simulations à l'intersection 3412 du réseau de Calgary

Revenons maintenant sur les plans de Année Base et Feux 2 et tentons de tirer des conclusions sur les phases de l'intersection 3412.

	Année Base					Feux 2			
Phase	1	2	3	4	Phase	1	2	3	4
Vert	27	35.5	15.5	43	Vert	9	66	2	39
Jaune	4	4	4	4	Jaune	4	4	4	4
Rouge	2	2	2	2	Rouge	2	2	2	2

Tableau 4.24 : Les plans de l'intersection 3412 dans Année Base et Feux 2

Comme on peut voir sur la figure 4.12, la plus grosse file d'attente dans Année Base est celle qui arrive de l'ouest. C'est en phase 2 que les voitures de cette file peuvent avancer et on voit dans le tableau 4.22 que le temps alloué à cette phase est passé de 35.5 secondes à 66 secondes dans Feux 2. Aussi, la file d'attente des voitures arrivant de l'est est presque vide dans Année Base. Puisque c'est en phase 3 que les voitures de cette file peuvent avancer, le temps alloué à cette dernière a diminué de 15.5 secondes à 2 secondes dans Feux 2.

Intersection 3413

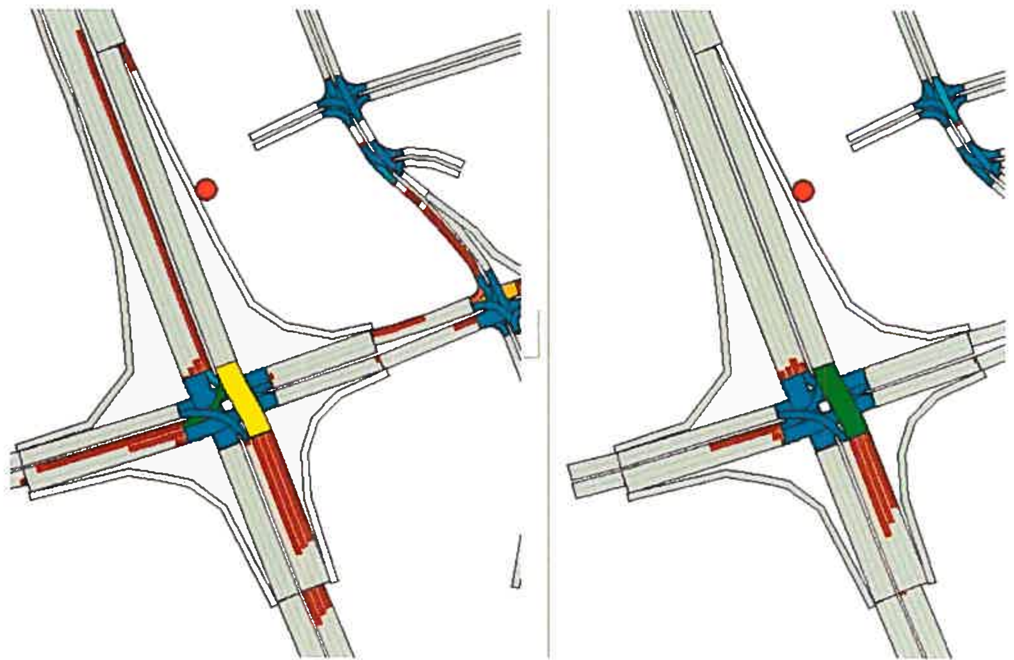


Figure 4.19 : Comparaison de la congestion de deux simulations à l'intersection 3413 du réseau de Calgary

Revenons maintenant sur les plans de Année Base et de Feux 2 et tentons de tirer des conclusions sur les phases de l'intersection 3413.

Année Base						Feux 2					
Phase	1	2	3	4	5	Phase	1	2	3	4	5
Vert	54.5	11	38	8.5	2	Vert	50	4	38	20	2
Jaune	4	4	4	4	4	Jaune	4	4	4	4	4
Rouge	2	2	2	2	2	Rouge	2	2	2	2	2

Tableau 4.25 : Les plans de l'intersection 3413 dans Année Base et Feux 2

La plus grosse file d'attente dans Année Base est celle où les voitures arrivent du nord et veulent tourner à gauche pour se diriger vers l'est. Les phases où ce mouvement est permis sont les phases 1, 4 et 5. Il y a déjà beaucoup de temps alloué à la phase 1, il faut donc augmenter le temps alloué à ce mouvement dans une autre phase. Lors de la phase 5, l'autre mouvement permis est d'aller tout droit, du nord au sud, mais il n'y a presque pas d'attente pour ce mouvement. C'est donc lors de la phase 4 où le temps alloué à cette dernière a été augmenté, passant de 8.5 secondes à 20 secondes. Le reste des phases se ressemble sensiblement.

En résumé, la procédure d'itérations permet de trouver une très bonne solution qui réduit le temps total de déplacement des voitures d'environ 24%. C'est en faisant adopter à Synchro une politique d'optimiser les feux et les délais que nous y parvenons et ce résultat est très significatif au niveau de la baisse de la congestion.

Conclusion

Dans ce travail, nous avions comme objectif de trouver une technique pour diminuer la congestion dans un réseau routier. Puisqu'il est difficile de résoudre ce genre de problème avec un appui théorique très fort, nous avons décidé de le résoudre en utilisant des outils informatiques mis à notre disposition.

Nous avons donc eu recours à l'utilisation d'une approche itérative pour optimiser les contrôles d'un réseau routier. En concevant un programme d'interface entre Synchro et Dynameq, nous avons réussi à utiliser ces deux logiciels comme compléments. Ces derniers ont été deux outils très importants dans notre travail. D'abord, Dynameq a permis de simuler la congestion engendrée par la demande des véhicules sur un réseau. Puis, Synchro a permis d'optimiser les contrôles d'un réseau initial de Dynameq.

En se basant sur une méthode d'ajustement heuristique, les essais numériques utilisant l'approche itérative ont donné de très bons résultats sur deux réseaux différents. Résumons les faits importants de ces essais.

D'abord, nous avons effectué des tests sur un réseau simple. Ce dernier ne contenant que quelques nœuds et quelques liens, l'objectif était de pouvoir comparer différentes simulations utilisant des plans de base, des plans trouvés par l'approche itérative ou des plans trouvés de manière intuitive. La simulation utilisant les plans trouvés par l'approche itérative a donné un résultat qui diminue considérablement la congestion par rapport à celle utilisant les plans de base. Ensuite, nous avons constaté que la simulation utilisant les plans trouvés de manière intuitive a donné le résultat qui diminue le plus la congestion. Néanmoins, l'utilisation de l'approche itérative semble très raisonnable comme technique pour arriver à nos fins.

Ensuite, nous nous sommes attaqués à un réseau de taille, celui de Calgary. Aucune solution intuitive n'a été donnée dans ce cas, étant donné la complexité du réseau. En appliquant l'approche itérative sur ce réseau, différentes politiques d'optimisation ont été testées avec Synchro. Dans le meilleur des cas, nous sommes parvenus à

trouver une solution qui diminue le temps total de déplacement des voitures d'environ 24% par rapport à celui d'origine. Ce résultat est très significatif au niveau de la baisse de la congestion.

Les gens pessimistes pourraient dire qu'étant donné l'existence d'une solution intuitive menant à la plus grande diminution de la congestion dans le réseau simple, il se pourrait très bien qu'il existe une autre solution que celle trouvée par l'approche itérative qui diminue davantage la congestion dans le réseau de Calgary. Cet argument est valide, mais si nous pouvons faire sauver à chaque conducteur en moyenne 24% du temps qu'il lui faut pour se rendre à destination en utilisant l'approche itérative, nous pouvons conclure que celle-ci garantit une certaine fiabilité.

Néanmoins, il y a grande place à amélioration de cette approche itérative. Synchro est un bon outil d'optimisation, mais il a tout de même ses inconvénients. L'un d'entre eux est le fait que les algorithmes utilisés dans ce logiciel ne sont pas disponibles aux usagers. Puisque nous ne connaissons pas les étapes de calcul lors de l'optimisation des contrôles, l'utilisation de Synchro est faite de manière quasi aveugle. Les concepteurs du logiciel n'ont toujours pas révélé ces détails suite à nos demandes. Tout au long du travail, nous avons donc supposé que les méthodes d'optimisation utilisées par Synchro sont les meilleures, mais nous ne pouvons pas en être certains.

En somme, l'utilisation de l'approche itérative est un succès, car les résultats démontrent qu'on peut améliorer la fluidité de la circulation d'un réseau de taille réaliste. Les applications pratiques reliées à cette dernière sont nombreuses, et il serait intéressant de pouvoir la tester sur d'autres réseaux de grandes villes urbaines à travers le monde.

Bibliographie

AIMSUN2: http://www.tss-bcn.com/

CORSIM: http://mctrans.ce.ufl.edu/featured/TSIS/Version5/corsim.htm

DRACULA: http://www.its.leeds.ac.uk/software/dracula/

FLORIAN, M. et HEARN, D.W. (1995), Network Equilibirum Methods. *Operations Research* – Network Routing, 8, pp. 485-550.

FLORIAN, M. et HEARN, D.W. (1999), Network Equilibirum and Pricing. *Transportation Science*, Frederick S. Hillier (ed.). Kluwer Academic Publishers, pp. 361-393.

FLORIAN, M., MAHUT, M., TREMBLAY, N., (2005). A Simulation Based Dynamic Traffic Assignement : The Model, Solution Algorithm and Applications. Article présenté à la conférence DTA 2006, Leeds, Angleterre.

FRIESZ, T. L. et AL. (1993). A Variational Inequality Formulation of the Dynamic Network Traffic Equilibrium Problem, *Operations Research*, 41(1), pp. 179-191.

LCN. (2006). Les feux de circulation seront synchronisés, http://lcn.canoe.com/lcn/regional/archives/2006/08/20060828-084019.html, Montréal.

MAHUT, M. (2000). Discrete Flow Model For Dynamic Network Loading. Ph.D. Thesis, Département d'informatique et de recherhe opérationelle, Université de Montréal. Published by the Center for Research on Transportation (CRT), University of Montreal.

MITSIM: http://mit.edu/its/mitsimlab.html/

PARAMICS: http://www.paramics-online.com/home/home.htm

SCHERR, W., ADAMS, D., BAUER, T. (2003) An Integrated Model For Planning and Traffic Engineering, Ninth TRB planning methods applications conference, Louisiana.

VISSIM: http://www.vissim.com/

Annexe 1 – Les classes de l'interface entre Synchro et Dynameq

Voici en ordre alphabétique, les 10 classes de l'interface entre Synchro et Dynameq.

CSVReader

```
/*****
Date: June 8 2005
Returns the content of a cell from an excel file in format CSV.
Requires file In.java to work.

TYPICAL USEAGE

CSVRead csvr = new SCV("LAYOUT.csv");
String DATA = csvr.readCell(1, 1);
if (DATA != null)
   //... use DATA ...

****/

    class CSVReader {

      private String DataFile;
      private int size;

   // construct a reader for a synchro file given from standard input
   // (command line argument).
       public CSVReader() {
         In in = new In();
         DataFile = in.readAll();
         size = DataFile.length();
      }

   // construct a reader for a synchro file named file.
       public CSVReader(String file){
         In in = new In(file);
         DataFile = in.readAll();
         size = DataFile.length();
      }

   // return a string contained in the file of SynchroRead
   // at (row, col) as specified by caller. Return null if cell doesn't exist
       public String readCell(int row, int col){

         int charpos = 0;

      // skip until we're at the line we want and read it
         for (int i = 1; i < row; i++) {
            charpos = readLine(charpos);
            if (charpos >= size)
               return null;
         }

      // skip until we're at the column we want and read
         int commas = 1;

         for (; charpos < size && commas != col; charpos++) {
            if (DataFile.charAt(charpos) == ',')
               commas++;
            if (DataFile.charAt(charpos) == '\n')
               return null;
         }

      // check if column was reached
         if (commas != col)
            return null;
```

```
      // read the cell in correct row and column
         StringBuffer output = new StringBuffer();
         for (; charpos < size; charpos++) {
            char c = DataFile.charAt(charpos);
            if (c == ',' || c == '\n')
               break;
            else output.append(c);
         }

         return new String(output);

      }

       public boolean isEmpty(int row, int col)
      {
         return (readCell(row, col) == null);
      }

       public int readInt(int row, int col)
      {
         String s = readCell(row, col);
      //-1 being an empty value for int
         if (s == null)
            return -1;
         return Integer.parseInt(s);
      }
       public double readDouble(int row, int col) {
         return Double.parseDouble(readCell(row, col)); }

   //return character position after one line of reading
       private int readLine(int charpos) {
         int pos = charpos;
         while (pos < size && DataFile.charAt(pos) != '\n') {
            pos++;
         }
         return pos + 1;
      }

   // testing the class with file piped through stdin
   // the first and second command line arguments are row and column
       public static void main (String args[]) {
         int row = Integer.parseInt(args[0]);
         int col = Integer.parseInt(args[1]);

         CSVReader csvr = new CSVReader();

         if (csvr.readCell(row, col) == null)
            System.out.println("error: null");
         else {
            if (csvr.readCell(row, col).length() == 0)
               System.out.println("warning: empty");
            else
               System.out.println(csvr.readCell(row, col));
            if (!csvr.readCell(row, col).equals(csvr.readCell(row, col)))
               System.out.println("error: inconsistent program");
         }

      }

   }
```

CSVWriter

```
/*****
Date: June 8 2005
Return a string formatted in CSV (Comma Separated Values) containing the
data given in a 2D matrix of strings.
A null cell is considered the last of a row.

TYPICAL USEAGE

int rowdim = ...;
```

```
int coldim = ...;
int[][] matrix = new int[rowdim][coldim]
matrix = {.......}

String output = CSVWriter.write(matrix, rowdim, coldim);

System.out.println(matrix)

****/

class CSVWriter {

//Return a string formatted in CSV (Comma Separated Values) containing the
//data given in a 2D matrix of objects which have string representations.
//A null cell is considered the last of a row.
   public static String write(String[][] matrix, int rowdim, int coldim) {
      int row = 0;
      int col = 0;
      String output = "";

      while(true) {

         if (row >= rowdim) break;

         String cell = matrix[row][col];

         if (cell == null) {
            col = 0;
            row++;
            output = output.concat("\n");
         }
         else {
            output = output.concat(cell);
            col++;
         }

         if (col >= coldim) {
            col = 0;
            row++;
            output = output.concat("\n");
         }
         else
            output = output.concat(",");

      }

      return output;

   }

   //testing
   public static void main (String args[]) {
      int rowdim = 3;
      int coldim = 4;
      String[][] matrix = new String[rowdim][coldim];

      for (int i = 0; i < rowdim; i++) {
         for (int j = 0; j < coldim; j++) {
            matrix[i][j] = (new Integer(i*j)).toString();
         }
      }

      String output = CSVWriter.write(matrix, rowdim, coldim);

      System.out.println(output);

   }

}
```

In

```
/*************************************************************************
 *  Compilation:  javac In.java
 *  Execution:    java In
 *
 *  Reads in data of various types from: stdin, file, URL.
 *
 *  Reads in a string, rest of line, or rest of file and returns
 *  it as a String or null if there is no more data.
 *
 *  The client can use Integer.parseInt(in.readString()) to read
 *  in an int.
 *
 *  Typically this class is used with another client, but main()
 *  provides a testing routine.
 *
 *  % java In
 *  Test if In.java works.
 *  Second line.
 *
 *  Test
 *  if
 *  In.java
 *  works.
 *  Second
 *  line.
 *
 *  Test if In.java works.
 *  Second line.
 *
 *************************************************************************/

import java.net.URLConnection;
import java.net.URL;
import java.net.Socket;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;


class In {
   private BufferedReader br;

   // system independent
   private final static String NEWLINE = System.getProperty("line.separator");

   // for stdin
   public In() {
      InputStreamReader isr = new InputStreamReader(System.in);
      br = new BufferedReader(isr);
   }

   // for stdin
   public In(Socket socket) {
      try {
         InputStream is        = socket.getInputStream();
         InputStreamReader isr = new InputStreamReader(is);
         br                    = new BufferedReader(isr);
      } catch (IOException ioe) { ioe.printStackTrace(); }
   }

   // for URLs
   public In(URL url) {
      try {
         URLConnection site    = url.openConnection();
         InputStream is        = site.getInputStream();
         InputStreamReader isr = new InputStreamReader(is);
         br                    = new BufferedReader(isr);
      } catch (IOException ioe) { ioe.printStackTrace(); }
   }

   // for files and web pages
   public In(String s) {
```

```
        // try to read file from local file system
        try {
           URL url = getClass().getResource(s);       // for files, even if included in
a jar
           if (url == null) url = new URL(s);         // no file found, try a URL

           URLConnection site    = url.openConnection();
           InputStream is        = site.getInputStream();
           InputStreamReader isr = new InputStreamReader(is);
           br = new BufferedReader(isr);
        } catch(IOException ioe) { ioe.printStackTrace(); }
     }


     // note read() returns -1 if EOF
     private int readChar() {
        int c = -1;
        try { c = br.read(); }
        catch(IOException ioe) { ioe.printStackTrace(); }
        return c;
     }

     // read a token - delete preceding whitespace and one trailing whitespace
character
     public String readString() {
         int c;
         while ((c = readChar()) != -1)
            if (!Character.isWhitespace((char) c)) break;

         if (c == -1) return null;

         String s = "" + (char) c;
         while ((c = readChar()) != -1)
            if (Character.isWhitespace((char) c)) break;
            else s += (char) c;

         return s;
     }

     // return rest of line as string and return it, not including newline
     public String readLine() {
         String s = null;
         try { s = br.readLine(); }
         catch(IOException ioe) { ioe.printStackTrace(); }
         return s;
     }


     // return rest of input as string, use StringBuffer to avoid quadratic run time
     // don't include NEWLINE at very end
     public String readAll() {
         StringBuffer sb = new StringBuffer();
         String s = readLine();
         if (s == null) return null;
         sb.append(s);
         while ((s = readLine()) != null) {
            sb.append(NEWLINE).append(s);
         }
         return sb.toString();
     }


     public void close() {
         try { br.close(); }
         catch (IOException e) { e.printStackTrace(); }
     }


     // This method is just here to test the class
     public static void main (String args[]) {
        In in;
        String s;

        // read from a URL
        in = new In("http://www.cs.princeton.edu/IntroCS/24inout/InTest.txt");
        System.out.println(in.readAll());
        System.out.println();
```

```
      // read one line at a time from URL
      in = new In("http://www.cs.princeton.edu/IntroCS/24inout/InTest.txt");
      while ((s = in.readLine()) != null)
         System.out.println(s);
      System.out.println();

      // read one string at a time from URL
      in = new In("http://www.cs.princeton.edu/IntroCS/24inout/InTest.txt");
      while ((s = in.readString()) != null)
         System.out.println(s);
      System.out.println();


      // read one line at a time from file
      in = new In("InTest.txt");
      while ((s = in.readLine()) != null)
         System.out.println(s);
      System.out.println();
   }

}
```

ListElement

```
/*****
Linked list of links
****/

public class ListElement {
   int ID;
   int StartID;
   int EndID;
   int vfree;
   double dist;
   int lanes;
   String extra;
   ListElement next;

   public ListElement(int ID, int StartID, int EndID, ListElement next, int lanes) {
      this.ID = ID;
      this.StartID = StartID;
      this.EndID = EndID;
      vfree = -1;
      this.extra = null;
      this.next = next;
      dist = -1;
      this.lanes = lanes;
   }

   public ListElement(int ID, int StartID, int EndID, int vfree, ListElement next) {
      this.ID = ID;
      this.StartID = StartID;
      this.EndID = EndID;
      this.vfree = vfree;
      this.extra = null;
      this.next = next;
      dist = -1;
      lanes = -1;
   }

   public ListElement(int ID, int StartID, int EndID, int vfree, String extra,
ListElement next) {
      this.ID = ID;
      this.StartID = StartID;
      this.EndID = EndID;
      this.vfree = vfree;
      this.extra = extra;
      this.next = next;
      dist = -1;
      lanes = -1;
   }

   public ListElement(int ID, int StartID, int EndID, String extra, ListElement next)
{
      this.ID = ID;
```

```
      this.StartID = StartID;
      this.EndID = EndID;
      vfree = -1;
      this.extra = extra;
      this.next = next;
      dist = -1;
      lanes = -1;
   }

   public ListElement(int ID, int StartID, int EndID, int vfree,
   double dist, int lanes, String extra, ListElement next) {
      this.ID = ID;
      this.StartID = StartID;
      this.EndID = EndID;
      this.vfree = vfree;
      this.extra = extra;
      this.next = next;
      this.dist = dist;
      this.lanes = lanes;
   }

   public void deleteFirst() {
      this.ID = next.ID;
      this.StartID = next.StartID;
      this.EndID = next.EndID;
      this.vfree = next.vfree;
      this.extra = next.extra;
      this.next = next.next;
      this.dist = next.dist;
      this.lanes = next.lanes;
   }

   public void setNext(ListElement newNext) {
      this.next = newNext;
   }

}
```

ListElementNodes

```
/*****
Linked list of Nodes
****/

public class ListElementNodes {
   int ID;
   double X;
   double Y;
   int type;
   ListElementNodes next;

   // type: 0) intersection 1) external 2) bend
   public ListElementNodes(int ID, double X, double Y, int type, ListElementNodes
next) {
      this.ID = ID;
      this.X  = X;
      this.Y  = Y;
      this.type = type;
      this.next = next;
   }

}
```

MyIn

```
/*****
Date: June 2005

Contructed using a file name that it will read. Lets the user read word by word
with the method readString(), verify if it is empty with isEmpty() (these should
be the two most common uses for this program). Code based on StdIn.java programmed
by unidentified professors at Princeton University.

*****/

import java.io.IOException;
import java.io.BufferedReader;
import java.io.FileReader;

public class MyIn {
   private int c = ' ';
   private final int EOF = -1;
   private BufferedReader br;
   private boolean isEndOfLine = false;

    // constructor
    public MyIn(String FileName) throws IOException {
      br = new BufferedReader(new FileReader(FileName));
   }

    // is the current character whitespace?
    private boolean isBlank() {
        return Character.isWhitespace((char) c);
    }

    // is it at end of the file already?
    private boolean isEOF() { return c == EOF; }

    // return EOF if end of file or IO error
    private void readC() {
        try { c = br.read(); }
        catch(IOException e) { c = EOF; }
    }

    // is there more input?
    public boolean isEmpty() {
      while (!isEOF() && isBlank())
         readC();
      return isEOF();
    }

   // was last word read last of a line? False is EOF is next
   public boolean wasLastStringOfLine() {
      return isEndOfLine;
   }

    // read a token - use StringBuffer for efficiency
    public String readString() {
        StringBuffer s = new StringBuffer();

        // eat up whitespace
        while (!isEOF() && isBlank())
            readC();

        // now get the string
        while (!isEOF() && !isBlank()) {
            s.append((char) c);
            readC();
        }

      // check if it was last string of line
      if (c == '\n') isEndOfLine = true;
      else {
         isEndOfLine = false;
         while (!isEOF() && isBlank()) {
            readC();
            if (c == '\n') isEndOfLine = true;
```

```
            }
        }

        if (s.length() == 0) throw new RuntimeException("Tried to read from empty
stdin");
        else return s.toString();
    }

    public int     readInt()     { return Integer.parseInt(readString());   }
    public double  readDouble()  { return Double.parseDouble(readString()); }
    public float   readFloat()   { return Float.parseFloat(readString());   }
    public long    readLong()    { return Long.parseLong(readString());     }

    public boolean readBoolean()    {
        String s = readString().toLowerCase();
        if (s.equals("true")  || s.equals("yes") || s.equals("1")) return true;
        if (s.equals("false") || s.equals("no")  || s.equals("0")) return false;
        throw new RuntimeException(s + " is not a valid boolean value");
    }

   public static void main (String[] args) throws IOException {
      MyIn a = new MyIn(args[0]);

      while(!a.isEmpty())
         System.out.println(a.readString() + " " + a.wasLastStringOfLine());

   }

}
```

NodeVolumes

```
class NodeVolumes
{
   private int nodeID;
   private int[] nodeVolumesByMovement = new int[12];

   public NodeVolumes(int nodeID, int[] nodeVolumesByMovement)
   {
      this.nodeID = nodeID;
      this.nodeVolumesByMovement = nodeVolumesByMovement;
   }

   public int getNodeID()
   {
      return nodeID;
   }

   public int[] getNodeVolumesByMovement()
   {
      return nodeVolumesByMovement;
   }
}
```

Ntw_D

```
/*****
Programmers: Lefong Hua and Pierre-Etienne Genest
writeTraffic, printTrafficPlan: L.H.
readCorTable and main: P.E.G.

CRT, Montreal
Started June 8 2005.

Convert a network from Synchro (PHASING.CSV) to
Dynameq (format .dat, sent to standard output).

Compile with: javac Ntw_D.java
Run with: java Ntw_D ControlFinaName PhasingFileName TimingFileName
******/

import java.util.Vector;
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;

class Ntw_D {

   private static CSVReader PHAread;
   private static CSVReader TIMread;

   private static final String fNameTrafficOutput = "trafficControl.dat";
   private static PrintWriter pwTraffic;

   private static Vector offsets = new Vector();

   public static void main (String args[]) throws IOException
   {
      loadOffsets(args[2]);
      writeTraffic(args[0], args[1], args[2]);
      pwTraffic.flush();
      pwTraffic.close();
   }

   //load the Offset of each intersection (node) and store it in the vector offsets
   private static void loadOffsets(String TimingFileName)
   {
      TIMread = new CSVReader(TimingFileName); //Timing file generated by Synchro

      int row = 3; //first row containing a node in Timing file
      final int col = 12; //col containing offset in Timing file

      while(!TIMread.isEmpty(row,col))
         offsets.addElement(TIMread.readCell(row++,col));
   }

   //Copy the information from the control data file to the output file, with
modification
   //on the traffic plan
   private static void writeTraffic(String ControlDataFileName, String
PhasingFileName, String TimingFileName) throws IOException
   {
      try { pwTraffic = new PrintWriter(new FileWriter(fNameTrafficOutput)); }
      catch (IOException e) { System.out.println(e); }

      MyIn ControlData = new MyIn(ControlDataFileName); //input file
      int offIndex = 0; //first offset at index 0 in Vector offsets
      String nodeID = ""; //Node we are currently working on
      boolean stop = false; //stop meaning it's the end of the input file

      String word = ControlData.readString(); //first String of file

      while(!stop)
      {
         //Copy the information in the NODE section to output file
         if (word.equals("NODE"))
         {
            pwTraffic.println(word);
```

```
            word = ControlData.readString();
            pwTraffic.print(word + " ");

            nodeID = word;
            word = ControlData.readString();
         }

         else
         {
            //Take the information on PLAN from the Phasing file
            if (word.equals("PLAN"))
            {
               pwTraffic.println(word);

               //prints the start and end time of PLAN
               pwTraffic.print(ControlData.readString() + " " +
ControlData.readString() + " ");

               //prints the offset
               ControlData.readString();
               pwTraffic.print(offsets.elementAt(offIndex++) + " ");

               //prints the 4 last elements of the line of PLAN
               for (int i=4; i<=6; i++) pwTraffic.print(ControlData.readString() + "
");
               pwTraffic.println(ControlData.readString());

               word = ControlData.readString();

               for (int stateNumber = 1; !word.equals("NODE") && !word.equals("PLAN")
&& !ControlData.isEmpty(); stateNumber++)
               {
                  pwTraffic.print(word + " ");

                  if (printTrafficPlan(nodeID, stateNumber, PhasingFileName))
                  {
                     //goes to the 4th element of the line
                     for (int i=2; i<=4; i++)
                        word = ControlData.readString();
                  }

                  else
                  {
                     //prints the 3 next elements of the line
                     for (int i=2; i<=4; i++)
                        pwTraffic.print(ControlData.readString() + " ");
                  }


                  word = ControlData.readString();

                  //prints the 2 last elements of the line
                  pwTraffic.print(word + " ");
                  word = ControlData.readString();
                  pwTraffic.println(word);
                  word = ControlData.readString();
               }

               //end of file
               if (ControlData.isEmpty()) break;

               //if there is a second plan, ignore it
               if (word.equals("PLAN")) word = readUntil(ControlData, "NODE");
            }

            else
            {
               while (!word.equals("NODE") && !word.equals("PLAN"))
               {
                  pwTraffic.print(word);

                  //end of file
                  if (ControlData.isEmpty())
                  {
                     stop = true;
                     break;
                  }
```

```
                    //space
                    if (!ControlData.wasLastStringOfLine())
                       pwTraffic.print(" ");

                    //end of line
                    else
                       pwTraffic.println();

                    word = ControlData.readString();
                 }
              }
           }
        }

        pwTraffic.println();
        System.out.println("Converted " + PhasingFileName + " and " + TimingFileName +
" to " + fNameTrafficOutput);
     }

     //Prints one line in the PLAN section, given a node ID and a state number
     private static boolean printTrafficPlan(String nodeID, int stateNumber, String
PhasingFileName)
     {
        PHAread = new CSVReader(PhasingFileName); //Phasing file generated by Synchro

        //first row containing MAX green in Phasing file
        int row = 5;

        //col just before first state in Phasing file
        int col = 2;

        String word = "";

        //looking for the node to work on
        //jump to next row containing MAX green
        while(true)
        {
           word = PHAread.readCell(row,2);

           //No plan defined for this intersection
           if (word == null)
              return false;

           if (word.equals(nodeID))
              break;
           else
              row += 23;
        }

        //looking for the state to work on
        for (int index=1; index <= stateNumber; index++)
        {
           col++;
           while (PHAread.readCell(row,col).equals(""))
              col++;
        }

        double green = Double.parseDouble(PHAread.readCell(row,col));
        double yellow = Double.parseDouble(PHAread.readCell(row+5,col));
        double red = Double.parseDouble(PHAread.readCell(row+6,col));

        //In the Phasing file, the max green time includes yellow and red, so remove
those values
        green = green - yellow - red;

        pwTraffic.print(green + " " + yellow + " " + red + " ");

        return true;
     }

     private static String readUntil(MyIn myin, String word) {
        String s;
        for(s = ""; !myin.isEmpty() && !s.equals(word); s = myin.readString());
        return s;
     }
}
```

Ntw_S

```
/****************
Programmers: Pierre-Etienne Genest and Lefong Hua
CRT, Montreal
Started June 14 2005.

Write Traffic Functions: L. Hua
Main and helper functions: P.E. Genest
(including get info, modify links, write layout, write lanes and write correspondence
table)

Convert a Dynameq network in ascii export format to the LAYOUT.CSV, LANES.CSV,
PHASING.CSV and TIMING.CSV
comma-separated values formatted excel documents for input into Synchro.

Input: 2 files formatted as network and traffic control Dynameq outputs
They are taken as command-line arguments.

Output: 4 files named LAY.CSV, LAN.CSV, PHA.CSV and TIM.CSV formatted as CSV files
ready for
Synchro input. Outputed as files with those names.

Compile with: javac Ntw_S.java
Run with: java Ntw_S networkFileName.dat traficControlFileName
****************/

import java.util.Vector;
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;

import java.util.Arrays;

class Ntw_S {

   private static final String layoutFileName = "LAY.CSV"; //Layout output file
   private static final String laneFileName = "LAN.CSV"; //Lanes output file
   private static final String phasingFileName = "PHA.CSV"; //Phasing output file
   private static final String timingFileName = "TIM.CSV"; //Timing output file
   private static final String correspondenceTableFileName = "corTable.txt";
//Correspondance table output file

   private static final double PI = Math.PI;
   private static final int emptyValue = -1;
   private static String word; //token string used throughout the program
   private static final String colSeparator = ",";
   private static final String rowSeparator = "\n";
   private static final int interLimit = 5;
   private static final double approxAngleParameter = 5; // degrees

   private static Vector unsignalizedNodes = new Vector(); //Vector containing nodes
that has no plan

   // Linked list, store all the data about the nodes coming from the
   // network (baseline) ascii file coming from Dynameq
   private static ListElementNodes nodes;

   //Linked list, same but for the links
   private static ListElement links;

   //Linked list, store the moves as printed by this program in the correspondence
   //table, useful to the traffic control part
   private static ListElement moves;

   private static PrintWriter layoutOut; //print writer to write the layout file
   private static PrintWriter laneOut; //print writer to write the lanes file
   private static PrintWriter timOut; //print writer to write the timing file
   private static PrintWriter phaOut; //print writer to write the timing file
   private static PrintWriter pwCorTable; //print writer for the Correspondence Table
file

   private static MyIn ControlData; //reader for the traffic control file

   public static void main (String[] args) throws IOException {
```

```
   pwCorTable = new PrintWriter(new FileWriter(correspondenceTableFileName));
   getInfo(args[0]);
   modifyLinks();
   writeOutput(args[1]);
   writeTraffic(args[1]);
}

/*******************************************/
/*************** GET INFO  ****************/
/*******************************************/

private static void getInfo(String NetworkDataFileName) throws IOException {

   MyIn NetworkData = new MyIn(NetworkDataFileName);

   // token string used throughout the program
   word = "";

   int ID, startID, endID, type, vfree, lanes;

   double X, Y, dist;

   // initialize the linked lists
   nodes = null;
   links = null;
   moves = null;

   // Get Node data
   while(!word.equals("type")) word = NetworkData.readString();
   while(true) {
      word = NetworkData.readString();
      if (word.equals("CENTROIDS")) break;
      ID = Integer.parseInt(word);
      X  = Double.parseDouble(NetworkData.readString());
      Y  = Double.parseDouble(NetworkData.readString());
      NetworkData.readString();
      type = Integer.parseInt(NetworkData.readString());
      nodes = new ListElementNodes(ID, X, Y, type, nodes);
   }

   // Get Link data (skip CENTROIDS)
   while(!word.equals("class(/lane)")) word = NetworkData.readString();
   while(true) {
      word = NetworkData.readString();
      if (word.equals("CONNECTORS")) break;
      ID = Integer.parseInt(word);
      startID = Integer.parseInt(NetworkData.readString());
      endID = Integer.parseInt(NetworkData.readString());
      NetworkData.readString();
      NetworkData.readString();
      dist = Double.parseDouble(NetworkData.readString());
      vfree = Integer.parseInt(NetworkData.readString());
      NetworkData.readString();
      NetworkData.readString();
      lanes = Integer.parseInt(NetworkData.readString());
      NetworkData.readString();
      NetworkData.readString();
      NetworkData.readString();
      links = new ListElement(ID, startID, endID, vfree,
      dist, lanes, "", links);
   }
}

/*******************************************/
/********** Modify Links and Nodes ********/
/********** and Start Correspondence ******/
/*******************************************/

private static void modifyLinks()
{
   // print the header in the Correspondance Table file
   pwCorTable.println("Correspondance Table for Synchro to Dynameq");
   // Start section "REPLACEMENTS", it might end up empty (not in Calgary)
   pwCorTable.println("REPLACEMENTNODES");

   // Intersections with 8 links in Dynameq which have 5 links in Synchro
   // Those are changed to have only 4 links in Synchro as it should be
```

```
// for Synchro to treat correctly

// iterate through all nodes
for(ListElementNodes node = nodes; node != null; node = node.next) {
   int[] adjacentLinksIn = new int[interLimit];
   int[] adjacentLinksOut = new int[interLimit];
   int linksCountIn = 0;
   int linksCountOut = 0;
   // iterate through the links to find adjacent links to that node
   for(ListElement link = links; link != null; link = link.next) {
      if (link.StartID == node.ID)
         adjacentLinksOut[linksCountOut++] = link.ID;
      if (link.EndID == node.ID)
         adjacentLinksIn[linksCountIn++] = link.ID;
   }

   // check only for nodes which have 4 branches for now
   if (linksCountIn == 4) {

      if (linksCountIn != linksCountOut) {
         System.out.println("Error: node #" + node.ID);
         continue;
      }

      // get angles
      double[] anglesIn = new double[linksCountIn];
      double[] anglesOut = new double[linksCountOut];
      for (int i = 0; i < linksCountIn; i++) {
         anglesIn[i] = getAngle(
         node.X,
         node.Y,
         getNode(getLink(adjacentLinksIn[i]).StartID).X,
         getNode(getLink(adjacentLinksIn[i]).StartID).Y);

         anglesOut[i] = getAngle(
         node.X,
         node.Y,
         getNode(getLink(adjacentLinksOut[i]).EndID).X,
         getNode(getLink(adjacentLinksOut[i]).EndID).Y);
      }

      // sort angles
      double[] anglesInSorted = new double[linksCountIn];
      double[] anglesOutSorted = new double[linksCountOut];
      for (int i = 0; i < linksCountIn; i++) {
         anglesInSorted[i] = anglesIn[i];
         anglesOutSorted[i] = anglesOut[i];
      }

      Arrays.sort(anglesInSorted);
      Arrays.sort(anglesOutSorted);

      // in case in and out angles are not equal

      // iterate through 4 sorted angles
      for (int i = 0; i < linksCountIn; i++) {
         // find which links correspond to angles..Sorted
         int inIndex, outIndex;
         for (inIndex = 0;
         anglesInSorted[i] != anglesIn[inIndex];
         inIndex++);
         for (outIndex = 0;
         anglesOutSorted[i] != anglesOut[outIndex];
         outIndex++);

         ListElement inLink = getLink(adjacentLinksIn[inIndex]);
         ListElement outLink = getLink(adjacentLinksOut[outIndex]);

         // if it happens that two links in the same direction
         // don't connect the same two nodes together
         if (inLink.StartID != outLink.EndID) {
            // change the links so that they all converge to same node

            // closer link to intersection is inLink so change outLink
            if (shortest(inLink, outLink) == inLink) {
               // delete the old link that was going from
               // farthest node to intersection node
```

```
                    deleteElement(outLink.ID, links);
                    // goes from intersection node to closest node
                    // replaces old outLink
                    links = new ListElement(
                    outLink.ID,
                    node.ID,
                    inLink.StartID,
                    links,
                    outLink.lanes);
                    // goes from closest node to farthest node
                    links = new ListElement(
                    newLinkID(),
                    inLink.StartID,
                    outLink.EndID,
                    links,
                    outLink.lanes);

                    // write replacement in Correspondence Table
                    pwCorTable.println(inLink.StartID + " " + outLink.EndID);

                    // set new link back in
                    anglesOut[outIndex] = getAngle(
                    node.X,
                    node.Y,
                    getNode(getLink(adjacentLinksOut[outIndex]).EndID).X,
                    getNode(getLink(adjacentLinksOut[outIndex]).EndID).Y);

                    anglesOutSorted[i] = anglesOut[outIndex];

                    Arrays.sort(anglesOutSorted);

                 }
                 // closer link to intersection is outLink so change inLink
                 else {
                    // delete the old link that was going from
                    // farthest node to intersection node
                    deleteElement(inLink.ID, links);
                    // goes from closest node to intersection node
                    // replaces old inLink
                    links = new ListElement(
                    inLink.ID,
                    outLink.EndID,
                    node.ID,
                    links,
                    inLink.lanes);
                    // goes from farthest node to closest node
                    links = new ListElement(
                    newLinkID(),
                    inLink.StartID,
                    outLink.EndID,
                    links,
                    inLink.lanes);

                    // write replacement in Correspondence Table
                    pwCorTable.println(outLink.EndID + " " + inLink.StartID);

                    // set new link back in
                    anglesIn[inIndex] = getAngle(
                    node.X,
                    node.Y,
                    getNode(getLink(adjacentLinksOut[inIndex]).EndID).X,
                    getNode(getLink(adjacentLinksOut[inIndex]).EndID).Y);

                    anglesInSorted[i] = anglesIn[inIndex];

                    Arrays.sort(anglesInSorted);
                 }
              }
           }
        }
     }
  }

/************* WRITE output files ************/

private static void writeOutput(String ControlDataFileName)  throws IOException  {
   //count the nodes
```

```
        int nodeCount = 0;
        for(ListElementNodes node = nodes; node != null; node = node.next)
           nodeCount++;

        //reader for the traffic control file
        ControlData = new MyIn(ControlDataFileName);

        //put the unsignalized nodes into a Vector
        word = "";
        boolean add = false;
        String ID = "";

        while(true) {
           word = readUntil(ControlData, "NODE", "PLAN");

           //end of file
           if (ControlData.isEmpty()) break;

           if (word.equals("NODE")) {
              if (add)
                 unsignalizedNodes.addElement(ID);
              ID = ControlData.readString();
              add = true;
           }

           //word = "PLAN'
           else
              add = false;
        }

        /***************************************/
        /********** Write LAYOUT ***************/
        /***************************************/

        layoutOut = new PrintWriter(new FileWriter(layoutFileName)); //print writer to
write the layout file

        // print the header in the layout file
        layoutOut.print("Layout Data" + rowSeparator);
        layoutOut.print("INTID" + colSeparator + "INTNAME" + colSeparator +
        "TYPE" + colSeparator + "X" + colSeparator +
        "Y" + colSeparator + "NID" + colSeparator +
        "SID" + colSeparator + "EID" + colSeparator +
        "WID" + rowSeparator);

        // adjacentNodes: store the nodes adjacent to each node
        // the first index uses the order in which nodes occur in
        // the linked list of nodes.
        // the second index uses this relationship for now: 0) N 1) S 2) E 3) W
        // Note that interLimit is set to 5 but that only the first 4 fields are used.
        // Since the implementation of Modify Links, no occurences of nodes with 5
        // branches are encountered in Synchro on the southern Calgary network
        int[][] adjacentNodes = new int[nodeCount][interLimit];
        int adjCount, nodeIndex = 0;
        double[] angles = new double[interLimit];

        String word = "";
        // iterate through all nodes and print out
        for(ListElementNodes node = nodes; node != null; node = node.next, nodeIndex++)
{
           int[] adjacentNodesTemp = new int[interLimit];
           for (int i = 0; i < interLimit; i++) adjacentNodesTemp[i] = emptyValue;
           adjCount = 0;
           //find adjacent nodes by checking the linked list of links
           for(ListElement link = links; link != null; link = link.next) {
              // in the case the node is a Start ID of the link
              if(link.StartID == node.ID) {
                 if (!contains(adjacentNodesTemp, adjCount, link.EndID)) {
                    adjacentNodesTemp[adjCount] = link.EndID;
                    adjCount++;
                    if(adjCount >= interLimit) {
                       //Add this node into that Vector, and treat it like an
unsignalized intersection
                       word = Integer.toString(node.ID);
                       if (!unsignalizedNodes.contains(word))
                          unsignalizedNodes.addElement(word);
```

```
                    System.out.println("Warning: Number of links at intersection "
                    + node.ID + " excedes the limit of " + interLimit);
                    adjCount = interLimit - 1;
                 }
              }
           }
           // in the case the node is an End ID of the link
           if(link.EndID == node.ID) {
              if (!contains(adjacentNodesTemp, adjCount, link.StartID)) {
                 adjacentNodesTemp[adjCount] = link.StartID;
                 adjCount++;
                 if(adjCount >= interLimit) {
                    //Add this node into that Vector, and treat it like an
unsignalized intersection
                    word = Integer.toString(node.ID);
                    if (!unsignalizedNodes.contains(word))
                       unsignalizedNodes.addElement(word);

                    System.out.println("Warning: Number of links at intersection "
                    + node.ID + " excedes the limit of " + interLimit);
                    adjCount = interLimit - 1;
                 }
              }
           }
        }

        //A node with no adjacent nodes is either a virtual node, or has no purpose
and should not occur
        if (adjCount != 0) {

           layoutOut.print(node.ID + colSeparator); //print node ID

           switch(adjCount) {

              case 1:
                 layoutOut.print("External #" + node.ID + colSeparator);
                 layoutOut.print("1" + colSeparator);
                 break;

              case 2:
                 layoutOut.print("Bend #" + node.ID + colSeparator);
                 layoutOut.print("2" + colSeparator);
                 break;

              //More than 2
              default:
                 layoutOut.print("Int #" + node.ID + colSeparator);
                 word = Integer.toString(node.ID);
                 if (unsignalizedNodes.contains(word))
                    layoutOut.print("3" + colSeparator);
                 else
                    layoutOut.print("0" + colSeparator);
           }

           // print node coordinates
           layoutOut.print(node.X + colSeparator);
           layoutOut.print(node.Y + colSeparator);

           // find angles for each adjacent Node
           for (int i = 0; i < interLimit; i++) angles[i] = -9999;
           for(int i = 0; i < adjCount; i++) {
              angles[i] = getAngle(
              node.X, node.Y,
              getNode(adjacentNodesTemp[i]).X, getNode(adjacentNodesTemp[i]).Y);
           }

           // adjacent nodes placed in geometrically correct spot
           // algorythm here is weak but works for now
           // this, with other spots regarding N-S-E-W directions, is
           // one of the best place to make the whole program more reliable
           int north = emptyValue, south = emptyValue, east = emptyValue, west =
emptyValue;

           for (int i = 0; i < adjCount; i++) {
              // 0) north
              if (angles[i] < 3.*PI/4. && angles[i] >= PI/4.)
                 adjacentNodes[nodeIndex][0] = adjacentNodesTemp[i];
```

```
            // 1) south
            else if (angles[i] < -1.*PI/4. && angles[i] >= -3.*PI/4.)
               adjacentNodes[nodeIndex][1] = adjacentNodesTemp[i];
            // 2) east
            else if (angles[i] < PI/4. && angles[i] >= -1.*PI/4.)
               adjacentNodes[nodeIndex][2] = adjacentNodesTemp[i];
            // 3) west
            else
               adjacentNodes[nodeIndex][3] = adjacentNodesTemp[i];
         }

         // print the adjacent nodes in each direction

         // print the adjacent nodes' ID in each direction which have a non-empty
         // node id value
         if (adjacentNodes[nodeIndex][0] != emptyValue)
            layoutOut.print(adjacentNodes[nodeIndex][0]);
         // print a column separator (",") between each direction
         layoutOut.print(colSeparator);

         if (adjacentNodes[nodeIndex][1] != emptyValue)
            layoutOut.print(adjacentNodes[nodeIndex][1]);
         layoutOut.print(colSeparator);

         if (adjacentNodes[nodeIndex][2] != emptyValue)
            layoutOut.print(adjacentNodes[nodeIndex][2]);
         layoutOut.print(colSeparator);

         if (adjacentNodes[nodeIndex][3] != emptyValue)
            layoutOut.print(adjacentNodes[nodeIndex][3]);

         // to separate with the next node's data, print a row separator (line
feed)
         layoutOut.print(rowSeparator);
      }
   }

   // output to the user that a layout file has been written
   System.out.println("Printed layout data in file " + layoutFileName);
   // close the file which was opened for writing, in order to flush the buffer
   layoutOut.flush();
   layoutOut.close();

   /**************************************************************/
   /************ Write Lanes Info in the LANES file **************/
   /*********** and Write the Correspondance Table file **********/
   /***************************************************************/

   // token string used throughout the program
   word = "";

   //reader for the traffic control file
   ControlData = new MyIn(ControlDataFileName);

   // print writer to write the lanes file
   laneOut = new PrintWriter(new FileWriter(laneFileName));

   // print the header in the lanes file
   laneOut.print("Lane Group Data" + rowSeparator);
   laneOut.print(
   "RECORDNAME" + colSeparator +
   "INTID" + colSeparator +
   "NBL" + colSeparator +
   "NBT" + colSeparator +
   "NBR" + colSeparator +
   "SBL" + colSeparator +
   "SBT" + colSeparator +
   "SBR" + colSeparator +
   "EBL" + colSeparator +
   "EBT" + colSeparator +
   "EBR" + colSeparator +
   "WBL" + colSeparator +
   "WBT" + colSeparator +
   "WBR" + rowSeparator   );

   /** Movements for one Node at a time **/
```

```
        pwCorTable.println("NODEMOVEMENTS");

        // loop for each NODE appearing in the traffic control ascii file
        while(true) {

           // read until the word NODE is read in the file
           if (!word.equals("NODE")) readUntil(ControlData, "NODE");

           // check if we have reached the end of the traffic control file and stop
           if (ControlData.isEmpty()) break;

           // read this node's ID
           int nodeID = ControlData.readInt();

           // true when this node has to be printed out in the Synchro file
           boolean needPrint = false;

           // find nodeIndex of the node read
           nodeIndex = 0;
           for(ListElementNodes node = nodes; node != null; node = node.next) {
              if (node.ID == nodeID) break;
              nodeIndex++;
           }

           // verify existence of NODE with ID nodeID
           if (nodeIndex >= nodeCount) {
              System.out.println("ERROR: Traffic Control Data File Incompatible");
              System.exit(1);
           }

           pwCorTable.println("NODE " + nodeID);

           /**
           Description of the lanes arrays
           **/
           // indices of the arrays are by (column - 3) of file LANES.CSV
           // The indices from zero to 11 are pictured here (they correspond to a
movement):
           /**       N
                 0  1  2
             11     |     6
           W 10   - * -   7 E
             9      |     8
                 5  4  3
                    S            **/
           // lanesByMovement: number of lanes at that intersection for each direction
           // read directly from traffic control
           int[] lanesByMovement = new int[12];
           // laneIDbyMovement: "in lane" (lane ID of incoming traffic) for that
movement
           // read directly from traffic control
           int[] laneIDbyMovement = new int[12];
           // sharedLanesByMovement: store the values to write in the Synchro Lanes
file
           int[] sharedLanesByMovement = new int[12];

           //initialize lanesByMovement to 0, which effectively cancels a movement
           for (int i = 0; i < 12; i++) lanesByMovement[i] = 0;

           /**
           Read movement Info at intersection
           (lane data for each movement in the STATE section(s))
           **/
           // first go to a word 'STATE' in the current node's traffic control data
           word = readUntil(ControlData, "STATE", "NODE");
           // for each STATE
           while(word.equals("STATE")) {
              //STATE number is read and ignored
              ControlData.readString();

              /**
              For each movement read: identify movement and store data
              in Correspondence Table and moves linked list
              **/
              //for each line in the STATE
              while(!ControlData.isEmpty()) {
                 // read the first word, it should be a number
```

```
word = ControlData.readString();
// if it's not a number then break from the loop (each line of STATE)
// (the word is likely STATE, PLAN or NODE)
if (!(word.charAt(0) >= '0' && word.charAt(0) <= '9'))
   break;
// read the data for one movement
// link ID of incoming traffic at that intersection for this movement
int inLinkID = Integer.parseInt(word);
// link ID of outgoing traffic
int outLinkID = ControlData.readInt();
// number of lanes for that movement
int laneNum = ControlData.readInt();
// ID of the left-most lane for the lanes of this movement
int laneIn = ControlData.readInt();
// out lane and control delay are not useful data here
// there are read and ignored
ControlData.readString();
ControlData.readString();

// find movement direction

// find the ListElement's from the linked list of links with the
correct ID
ListElement inLink = getLink(inLinkID);
ListElement outLink = getLink(outLinkID);

// verify if line contains useful data
// take out movements related to connectors (we ignore connectors)
// and take out lines with a link of 0 lane
if (laneNum == 0 || inLink == null || outLink == null)
   continue;

// After verification is made, set needPrint to true because
// data will have to be printed for this node in the
// lanes file.
needPrint = true;

//inNodeID: node from which incoming traffic arrives intersection
//outNodeID: ndoe from which outgoing traffic leaves intersection
int inNodeID = inLink.StartID;
int outNodeID = outLink.EndID;

// inDirection: contains the direction from which
// incoming traffic arrives intersection (N-S-E-W)
// outDirection: contains the direction from which
// outgoing traffic leaves intersection (N-S-E-W)
int inDirection = emptyValue;
int outDirection = emptyValue;
// loop through all (4) directions of the adjacent nodes of
// the intersection to find what is the direction of the
// incoming and outgoing traffic ndoes
for(int i = 0; i < interLimit; i++) {

   if(adjacentNodes[nodeIndex][i] == inNodeID)
      inDirection = i;

   else if (adjacentNodes[nodeIndex][i] == outNodeID)
      outDirection = i;
}

// contains the type of current movement
// determined with inDirection and outDirection
int movementType = emptyValue;
String movementTypeString = String.valueOf(emptyValue);

// According to inDirection and outDirection, determine the
// movement's type (there are 12 types as explained above)

// 0) North left turn (incoming: 1-South; outgoing: 3-West)
if(inDirection == 1 && outDirection == 3) {
   movementType = 0;
   movementTypeString = "NBL";
}
// 1) North Toward
else if(inDirection == 1 && outDirection == 0) {
   movementType = 1;
   movementTypeString = "NBT";
```

```
            }
            // 2) North right turn
            else if(inDirection == 1 && outDirection == 2) {
               movementType = 2;
               movementTypeString = "NBR";
            }
            // 3) South left turn
            else if(inDirection == 0 && outDirection == 2) {
               movementType = 3;
               movementTypeString = "SBL";
            }
            // 4) South toward
            else if(inDirection == 0 && outDirection == 1) {
               movementType = 4;
               movementTypeString = "SBT";
            }
            // 5) South right turn
            else if(inDirection == 0 && outDirection == 3) {
               movementType = 5;
               movementTypeString = "SBR";
            }
            // 6) East left turn
            else if(inDirection == 3 && outDirection == 0) {
               movementType = 6;
               movementTypeString = "EBL";
            }
            // 7) East Toward
            else if(inDirection == 3 && outDirection == 2) {
               movementType = 7;
               movementTypeString = "EBT";
            }
            // 8) East right turn
            else if(inDirection == 3 && outDirection == 1) {
               movementType = 8;
               movementTypeString = "EBR";
            }
            // 9) West left turn
            else if(inDirection == 2 && outDirection == 1) {
               movementType = 9;
               movementTypeString = "WBL";
            }
            // 10) West Toward
            else if(inDirection == 2 && outDirection == 3) {
               movementType = 10;
               movementTypeString = "WBT";
            }
            // 11) West right turn
            else if(inDirection == 2 && outDirection == 0) {
               movementType = 11;
               movementTypeString = "WBR";
            }

            else {
               String temp = Integer.toString(nodeID);
               if (!unsignalizedNodes.contains(temp))
                  unsignalizedNodes.addElement(temp);
            }

            // write the data for the movement into both the correspondence table
            // and the movement linked list (same data)
            pwCorTable.println(movementTypeString + " " + inLinkID + " " +
            outLinkID + " " + inNodeID + " " + outNodeID);
            moves = new ListElement(nodeID, inLinkID, outLinkID,
            movementTypeString, moves);

            // write the number of lanes for a given movement at the intersection
            if (movementType != emptyValue) {
               lanesByMovement[movementType] = laneNum;
               laneIDbyMovement[movementType] = laneIn;
            }
         }
      }

      // in the case there is no state, the information about the lanes
      // would be unreliable so go on to the next node
      if (!needPrint) continue;
```

```
        /**
        adjust number of lanes for Synchro and gather shared lanes data
        **/
        // initialize shared lanes matrix to 0 (means no sharing in Synchro)
        for(int i = 0; i < 12; i++) sharedLanesByMovement[i] = 0;
        //iterate through each straight movement (see picture of movement types
indexing above)
        for(int i = 1; i < 12; i += 3) {
           int maxLaneTurnRight = laneIDbyMovement[i+1];
           int minLaneForward = laneIDbyMovement[i] - lanesByMovement[i] + 1;
           int maxLaneForward = minLaneForward + lanesByMovement[i] - 1;
           int minLaneTurnLeft = laneIDbyMovement[i-1] - lanesByMovement[i-1] + 1;

           //T with shared lanes
           if (minLaneTurnLeft == maxLaneTurnRight &&
           lanesByMovement[i] == 0) {
              sharedLanesByMovement[i-1] = 2; // special case
              lanesByMovement[i+1] -= 1;
           }
           // double share left and right
           else if (maxLaneTurnRight == minLaneForward &&
              maxLaneForward == minLaneTurnLeft &&
           lanesByMovement[i+1] > 0 &&
           lanesByMovement[i-1] > 0) {
              sharedLanesByMovement[i] = 3;
              lanesByMovement[i+1] -= 1;
              lanesByMovement[i-1] -= 1;
           }
           // shared to the left
           else if (maxLaneForward == minLaneTurnLeft &&
           lanesByMovement[i-1] > 0) {
              sharedLanesByMovement[i] = 1;
              lanesByMovement[i-1] -= 1;
           }
           // shared to the right
           else if (maxLaneTurnRight == minLaneForward &&
           lanesByMovement[i+1] > 0) {
              sharedLanesByMovement[i] = 2;
              lanesByMovement[i+1] -= 1;
           }
        }

        /*** Print out "Lanes" and "Shared" data in the lanes file for this
intersection ***/
        // Lanes
        laneOut.print("Lanes" + colSeparator + nodeID);
        for (int i = 0; i < 12; i++)
           laneOut.print(colSeparator + lanesByMovement[i]);
        laneOut.print(rowSeparator);
        // Shared
        laneOut.print("Shared" + colSeparator + nodeID);
        for (int i = 0; i < 12; i++)
           laneOut.print(colSeparator + sharedLanesByMovement[i]);
        laneOut.print(rowSeparator);
     }

     //System.out.println(unsignalizedNodes.toString());

     System.out.println("Printed \"Lanes\" and \"Shared\" parts of lane data in file
" + laneFileName);

     /** print second part of the correspondence table **/
     pwCorTable.println("LINKS");
     // simply prints all the links data necessary for
     for(ListElement link = links; link != null; link = link.next)
        pwCorTable.println(link.ID + " " +
     link.StartID + " " + link.EndID + " " + link.lanes);

     System.out.println("Printed correspondence table in file " +
     correspondenceTableFileName);
     pwCorTable.flush();
     pwCorTable.close();
  }

  /*******************************************/
  /*********** WRITE TRAFFIC ***************/
  /*******************************************/
```

```
   private static void writeTraffic(String ControlDataFileName) throws IOException  {

      ControlData = new MyIn(ControlDataFileName);

      // printWriter for the Timing file
      phaOut = new PrintWriter(new FileWriter(phasingFileName));

      // printWriter for the Timing file
      timOut = new PrintWriter(new FileWriter(timingFileName));

      // print the header in the Phasing file
      phaOut.print("Phasing Data" + rowSeparator);
      phaOut.print(
      "RECORDNAME" + colSeparator +
      "INTID" + colSeparator +
      "D1" + colSeparator +
      "D2" + colSeparator +
      "D3" + colSeparator +
      "D4" + colSeparator +
      "D5" + colSeparator +
      "D6" + colSeparator +
      "D7" + colSeparator +
      "D8" + rowSeparator   );

      // print the header in the Timing file
      timOut.print("Timing Plans" + rowSeparator);
      timOut.print(
      "PLANID" + colSeparator +
      "INTID" + colSeparator +
      "S1" + colSeparator +
      "S2" + colSeparator +
      "S3" + colSeparator +
      "S4" + colSeparator +
      "S5" + colSeparator +
      "S6" + colSeparator +
      "S7" + colSeparator +
      "S8" + colSeparator +
      "CL" + colSeparator +
      "OFF" + colSeparator +
      "LD" + colSeparator +
      "REF" + colSeparator +
      "CLR" + rowSeparator   );


      Vector states = new Vector(); //Vector containing all the other Vectors
(states) with their of movements
      Vector plan = new Vector(); //Vector containing only the states included in the
plan

      word = "";
      String direction = "";

      int ID = -1; //Node ID
      int state, startID, endID, lanes;

      //static Node in Dynameq
      int staticNode = -1;

      //Goes through the traffic control file of Dynameq
      while(true) {

         if (!word.equals("NODE") && !word.equals("STATE")) word =
readUntil(ControlData, "STATE", "NODE");

         //end of file
         if (ControlData.isEmpty()) break;

         if (word.equals("NODE")) {

            ID = ControlData.readInt();

            //if this node is an external node, skip it, and jump to the next node
            //if this node is a problem node (in Vector unsignalizedNodes), skip it,
and jump to the next node
            //Nodes 4968 and 6059 (In Calgary project) are problems right now, so
skip them
```

```
            if (nodeIsExternal(ID) ||
unsignalizedNodes.contains(Integer.toString(ID)) || ID == 4968 || ID == 6059)  {

               readUntil(ControlData, "NODE");
               word = "NODE";
            }

            //this node is not external, so read the next element in file
            else
            {
               word = ControlData.readString();
               staticNode = Integer.parseInt(word);
            }
         }

         //word.equals("STATE")
         else {

            //goes through the states of a given node
            while (true) {

               word = ControlData.readString();

               state = Integer.parseInt(word);

               //state 0 (and/or staticNode) is all permitted movements, so not
included in a plan for sure
               //Jump to the next state or node
               if ((state == 0) || (state == staticNode))
                  break;

               word = ControlData.readString();

               while(!word.equals("STATE") && !word.equals("PLAN") &&
!word.equals("NODE")) {

                  startID = Integer.parseInt(word);
                  endID = ControlData.readInt();

                  //if the phase does not exist yet, create it and add it to Vector
phases
                  if (state-10 > states.size())
                     states.addElement(new Vector());

                  word = ControlData.readString();

                  //if lanes value is not 0, keep this movement
                  //else, skip it
                  if (!word.equals("0")) {

                     //add a movement to Vector pl
                     direction = findDirection(ID, startID, endID);

                     if (!direction.equals("error")) {

                        //System.out.println(ID);
                        //System.out.println(direction);
                        //System.out.println(state);
                        Vector p = (Vector)states.elementAt(state-11); //gets the n-
1th Vector from phases
                        p.addElement(direction);
                        states.setElementAt(p, state-11); //replaces p in phases with
a direction added
                     }
                  }

                  //skips 3 elements: in lane, out lane, control delay
                  ControlData.readString();
                  ControlData.readString();
                  ControlData.readString();
                  word = ControlData.readString();
               }

               //ready to put the states into the plan
               if (word.equals("PLAN")) {

                  timOut.print("DEFAULT" + colSeparator + ID + colSeparator);
```

```
                    int offset = -1; //offset value
                    double green = 0.0;
                    double yellow = 0.0;
                    double red = 0.0;
                    double maxGreen = 0.0;
                    double maxSplit = 0.0;
                    double cycle = 0.0; //cycle length value

                    int phaseNum = 0;

                    //maximum of 8 phases
                    double[] vmaxGreen = newdTable(8, emptyValue);
                    double[] vyellow = newdTable(8, emptyValue);
                    double[] vred = newdTable(8, emptyValue);

                    //first line of Plan
                    ControlData.readString();
                    ControlData.readString();
                    offset = ControlData.readInt(); //offset value
                    ControlData.readString();
                    ControlData.readString();
                    ControlData.readString();
                    ControlData.readString();
                    word = ControlData.readString();

                    //add each state in plan
                    while(!word.equals("NODE") && !word.equals("PLAN") &&
!ControlData.isEmpty()) {

                       int s = Integer.parseInt(word) - 11;
                       plan.addElement((Vector)states.elementAt(s));

                       green = ControlData.readDouble();
                       yellow = ControlData.readDouble();
                       red = ControlData.readDouble();
                       maxGreen = green + yellow + red; //Add yellow and red time one
more time in Synchro
                       maxSplit = maxGreen + yellow + red; //Maximum split

                       vmaxGreen[phaseNum] = maxGreen;
                       vyellow[phaseNum] = yellow;
                       vred[phaseNum] = red;
                       phaseNum++;

                       timOut.print(maxSplit + colSeparator);
                       cycle += maxSplit;

                       //skips until next state or Node
                       ControlData.readString();
                       ControlData.readString();
                       word = ControlData.readString();
                    }


                    phaOut.print("MinGreen" + colSeparator + ID);
                    for (int i=0; i<phaseNum; i++)
                       phaOut.print(colSeparator + (yellow + red + 1.0));

                    phaOut.print(rowSeparator);

                    phaOut.print("MinSplit" + colSeparator + ID);
                    for (int i=0; i<phaseNum; i++)
                       phaOut.print(colSeparator + (2*(yellow + red + 1.0)));

                    phaOut.print(rowSeparator);

                    phaOut.print("MaxGreen" + colSeparator + ID);
                    for (int i=0; vmaxGreen[i] != (double)emptyValue; i++)
                       phaOut.print(colSeparator + vmaxGreen[i]);

                    phaOut.print(rowSeparator);

                    phaOut.print("Yellow" + colSeparator + ID);
                    for (int i=0; vyellow[i] != (double)emptyValue; i++)
                       phaOut.print(colSeparator + vyellow[i]);
```

```
                    phaOut.print(rowSeparator);

                    phaOut.print("AllRed" + colSeparator + ID);
                    for (int i=0; vred[i] != (double)emptyValue; i++)
                       phaOut.print(colSeparator + vred[i]);

                    phaOut.print(rowSeparator);

                    phaOut.print("Walk" + colSeparator + ID + rowSeparator);
                    phaOut.print("DontWalk" + colSeparator + ID + rowSeparator);

                    for (int i=0; i<(8-phaseNum); i++)
                       timOut.print(colSeparator);

                    timOut.print(cycle + colSeparator + offset + rowSeparator);

                    //in the case there is a second plan, ignore it and consider only
the first one
                    if (word.equals("PLAN")) {

                       readUntil(ControlData, "NODE");
                       word = "NODE";
                    }

                    makeOutput(ID, plan);
                    states.removeAllElements();
                    plan.removeAllElements();

                    //finished a plan for a given node, jump to the next one
                    break;
                 }

                 //no plan defined for the states, so jump to the next Node
                 else if (word.equals("NODE")) {

                    states.removeAllElements();
                    plan.removeAllElements();
                    break;
                 }
              }
           }
        }

        System.out.println("Printed traffic part of lane data in file " +
laneFileName);
        System.out.println("Printed phasing information in file " + phasingFileName);
        System.out.println("Printed timing information in file " + timingFileName);
        laneOut.flush();
        laneOut.close();
        phaOut.flush();
        phaOut.close();
        timOut.flush();
        timOut.close();

     }

     //search for the direction of a movement in moves
     private static String findDirection(int ID, int startID, int endID) {

        ListElement move;

        for (move = moves; !(move == null) && (move.ID != ID || move.StartID != startID
|| move.EndID != endID); move = move.next);

        //move does not exists (wich happens rarely, when there is a bug)
        if (move == null)
           return "error";

        return move.extra;
     }

     //prepares the output to write in Lanes file
     private static void makeOutput(int ID, Vector plan) {

        Vector outputPhases = new Vector(); //contains Phase1, Phase2, etc...
        Vector outputPermPhases = new Vector(); //contains PermPhase1, PermPhase2,
etc...
```

```
      Vector p;

      //i+1 being the phase number worked on
      for (int i = 0; i < plan.size(); i++) {

         p = (Vector)plan.elementAt(i);

         if (conflict(p)) {

            outputPermPhases = fillOutputs(ID, 2, i, p, outputPermPhases);
            p = keepXBT(p);
         }

         outputPhases = fillOutputs(ID, 1, i, p, outputPhases);
      }

      printOutput(1, ID, outputPhases);
      printOutput(2, ID, outputPermPhases);
   }

   //test if there is a conflict between a left turn and a through movement
   private static boolean conflict(Vector p) {

      if (p.contains("NBL")) {

         if(p.contains("SBT") || p.contains("EBT") || p.contains("WBT"))
            return true;
      }

      if (p.contains("SBL")) {

         if(p.contains("NBT") || p.contains("EBT") || p.contains("WBT"))
            return true;
      }

      if (p.contains("EBL")) {

         if(p.contains("NBT") || p.contains("SBT") || p.contains("WBT"))
            return true;
      }

      if (p.contains("WBL")) {

         if(p.contains("NBT") || p.contains("SBT") || p.contains("EBT"))
            return true;
      }

      return false;
   }

   //removes all the through movements from Vector p
   private static Vector keepXBT(Vector p) {

      p.remove("NBL");
      p.remove("NBR");
      p.remove("SBL");
      p.remove("SBR");
      p.remove("EBL");
      p.remove("EBR");
      p.remove("WBL");
      p.remove("WBR");

      return p;
   }

   private static Vector fillOutputs(int ID, int a, int i, Vector p, Vector output) {

      //line "Phase1" or "PermPhase1" needs to be created
      if (output.isEmpty()) output.addElement(newiTable(12, emptyValue));

      String direction;
      int[] t;

      for (int j = 0; j < p.size(); j++) {

         int index = -1;
```

```
            direction = (String)p.elementAt(j);

            //if a=2, that means there is a conflict between movements in current phase
            //We don't want to add through movements and outputPermPhase, and we will
            ///add it in outputPhase
            if (direction.equals("NBL")) index = 0;
            else if (direction.equals("NBT") && a != 2) index = 1;
            else if (direction.equals("NBR")) index = 2;
            else if (direction.equals("SBL")) index = 3;
            else if (direction.equals("SBT") && a != 2) index = 4;
            else if (direction.equals("SBR")) index = 5;
            else if (direction.equals("EBL")) index = 6;
            else if (direction.equals("EBT") && a != 2) index = 7;
            else if (direction.equals("EBR")) index = 8;
            else if (direction.equals("WBL")) index = 9;
            else if (direction.equals("WBT") && a != 2) index = 10;
            else if (direction.equals("WBR")) index = 11;

            if (index != -1) {

               for (int k=0; k < output.size(); k++) {

                  t = (int[])output.elementAt(k);

                  //phase i+1 written at t[index];
                  if (t[index] == emptyValue){

                     t[index] = i+1;
                     break;
                  }

                  else {

                     //another "line" in Lanes file needs to be created
                     if (k == output.size()-1) {

                        int[] x = newiTable(12, emptyValue);
                        x[index] = i+1;
                        output.addElement(x);
                        break;
                     }
                  }
               }
            }
         }

         return output;
      }


      //with a Vector of tables, print outputs to Lanes file (CSV format)
      private static void printOutput(int a, int ID, Vector output) {

         String phase;

         //1 is Phase, 2 is PermPhase
         if (a==2) phase = "PermPhase";
         else phase = "Phase";

         int[] t;

         for (int i=0; i < output.size(); i++) {

            laneOut.print(phase + (i+1) + colSeparator + ID);
            t = (int[])output.elementAt(i);

            for (int j=0; j<t.length; j++) {

               laneOut.print(colSeparator);

               if (t[j] != emptyValue)
                  laneOut.print(t[j]);

               //next line
               if (j == t.length-1)
                  laneOut.print(rowSeparator);
            }
```

```
      }
   }

   /****************************************/
   /********** HELPER FUNCTIONS ************/
   /****************************************/

   //creates a new table of int with a fixed size, default value (emptyValue) for all
cells
   private static int[] newiTable(int size, int emptyValue) {

      int[] t = new int[size];

      for (int i=0; i<size; i++)
         t[i] = emptyValue;

      return t;
   }

   //creates a new table of double with a fixed size, default value (emptyValue) for
all cells
   private static double[] newdTable(int size, double emptyValue) {

      double[] t = new double[size];

      for (int i=0; i<size; i++)
         t[i] = emptyValue;

      return t;
   }

   // delete an Element with ID ID from the list LE
   private static void deleteElement(int ID, ListElement LE) {
      if (LE.ID == ID)
         LE.deleteFirst();
      else {
         ListElement previous = null, path;
         for(path = LE, previous = null; path != null; path = path.next) {
            if (ID == path.ID) {
               previous.setNext(path.next);
            }
            previous = path;
         }
      }
   }

   // return true if the node with id nodeID is an external node
   private static boolean nodeIsExternal(int nodeID) {
      int count = 0;
      for (ListElement link = links; link != null; link = link.next) {
         if (link.StartID == nodeID)
            count++;
      }
      return (count <= 1);
   }

   // return the ListElement object of the link with ID linkID
   private static ListElement getLink(int linkID) {
      for(ListElement path = links; path != null; path = path.next)
         if (linkID == path.ID) return path;
      return null;
   }

   //return the ListElementNodes object of the node with ID ID
   private static ListElementNodes getNode(int ID) {
      for(ListElementNodes path = nodes; path != null; path = path.next)
         if (ID == path.ID) return path;
      return null;
   }

   // read in myin until word 'word' is encountered
   private static String readUntil(MyIn myin, String word) {
      String s;
      for(s = "";
      !myin.isEmpty() && !s.equals(word);
      s = myin.readString());
      return s;
```

```
   }

   // read in myin until word 'word1' or 'word2' is encountered
   private static String readUntil(MyIn myin, String word1, String word2) {
      String s;
      for(s = "";
      !myin.isEmpty() && !s.equals(word1) && !s.equals(word2);
      s = myin.readString());
      return s;
   }

   // return true if value is found as an element of the array A with size SizeA
   private static boolean contains(int[] A, int SizeA, int value) {
      for(int i = 0; i < SizeA; i++)
         if(A[i] == value) return true;
      return false;
   }

   // swap the two elements of the array adjNodes which are at index1 and index2
   private static void swap(int[] adjNodes, int index1, int index2) {
      int a = adjNodes[index1];
      adjNodes[index1] = adjNodes[index2];
      adjNodes[index2] = a;
   }

   // compute polar angle of a point at (pX, pY) with regard to a point at (originX,
originY)
   private static double getAngle(double originX, double originY, double pX, double
pY) {
      double x = pX - originX;
      double y = pY - originY;
      return Math.atan2(y, x);
   }

   // return the ListElement link1 or link2 which has the shortest distance
   // between its two connected nodes
   private static ListElement shortest(ListElement link1, ListElement link2) {
      double dist1 = distance(
      getNode(link1.StartID).X, getNode(link1.StartID).Y,
      getNode(link1.EndID).X, getNode(link1.EndID).Y);
      double dist2 = distance(
      getNode(link2.StartID).X, getNode(link2.StartID).Y,
      getNode(link2.EndID).X, getNode(link2.EndID).Y);

      if (dist1 < dist2) return link1;
      else return link2;
   }

   // compute the the euclidean distance between points (X1, Y1) and (Y1, Y2)
   private static double distance(double X1, double Y1, double X2, double Y2) {
      return Math.sqrt((Y2-Y1)*(Y2-Y1) + (X2-X1)*(X2-X1));
   }

   // return an unused link ID
   private static int newLinkID() {
      int newID = 0;
      for(ListElement path = links; path != null; path = path.next)
         if (path.ID > newID) newID = path.ID;
      return newID + 1;
   }

   // return an unused node ID
   private static int newNodeID() {
      int newID = 0;
      for(ListElementNodes path = nodes; path != null; path = path.next)
         if (path.ID > newID) newID = path.ID;
      return newID + 1;
   }
}
```

Vol_S

```
/****************

Programmer: Pierre-Etienne Genest & Lefong Hua
CRT, Montreal
Started July 15 2005.

Convert a Dynameq volume data file in ascii export format to the VOLUME.CSV
comma-separated values formatted excel documents for input into Synchro.

Uses Correspondence table produced by Ntw_S
Requires execution of Ntw_S.java on the network from which the movementscounts
have been taken.

Compile with: javac Vol_S.java
Run with: java Vol_S volumeFileName.out

****************/

import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;

//import java.util.Iterator;
import java.util.HashMap;
import java.util.Vector;

class Vol_S {

   static final private String volFileName = "VOL.CSV";
   static final private String corTableInFileName = "corTable.txt";

   static final private String colSeparator = ",";
   static final private String rowSeparator = "\n";

   static final private int EMPTYVALUE = 0;

   public static void main (String[] args) throws IOException {

      MyIn volumesIn = new MyIn(args[0]); //Dynameq volume file reader
      PrintWriter volOut = new PrintWriter(new FileWriter(volFileName)); //Synchro
volume file writer
      MyIn corTableIn = new MyIn(corTableInFileName); //correspondence table file
reader
      ListElement Movements = null; //linked list of Movements
      String word = ""; //token string

      /**** store the replacements and movement sections of the correspondence table
****/

      // replacements
      HashMap replace = new HashMap();

      // read replacements and store them in a hash map
      readUntil(corTableIn, "REPLACEMENTNODES");
      word = corTableIn.readString();
      while(!word.equals("NODEMOVEMENTS")) {
         Integer node1 = new Integer(Integer.parseInt(word));
         Integer node2 = new Integer(corTableIn.readInt());

         replace.put(node2, node1);

         word = corTableIn.readString();
      }

      // read movements and store them in a linked list
      while(true) {

         // read until NODE, break if word is LINKS
         if (word == null)
            word = readUntil(corTableIn, "NODE");
         else if (word.equals("LINKS"))
            break;
```

```
         else if (!word.equals("NODE"))
            word = readUntil(corTableIn, "NODE");

         // read the node's id
         int nodeID = corTableIn.readInt();

         // read until first word read is NODE again or word is LINKS
         while(true) {
            word = corTableIn.readString();
            if (word.equals("NODE") || word.equals("LINKS"))
               break;

            // temporarily store data of the line
            String type = word;
            corTableIn.readInt();
            corTableIn.readInt();
            int startID = corTableIn.readInt();
            int endID = corTableIn.readInt();

            // store that movement's (line's) data in the movement linked list
            Movements = new ListElement(nodeID, startID, endID, type, Movements);
         }
      }

      /*** store the volumes from the movement counts ****/

      Vector movementCountsByNode = new Vector(); //movementCountsByNode: maps node
ID to array of movement counts

      // headers for the output file
      volOut.println("Turning Movement Count");
      volOut.println("60 Minute Counts");
      volOut.println("INTID,NBL,NBT,NBR,SBL,SBT,SBR,EBL,EBT,EBR,WBL,WBT,WBR");

      int nodeID = EMPTYVALUE;

      NodeVolumes n;
      int[] nodeVolumesByMovement = null;

      //for each line in the file
      while(!volumesIn.isEmpty()) {

         nodeID = volumesIn.readInt(); //read nodeID for that line

         boolean present = false;
         int index = 0;

         for(int i=0; i<movementCountsByNode.size(); i++)
         {
            n = (NodeVolumes)movementCountsByNode.elementAt(i);
            if (n.getNodeID() == nodeID)
            {
               present = true;
               index = i;
               nodeVolumesByMovement = n.getNodeVolumesByMovement();
               break;
            }
         }

         if (!present)
         {
            nodeVolumesByMovement = new int[12]; //create an array of int (object)
and initialize it
            for (int i = 0; i < 12; i++)
               nodeVolumesByMovement[i] = EMPTYVALUE;
         }

         //ID of the incoming/outgoing traffic nodes
         int nodeIDin = volumesIn.readInt();
         int nodeIDout = volumesIn.readInt();

         //object representation of nodeIDin and out
         Integer oNodeIDin = new Integer(nodeIDin);
         Integer oNodeIDout = new Integer(nodeIDout);

         //if they are in the replacement hash map, replace the ID
         if (replace.containsKey(oNodeIDin))
```

```
            nodeIDin = ((Integer)replace.get(oNodeIDin)).intValue();
         if (replace.containsKey(oNodeIDout))
            nodeIDout = ((Integer)replace.get(oNodeIDout)).intValue();

         //count up the rest of the line, assuming the counts were for one hour
         int volCount = 0;
         while (!volumesIn.wasLastStringOfLine())
            volCount += volumesIn.readInt();

         //find movement type
         //String representation of the type of movement
         String sType = null;

         //go through the linked list of movements to find the right one and its type
         for (ListElement move = Movements; move != null; move = move.next)
         {
            if(move.ID == nodeID && move.StartID == nodeIDin && move.EndID ==
nodeIDout)
            {
               sType = move.extra;
               break;
            }
         }

         //type of movement with the usual values between 0 and 11 (see Ntw_S.java)
         int iType = EMPTYVALUE;

         if (sType != null)
         {
            if (sType.equals("NBL")) iType = 0;
            else if (sType.equals("NBT")) iType = 1;
            else if (sType.equals("NBR")) iType = 2;
            else if (sType.equals("SBL")) iType = 3;
            else if (sType.equals("SBT")) iType = 4;
            else if (sType.equals("SBR")) iType = 5;
            else if (sType.equals("EBL")) iType = 6;
            else if (sType.equals("EBT")) iType = 7;
            else if (sType.equals("EBR")) iType = 8;
            else if (sType.equals("WBL")) iType = 9;
            else if (sType.equals("WBT")) iType = 10;
            else if (sType.equals("WBR")) iType = 11;
            else continue;

            nodeVolumesByMovement[iType] = volCount;
         }

         n = new NodeVolumes(nodeID, nodeVolumesByMovement);

         if (sType != null)
         {
            if(present)
               movementCountsByNode.setElementAt(n,index);
            else
               movementCountsByNode.add(n);
         }
      }

      /*** print out the volumes data ***/

      for(int i=0; i<movementCountsByNode.size(); i++)
      {
         n = (NodeVolumes)movementCountsByNode.elementAt(i);
         nodeID = n.getNodeID(); //get the nodeID
         nodeVolumesByMovement = n.getNodeVolumesByMovement(); //get the movement
counts for this node
         volOut.print(nodeID); //print ID of the node

         //print all the volumes (for each movement)
         for (int j=0; j<12; j++)
            volOut.print(colSeparator + nodeVolumesByMovement[j]);
         volOut.print(rowSeparator);
      }

      // close the file. Writing completed
      volOut.close();
      System.out.println("Printed Volume Data in File " + volFileName);
   }
```

```
   // read in myin until word 'word' is encountered
   private static String readUntil(MyIn myin, String word) {
      String s;
      for(s = ""; !myin.isEmpty() && !s.equals(word); s = myin.readString());
      return s;
   }

}
```

Annexe 2 – Des exemples de fichiers d'entrée et de sortie

Voici des exemples de fichier d'entrée et de sortie dans Synchro et Dynameq. Il est à noter que ceux-ci ne sont pas représentés en entier, car le but n'est que d'illustrer le format de ces derniers.

Réseau de Dynameq

```
*NETWORK ASCII FILE - GENERATED mer. janvier 03 2007
VEH_CLASSES
Default
NODES
*id     x                     y                     z             type
1       251.402423            566.248275            1             1
2       351.237820            566.248275            1             1
3       251.402423            466.050585            1             1
4       351.237820            466.050585            1             1
5       251.449374            365.787768            1             1
6       351.237820            366.242013            1             1
7       251.402423            666.251406            1             1
8       351.237820            666.056847            1             1
9       151.567026            566.248275            1             1
10      151.180067            466.050585            1             1
11      450.879738            566.248275            1             1
12      451.096592            465.976903            1             1
15      51.334848             566.322855            1             1
16      51.334848             466.038775            1             1
17      51.334848             516.342044            1             1
20      23.576265             516.378814            1             99
CENTROIDS
*id     x                     y                     z
2       614.352480            516.098787            1
1       -18.548492            516.415585            1
3       300.960494            200.102962            1
4       299.014176            839.192247            1
LINKS
*id     start   end     reverse type    len     vfree   kjam    qsat            lanes
        rabout  wpen    class(/lane)
1       1       2       -1      1       0.099   60      200     2400            1       0
        0       *
2       3       4       -1      1       0.099   60      200     2400            1       0
        0       *
3       3       1       -1      1       0.1     60      200     2400            1       0
        0       *
4       4       2       -1      1       0.1     60      200     2400            1       0
        0       *
5       6       4       -1      1       0.099   60      200     2400            1       0
        0       *
6       5       3       -1      1       0.1     60      200     2400            1       0
        0       *
7       2       8       -1      1       0.099   60      200     2400            1       0
        0       *
8       1       7       -1      1       0.1     60      200     2400            1       0
        0       *
9       10      3       -1      1       0.1     60      200     2400            1       0
        0       *
10      9       1       -1      1       0.099   60      200     2400            1       0
        0       *
11      2       11      -1      1       0.099   60      200     2400            1       0
        0       *
12      4       12      -1      1       0.099   60      200     2400            1       0
        0       *
13      15      9       -1      1       0.1     60      200     2400            1       0
        0       *
14      16      10      -1      1       0.099   60      200     2400            1       0
        0       *
```

```
15      17      15      -1      1       0.049   60      200     2400            1       0
        0       *
16      17      16      -1      1       0.05    60      200     2400            1       0
        0       *
19      11      21      -1      1       0.099   60      200     2400            1       0
        0       *
20      12      22      -1      1       0.099   60      200     2400            1       0
        0       *
CONNECTORS
*id     start   end     vnode   linkRef len     vfree   kjam    qsat            lanes
        rabout  wpen    class(/lane)
2       -1      17      20      18      0.05    60      200     2400            2       0
        0       *
3       23      -2      24      23      0.05    60      200     2400            2       0
        0       *
5       -3      27      31      29      0.05    60      200     2400            2       0
        0       *
6       34      -4      36      34      0.05    60      200     2400            2       0
        0       *
MOVEMENTS
*at     in      out     vfree           class
1       3       1       60              -
4       2       4       60              -
4       5       4       60              *
3       6       2       60              -
3       6       3       60              *
2       1       7       60              -
2       4       7       60              *
1       3       8       60              *
3       9       2       60              *
3       9       3       60              -
1       10      1       60              *
1       10      8       60              -
2       1       11      60              *
2       4       11      60              -
4       2       12      60              *
4       5       12      60              -
9       13      10      60              *
```

Plan de traffic de Dynameq

```
*TRAFFIC CONTROL ASCII FILE - GENERATED mer. janvier 03 2007
NODE
1 0
STATE
0
3 1 0 0 0 0
3 8 1 1 1 0
10 1 1 1 1 0
10 8 0 0 0 0
STATE
11
3 1 0 0 0 0
3 8 0 0 0 0
10 1 1 1 1 0
10 8 0 0 0 0
STATE
12
3 1 0 0 0 0
3 8 1 1 1 0
10 1 0 0 0 0
10 8 0 0 0 0
PLAN
700 800 0 1 0 1 1
11 40 4 1 3 0
12 20 4 1 3 0
NODE
2 0
STATE
0
1 7 0 0 0 0
4 7 1 1 1 0
1 11 1 1 1 0
4 11 0 0 0 0
STATE
11
1 7 0 0 0 0
4 7 0 0 0 0
1 11 1 1 1 0
4 11 0 0 0 0
STATE
12
1 7 0 0 0 0
4 7 1 1 1 0
1 11 0 0 0 0
4 11 0 0 0 0
PLAN
700 800 0 1 0 1 1
11 40 4 1 3 0
12 20 4 1 3 0
NODE
3 0
STATE
0
6 2 0 0 0 0
6 3 1 1 1 0
9 2 1 1 1 0
9 3 0 0 0 0
STATE
11
6 2 0 0 0 0
6 3 0 0 0 0
9 2 1 1 1 0
9 3 0 0 0 0
STATE
12
6 2 0 0 0 0
6 3 1 1 1 0
9 2 0 0 0 0
9 3 0 0 0 0
PLAN
700 800 0 1 0 1 1
11 40 4 1 3 0
12 20 4 1 3 0
```

Mouvements de virage dans Dynameq

```
4 6 2 6 24 9 24 13 13 13 31 18 11 15 32 19 23 13 21 13 19 19 19 8 21 14 21 13 28 17
25 9 14 0
3 5 1 13 18 18 14 13 18 14 20 24 25 15 10 16 11 11 16 14 20 12 21 12 13 13 13 17 11
14 18 13 20 0
2 4 8 6 24 9 23 13 14 10 33 19 11 15 32 17 24 13 22 13 16 22 19 8 21 13 22 12 27 18
26 8 14 0
1 3 7 12 18 17 16 11 19 14 20 23 26 16 9 17 10 12 15 13 21 11 21 13 13 14 13 15 11 15
19 11 21 0
3 10 4 15 16 24 15 24 10 23 16 16 18 13 6 19 11 15 11 19 17 20 19 24 16 25 16 22 14
21 21 13 8 0
1 9 2 13 11 22 11 21 14 18 7 13 10 17 21 20 14 23 11 22 12 29 11 23 6 28 13 18 8 20
12 23 7 0
2 1 11 13 9 23 12 20 15 18 6 14 10 17 17 22 16 23 11 21 12 30 10 24 6 28 13 16 10 20
12 22 8 0
4 3 12 14 16 24 15 25 9 23 15 16 19 14 6 16 14 13 12 20 17 19 20 20 20 25 15 23 13 22
19 14 9 0
9 15 1 13 17 16 12 20 21 12 9 10 13 16 27 14 18 19 13 18 16 26 19 14 13 23 13 16 14
16 13 20 12 0
10 16 3 18 19 20 21 16 13 23 17 13 18 12 10 16 17 8 14 17 20 21 19 23 19 20 20 19 15
22 19 10 12 0
15 17 9 14 16 16 12 20 25 8 9 10 14 16 28 12 19 19 13 18 17 25 20 14 11 23 14 15 16
14 13 20 14 0
16 17 10 19 20 19 20 16 13 23 17 15 16 13 11 14 18 9 13 16 20 22 19 23 19 19 21 18 18
20 18 11 12 0
17 20 15 14 16 16 12 21 26 6 9 11 13 16 29 11 19 19 14 17 17 26 19 15 11 22 14 15 16
14 15 19 14 0
17 20 16 19 21 19 21 14 13 24 17 14 16 13 13 13 17 9 14 15 20 22 19 23 21 18 22 16 20
18 18 11 13 0
11 2 21 13 9 22 13 19 16 18 6 13 11 17 17 22 16 21 13 20 13 28 12 23 7 27 14 16 10 19
13 22 8 0
12 4 22 13 17 24 15 24 10 22 16 16 19 14 6 15 15 13 12 20 17 17 22 18 22 25 15 23 13
22 19 14 9 0
22 12 23 9 21 23 16 20 14 21 17 15 20 10 10 15 15 13 12 18 19 14 25 14 26 23 17 22 14
21 20 11 12 0
21 11 23 13 9 20 15 15 20 18 6 10 14 17 17 21 17 18 16 16 17 24 16 20 10 24 17 15 11
16 16 18 12 0
23 21 24 11 11 18 17 13 22 17 7 9 15 15 19 19 19 16 18 14 19 22 18 18 12 22 19 13 13
14 18 16 14 0
23 22 24 7 23 21 18 18 16 19 19 13 22 8 12 15 15 12 13 16 21 12 27 12 28 21 19 20 16
21 20 9 14 0
5 25 3 17 14 18 14 16 15 21 14 30 20 16 9 15 12 11 15 21 14 12 19 15 11 17 12 16 9 19
13 21 13 0
6 26 4 13 18 13 20 14 12 20 26 19 8 23 25 22 18 20 15 15 16 25 13 11 20 19 16 20 21
24 16 11 14 0
25 27 5 19 14 17 14 15 17 21 13 30 19 16 10 14 13 10 15 21 14 16 15 15 12 16 13 15 10
18 14 20 13 0
26 27 6 14 18 13 19 15 12 20 25 20 7 23 26 21 18 20 15 16 17 24 12 12 20 20 14 23 19
24 15 15 11 0
27 31 26 14 18 13 20 14 13 21 24 19 8 23 25 22 17 21 15 16 16 24 14 11 19 21 13 23 20
24 15 14 11 0
27 31 25 19 15 16 15 14 18 21 13 30 19 15 10 14 13 11 15 20 14 17 15 14 12 16 13 15
10 19 13 20 14 0
8 2 33 6 22 11 21 15 14 10 33 19 11 15 31 18 22 15 22 13 16 22 18 9 19 15 21 13 26 19
26 8 13 0
7 1 32 12 17 18 16 11 19 14 20 23 25 17 9 17 9 13 14 14 20 12 20 14 13 14 13 15 11 15
18 12 21 0
32 7 34 12 15 20 16 11 16 17 18 25 23 19 9 17 9 13 13 15 19 13 20 14 10 17 12 16 11
15 16 14 21 0
33 8 34 6 18 15 20 16 13 11 29 23 11 15 27 22 18 19 19 16 16 22 16 11 15 19 19 15 25
20 22 12 13 0
34 33 36 6 16 17 19 17 11 13 27 25 10 16 25 24 16 21 17 18 15 23 14 13 13 21 17 17 24
21 20 14 13 0
34 32 36 12 13 22 14 13 14 19 16 27 21 21 7 19 8 14 11 17 17 15 18 16 9 18 10 18 10
16 14 16 19 0
```

Table de correspondance

```
Correspondance Table for Synchro to Dynameq
REPLACEMENTNODES
NODEMOVEMENTS
NODE 1
NBT 3 8 3 7
EBT 10 1 9 2
EBT 10 1 9 2
NBT 3 8 3 7
NODE 2
NBT 4 7 4 8
EBT 1 11 1 11
EBT 1 11 1 11
NBT 4 7 4 8
NODE 3
NBT 6 3 5 1
EBT 9 2 10 4
EBT 9 2 10 4
NBT 6 3 5 1
NODE 4
NBT 5 4 6 2
EBT 2 12 3 12
EBT 2 12 3 12
NBT 5 4 6 2
NODE 5
NBT 24 6 25 3
NODE 6
NBT 25 5 26 4
NODE 7
NBT 8 31 1 32
NODE 8
NBT 7 30 2 33
NODE 9
EBT 13 10 15 1
LINKS
33 33 34 1
32 32 34 1
31 7 32 1
30 8 33 1
28 27 26 1
27 27 25 1
25 26 6 1
24 25 5 1
22 21 23 1
21 22 23 1
20 12 22 1
19 11 21 1
16 17 16 1
15 17 15 1
14 16 10 1
```

Réseau de Synchro

Layout
Data

INTID	INTNAME	TYPE	X	Y	NID	SID	EID	WID
34	Bend #34	2	301.238539	766.352814	0	0	33	32
33	Bend #33	2	351.286722	766.352814	0	8	0	34
32	Bend #32	2	251.468403	766.352814	0	7	34	0
27	Bend #27	2	301.238539	266.244517	0	0	26	25
26	Bend #26	2	351.286722	266.244517	6	0	0	27
25	Bend #25	2	251.468403	266.244517	5	0	27	0
23	Bend #23	2	550.730856	516.098787	21	22	0	0
22	Bend #22	2	550.730856	466.044613	23	0	0	12
21	Bend #21	2	550.730856	566.311359	0	23	0	11
17	Bend #17	2	51.334848	516.342044	15	16	0	0
16	Bend #16	2	51.334848	466.038775	17	0	10	0
15	Bend #15	2	51.334848	566.322855	0	17	9	0
12	Bend #12	2	451.096592	465.976903	0	0	22	4
11	Bend #11	2	450.879738	566.248275	0	0	21	2
10	Bend #10	2	151.180067	466.050585	0	0	3	16
9	Bend #9	2	151.567026	566.248275	0	0	1	15
8	Bend #8	2	351.23782	666.056847	33	2	0	0
7	Bend #7	2	251.402423	666.251406	32	1	0	0
6	Bend #6	2	351.23782	366.242013	4	26	0	0
5	Bend #5	2	251.449374	365.787768	3	25	0	0
4	Int #4	0	351.23782	466.050585	2	6	12	3
3	Int #3	0	251.402423	466.050585	1	5	4	10
2	Int #2	0	351.23782	566.248275	8	4	11	1
1	Int #1	0	251.402423	566.248275	7	3	2	9

Voies dans Synchro

Lane Group Data

RECORDNAME	INTID	NBL	NBT	NBR	SBL	SBT	SBR	EBL	EBT	EBR	WBL	WBT	WBR
Lanes	1	0	1	0	0	0	0	0	1	0	0	0	0
Shared	1	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	2	0	1	0	0	0	0	0	1	0	0	0	0
Shared	2	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	3	0	1	0	0	0	0	0	1	0	0	0	0
Shared	3	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	4	0	1	0	0	0	0	0	1	0	0	0	0
Shared	4	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	5	0	1	0	0	0	0	0	0	0	0	0	0
Shared	5	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	6	0	1	0	0	0	0	0	0	0	0	0	0
Shared	6	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	7	0	1	0	0	0	0	0	0	0	0	0	0
Shared	7	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	8	0	1	0	0	0	0	0	0	0	0	0	0
Shared	8	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	9	0	0	0	0	0	0	0	1	0	0	0	0
Shared	9	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	10	0	0	0	0	0	0	0	1	0	0	0	0
Shared	10	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	11	0	0	0	0	0	0	0	1	0	0	0	0
Shared	11	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	12	0	0	0	0	0	0	0	1	0	0	0	0
Shared	12	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	15	0	0	0	0	0	0	0	0	0	0	0	0
Shared	15	2	0	0	0	0	0	0	0	0	0	0	0
Lanes	16	0	0	0	1	0	0	0	0	0	0	0	0
Shared	16	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	21	0	0	0	0	0	0	0	0	0	0	0	0
Shared	21	0	0	0	0	0	0	2	0	0	0	0	0
Lanes	22	0	0	0	0	0	0	1	0	0	0	0	0
Shared	22	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	25	0	0	0	0	0	0	0	0	0	0	0	0
Shared	25	0	0	0	0	0	0	0	0	0	2	0	0
Lanes	26	0	0	0	0	0	0	1	0	0	0	0	0
Shared	26	0	0	0	0	0	0	0	0	0	0	0	0
Lanes	32	0	0	0	0	0	0	0	0	0	0	0	0
Shared	32	2	0	0	0	0	0	0	0	0	0	0	0
Lanes	33	1	0	0	0	0	0	0	0	0	0	0	0
Shared	33	0	0	0	0	0	0	0	0	0	0	0	0
Phase1	1		2						1				
Phase1	2		2						1				
Phase1	3		2						1				
Phase1	4		2						1				

Temps dans Synchro

Timing Plans

PLANID	INTID	S1	S2	S3	S4	S5	S6	S7	S8	CL	OFF	LD	REF	CLR
DEFAULT	1	50	30							80	0			
DEFAULT	2	50	30							80	0			
DEFAULT	3	50	30							80	0			
DEFAULT	4	50	30							80	0			

Phases dans Synchro

Phasing Data

RECORDNAME	INTID	D1	D2	D3	D4	D5	D6	D7	D8
MinGreen	1	6	6						
MinSplit	1	12	12						
MaxGreen	1	45	25						
Yellow	1	4	4						
AllRed	1	1	1						
Walk	1								
DontWalk	1								
MinGreen	2	6	6						
MinSplit	2	12	12						
MaxGreen	2	45	25						
Yellow	2	4	4						
AllRed	2	1	1						
Walk	2								
DontWalk	2								
MinGreen	3	6	6						
MinSplit	3	12	12						
MaxGreen	3	45	25						
Yellow	3	4	4						
AllRed	3	1	1						
Walk	3								
DontWalk	3								
MinGreen	4	6	6						
MinSplit	4	12	12						
MaxGreen	4	45	25						
Yellow	4	4	4						
AllRed	4	1	1						
Walk	4								
DontWalk	4								

Volumes dans Synchro

Turning Movement Count

60 Minute Counts

INTID	NBL	NBT	NBR	SBL	SBT	SBR	EBL	EBT	EBR	WBL	WBT	WBR
4	0	525	0	0	0	0	0	507	0	0	0	0
3	0	467	0	0	0	0	0	507	0	0	0	0
2	0	524	0	0	0	0	0	478	0	0	0	0
1	0	466	0	0	0	0	0	478	0	0	0	0
9	0	0	0	0	0	0	0	483	0	0	0	0
10	0	0	0	0	0	0	0	511	0	0	0	0
15	0	0	485	0	0	0	0	0	0	0	0	0
16	0	0	0	512	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	478	0	0	0	0
12	0	0	0	0	0	0	0	507	0	0	0	0
22	0	0	0	0	0	0	507	0	0	0	0	0
21	0	0	0	0	0	0	0	0	478	0	0	0
5	0	469	0	0	0	0	0	0	0	0	0	0
6	0	527	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	469
26	0	0	0	0	0	0	528	0	0	0	0	0
8	0	523	0	0	0	0	0	0	0	0	0	0
7	0	466	0	0	0	0	0	0	0	0	0	0
32	0	0	466	0	0	0	0	0	0	0	0	0
33	523	0	0	0	0	0	0	0	0	0	0	0