UNIVERSITÉ DU QUÉBEC À MONTRÉAL

DÉVELOPPEMENT D'UN ALGORITHME DE TYPE VOYAGEUR DE COMMERCE GÉNÉRALISÉ POUR UN PROBLÈME DE TRAJET OPTIMAL DANS UNE VILLE

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

TANIA JOLY

DÉCEMBRE 2011

REMERCIEMENTS

Je remercie vivement le Dr Vladimir Makarenkov, mon directeur de recherche, pour m'avoir soutenue, conseillée et accompagnée tout au long de mon travail de maîtrise. C'est à lui que je dois les défis qui m'ont poussée à surpasser mes limites.

Je tiens aussi à remercier son assistant de recherche Alix Boc, auprès duquel j'ai toujours pu trouver support et conseil.

Mes remerciements s'adressent aussi à mes parents pour leur soutien constant.

Je me dois aussi d'adresser ma plus grande reconnaissance au comité d'accréditation des bourses FARE et Quebecor, qui ont contribué au financement de ce projet de maîtrise.

À tous ceux qui m'ont épaulée durant la réalisation de ce projet, qu'ils trouvent ici mes remerciements les plus sincères.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

L'environnement de départ de ce projet était le site Web Smartshopping, un portail permettant de naviguer à travers les différents spéciaux quotidiens des magasins d'alimentation de l'île de Montréal, puis de les ajouter à un panier, et enfin d'observer le trajet nécessaire afin de visiter les différents magasins d'où proviennent ces spéciaux. Le but du projet était d'implémenter l'affichage d'un trajet optimal de type Voyageur de Commerce Généralisé entre les différentes franchises des enseignes à visiter, sur une carte Google Maps™, puis incorporer cette fonctionnalité au site Web SmartShopping. L'algorithme précédemment en place choisissait, pour établir un trajet, les magasins qui se trouvaient les plus proches du point de départ, soit l'adresse du client, pour chaque enseigne à visiter. Ce travail consistait donc à comparer les algorithmes de pointe du moment afin d'implémenter le meilleur d'entre eux en termes de rapidité et d'optimalité, pour un échantillon de petite taille. Après analyse et comparaison, un algorithme de type génétique créé par Tasgetiren et al. [TSPL07] a été implémenté en langage C++, en relation avec une page Web codée en PHP, et avec transmission des paramètres par fichiers texte. Par rapport à l'ancien algorithme, les résultats de ce travail montrent une nette amélioration des trajets proposés, et ceci dans l'ensemble des cas testés, avec une moyenne de baisse des distances de 12%, pour les cas de 2 à 5 magasins. Le site Web avec sa nouvelle fonctionnalité peut être consulté à l'adresse URL suivante : http://www.trex.uqam.ca/~smartshopping.

Mots clefs : algorithme, algorithme de voyageur de commerce généralisé, trajet optimal, site web

INTRODUCTION

Le problème du Voyageur de Commerce est un problème fréquemment enseigné en cours d'algorithmique, et les différentes façons de le résoudre sont bien connues de la plupart des personnes versées en informatique théorique. Il existe cependant une version plus complexe de ce problème, de complexité NP-difficile également. Il n'existe donc pas d'algorithme polynomial, au moins à ce jour, qui pourrait trouver la solution optimale de ce problème. Il s'agit du problème du Voyageur de Commerce Généralisé. Ce problème s'énonce de la manière suivante : soit un échantillon de villes réparties en clusters, ou groupements, de plusieurs villes, où un voyageur de commerce doit visiter précisément une ville de chaque cluster, et revenir à la ville de départ, et ce, au moindre coût. La complexité de ce problème est telle que l'on se voit obligé d'utiliser des heuristiques afin de simplifier son calcul, induisant donc dans la résolution du problème un facteur de hasard. C'est pour cette raison que, selon le problème à résoudre, et ses caractéristiques, on se doit de choisir la meilleure heuristique possible, car chacune d'elles est plus appropriée à un certain ensemble de problèmes.

Ce document est structuré comme suit : nous allons d'abord couvrir certaines bases théoriques afin de mieux comprendre le problème, définir notre problème et son environnement, comparer les différents algorithmes qui s'offrent à nous, implémenter le plus adéquat, puis observer et comparer les résultats qu'il donne dans l'environnement de déploiement.

0.1 Notions à connaître sur les graphes [Bon95]

Le graphe est la façon la plus simple de représenter toute sorte de réseau de villes. Nous allons donc revisiter brièvement la théorie des graphes avant de se plonger dans le vif du sujet.

Un graphe est une représentation abstraite d'objets, composée de sommets, qui sont reliés entre eux par des arêtes, c'est-à-dire des arcs ou encore des chemins. Si un graphe est symétrique, alors passer d'un sommet A à un sommet B, ou de B à A, prendra le même temps, selon le poids, ou le coût, du chemin (A, B). Si le graphe est asymétrique, alors le temps de passage dans un sens ou dans l'autre entre deux sommets peut être différent, comme par exemple le trajet d'une adresse à une autre dans une ville, selon les sens uniques et les autres règles de circulation. Les arcs sont **dirigés** dans ce dernier cas.

Comme tous les sommets ne possèdent pas forcément d'arêtes les reliant, on note communément que l'arête, ou l'arc, qui relie deux sommets non-adjacents possède un poids, ou un coût, infini. On peut aussi tirer des arcs entre tous les sommets, ce qui rend le graphe complet, et facilite son codage dans un algorithme.

Notons qu'en général, les graphes ne sont pas forcément complets, et que leurs arcs n'ont nécessairement ni un poids, ni un sens.

0.2 Notion de complexité algorithmique [AVV92]

La complexité d'un algorithme se définit par la quantité de ressources, en temps d'exécution et en espace mémoire, que l'algorithme consomme.

La notion de « grand O » sert donc à caractériser cette complexité (voir figure 0.1), et donne la fonction qui borne asymptotiquement, à un facteur près, la fonction qui représente le temps de calcul de l'algorithme, selon la taille de l'échantillon. Ainsi, si on dit que l'algorithme est de complexité $O(n^2)$ dans le pire des cas, on considère que l'algorithme

prendra au maximum un temps $x * n^2 + y, x \in R$ avant de trouver la solution au problème, et y peut être une expression polynomiale de degré inférieur à n^2.

Notation	Type de complexité
$O(1)$	complexité constante (indépendante de la taille de la donnée)
$O(\log(n))$	complexité logarithmique
$O(\sqrt{n})$	complexité racinaire
$O(n)$	complexité linéaire
$O(n\log(n))$	complexité linéarithmique
$O(n^2)$	complexité quadratique
$O(n^3)$	complexité cubique
$O(n^p)$	complexité polynomiale
$O(e^n)$	complexité exponentielle
$O(n!)$	complexité factorielle
$O(2^{2^n})$	complexité doublement exponentielle

Figure 0.1 Notation de complexité algorithmique.

(source : http://fr.wikipedia.org/wiki/Théorie_de_la_complexité_des_algorithmes)

On classe ces problèmes et leurs complexités dans de grandes catégories de complexité :

- *L* et *NL* : un problème fait partie de *L* s'il peut être résolu en temps logarithmique sur une machine déterministe (qui n'a qu'une possibilité à un moment donné); il fait partie de *NL* s'il peut être résolu en temps logarithmique sur une machine non-déterministe (qui a plusieurs choix à un moment donné, et qui nécessite donc d'explorer ces différentes possibilités).

- *P* et *NP* : un problème fait partie de *P* s'il peut être résolu en temps polynomial sur une machine déterministe ; il appartient à *NP* s'il peut être résolu sur une machine non-déterministe en temps polynomial.
- *NP-complet* : englobe les problèmes dont les solutions peuvent être vérifiées en temps polynomial, mais ces solutions prennent un temps exponentiel à être trouvées.
- *NP-difficile* : un problème est *NP-difficile* si un autre problème qui a déjà été démontré comme étant *NP-difficile* peut être réduit à ce problème en temps polynomial.

0.3 Voyageur de commerce classique

Afin de mieux comprendre le procédé de résolution du problème du Voyageur de Commerce Généralisé, qui fera l'objet de ce mémoire, attardons-nous tout d'abord sur le problème du Voyageur de Commerce Standard [Lap92].

Celui-ci utilise un échantillon d'un certain nombre de villes, avec des distances données, ou des poids, entre chacune d'elles, et cherche à trouver le chemin le plus court qui passe une seule fois par chaque ville (figure 0.2).

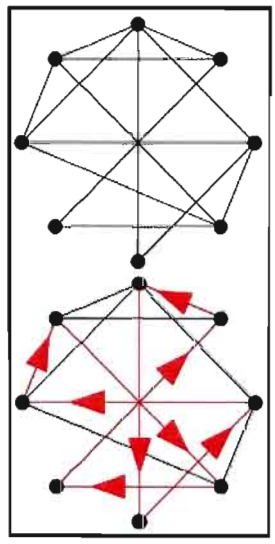

Figure 0.2 Exemple de parcours pour le problème du Voyageur de Commerce Standard.

En 1992, Laporte définit le problème comme étant le suivant [Lap92]: soit un graphe **G** = (V, A) composé d'un ensemble V de n sommets, et d'un ensemble A d'arêtes. Soit C : (C_{ij}) une matrice de distances relative à A. On cherche à trouver un circuit Hamiltonien, c'est-à-dire un circuit qui ne passe qu'une seule fois par chaque sommet, de longueur minimale. Il est important de déterminer si le problème est symétrique, et donc que chaque distance i vers j est la même que la distance de j vers i, ou asymétrique, si ce n'est pas le cas. Dans le cas symétrique, on notera que la matrice C satisfait l'inégalité triangulaire (pour tout triplet de points (x, y, z), la distance $d(x, y)$ est plus petite ou égale à $d(x, z) + d(z, y)$) dans le cas où V se trouve sur un plan en deux dimensions, et que les distances entre les sommets sont à vol d'oiseau (distances Euclidiennes).

En matière de complexité, le problème de circuit Hamiltonien est un problème *NP-difficile*, le problème du Voyageur de Commerce est aussi un problème *NP-difficile*, tout comme le problème du Voyageur de Commerce Généralisé. [GJS74, RSLI77]

Nous avons deux grandes catégories de méthodes de résolution du problème du Voyageur de Commerce : les **méthodes exactes**, qui entraînent un temps de résolution exponentiel qui explose avec de grands échantillons (on se voit confronté à des dizaines d'heures de calcul pour un échantillon de 20 villes), et les **méthodes approximatives, ou heuristiques**, qui ne considèrent qu'une partie plus intéressante du problème pour s'approcher de la solution optimale, et même la trouver parfois, en un temps de calcul raisonnable [LKB08].

Pour les **méthodes exactes**, le plus simple est une exploration de toutes les permutations possibles, que ce soit en arbre ou pas, de façon « brute-force ». Il est néanmoins possible de mettre des conditions sur l'exploration de ces permutations afin de diminuer le temps de calcul total : on commence par déterminer un premier trajet dit « maximal » parmi l'ensemble des trajets; ainsi, le trajet optimal sera forcément plus court ou égal à ce trajet maximal. On calcule aussi un trajet « minimal » pour un chemin donné : à chaque sommet que l'on parcoure, on additionne les distances parcourues aux X distances minimales qu'il reste entre les X sommets non-parcourus. Si le trajet « minimal » est plus grand que le trajet « maximal », alors toutes les permutations qui sont issues des sommets déjà parcourus peuvent être abandonnées. Si l'on trouve un meilleur chemin que notre trajet maximal, on met à jour ce dernier avec le meilleur chemin, et on continue à explorer les permutations restantes. On peut aussi, afin d'espérer diminuer le temps de calcul, choisir d'explorer les sommets selon le rapprochement de ceux-ci, à la place d'un autre ordre (le plus commun étant l'ordre alphabétique). Il existe aussi une solution de programmation dynamique qui, en utilisant un système d'inclusion-exclusion, peut résoudre le problème en un temps $O(2^n)$ [LKB08].

Quant aux **méthodes heuristiques**, il existe plusieurs approches [LKB08]:

- Par algorithme dit de type « glouton » (algorithme le plus basique, qui va toujours choisir le chemin le plus avantageux à un moment donné, ce qui n'est pas forcément optimal, car il ne voit pas le problème dans son ensemble), la méthode du **plus proche voisin** va, à chaque étape choisir comme prochaine ville celle qui est à moindre distance dans la liste de ses voisins non parcourus. Cet algorithme

trouve en moyenne un chemin 25% plus long que le chemin optimal, pour des villes distribuées au hasard sur un plan.

- Par algorithme glouton, la méthode d'**insertion** va partir d'un chemin entre deux villes, puis va insérer chaque autre ville l'une après l'autre, de manière à ce que la longueur du trajet reste minimale (on va tester, selon les arêtes qui nous sont proposées, toutes les positions de la nouvelle ville dans le trajet, et choisir l'emplacement pour celle-ci qui augmente le parcours total de manière minimale).
- Par composition de méthodes, on a l'algorithme de **descente locale**, qui consiste à appliquer en un premier temps un algorithme glouton, puis améliorer la solution, en essayant par exemple pour chaque nœud de le permuter dans le chemin avec un autre du même voisinage, et de comparer la nouvelle longueur de trajet avec l'ancienne. On choisit la meilleure solution du voisinage avant de passer à un autre.
- Le **recuit simulé** est une méthode qui consiste à utiliser la descente locale, mais de considérer aussi les moins bonnes solutions pour un voisinage avant de passer à un autre : deux solutions locales quasi-optimales pour deux voisinages peuvent des fois mieux s'approcher de la solution optimale globale que deux solutions locales optimales. La condition d'arrêt est une trop grande dégradation de la solution selon le nombre d'itérations à travers les voisinages.
- Les **algorithmes génétiques**, quant à eux, sont inspirés de la loi d'évolution de Darwin : On génère plusieurs « individus », ou parcours, avec un algorithme glouton, on choisit ceux de plus petite longueur, qui sont plus « prompts à survivre », et on les accouple, c'est-à-dire qu'on les croise. Par exemple, on peut utiliser une moitié de parcours de l'un et de l'autre, en faisant attention de supprimer les doublons et ajouter à la fin les villes oubliées. Le point de coupure est appelé un « point de croisement ». On peut enfin améliorer le trajet obtenu à l'aide d'un algorithme de recherche locale.
- L'algorithme des **colonies de fourmis** suit un principe simple : on envoie des fourmis d'un point A à un point B, et comme chaque fourmi laisse des phéromones de marquage, et que la fourmi qui aura trouvé le chemin le plus court fera plus d'allers-retours, le meilleur trajet sera marqué plus fortement par les

phéromones. Les phéromones s'évaporant après un certain temps, les trajets peu avantageux seront de moins en moins marqués, et n'inciteront pas les autres fourmis à les emprunter. Il suffit donc de lancer les fourmis dans le graphe, avec quelques contraintes : passer une seule fois par tous les nœuds, revenir au point de départ, marquer chaque arête avec des phéromones inversement proportionnelles à la longueur de l'arête, suivre le chemin avec le plus de phéromones, et, finalement, faire que le marquage par les phéromones s'atténue dans le temps. Ainsi, après un certain nombre d'itérations, toutes les fourmis passeront par le chemin le plus court.

La plupart de ces méthodes se retrouvent dans le chapitre traitant du problème du Voyageur de Commerce Généralisé, avec cependant quelques ajustements afin d'adapter le problème à des clusters, ou groupements, de villes, dont on doit visiter uniquement une ville.

CHAPITRE I

PROBLÉMATIQUE

Le problème du Voyageur de Commerce Généralisé (Generalized Traveling Salesman Problem, ou **GTSP**) [FGT97] a poussé de nombreux scientifiques à chercher des solutions proches de la solution optimale, toutes avec des temps de calcul différents, selon la taille de l'échantillon.

Nous avons comparé ces différentes méthodes existantes afin de choisir la plus pertinente pour notre problème, de l'adapter, de l'implémenter, et ainsi d'apporter une fonctionnalité supplémentaire très importante à des sites Web qui mettent en œuvre des cartes de type Google Maps™ [Goo11].

Notre problème exact est le suivant :

Nous possédons une liste de tous les magasins d'alimentation de l'île de Montréal. Nous avons comme facteur extérieur une liste des enseignes (ou clusters) de magasins d'alimentation par lesquelles un client désire passer. Notre but est donc de choisir une et une seule franchise (ou magasin) par enseigne de magasin, afin de livrer au client le trajet optimal entre les enseignes de magasins qu'il a choisies. Une fois les franchises déterminées, il s'agit d'afficher le trajet optimal sur une carte Google Maps™. Il a fallu aussi construire un site Web entier autour de cette fonctionnalité.

La problématique du Voyageur de Commerce Généralisé est toujours d'actualité, avec de nombreux chercheurs qui continuent à trouver des heuristiques toujours meilleures, donnant des solutions de plus en plus proches de la solution optimale pour des échantillons de plus en plus grands.

Il existe plusieurs types d'approches pour résoudre le problème du Voyageur de Commerce Généralisé:

- Approche avec prétraitement des données [GK08],
- Approche branch-and-cut [FGT97],
- Approche de réduction du problème pour se rapprocher d'un simple problème de Voyageur de Commerce [BM02, DS97],
- Approche avec heuristique composite de création de solutions partielles, et d'amélioration [RB98],
- Approche avec recherche locale approfondie des clusters [KG10b],
- Approche d'adaptation de l'heuristique de Lin-Kernighan pour le Voyageur de Commerce Généralisé [KG10a],
- Approche avec heuristique qui mime la sélection naturelle génétique [SG07, MP10, SD04, GK10, TSPL07, BAF10].

Chacune de ces approches a ses propres caractéristiques: un certain taux d'erreur par rapport à la solution optimale, un temps d'exécution plus ou moins long, ou une précision variable par taille d'échantillon. Notre but était de les comparer et de trouver celle offrant la meilleure optimalité pour un site Web, qui a besoin d'un temps d'exécution rapide et d'une grande précision pour un petit échantillon.

L'API (Application Programming Interface) de Google Maps™ procure déjà une fonctionnalité de trajet optimal entre plusieurs destinations fixes qui lui sont données. Néanmoins, il ne propose aucune solution pour un problème comme celui du Voyageur de

Commerce Généralisé, sans obligation de passer par tous les lieux, mais avec obligation d'en choisir certains.

C'est pourquoi l'implémentation d'un algorithme de Voyageur de Commerce Généralisé est indispensable en complément à ce qu'une carte Google Maps™ peut offrir dans le cadre d'un problème de recherche de parcours optimal comme le nôtre. Cet algorithme permettra d'offrir aux utilisateurs un temps de parcours réduit, et donc de leur faire gagner un temps précieux, ce qui est indispensable pour l'image d'un site Web, et donc pour sa popularité.

CHAPITRE II

MÉTHODOLOGIE

2.1 Environnement d'implantation de l'algorithme

L'algorithme développé a été nécessaire pour apporter une fonctionnalité de calcul et d'affichage d'un trajet optimal au site Web **SmartShopping** (http://www.trex.uqam.ca /~smartshopping). L'idée de fond du site est de permettre aux montréalais(es) de faire leurs plans de courses en ligne, d'avoir accès aux prix les plus bas parmi les différentes offres spéciales («spéciaux»), de même que les prix réguliers des différents magasins d'alimentation de Montréal, et finalement, de pouvoir visualiser une carte leur indiquant le trajet optimal en voiture ou à pied entre les différents magasins des enseignes choisies. Ainsi, les utilisateurs de ce site feront de grandes économies dans leurs courses grâce aux prix réduits des produits, tout en minimisant le coût du trajet.

Etant donné que l'idée du site était aussi de faire gagner du temps à ses clients, il était donc indispensable de rendre le trajet le plus économique, ce qui n'était pas possible sans l'intégration de notre algorithme GTSP.

Nous avons donc créé plusieurs pages : l'accueil ; l'accès aux produits, en « spécial » ou non, classés par catégorie, avec la possibilité de rechercher des produits ; l'accès au panier courant, avec la carte de trajet automatiquement affichée en dessous ; l'identification du client ; et enfin, la liste et la présentation des magasins d'alimentation de Montréal.

La conception et la réalisation du site internet dans lequel va être implémenté l'algorithme de Voyageur de Commerce Généralisé est à mettre au crédit d'Alix Boc, Mathias Lino, Benjamin Geyre, Yury Kaptsevich et Tania Joly, l'auteure de ce mémoire.

2.1.1 Fonctionnalités du site

Mentionnons qu'avant l'implantation de l'algorithme, le site était déjà fonctionnel. Voici quelques détails de son fonctionnement. Grâce à la page d'accueil (voir figure 2.1), le client peut avoir accès à la liste des produits par catégorie, ainsi qu'à une fonction de recherche à travers les produits. Une fois la recherche lancée ou une catégorie sélectionnée, le client peut ajouter à son panier les produits qui lui plaisent (pour un exemple de produits pour une catégorie, voir figure 2.2, et pour un exemple de page de recherche, voir figure 2.3). Une fois les produits ajoutés à son panier, le client peut aller voir le contenu de celui-ci sur la page « Panier courant », et changer les quantités de ses produits, ou encore supprimer un produit complètement (voir figure 2.4). Ce sont les enseignes des magasins recensées dans ce panier que l'on affiche dans le parcours proposé au client.

Les produits sont mis à jour chaque matin à l'aide d'un script automatisé qui va chercher les spéciaux actuels depuis les sites web des différents fournisseurs.

Une fois son panier créé, le client peut à tout instant vérifier son contenu, ainsi qu'un trajet semi-optimal sur la page du panier. Avant l'implantation de l'algorithme, le trajet était calculé sur la base de la liste des enseignes à visiter, et de leurs franchises les plus proches de l'adresse du client. Ce dernier aura précisé son adresse dans la barre d'entête du site, ainsi que le rayon maximal où il est prêt à se déplacer. C'était une solution originelle temporaire peu efficace, en attendant l'implantation de l'algorithme du Voyageur de Commerce Généralisé, car elle ne considérait pas la possibilité que deux franchises un peu plus éloignées de l'adresse du client, mais très proches l'une de l'autre, puissent alors faire partie d'un trajet total plus court. Il fallait donc remplacer le choix des franchises les plus proches de l'adresse du client par celles données par notre algorithme du Voyageur de Commerce Généralisé.

Sur le site SmartShopping, le client peut aussi s'enregistrer, avec la possibilité d'avoir son adresse personnelle retenue en mémoire, de même que ses anciens paniers.

Figure 2.1 Page d'accueil du site SmartShopping, avec les catégories de produits affichées sur la droite.

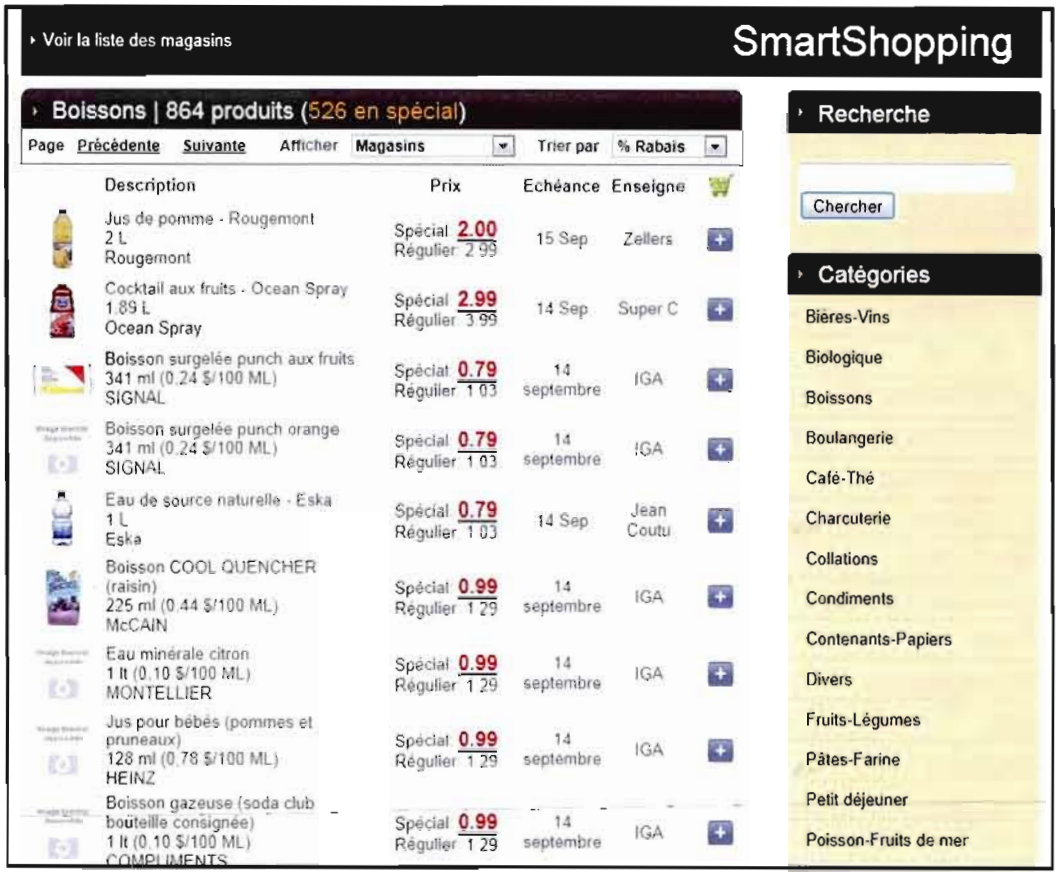

Figure 2.2 Une exemple : la liste des produits de la catégorie « Boissons ».

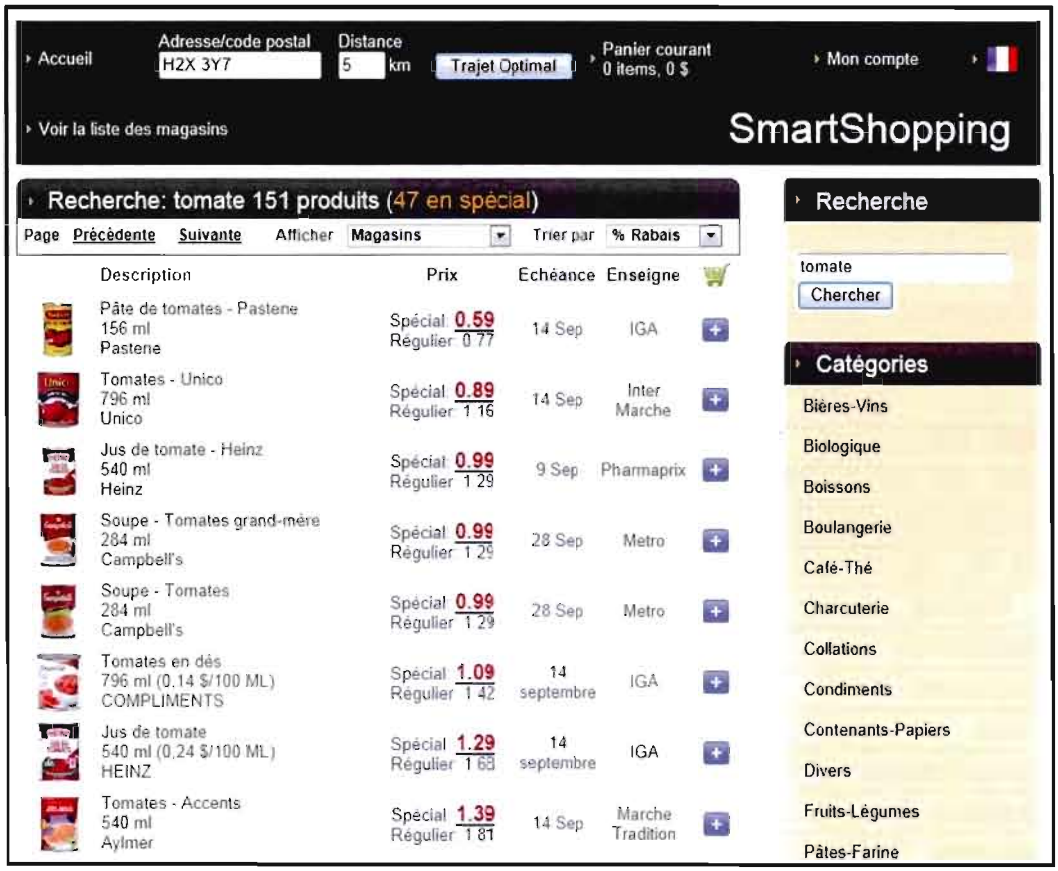

Figure 2.3 Une exemple : la liste des produits après recherche effectuée avec le mot-clef « tomate ».

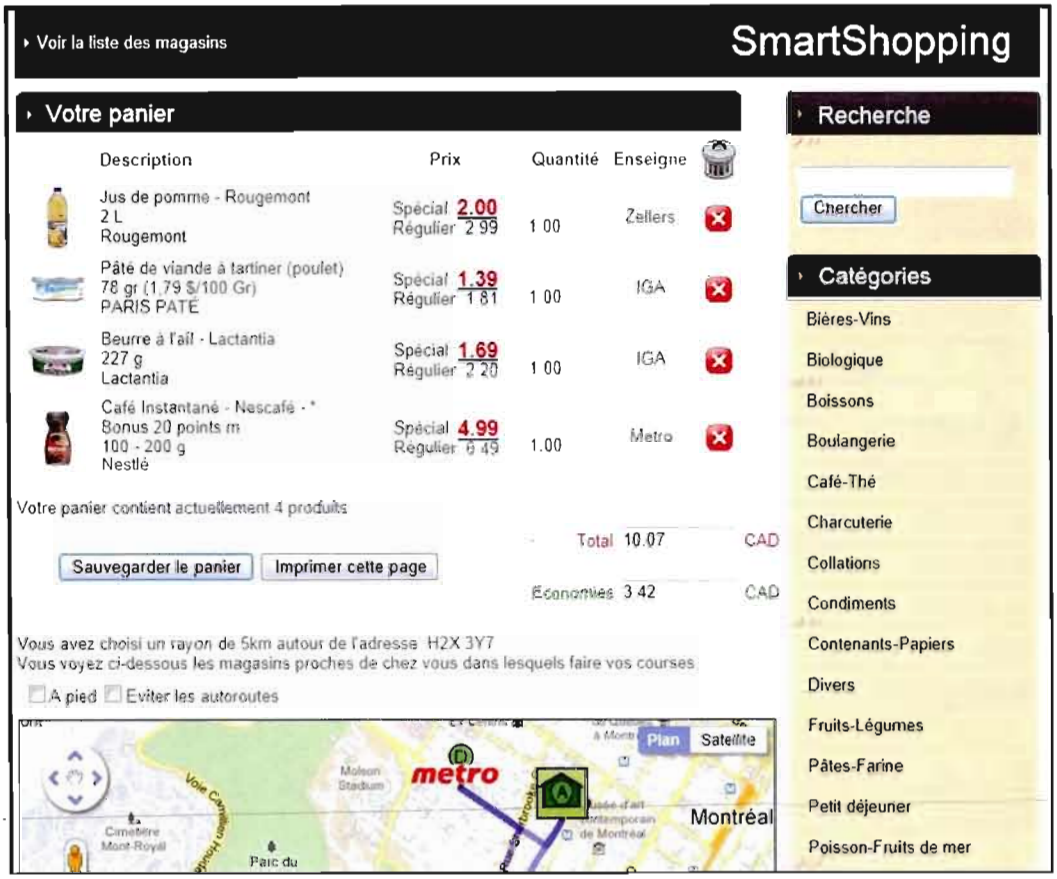

Figure 2.4 Contenu du panier courant : l'affichage de la carte se fait par les magasins listés dans le panier.

2.1.2 Accès à la base de données

Une base de données est nécessaire pour stocker les produits, leurs attributs, et leur prix selon le magasin, ainsi que les informations liées aux magasins et aux clients. La base utilisée est une base MySQL [Ora11], dont on peut voir le schéma UML sur la figure 2.5.

La base de données est appelée lors de l'affichage des produits, de la création du panier du client, de la mise à jour du panier du client selon s'il ajoute ou retire des objets dans son panier, lors de la recherche de franchises proches du client, puis enfin, lors de l'enregistrement ou de la connexion du client.

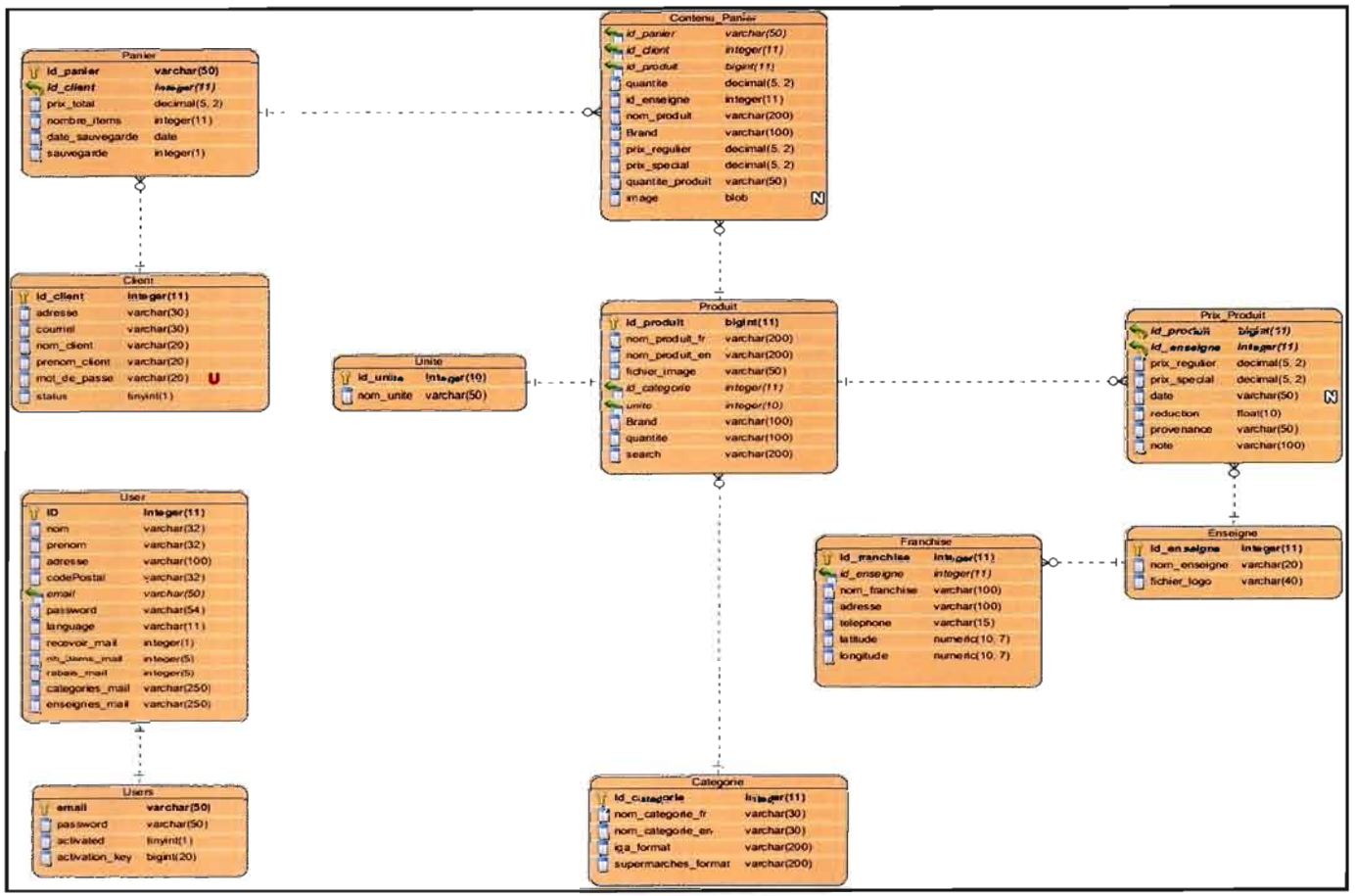

Figure 2.5 Schéma UML de la base de données du site SmartShopping.

2.1.3 Accès à Google Maps™

L'accès à Google Maps ™ se fait en incluant à la page qui abrite la carte Google, une librairie JavaScript de source «*http://maps.googleapis.com/maps/api/js?sensor=false* » qui nous permet d'inclure une carte Google et d'utiliser toutes les méthodes JavaScript qui y sont attachées.

Avant l'implantation, l'utilisation des méthodes se résume ainsi : on commence par assigner un <*div*> HTML à la future carte Google, un autre au parcours textuel, puis à l'aide des informations du panier, on fait une sélection des franchises nécessaires les plus proches de l'adresse du client avec une distance à vol d'oiseau, en prenant soin d'éliminer toutes les franchises qui se trouvent plus loin que la limite de distance donnée par le client. On produit ensuite la liste des adresses de ces franchises à la méthode « *directionsService.route*» de l'application Google Maps™, et le trajet, ainsi que les instructions, apparaîtront lors de l'affichage de la page. En surimpression, on ajoute le logo d'une maison pour symboliser l'adresse d'origine du client, ainsi que les logos de chaque enseigne, avec la fonction d'ajout de markers sur la carte (« *markersArray.push(marker)* »). On calcule finalement le temps total et la distance totale du trajet en ajoutant les différents tronçons du parcours, pour le présenter au client (voir figure 2.6).

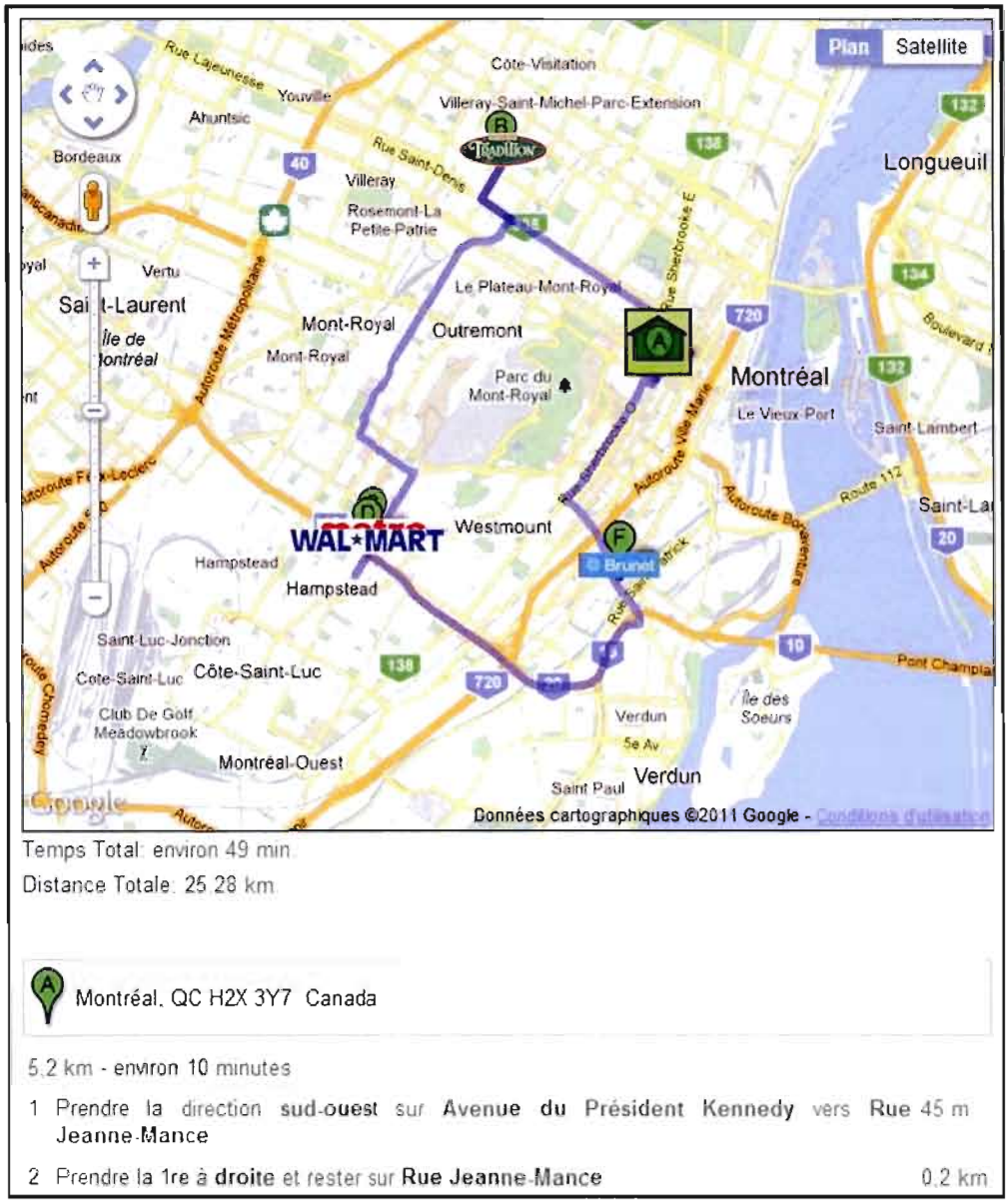

Figure 2.6 Exemple de carte Google Maps™, avec indications routières, et étiquettes ajoutées pour indiquer le point de départ et la position des magasins.

2.2 Algorithme

Il existe de nombreuses approches à la résolution du problème du Voyageur de Commerce Généralisé [GK08, FGT97, BM02, DS97, RB98, KG10b, KG10a, SG07, MP10, SD04, GK10, TSPL07, BAF10]. Toutes possèdent leurs propres caractéristiques de temps de calcul et d'optimalité sur un échantillon de taille spécifique. Il s'agissait pour nous de trouver l'approche qui réduirait au mieux le temps de calcul afin de rendre notre site le plus efficace possible, ainsi que d'assurer un bon taux d'optimalité pour un échantillon de petite taille.

Le problème de base est énoncé de la façon suivante : nous possédons un graphe **G** complet et non-dirigé dans un premier temps (nous commençons par élaborer l'algorithme en tenant compte de distances à vol d'oiseau), possédant un poids pour chaque arête ; **G** possède n sommets, et V l'ensemble des clusters, qui sont des partitions de ces n sommets, où V_i représente un cluster. Dans notre cas, les sommets sont les franchises (ou magasins) de toutes les enseignes de Montréal, et chaque cluster représente une enseigne, contenant un certain nombre de magasins. L'objectif est de trouver un parcours de coût minimal tel que le trajet ne passe que par un seul sommet de chaque cluster. Comme il s'agit d'une extension du problème du Voyageur de Commerce Classique (celui-ci est un cas particulier du généralisé, lorsque, pour tout i, V_i ne possède qu'un seul sommet, ou, dans notre cas, une seule franchise), il s'agit aussi d'un problème *NP-difficile*. Ici, les poids seront souvent décrits comme des distances entre deux points, ou sommets, x et y, et seront notés $dist(x, y)$.

Précisons que de nombreux algorithmes utilisent des méthodes d'optimisation des trajets nommées 2-opt (figure 2.7) et 3-opt ; il s'agit de méthodes qui tentent, pour toutes deux et trois arêtes respectivement, de supprimer ces arêtes et les réarranger, jusqu'à l'obtention d'un tour de longueur plus courte.

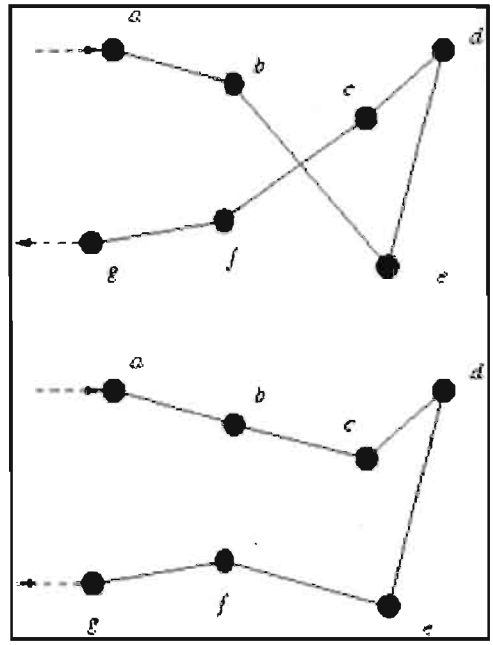

Figure 2.7 Démonstration de la méthode 2-opt.

(source : http://fr.wikipedia.org/wiki/Fichier:2-opt_wiki.png)

Quelques algorithmes utilisent aussi la méthode du « Swap », qui implique l'échange de deux nœuds ; soit par échange de deux nœuds du trajet (donc changement de l'ordre des clusters dans le tour), soit par remise en question de l'un des nœuds choisis pour un cluster, et remplacement par chaque nœud de ce cluster jusqu'à obtenir un meilleur trajet.

Dans ce travail, le graphe utilisé sera un graphe complet pour chaque enseigne de magasin, car tout magasin est atteignable depuis un autre dans la ville. Dans un premier temps, nous utiliserons un graphe symétrique non-dirigé, car nous considérerons les distances à vol d'oiseau entre les magasins, qui sont les mêmes dans un sens comme dans l'autre. Nous utiliserons ensuite un graphe asymétrique, une fois que l'on aura obtenu les distances effectives dans la circulation, qui ne sont pas les mêmes d'un point A vers un point B, ou de B vers A (pour causes de sens interdits et autres règles de circulation), s'il nous est possible de les récupérer.

2.2.1 Prétraitement des données [GK08]

Comme Gutin et Karapetyan l'ont démontré [GK08], prétraiter les données se révèle d'une grande importance pour la réduction du temps de résolution du problème du Voyageur de Commerce Généralisé. Ils utilisent comme prétraitement deux algorithmes, l'un dédié à la réduction du nombre de sommets (nœuds, ou villes) et l'autre à la réduction du nombre d'arêtes.

2.2.1.1 Réduction des sommets

Afin d'éliminer les sommets inutiles au trajet optimal du Voyageur de Commerce, Gutin et Karapetyan [GK08] utilisent un système de nœuds redondants. Ils définissent ces derniers comme ceci : C est un cluster non-vide avec au moins 2 nœuds ($|C| > 1$) ; un nœud $r \in C$ est redondant si, pour chaque paire de nœuds x_1 et x_2 qui se trouvent chacun dans deux clusters distincts et différents de C, il existe un nœud $s \in C$ différent de r qui respecte la formule suivante : $dist(x_1, s) + dist(s, x_2) \leq dist(x_1, r) + dist(r, x_2)$.

Ainsi, on note comme redondants tous les nœuds qui ne seront jamais favorables au Voyageur de Commerce, peu importe les nœuds des autres clusters choisis. Le calcul de cette redondance pour chaque nœud prend un temps d'ordre $O(n^3 + n/m)$, où m est le nombre de clusters, et n/m est la taille moyenne de ceux-ci.

Afin de comparer les différentes distances entre s, r, et x_i, on crée une table de différences (figure 2.8). Chaque entrée du tableau est égale à $dist(r, x_i) - dist(s, x_i)$. Si r est redondant, cela signifie que l'ensemble des s (qui peuvent être les sommets 1 ou 2 (v.1 ou v.2) sur la figure 2.9) est plus proche de tous les nœuds des deux autres clusters. Donc r n'est pas redondant si pour tous les s possibles, il existe x_i et x_j tels que : $dist(x_i, r, x_j) - dist(x_i, s, x_j) < 0$.

On calcule donc pour notre exemple en figure 2.8 les différents $dist(x_i, r, x_j) - dist(x_i, s, x_j)$, et à cause de la ligne 3 sur la figure 2.9, qui montre des sommes négatives pour les deux s possibles (-3 et -1), nous ne pouvons pas déclarer r redondant.

$s \backslash x$	cl.2 v.1	cl.2 v.2	cl.2 v.3	cl.3 v.1	cl.3 v.2	Negative #
v.2	2	0	−1	−3	4	2
v.3	−1	−2	−1	1	2	3
max	2	0	−1	1	4	
		$\min\{2, 0, -1\} = -1$		$\min\{1, 4\} = 1$		

Figure 2.8 Exemple d'une table de différences [GK08].

Pair	Sum for s = v.2	Sum for s = v.3
cl.2 v.1—cl.3 v.1	−1	0
cl.2 v.1—cl.3 v.2	6	−1
cl.2 v.2—cl.3 v.1	−3	−1
cl.2 v.2—cl.3 v.2	4	0
cl.2 v.3—cl.3 v.1	−4	0
cl.2 v.3—cl.3 v.2	3	1

Figure 2.9 Sommes des distances entre les paires de nœuds [GK08].

2.2.1.2 Réduction des arêtes

Soit u et v des sommets qui appartiennent à deux clusters différents U et V, respectivement. L'arête uv est redondante si, pour tout sommet x qui appartient à un troisième cluster C distinct de U et V, il existe un chemin x-v'-u qui est plus court que le chemin x-v-u ($dist(x, v') + dist(v', u) \leq dist(x, v) + dist(v, u)$). Le calcul de cette condition prend lui aussi

$O(n^3 + n/m)$, où m est le nombre de clusters, et n/m est la taille moyenne de ceux-ci. L'article de Gutin et Karapetyan [GK08] décrit une méthode de réduction des arêtes qui peut s'appliquer à un GTSP symétrique (avec des distances entre deux sommets A et B égales, que le trajet soit de A vers B ou de B vers A), qui parcourt les différentes arêtes entre les trois clusters U, V et C, et élimine les arêtes redondantes.

2.2.2 Branch-and-Cut

Fischetti et al. [FGT97] nous proposent un algorithme à solution exacte, nommé branch-and-cut. Cet algorithme considère un graphe $\mathbf{G}=(V, A)$ non-dirigé et complet, à clusters. On cherche donc un sous-ensemble S de nœuds qui appartiennent à V, avec S qui possède au moins un nœud par cluster. Après, il nous faut trouver le cycle Hamiltonien le plus court dans S.

Cet algorithme a longtemps été la base de comparaison pour tous les algorithmes à solution approximative, afin de savoir le pourcentage d'erreur moyen d'une heuristique pour un échantillon donné. Ainsi, il nous permet de mesurer la précision d'un algorithme heuristique. Cependant, comme il parcourt presque toutes les solutions, il est extrêmement lent, et ne nous intéresse pas dans le cadre de ce projet.

2.2.3 Réduction du problème du Voyageur de Commerce Généralisé au problème standard

L'idée de transformer un problème complexe en un problème dont on connait bien les solutions, comme celui du Voyageur de Commerce Standard est tentante, et de nombreux chercheurs se sont penchés dessus, et sont parvenus à créer des algorithmes à complexité intéressante [DS97] [BM02]. Pourtant, comme l'expliquent Karapetyan et Gutin dans leur article de 2010 [KG10b], cette approche a une application très limitée, car elle requiert que, une fois le GTSP transformé en TSP, on puisse trouver la solution exacte et optimale du TSP, car une solution quasi-optimale d'un TSP peut correspondre à un trajet impossible pour le

GTSP. Ceci est d'autant plus difficile que le TSP obtenu par transformation possède une structure très compliquée que les méthodes de résolution habituelles pour TSP ont de la peine à résoudre avec exactitude.

On se rend donc compte que cet algorithme n'est pas approprié à nos besoins, car il est trop coûteux en temps de calcul.

2.2.4 Heuristique d'amélioration

Comme Renaud et al. [RB98] le démontrent dans leur papier datant de 1998, on peut souvent améliorer des solutions non-optimales à l'aide d'heuristiques d'amélioration, après un certain nombre d'itérations. Ils proposent un algorithme en trois temps, nommé le **GI^3**, avec une première phase d'initialisation, une deuxième phase d'insertion, et une troisième phase d'amélioration, qui s'applique au GTSP symétrique.

Lors de la première étape, pour chaque cluster C, et chaque nœud x du cluster C, on fait la somme des distances du nœud x à tous les nœuds des autres clusters. On créera un premier trajet provisoire composé du x de chaque cluster avec une somme de valeur minimale.

Une deuxième étape, intitulée heuristique « CLOCK », consiste à insérer les nœuds les plus extérieurs du trajet provisoire dans une suite H, avec les nœuds ordonnés à la suite selon leur emplacement géographique, et selon un parcours dans le sens des aiguilles d'une montre. Pour ce faire, on choisit cinq nœuds : le nœud *le plus au nord*, le prochain nœud *le plus au nord* à l'est du premier, le prochain nœud *le plus à l'est* au sud du second, le prochain nœud *le plus au sud* à l'ouest du troisième, et finalement, le prochain nœud *le plus à l'ouest* au nord du quatrième. Ce chemin crée une sorte d'enveloppe autour de nos points de somme minimale. Ces nœuds-là seront la base de notre trajet optimal, comme il s'agit de nœuds les plus proches des clusters les plus éloignés.

Le temps de calcul de ces deux premières phases est de $O(n^2)$.

On en arrive à l'étape d'insertion : pour chaque cluster non-visité dans la suite H, on observe quel nœud sera plus avantageux de placer entre deux nœuds de la suite H. Il faut alors comparer tous les emplacements entre deux clusters X et Y donnés de la suite H. Afin de réellement trouver la meilleure association de nœuds, on se permet même, dans chaque cluster X et Y de H choisis, de vérifier si un autre nœud est plus avantageux que celui d'origine choisi par CLOCK. On effectue cette opération pour tous les clusters restants.

Ensuite, on se trouve face à un problème de Voyageur de Commerce Classique. On lui applique donc une heuristique d'amélioration dérivée de la descente locale (2-opt) : on permute des portions de chemin, et si la solution est meilleure, on la garde ; on fixe un nombre d'itérations, qui sera proportionnel à l'optimalité de la solution finale. On ne peut pas prédire le nombre d'itérations que prendra cette partie de l'algorithme pour atteindre la solution optimale, ce qui nous empêche de connaître son temps de calcul total.

Comparée à d'autres algorithmes combinés avec heuristiques d'amélioration, cet algorithme donne des solutions plutôt proches de l'optimum, avec une erreur moyenne de 0.98% par rapport à l'optimum, et utilise environ 83 tics d'horloge interne de processeur, tandis que la plupart des autres méthodes prennent plus de 3900 tics d'horloge interne en moyenne. Cependant, ce genre de temps et d'optimalité de sont pas compétitifs comparés aux algorithmes génétiques, qui sont beaucoup plus rapides et proches de la solution exacte, comme on le verra plus loin.

2.2.5 Amélioration de solution avec recherche locale approfondie

Karapetyan et Gutin [KG10b] proposent dans leur article de 2010 plusieurs méthodes pour améliorer un algorithme de résolution du problème de Voyageur de Commerce Généralisé, avec plusieurs niveaux de recherche locale de l'optimum.

Ils proposent tout d'abord une méthode d'optimisation appelée « Cluster Optimization » (voir figure 2.10), qui cherche, pour un ordre donné de cluster à visiter, à trouver pour chacun d'entre eux le nœud qui minimise la longueur du tour.

Algorithm 1 Cluster Optimization. Basic implementation.

Require: Tour $T = T_1 \to T_2 \to \ldots \to T_m$.
  Let $\mathcal{T}_i = Cluster(T_i)$ for every i.
  for all $v \in \mathcal{T}_1$ and $r \in \mathcal{T}_2$ **do**
    Initialize the shortest path from v to r: $p_{v,r} \leftarrow (v \to r)$.
  for $i \leftarrow 3, 4, \ldots, m$ **do**
    for all $v \in \mathcal{T}_1$ and $r \in \mathcal{T}_i$ **do**
      Set $p_{v,r} \leftarrow p_{v,u} + (u \to r)$, where $u \in \mathcal{T}_{i-1}$ is selected to minimize $w(p_{v,u} + (u \to r))$.
  return $p_{v,r} + (r \to v)$, where $v \in \mathcal{T}_1$ and $r \in \mathcal{T}_m$ are selected to minimize $w(p_{v,r} + (r \to v))$.

Figure 2.10 Pseudo-code du "Cluster Optimization" [KG10b].

Karapetyan et Gutin proposent ensuite d'utiliser une méthode de voisinage déjà existante parmi les *k*-opt (qui suppriment *k* arêtes et testent toutes les façons de les réinsérer), les insertions (qui déplacent un nœud donné à tous les emplacements possibles du tour), les or-opt (qui prennent trois sommets et leur appliquent une insertion), ou le swap (qui échange deux sommets dans un tour). Ils appliquent une de ces méthodes jusqu'à l'obtention du meilleur tour possible.

Afin de trouver la solution la plus proche de l'optimal, ils appliquent tout d'abord l'optimisation des clusters jusqu'à ne plus trouver de meilleure solution, puis ils appliquent une recherche locale jusqu'à ne plus trouver de solution, et ils reviennent à l'optimisation des clusters, car la recherche locale ouvre de nouvelles possibilités de solutions. Ainsi, on applique tour à tour chaque optimisation jusqu'à ne plus trouver de meilleure solution.

Les auteurs notent cependant que, bien que l'application d'algorithmes de recherche locale améliore drastiquement la qualité d'une solution partielle, ces algorithmes augmentent énormément le temps d'exécution. Une optimisation de cluster prend en moyenne de 4 à 9 cycles d'horloge de processeur, et les autres optimisations, de 0.17 à 30 cycles. Cette association d'algorithmes n'est donc pas compétitive face aux algorithmes génétiques, qui prennent en moyenne en dessous de 3 cycles d'horloge de processeur pour leur exécution.

2.2.6 Heuristique de Lin-Kernighan

L'heuristique de Lin-Kernighan (Lin et Kernighan, 1973) est très connue pour être une heuristique de pointe pour le problème du Voyageur de Commerce Standard. Karapetyan et Gutin discutent dans un article de 2010 [KG10a] comment appliquer cet algorithme au problème généralisé, et trouver la solution avec un très bon temps d'exécution.

Ils expliquent tout d'abord le fonctionnement général de l'algorithme pour le Voyageur de Commerce Standard :

On choisit T un tour originel non-optimal. Pour chaque arête x de T, on enlève x, et on optimise le chemin non-circulaire obtenu P ($T\backslash x$) en intervertissant l'ordre de passage. Si on trouve un nouveau chemin P' plus court que le P originel, on essaie de refermer le tour (lier les deux extrémités), et si ce tour est plus court que T, on remplace T par celui-ci et on recommence la manœuvre. Si le nouveau tour n'est pas plus court, on passe alors à une autre arrête x.

Le pseudo-code de cet algorithme est présenté sur les figures 2.11, 2.12, et 2.13.

Algorithm 2. LK general implementation

Require: The original tour T.
  Initialize the number of idle iterations $i \leftarrow 0$.
  while $i < m$ **do**
    Cyclically select the next edge $e \rightarrow b \in T$.
    Let $P_o = b \rightarrow \cdots \rightarrow e$ be the path obtained from T by removing the edge $e \rightarrow b$.
    Run $T' \leftarrow ImprovePath(P_o, 1,)$ (see below).
    if $w(T') < w(T)$
      Set $T = ImproveTour(T')$.
      Reset the number of idle iterations $i \leftarrow 0$.
    else
      Increase the number of idle iterations $i \leftarrow i + 1$.

Figure 2.11 Pseudo-code général de l'algorithme de Lin-Kernighan [KG10a].

Procedure. *ImprovePath*$(P, depth, R)$

Require: The path $P = b \rightarrow \cdots \rightarrow e$, recursion depth *depth* and the set of restricted vertices R.
  if *depth* α **then**
    Find the edge $x \rightarrow y \in P, x \neq b, x \notin R$ such that it maximizes the path gain $Gain(P, x \rightarrow y)$.
  else
    Repeat the rest of the procedure for every edge $x \rightarrow y \in P$, $x \neq b$, $x \notin R$.

  Conduct the local search move:
  $P \leftarrow RearrangePath(P, x \rightarrow y)$.
  if *GainIsAcceptable*$(P, x \rightarrow y)$ **then**
    Replace the edge $x \rightarrow y$ with $x \rightarrow e$ in P.
    $T' = CloseUp(P)$.
    if $w(T')$ $w(T)$ **then**
      Run $T' \leftarrow ImprovePath(P, depth + 1, R \cup \{x\})$.
    if $w(T') < w(T)$ **then**
      return T'.
    else
      Restore the path P.
  return T.

Figure 2.12 Fonction *ImprovePath* de l'algorithme de Lin-Kernighan [KG10a].

- *Gain*$(P, x \to y)$ is intended to calculate the gain of breaking a path P at an edge $x \to y$.
- *RearrangePath*$(P, x \to y)$ removes an edge $x \to y$ from a path P and adds the edge $x \to e$, where $P = b \to \cdots \to x \to y \to \cdots \to e$, see Fig. 1. Together with *CloseUp*, it includes an implementation of *QuickImprove*(T) (see Section 3.1), so *RearrangePath* may also apply some cluster optimization.
- *GainIsAcceptable*$(P, x \to y)$ determines if the gain of breaking a path P at an edge $x \to y$ is worth any further effort.
- *CloseUp*(P) adds an edge to a path P to produce a feasible tour. Together with *RearrangePath*, it includes an implementation of *QuickImprove*(T) (see Section 3.1), so *CloseUp* may also apply some cluster optimization.
- *ImproveTour*(T) is a tour improvement function. It is an analogue to *SlowImprove*(T) (see Section 3.1).

Figure 2.13 Autres fonctions nécessaires à l'algorithme de Lin-Kernighan [KG10a].

Les résultats de cet algorithme sont très prometteurs en optimalité, car, en appliquant toutes les étapes d'amélioration de solution, on est presque à 0% de taux d'erreur par rapport à la solution optimale, pour des échantillons de petites et moyennes tailles. Par contre, les temps d'exécution sont peu compétitifs, avec des temps moyens de 11 à 32 cycles d'horloge de processeur pour les petits échantillons, et environ 4310 cycles d'horloge de processeur pour les gros échantillons (plus de 800 lieux). [KG10a]

2.2.7 Heuristique génétique

Silberholz et Golden [SG07] expliquent dans leur article de 2007 le fonctionnement général d'une heuristique génétique : on part du principe que les tours dans un GTSP sont des individus, et leur composition en nœuds imite celle des chromosomes. Ainsi, à la première génération/itération, on tire un certain nombre d'individus aux chromosomes aléatoires, c'est-

à-dire qu'on créé un échantillon de tours de composition de nœuds aléatoires, toujours en prenant soin que chaque tour visite chaque cluster une seule fois.

Parmi cet échantillon, on sélectionne les individus qui sont les plus aptes à survivre, c'est-à-dire les tours de plus courte longueur, et on les « croise », pour obtenir leurs enfants. En matière de tours, cela signifie que l'on choisit un ou plusieurs points de croisements, et qu'on mixe le tour, en prenant soin de ne pas utiliser de doublons, ou plusieurs nœuds du même cluster, et de réinsérer les clusters oubliés. Par exemple, si l'on possède un tour (*A*1 *D*5 *E*3 *B*6 *C*2) et un tour (*B*3 *D*1 *C*5 *A*7 *E*1), où *A* est le nom du cluster et 1 est le numéro du nœud, on peut fixer un point de croisement : (*A*1 *D*5|*E*3 *B*6 *C*2) et (*B*3 *D*1|*C*5 *A*7 *E*1). On obtiendra donc deux enfants (*A*1 *D*5| *C*5 *X E*1) et (*B*3 *D*1| *E*3 *Y C*2), *X* et *Y* représentant les doublons *A*7 et *B*6. On inverse donc *X* et *Y*, et on obtient deux enfants (*A*1 *D*5| *C*5 *B*6 *E*1) et (*B*3 *D*1| *E*3 *A*7 *C*2). Bien entendu, il existe plusieurs méthodes de « crossover » qui permettent de mélanger le matériel génétique de deux parents. Celle-ci est la plus utilisée dans les différents papiers, et est très bien expliquée dans l'article de Matei et Pop [MP10].

De façon à explorer le plus de combinaisons possibles, on peut aussi considérer tous les enfants dont la deuxième partie du tour est une permutation des deux que l'on vient de trouver (par exemple, pour *CBE*, on a aussi *BEC*, *EBC*, *CEB*, *BCE*, et *ECB*). Silberholz et al [SG07] nomment cette méthode rOX, ou *rotational ordered crossover*.

Ils ajoutent à leur méthode rOX une rotation supplémentaire, qui requiert de remplacer les nœuds du premier et du dernier cluster de la fin du tour du deuxième parent, par les deux nœuds qui minimisent la longueur du tour. Ils nomment cette méthode améliorée mrOX, ou *modified rOX*.

A chaque itération, on analyse les enfants comme les parents, et si les parents font partie des trajets les plus avantageux, on les garde parmi les enfants, et on supprime toujours les individus aux trajets les plus longs.

Afin d'améliorer chaque génération, les auteurs ont ajouté à leur algorithme la méthode d'optimisation locale 2-opt, ainsi qu'un « Swap ». Afin d'améliorer la diversité de la population, et l'empêcher de converger vers un minimum local, ils ont aussi ajouté un facteur de mutation de 5% de probabilité pour chaque nœud de chaque individu, qui, concrètement, intervertit deux nœuds.

La performance de cet algorithme est présentée sur le Tableau 2.1, qui démontre son efficacité par rapport à plusieurs algorithmes compétiteurs. On voit en jaune son taux d'erreur et son temps moyen ; son taux d'erreur est le plus bas parmi les algorithmes comparés ; son temps moyen est très bon, mais pas le plus bas : l'algorithme de Snyder et Daskin [SD04] est meilleur, et son taux d'erreur est très bon aussi.

Tableau 2.1 Performances de l'algorithme génétique mrOX selon [SG07], comparées à celles d'autres algorithmes GTSP compétiteurs.

Dataset Name	mrOX GA		S+D GA		GI³		NN		FST-Lagr		FST-Root		B&C
	Pct.	Time	Pct.	Time	Pct.	Time	Pct.	Time	Pct.	Time	Pct.	Time	Time
10ATT48	0.00	0.36	0.00	0.18	*		*		0.00	0.90	0.00	2.10	2.10
10GR48	0.00	0.32	0.00	0.08	*		*		0.00	0.50	0.00	1.90	1.90
10HK48	0.00	0.31	0.00	0.08	*		*		0.00	1.10	0.00	3.80	3.80
11EIL51	0.00	0.26	0.00	0.08	*		*		0.00	0.40	0.00	2.90	2.90
12BRAZIL58	0.00	0.78	0.00	0.10	*		*		0.00	1.40	0.00	3.00	3.00
14ST70	0.00	0.35	0.00	0.07	0.00	1.70	0.00	0.80	0.00	1.20	0.00	7.30	7.30
16EIL76	0.00	0.37	0.00	0.11	0.00	2.20	0.00	1.10	0.00	1.40	0.00	9.40	9.40
16PR76	0.00	0.45	0.00	0.16	0.00	2.50	0.00	1.90	0.00	0.60	0.00	12.90	12.90
20RAT99	0.00	0.50	0.00	0.24	0.00	5.00	0.00	7.30	0.00	3.10	0.00	51.40	51.50
20KROA100	0.00	0.63	0.00	0.25	0.00	6.80	0.00	3.80	0.00	2.40	0.00	18.30	18.40
20KROB100	0.00	0.60	0.00	0.22	0.00	6.40	0.00	2.40	0.00	3.10	0.00	22.10	22.20
20KROC100	0.00	0.62	0.00	0.23	0.00	6.50	0.00	6.30	0.00	2.20	0.00	14.30	14.40
20KROD100	0.00	0.67	0.00	0.43	0.00	8.60	0.00	5.60	0.00	2.50	0.00	14.20	14.30
20KROE100	0.00	0.58	0.00	0.15	0.00	6.70	0.00	2.80	0.00	0.90	0.00	12.90	13.00
20RD100	0.00	0.51	0.00	0.29	0.08	7.30	0.08	8.30	0.08	2.60	0.00	16.50	16.60
21EIL101	0.00	0.48	0.00	0.18	0.40	5.20	0.40	3.00	0.00	1.70	0.00	25.50	25.60
21LIN105	0.00	0.60	0.00	0.33	0.00	14.40	0.00	3.70	0.00	2.00	0.00	16.20	16.40
22PR107	0.00	0.53	0.00	0.20	0.00	8.70	0.00	5.20	0.00	2.10	0.00	7.30	7.40
24GR120	0.00	0.66	0.00	0.32	*		*		1.99	4.90	0.00	41.80	41.90
25PR124	0.00	0.68	0.00	0.26	0.43	12.20	0.00	12.00	0.00	3.70	0.00	25.70	25.90
26BIER127	0.00	0.78	0.00	0.28	5.55	36.10	9.68	7.80	0.00	11.20	0.00	23.30	23.60
28PR136	0.00	0.79	0.16	0.36	1.28	12.50	5.54	9.60	0.82	7.20	0.00	42.80	43.00
29PR144	0.00	1.00	0.00	0.44	0.00	16.30	0.00	11.80	0.00	2.30	0.00	8.00	8.20
30KROA150	0.00	0.98	0.00	0.32	0.00	17.80	0.00	22.90	0.00	7.60	0.00	100.00	100.30
30KROB150	0.00	0.98	0.00	0.71	0.00	14.20	0.00	20.10	0.00	9.90	0.00	60.30	60.60
31PR152	0.00	0.97	0.00	0.38	0.47	17.60	1.80	10.30	0.00	9.60	0.00	51.40	94.80
32U159	0.00	0.98	0.00	0.55	2.60	18.50	2.79	26.50	0.00	10.90	0.00	139.60	146.40
39RAT195	0.00	1.37	0.00	1.33	0.00	37.20	1.29	86.00	1.87	8.20	0.00	245.50	245.90
40D198	0.00	1.63	0.07	1.47	0.60	60.40	0.60	118.80	0.48	12.00	0.00	762.50	763.10
40KROA200	0.00	1.66	0.00	0.95	0.00	29.70	5.25	53.00	0.00	15.30	0.00	183.30	187.40
40KROB200	0.05	1.63	0.01	1.29	0.00	35.80	0.00	135.20	0.05	19.10	0.00	268.00	268.50
45TS225	0.14	1.71	0.28	1.09	0.61	89.00	0.00	117.80	0.09	19.40	0.09	1298.40	37875.90
46PR226	0.00	1.54	0.00	1.09	0.00	25.50	2.17	67.60	0.00	14.60	0.00	106.20	106.90
53GIL262	0.45	3.64	0.55	3.05	5.03	115.40	1.88	122.70	3.75	15.80	0.89	1443.50	6624.10
53PR264	0.00	2.36	0.09	2.72	0.36	64.40	5.73	147.20	0.33	24.30	0.00	336.00	337.00
60PR299	0.05	4.59	0.16	4.08	2.23	90.30	2.01	281.80	0.00	33.20	0.00	811.40	812.80
64LIN318	0.00	8.08	0.54	5.39	4.59	206.80	4.92	317.00	0.36	52.50	0.36	847.80	1671.90
80RD400	0.58	14.58	0.72	10.27	1.23	403.50	3.98	1137.10	3.16	59.80	2.97	5031.50	7021.40
84FL417	0.04	8.15	0.06	6.18	0.48	427.10	1.07	1341.00	0.13	77.20	0.00	16714.40	16719.40
88PR439	0.00	19.06	0.83	15.09	3.52	611.00	4.02	1238.90	1.42	146.60	0.00	5418.90	5422.80
89PCB442	0.01	23.43	1.23	11.74	5.91	567.70	6.22	838.40	4.22	78.80	0.29	5353.90	58770.50
Averages	0.03	2.69	0.11	1.77	0.98	83.09	1.48	171.56	0.46	16.44	0.11	964.79	3356.47
#Trials	5		5		1		1		1		1		1
Platform	Dell Dimension 8400				Sun Spare Station LX				HP 9000 / 720				

Daskin et al. [SD04] qui se voient comparés sur le Tableau 2.1, ont créé un algorithme génétique eux aussi, à la différence qu'ils n'utilisent pas les mutations pour garder une certaine diversité dans leur population. A cette fin de diversité, ils introduisent à chaque génération des nouveaux individus complètement aléatoires, avec comme condition que leurs trajets n'existent pas déjà dans la génération où ils sont introduits. De plus, ils imposent que chaque génération soit composée de 20% des parents, 70% d'enfants, et de 10% de ces nouveaux individus composés de nœuds aléatoires.

Une recherche similaire à celle de Silberholz et al. [SG07] a été effectuée par Tasgetiren et al., également en 2007 [TSPL07]. Les résultats des deux études sont très similaires.

2.2.7.1 Heuristique mémétique

Gutin et Karapetyan [GK10] expliquent dans leur article de 2010 le concept d'un algorithme mémétique : il s'agit d'un algorithme génétique associé à un algorithme de recherche locale puissant.

L'algorithme mémétique fonctionne comme suit :

- On initialise l'échantillon avec des individus de composition aléatoire.
- On optimise ce premier échantillon avec une procédure de recherche locale qui permet de supprimer les individus les moins intéressants et de conserver les meilleurs individus, puis on supprime les doublons.
- On crée la prochaine génération, en croisant les chromosomes (échange de moitiés de parcours), en faisant muter certains individus avec une certaine probabilité, et en prenant les meilleurs éléments de la génération précédente.
- On optimise la nouvelle génération avec la recherche locale (on laisse les individus déjà optimisés de la génération antérieure de côté), et on supprime les doublons.
- On répète ces deux dernières étapes jusqu'à atteindre la condition de fin.

La particularité de l'algorithme de Gutin et Karapetyan [GK10] est qu'il optimise la recherche locale afin de la rendre extrêmement efficace, car il s'agit de la partie la plus onéreuse en temps. Une recherche locale efficace donne donc un net avantage à cet algorithme, par rapport aux autres algorithmes mémétiques qui n'optimisent pas la recherche locale, et seront donc plus lents.

Cependant, ce genre d'algorithme est spécifique à des instances avec un grand échantillon (de 40 à 217 clusters), contrairement aux algorithmes génétiques simples, qui sont ciblés pour les échantillons de petite à moyenne taille. Bien qu'ils soient faits pour trouver la solution optimale pour des échantillons réduits aussi, ils ont un temps d'exécution que les auteurs avouent peu compétitif pour les situations comme celle de notre problème, et ne seront donc pas considérés pour l'implémentation.

Bontoux et al. [BAF10] ont effectué une recherche sur un thème très similaire (algorithme mémétique avec une opération de crossover large sur le voisinage), avec des résultats très semblables à ceux de Gutin et Karapetyan [GK10].

2.2.8 Comparaison

Afin de choisir l'algorithme qui est le plus adéquat à nos besoins, il fallait établir quels sont ceux qui ont un temps de calcul minimal avec taux d'erreur le plus proche de zéro, pour un échantillon de petite taille : 1 à ~450 franchises divisées en 1 à 23 clusters (nous sommes limités par Google Maps™, qui bornent les tours à 23 lieux), plus le cluster de départ contenant une seule adresse, qui est l'adresse du client. En effet, le client peut vouloir visiter un seul magasin, tout comme il peut vouloir faire le tour de tous les différents magasins d'alimentation existants sur l'île de Montréal.

Un prétraitement des données n'est pas nécessaire dans notre cas, car la suppression de franchises trop éloignées nous ferait perdre plus de temps que nous donnerait d'avantages. En effet, le prétraitement des données est un algorithme couteux en temps, ce qui n'est pas à

notre avantage pour notre site Web. De plus, ce prétraitement, pour des échantillons de petite à moyenne taille comme le nôtre, a tendance à faire perdre en optimalité.

La plupart des articles que nous avons considérés dans la section précédente comparent leurs résultats aux autres algorithmes à l'aide d'un ensemble de données de test pour le problème du Voyageur de Commerce : la librairie TSPLIB de Reinelt (1991), qui propose des situations de 48 à 442 nœuds, partitionnés en clusters pour reproduire un GTSP. Cette quantité de nœuds est parfaite pour nous, car un algorithme capable de correctement gérer cette quantité de nœuds serait parfaitement approprié à nos besoins. Les tableaux de 2.1 (vu dans une section précédente), 2.2, et 2.3 présentent les résultats détaillés de ces différentes comparaisons de performances.

Tableau 2.2 Performances de l'algorithme génétique de [TSPL07] en comparaison aux algorithmes compétiteurs.

	GA		RKGA		GI[3]		NN		FST-Lagr		FST-Root		B&C
Problem	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}	Δ	t	Δ	t	Δ	t	Δ	t	t
11EIL51	0.00	0.01	0.00	0.20	0.00	0.30	0.00	0.40	0.00	0.40	0.00	2.90	2.90
14ST70	0.00	0.03	0.00	0.20	0.00	1.70	0.00	0.80	0.00	1.20	0.00	7.30	7.30
16EIL76	0.00	0.03	0.00	0.20	0.00	2.20	0.00	1.10	0.00	1.40	0.00	9.40	9.40
16PR76	0.00	0.05	0.00	0.20	0.00	2.50	0.00	1.90	0.00	0.60	0.00	12.9	12.90
20KROA100	0.00	0.08	0.00	0.40	0.00	6.80	0.00	3.80	0.00	2.40	0.00	18.30	18.40
20KROB100	0.00	0.08	0.00	0.40	0.00	6.4	0.00	2.40	0.00	3.10	0.00	22.10	22.20
20KROC100	0.00	0.06	0.00	0.30	0.00	6.50	0.00	6.30	0.00	2.20	0.00	14.30	14.40
20KROD100	0.00	0.06	0.00	0.40	0.00	8.60	0.00	5.60	0.00	2.50	0.00	14.20	14.30
20KROE100	0.00	0.07	0.00	0.60	0.00	6.70	0.00	2.80	0.00	0.90	0.00	12.90	13.00
20RAT99	0.00	0.06	0.00	0.50	0.00	5.00	0.00	7.30	0.00	3.10	0.00	51.4	51.5
20RD100	0.00	0.09	0.00	0.50	0.08	7.30	0.08	8.30	0.08	2.60	0.00	16.5	16.6
21EIL101	0.00	0.08	0.00	0.40	0.40	5.20	0.40	3.00	0.00	1.70	0.00	25.50	25.60
21LIN105	0.00	0.08	0.00	0.50	0.00	14.40	0.00	3.70	0.00	2.00	0.00	16.20	16.40
22PR107	0.00	0.08	0.00	0.40	0.00	8.70	0.00	5.20	0.00	2.10	0.00	7.30	7.40
25PR124	0.00	0.14	0.00	0.80	0.43	12.20	0.00	12.00	0.00	3.70	0.00	25.70	25.90
26BIER127	0.00	0.25	0.00	0.40	5.55	36.10	9.68	7.80	0.00	11.20,	0.00	23.30	23.60
28PR136	0.00	0.31	0.00	0.50	1.28	12.5	5.54	9.60	0.82	7.20	0.00	42.80	43.00
29PR144	0.00	0.24	0.00	1.00	0.00	16.30	0.00	11.8	0.00	2.30	0.00	8.00	8.20
30KROA150	0.00	0.17	0.00	0.70	0.00	17.80	0.00	22.90	0.00	7.60	0.00	100.00	100.30
30KROB150	0.00	0.37	0.00	0.90	0.00	14.20	0.00	20.10	0.00	9.90	0.00	60.30	60.60
31PR152	0.00	0.73	0.00	1.20	0.47	17.60	1.80	10.30	0.00	9.60	0.00	51.40	94.80
32U159	0.00	0.32	0.00	0.80	2.60	18.50	2.79	26.50	0.00	10.90	0.00	139.60	146.40
39RAT195	0.00	1.59	0.00	1.00	0.00	37.2	1.29	86.00	1.87	8.20	0.00	245.50	245.90
40D198	0.00	1.07	0.00	1.60	0.60	60.40	0.60	118.80	0.48	12.00	0.00	762.50	763.10
40KROA200	0.00	0.49	0.00	1.80	0.00	29.70	5.25	53.00	0.00	15.30	0.00	183.30	187.40
40KROB200	0.00	2.06	0.00	1.90	0.00	35.80	0.00	135.20	0.05	19.10	0.00	268.00	268.50
45TS225	0.02	1.8	0.02	2.10	0.61	89.00	0.00	117.80	0.09	19.40	0.09	1298.40	37875.90
46PR226	0.00	0.44	0.00	1.50	0.00	25.50	2.17	67.60	0.00	14.60	0.00	106.20	106.90
53GIL262	0.14	4.96	0.75	1.90	5.03	115.40	1.88	122.7	3.75	15.80	0.89	1443.50	6624.10
53PR264	0.00	1.52	0.00	2.10	0.36	64.40	5.73	147.20	0.33	24.30	0.00	336.00	337.00
60PR299	0.04	6.6	0.11	3.20	2.23	90.30	2.01	281.30	0.00	33.20	0.00	811.40	812.30
64LIN318	0.00	5.74	0.62	3.50	4.59	206.80	4.92	317.00	0.36	52.50	0.36	847.80	1671.90
80RD400	0.08	12.37	1.19	5.90	1.23	403.50	3.98	1137.10	3.16	59.80	2.97	5031.50	7021.40
84FL417	0.01	14.13	0.05	5.30	0.48	427.10	1.07	1341.00	0.13	77.20	0.00	16714.40	16719.40
88PR439	0.13	14.48	0.27	9.50	3.52	611.00	4.02	1233.90	1.42	146.6	0.00	5418.90	5422.80
89PCB442	0.10	21.19	1.70	9.00	5.91	567.70	0.22	838.40	4.22	78.80	0.29	5333.90	58770.30
Overall Avg	0.01	2.55	0.13	1.72	0.98	83.09	1.48	171.56	0.47	18.48	0.13	1097.32	3821.19

Tableau 2.3 Performances de l'algorithme génétique de [BAF10] en comparaison aux algorithmes compétiteurs.

Instance	LNS–1 trial		LNS–5 trials		mrOX		Snyder		GI	
	Value	CPU	Gap	CPU	Gap	CPU	Gap	CPU	Gap	CPU
10att48	5394	0.57	0	0.76	0	0.36	0	0.18	•	•
10gr48	1834	0.97	0	0.79	0	0.32	0	0.08	•	•
10hk48	6386	0.58	0	0.5	0	0.31	0	0.08	•	•
11eil51	174	0.84	0	0.81	0	0.26	0	0.08	0	0.3
12brazil58	15 332	0.85	0	0.65	0	0.78	0	0.1	•	•
14st70	316	1.02	0	0.93	0	0.35	0	0.07	0	1.7
16eil76	209	1.18	0	1	0	0.37	0	0.11	0	2.2
16pr76	64 925	1.27	0	1.17	0	0.45	0	0.16	0	2.5
20kroA100	9711	1.98	0	1.81	0	0.5	0	0.24	0	5
20kroB100	10 328	2.01	0	2.17	0	0.63	0	0.25	0	6.8
20kroC100	9554	1.84	0	1.85	0	0.6	0	0.22	0	6.4
20kroD100	9450	2.93	0	2.77	0	0.62	0	0.23	0	6.5
20kroE100	9523	1.87	0	1.81	0	0.67	0	0.43	0	8.6
20rat99	497	3.52	0	3.89	0	0.58	0	0.15	0	6.7
20rd100	3650	2.93	0	2.91	0	0.51	0	0.29	0.08	7.3
21eil101	249	2.07	0	2.09	0	0.48	0	0.18	0.4	5.2
21lin105	8213	3.25	0	3.18	0	0.6	0	0.33	0	14.4
22pr107	27 898	4.67	0	4.78	0	0.53	0	0.2	0	8.7
24gr120	2769	2.30	0	2.34	0	0.66	0	0.32	•	•
25pr124	36 605	2.89	0	2.84	0	0.68	0	0.26	0.43	12.2
26bier127	72 418	3.02	0	3.35	0	0.78	0	0.28	5.55	36.1
28pr136	42 570	4.17	0	4.23	0	0.79	0.16	0.36	1.28	12.5
29pr144	45 886	5.38	0	5.42	0	1	0	0.44	0	16.3
30kroA150	11 018	5.27	0	5.95	0	0.98	0	0.32	0	17.8
30kroB150	12 196	4.61	0	5.02	0	0.98	0	0.71	0	14.2
31pr152	51 576	4.45	0	5.24	0	0.97	0	0.38	0.47	17.6
32u159	22 664	5.53	0	5.58	0	0.98	0	0.55	2.6	18.5
39rat195	854	10.42	0	11.01	0	1.37	0	1.33	0	37.2
40d198	10 557	8.72	0	10.15	0	1.63	0.07	1.47	0.6	60.4
40kroA200	13 406	6.74	0	10.41	0	1.66	0	0.95	0	29.7
40kroB200	13 111	8.78	0	10.81	0.05	1.63	0.01	1.29	0	35.8
45ts225	68 340	35.31	0.04	31.45	0.14	1.71	0.28	1.09	0.61	89
46pr226	64 007	6.92	0	8.25	0	1.54	0	1.09	0	25.5
53gil262	1013	25.12	0.14	24.34	0.45	3.64	0.55	3.05	5.03	115.4
53pr264	29 549	16.64	0	18.27	0	2.36	0.09	2.72	0.36	64.4
60pr299	22 615	20.19	0	21.25	0.05	4.59	0.16	4.08	2.23	90.3
64lin318	20 765	24.89	0	26.33	0	8.08	0.54	5.39	4.59	206.8
80rd400	6361	38.33	0.42	32.21	0.58	14.58	0.72	10.27	1.23	403.5
84fl417	9651	21.9	0	31.63	0.04	8.15	0.06	6.18	0.48	427.1
88pr439	60 099	56.46	0	42.55	0	19.06	0.83	15.09	3.52	611
89pcb442	21 673	76.64	0.19	62.53	0.01	23.43	1.23	11.74	5.91	567.7
Averages	0.001	9.12	0.02	10.12	0.03	2.69	0.11	1.77	0.98	83.09
Trials	1		5		5		5		1	

On peut voir sur le tableau 2.1 que les deux meilleurs algorithmes du tableau sont ceux de Silberholz et al. [SG07] avec leur mrOX, ainsi que Snyder et Daskin [SD04] avec leur RKGA (Random Key Genetic Algorithm), avec soit le taux d'erreur le plus bas du tableau et un temps très bon, ou le meilleur temps et un taux d'erreur très bon. Sur le tableau 2.2, on observe que l'algorithme de Tasgetiren et al. [TSPL07] est lui aussi très bon, avec le taux d'erreur le plus bas du tableau, et un bon temps, avec l'algorithme de Snyder et Daskin

[SD04] de nouveau meilleur au niveau du temps. Pour finir, on peut voir sur le tableau 2.3 que, bien que l'algorithme mémétique de Bontoux et al. [BAF10] est excellent au niveau du taux d'erreur, son temps d'exécution est bien trop élevé pour nos besoins et notre taille d'échantillon. Il ne sera donc pas considéré pour l'implémentation.

En observant les tables de comparaison d'efficacité d'algorithmes pour GTSP (tableau 2.1, 2.2, 2.3) on remarque que les algorithmes génétiques ont les temps les plus avantageux: ils sont toujours à moins de 3 cycles d'horloge interne du processeur en temps d'exécution.

Tableau 2.4 Comparaison des performances des trois meilleurs algorithmes génétiques avec les algorithmes Branch&Cut et GI^3, selon [SG07, TSPL07 et BAF10].

	Branch&cut	GI^3	GA SD	GA mrOX	GA Tasgetiren
%	0	0.98	0.11	0.03	0.01
time	3356.47	83.09	1.77	2.69	2.55

On observe sur le tableau 2.4 que l'algorithme génétique de Tasgetiren et al. [TSPL07] (GA Tasgetiren) propose un temps plus court et une meilleure optimalité que celui de Silberholz et Golden [SG07] (GA mrOX). On remarque néanmoins que l'algorithme de Snyder et Daskin [SD04] (RKGA, ou GA SD) propose un temps encore plus avantageux, mais avec une moins bonne optimalité. Les deux algorithmes en compétition sont donc celui de Snyder et Daskin [SD04] avec un taux d'erreur moyen par rapport à la solution optimale de 0.11% et un temps moyen de 1.77 (cycles d'horloge interne du processeur), et celui de Tasgetiren et al. [TSPL07] avec un taux d'erreur de 0.01% et un temps de 2.55. La solution de Tasgetiren et al. [TSPL07] est donc 11 fois plus proche de la solution optimale, tout en étant 1,44 fois plus lent. Comme un cycle de processeur est estimé correspondre à 1 à 10 millisecondes, une différence de 0.78 ms à 7.8 ms est considérée minimale lors de l'affichage d'une page Web. On met donc l'accent sur l'optimalité, qui est assurée par l'algorithme de Tasgetiren et al. [TSPL07].

Nous avons donc implémenté une version adaptée de l'algorithme de Tasgetiren et al. [TSPL07] pour le site SmartShopping, et nous montrerons dans les deux chapitres suivants

son fonctionnement pratique, en la comparant avec la version standard déjà implémentée auparavant sur le site, qui prenait en compte les magasins les plus proches de l'adresse du client. Nous mettrons cette comparaison en évidence dans le cadre de notre projet SmartShopping.

CHAPITRE III

IMPLÉMENTATION

Ce chapitre présente les principales étapes d'implémentation de l'algorithme de Voyageur de Commerce Généralisé décrit dans [TSPL07].

3.1 Etapes de l'algorithme de Voyageur de Commerce Généralisé de [TSPL07]

L'algorithme de Tasgetiren et al. [TSPL07] que nous avons implémenté se déroule ainsi :

1. Une population de base de taille N est créée de façon aléatoire.
2. Chaque individu de cette population se voit appliquer deux algorithmes de recherche locale, afin d'améliorer leur qualité.
    a. Le premier algorithme appliqué est le 2-opt, qui consiste à ôter deux arêtes du tour et relier les nœuds différemment.
    b. Le deuxième algorithme est un « Iterated Local Search », ou ILS, qui consiste premièrement à chercher, pour chaque cluster, le nœud qui optimise la longueur du tour, remplacer l'ancien nœud par ce nouveau nœud, et recommencer la procédure avec ce nouveau tour.
3. Une nouvelle génération de taille M (les enfants) est créée en croisant les parents deux par deux (le premier, le meilleur parcours entre deux individus choisis au hasard, le deuxième aléatoire) avec un opérateur de croisement appelé le « 2-cut PTL Crossover » [PTL08] qui consiste à couper le parcours d'un individu en trois parties avec deux points de coupe aléatoires équilibrés, et utiliser le segment du centre comme début de parcours du 1er enfant, et comme fin de parcours du 2ème enfant, avec le reste de la séquence provenant du 2ème parent, dans l'ordre des nœuds manquants. Cette méthode permet de

pouvoir créer deux enfants de trajets différents, même avec deux parents identiques, et donc de trajets identiques ; souvent, les autres méthodes de crossover génèreraient deux fois le même enfant, si les parents sont les mêmes (voir tableau 3.1).

Tableau 3.1 Exemple de crossover PTL [TSPL07].

TWO-CUT PTL CROSSOVER OPERATOR.

	j	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*1*
P_1	n_j	10	5	7	2	**6**	**4**	**11**	9	8	1	3	10
	π_j	1	51	22	20	**50**	**33**	**10**	44	25	41	24	1
P_2	n_j	10	5	7	2	6	4	11	9	8	1	3	10
	π_j	1	51	22	20	50	33	10	44	25	41	24	1
	j	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*1*
O_1	n_j	**6**	**4**	**11**	10	5	7	2	9	8	1	3	6
	π_j	**50**	**33**	**10**	1	51	22	20	44	25	41	24	50
O_2	n_j	10	5	7	2	9	8	1	3	**6**	**4**	**11**	10
	π_j	1	51	22	20	44	25	41	24	**50**	**33**	**10**	1

4. La nouvelle génération subit une mutation avec une probabilité de 5%, en prenant un cluster aléatoire, et en remplaçant le nœud visité de ce cluster par un autre nœud, choisi aléatoirement.
5. Dans le pool de M nouveaux individus et N anciens individus, un « tournoi de sélection de taille 2 » est appliqué, c'est-à-dire que deux individus sont choisis au hasard, celui de plus court chemin est conservé, et l'autre est remis dans le pool, jusqu'à avoir N individus sauvegardés, qui composent la nouvelle génération.
6. Les points 2 à 6 sont répétés, jusqu'à atteindre une condition d'arrêt, ou obtenir les mêmes meilleurs individus pendant un nombre de générations donné.

Pour l'implémentation de notre programme, nous avons choisi le langage C++, qui est reconnu pour produire des exécutables très rapides. Nous avons choisi comme plateforme de développement la plateforme Eclipse Indigo 3.7, additionnée de CDT 8.0.

3.2 Détails de l'implémentation de l'algorithme de Voyageur de Commerce Généralisé

Nous commençons par définir une variable globale, *POPSIZE*, qui représente la taille maximale possible de notre population, parents et enfants compris, avant de leur appliquer une « sélection naturelle ». Nous définissons aussi la taille de la population parentale *PopSize*, qui représente les meilleurs individus sélectionnés pour être parents, et qui seront gardés à la prochaine génération.

```
#define POPSIZE  200
#define PopSize  100
```

Nous définissons aussi en variables le nombre de générations maximal que nous allons générer avant d'arrêter le programme, ainsi que la probabilité de mutation qui est de 5% pour chaque nouvel enfant.

```
int MaxGeneration=800;
double ProbMut=0.05;
```

Afin de stocker les informations relatives à chaque individu, nous créons une structure nommée *individual*, qui est composée d'un tableau d'entiers, afin de sauvegarder l'ordre dans lequel l'individu visite les nœuds, et un **double** qui contient la distance totale de son tour. Notons que la taille maximale de notre tour, ou chromosome, est fixée à 24, car l'API Google Maps™ impose une limite de 23 arrêts en plus de l'adresse de départ.

```
struct individual
{
        int chrom[24];
        double totaldist;

};
```

Nous déterminons deux individus clefs, qui seront eux aussi globaux, afin d'en faciliter la modification : le meilleur individu à un temps donné *bestindividual*, et le meilleur individu jusqu'au tour précédent, *currentbest*. Nous générons aussi un tableau d'individus *population* de taille *POPSIZE*, afin de pouvoir contenir enfants et parents au moment où ils sont le plus nombreux.

```
struct individual bestindividual;
struct individual currentbest;
struct individual population[POPSIZE];
```

La première étape est la création de la population initiale. On prend en paramètre la matrice qui donne à chaque lieu le numéro du cluster auquel il appartient, afin de s'assurer d'avoir exactement un lieu par cluster dans chacun de nos tours.

```
void CreateFirstIndividuals( int Clusters[])
```

On boucle jusqu'à avoir créé une population initiale de cent individus, soit une population de taille *PopSize*. On suppose que le premier nœud à visiter est le nœud 1 du cluster 1, soit l'adresse de départ du client, qui est la seule dans son cluster de taille 1. Ensuite, pour chaque nœud qui s'ensuit, on tire des lieux au hasard de manière qu'ils n'aient pas encore été tirés, et n'appartiennent pas à un cluster déjà présent dans le tour créé. Ainsi, on se trouve avec des tours aléatoires qui passent une seule fois dans chaque cluster.

```
for(i = start; i < stop; i++)
{
    population[i].chrom[0] = 1;
    for(j = 1; j < NBCLUST; j++){
        token = 0;
        while(token == 0)
        {
            token = 1;
            population[i].chrom[j] = random(CITYNB) + 1;
            for(k = 0; k < j; k++){
                if((population[i].chrom[j] ==
                population[i].chrom[k]) ||
                (Clusters[population[i].chrom[k] - 1] ==
                Clusters[population[i].chrom[j] - 1])){
                    token=0;
                }
            }
        }
    }
}
```

Une fois la population de base créée, on calcule la qualité de chacun des individus en déterminant leur longueur. A cette fin, nous avons besoin d'une matrice de distances lue au

préalable dans un fichier. L'obtention de cette dernière se fait au niveau de la fonction *main()*.

Ainsi, pour chaque individu de la population de taille *PopSize*, on cherche la distance entre chacun de ses nœuds, en n'oubliant pas la distance qui ramène du dernier nœud au premier nœud, soit le retour au point de départ. On stocke ensuite cette information dans la variable *totaldist*, qui est un paramètre de chaque individu.

```
void CalcTourLength(int Temporary[], double *Distance[])
{
      int i, j;
      double TotalDistance = 0;

      for(i = 0; i < PopSize; i++){
      for(int h = 0; h < NBCLUST; h++){
            Temporary[h] = population[i].chrom[h];
      }
      for(j = 1; j < NBCLUST; j++){
            TotalDistance =
                (TotalDistance +
                Distance[Temporary[j-1] - 1][Temporary[j] - 1]);
      }
      TotalDistance =
          TotalDistance + Distance[Temporary[NBCLUST-1] - 1][0];
      population[i].totaldist = TotalDistance;
      TotalDistance = 0;
      }
}
```

On applique sur cette première génération d'individus l'algorithme de recherche locale *2-opt*, qui consiste à tenter d'échanger toutes les arêtes deux par deux, et garder le tour résultant si sa taille est meilleure que celle de l'ancien tour.

2-opt prend en paramètres le début et la fin de l'échantillon de population à traiter, ainsi que la matrice des distances entre les lieux.

```
void TwoOpt(int debut,int fin, int Temporary[], double *Distance[])
```

Afin d' « échanger les deux arêtes», on parcourt tout le vecteur, on prend deux nœuds séparés d'un segment de toutes les tailles possibles, c'est-à-dire de 2 nœuds à *CHROMLENGTH*-2 nœuds, et on inverse simplement l'ordre des nœuds du segment du

centre. On effectue ce calcul sur le même tour jusqu'à ce qu'on n'observe plus de changement.

```
for(l = debut;l < fin; l++)
	{
		while(count == 1){
			count = 0;
			for(k = 1; k < CHROMLENGTH - 2; k++)
			{
				for(i = 1; i < CHROMLENGTH - k; i++)
				{
					for(j = k + 1; j < CHROMLENGTH; j++)
					{
						t = i;
						u = j;
						tempo[l] = population[l];
while(t < u)
{
	temp = tempo[l].chrom[t];
	tempo[l].chrom[t] = tempo[l].chrom[u];
	tempo[l].chrom[u] = temp;
	t++;
	u--;
}
```

Ensuite, on calcule la longueur totale de ce nouveau tour, et si elle est plus courte que celle de l'ancien tour, on remplace l'ancien tour par le nouveau, et on recommence le parcours des arêtes depuis le début. Sinon, on continue à parcourir les arêtes jusqu'à trouver une configuration qui réduit le plus la longueur du tour, jusqu'à avoir essayé toutes les combinaisons.

```
TotalDistance=0.0;
for(int h = 0; h < NBCLUST; h++)
{
      Temporary[h] = population[i].chrom[h];
}
for(p = 1; p < CITYNB; p++){
      TotalDistance=
               TotalDistance +
               Distance[Temporary[p - 1] - 1][Temporary[p] - 1];
}
TotalDistance = TotalDistance + Distance[Temporary[CITYNB - 1] - 1][0];
tempo[l].totaldist = TotalDistance;
if(tempo[l].totaldist < population[l].totaldist){
      population[l] = tempo[l];
      count = 1;
                  }
            }
      }
}
}
}
```

Après *2-opt*, on applique à la première génération d'individus un autre algorithme de recherche locale, *ILS,* soit *Iterated Local Search*. Il consiste en la recherche, pour un ordre de clusters donné, du meilleur nœud pour chaque cluster, dans le but de minimiser la longueur du tour.

Cet algorithme requiert la liste des clusters pour chaque nœud, sous la forme du tableau *Clusters*, ainsi que la matrice des distances.

On applique cet algorithme à toute la population des parents. Pour chaque cluster, on itère dans les nœuds, et s'ils font partie du même cluster, on compare la longueur du tour actuel à la longueur du tour avec le nouveau nœud. S'il est meilleur, on le garde, et on continue de le comparer aux autres nœuds, jusqu'à trouver le nœud pour chaque cluster qui minimise au mieux la longueur du tour.

```
void ILS(int Clusters[],  int Temporary[], double *Distance[])
{     int i;
int p;

for(i = 0; i < PopSize; i++){
      int w = 0;
      struct individual best;

      while(w < NBCLUST){
            w++;
            for(int k = 1; k < CITYNB; k++){

                  best = population[i];
                  if(Clusters[k] == Clusters[best.chrom[w] - 1]){

                        best.chrom[w] = k + 1;
                        double TotalDistance = 0.0;

                        for(int h = 0; h < NBCLUST; h++){
                              Temporary[h] = best.chrom[h];
                        }

                        for(p = 1; p < NBCLUST; p++){
                         TotalDistance =
                           TotalDistance +
                           Distance[Temporary[p-1] -1][Temporary[p] -1];
                        }
                        TotalDistance =
                           TotalDistance +
                           Distance[Temporary[NBCLUST-1] - 1][0];
                        best.totaldist = TotalDistance;

                        if(best.totaldist < population[i].totaldist){
                              population[i] = best;
                              w = 0;
                        }     }     }     }     }     }
```

Une fois que les algorithmes de recherche locale ont été appliqués à la population initiale, on cherche le meilleur individu afin de pouvoir l'afficher à l'utilisateur, ce qui nous permet d'avoir un contrôle de la convergence de l'algorithme vers la solution optimale.

On fixe comme point de départ pour la variable *bestindividual* le premier individu de la population, puis on parcourt toute la population afin de déterminer si un autre individu a un tour plus court, et s'il existe, il devient le *bestindividual*.

```
void FindBestIndividual(int Temporary[])
{
	int i;
	bestindividual = population[0];
	for(i = 1; i < PopSize ; i++){
		if(population[i].totaldist < bestindividual.totaldist){
			bestindividual = population[i];
		}
	}
```

Ensuite, si on se trouve à la première génération d'individus, on imprime notre résultat, sinon, pour toutes autres générations, on imprime notre résultat uniquement s'il dénote une amélioration dans le temps de parcours.

```
if(generation == 0){
		currentbest = bestindividual;
		PrintOutput(Temporary);
	}
	else
	{
		if(bestindividual.totaldist < currentbest.totaldist){
			currentbest = bestindividual;
			PrintOutput(Temporary);
		}

	}
}
```

En ce qui concerne notre fonction d'impression des données, elle imprime le numéro de génération, la longueur totale du meilleur individu, ainsi que sa composition en nœuds. Cette fonction est très utile lorsque le programme est appelé depuis un terminal, car elle permet d'observer la convergence rapide de l'algorithme vers la solution optimale. Cependant, dans le cadre de notre projet, elle n'est pas utilisée, car nous n'avons besoin comme résultat que du tour final optimal.

```
void PrintOutput()
{
printf("Generation=%d, Best Total
Distance=%f:\n",generation,currentbest.totaldist);
printf("Best individual:");
for(int j=0;j<NBCLUST;j++){printf("(%d)",currentbest.chrom[j]);}
printf("\n\n");
}
```

Une fois la première génération d'individus créée, améliorée, et le meilleur individu trouvé, on passe à la création de la nouvelle génération. A cette fin, nous avons besoin du tableau de clusters, qui nous permettra de vérifier qu'on ne visite toujours qu'un seul et unique nœud par cluster.

```
void CreateChildren(int Clusters[])
{
SelectOperator();
PTLCrossover(Clusters);
Mutation(Clusters);
}
```

Nous commençons donc par trier nos individus.

```
void SelectOperator(void)
{
      struct individual newpopulation[PopSize];
      int start, stop;
      start = 0;
      stop = (int)(PopSize);
      SelectBest(start, stop, newpopulation);

}
```

On parcourt la totalité de notre population (qui pour l'instant n'est composée que de parents, placés dans les index 0 à 100, ou *PopSize*, de notre vecteur d'individus), et on sélectionne *PopSize*=100 individus par tournoi de deux, c'est-à-dire qu'on choisit deux individus aléatoirement, et que celui de plus court chemin est gardé pour être placé dans la population adulte. Cette fonction est particulièrement utile aux générations suivantes, quand on possède aussi une population d'enfants, qui se trouve dans la portion *PopSize* à *POPSIZE* (index 100 à 200) de notre vecteur d'individus : ainsi, on place dans la portion de 0 à *PopSize* les 100 individus choisis par tournoi de deux, et le reste du vecteur peut être effacé lorsque la prochaine génération d'enfants est créée.

```
void SelectBest(int start, int stop, struct individual *newpopulation)
{
	int i, p1, p2;
	struct individual temp[PopSize];

	for (i = start; i < stop; i++){

		do{
			p1 = random(POPSIZE);
			p2 = random(POPSIZE);
		}while((p1 == p2) ||
			(population[p1].totaldist == 0) ||
			(population[p2].totaldist == 0));

		if(population[p1].totaldist < population[p2].totaldist){
			temp[i] = population[p1];
		}
		else{
			temp[i]=population[p2];
		}
	}
	for (i = start; i < stop; i++){
		newpopulation[i] = temp[i];
		population[i] = newpopulation[i];
	}
}
```

On passe ensuite au crossover, c'est-à-dire au mélange du patrimoine génétique, ou ordonnancement des nœuds, de deux parents, afin de créer une nouvelle génération.

```
void PTLCrossover(int Clusters[])
```

On vérifie en premier lieu que le nombre de clusters est supérieur à deux, sans quoi tout crossover serait inutile, car les enfants seraient parfaitement identiques aux parents.

```
if(NBCLUST>2){…}
```

Comme le demande le crossover PTL, on choisit un premier parent par tournoi de deux, ce qui signifie que l'on choisit deux adultes aléatoirement, et on garde celui de meilleure longueur. Le deuxième parent est choisi aléatoirement.

```
p1 = random(PopSize);
      do{
      p1bis = random(PopSize);
      p2 = random(PopSize);
      }while((p1 == p2) || (p1 == p1bis) || (p2 == p1bis));

      if (population[p1].totaldist > population[p1bis].totaldist){
                        p1=p1bis;
                  }
```

On effectue ensuite sur le premier parent deux coupures dans son tour. Le segment du milieu est placé en début du tour du premier enfant, et en fin de tour du deuxième enfant. On fait toujours attention de garder l'unique nœud 1 du cluster 1 en première position, car il s'agit de notre point de départ.

```
do{
      cut1 = random(NBCLUST);
      cut2 = random(NBCLUST);
}while((cut2 <= cut1) || (cut1 == 0));
      int g = 0;
      population[i].chrom[0] = 1;
      population[i+1].chrom[0] = 1;
      for (int k = 0; k < cut2 - cut1; k++){
      if(population[p1].chrom[cut1 + k] == 1){
            g++;
      }
      population[i].chrom[k + 1] = population[p1].chrom[cut1 + k + g];
      population[i + 1].chrom[NBCLUST - (cut2 - cut1) + k] =
               population[p1].chrom[cut1 + k + g];
}
```

On remplit par la suite le reste des deux enfants avec les nœuds des clusters non-utilisés du deuxième parent, dans l'ordre. On vérifie donc que chaque nœud provenant du deuxième parent n'aie pas été déjà fourni par le premier parent, et n'appartienne pas au même cluster qu'un nœud déjà fourni par le premier parent.

```
for (int p = 0; p < NBCLUST; p++){
      int boo = 0;
      for (int ok = cut1; ok < cut2; ok++){
            if((population[p2].chrom[p] == population[p1].chrom[ok]) ||
                (population[p2].chrom[p] == 1)){
                                    boo = 1;
                                }
      if(Clusters[population[p2].chrom[p] - 1] ==
      Clusters[population[p1].chrom[ok] - 1]){boo = 1;}
                                }
if ((boo == 0) && (population[p2].chrom[p] != 1)){
      population[i].chrom[(cut2 - cut1) + 1 + h] =
            population[p2].chrom[p];
      population[i + 1].chrom[NBCLUST - (cut2 - cut1) - 1 - h] =
            population[p2].chrom[p];
                  h++;
                  }
            }
      }
}
```

On finit la création de cette nouvelle génération avec des mutations sur toute la population, avec une chance de 5% de mutation, pour chacun des nœuds de chaque individu. La mutation elle-même consiste à échanger le nœud à muter contre un autre nœud du même cluster choisi aléatoirement.

```
void Mutation(int Clusters[])
{
      int i, j, temp, k;
      double p;

      for(i = 0; i < PopSize; i++)
      {
            for(j = 0; j < NBCLUST; j++)
            {
                  p = rand()%1000/1000.0;
                  if(p <= (ProbMut))
                  {
                        temp = population[i].chrom[j];
                        do{
                              k = random(CITYNB) + 1;
                        }while((k == 0) ||
                              (Clusters[k-1] !=
                               Clusters[population[i].chrom[j] - 1]));

                        population[i].chrom[j] = k;
                  }     }     }     }
```

Finalement, nous avons la fonction *main()*. Celle-ci commence par initialiser la fonction *rand()* en la réglant sur le temps système.

```
int main( int argc,  char* argv[])
{
      srand(time(0));
```

Notons que notre programme doit être exécuté avec 3 paramètres : le premier (*argv[1]*) étant le fichier texte contenant les distances entre les nœuds sous forme de matrice, chaque colonne étant séparée par un caractère espace ; le deuxième (*argv[2]*) étant la taille de cette matrice ; le troisième, le fichier texte contenant le vecteur de taille identique au nombre de colonnes de la matrice, qui indique pour chaque nœud à quel cluster il appartient. Notons aussi qu'avec notre format de données, de lecture, et de traitement, il est tout aussi possible de traiter les matrices symétriques qu'asymétriques, ce qui signifie que l'on peut aussi bien traiter les distances à vol d'oiseau que les distances routières effectives.

```
int tailleMat = atoi(argv[2]);
```

Afin de pouvoir lire et stocker les données placées dans le fichier texte contenant la matrice des distances, nous créons au préalable une matrice *distArray* de taille *[tailleMat][tailleMat]*. Nous faisons de même pour le vecteur de clusters, et nous initialisons les deux.

```
int nrows = tailleMat;
int ncols = tailleMat;

double **distArray;
// Allocation
distArray = new double*[nrows];
for (int i = 0; i < tailleMat; i++){
      distArray[i] = new double[ncols];
}
//initialization
for (int i = 0; i < nrows; i++){
      for (int j = 0; j < ncols; j++){
            distArray[i][j] = 0;
      }
}

int *clust;
// allocation
clust = new int[ncols];
// initialization
for (int i = 0; i < ncols; i++){
      clust[i] = 0;}
```

Nous créons ensuite un flot de données depuis le fichier texte contenant la matrice des distances, et vérifions que nous arrivons à l'ouvrir, sans quoi un message d'erreur est affiché.

```
int x, y;
ifstream in(argv[1]);
if (!in) {
      cout << "Cannot open file.\n";
}
```

Une fois le fichier ouvert, on le lit et on déverse les données dans notre matrice, puis on ferme le fichier. On applique ensuite la même procédure au fichier texte contenant les informations des clusters, qu'on place dans un tableau.

```
for (y = 0; y < tailleMat; y++) {
      for (x = 0; x < tailleMat; x++) {
            in >> distArray[x][y];
      }
}
in.close();

ifstream inC(argv[3]);
if (!inC) {
      cout << "Cannot open file.\n";
}

for (x = 0; x < tailleMat; x++) {
      inC >> clust[x];
}
inC.close();
```

On initialise ensuite les variables utilisées tout au long des fonctions. On récupère le nombre de clusters totaux en retirant la dernière valeur de la matrice des clusters, qui indique le numéro de cluster final.

```
CITYNB = tailleMat;
NBCLUST = clust[tailleMat - 1];
int *Temporary;
Temporary = new int[CITYNB];
for (int i = 0; i < CITYNB; i++){
      Temporary[i] = 0;
}

void  *population = malloc(sizeof(individual));
```

Une fois tout cela effectué, on commence l'appel des fonctions : création des premiers parents, calcul de leur longueur de tour, algorithmes de recherche locale, recherche du meilleur individu.

```
generation = 0;

CreateFirstIndividuals(clust, 0, PopSize);

CalcTourLength(Temporary, distArray);

TwoOpt(0, PopSize, Temporary,distArray);

ILS(clust, Temporary, distArray);

FindBestIndividual(Temporary);
```

Après cette première étape, on boucle sur la création, et amélioration des enfants, jusqu'à atteindre le nombre de générations fixée *MaxGeneration*, c'est-à-dire 500 (nombre trouvé après de nombreux tests comme étant au-dessus de la moyenne de générations nécessaires pour converger vers la solution optimale, tout en ne ralentissant pas le programme).

```
while(generation < MaxGeneration)
{

	generation++;

	CreateChildren(clust);

	CalcTourLength(Temporary, distArray);

	TwoOpt(0, PopSize, Temporary,distArray);

	ILS(clust, Temporary, distArray);

	FindBestIndividual(Temporary);
}
```

Après avoir atteint le nombre de générations voulues, on imprime le meilleur tour trouvé afin de le récupérer au niveau du code **PHP**.

```
for(int j = 0;j < NBCLUST; j++){
	printf("%d ",currentbest.chrom[j]);}
```

On finit par libérer la mémoire utilisée.

```
// free memory
delete [] clust;
delete [] Temporary;
delete [] population;

for (int i=0; i < nrows; i++){
      delete [] distArray[i];
}
```

3.3 Intégration en PHP

Maintenant que nous avons une fonction qui est capable de nous retourner le trajet optimal, il nous faut l'intégrer au site Web.

Nous possédons une page PHP nommée *viewPanier.php* qui affiche le panier courant de l'utilisateur, dans laquelle nous avons intégré le code qui nous permettra d'afficher la carte Google Maps™ et le trajet optimal.

viewPanier.php

Nous commençons par récupérer la liste des enseignes à visiter grâce à une commande **SQL**. Nous plaçons ensuite ces noms dans un vecteur.

```
<?php
$sql2 = "SELECT * FROM Panier,Contenu_Panier, Prix_Produit ,Produit,
Enseigne  WHERE Contenu_Panier.id_panier = Panier.id_panier AND
Prix_Produit.id_produit = Contenu_Panier.id_produit    AND
Prix_Produit.id_produit = Produit.id_produit AND Contenu_Panier.id_enseigne
= Prix_Produit.id_enseigne AND Enseigne.id_enseigne =
Prix_Produit.id_enseigne and Contenu_Panier.id_client =
".$_SESSION['id_client']. " and Contenu_Panier.id_panier = '".
$_SESSION['id_panier'] ."'";

$res2 = mysql_query($sql2);
$nb2 = mysql_numrows($res2);

$j = 0;

$enseignes="(";
while ($j < $nb2-1){
$enseignes=$enseignes."'".mysql_result($res2,$j, nom_enseigne") ."'.", ";
$j++;        }

if($nb2-1<0){$enseignes=$enseignes.")";}

else{$enseignes=$enseignes."'".mysql_result($res2, $j,
"nom_enseigne")."'.")";
```

On cherche ensuite dans la base de données toutes les coordonnées des magasins qui sont rattachés aux enseignes trouvées. On place ces informations dans des vecteurs **PHP** ainsi que des vecteurs **JavaScript**, pour s'en servir plus tard.

Notons que les coordonnées de tous les magasins sont enregistrées au préalable dans la base de données à l'aide d'un script qui va chercher les enseignes sur le site Google Maps, et en retire un fichier de type KML. Les adresses redondantes, ou ne répondant pas à la requête sont enlevées à la main. Le script filtre ensuite dans le fichier KML les coordonnées, et les enregistre dans la base de données.

```
$sql = "SELECT * FROM Enseigne,Franchise WHERE Enseigne.id_enseigne =
Franchise.id_enseigne AND Enseigne.nom_enseigne IN $enseignes";

$res = mysql_query($sql);
$nb = mysql_numrows($res);  // on recupere le nombre d'enregistrements
$i = 0;
$prix_total=0;
$nb_items=0;
echo "<script language='JAVASCRIPT'>var Adresses=new Array();</script>";
```

```
echo "<script language='JAVASCRIPT'>var Magas=new Array();</script>";
echo "<script language='JAVASCRIPT'>var lng=new Array();</script>";
echo "<script language='JAVASCRIPT'>var lat=new Array();</script>";
echo "<script language='JAVASCRIPT'>var img=new Array();</script>";

while ($i < $nb){
$adresse= mysql_result($res, $i, "adresse");
$enseigne= mysql_result($res, $i, "nom_enseigne");
$longitude= mysql_result($res, $i, "longitude");
$latitude= mysql_result($res, $i, "latitude");
$img= mysql_result($res, $i, "fichier_logo");

echo "<script language='JAVASCRIPT'>Magas[$i]= '$enseigne' ; </script>";
echo "<script language='JAVASCRIPT'>Adresses[$i]= '$adresse' ;</script>";
echo "<script language='JAVASCRIPT'>lng[$i]= '$longitude' ;</script>";
echo "<script language='JAVASCRIPT'>lat[$i]= '$latitude' ;</script>";
echo "<script language='JAVASCRIPT'>img[$i]= '$img' ;</script>";

$enseigneE[$i]= $enseigne ;
$longitudeE[$i]= $longitude ;
$latitudeE[$i]= $latitude ;
$adresseE[$i]= $adresse ;

$i++;
}
}
```

On fait finalement le lien vers notre fichier **PHP** de génération de trajet optimal, *GoogleMap.php*.

```
<?php include("GoogleMap.php"); ?>
```

GoogleMap.php

On commence par définir la fonction de distances qui nous permettra de remplir le fichier texte des distances. Celle-ci calcule les distances en prenant en compte la courbure de la Terre.

```
function distance($lat1, $lng1, $lat2, $lng2)
{
  $pi80 = M_PI / 180;
  $lat1 *= $pi80;
  $lng1 *= $pi80;
  $lat2 *= $pi80;
  $lng2 *= $pi80;
  $r = 6372.797; // mean radius of Earth in km
  $dlat = $lat2 - $lat1;
  $dlng = $lng2 - $lng1;
  $a = sin($dlat / 2) * sin($dlat / 2) +
       cos($lat1) * cos($lat2) * sin($dlng / 2) * sin($dlng / 2);

  $c = 2 * atan2(sqrt($a), sqrt(1 - $a));
  $km = $r * $c;
  return $km;       }
```

Afin de récupérer la latitude et la longitude de l'adresse du client, on envoie une requête de géocodage à Google Maps™, et on récupère le résultat dans le format **JSON** (JavaScript Object Notation). On décode et sépare le résultat afin d'en obtenir uniquement les coordonnées.

```
$json =
file_get_contents('http://maps.googleapis.com/maps/api/geocode/json?
       address='.urlencode($ad).'&sensor=false');

$output = json_decode($json);

$latitudeH = $output->results[0]->geometry->location->lat;
$longitudeH = $output->results[0]->geometry->location->lng;
```

On précise ensuite au programme le dossier dans lequel on va stocker les matrices des distances, et s'il n'existe pas, on le crée. On vérifie que le panier n'est pas vide, afin de ne pas faire de calculs inutiles.

```
$dir = 'dist';

if ( !file_exists($dir) ) {
  mkdir ($dir, 0777);
 }

if(sizeof($longitudeE)!=0){……}
```

Dans le but de créer le vecteur des appartenances des nœuds aux clusters, on parcours le vecteur de noms d'enseignes de chaque magasin, et on remplace chaque nom par un numéro de cluster, en commençant par le numéro 2, le numéro 1 étant réservé à l'habitation du client.

```
$k=2;
for($i = 0; $i < sizeof($longitudeE); $i++){
if(!is_int($enseigneE[$i])){
$token=$enseigneE[$i];
for($j = 0; $j < sizeof($longitudeE); $j++){
   if($enseigneE[$j]==$token){$enseigneE[$j]=$k;}
}
$k++;}       }
```

On crée un fichier texte personnalisé des distances, avec l'identification de la session de client et de son panier. On insère tout d'abord les distances entre l'adresse du client et les magasins, sur la première ligne et la première colonne de la matrice, avant de remplir les distances des magasins entre eux.

```
file_put_contents ($dir.'/dist'.$_SESSION['id_panier'].'.txt', '0 ');
for($i = 0; $i <sizeof($longitudeE); $i++){
$distanceH=distance($latitudeH, $longitudeH, $latitudeE[$i],
                                                $longitudeE[$i]);
file_put_contents ($dir.'/dist'.$_SESSION['id_panier'].'.txt', $distanceH.
                                            " ", FILE_APPEND | LOCK_EX);}


for($i = 0; $i <sizeof($longitudeE); $i++){
file_put_contents ($dir.'/dist'.$_SESSION['id_panier'].'.txt', "\r\n",
                                            FILE_APPEND | LOCK_EX);
$distanceH=distance($latitudeE[$i], $longitudeE[$i], $latitudeH,
                                            $longitudeH);
file_put_contents ($dir.'/dist'.$_SESSION['id_panier'].'.txt', $distanceH.
                                            " ", FILE_APPEND | LOCK_EX);
for($j = 0; $j <sizeof($longitudeE); $j++){
$distanceH=distance($latitudeE[$i], $longitudeE[$i], $latitudeE[$j],
                                            $longitudeE[$j]);
file_put_contents ($dir.'/dist'.$_SESSION['id_panier'].'.txt', $distanceH.
                                            " ", FILE_APPEND | LOCK_EX);
} }
```

On fait de même pour la matrice des clusters, en créant un fichier texte qui est personnalisé avec l'identification de la session du client et son panier. On le remplit ensuite du vecteur de numéros de clusters.

```
file_put_contents ($dir.'/clust'.$_SESSION['id_panier'].'.txt', '1 ');
for($i = 0; $i < sizeof($longitudeE); $i++){
file_put_contents ($dir.'/clust'.$_SESSION['id_panier'].'.txt',
                                $enseigneE[$i]." ", FILE_APPEND | LOCK_EX);
}
```

Une fois les deux fichiers texte remplis, on peut appeler notre algorithme génétique nommé *OptimalTourClusters*, qui prend comme paramètre le fichier des distances, la taille de la matrice, et le fichier des clusters, dans cet ordre. On récupère la sortie de l'exécutable, qui représente les nœuds à parcourir pour le trajet optimal, dans l'ordre. On crée aussi un vecteur JavaScript avec ces valeurs, afin de pouvoir les utiliser dans la partie **JavaScript** qui génère la carte Google Maps™. On passe aussi en **JavaScript** les coordonnées de l'adresse client.

```
$last_line = exec('./OptimalTourClusters '
                  .$dir.'/dist'.$_SESSION['id_panier'].
                  '.txt'.' '.(sizeof($longitudeE)+1).' '.$dir.
                  '/clust'.$_SESSION['id_panier'].'.txt');

$pieces = array_map('trim',explode(" ", $last_line));

echo "<script language='JAVASCRIPT'>var pieces=new Array();</script>";
for ($i = 1; $i < sizeof($pieces); $i++){
echo "<script language='JAVASCRIPT'>pieces[$i-1]=
                                    '$pieces[$i]' ;</script>";
}

echo "<script language='JAVASCRIPT'>var latitudeH;</script>";
echo "<script language='JAVASCRIPT'>latitudeH= '$latitudeH' ;</script>";
echo "<script language='JAVASCRIPT'>var longitudeH;</script>";
echo "<script language='JAVASCRIPT'>longitudeH= '$longitudeH' ;</script>";
}
?>
```

On passe à présent à la partie **JavaScript** de la page *GoogleMap.php*. On commence avec une fonction *clickedAddAddress2*, qui, au démarrage de la page Web, s'occupe d'appeler une autre fonction nommée *directions(..)*, uniquement si on possède une adresse de départ fournie par le client, ou l'adresse par défaut.

```
<script type="text/javascript">

function clickedAddAddress2() {


if ((adr!=null)&&(adr!="undefined")&&(adr!="")){

directions(0, document.forms['travelOpts'].walking.checked,
            document.forms['travelOpts'].avoidHighways.checked);
}
else{alert("Vous devez donner une adresse de d\351part.");}
}
```

On définit une fonction d'ajout de markers sur la carte Google Maps™, qui utilise des icônes personnalisées, et ajoute le marker à l'endroit spécifié en superposition des markers déjà existants.

```
function addMarker(location, num) {
if(num==1){letter="r";}
else{letter="b";}
  marker = new google.maps.Marker({
    position: location,
    map: map,
 icon:"icons/icon" + letter + num + ".png",
 zIndex: 10000000089
  });
  markersArray.push(marker);}
```

On définit aussi une fonction qui permet d'afficher tous les markers d'une carte, une fois qu'ils ont tous été définis, et placés dans le vecteur des markers.

```
function showOverlays() {


  if (markersArray) {
    for (i in markersArray) {

      markersArray[i].setMap(map);
    }
  }
}
```

La première fonction qui sera appelée pour initialiser la carte Google Maps™ est la fonction *initialize()*, qui instancie un afficheur d'instructions de direction, ainsi que l'objet

directionsService, qui sera celui qui va faire les requêtes de route. On centre ensuite la carte sur les coordonnées du centre de Montréal, et on attribue à la carte le * * qui se nomme « *map* », et aux instructions le * * qui se nomme « *path* ».

```
function initialize() {
directionsDisplay = new google.maps.DirectionsRenderer();
directionsService = new google.maps.DirectionsService();

var latlng = new google.maps.LatLng(45.50154, -73.55071);
var myOptions = {
      zoom: 12,
      center: latlng,
      mapTypeId: google.maps.MapTypeId.ROADMAP
    };
map = new google.maps.Map(document.getElementById("map"),
        myOptions);
directionsDisplay.setMap(map);
directionsDisplay.setPanel(document.getElementById("path"));

}
```

On définit ensuite la fonction la plus importante, *directions()*, qui nous permet de générer notre trajet optimal sur la carte. Cette fonction prend en paramètres les choix du client par rapport à son moyen de transport (en voiture ou à pied), et à son exigence vis-à-vis des autoroutes (les éviter ou pas).

```
function directions(m, walking, avoid) {
```

On commence par initialiser un tableau de nœuds par lesquels on voudra passer dans le tour. On passe donc à ce tableau les valeurs du trajet optimal que l'on a obtenues auparavant, au format de leurs latitudes et longitudes. On crée ensuite une variable *request* qui donnera à Google Maps™ le point d'origine et la destination du tour, qui sont tous deux l'adresse donnée par le client, le tableau de nœuds à visiter sur le chemin, ainsi que les paramètres du tour. On note que l'on garde l'option « *optimizeWaypoints* » à *true*, car notre programme a calculé le meilleur tour basé sur les distances à vol d'oiseau. Ainsi, ajouter l'option d'optimiser le tour avec les véritables valeurs de trajet par la route permettra de procurer au client un chemin encore plus proche de l'optimal.

```
var WPoints=new Array();

for (t=0;t<pieces.length;t++){
WPoints[t] = {location: lat[pieces[t]-2]+", "+
                                lng[pieces[t]-2], stopover: true};}

if(walking){var mode=google.maps.DirectionsTravelMode.WALKING}
else{var mode=google.maps.DirectionsTravelMode.DRIVING}


var request = {
  origin:adr,
  destination:adr,
  waypoints: WPoints,
  optimizeWaypoints: true,
  avoidHighways: avoid,
  travelMode: mode,
  unitSystem: google.maps.DirectionsUnitSystem.METRIC
};
```

On utilise l'objet *directionsService* pour envoyer cette requête à Google Maps™, et afficher les instructions. On affiche les markers, afin de pouvoir visualiser le trajet sur la carte.

```
directionsService.route(request, function(result, status) {
    if (status == google.maps.DirectionsStatus.OK) {
      directionsDisplay.setDirections(result);
    }
});
showOverlays();
```

On crée un marker aux coordonnées de l'adresse fournie par le client, et on choisit pour ce marker un logo de maison, afin qu'il situe facilement sur la carte son point de départ. On affiche ce marker sur la carte.

```
var myLatLng = new google.maps.LatLng(latitudeH, longitudeH);
addMarker(myLatLng,1987);

showOverlays();
```

On parcourt ensuite les nœuds de notre tour optimal, et pour chacune de leurs coordonnées, on crée un marker, avec comme icône l'image officielle de l'enseigne, afin que le client puisse facilement reconnaître les lieux auxquels il devra s'arrêter dans son tour. On affiche ces markers sur la carte.

```
var myLatLng;
var marker;
for (i=0;i<pieces.length;i++){
if(img[pieces[i]-2]=="defaut.jpg"){}
else{

myLatLng = new google.maps.LatLng(lat[pieces[i]-2], lng[pieces[i]-2]);
marker = new google.maps.Marker({
  position: myLatLng,
 map: map,
 icon: "admin/images/"+img[pieces[i]-2],
 zIndex: 10000000090+i
  });
markersArray.push(marker);}
}

showOverlays();}
```

Google Maps™, quand on lui demande une route avec arrêts, affiche les instructions pour chaque tronçon entre les arrêts, en indiquant la distance et le temps pour chaque tronçon. Malheureusement, il n'affiche pas le temps et la distance totale, car il y a une notion d'arrêt. On définit donc finalement une fonction qui va nous permettre d'afficher à l'utilisateur le temps total et la distance totale de son parcours, afin qu'il puisse se rendre compte de l'efficacité du trajet optimal proposé.

Cette fonction prend chacun des tronçons de la route obtenue par Google Maps™, et ajoute dans deux variables la distance totale et la durée totale du trajet. Elle leur attribue un * * chacune, nommés « *disttot* » et « *totalt* ». On affiche la distance en kilomètres, et la durée en minutes.

```
function computeTotalDistance(result) {
    var total = 0;
    var myroute = result.routes[0];
    for (i = 0; i < myroute.legs.length; i++) {
      total += myroute.legs[i].distance.value;
    }
    total = total / 1000;

    document.getElementById("disttot").innerText =
       document.getElementById("disttot").textContent = total + " km.";
}
function computeTotalTime(result) {
```

```
    var total = 0;
    var myroute = result.routes[0];
    for (i = 0; i < myroute.legs.length; i++) {
      total += myroute.legs[i].duration.value;
    }
    total = Math.round(total / 60.);

    document.getElementById("totalt").innerText =
       document.getElementById("totalt").textContent = total + " min."; }
```

Toutes ces fonctions sont appelées au niveau du *<body>* de la page Web *viewPanier.php*, où l'on commence par initialiser la carte Google Maps™ avec *initialize()*, puis on appelle *clickedAddAddress2()* qui va à son tour appeler *directions()*. On ajoute un *listener*, ou écouteur, sur la carte qui va noter quand le trajet va être affiché, et va mettre à jour la distance totale et le temps total, pour l'afficher au client. Notons que les cases à cocher qui proposent un trajet à pied ou d'éviter les autoroutes rechargent la carte automatiquement dès qu'on clique dessus.

```
$(document).ready(function() {

 initialize();
 clickedAddAddress2();

  google.maps.event.addListener(directionsDisplay, 'directions_changed',
function() {
  computeTotalTime(directionsDisplay.directions);
    computeTotalDistance(directionsDisplay.directions);
  });
 });
```

3.4 Distances réelles pour un trajet en voiture

Notons que le trajet optimal actuel est calculé en tenant compte des distances à vol d'oiseau entre les lieux. Mais il est tout à fait possible de récupérer les distances effectives en voiture entre les lieux grâce à une simple commande envoyée à Google Maps™ :

'http://maps.google.com/maps/nav?output=json&q=from:'.$a.'%20to:'.$b

avec *$a* l'adresse de départ, et *$b*, l'adresse d'arrivée.

Cependant, la création de la matrice des distances doit se faire au niveau du serveur, afin de pouvoir appeler notre algorithme de trajet optimal qui s'y trouve, et d'éviter au client d'accepter l'installation d'un programme ActiveX sur son ordinateur. Cela nous impose donc d'être en possession des distances entre les lieux au niveau du serveur, et donc du code PHP, et de faire la requête de distance à Google depuis une même adresse IP. Comme la limite de requêtes imposée par Google est de 2500 par 24 heures et par adresse IP (limite pour les requêtes gratuites), si un client devait vouloir parcourir plus de trois enseignes à succursales multiples (ce qui équivaut déjà à un minimum d'une cinquantaine de franchises, donc 2500 requêtes), son calcul serait déjà bloqué après une seule tentative de calcul du trajet optimal.

Nous aurions aussi comme possibilité de calculer au préalable les distances entre tous les magasins de la ville, et ne demander pour chaque client que la distance entre son habitation et les magasins potentiels à visiter. Cependant, comme nous répertorions 428 franchises de magasins divers sur l'île de Montréal, établir les distances entre elles en utilisant le service gratuit de Google Maps™ prendrait 73 jours. Et même avec une base de données des distances déjà calculées, nous serions tout de même obligés de calculer les distances entre l'adresse du client et toutes les franchises des enseignes qu'ils veut visiter. Avec en moyenne trois enseignes à visiter par client, donc une cinquantaine de franchises, on enverrait ainsi une centaine de requêtes vers Google Maps™. Nous serions donc limités à environ 25 clients par jour, ce qui n'est pas acceptable.

Il reste cependant deux possibilités pour avoir un crédit suffisant de requêtes pour obtenir les distances réelles des trajets en voiture: Google Maps™ propose de pouvoir acheter à un prix non précisé les transactions excédentaires ; il propose aussi un abonnement Premier, qui donne accès à 100'000 requêtes par jour (http://code.google.com/apis/maps/documentation/premier/).

CHAPITRE IV

RÉSULTATS

Rappelons que notre algorithme a été développé en trois phases : la première version, que nous nommons la version originelle, choisissait pour chaque enseigne le magasin le plus proche de l'adresse du client ; la deuxième version, soit celle de l'algorithme GTSP à vol d'oiseau, appliquait un algorithme génétique au problème, tout en s'appuyant sur les distances à vol d'oiseau entre les lieux ; la troisième version, qui est celle de l'algorithme GTSP à distances de voiture, n'est pas applicable avec nos moyens actuels, mais va tout de même être comparée en termes d'optimalité dans ces pages.

Comme le système mis en place sur le site SmartShopping au préalable considérait les magasins les plus proches à vol d'oiseau de l'adresse du client, et en faisait un parcours, il ne prenait pas du tout en compte le fait qu'un magasin plus lointain, mais se trouvant plus sur la route générale vers les autres magasins, pourrait diminuer le coût du trajet. Au lieu de rayonner autour de l'adresse du client, l'algorithme de Voyageur de Commerce Généralisé considère souvent quel est le magasin le plus éloigné du point de départ, et place les autres magasins le long de son chemin.

Nous avons observé une nette amélioration des temps de parcours pour presque tous les trajets qui visitent plus d'une enseigne en utilisant l'algorithme GTSP avec les distances à vol d'oiseau, par rapport aux temps de parcours de l'algorithme originel. Evidemment, avec une seule enseigne à visiter, l'ancien algorithme produisait déjà la solution optimale. Pour pratiquement toutes les adresses de départ, et toutes les combinaisons de magasins, le trajet était plus court avec l'algorithme GTSP à vol d'oiseau; les seuls cas de figure où l'ancien algorithme était aussi bon que le nouveau, arrivaient lorsque l'adresse du client était au Centre-ville, et quand les magasins à visiter étaient des enseignes à nombreuses franchises. Il n'y avait cependant aucun cas de figure testé où l'algorithme GTSP à vol d'oiseau n'était pas au moins aussi bon que l'ancien algorithme mis en place.

Avec le nouvel algorithme, on note un temps d'exécution de la page légèrement plus long, ce qui est dû à l'écriture et à la lecture des fichiers de distances. Il n'était malheureusement pas possible d'éviter cette étape, car le langage PHP ne peut pas exécuter de programme extérieur avec des variables qui lui sont propres. Il faut donc faire appel à une source de données extérieure, base de données ou fichier de données.

Les figures 4.1-4.9 présentent trois exemples de trajet optimal trouvés par trois différentes versions de l'algorithme que nous avons testées.

On peut observer sur les figures 4.2, 4.5, et 4.8, où les résultats de l'algorithme utilisant les distances à vol d'oiseau sont présentés, que les temps de parcours sont grandement améliorés, et que les parcours sont beaucoup plus réguliers par rapport à la version originelle, dont les résultats sont présentés sur les figures 4.1, 4.4, et 4.7, respectivement.

On remarque clairement sur les figures 4.1, 4.4, et 4.7, qu'avec l'ancien algorithme le trajet impose de faire des allers et retours de part et d'autre du point de départ, et ceci bien que les magasins soient tous au plus proche de l'adresse du client. En effet, comme ils ne sont pas tous placés stratégiquement dans la même direction par rapport à l'adresse de départ, le va-et-vient fait perdre du temps. Il n'y a donc aucune considération au niveau de l'économie de trajet, et la seule optimisation est celle de l'ordre dans lequel on visite les magasins proches.

L'algorithme GTSP permet de minimiser réellement le parcours, et ne s'arrête pas aux distances propres de chaque lieu. On pourrait ainsi avoir un magasin qui fait partie des plus éloignés par rapport à l'adresse du client, et tout de même trouver qu'il fait partie du trajet le plus court, si par hasard tous les autres magasins à visiter se trouvent le long de sa route.

On voit bien sur les figures 4.2, 4.5, et 4.8, que le trajet produit par l'algorithme génétique à vol d'oiseau cherche à regrouper les magasins qu'il visite dans une même direction, celle du magasin le plus lointain.

Il ne serait donc plus nécessaire d'avoir sur le site Web une option de rayon maximal où le client accepte de se déplacer, car cela ne ferait qu'empêcher l'algorithme de donner un trajet réellement optimal. Il faudrait à présent considérer la notion de distance maximale que le client accepte de parcourir, qui serait une borne réellement utile. Cependant, cette option requiert de faire tourner notre algorithme de nombreuses fois, en enlevant tour à tour chaque enseigne de son panier jusqu'à trouver un parcours plus court que sa borne, et cela peut se révéler très couteux en temps.

Afin d'avoir un aperçu des résultats qui seraient obtenus avec l'algorithme GTSP *à distance de voiture* (les distances routières réelles), les mêmes tests ont été effectués avec les distances effectives fournies par Google Maps™, dans le cadre des restrictions imposées au niveau du nombre de requêtes disponibles par jour, soit 2500 requêtes.

On peut observer, pour les trois exemples fournis précédemment, leur équivalent produit par l'algorithme GTSP à distances de voiture sur les figures 4.3, 4.6, et 4.9. Pour le trajet A, on constate une petite amélioration du temps et de la distance de parcours entre l'algorithme GTSP à vol d'oiseau et l'algorithme GTSP à distance de voiture : on gagne ainsi 1 minute, et 1.275km dans le trajet. Pour le trajet B, entre les deux algorithmes GTSP, on ne gagne pas de temps dans le trajet, mais un peu de distance, soit 228 mètres. Pour ce qui est du trajet C, il n'y a aucune différence entre les deux algorithmes GTSP, à vol d'oiseau ou à distance de voiture.

Afin de mettre en évidence le gain en temps et en distance entre les deux algorithmes GTSP, à vol d'oiseau ou à distance en voiture, nous avons effectué une série de tests de parcours (les résultats moyens obtenus sont présentés sur les figures 4.10-4.14), en faisant varier le nombre de magasins dans le panier, et en prenant comme point d'origine l'adresse par défaut du site. Comme le temps de parcours est directement proportionnel à la distance, nous n'avons créé des graphes que pour les différences de distances.

Nous pouvons observer sur les figures 4.10, 4.11, 4.12, et 4.13 que le gain de distance entre l'ancien algorithme et le GTSP à vol d'oiseau est considérable, avec de 2.75% à 21.68% de gain en distance moyenne (voir figure 4.14, qui montre le gain de temps, en pourcentages, en fonction du nombre de magasins). Il est donc confirmé que

l'implémentation d'un algorithme GTSP est extrêmement utile pour améliorer la fonctionnalité de parcours optimaux.

En observant ces mêmes figures, nous constatons aussi que l'amélioration apportée par les distances en voiture n'est pas significative, et on constate que la plupart des trajets sont donc les mêmes que ceux trouvés par l'algorithme à vol d'oiseau. Ce manque d'amélioration est encore plus flagrant sur la figure 4.14, qui montre que les gains de distances pour les deux algorithmes GTSP sont presque identiques.

On peut se demander si, pour le peu de gain qu'un algorithme à distance de voiture apporte, cela vaut le coût de prendre un abonnement payant auprès de Google Maps™. De plus, même si l'on remplit une matrice dans la base de données de distances effectives en voiture, il faudra toujours faire des requêtes sur le moment auprès de Google Maps™ afin de connaître les distances entre les magasins et l'adresse du client. Cela équivaut à une moyenne de 60 à 120 requêtes par client. Malheureusement, Google Maps™ exige que les requêtes lui soient envoyées à une vitesse non-excessive, ce qui se traduit dans notre cas à une requête par seconde, sans quoi Google Maps™ répond par une erreur. Il est donc difficile d'imposer au client un temps de chargement pour sa page qui est entre 1 ou 2 minutes, en sachant que le gain en temps est minime.

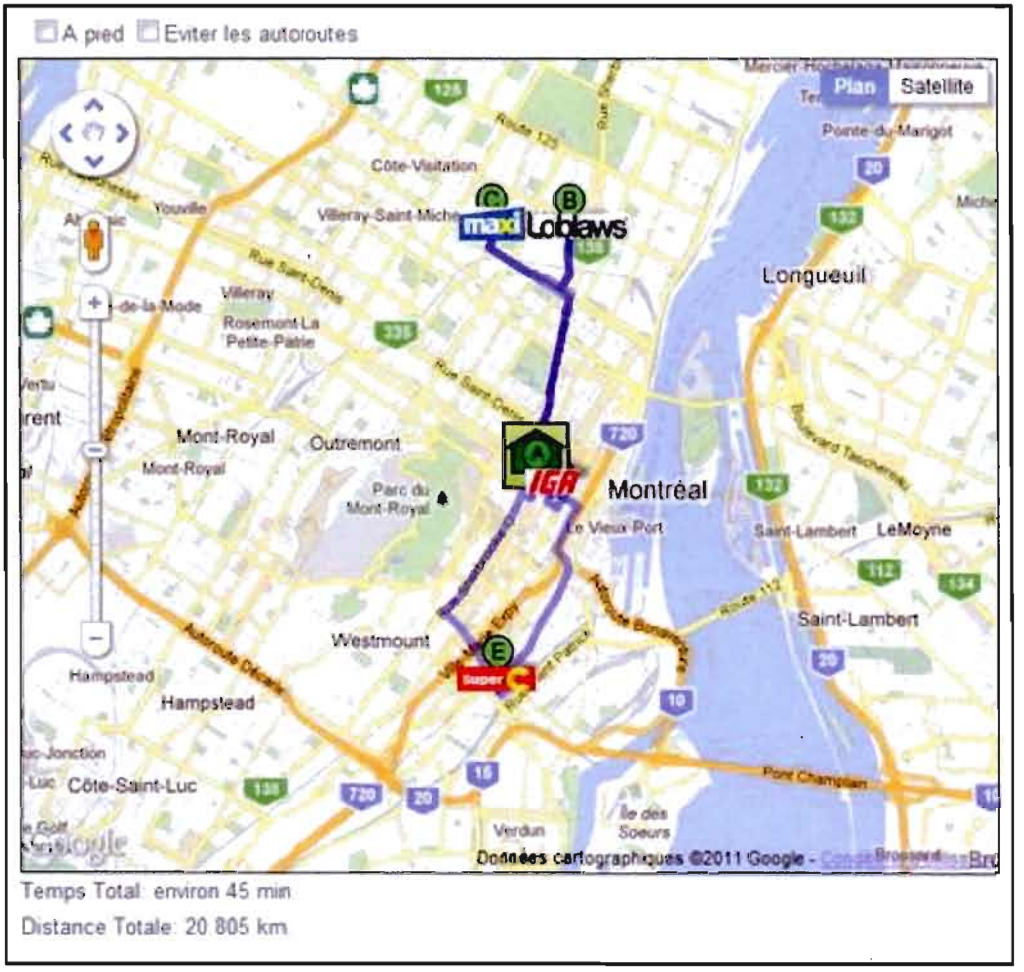

Figure 4.1 Trajet A, obtenu par l'algorithme originel, et prenant 45 minutes de trajet.

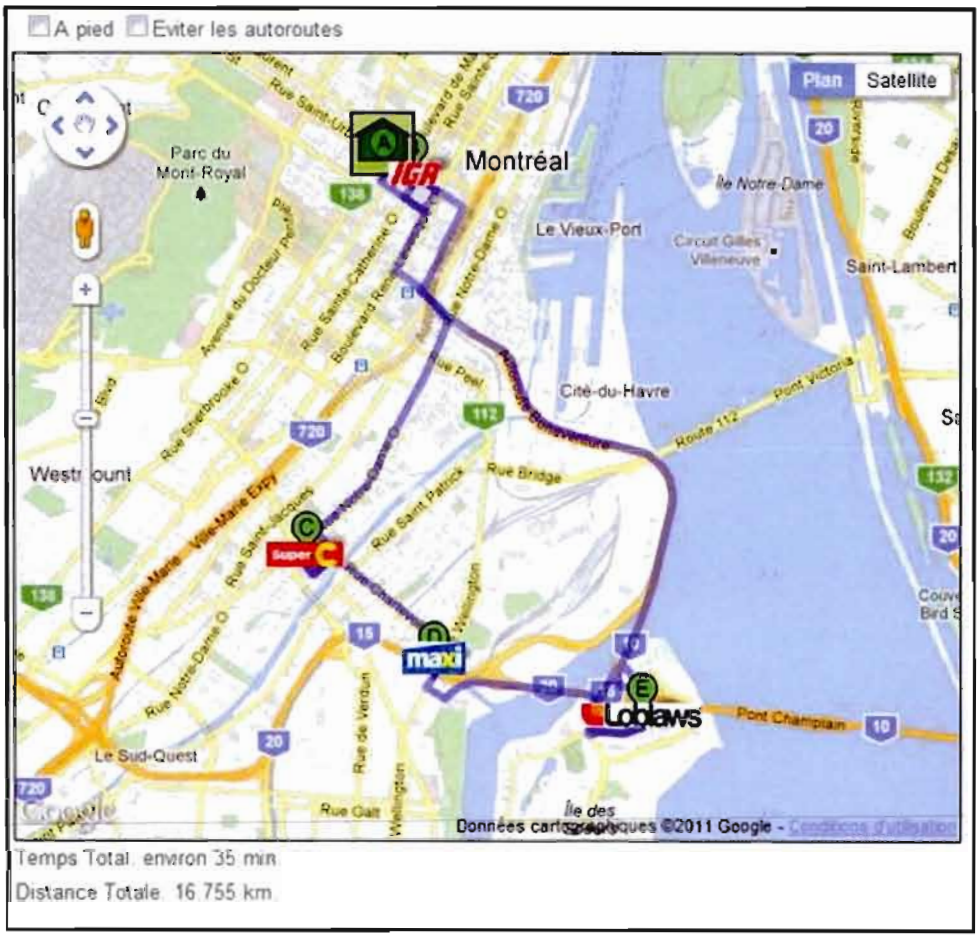

Figure 4.2 Trajet A, obtenu par l'algorithme GTSP à vol d'oiseau, et apportant un gain de 10 minutes par rapport à l'algorithme originel.

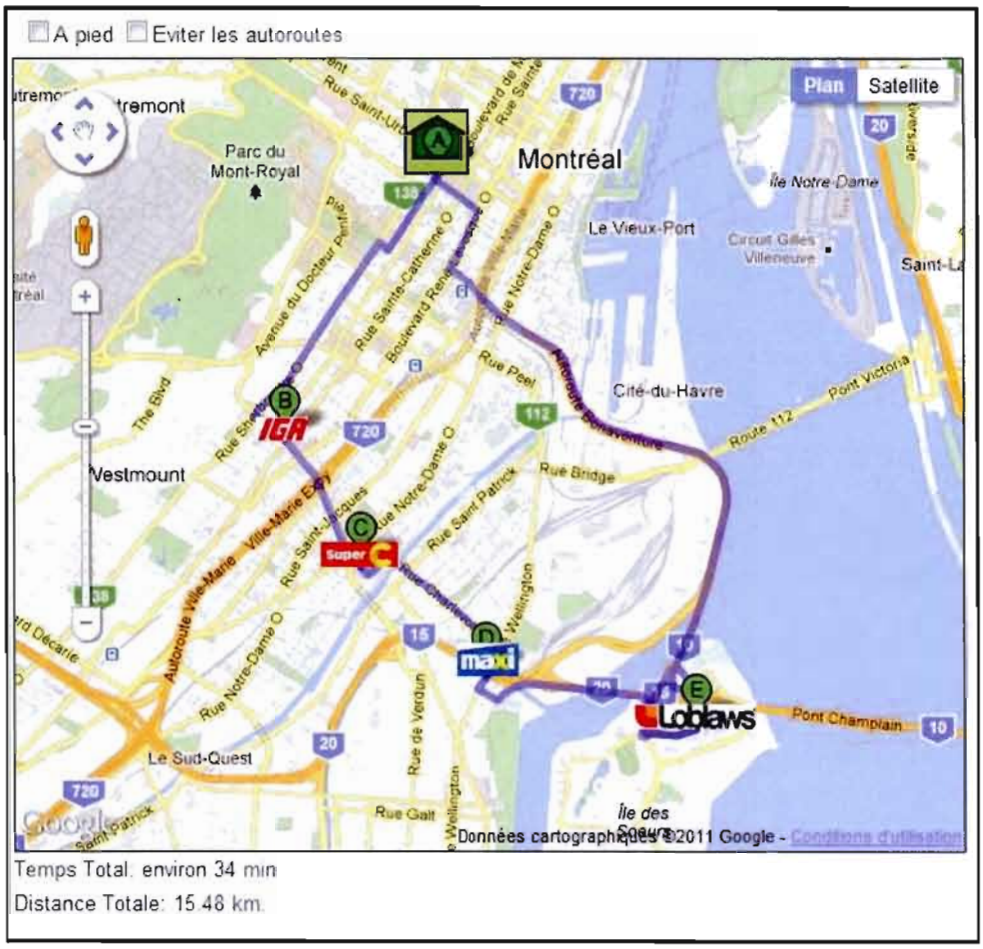

Figure 4.3 Trajet A, obtenu par l'algorithme GTSP à distances de voiture, et apportant un gain de 11 minutes par rapport à l'algorithme originel.

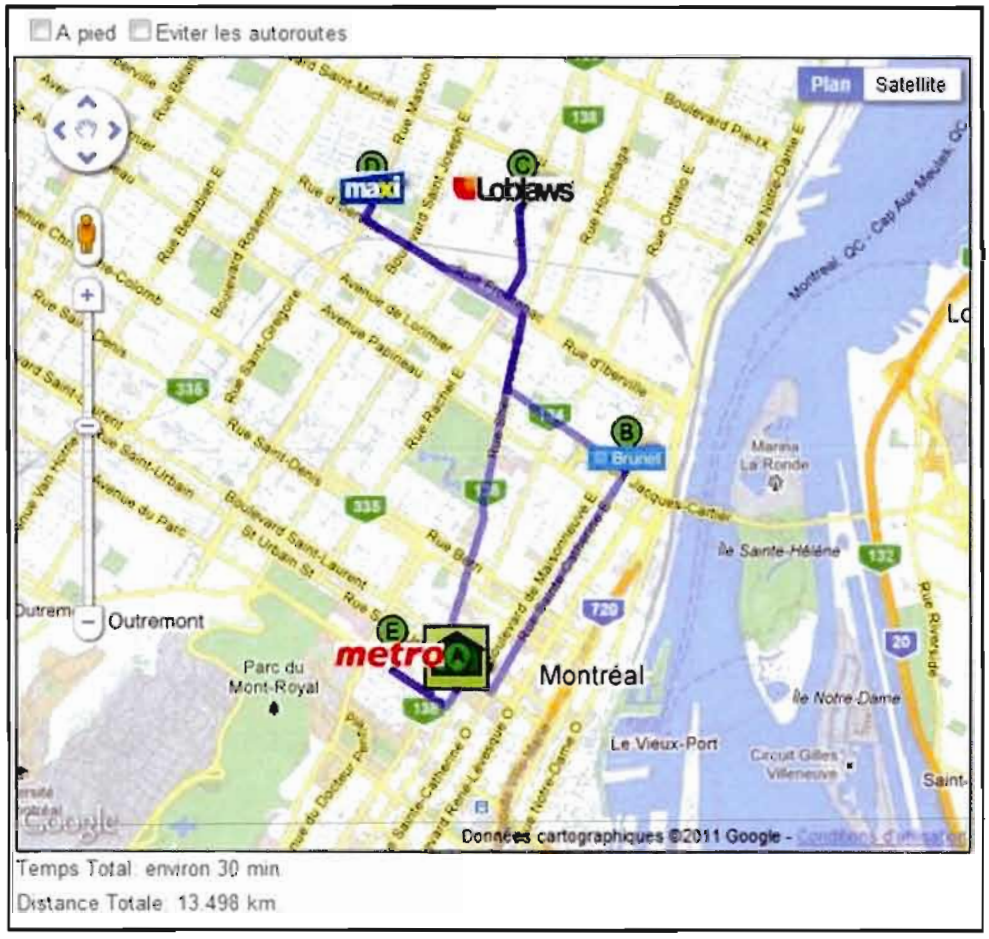

Figure 4.4 Trajet B, obtenu par l'algorithme originel, et prenant 30 minutes de trajet.

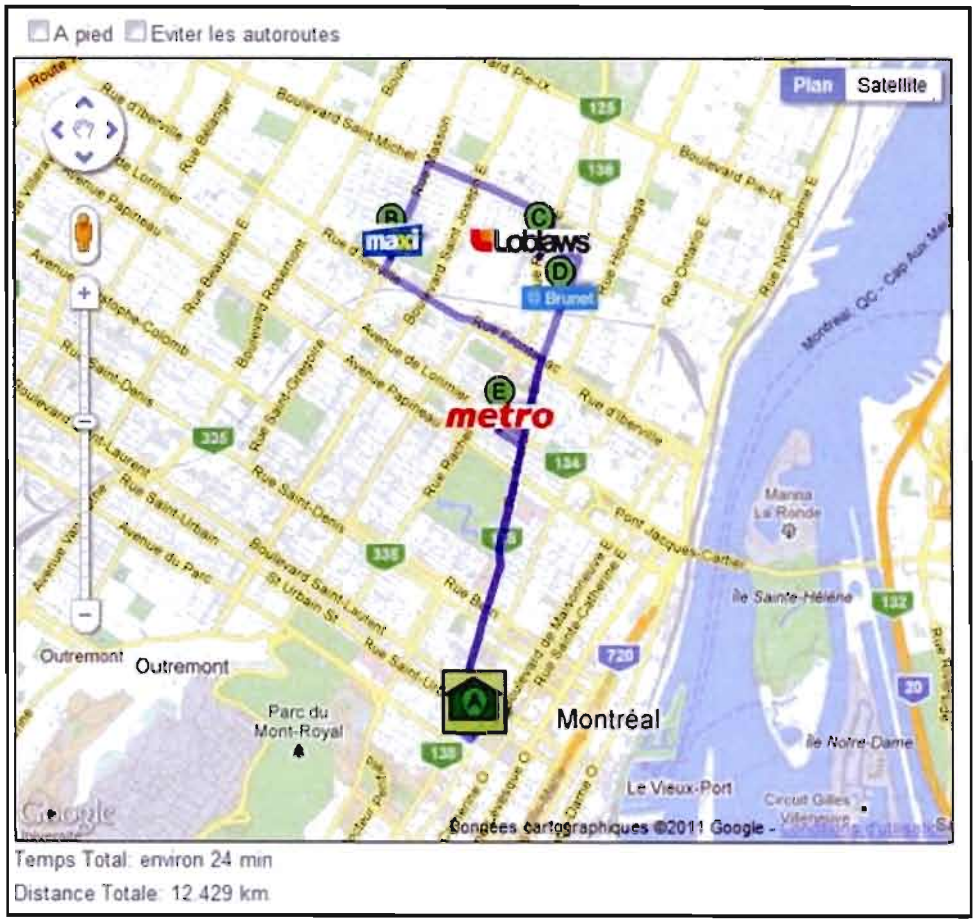

Figure 4.5 Trajet B, obtenu par l'algorithme GTSP à vol d'oiseau, et apportant un gain de 6 minutes par rapport à l'algorithme originel.

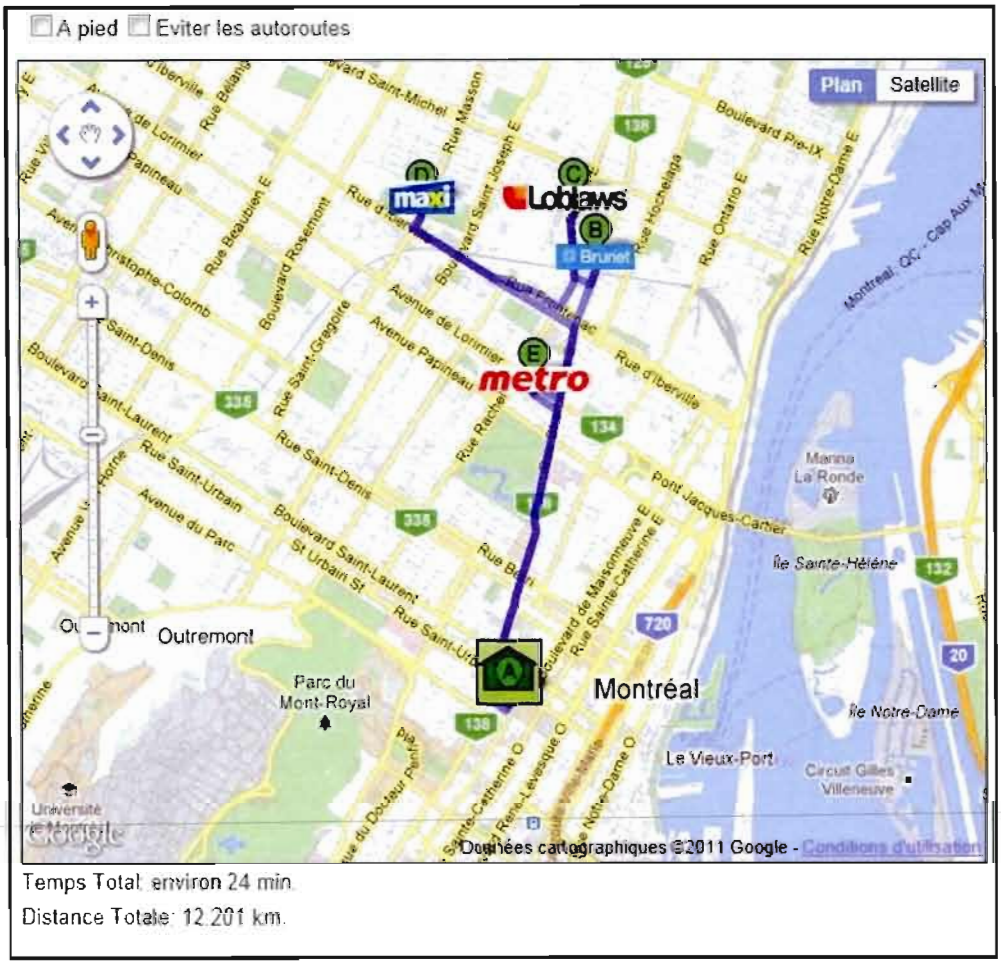

Figure 4.6 Trajet B, obtenu par l'algorithme GTSP à distances de voiture, et apportant un gain de 6 minutes par rapport à l'algorithme originel.

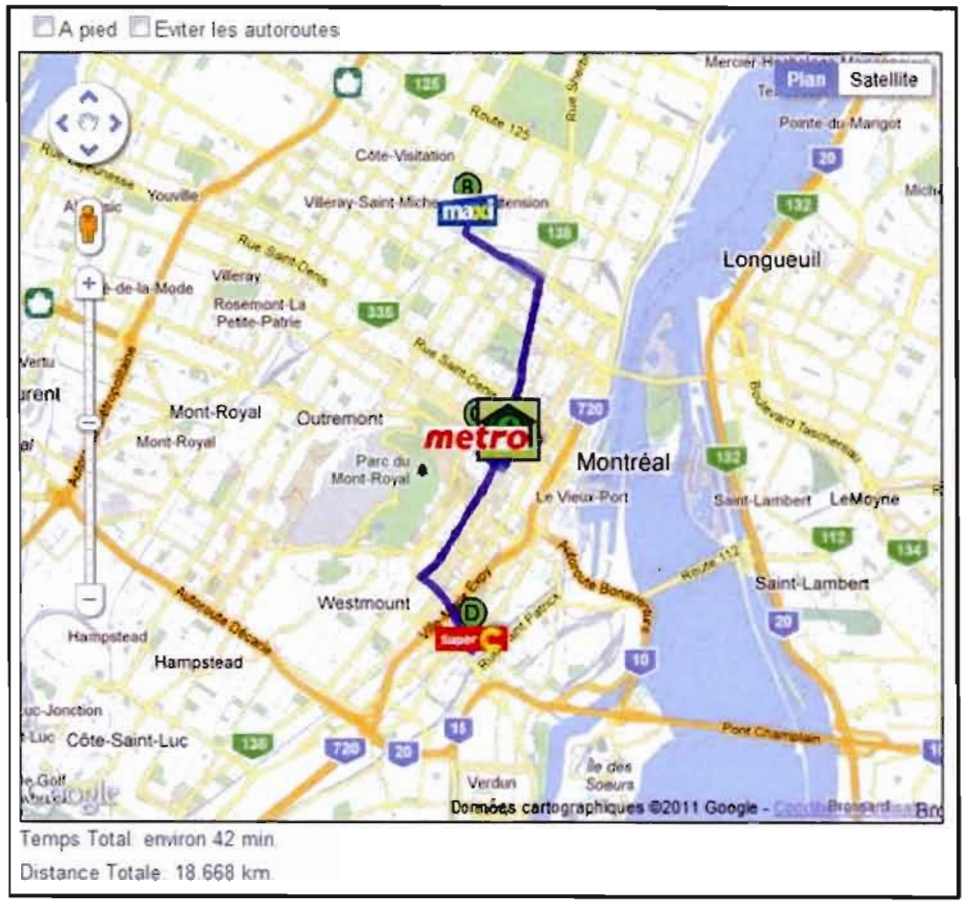

Figure 4.7 Trajet C, obtenu par l'ancien algorithme, et prenant 42 minutes de trajet.

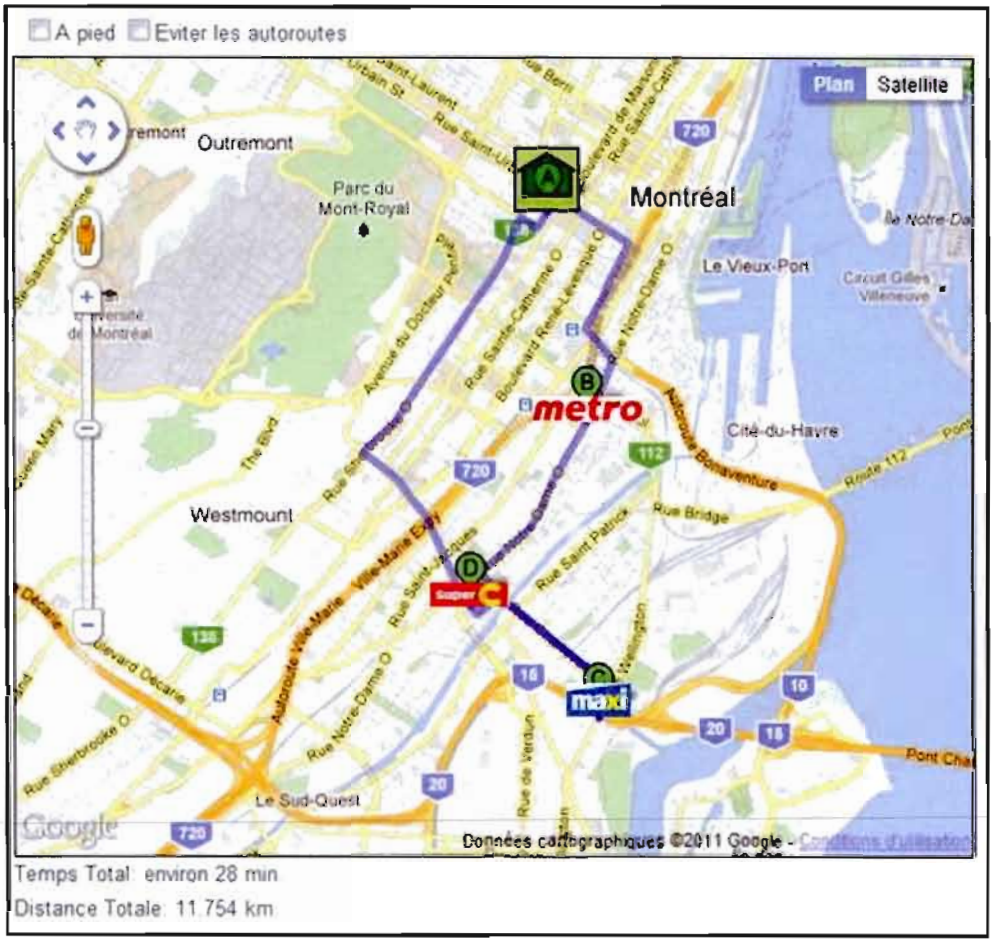

Figure 4.8 Trajet C, obtenu par l'algorithme GTSP à vol d'oiseau, et apportant un gain de 14 minutes par rapport à l'algorithme originel.

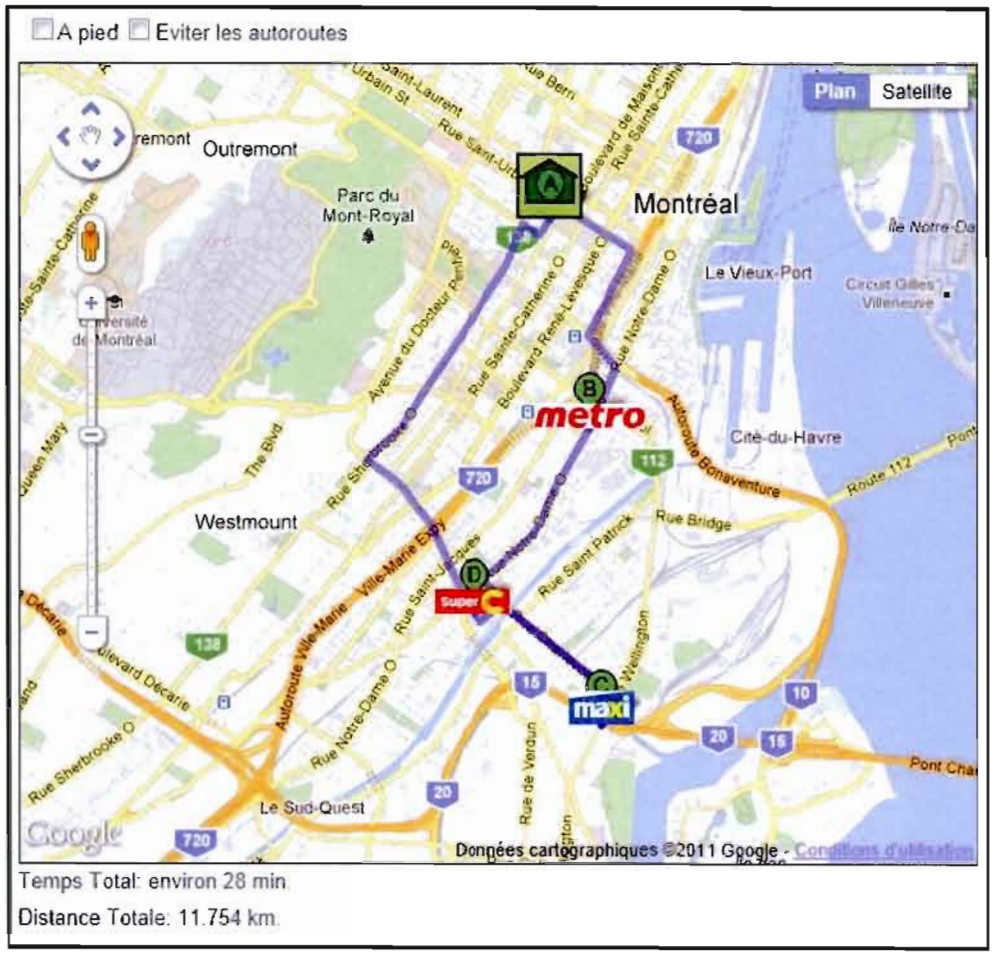

Figure 4.9 Trajet C, obtenu par l'algorithme GTSP à distances de voiture, et apportant un gain de 14 minutes par rapport à l'algorithme originel.

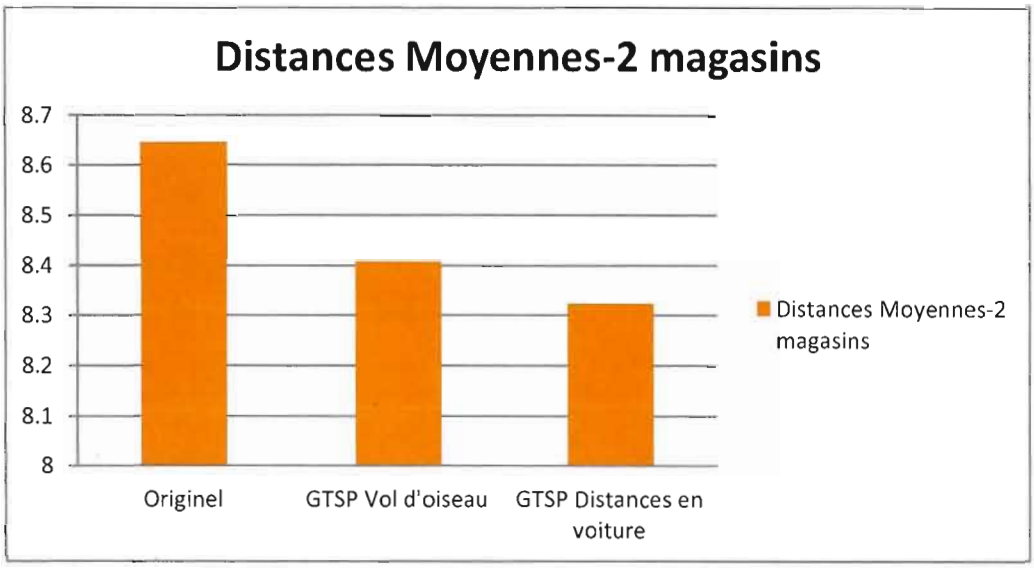

Figure 4.10 Distances moyennes à parcourir en kilomètres pour les 3 algorithmes, et le cas de 2 magasins.

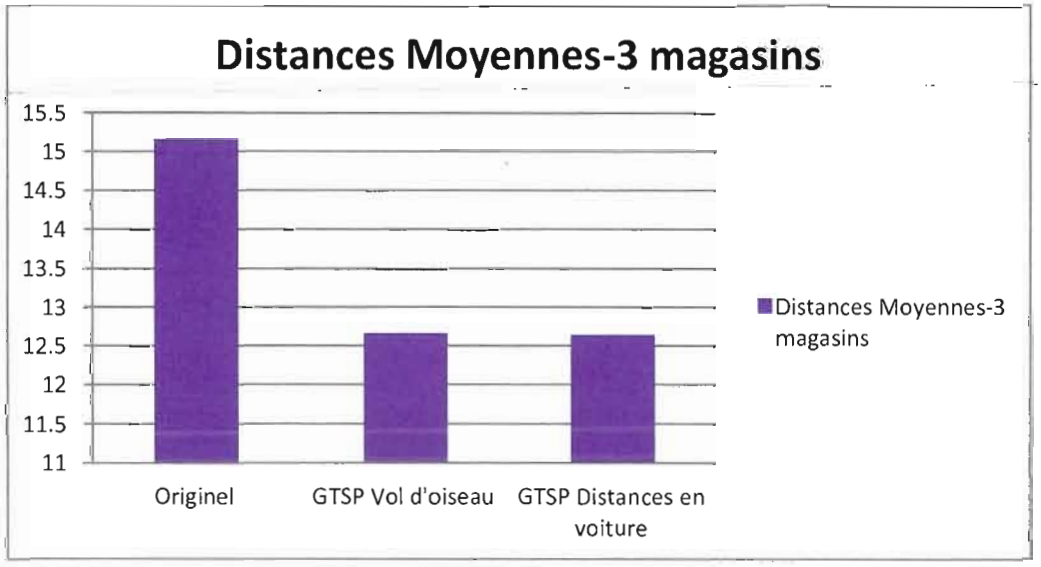

Figure 4.11 Distances moyennes à parcourir en kilomètres pour les 3 algorithmes, et le cas de 3 magasins.

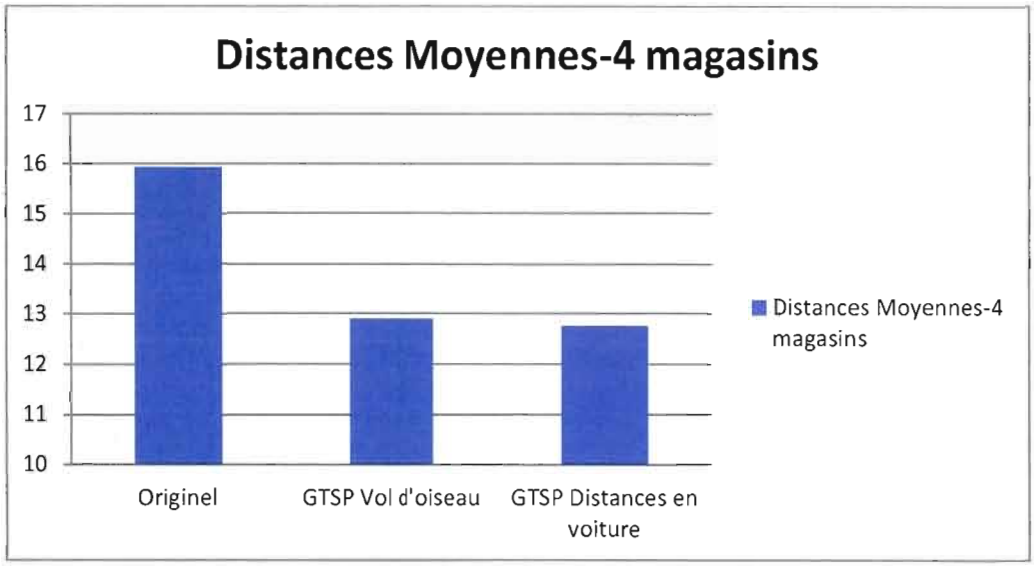

Figure 4.12 Distances moyennes à parcourir en kilomètres pour les 3 algorithmes, et le cas de 4 magasins.

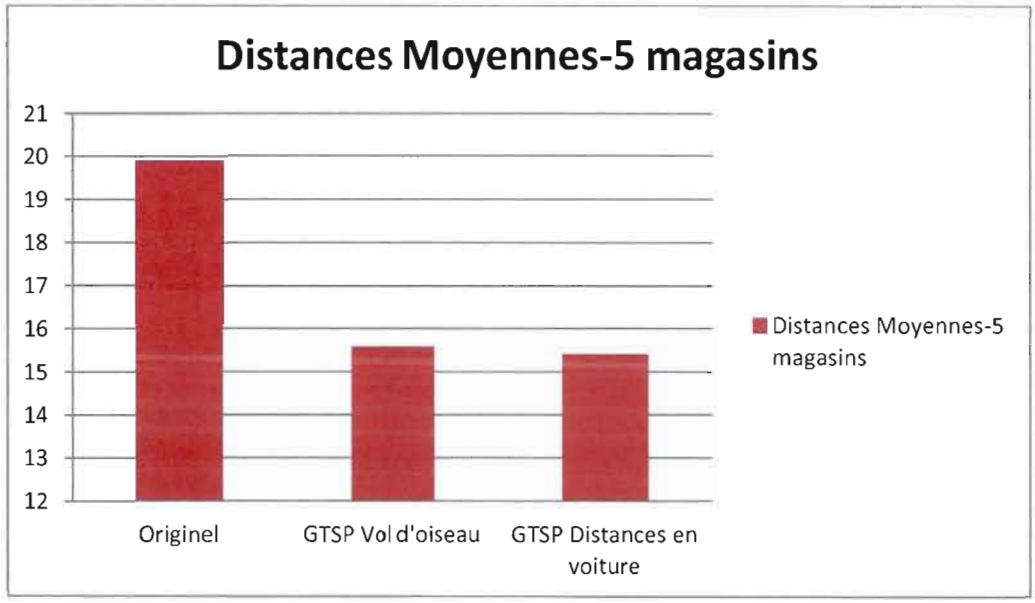

Figure 4.13 Distances moyennes à parcourir en kilomètres pour les 3 algorithmes, et le cas de 5 magasins.

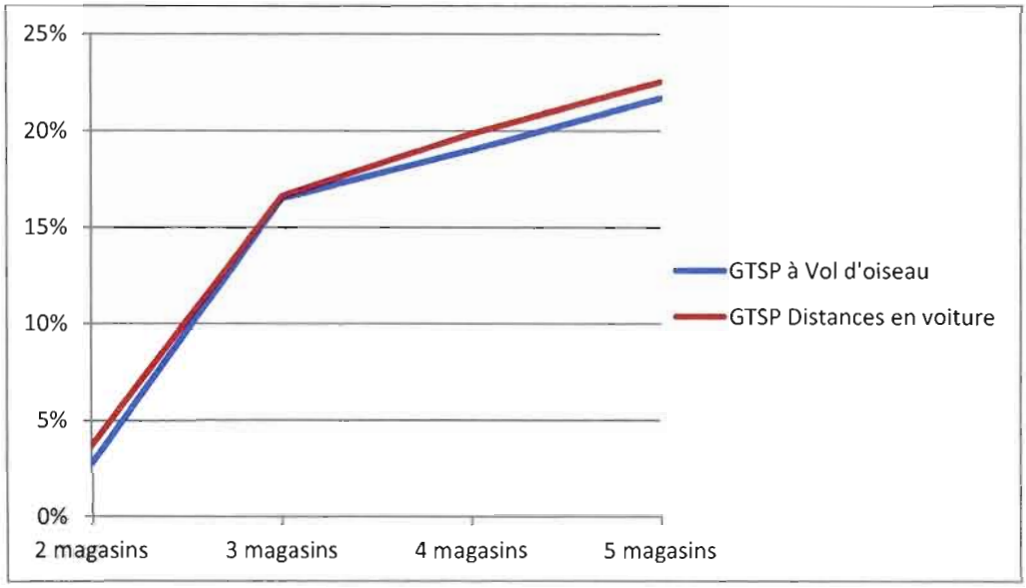

Figure 4.14 Pourcentages de gains moyens de distance, obtenus par les algorithmes GTSP, versions : distances à vol d'oiseau et distances routières, par rapport à l'algorithme originel.

CHAPITRE V

CONCLUSION

Ce projet de maîtrise avait pour but de comparer différents algorithmes pour le problème de Voyageur de Commerce Généralisé, puis de choisir le plus adapté pour calculer un trajet optimal entre les magasins d'alimentation de l'île de Montréal sur le site internet SmartShopping (http://www.trex.uqam.ca/~smartshopping). Nous étions intéressés par l'algorithme le plus efficace possible pour résoudre le problème du Voyageur de Commerce Généralisé dans le cas d'un échantillon de petite taille. A cette fin, les meilleurs algorithmes actuels ont été comparés afin de choisir le plus adapté à une page Web et à l'outil Google Maps™. Les algorithmes génétiques montrant des résultats nettement plus performants, l'algorithme décrit dans l'article de Tasgetiren et al. [TSPL07] a été implémenté, celui-ci étant le plus approprié pour répondre aux besoins du projet.

Notre objectif était de fournir la possibilité d'agrémenter une carte de type Google Maps™ avec un algorithme de Voyageur de Commerce Généralisé. Ainsi, plutôt que de trouver les lieux les plus proches du point de départ, l'algorithme implémenté calcule un trajet optimal, ou proche de l'optimal, et place tous les lieux au mieux sur un trajet fluide.

L'algorithme de Voyageur de Commerce Généralisé a été implémenté en langage C++, et intégré à une page PHP, qui fait appel aux services de Google Maps™ en JavaScript. Afin de faire le lien entre PHP et l'exécutable C++, les données de distances ont été stockées dans un fichier texte.

L'obstacle le plus important rencontré lors de l'implémentation de l'algorithme fut les interactions avec l'API de Google Maps™, ce dernier limitant la quantité de requêtes gratuites qu'un site Web peut effectuer par jour et par adresse IP. Cela a été un facteur contraignant qui a empêché la création d'un algorithme complet pour l'île de Montréal, produisant les parcours optimaux sur la base des distances routières effectives. En effet, au vu

du nombre élevé de magasins recensés sur le site, cela prendrait plusieurs mois pour remplir la matrice des distances complète entre les différents lieux.

En comparant les résultats avant et après le déploiement de l'algorithme de Voyageur de Commerce Généralisé à vol d'oiseau, nous avons pu constater une nette amélioration des temps de parcours, avec au pire des cas une longueur du trajet égale à celle fournie par l'ancien algorithme.

Dans le cadre d'une suite possible à ce projet, on pourrait envisager, soit l'achat d'une licence Google Maps™ Premier, afin d'avoir à disposition une plus grande quantité de requêtes possibles (soit 100'000 par jour et par IP), soit adhérer aux requêtes supplémentaires payables à la pièce, et ainsi pouvoir apporter une nouvelle dimension au trajet optimal, celle de la distance routière effective. Cependant, on a pu constater au chapitre des résultats que la distance routière n'apportait que très peu d'amélioration au trajet trouvé en utilisant les distances à vol d'oiseau. L'achat d'une telle licence serait donc questionnable. De plus, on a pu observer que le temps de chargement moyen d'une page avec les distances effectives en voiture prendrait plus d'une minute, ce qui pourrait se révéler peu attractif pour les utilisateurs de notre site Web.

Nous constatons donc que l'algorithme de Voyageur de Commerce Généralisé utilisant les distances à vol d'oiseau est un excellent compromis, donnant des résultats très proches de la solution optimale, et assurant un temps d'exécution qui reste agréable pour le client d'une page Web, de même qu'un coût minime, car l'API de Google Maps™ est gratuite pour ce genre d'utilisation.

APPENDICE A

CODE C++

Code C++, permettant le calcul d'un tour optimal face à un problème de Voyageur de Commerce Généralisé, avec comme paramètre une matrice de distances.

```
//============================================================================
// Nom         : OptimalTour.cpp
// Auteur      : Tania Joly
// Version     : 2011
// Description : Tasgetiren et al. Genetic Algorithm in C++, Ansi-style
//============================================================================
#include <iostream>
#include <cstdlib>
#include  "stdlib.h"
#include  "stdio.h"
#include  "time.h"
#include <fstream>
#include <sstream>
#include <vector>
#include <assert.h>
#include <string.h>

using namespace std;

// ********** Constantes et variables générales**********
#define POPSIZE  200
```

```
#define PopSize  100

int MaxGeneration = 500;
double ProbMut = 0.05;
int CITYNB;
int NBCLUST;
int CHROMLENGTH = NBCLUST;
int generation;
struct individual{
        int chrom[25];
        double totaldist;
};
struct individual bestindividual;
struct individual currentbest;
struct individual population[POPSIZE];

// ********** Déclaration des fonctions **********
int random(int randmax);
void CreateFirstIndividuals(int Clusters[], int start, int stop);
void CreateChildren(int Clusters[]);
void FindBestIndividual(int Temporary[]);
void SelectBest(int start, int stop, struct individual *newpopulation);
void SelectOperator();
void PTLCrossover(int Clusters[]);
void Mutation(int Clusters[]);
void PrintOutput(int Temporary[]);
void CalcTourLength(int Temporary[], double *Distance[]);
void TwoOpt(int debut, int fin, int Temporary[], double *Distance[]);
void ILS(int Clusters[], int Temporary[], double *Distance[]);
//=========================================================================
```

```
// ********** Sélection par tournoi de deux **********
void SelectBest(int start, int stop, struct individual *newpopulation){
        int i, p1, p2;
        struct individual temp[PopSize];

        for (i = start; i < stop; i++){
                do{
                        p1 = random(POPSIZE);
                        p2 = random(POPSIZE);
                }while((p1 == p2) || (population[p1].totaldist == 0) || _
                        (population[p2].totaldist == 0));
                if(population[p1].totaldist < population[p2].totaldist){
                        temp[i] = population[p1];
                }
                else{
                        temp[i]=population[p2];
                }
        }
        for (i = start; i < stop; i++){
                newpopulation[i] = temp[i];
                population[i] = newpopulation[i];
        }
}
//=============================================================================
// ********** Sélection sur toute la population **********
void SelectOperator(void){
        struct individual newpopulation[PopSize];
        int start, stop;
        start = 0;
        stop = (int)(PopSize);
```

```
        SelectBest(start, stop, newpopulation);
}
//=============================================================================
// ********** random modulo randmax **********
int random(int randmax){
        return rand()%randmax;
}
//=============================================================================
// ********** Crossover PTL avec tournoi de deux pour 1er parent **********
void PTLCrossover(int Clusters[]){
        if(NBCLUST>2){
                int i, p1, p2, p1bis;
                for(i = PopSize; i < POPSIZE; i+=2){
                        p1 = random(PopSize);
                        do{
                                p1bis = random(PopSize);
                                p2 = random(PopSize);
                        }while((p1 == p2) || (p1 == p1bis) || (p2 == p1bis));
                        if (population[p1].totaldist > population[p1bis].totaldist){
                                p1 = p1bis;
                        }
                        int cut1;
                        int cut2;
                        do{
                                cut1 = random(NBCLUST);
                                cut2 = random(NBCLUST);
                        }while((cut2 <= cut1) || (cut1 == 0));
                        int g = 0;
                        population[i].chrom[0] = 1;
                        population[i+1].chrom[0] = 1;
```

```
for (int k = 0; k < cut2 - cut1; k++){
        if(population[p1].chrom[cut1 + k] == 1){
                g++;
        }
        population[i].chrom[k + 1] = population[p1].chrom[cut1 +
_
        k + g];
        population[i + 1].chrom[NBCLUST - (cut2 - cut1) + k] = _
                population[p1].chrom[cut1 + k + g];
}
int h = 0;
for (int p = 0; p < NBCLUST; p++){
        int boo = 0;
        for (int ok = cut1; ok < cut2; ok++){
                if((population[p2].chrom[p] == _
                        population[p1].chrom[ok]) || _
                        (population[p2].chrom[p] == 1)){
                        boo = 1;
                }
                if(Clusters[population[p2].chrom[p] - 1] == _
                        Clusters[population[p1].chrom[ok] - 1]) _
                        {boo = 1;}
        }
        if ((boo == 0) && (population[p2].chrom[p] != 1)){
                population[i].chrom[(cut2 - cut1) + 1 + h] = _
                        population[p2].chrom[p];
                population[i + 1].chrom[NBCLUST - (cut2 - cut1) _
                - 1 - h] = population[p2].chrom[p];
                h++;
        }
```

```
                        }
                }
        }
}
//=============================================================================
// ********** Mutation avec 5% de chance sur un noeud au hasard **********
void Mutation(int Clusters[]){
        int i, j, k, temp;
        double p;
        for(i = 0; i < PopSize; i++){
                for(j = 0; j < NBCLUST; j++){
                        p = rand()%1000 / 1000.0;
                        if(p <= (ProbMut)){
                                temp = population[i].chrom[j];
                                do{
                                        k = random(CITYNB) + 1;
                                }while((k == 0) || (Clusters[k - 1] != _
                                        Clusters[population[i].chrom[j] - 1]));
                                population[i].chrom[j] = k;
                        }
                }
        }
}
//=============================================================================
void TwoOpt(int debut, int fin, int Temporary[], double *Distance[]){
        int i, j, k, l, p, t, u, temp;
        struct individual tempo[POPSIZE];
        double TotalDistance;
        int count = 1;
        for(l = debut;l < fin; l++){
```

```
while(count == 1){
        count = 0;
        for(k = 1; k < CHROMLENGTH - 2; k++){
                for(i = 1; i < CHROMLENGTH - k; i++){
                        for(j = k + 1; j < CHROMLENGTH; j++){
                                t = i;
                                u = j;
                                tempo[l] = population[l];
                                while(t < u){
                                        temp = tempo[l].chrom[t];
                                        tempo[l].chrom[t]                =
tempo[l].chrom[u];
                                        tempo[l].chrom[u] = temp;
                                        t++;
                                        u--;
                                }
                                TotalDistance = 0.0;
                                for(int h = 0; h < NBCLUST; h++){
                                        Temporary[h]                     =
population[i].chrom[h];
                                }
                                for(p = 1; p < CITYNB; p++){
                                        TotalDistance = TotalDistance + _
                                                Distance[Temporary[p  -  1]
_
                                                - 1][Temporary[p] - 1];
                                }
                                TotalDistance = TotalDistance + _
                                        Distance[Temporary[CITYNB  -  1]  -
1][0];
                                tempo[l].totaldist = TotalDistance;
                                if(tempo[l].totaldist < _
```

```
                                                    population[l].totaldist){
                                                    population[l] = tempo[l];
                                                    count = 1;
                                            }
                                    }
                            }
                    }
            }
    }
    //=========================================================================
    // ********** on cherche pour chaque cluster, le noeud qui minimise le trajet **********
    void ILS(int Clusters[], int Temporary[], double *Distance[]){
            int i;
            int p;
            for(i = 0; i < PopSize; i++){
                    int w = 0;
                    struct individual best;
                    while(w < NBCLUST){
                            w++;
                            // pour chaque h, on parcoure toutes les villes, vérifie même
    cluster,
                            // essaie insérer, si mieux on garde
                            for(int k = 1; k < CITYNB; k++){
                                    best = population[i];
                                    if(Clusters[k] == Clusters[best.chrom[w] - 1]){
                                            best.chrom[w] = k + 1;
                                            double TotalDistance = 0.0;
                                            for(int h = 0; h < NBCLUST; h++){
                                                    Temporary[h] = best.chrom[h];
```

```
                                        }
                                        for(p = 1; p < NBCLUST; p++){
                                                TotalDistance = TotalDistance + _
                                                        Distance[Temporary[p - 1] - _
                                                        1][Temporary[p] - 1];
                                        }
                                        TotalDistance = TotalDistance + _
                                                Distance[Temporary[NBCLUST - 1] - 1][0];
                                        best.totaldist = TotalDistance;
                                        if(best.totaldist < population[i].totaldist){
                                                population[i] = best;
                                                w = 0;
                                        }
                                }
                        }
                }
        }
}
//=============================================================================
void CreateFirstIndividuals(int Clusters[], int start, int stop){
        int i, j, k, token;
        for(i = start; i < stop; i++){
                population[i].chrom[0] = 1;
                for(j = 1; j < NBCLUST; j++){
                        token = 0;
                        while(token == 0){
                                token = 1;
                                population[i].chrom[j] = random(CITYNB) + 1;
                                for(k = 0; k < j; k++){
```

```
                                        if((population[i].chrom[j]                    ==
population[i].chrom[k]) _
                                                || (Clusters[population[i].chrom[k] - 1]
== _
                                                Clusters[population[i].chrom[j] - 1])){
                                                token=0;
                                        }
                                }
                        }
                }
        }
}
//=============================================================================
void CreateChildren(int Clusters[]){
        SelectOperator();
        PTLCrossover(Clusters);
        Mutation(Clusters); _
}
//=============================================================================
void PrintOutput(int Temporary[]){
        printf("Generation=%d,     Best     Total     Distance=%f:\n",     generation,
currentbest.totaldist);
        printf("Best individual:");
        for(int j = 0; j < NBCLUST; j++){
                printf("(%d)", currentbest.chrom[j]);
        }
        printf("\n\n");
}
//=============================================================================
void CalcTourLength(int Temporary[], double *Distance[])
{
        int i, j;
```

```
        double TotalDistance = 0;
        for(i = 0; i < PopSize; i++){
                for(int h = 0; h < NBCLUST; h++){
                        Temporary[h] = population[i].chrom[h];
                }
                for(j = 1; j < NBCLUST; j++){
                        TotalDistance = (TotalDistance + Distance[Temporary[j - 1] - _
                                1][Temporary[j] - 1]);
                }
                TotalDistance = TotalDistance + Distance[Temporary[NBCLUST - 1] -
1][0];
                population[i].totaldist = TotalDistance;
                TotalDistance = 0;
        }
}
//=============================================================================
void FindBestIndividual(int Temporary[]){
        int i;
        bestindividual = population[0];
        for(i = 1; i < PopSize ; i++){
                if(population[i].totaldist < bestindividual.totaldist){
                        bestindividual = population[i];
                }
        }
        if(generation == 0){
                currentbest = bestindividual;
        }
        else{
                if(bestindividual.totaldist < currentbest.totaldist){
                        currentbest = bestindividual;
```

```
                }
        }
}
//=============================================================================
// ********** main program **********
int main( int argc,  char* argv[]){
        srand(time(0));
        int tailleMat = atoi(argv[2]);
        int nrows = tailleMat;
        int ncols = tailleMat;
        double **distArray;
        // allocation mémoire
        distArray = new double*[nrows];
        for (int i = 0; i < tailleMat; i++){
                distArray[i] = new double[ncols];
        }
        //initialisation
        for (int i = 0; i < nrows; i++){
                for (int j = 0; j < ncols; j++){
                        distArray[i][j] = 0;
                }
        }

        int *clust;
        // allocation
        clust = new int[ncols];
        // initialisation
        for (int i = 0; i < ncols; i++){
                clust[i] = 0;
        }
```

```
// ********** lecture fichier des distances **********
int x, y;
ifstream in(argv[1]);
if (!in) {
        cout << "Cannot open file.\n";
}
for (y = 0; y < tailleMat; y++){
        for (x = 0; x < tailleMat; x++) {
                in >> distArray[x][y];
        }
}
in.close();
// ********** fin lecture fichier des distances **********

//  ********** lecture fichier des clusters **********
ifstream inC(argv[3]);
if (!inC){
        cout << "Cannot open file.\n";
}
for (x = 0; x < tailleMat; x++) {
        inC >> clust[x];
}
inC.close();
// ********** fin lecture fichier des clusters **********

CITYNB = tailleMat;
NBCLUST = clust[tailleMat - 1];
// init Temporary
int *Temporary;
```

```
// allocation
Temporary = new int[CITYNB];
// initialisation
for (int i = 0; i < CITYNB; i++){
        Temporary[i] = 0;
}
void  *population = malloc(sizeof(individual));
generation = 0;

CreateFirstIndividuals(clust, 0, PopSize);
CalcTourLength(Temporary, distArray);
TwoOpt(0, PopSize, Temporary,distArray);
ILS(clust, Temporary, distArray);
FindBestIndividual(Temporary);

while(generation < MaxGeneration){
        generation++;
        CreateChildren(clust);
        CalcTourLength(Temporary, distArray);
        TwoOpt(0, PopSize, Temporary, distArray);
        ILS(clust, Temporary, distArray);
        FindBestIndividual(Temporary);
}

for(int j = 0; j < NBCLUST; j++){
        printf("%d ", currentbest.chrom[j]);
}

// free memory
delete [] clust;
```

```
        delete [] Temporary;
        delete [] population;

        for (int i = 0; i < nrows; i++){
                delete [] distArray[i];
        }

        return 0;
}
//=========================================================================
```

APPENDICE B

CODE PHP

Code PHP de la page GoogleMap.php sur le site Web SmartShopping, permettant l'affichage d'un tour optimal, selon les magasins choisis par le client de par son panier.

```
<!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//FR">
<html>
<head>
	<meta name="viewport" content="initial-scale=1.0, user-scalable=no"/>
	<meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
	<?php
		if ($_SESSION["langue"] == "en"){
			echo '<script type="text/javascript"
	src="http://maps.google.com/maps/api/js?sensor=false&language=en">
			</script>';
			$pied = "By foot";
			$eviterAutoroutes = "Avoid highways";
			$tempsTotal = "Total time: around ";
			$distanceTotal = "Total Distance: ";
		}
		else if($_SESSION["langue"] == "fr"){
```

```
                echo '<script type="text/javascript"

src="http://maps.google.com/maps/api/js?sensor=false&language=fr">

                </script>';

                $pied = "A pied";

                $eviterAutoroutes = "Eviter les autoroutes";

                $tempsTotal = "Temps Total: environ ";

                $distanceTotal = "Distance Totale: ";

        }


function deleteDir($dir){

        if (substr($dir, strlen($dir)-1, 1) != '/') $dir .= '/';

        echo $dir;

        if ($handle = opendir($dir)){

                while ($obj = readdir($handle)){

                        if ($obj != '.' && $obj != '..'){

                                if (is_dir($dir.$obj)){

                                        if (!deleteDir($dir.$obj)) return false;

                                }

                                elseif (is_file($dir.$obj)){

                                        if (!unlink($dir.$obj)) return false;

                                }

                        }

                }

                closedir($handle);
```

```
                if (!@rmdir($dir)) return false;
                return true;
        }
        return false;
}

function distance($lat1, $lng1, $lat2, $lng2){
        $pi80 = M_PI / 180;
        $lat1 *= $pi80;
        $lng1 *= $pi80;
        $lat2 *= $pi80;
        $lng2 *= $pi80;
        $r = 6372.797; // mean radius of Earth in km
        $dlat = $lat2 - $lat1;
        $dlng = $lng2 - $lng1;
        $a = sin($dlat / 2) * sin($dlat / 2) + _
                cos($lat1) * cos($lat2) * sin($dlng / 2) * sin($dlng / 2);
        $c = 2 * atan2(sqrt($a), sqrt(1 - $a));
        $km = $r * $c;
        return $km;
}
$json = file_get_contents('http://maps.googleapis.com/maps/api/geocode/json? _
        address='.urlencode($ad).'&sensor=false');
$output = json_decode($json);
$latitudeH = $output->results[0]->geometry->location->lat;
```

```
$longitudeH = $output->results[0]->geometry->location->lng;
$dir = 'dist';
// create new directory with 777 permissions if it does not exist yet
// owner will be the user/group the PHP script is run under
if ( !file_exists($dir) ){
        mkdir ($dir, 0777);
}
if(sizeof($longitudeE)!=0){
        $k=2;
        for($i = 0; $i < sizeof($longitudeE); $i++){
                if(!is_int($enseigneE[$i])){
                        $token=$enseigneE[$i];
                        for($j = 0; $j < sizeof($longitudeE); $j++){
                                if($enseigneE[$j]==$token){
                                        $enseigneE[$j]=$k;
                                }
                        }
                        $k++;
                }
        }
        file_put_contents ($dir.'/dist'.$_SESSION['id_panier'].'.txt', '0 ');
        for($i = 0; $i <sizeof($longitudeE); $i++){
                $distanceH = distance($latitudeH, $longitudeH, $latitudeE[$i],
                                $longitudeE[$i]);
                file_put_contents ($dir.'/dist'.$_SESSION['id_panier'].'.txt',_
```

```
                    $distanceH." ", FILE_APPEND | LOCK_EX);
}
for($i = 0; $i < sizeof($longitudeE); $i++){
        file_put_contents ($dir.'/dist'.$_SESSION['id_panier'].'.txt',_
                "\r\n", FILE_APPEND | LOCK_EX);
        $distanceH = distance($latitudeE[$i], $longitudeE[$i], _
                $latitudeH, $longitudeH);
        file_put_contents ($dir.'/dist'.$_SESSION['id_panier'].'.txt',_
                $distanceH." ", FILE_APPEND | LOCK_EX);
        for($j = 0; $j < sizeof($longitudeE); $j++){
                $distanceH = distance($latitudeE[$i], $longitudeE[$i], _
                        $latitudeE[$j], $longitudeE[$j]);
                file_put_contents ($dir.'/dist'.$_SESSION _
                        ['id_panier'].'.txt', $distanceH." ", _
                        FILE_APPEND | LOCK_EX);
        }
}
file_put_contents ($dir.'/clust'.$_SESSION['id_panier'].'.txt', '1 ');
for($i = 0; $i < sizeof($longitudeE); $i++){
   file_put_contents ($dir.'/clust'.$_SESSION['id_panier'].'.txt',_
                $enseigneE[$i]." ", FILE_APPEND | LOCK_EX);
}
$last_line = exec('./OptimalTourClusters _
        '.$dir.'/dist'.$_SESSION['id_panier'].'.txt'.' _
        '.(sizeof($longitudeE)+1).' '.$dir.'/clust'.$_SESSION _
```

```
                ['id_panier'].'.txt');
        $pieces = array_map('trim',explode(" ", $last_line));
        echo "<script language='JAVASCRIPT'>var pieces=new Array();</script>";
        for ($i = 1; $i < sizeof($pieces); $i++){
                echo "<script language='JAVASCRIPT'>_
                        pieces[$i-1]= '$pieces[$i]';</script>";
        }
        echo "<script language='JAVASCRIPT'>var latitudeH;</script>";
        echo "<script language='JAVASCRIPT'>latitudeH= '$latitudeH';</script>";
        echo "<script language='JAVASCRIPT'>var longitudeH;</script>";
        echo "<script language='JAVASCRIPT'>longitudeH= '$longitudeH' _
                </script>";
    }
?>
<script type="text/javascript" src="jquery-1.6.2.js"></script>
<script type="text/javascript" src="jquery.blockUI.js"></script>
<script type="text/javascript"
    src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
</script>
<script type="text/javascript">
    var map;
    var map2;
    var directionsDisplay;
    var directionsService;
    var directions;
```

```
var dirn;

var dirn2;

var markersArray = [];

var geocoder, location1, location2;

var directionPanel;

var Magasins=new Array();

var missing=new Array();

var Distances=new Array();

var Lati=new Array();

var Lon=new Array();

var images=new Array();


function clickedAddAddress2(){

        var result;

        geocoder = new google.maps.Geocoder();

        var compte = 0;

        if ((adr != null) && (adr != "undefined") && (adr != "")){

                directions(0, document.forms['travelOpts'].walking.checked, _

                        document.forms['travelOpts'].avoidHighways.checked);

        }

        else{

                alert("Vous devez donner une adresse de d\351part.");

        }

}
```

```
function addMarker(location, num){
        if(num == 1){
                letter="r";
        }
        else{
                letter="b";
        }
        marker = new google.maps.Marker({
                        position: location,
                        map: map,
                        icon:"icons/icon" + letter + num + ".png",
                        zIndex: 10000000089
                        });

        markersArray.push(marker);
}


function clearOverlays(){
        if (markersArray){
                for (i in markersArray){
                        markersArray[i].setMap(null);
                }
                markersArray = [];
        }
}
```

```
function showOverlays(){
        if (markersArray){
                for (i in markersArray){
                        markersArray[i].setMap(map);
                }
        }
}

function initialize(){
        directionsDisplay = new google.maps.DirectionsRenderer();
        directionsService = new google.maps.DirectionsService();
        var latlng = new google.maps.LatLng(45.50154, -73.55071);
        var myOptions = {
                zoom: 12,
                center: latlng,
                mapTypeId: google.maps.MapTypeId.ROADMAP
        };
        map = new google.maps.Map(document.getElementById("map"),myOptions);
        directionsDisplay.setMap(map);
        directionsDisplay.setPanel(document.getElementById("path"));
}

function directions(m, walking, avoid){
        clearOverlays();
```

```
var WPoints = new Array();
for (t = 0; t < pieces.length; t++){
        WPoints[t]={location: lat[pieces[t]-2]+", "+lng[pieces[t]-2],_
                stopover: true};
}
if(walking){
        var mode = google.maps.DirectionsTravelMode.WALKING
}
else{
        var mode = google.maps.DirectionsTravelMode.DRIVING
}
var request = {
        origin:adr,
        destination:adr,
        waypoints: WPoints,
        optimizeWaypoints: true,
        avoidHighways: avoid,
        travelMode: mode,
        unitSystem: google.maps.DirectionsUnitSystem.METRIC
};
directionsService.route(request, function(result, status) {
        if (status == google.maps.DirectionsStatus.OK) {
                directionsDisplay.setDirections(result);
        }
});
```

```
showOverlays();

var decalage = 0.0000;

var myLatLng = new google.maps.LatLng(latitudeH-decalage, longitudeH);

addMarker(myLatLng,1987);


showOverlays();


var myLatLng;

var marker;

for (I = 0; I < pieces.length; i++){

        if(img[pieces[i]-2] == "defaut.jpg"){}

        else{

                myLatLng = new google.maps.LatLng(lat[pieces[i]-2]-decalage, _

                        lng[pieces[i]-2]);

                marker = new google.maps.Marker({

                        position: myLatLng,

                        map: map,

                        icon: "admin/images/"+img[pieces[i]-2],

                        zIndex: 10000000090+i

                });

                markersArray.push(marker);}

        }


        showOverlays();
```

```
        }

        function computeTotalDistance(result){
                var total = 0;
                var myroute = result.routes[0];
                for (i = 0; i < myroute.legs.length; i++){
                        total += myroute.legs[i].distance.value;
                }
                total = total / 1000;
                document.getElementById("disttot").innerText = _
                        document.getElementById("disttot").textContent=total + " km.";
        }

        function computeTotalTime(result){
                var total = 0;
                var myroute = result.routes[0];
                for (i = 0; i < myroute.legs.length; i++){
                        total += myroute.legs[i].duration.value;
                }
                total = Math.round(total / 60.);
                document.getElementById("totalt").innerText = _
                        document.getElementById("totalt").textContent=total + " min.";
        }
        </script>
</head>
```

```
<body>
	<script language="JavaScript">
	$(document).ready(function() {
		initialize();
		clickedAddAddress2();
		google.maps.event.addListener(directionsDisplay, 'directions_changed',_
			function(){
				computeTotalTime(directionsDisplay.directions);
				computeTotalDistance(directionsDisplay.directions);
			});
		});
	</script>
	<div id="div1" style="width: 100%; float:left;font-weight:900; font-
size:15px">
	</div>
	<div style="width: 100%; float:left;">
		<form name="travelOpts">
			<table width=100% >
			<td>
			<table>
			<td>
				<input id="walking"
				type="checkbox"
				onClick="directions(0,
				document.forms['travelOpts'].walking.checked,
```

```
                        document.forms['travelOpts'].avoidHighways.checked)">
                        <?php echo $pied;?>
                        </input>
                </td>
                <td>
                        <input id="avoidHighways"
                        type="checkbox"
                        onClick="directions(0,
                        document.forms['travelOpts'].walking.checked,
                        document.forms['travelOpts'].avoidHighways.checked)">
                        <?php echo $eviterAutoroutes;?>
                        </input>
                </td>
                </table>
                </td>
                </table>
        </form>
        <table class="buttonTable">
                <tr></tr>
        </table>
</div>
<div id="map" style="width: 98%; height: 480px;
        float:left; border: 1px solid black;">
</div>
<br/>
```

```
        <div>
                <form name="address" onSubmit="clickedAddAddress2(); return false;">
                </form>
        </div>
        <p>
                <?php echo $tempsTotal;?><span id="totalt"></span><br>
                <?php echo $distanceTotal;?><span id="disttot"></span>
        </p>
        <div id="path"style="width: 98%; border; 1px solid black; float:left;"></div>
</body>
</html>
```

BIBLIOGRAPHIE

[AVV92] Serge Abiteboul, Moshe Y. Vardi, et Victor Vianu. Fixpoint logics, relational machines, and computational complexity. In *Structure in Complexity Theory Conference, 1992., Proceedings of the Seventh Annual*, pages 156 –168, Juin 1992.

[BAF10] Boris Bontoux, Christian Artigues, et Dominique Feillet. A memetic algorithm with a large neighborhood crossover operator for the generalized traveling salesman problem. *Computers & Operations Research*, 37:1844–1852, Novembre 2010.

[BM02] Arash Behzad et Mohammad Modarres. A new efficient transformation of the generalized traveling salesman problem into traveling salesman problem. In *Proceedings of the 15th International Conference of Systems Engineering (ICSE)*, Las Vegas, Nevada, Août 2002.

[Bon95] John Adrian Bondy. *Basic graph theory: paths and circuits*, pages 3–110. MIT Press, Cambridge, MA, USA, 1995.

[DS97] Vladimir Dimitrijevic et Zoran Saric. An efficient transformation of the generalized traveling salesman problem into the traveling salesman problem on digraphs. *Information Sciences*, 102:105–110, Novembre 1997.

[FGT97] Matteo Fischetti, Juan Jose Salazar Gonzalez, et Paolo Toth. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. *Operations Research*, 45(3):pp. 378–394, 1997.

[GJS74] Michael R. Garey, David S. Johnson, et Larry Stockmeyer. Some simplified np-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[Goo11] Google. 2011. «Google Maps Javascript API V3 Reference». In Google Code. En ligne. <http://code.google.com/apis/maps/documentation/javascript /reference.html>. Consulté le 13 septembre 2011.

[GK08] Gregory Gutin et Daniel Karapetyan. Generalized traveling salesman problem reduction algorithms. *Computing Research Repository*, abs/0804.0735, 2008.

[GK10] Gregory Gutin et Daniel Karapetyan. A memetic algorithm for the generalized traveling salesman problem. 9:47–60, Mars 2010.

[KG10a] Daniel Karapetyan et Gregory Gutin. Lin-kernighan heuristic adaptation for the generalized traveling salesman problem. *Computing Research Repository*, abs/1003.5330, 2010.

[KG10b] Daniel Karapetyan et Gregory Gutin. Local search algorithms for the generalized traveling salesman problem. *Computing Research Repository*, abs/1005.5525, 2010.

[Lap92] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, Juin 1992.

[LKB08] Yifang Li, Yannick Kergosien, et Jean-Charles Billaut. 2008 (16 octobre). «Le problème du voyageur de commerce». In *Interstices*. En ligne. <http://interstices.info/jcms/c_37686/le-probleme-du-voyageur-de-commerce>. Consulté le 4 septembre 2011.

[MP10] Oliviu Matei et Petrica Pop. An efficient genetic algorithm for solving the generalized traveling salesman problem. In *Intelligent Computer*

Communication and Processing (ICCP), 2010 IEEE International Conference, pages 87 –92, Août 2010.

[Ora11] Oracle. 2011. «MySQL 5.5 Reference Manual». In MySQL. En ligne. <http://dev.mysql.com/doc/refman/5.5/en/>. Consulté le 13 septembre 2011.

[PTL08] Quan-Ke Pan, Mehmet Fatih Tasgetiren, et Yun-Chia Liang. A discrete particle swarm optimization algorithm for the no-wait flowshop scheduling problem. *Computers & Operations Research*, 35(9):2807 – 2839, 2008. Part Special Issue: Bio-inspired Methods in Combinatorial Optimization.

[RB98] Jacques Renaud et Fayez Boctor. An efficient composite heuristic for the symmetric generalized traveling salesman problem. *European Journal of Operational Research*, (108):571–584, 1998.

[RSLI77] Daniel J. Rosenkrantz, Richard E. Stearns, et Philip M. Lewis. An analysis of several heuristics for the traveling salesman problem. 6(3):563–581, 1977.

[SD04] Lawrence V. Snyder et Mark S. Daskin. A random-key genetic algorithm for the generalized traveling salesman problem. *European Journal of Operational research*, 174, 2004.

[SG07] John Silberholz et Bruce Golden. The generalized traveling salesman problem: A new genetic algorithm approach. In *Extending the Horizons: Advances in Computing, Optimization, and Decision Technologies*, volume 37 of *Operations Research/Computer Science Interfaces Series*, pages 165–181. Springer US, 2007.

[TSPL07] Mehmet Fatih Tasgetiren, Ponnuthurai Nagaratnam Suganthan, Quan-Ke Pan, et Yun-Chia Liang. A genetic algorithm for the generalized traveling salesman problem. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 2382 –2389, Septembre 2007.