Université de Bordeaux

Département Informatique

Environnement de développement & Projet de Programmation 1

Problème du Voyageur de Commerce

Auteurs :
David Phan
Jason Pindat
Nicolas Marcy
Raphaël Afanyan

Encadrant :
Irène Durand

19 avril 2014

Table des matières

Chapitre 1

Introduction

1.1 Présentation

Dans le cadre de ce projet, on s'intéresse au problème classique du *voyageur de commerce* qui consiste à, étant donné un ensemble de villes séparées par des distances données, trouver le *cycle hamiltonien*, chemin passant par toutes les villes, une fois et une seule. Il s'agit d'un problème d'optimisation pour lequel on ne connait pas d'algorithme permettant de trouver une solution exacte en un temps polynomial bien qu'on puisse vérifier rapidement toute solution proposée, c'est un problème NP-complet. En revanche, de nombreux autres algorithmes existent afin de résoudre ce problème plus ou moins efficacement. Il nous a été demandé d'écrire au moins 4 algorithmes permettant, à partir d'un fichier TSP, de trouver une solution et d'en donner la longueur pour pouvoir la comparer avec les autres. Au total, 8 différentes versions et implémentations de ces algorithmes ont été réalisées.

1.2 Notre programme

Notre programme répond donc au cahier des charges initial auquel nous avons ajouté d'autres options :

- un algorithme avec une heuristique simple (*Nearest Neighbour*).
- un algorithme d'approximation (*Minimum Spanning Tree*).
- un algorithme exact par recherche exhaustive (plusieurs versions différentes en recherchant le meilleur temps d'exécution).

— un algorithme exact utilisant la technique de "*Branch and Bound*", et une version comprenant une relaxation avec *Nearest Neighbour* et *Minimum Spanning Tree*, une autre avec relaxation de *Held-Karp*
— pouvoir charger des instances **TSPLIB** (.tsp) codées sous la forme de matrices complètes.
— pouvoir retourner les solutions dans le format **TSPLIB** (.tour).
— effectuer des tests de performance.
— visualiser graphiquement la route calculée (en utilisant **DISPLAY_DATA_SECTION**)
— retourner les données et traitements des instances faits par le programme dans un format générique (**JSON**).

Chapitre 2

Le *Traveling Salesman Problem Solver*

Nous avons nommé notre programme *Traveling Salesman Problem Solver*, qui est une traduction à Résolveur du Problème du Voyageur de commerce.

2.1 Fonctionnalités

Le programme peut s'utiliser en deux modes : le mode console, et le mode graphique. Les deux modes présentent les fonctionnalités suivantes :

- Ouvrir et instancier un ou des fichiers **TSPLIB** (matrice complète)
- Résoudre avec un ou plusieurs algorithmes les fichiers instanciés en pouvant indiquer l'indice de la ville de départ
- Sauvegarder les résultats fournis au format **TSPLIB**
- Générer et instancier plusieurs problèmes avec des données aléatoires
- Afficher et visualiser les instances résolues

En plus de cela, le mode console a la capacité de :

- Définir le mode de calcul des longueurs (*Manhattan* ou Euclidien)
- Générer des fichiers **TSPLIB** d'instances aléatoires
- Faire des batteries de tests de performance
- Sauvegarder au format **JSON** (mode *API*) toutes les données d'une instance

2.2 Documentation

2.2.1 Compilation

Pour compiler le TSP Solver, rendez-vous dans le dossier trunk/ et effectuez la commande *make* L'exécutable se trouvera alors dans le dossier /bin Vous pourrez le lancer via la commande *./VDC*

Pour afficher l'aide concernant les options du programme tapez *./VDC -h*

Afin de vérifier le bon fonctionnement de la compilation, vous pouvez lancer des tests de lecteur de TSP, et de résolution selon des algorithmes. Pour cela, placez-vous dans le répertoire *test/* et lancez la commande *cmake .* suivi de la commande *ctest*.

2.2.2 Mode console

Résolution d'instances

Pour résoudre une instance instance10.tsp avec une City de départ 4 simplement avec *nearest Neighbour*, vous devez lancer la commande
./VDC -nn instance10.tsp 4
(n'oubliez pas de bien spécifier le chemin où se trouve le TSP, dans notre projet en l'occurence ../tsp/instance10.tsp)

Pour résoudre plusieurs instances instance10.tsp et instance16.tsp, avec 2 et 3 en City de départ, avec *Minimum Spanning Tree*, il faut lancer
./VDC -mst instance10.tsp 2 instance16.tsp 3

Pour des options plus détaillées, veuillez lire la suite.
Si une City de départ n'est pas définie, alors la 1ère ville sera utilisée.

Choix des algorithmes

Il y a sept paramètres pour choisir le ou les algorithmes à lancer.
— *-all* permet de lancer tous les algorithmes.
— *-nn* permet de lancer *Nearest Neighbour*.
— *-mst* permet de lancer *Minimum Spanning Tree*.
— *-bf* permet de lancer *BruteForce*.
— *-bfrec* permet de lancer *BruteForce* Recursif.

— *-bfmt* permet de lancer *Bruteforce* Multithreadé.
— *-bb* permet de lancer *Branch and Bound*.
— *-bbr* permet de lancer *Branch and Bound* avec Relaxation *Nearest Neighbour* et *Minimum Spanning Tree*
— *-bbrhk* permet de lancer *Branch and Bound avec Relaxation Held Karp*

Exemple : Branch and bound et MST pour deux fichiers : *./VDC -bb -mst fichier1.tsp fichier2.tsp 4*

Génération et enregistrement d'instances aléatoires

Nous pouvons lancer une instance aléatoire grâce à l'option *-r <Nombre1> <Nombre2>* Nombre1 étant la taille de l'instance que vous souhaitez générer, Nombre2 la ville de départ, par défaut si Nombre2 n'est pas mentionné, la ville de départ est la numéro 1.

Exemple : *./VDC -r 10* génèrera seulement une instance aléatoire de taille 10.
Pour les traiter il suffit d'ajouter les commandes habituelles.
Par exemple : *./VDC -all -r 10*

Pour générer un fichier **TSPLIB** de nom *fichiersortie.tsp* aléatoire de taille *N*, vous pouvez utiliser l'option *-o fichiersortie.tsp*.
./VDC -r 10 -o <fichiersortie>.tsp

Mode de calcul des longueurs

Pour les instances aléatoires, nous générons les positions des villes aléatoirement, pour générer les matrices de distances, nous pouvons modifier le mode de calcul des distances.
Pour calculer avec la longueur de Manhattan, utilisez l'option *-lm*. Pour calculer avec la longueur Euclidienne, utilisez l'option *-le*. Au cas où les fichiers TSPLIB n'ont pas de matrice de distances, ce qui ne doit pas être le cas, mais qui ont leur **DISPLAY_DATA_SECTION** (coordonnées des points), alors la matrice de distances pourra être calculée via la valeur de cette option. Par défaut si rien n'est mentionné, le calcul des distances est Manhattan.

Génération du fichier résultat Tour

Pour sauvegarder le meilleur résultat des algorithmes lancés à l'instance, vous pouvez utiliser l'option *-to* qui sortira un fichier au format **TSPLIB**,

dans le même dossier que le fichier TSP avec l'extension *.tour*, le programme sélectionnera l'algorithme qui a été le plus performant parmis ceux éxécutés. Pour la génération des fichiers des instances aléatoires, le nom du fichier sera généré à partir de la date actuelle. Exemple : *Random_Map_2014_04_15_18_50_21.tour*

Lancement de tests de performances

Vous pouvez lancer des tests de performances d'instance aléatoires de tailles que vous définirez. Vous devez donner en paramètrer le nombre de tests, le nombre de villes, et les algos que vous souhaitez tester.

Exemple : **./VDC -t 100 20 -nn -mst -bbrhk**
Cela lancera 100 fois les algorithmes sélectionnés, de 3 à 20 villes et afficher les résultats de temps.

Génération d'un fichier au format JSON d'une instance

Pour obtenir un fichier d'une instance au format JSON, vous devez ajouter l'option *-api*.

Exemple *./VDC -api -nn fichier10.tsp 5*
Par la suite, un fichier sera généré du nom de fichier10.tsp_nn.json (pour l'algorithme Nearest Neighbour)
Pour les autres algorithmes :
fichiers10.tsp_mst.json, fichiers10.tsp_bf.json, fichiers10.tsp_bfrec.json, fichiers10.tsp_bfmt.json, fichiers10.tsp_bb.json. stacj

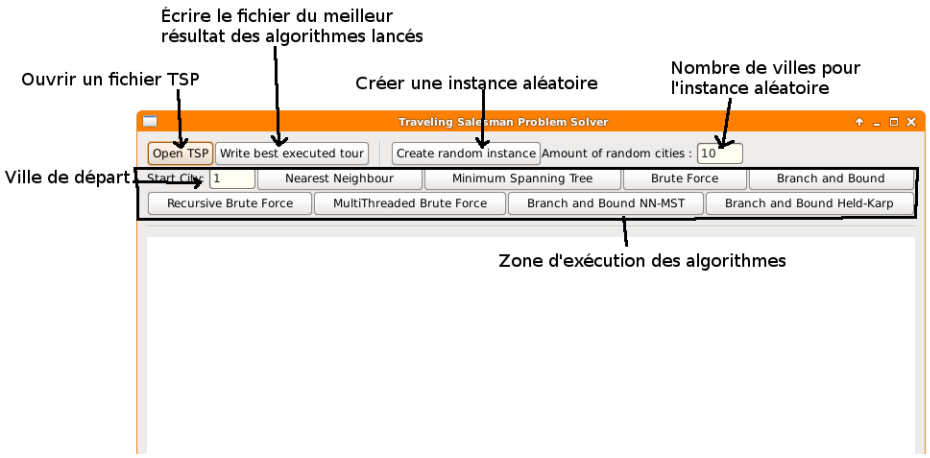
Mode graphique console

Vous pouvez afficher le mode graphique textuel avec l'option *-gt*
Exemple : *./VDC -gt -nn fichier10.tsp fichier15.tsp*

2.2.3 Mode graphique

Lancer le mode graphique

Vous pouvez lancer le mode graphique avec *./VDC -g*



Vous pouvez lancer le mode graphique avec un fichier préchargé en ajoutant le nom du fichier
./VDC -g fichier1.tsp

Charger une instance

Cliquez sur "Open TSP" et choisissez le fichier TSPLIB à charger.

Générer une instance aléatoire

Remplissez l'entrée de texte "*Amount of random Cities*" avec la taille de l'instance aléatoire que vous souhaitez générer. En suite, cliquer sur le bouton "*Create Random Instance*".

Lancer un algorithme

Vous devez avoir ouvert une instance. Remplissez l'entrée de texte "Start City" avec un nombre entier compris entre 1 et la taille de l'instance chargée, puis cliquez sur un des boutons choisissant l'algorithme que vous souhaitez lancer.

Il est possible de sauvegarder le résultat de vos algorithmes au format **TSPLIB** (.tour) : Executez les algorithmes de votre choix puis cliquez sur "Write best executed tour", le programme selectionnera le résultat de l'algorithme ayant réalisé le plus court chemin et exportera le fichier.

2.2.4 Mettre en route le site web TSP

Pour que le site web PHP disponible dans le dossier **website**/ fonctionne :
Il faut que le dossier **uploads**/ soit en permission **777**.
Il faut que l'executable compilé soit dans le dossier **./**.
Avoir la fonction **shell_exec** de **PHP** activée.

Chapitre 3

Architecture du *Traveling Salesman Problem Solver*

3.1 Introduction

Nous avons défini le problème par une instance *Map* qui contiendrait des instances *City*.
Chaque instance City contient un tableau de distances relatives aux autres *City* de la *Map*, de plus elle contient un Point si ses coordonnées sont explicitées dans le fichier TSP. Un point est défini par ses coordonnées (x,y).

Pour faciliter l'implémentation du reste du programme et la compréhension du code, notamment des algorithmes, nous avez veillé à utiliser des prototypes de fonction facilement assimilables et le plus explicite possible. Ainsi par exemple, les fonctions des modules *City* et *Map* contiennent toutes le préfixe associé à leur nature. Nous avons bien entendu opté pour le choix de les nommer en anglais pour une question d'universalité.

L'exécution des algorithmes ainsi que leur implémentation sont faites dans le module *Algos*.
Pour ouvrir, enregistrer les fichiers TSPLIB et exporter les résultats, un module *TSP* a été réalisé.

Nous avons implémenté une interface graphique (*GUI*) à l'aide de la librairie *GTK+*.

Nous sommes allés plus loin avec le module API, qui sert à écrire le ré-

sultat dans fichier au format générique *JSON*, qui est lisible et peut être facilement représenté et analysé dans n'importe quel langage de programmation.
Nous avons implémenté un petit site en *PHP* qui exploite nos fichiers au format *JSON*.

3.2 *main*

La base du programme contient les fonctions d'usage, d'aide et de gestion d'options, il permet l'intéraction facilitée entre l'utilisateur et les fonctions implementées dans les autres modules.

3.3 Module *point*

Le module point sert à définir les City dans un repère cartésien, si **DISPLAY_DATA_SECTION** est dans le fichier d'instance.
Il sert aussi à calculer les distances entre deux points.

3.3.1 Coordonnées

Un point est défini par ses coordonnées cartésiennes, qui sont deux nombres de type double : x et y.

3.3.2 Fonctions

Les fonctions que nous avons définis sont :
- setLengthType(bool) qui défini le mode de calcul de distance
- lengthEuc(Point,Point) qui calcule et renvoie la distance euclidienne entre deux points
- lengthMan(Point,Point) qui calcule et renvoie la distance de Manhattan entre deux points
- length(Point,Point) qui renvoie la distance selon le mode défini
- pointGetX(Point) qui renvoie l'abscisse du Point
- pointGetY(Point) qui renvoie l'ordonnée du Point
- boolEquals(Point,Point) qui renvoie l'égalité de deux points

— point(double,double) qui crée et renvoie un point

3.4 Module *city*

Une ville (*City*) est définie par son tableau de distance avec les autres villes, et par un *Point* optionnellement.

3.4.1 Tableau de distance *dists*

Le tableau de distances *dists* est un tableau d'une dimension du même nombre de villes que la *Map*.

3.4.2 Position *point*

Si *isPos*, la condition qui dit si la ville est définie aussi par un Point, est à VRAIE, alors la ville a un Point défini dans sa structure.

3.4.3 Fonctions

— cityCreate instancie une "ville" caractérisée par sa position et sa distance avec les autres villes.
— cityDelete détruit une instance City
— cityGetDist qui permet d'obtenir la distance d'une *City* avec une autre
— cityGetDistsSize retourne le nombre de distances que contient une *City* c
— cityGetIsPos indique si un Point est défini par des coordonnées ou non.
— cityGetPos si cityGetIsPos est vraie, fournit les coordonnées de la *City* (*Point*)
— cityEquals indique si 2 objets de type *City* sont les mêmes. (comparaison des tableaux de distance)
— cityDataDump affiche toutes les informations connues sur une *City*.
— cityGetIndice retourne l'indice de la *City*.

3.5 Module *map*

La structure *Map* est définie par un tableau de *City*
La structure Map, étant une instance du problème, nous avons ajouté la sauvegarde des résultats à l'intérieur même de cette structure.

3.5.1 Nom de l'instance *name*

Elle est définie par le nom du fichier instancié, ou de Random_Map_ si c'est une instance aléatoire.
Elle sert par exemple à nommer le nom du fichier de résultat.

3.5.2 Tableau d'objets de type City *cities*

C'est un tableau extensible qui contient les villes de type City
On utilise nbCitiesMax pour connaître la taille max du tableau (pour la réallocation), ainsi que nbCities pour connaître la taille de l'instance.

3.5.3 Sauvegarde des résultats

Etant donné que Map est une instance de problème, nous avons besoin de sauvegarder les résultats dedans pour pouvoir sauvegarder la meilleure solution (le plus court chemin) calculée lors des appels d'un ou plusieurs algorithmes.

Tableau de chemins *paths*

Prototype : City **paths
Un résultat est un chemin, un chemin est un tableau de City.
Nous utilisons le tableau de chemins pour pouvoir générer le fichier de résultat du meilleur algorithme lancé.
Il nous sert aussi dans l'interface graphique, pour ne pas recalculer des algorithmes déjà exécutés auparavant. Paths contient un chemin pour chaque algorithme.

Tableau de durées d'exécutions *duration*

Nous avons défini un tableau de réels (double) pour pouvoir sauvegarder les temps d'exécution des algorithmes avec leur chemin pour l'interface

graphique.

3.5.4 Les fonctions

Les fonctions implementeés pour intéragir avec la Map sont les suivantes :
— mapCreate() qui renvoie une Map vierge.
— mapGetName(Map) qui renvoie le nom de la Map.
— mapSetName(Map, Str) qui modifie le champ name de la Map.
— mapGetPath(Map, int) qui renvoie le tableau de City pour l'algorithme spécifié par son numéro.
— mapSetPath(Map, int, City *) qui modifie le champ paths[int] par celui passé en paramètre
— mapGetDuration(Map, int) qui renvoie la durée de calcul de l'algorithme spécifié par son numéro.
— mapSetDuration(Map, int, double) qui modifie la durée de calcul
— mapDelete(Map) qui supprime la Map
— mapDeleteRec(Map) qui supprime la Map et récursivement ses City.
— mapAddCity(Map,City) qui ajoute une ville à une Map
— mapGetCity(Map,int) qui renvoie une ville par son indice
— mapSetCity(Map,City, int) qui remplace une ville d'indice spécifié par une autre
— mapGetSize(Map) qui renvoie la taille de l'instance (le nombre de City)
— mapGetIndiceCity(Map, City) qui renvoie l'indice de la City de la Map
— mapGetIsPos(Map) qui renvoie si les City de la map sont définis par des points
— mapDataDump(Map) qui affiche en mode verbeux tout le contenu de la Map
— mapCreateFromPoints(Point *, int , Str) crée une Map et ses villes en calculant les tableaux de distances à partir du tableau de Point
— mapCreateRandom(int) qui renvoie une Map créée aléatoirement avec le nombre spécifié de City
— mapDraw(Map) dessine en mode textuel les points de la Map

3.6 Algorithmes

3.6.1 Nearest Neighbour

Présentation de Nearest Neighbour

Le problème de la recherche des plus proches voisins *Nearest Neighbour* (ou des k plus proches voisins) est très courante en algorithmique et de nombreux auteurs ont proposé des algorithmes efficaces pour le résoudre rapidement.

C'est un algorithme avec une heuristique simple, aussi appelé algorithme glouton, le principe est qu'il fait un choix à chaque étape, sans jamais remettre en cause ce choix.

Soient :

— un espace E de dimension D ;

— un ensemble A de N points dans cet espace ;

— un entier k plus petit que N.

La recherche des plus proches voisins consiste, étant donné un point x de E n'appartenant pas nécessairement à A, à déterminer quels sont les k points de A les plus proches de x. On parle alors de trouver un voisinage de taille k autour du point x.

Algorithme

```
fonction nearestNeighbour {
pour i allant de 1 à k
    mettre le point D[i] dans proches_voisins
fin pour
pour i allant de k+1 à N
    si la distance entre D[i] et x est inférieure à la distance d'un des points
de proches_voisins à x
        supprimer de proches_voisins le point le plus éloigné de x
        mettre dans proches_voisins le point D[i]
    fin si
fin pour
proches_voisins contient les k plus proches voisins de x
```

}

L'algorithme Nearest Neighbour a été implémenté en complexité de temps $\theta(n^2)$

3.6.2 Minimum Spanning Tree

Présentation de Minimum Spanning Tree

En théorie des graphes, étant donné un graphe non orienté connexe dont les arêtes sont pondérées, un arbre couvrant de poids minimal *minimum spanning tree* de ce graphe est un arbre couvrant (sous-ensemble qui est un arbre et qui connecte tous les sommets ensemble) dont la somme des poids des arêtes est minimale. L'arbre couvrant de poids minimal est aussi connu sous certains autres noms, tel qu'arbre couvrant minimum ou encore arbre sous-tendant minimum.

Implémentation de Minimum Spanning Tree

L'arbre couvrant de poids minimal est implémenté par une structure arbre sous forme de liste chaînée. Cette structure contient aussi un tableau des sommets de l'arbre pour un rajout de sommet plus facile. Le chemin est ensuite généré en parcourant cette arbre.

L'algorithme a été implemenété avec une complexité de temps $\theta(n^3)$

3.6.3 Recherche exhaustive *BruteForce*

La recherche exhaustive ou recherche par force brute est un terme qui s'applique à une catégorie de méta-algorithmes.
Elle calcule toutes les possibilités de chemins possible.
Soit une complexité de l'ordre $\theta(n!)$

Version itérative

Nous avons implementé une version itérative de Bruteforce, en effet, une version récursive est plus lente car elle utilise la pile d'exécution.

Nous utilisons un algorithme de Permutation non ordonnée *QuickPerm*, du type :

```
fonction permutation()
    entier nombre ville N
    tableau d'entiers t[N] initialisé avec (1,2,..., N) , p[N] initialisé à 0
    entier i, j, tmp
    i=1
    tantque(i<N)
        si(p[i]<i)
            j=i%2*p[i]
            tmp = a[j]
            a[j] = a[i]
            a[i] = tmp
            p[i]++
            // on calcule la distance à ce moment là
        sinon
            p[i]=0
            i++
        finsi
    fintantque
}
```

1	2	3
2	1	3
3	1	2
1	3	2
2	3	1
3	2	1

Exemple de permutations avec 1,2,3

Version récursive

La version récursive, est plus facile à implémenter, mais elle utilise la pile d'exécution.
Le plus dans cette méthode, c'est qu'elle fait les permutations ordinales.

L'algorithme de l'université d'Exeter a été implémenté.

```
fonction permutation(tableau de villes v, entier debut, entier n) {
si (debut == n-1)
    // on a le chemin complet
sinon
    pour (entier i = debut ; i < n ; i++)
        entier tmp = v[i] ;

        v[i] = v[debut] ;
        v[debut] = tmp ;
        permutation(v, debut+1, n) ;
        v[debut] = v[i] ;
        v[i] = tmp ;
    finpour
finsi
}
```

Version multi-threadée

Nous avons exprimé le besoin de paralleliser le bruteforce, pour peut être gagner un peu de temps.
Nous avons donc décomposé le problème d'une Map de N Cities, à N-1 sous-problème, que nous enverrons à un thread respectivement.

1 (Ville de départ)	2	3	4	5
Sous problème 1	2	3	4	5
Sous problème 2	3	2	4	5
Sous problème 3	4	2	4	5
Sous problème 4	5	2	3	4

Cela, en pratique, nous diminue le temps (réel) par le nombre de threads lancé.

3.6.4 Séparation et évaluation *Branch and Bound*

Présentation

Un algorithme par séparation et évaluation, également appelé selon le terme anglophone branch and bound, est une méthode générique de résolution de problèmes d'optimisation, et plus particulièrement d'optimisation combinatoire ou discrète. C'est une méthode d'énumération implicite : toutes les solutions possibles du problème peuvent être énumérées mais, l'analyse des propriétés du problème permet d'éviter l'énumération de larges classes de mauvaises solutions. Dans un bon algorithme par séparation et évaluation, seules les solutions potentiellement bonnes sont donc énumérées.

Algorithme natif

L'algorithme que nous avons implémenté est basé sur l'algorithme Bruteforce, version récursive.
Elle nous permettra de stopper la récursion dès que la distance de la branche déjà générée.

```
fonction permutation(tableau de villes v, entier debut, entier n) {
si (debut == n-1)
    // on a le chemin complet
    si (longueur(v)<pluspetitcheminactuel)
        pluspetitcheminactuel=longueur(v)
    finsi
sinon
    pour (entier i = debut ; i < n ; i++)
        entier tmp = v[i] ;

        v[i] = v[debut] ;
        v[debut] = tmp ;
        si (longueur(v) > pluspetitcheminactuel)
            interruption
        finsi
        permutation(v, debut+1, n) ;
        v[debut] = v[i] ;
        v[i] = tmp ;
    finpour
```

finsi
}

Algorithme relaxé avec Nearest Neighbour et Minimum Spanning Tree

L'algorithme a juste une initialisation de plus petit chemin actuel (lower bound) au minimum des distances des résultats de Nearest Neighbour et Minimum Spanning Tree.

Nous avons décidé d'initialisé par les deux valeurs, car :

— Étant donné qu'en pratique (sur des instances variants de 10 à 100 villes) Nearest Neighbour donne des résultats plutôt meilleurs que Minimum Spanning Tree.

— Minimum Spanning Tree donne un résultat maximum à deux fois l'optimal, du coup cela sera borné.

Algorithme relaxé de Held Karp

Pour toutes les arêtes e, avec w_e la longueur de l'arête e, et la variable x_e est à 1 si l'arête e est dans le tour, sinon 0. Pour tous les sous-ensembles S d'un sommet, $\delta(S)$ définissent les arêtes connectants un sommet dans S avec un sommet extérieur à S.

1. pour tous les sommets v, sommer$_{\text{arêtes e dans } \delta(v)}$ $x_e = 2$
2. pour tous les sous-ensembles non vides S de sommets, sommer$_{\text{arêtes e dans } \delta(S)}$ $x_e >= 2$
3. pour toutes les arêtes e dans E, x_e appartient à $\{0, 1\}$

Il faut que l'ensemble des arêtes soit un groupement de tour (1), et qu'il n'y en a qu'une (2). (Sinon il faudra définir S comme étant un ensemble de sommets visités par un seul des tours).

3. pour toutes les arêtes e dans E, $0 <= x_e <= 1$

Nous appliquons les optimisations des multiplicateurs de Lagrange, puis l'optimisation de l'algorithme du gradient. Cela se résume à calculer un minimum spanning tree et en suite modifier certaines directions via le calcul de one-tree.

L'algorithme de *Branch and Bound* a été implémenté via un tas Minimum de solutions partielles triées par la valeur de *Held Karp* de ces solutions.

Soit h étant un tas min de solutions partielles, triée par *Held-Karp*
soit $meilleursolution$ = null
soit *solactuelle* la solution partielle de *Held-Karp*

tantque *solactuelle* n'est pas une solution complète et la valeur de *Held-Karp* de *solactuelle* est meilleure que $meilleursolution$
choisir un embranchement v
soit $sol0$ la connexion avec solactuelle $\{v- > 0\}$
soit $sol1$ la connexion avec solactuelle $\{v- > 1\}$
calculer $sol0$ et $sol1$
soit *solactuelle* la meilleure des deux solutions $sol0$ et $sol1$; mettre la moins bonne dans le tas Min h
fintantque
si solactuelle est meilleure que meilleursolution alors
soit meilleursolution = solactuelle
supprimer tous les vertex du tas Min h ayant une valeur de HK pire que *solactuelle*
finsi
si h est vide alors arrêter ; // nous avons trouver la solution optimale
récupérer le sommet du tas Min h et le mettre dans *solactuelle*

3.7 Module *tsp*

Il nous a été demandé d'implémenter un lecteur de fichiers TSP au format TSPLIB.

- Map tspLoad(filename) => Prend un nom de fichier en paramètre et retourne un objet Map
- void tspWrite(Map, Str) => Prend un objet Map et un nom de fichier en paramètre et écrit le fichier résultant
- void tspOut(Map, City*, Str, Str) => Prend une Map, un parcours, un nom de fichier et un commentaire en paramètre et écrit le résultat dans le fichier

3.7.1 Module *string*

C'est une librairie qu'a écrit Jason pour un autre projet, elle décrit des fonctions de gestion des chaines de caractères plus évoluées que celles prévues de base, comme par exemple indexOf(), fonction native en JAVA.

3.7.2 Ouverture de fichiers TSPLIB

Le fichier est lu ligne par ligne et sauvegarde les informations petit à petit, si tout s'est bien passé en fin de lecture il crée un objet Map qu'il remplit avec les City nécessaires et le renvoie, prêt à l'emploi. Le lecteur est capable de lire des fichier TSP qui ne comportent pas de DISPLAY_DATA_SECTION, dans ce cas il n'y aura pas d'affichage possible, à l'inverse le lecteur lit les fichiers qui ont le DISPLAY_DATA_SECTION mais pas le EDGE_WEIGHT_SECTION, dans ce cas le lecteur va calculer les distances automatiquement selon le mode choisi avec les options (Manhattan par défaut) et retournera une Map conforme à l'emploi. Ex d'utilisation : *./VDC -nn toto.tsp*

3.7.3 Sauvegarde de fichiers TSPLIB

C'est la fonction inverse de la lecture, à partir d'une Map la fonction va générer un fichier TSP, cette fonctionnalité est utilisé pour rendre des TSP à partir de maps aléatoires. Ex d'utilisation : *./VDC -r 10 -o toto.tsp*

3.7.4 Sauvegarde des résultats au format TSPLIB

Cette fonction sert a rendre des fichiers TSP TOUR, c'est-à-dire des chemins à parcourir selon les résultats d'un ou plusieurs algorithmes à partir d'une Map, d'une liste de villes, d'un nom de fichier et d'un commentaire.Ex d'utilisation : *./VDC toto.tsp -all -to*

3.8 Module *tree*, *vertex*, *File de priorité pri_ queue*

Nous avons du implémenter des modules complémentaires à notre structure de base pour permettre l'implémentation de certains algorithmes.

3.8.1 tree

C'est un module qui permet de créer un arbre planaire et de le parcourir. Il a été implémenté pour l'algorithme Minimum Spanning Tree. Nous avons un accès au père et aux fils. Il contient aussi un tableau avec toutes les adresses des sommets dans l'arbre.

3.8.2 vertex

C'est un module qui implémente un vertex (sommet). Nous l'avons implémenté pour l'algorithme de branch and Bound. Il permet entre autres :

- d'accéder au tableau d'exclusions d'arêtes
- d'accéder à la distance ajustée
- d'accéder au lower bound (la distance minimale)
- d'accéder au degré
- d'accéder au parent

3.8.3 priority queue

C'est un module d'implémentation d'une file de priorité. Elle nous permet de l'utiliser en tant que tas min. On peut entre autres :

- ajouter un vertex au tas min
- connaître la valeur du tas min
- enlever le sommet du tas min
- combiner deux tas min

3.9 Fichier de sortie d'une instance au format JSON (API)

Nous avons eu l'idée de rendre les traitements de notre programme disponibles dans un format facile à comprendre et à analyser syntaxiquement. C'est pour cela que nous avons mis une option -api, qui va sortir un fichier au format JSON pour chaque algorithme différent.

3.9.1 Format JSON

JSON (JavaScript Object Notation – Notation Objet issue de JavaScript) est un format léger d'échange de données. Il est facile à lire ou à écrire pour des humains. Il est aisément analysable ou générable par des machines. Ces propriétés font de JSON un langage d'échange de données idéal.

3.9.2 Sortie des fichiers JSON

Les données sorties sont les traitements (nom, matrice de distance, positions s'il y a, chemins (par indice), tableau de distances (entre deux villes), le nom de l'algorithme.
Les noms des fichiers sorties selon les algorithmes sont les noms des fichiers d'entrées avec le suffixe de l'option d'algorithme appelé, rajoutez à cela .json. Par exemple :
<NomFichier>_nn.json pour l'algorithme Nearest Neighbour.
<NomFichier>_mst.json pour l'algorithme Minimum Spanning Tree.
<NomFichier>_opt.json pour l'algorithme Branch and Bound avec relaxation de Held Karp.

3.9.3 Implémentation dans un site web

Il est possible de résoudre des problèmes (de petites instances pour des raisons de performances) via un site web, où l'utilisateur peut envoyer un fichier TSP (que notre programme peut lire, soit FULL_MATRIX).
L'implémentation est disponible en ligne sur `http://david.phan.emi.u-bordeaux1.fr/S4/EDD/API_WEB_TSP/`. Nous utilisons la bibliothèque Javascript Graph Dracula pour afficher les instances.

3.9.4 Analyse des données JSON

La récupération des données du programme étant facilité, avec des fonctions PHP json_decode, nous avons récupéré les résultats de lectures, et d'algorithmes facilement. (disponible dans le fichier *libjson.php*)

3.9.5 Fonctions

— executeApis(Map, bool*, int, char *) qui execute les algorithmes avec la Map donnée en paramètre et les algorithmes choisis

— printJSON(Map,City *, int, char*) écrit le fichier JSON

Chapitre 4

Tests et performances

4.1 Tests

4.1.1 Tests du lecteur de TSP

Nous avons effectué des tests basiques qui consistent à lire les fichiers TSP avec le programme, puis de le comparer à ce que voyons dans le fichier TSPLIB. Ce genre de tests est très limité (pour de grandes instances).
Par la suite, pour automatiser la vérification du lecteur de TSP, nous avons créé un *CTest* automatisé qui nous permet de lancer la lecture de fichiers connus exemple10.tsp exemple14.tsp bays29.tsp et de les analyser et comparer avec les résultats précédents (vérifiés par nos soins).

4.1.2 Tests de fichiers TSP fournis

Nous avons lancé les différents algorithmes sur les instances, et comparés les résultats (longueur) de celles ci avec les optimales fournies.

	exemple10	exemple14	bays29	spanning30	ch130.	pr1002
NN	46	38.688102	2258	60.004190	7575.285301	315596.596000
MST	52	34.962292	2423	60.004190	8128.755901	351387.186000
OPT	42	30.878491	2020	60.004190	6110	259045

Comme nous pouvons le voir, la longueur résultante de Minimum Spanning Tree est toujours inférieur à 2 fois la longueur optimale.

4.1.3 Tests d'instances générées

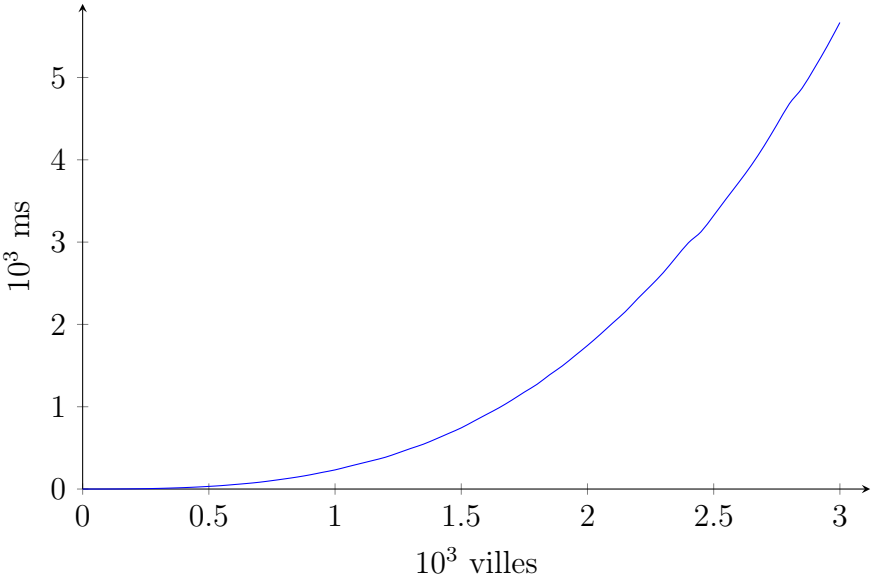
L'option "-r" vue en exemple précédemment permet d'effectuer des tests sur des matrices de point aléatoire, en spécifiant le nombre de villes voulu, la fonction main va faire appel à la fonction **mapCreateRandom** du module *Map*, le fonctionnement de cette dernière est le suivant :

- création d'un tableau de *Point* de taille le nombre de villes spécifié
- initialisation des champs x et y du tableau à une valeur aléatoire comprise entre 0 et 1000
- initialisation du nom de la fonction
- appel à la fonction **mapCreateFromPoints** qui instancie la map et la remplit avec le tableau de *Point*

4.2 Performances

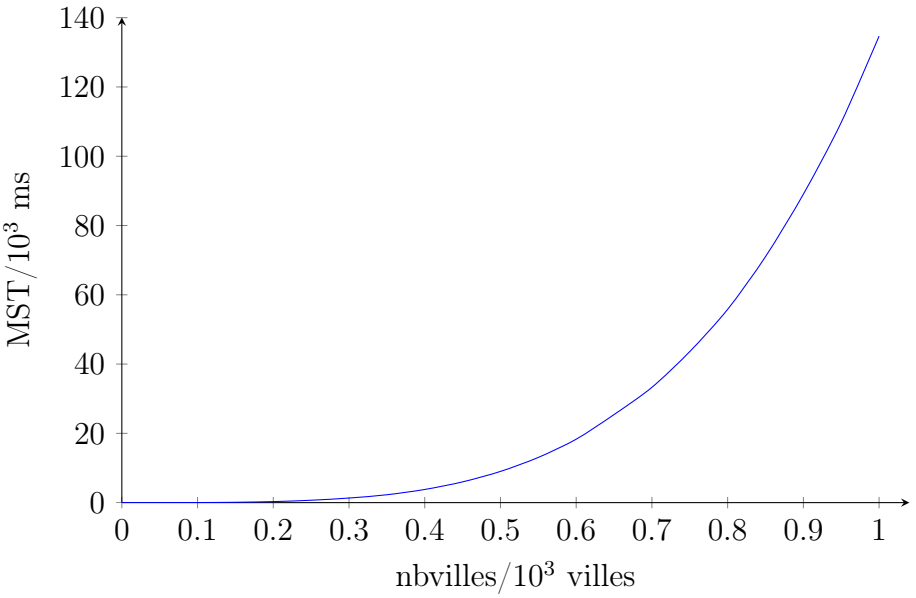
Pour nos tests de performances, nous avons lancé nos instances sur des ordinateurs de même configuration technique, *infini1 infini2 infini3 infini4*.

4.2.1 Nearest Neighbour



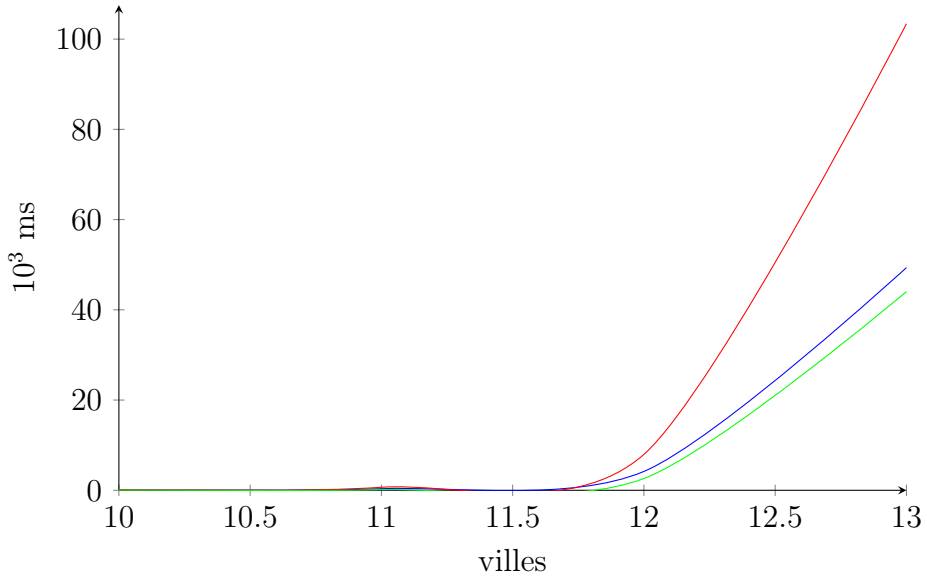
Taille de l'instance	100	200	300	400	500	800	1000	2000	3000
Nearest Neighbour (s)	0.001	0.003	0.008	0.017	0.032	0.123	0.233	1.744	5.668

4.2.2 Minimum Spanning Tree



Taille de l'instance	100	200	300	400	500	800	1000
Temps d'exécution(s)	0.023	0.296	1.326	3.800	9.011	55.80	134.7

4.2.3 BruteForce



Bleu : BF itératif
Rouge : BF récursif
Vert : BF multithread

BruteForce Iteratif

Taille de l'instance	exemple10	11	12	13	exemple14
Temps d'exécution(ms)	32.1	349	4096	47856	654195

BruteForce Multi-threadé

Taille de l'instance	exemple10	11	12	13	exemple14
Temps d'exécution(ms)	90	808	24441	382495	138881517
Temps réel(ms)	18	119	2521	37772	1142940

La version multithreadé n'est pas très performante comme nous pouvons le voir. Le fait d'avoir un problème parallelisable ne nous signifie pas qu'on peut gagner des performances.

BruteForce Récursif

Taille de l'instance	exemple10	11	12	13	exemple14
Temps d'exécution(ms)	58.1	638.6	7750.3	99740	1385947

Il est clair d'après les tableaux et le graphique que le BruteForce itératif est le plus efficace des trois versions (environ 10 minutes pour 14 villes). Cependant, à partir de 15 villes (inclues) il n'est pas la peine d'essayer de résoudre un tsp avec un algorithme de BruteForce, cela prendrait beaucoup trop de temps.

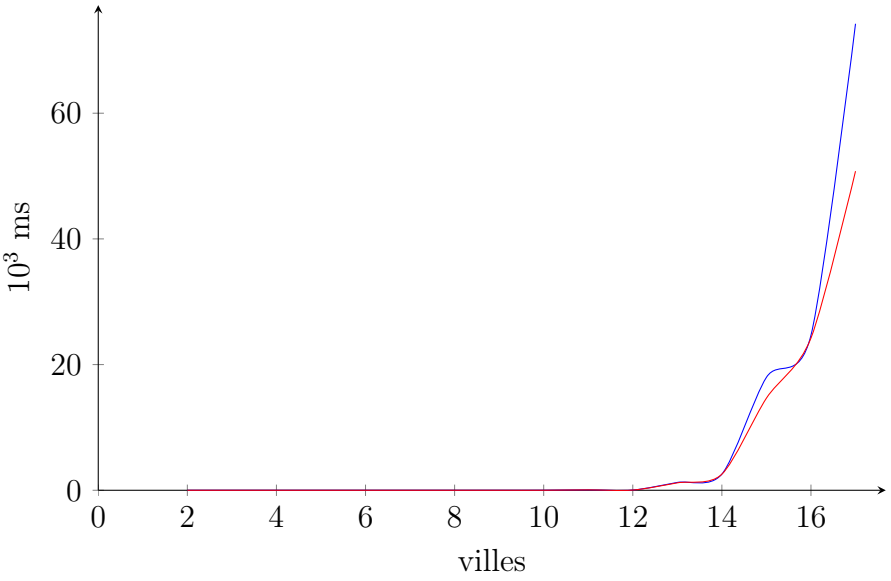
4.2.4 Branch and Bound

Taille de l'instance	exemple10	12	exemple14	16
Branch and Bound	6.356	40-300	6707	23884
B&BR(nn-mst)	6	40-300	6685	16483
B&BRHK	0.036	0.058	0.115	0.225

Branch and Bound

Pour la relaxation de Held-Karp uniquement, il est possible de résoudre des problème une centaine de villes, mais la rapidité varie énormément selon l'instance, en effet cela dépend du calcul du *lower Bound* initial qui peut nous donner le pire des cas de $\theta(2^n * n^2)$. En revanche on peut clairement dire que celui-ci est le plus optimisé et donne les meilleurs résultats. Cela peut aussi s'expliquer par l'implémentation non récursive de Branch and Bound.

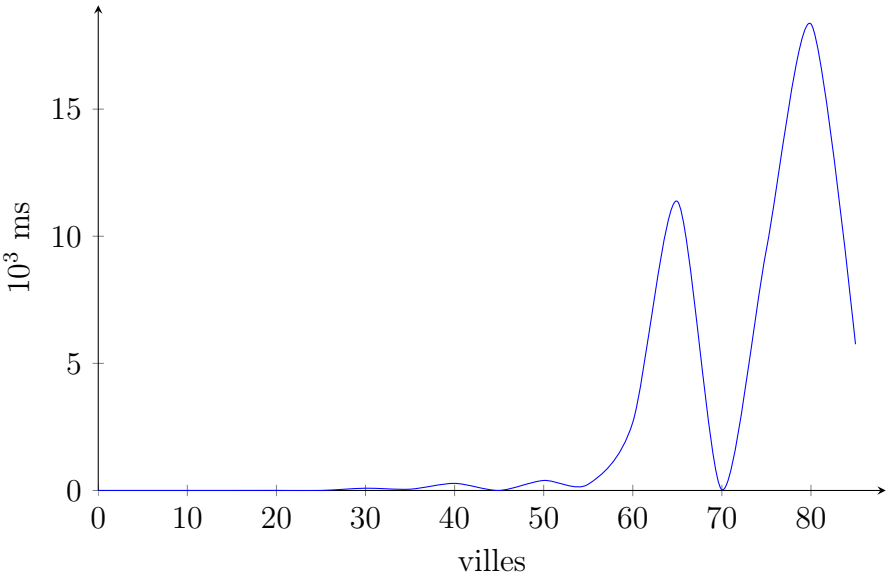
Branch and Bound avec relaxation NN-MST



Bleu : BB classique
Rouge : BB relaxé NN-MST

On constate une légère amélioration du BB classique qu'on a encore plus creusé avec la relaxation de Held-Karp.

Branch and Bound avec relaxation de Held-Karp



Ce graphe montre bien la non linéarité de l'algorithme. En effet, le temps d'exécution dépend de l'organisation profitable ou non des villes. Cela s'explique avec le calcul du *lowerBound*, et donc de la séparation des possibilités.

Chapitre 5

Organisation du travail, répartition des tâches, utilisation des outils, et difficultés rencontrées

5.1 Organisation du travail

5.1.1 Premiers pas réflexion sur le découpage modulaire, premier algorithme

Au départ, n'ayant pas connaissance des fichiers TSP, nous avons opté pour une implémentation résident sur des points décrivant la position de nos villes et une utilisation des calculs de longueurs pour éxecuter nos algorithmes. Dés le départ nous avons donc créé le type Point, City (les coordonnées) et Map (Une liste de City).

Nous avons commencé à implémenter Nearest Neighbour mais nous avons eu un problème, au lieu de rentrer manuellement la Map ou de coder le TSP reader tout d'abord, nous avons pensé à faire des Maps aléatoires de tailles données pour pouvoir ensuite vérifier les résultats. Par la suite nous avons gardé la fonctionnalité de Map aléatoires qui s'est avérée très utile.

Découpage modulaire

Par la suite, ayant pris connaissance de la structure des fichiers TSP et du fonctionnement du calcul des algorithmes, nous avons décidé que City contiendrait en plus un tableau de distances de la longueur du nombres de villes que contient la Map, ainsi chaque City contient les distances entre elle-même et les autres villes.

Les modules suivants s'ajoutent au programme :

— Api => Gestion de la sortie au format JSON
— Fcts => Fonctions globales a plusieurs fichiers pour le fonctionnement du programme
— Pri_queue => Objet file de priorité
— String => Librairie de gestion des chaines de caractères
— Tsp => Librairie de gestion des fichiers TSP
— Algos => Gestion de l'execution des algorithmes, coeur du programme
— Tree => Objet arbre planaire pour MST
— Vertex => Objet arbre planaire pour BBHK
— Gui => Gestion des tampons pour l'affichage
— Guisys => Gestion de l'interface graphique

5.1.2 Implémentations de tests, Automatisation des tests

Implémentations des tests

Tests de fichiers fournis

Nous avons implémenté des tests automatisés avec CTest, qui lancent quelques algorithmes et vérifient les résultats mis en mémoire via les correspondances d'expressions régulières de nos chemins. De plus nous avons ajoutés des tests de lecture de TSP en vérifiant via des vérifications de châines de caractères.

Tests de performances automatisés

Nous avons implémenter une option pour lancer des batteries de tests sur des instances aléatoires avec l'option -t <nbTests> <nbVilles> commençant par 3 villes puis en allant jusqu'à <nbVilles>. Cela pour avoir des temps d'exécution.

Fuites de mémoires

Nous avons analysé avec Valgrind et supprimé au maximum les fuites de mémoires. GTK+ provoque le reste des fuites mémoires.

Analyse de complexité de Nearest Neighbour et Minimum Spanning Tree

Complexité de Nearest Neighbour Le fait que nous avons implémenter l'algorithme Nearest Neighbour avec deux boucles imbriquées, nous indiquent que la complexité est de l'ordre de $\theta(n^2)$

Complexité de Minimum Spanning Tree Nous avons trois boucles implémentés dans notre implémentation. La complexité est dans l'ordre de $\theta(n^3)$

5.1.3 Nouvelles versions de Bruteforce et de Branch and Bound

Nouvelles versions de Bruteforce

Nous avons rajouté une version récursive pour pouvoir faire des tests de performances, le fait est, qu'une fonction récursive utilise la pile d'exécution et ne permet donc pas d'avoir de meilleures performances que la version itérative. Par contre l'algorithme utilisé nous génère des permutations ordonnées.

Nouvelle version de Branch and Bound

Nous avons implémenté une version sûre de Branch and Bound qui ne rentrera pas dans les mauvaises branches vu que nous arrêterons la récursion.
Puis nous avons ajouté une relaxation à cette version, c'est-à-dire d'initialiser le chemin minimal actuel à celui du chemin le plus court trouvé par Nearest Neighbour et Minimum Spanning Tree.
Nous avons en suite cherché un algorithme efficace sur internet, nous avons trouvé celui de Held-Karp qui fait une relaxation très bonne, et une implémentation de Branch and Bound avec un tas min.

5.1.4 Ajouts de fonctions, modification de la structure, amélioration de l'interface graphique

Ajouts de fonctions

Fonction d'écriture de résultat tspOut La fonction tspOut écrit le fichier .tour (au format TSPLIB) avec le meilleur résultat obtenu des algorithmes lancés.

Amélioration de l'interface graphique Nous avons ajouté la possibilité d'ouvrir un fichier depuis l'interface graphique, et donc de la rendre indépendante et facile d'utilisation. (Lancement avec un simple **./VDC -g**)
Nous avons ajouté la possibilité de sauvegarder le meilleur tour, la possibilité de générer une instance aléatoire, la possibilité de définir la ville de départ.

Fonction d'écriture de résultat en format JSON Nous avons ajouté une fonction d'écriture du traitement et des résultats des algorithmes. Cela permettrait à un utilisateur qui voudrait, par exemple utiliser les données traitées par notre programme en détail.

Petit site web utilisant le format JSON Par exemple, nous avons implémenté un petit site écrit en PHP, utilisant des modules Javascript pour le dessin des graphes. Il permet au visiteur de la page d'envoyer son fichier TSPLIB, de choisir l'algorithme à lancer, puis de voir le résultat.

Ecriture du rapport Nous avons écrit le rapport avec LaTex et l'avons exporté en pdf.

5.2 Utilisation des outils

5.2.1 Utilisation de Subversion

Nous avons utilisé SVN avec Savane pour versionner notre projet et travailler à plusieurs, le stockage distant des données nous a aussi permis de travailler aussi bien au CREMI que chez nous.

5.2.2 Utilisation de Code : :Blocks

Nous n'avons pas (ou très peu) utilisé emacs pour développer et avons préféré Code : :Blocks, un IDE pour le C/C++ qui a de nombreuses fonctions facilitant le développement comme l'auto-complétion, la compilation et le Debug intégré... Code : :Blocks fonctionne un peu comme Eclipse

5.2.3 Makefile

Nous avons choisi de faire un Makefile pour la compilation de notre projet, il est récursif et éxecute les Makefile des dossiers gui et algos, les options disponibles sont clean, cleanAll et doc qui respectivement suppriment tous les .o (fichiers objet), suppriment tous les .o et l'éxecutable, crée la documentation. Tous les fichiers Makefile ont leur double Makefile.db a éxecuter avec l'option make -f Makefile.db et qui compilent avec l'option -g de gcc soit la version Debug, a l'inverse les Makefile principaux compilent avec une optimisation maximum du code soit l'option -O3 de gcc

5.2.4 Documentation du code par DOxygen

Documentation du code par DOxygen

Doxygen est un générateur de documentation sous licence libre capable de produire une documentation logicielle à partir du code source d'un programme. Pour cela, il tient compte de la grammaire du langage dans lequel est écrit le code source, ainsi que des commentaires s'ils sont écrits dans un format particulier.
Nous avons commencé à commenter les fichiers .h et .c de plusieurs fichiers sources.

5.3 Répartitions des tâches et difficultés rencontrées

5.3.1 Répartitions des tâches

Travail collectif

Nous nous sommes accordés sur le découpage modulaire.

Nous avons débogué les modules des uns et des autres.
Nous avons tous participé aux tests et fait la documentation.

Tâches séparées

— Raphaël Afanyan : Algos BB Held Karp, structures de données(File prio, vertex), 1e Makefile
— Nicolas Marcy : Algorithme MST, tree.c, Les tests (prog+ctest), 2e Makefile
— David Phan : Algos Brute Force (toutes versions), Branch and Bound (toutes versions, Held Karp avec Raphael), Api + Site Web, Amélioration GUI, implémentation test TSP Reader (CTest)
— Jason Pindat : Structures de données (Point, City et Map), Gestionnaire d'execution des algos + gestionnaire d'erreurs + gestionnaire d'options du Main, Nearest Neighbour, TSP Reader, GUI, 3e Makefile

5.3.2 Difficultés rencontrées

Nous avons eu des problèmes avec SVN qui nous a supprimé des fichiers à certains moments et qui a fait des Merge automatique en supprimant du code qui aurait dû créer des conflits.

Le choix de la bibliothèque graphique pour la GUI et l'implémentation de cette dernière avec GTK+ a été difficile par moments.

Les Makefile à plusieurs reprises n'ont pas réagi comme nous l'avions prévu, la récursivité de ceux-ci et la mise en place des dépendances automatiques n'ont pas été les plus simples.

Chapitre 6

Conclusion

Pour clore ce rapport, nous voulions exprimer le fait que nous avons pris chacun beaucoup de plaisir à réaliser ce projet tous ensemble, pouvoir travailler en groupe de plus de 2, de plus avec des gens que nous apprécions a été quelque chose de nouveau et de très plaisant, la communication était très bonne, l'état d'esprit également. Nous sommes satisfaits du programme que nous avons produit, tout ce que nous avions en tête à l'origine a été réalisé, d'autres fonctionnalités auxquelles nous n'avions pas pensé sont venues au fur et à mesure et nous avons été capable de les implémenter. Grâce à cela, nous nous sentons désormais beaucoup plus à l'aise dans le langage C et dans la gestion de projet. Au final, nous remercions grandement notre professeur Irène DURAND pour nous avoir guidé sur la bonne voie tout au long du projet, ses conseils pour l'améliorer et ses suggestions qui ont poussé notre réflexion plus loin et nous ont permis de pouvoir réaliser plus de tâches.