EDD - Rapport

Florian KAUDER - Kinda AL CHAHID - Amélie GUEMON - Lilian CHAMPROY

16 avril 2014

Table des matières

Introduction

> *Le problème du voyageur de commerce est un problème mathématique qui consiste, étant donné un ensemble de villes séparées par des distances données, à trouver le plus court chemin qui relie toutes les villes. 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. De plus, la version décisionnelle de l'énoncé (pour une distance D, existe-t-il un chemin plus court que D passant par toutes les villes ?) est connue comme étant un problème NP-complet.*
>
> Problème du voyageur de commerce - Wikipédia

Nos travaux, présentés dans ce rapport, concerne les méthodes de résolutions de ce problème. On parle des algorithmes et de leurs implémentations, mais aussi des outils de travail en équipe et autres méthodes de résolution des problèmes.

Chapitre 1

Tâches et partage

Les tâches signalées par le symbole **(x)** sont achevées. A l'inverse, les tâches signalées par le symbole **( )** n'ont pas encore été effectuées.

Cette liste ne fait que généraliser les tâches : plusieurs membres du groupes ont pu aidé sur une tâche où seule une personne est marquée.

1.1 Algorithmes et implémentations

1.1.1 Heuristique Simple - Nearest Neighbour

(x) Amélie Création de l'algorithme - Implémentation de l'algorithme - Implémentation via *Matrix* - Test

1.1.2 Algorithme d'approximation - Minimum Spanning Tree

(x) Lilian Création de l'algorithme - Implémentation de l'algorithme via *Matrix* - Test

1.1.3 Algorithme exact - Bruteforce

(x) Florian Création de l'algorithme - Implémentation de l'algorithme via *Matrix* - Test

1.1.4 Algorithme exact - Branch and Bound

(x) Kinda Création de l'algorithme - Implémentation de l'algorithme via *Matrix* - Test

1.1.5 Module *Matrix*

(x) Florian Implémentation du module

(x) Kinda Test du module

1.1.6 Module *Graph*

(x) Florian Implémentation du module - Test du module

1.1.7 Lecture et écritures des fichiers TSP

(x) Amélie - Lilian Module de lecture des fichiers TSP - Test

1.1.8 Interface graphique

(x) Florian Interface graphique

1.1.9 Unification des modules - Intégration

(x) Florian Intégration - TSPSolver

1.2 Installation et compilation

1.2.1 Makefile générique

(x) Kinda Création et maintenance de CMake

(x) Florian Scripts de *memcheck* et *speedcheck*

1.3 Gestion des sources

1.3.1 Git

(x) Florian Mise en place de Git - Maintenance du projet - Exclusion des fichiers .o, executables, depends.txt, ...

Chapitre 2

Gestion du projet avec Git

Git est un logiciel de versions décentralisé. Il permet de synchroniser les différentes versions d'un projet, et de travailler en groupe sur un même projet.

L'hébergement du projet est sur GitHub, qui propose des hébergements gratuits de répertoires Git. Notre répertoire de projet est disponible à cet endroit : `https://github.com/Lumi-Bjorn/Edd_project`.

2.1 Avantages

Gestion des branches Lors de la création d'une branche, *Git* ne recrée pas intégralement l'arborescence du projet, permettant d'avoir une économie de temps et d'espace pour des projets denses.

Fusion efficace *Git* sait s'adapter à des modifications d'un même fichier par plusieurs utilisateurs, et ainsi de faciliter les démarches de fusion de branches.

Répertoire local *Git* copie le répertoire distant, ce qui permet de pouvoir travailler sur son projet, même sans accès à Internet.

Rapidité *Git* travaille sur le répertoire local. De plus, les mises à jour du répertoire distant sont extrêmement rapides.

2.2 Défauts

Complexité Une même commande *Git* peut être utilisée pour effectuer 2 tâches différentes. Aussi, les commandes ne sont pas très intuitives comparées à *SVN*.

Beaucoup de commandes *Git* possède beaucoup de commandes. Cependant, pour un usage quotidien "classique", ce nombre reste assez faible.

Comparé à *SVN*, *git* jouit d'une simplicité pour développer plusieurs fonctions instantanément, ainsi que de la possibilité de travailler sur son répertoire local et d'effectuer une simple mise à jour lors de la récupération d'Internet. C'est pour ces différentes raisons que *Git* a été choisi.

Chapitre 3

Compilation - Installation

Pour la compilation et l'installation du programme, nous utilisons *Make* avec des *Makefiles*.

3.1 Propriétés de *Make*

Make est un programme qui permet de compiler et de générer les exécutables du programme. Pour cela, il suffit de faire un simple **make** dans un terminal.

Un *Makefile* générique, mis à la racine du projet, permet l'exécution des différents *Makefiles* du projet pour simplifier la compilation et la génération d'exécutables.

Un des avantages de *Make* est qu'une étape est re-réalisée que si cela est nécessaire. Par exemple, si on modifie un seul fichier dans le projet, il n'est pas nécessaire de recompiler l'intégralité du projet, mais seulement les parties concernant ce fichier.

3.2 Génération automatique et installation

Cmake est un outil permettant de générer les *Makefiles* automatiquement. A partir de fichier *CMakeLists.txt*, *Cmake* va "construire" les fichiers d'installation du programme. Une fois les fichiers construits, il est nécessaire d'utiliser *Make* afin de compiler les fichiers nécessaire, puis d'installer les exécutables. La compilation et l'installation est donc réalisée en 3 étapes.

3.3 Automatisation des tests

Cmake peut aussi compiler et exécuter des programmes de tests. Pour celà, on construit normalement les exécutables de tests dans les différents modules du programmes, puis on ajoute une ligne dans le *CMakeLists.txt* placé à la racine du projet demandant l'exécution des tests spécifiés. L'étape de test est optionnelle, et s'effectue via la commande suivante : **make check**. Le résultat est le suivant :

```
$ make check

Test project /home/amolith/Edd/Projet/Git/Edd_project/build
            Start 1 : TestBruteforce
1/4 Test #1 : TestBruteforce ..... Passed 0.24 sec
            Start 2 : TestMatrix
2/4 Test #2 : TestMatrix ..... Passed 0.00 sec
            Start 3 : TestNearestNeighbour
3/4 Test #3 : TestNearestNeighbour ..... Passed 0.00 sec
            Start 4 : TestMST
4/4 Test #4 : TestMST ..... Passed 0.00 sec

100% tests passed, 0 tests failed out of 4
```

3.4 Tests de mémoire et de temps

Afin de vérifier rapidement si les modules ne contiennent pas de fuites mémoire ou s'ils sont rapides, deux commandes ont été rajoutées :

- **make memcheck** : vérifie la mémoire sur les tests
- **make speedcheck** : vérifie la rapidité sur le test du Bruteforce

Chapitre 4

Modélisation de matrice - Module *Matrix*

Le module Matrix permet de disposer d'une structure de données émulant une matrice. Comme toute structure de données, elle nécessite des fonctions élémentaires :

1. Créer la structure (*create* et *init*)
2. Détruire la structure (*destruct*)
3. Modifier une valeur (*setter*)
4. Accéder à une valeur (*getter*)

4.1 Composition de *Matrix*

Matrix se compose seulement d'un tableau à deux dimensions (matrice) et d'un entier (taille de la matrice).

Listing 4.1 – Matrix.c - Lignes 10 à 13

```
1 struct matrix{
2         int length;
3         double** array;
4 };
```

4.2 Initialisation des valeurs de *Matrix*

L'initialisation des valeurs est possible de deux façons :
– On initialise les valeurs une à une via *setMatrixValue()*
– On initialise les valeurs à partir d'une matrice statique via *setMatrixArray()*

Dans le deuxième cas, la solution trouvée actuellement en vigueur est de transtyper la matrice statique en *double**, et de parcourir la matrice comme un tableau à une dimension. Cela est possible car une matrice statique est stockée en mémoire de façon continue. L'ancienne solution cherchait à faire passer la matrice via un *double***, cependant, il était impossible de récupérer la deuxième dimension du tableau à cause d'une erreur inconnue : le passage vers un pointeur *double*** "détruisait" la deuxième dimension en l'initialisant à 0. Il n'y a donc plus de possibilités de récupérer les valeurs.

4.3 Test de *Matrix*

Pour vérifier l'intégrité des résultats du module, il est nécessaire de tester le module Matrix.

D'abord, il faut tester si une Matrix est créée et détruite correctement, puis si les différents getters et setters marchent correctement. Pour cela, on effectue un premier test en injectant des valeurs dans une Matrix et on vérifie que les valeurs soient les bonnes, puis on injecte une matrice et on vérifie que les cases correspondent.

Le code est disponible à l'annexe B.

Chapitre 5

Heuristique simple - *Nearest Neighbour*

5.1 Algorithme

Algorithm 1 Algorithme par *Nearest Neighbour*

```
function TROUVERROUTE(ref distanceVilles : matrice de décimaux)
    i ← 0
    tabChemin[0...nbVilles]
    indiceTabChemin ← 0
    while !ISENDOF(tabChemin) do
        distanceMin ← 2 * nbVilles
        indiceMin ← 0
        colonne ← 0
        while colonne < nbVilles do
            if (distanceVille[i][colonne] < distanceMin) ∧ (i! = colonne) ∧ (colonne + 1 ∉
tabChemin[0...indiceTabChemin][0...indiceTabChemin]) then
                distanceMin ← distanceVilles[i][colonne]
                incideMin ← colonne
            end if
            colonne + +
        end while
        tabChemin[indiceTabChemin] ← indiceMin + 1
        indiceTabChemin + +
        i ← indiceMin
    end while
     return tabChemin
end function
```

L'algorithme se base principalement sur la recherche du point le plus proche. On part du premier point, on récupère le plus proche, puis on récupère le plus proche de ce dernier, et ainsi de suite ...

5.2 Implémentation et problèmes

L'implémentation du précédent algorithme est présent à l'annexe C.

Nous devions implémenter, lors du premier TP, la fonction *NearestNeighbour*. Mais la fonction ne fonctionnait pas car, avant même de concevoir la partie algorithmique simple de *NearestNeighbour*, nous cherchions déjà à essayer d'utiliser des fonctions de gestion de matrices.

C'est pourquoi, *NearestNeighbour* a été développé en utilisant une matrice rentrée en brut, ainsi que le test renvoyant le chemin (chemin exact car vérifié « manuellement »). Mais, à ce moment là, tout était encore codé en un unique fichier(header, client, fournisseur), et il a été difficile séparer tous les fichiers, car un soucis de déclaration de variables persistait (impossible de savoir s'il y avait une histoire de variable globale ou non). Cependant, après une intense phase de débogage, le problème a fini par être résolu.

L'implémentation de *NearestNeighbour* a ensuite été changée, de manière à utiliser le module *Matrix* dans le code. La gestion de la mémoire est bonne sur le module *NearestNeighbour*, toute mémoire allouée est libérée (*vérifié par valgrid ./TestNearestNeighbour*).

5.3 Test

Le test de l'algorithme s'effectue via la comparaison entre le tableau de résultats qui devrait être trouvé et le tableau retourné par l'algorithme.

Listing 5.1 – TestNearestNeighbour.c - Lignes 30 à 33

```
1 */
2 int main(){
3   bool success=false;
4   double * matrice_distance= (double*)tab_distance;
```

Chapitre 6

Algorithme d'approximation - *Minimum Spanning Tree*

Dans le cas du *Minimum Spanning Tree*, plusieurs algorithmes existent déjà. Il fallait dans un premier temps définir quel algorithme utiliser.

6.1 Algorithme

Le choix s'est fait entre les algorithmes suivants :
- L'algorithme de Prim
- L'algorithme de Kruskal
- L'algorithme de Borůvka

Parmi ces trois algorithmes, celui retenu est celui de Kruskal. Beaucoup plus simple à coder et tout aussi efficace. L'algorithme de Kruskal consiste à choisir les arêtes de poids le plus faible sans qu'aucun cycle ne soit créé. Le résultat est un arbre, comprenant tous les sommets, de poids minimal.

Algorithm 2 Algorithme de Kruskal

```
function TROUVERROUTEOPTIMALE(ref G : graphe, ref poidsTab : matrice de décimaux)
    E : ensemblevide
    a : arete
    TRIERARETEORDRECROISSANT(G, poidsTab)
    a ← PRENDREPROCHAINEARETE(G)
    while a! = NULL do
        if !(CREEUNCYCLE(E, a)) then
            AJOUTERARETE(E, a)
        end if
        a ← PRENDREPROCHAINEARETE(G)
    end while
    return E
end function
```

6.2 Implémentation

Pour pouvoir utiliser les "arêtes", il a fallu créer une structure *Edge* qui prend trois int : deux pour les sommets et un autre pour contenir le poids de l'arête.

6.2.1 Implémentation simple de l'algorithme

Ensuite, il a fallut implémenter l'algorithme de Kruskal. Pour celà, il faut d'abord commencer par appeler la fonction *parcoursMST()*. Celle-ci va, à son tour, appeler la fonction *readMatrixAndCreateEdges()* qui permet de créer un tableau contenant toutes les arêtes possibles. Le tableau, pendant son remplissage, est automatiquement trié.

La fonction va alors récupérer les arêtes via *getNextEdge()*. Celle-ci récupère les arêtes du tableau triées par ordre de poids croissant, et vérifies à l'aide de *isCyclic()* si, en ajoutant la nouvelle arête à la route, un cycle se crée.

6.2.2 Implémentation de la recherche de cycle

L'algorithme utilise la recherche de cycle dans un graphe. Pour effectuer cette recherche, deux fonctions sont utilisées :

– *isCyclic()*
– *searchRecur()*

La première fonction est la fonction de base. Elle commence par ajouter l'arête à tester dans la route, puis elle va tester tous les chemins possibles à partir des arêtes contenus dans la route.

La recherche est effectuée comme si le graphe était non orienté. Dans le cas d'une arête *(u,v)*, la fonction va vérifier une à une toutes les villes possibles si il existe une arête avec u dans la route étant la ville, puis de même avec v. Si on trouve une arête répondant à ce critère, on considère que la ville est traversée, puis on fait appel à *searchRecur()* pour tester toutes les possibilités de chemin commençant par la précédente arête.

La deuxième fonction suit le même raisonnement que la première, excepté qu'elle est récursive et qu'elle vérifie que la prochaine ville qui va être visité ne l'a pas déjà été.

6.3 Complexité

La première fonction va être effectuée, dans le pire des cas, le nombre d'arêtes de la matrice. Une optimisation a été effectuée, qui a été de prendre uniquement les arêtes au-dessus de la diagonale de 0 dans la matrice. Soit t la taille de la matrice, on arrive donc à :

$$nbAretesPossibles = t^2 \rightarrow \sum_{i=0}^{t} i \rightarrow \frac{n(n-1)}{2} \tag{6.1}$$

De plus, on va vérifier à chaque fois si un cycle existe en ajoutant l'arête. Ce qui signifie que nous obtenons :

$$Complexite = nbAretesPossibles = nbVerificationCycles \tag{6.2}$$

Pour chaque vérification de cycle, on effectue au maximum i fois le nombre de valeurs de villes possibles, et $i*j$ fois le nombre d'arête dans la route actuelle. On trouve donc :

$$Complexite = nbVerificationCycles * nbVillesPossibles * nbAreteActuelles \tag{6.3}$$

De la même façon, *searchRecur()* peut être appelée au maximum i fois le nombre d'arêtes dans la route, et à chaque appel, elle appelle i fois via la boucle *for*. Ce qui donne :

$$Complexite = nbVerificationCycles * nbVillesPossibles * nbAreteActuelles^3 \tag{6.4}$$

En considérant que le nombre d'arêtes actuellement dans la route est quasiment à son maximum, soit n le nombre d'arête de la route, et v le nombre de villes possibles, on trouve finalement que, au maximum, la complexité est de :

$$Complexite = v * n^3 * \frac{n(n-1)}{2} \tag{6.5}$$

On peut noter que le nombre de villes possible est équivalent au nombre d'arêtes de la route, d'où, dans le pire des cas :

$$Complexite = n * n^3 * \frac{n(n-1)}{2} = O(n^5) \tag{6.6}$$

Cette complexité est néanmoins améliorable, en supprimant notamment l'utilisation de la variable *nbVillesPossibles* dans la fonction *isCyclic()*.

Chapitre 7

Algorithme exact par recherche exhaustive - *Bruteforce*

Cet algorithme doit permettre de trouver le chemin le plus optimisé en testant toutes les chemins possibles.

7.1 Algorithme

Algorithm 3 Algorithme par *Bruteforce*

```
function TROUVERROUTEOPTIMALE(ref route : matrice [0...nbVilles] d'entiers, ref nbPointsRestant : entier)
    poidsMinimal ← 0
    poids ← 0
    if nbPointsRestant == 0 then
        poids ← CALCULERPOIDSROUTE(route)
        if poidsMinimal > poids then
            poidsMinimal ← poids STOCKERROUTEOPTIMAL(route)
        end if
    else
        for i ∈ [0...nbVilles] do
            if !ESTDANSROUTE(route, i) then
                AJOUTERDANSROUTE(route, i)
                TROUVERROUTEOPTIMALE(route, nbPointsRestant -1)
                SUPPRIMERDANSROUTE(route, i)
            end if
        end for
    end if
end function
```

L'explication est faite en prenant 10 villes (ville 0, ville 1, ..., ville 9).
Initialement, la fonction est appelée avec la ville 0 comme première ville.

L'algorithme commence par remplir un tableau avec les entiers 0,1,2,3,...,9 (première série d'appel récursif). Le 10e appel récursif va calculer le poids, puis quitter la fonction. Le 9e appel supprime alors la ville 9 de la route, sort de la boucle et termine l'appel. Le 8e appel supprime la ville 8, puis avance dans la boucle et ajoute la ville 9 dans la route (qui est du coup 0 1 2 3 4 5 6 7 9). Un 9e appel est à nouveau effectuée, et ajoute la seule ville non présente, c'est-à-dire la ville 8. La route est alors 0 1 2 3 4 5 6 7 9 8.

Les 10 premières routes calculées sont :

1. 0-1-2-3-4-5-6-7-8-9
2. 0-1-2-3-4-5-6-7-9-8
3. 0-1-2-3-4-5-6-8-7-9
4. 0-1-2-3-4-5-6-8-9-7
5. 0-1-2-3-4-5-6-9-7-8
6. 0-1-2-3-4-5-6-9-8-7
7. 0-1-2-3-4-5-7-6-8-9
8. 0-1-2-3-4-5-7-6-9-8
9. 0-1-2-3-4-5-7-8-6-9
10. 0-1-2-3-4-5-7-8-9-6

Le nombre de route calculée est de :

$$nbRoutes(x) = (x-1)! \tag{7.1}$$

Dans notre cas de 10 villes, on trouve :

$$nbRoutes(10) = 9! = 362880 \tag{7.2}$$

7.2 Code

L'implémentation de l'algorithme se concentre sur les fonctions *searchOptimalPath()* et *calcPathByRecursivity()*. Elle est disponible à l'annexe D.

Quelques petites choses sont à noter :

- L'implémentation utilise une unique route de test, puis copie la route lorsque cette dernière est optimale. Cela évite de polluer la mémoire.
- Le nombre de villes restantes (*nbRemainingPoints*) permet de calculer la place de la ville à ajouter/-supprimer dans la route.
- La variable *pathSize* est une variable globale, car elle est utilisée dans de nombreuses fonctions et n'est modifiée que lorsqu'on souhaite calculer à nouveau la ou les routes optimales.

7.3 Fonctions ajoutées lors de l'implémentation

Pour une implémentation correcte, il a été nécessaire d'ajouter plusieurs fonctions pour gérer le stockage des routes optimales, ainsi que des fonctions pour voir et modifier des tableaux.

7.3.1 Stockage des routes optimales

Le stockage des routes s'effectue via un tableau de tableaux. Le comportement est similaire à une pile : lorsqu'une route optimale est trouvée, la route est ajoutée à la pile. Cependant, quand une route plus optimale est trouvée, la pile est alors vidée, puis on ajoute la nouvelle route optimale. Pour économiser de l'espace mémoire, les routes supprimées de la pile sont libérées.

7.3.2 Routes et tableaux

Dans l'implémentation, les routes sont des tableaux d'entiers. L'utilisation de ces routes nécessite donc d'avoir quelques fonctions pour manipuler les tableaux. Dans notre cas, il faut savoir si un entier est présent dans un tableau (*isInArray()*) et voir les entiers dans le tableau (*showArray*).

7.4 Test du programme

Si le programme fonctionne correctement, le poids optimal pour la matrice de test de 10 points est de 42. Pour tester ce fonctionnement, un simple *if ... else ...* suffit.

Listing 7.1 – TestBruteforce.c - Lignes 33 à 36

```
1  if (getOptimalWeight() == 42){
2    printf("\n-> Bruteforce : Poids optimal trouvé et correct\n");
3    success = true;
4  }
```

Chapitre 8

Algorithme exact - *Branch and Bound*

8.1 Algorithme

Algorithm 4 Algorithme par *Branch and Bound*

```
function TROUVERROUTEOPTIMALE(ref route : matrice [0...nbVilles] d'entiers, ref nbPointsRestant : entier)
    poidsMinimal ← 0
    poids ← CALCULERPOIDSROUTE(route)
    if nbPointsRestant == 0 then
        if poidsMinimal > poids then
            poidsMinimal ← poids STOCKERROUTEOPTIMAL(route)
        end if
    else if poids < poidsMinimal then
        for i ∈ [0...nbVilles] do
            if !ESTDANSROUTE(route, i) then
                AJOUTERDANSROUTE(route, i)
                TROUVERROUTEOPTIMALE(route, nbPointsRestant -1)
                SUPPRIMERDANSROUTE(route, i)
            end if
        end for
    end if
end function
```

L'algorithme de *Branch and Bound* est très similaire à un algorithme de *Bruteforce*, excepté que une vérification supplémentaire est faite : une route est abandonnée à partir du moment où son poids est supérieur au poids minimal. Il s'agit du principe d'un algorithme *Branch and Bound*.

> *La méthode de branch and bound (procédure par évaluation et séparation progressive) consiste à énumérer ces solutions d'un manière intelligente en ce sens que, en utilisant certaines propriétés du problème en question, cette technique arrive à éliminer des solutions partielles qui ne mènent pas à la solution que l'on recherche. De ce fait, on arrive souvent à obtenir la solution recherchée en des temps raisonnables. Bien entendu, dans le pire cas, on retombe toujours sur l'élimination explicite de toutes les solutions du problème.*
>
> Branch and Bound - Université de Québec à Chicoutimi

8.2 Implémentation

L'implémentation de l'algorithme est essentiellement la même chose. Afin de limiter la redondance de code, une partie des fonctions utilisées par le module *Branch and Bound* ont été supprimées, et sont maintenant récupérer à partir du module *Bruteforce*. Les changements notables concernent l'ajout de cette fameuse condition pour obtenir le *Branch and Bound* ou encore l'ajout d'un paramètre à la fonction de calcul de poids d'une route, afin de pouvoir calculer des routes partielles.

Listing 8.1 – BranchAndBound.c - Lignes 184 à 185

```
1         //Si le poids est toujours inférieur au poids minimal actuel, on continue la route
2         else if (calcWeightForPath(m, path, pathSize-nbRemainingPoints) < currentMinWeight)
```

Chapitre 9

Interface graphique

L'interface graphique de l'application permet de visualiser le résultat d'un des précédents algorithmes de recherche de chemin.

9.1 Librairies utilisées

La base de l'interface se compose des librairies suivantes :
- SDL2 - C'est une librairie multimédia qui permet d'utiliser facilement les différentes interfaces d'un ordinateur (contrôleurs, interfaces vidéo, interfaces son, etc.). La particularité de cette librairie est qu'elle s'adapte en fonction du système d'exploitation et des drivers disponibles sur les machines, ce qui en fait une librairie ultra-portable.
- SDL2-TTF - Il s'agit d'une extension de SDL2 qui permet d'utiliser des polices d'écritures en *.ttf* afin d'afficher des caractères, chose que la librairie originelle ne permet pas.
- FreeType - C'est le prérequis de la librairie SDL2-TTF. Il s'agit d'une librairie permettant l'usage des fichier *.ttf*.

9.2 Fonctionnement

Dans ce paragraphe, seul le fonctionnement général de l'interface graphique est expliqué.

Le premier module de l'interface est le module fusionné avec **TSPSolver**, c'est-à-dire le programme principal. Il s'agit d'un module permettant d'initialiser les composants graphiques (fenêtres, moteurs de rendus, contrôleurs, etc.) et de tenir en vie la fenêtre de l'interface graphique. Une fois que la fenêtre est fermée, le module finit par détruire les composants devenus inutiles.

Le deuxième module du programme est le module **Drawer**. Comme son nom l'indique, ce module possède toutes les fonctions permettant de dessiner des rectangles, des caractères, ou encore des lignes. Il contient aussi la fonction de rafraîchissement de l'interface graphique.

Le troisième module est le module **TSPDrawer**. Il s'agit d'une surcouche du module **Drawer** qui permet de dessiner des villes et des chemins. Il contient un système de sauvegarde des villes en mémoire, ainsi des fonctions de mises à l'échelle des coordonnées des villes par rapport à la taille de la fenêtre.

Le quatrième module est **Input**, qui est actuellement inutile. Il est prévu pour gérer des entrées au clavier afin d'effectuer divers commandes dans une utilisation future.

Le programme suit le fonctionnement suivant :
- Initialisation des composants graphiques
- Lecture des informations sur les villes et la route à dessiner
- Passage des contrôleurs (graphiques, etc.) aux différents modules
- Affichage de la fenêtre et de la route TANT que celle-ci n'est pas fermée
- Suppression des composants graphiques

9.3 Améliorations possibles

Les améliorations possibles sont nombreuses : l'interface graphique n'est que très primaire. Il peut être possible d'ajouter une fonction de zoom/dézoom sur le plan, de déplacement sur le plan, la modification des villes en points dont les informations apparaîtraient lors du survol de la souris, ou encore du changement d'algorithme utilisé par entrée au clavier.

Chapitre 10

Module de lecture des fichiers TSP

Afin de récupérer des matrices différentes, il est nécessaire d'implémenter un module pour lire les fichiers TSP.

10.1 Algorithme

Algorithm 5 Algorithme pour lire un fichier

```
function RECUPERERMATRICE(ref nomFichier : chaîne de caractères)
    file ← OUVRIRFICHIER(nomFichier)
    buffer ← LIREFICHIER
    while buffer! = "DIMENSION" do
        buffer ← LIREFICHIER
    end while
    dimension ← LIREPROCHAINEVALEUR
    while buffer! = "EDGE_WEIGHT_SECTION" do
        buffer ← LIREFICHIER
    end while
    matrice ← CREERMATRICE(dimension)
    while buffer! = "DISPLAY_DATA_SECTION" do
     AJOUTERDANSMATRICE(buffer)
    end while
     return matrice
end function
```

10.2 Utilisation

Pour utiliser le module, il suffit donc de faire passer le chemin du fichier à ouvrir à la fonction *fopen()*. De plus, une autre fonction est disponible dans le module, et qui permet de lire les coordonnées des villes placées à la fin du fichier. Ceci est notamment utile pour l'interface graphique qui a besoin des coordonnées des villes pour fonctionner.

Chapitre 11

Intégration des modules - TSPSolver

Tous les modules sont à présent fonctionnels. Cependant, il reste une tâche à effectuer : unir tous les modules dans un unique programme.

11.1 Correction des précédents modules

La première chose à faire a été de faire créer des librairies par chacun des modules développés jusqu'à présent. Pour celà, une simple modification des fichiers *CMakeLists.txt* a corrigé le problème.

De plus, certaines fonctions possédaient les mêmes noms, ce qui n'est pas possible en C. Le cas le plus notable est le module **BranchAndBound** et le module **Bruteforce** qui sont semblables à quelques lignes près.

Enfin, il a fallut réaliser quelques modifications afin que tous les modules implémentant un algorithme de recherche renvoie une route au même format (c'est-à-dire actuellement un *double**).

11.2 Passage des paramètres

Le plus important était de savoir comment pouvait-on effectuer 2 algorithmes différents à partir d'une même commande. Pour cela, la librairie GNU possède un programme nommée **getopt**, permettant de récupérer les différentes options passées à un programme en paramètre.

Les options sont configurées comme cela :

- **-a** : permet de choisir l'algorithme à utiliser. 0 correspond à Nearest Neighbour, 1 pour Minimum Spanning Tree, 2 pour Bruteforce et 3 pour Branch And Bound.
- **-g** : si l'option est donnée, l'interface graphique sera ouverte après calcul de la route.
- **<nameFile.tsp>** : en dehors de ces options, le paramètre nécessaire pour le fonctionnement du programme est le fichier .tsp contenant la disposition des villes et leurs distances.

11.3 Fonctionnement

Après avoir déterminés ce qu'il faut faire avec les options, il faut maintenant les utiliser. On commence par lire le fichier passé en paramètre au programme, puis en fonction de l'option **-a**, on choisit l'algorithme à utiliser. Après avoir calculer la route, on fait appel à l'interface graphique si l'option **-g** avait été donnée en paramètre.

Dans ce cas, l'interface graphique va être appelée avec la route trouvée, mais aussi avec les villes et leurs coordonnées lues directement dans le fichier .tsp. A partir de là, l'interface graphique va effectuer une étape de mise à l'échelle du plan (si possible), puis rend un affichage du plan.

Conclusion

TSPSolver est un programme qui se veut simple et fonctionnel. Il permet d'utiliser tous les algorithmes implémentés et présentés dans ce rapport, ainsi que de fournir la lecture de fichiers TSP et l'affichage à travers une interface graphique.

De façon plus générale, le projet de résolution du problème du voyageur du commerce nous a été une source de bénéfices sur la gestion de projet et le développement logiciel en équipe. Chacun a appris à travailler ensemble et à communiquer afin de réussir à obtenir un programme stable et fonctionnel.

> *L'une des principales causes de la chute de l'Empire romain a été que, faute du zéro, ils n'avaient aucun moyen d'indiquer l'achèvement avec succès de leurs programmes C.*
>
> Robert Firth - Auteur d'ouvrages sur le programmation

Annexe A

HowToUseGit

Listing A.1 – HowToUseGit

```
1  ######################
2  ## How to Use Git ? ##
3  ######################
4
5  ## Initialisation de Git sur une nouvelle machine
6
7  # Synchroniser son compte GitHub
8
9  Cette étape permet de faire correspondre son compte GitHub avec sa machine
10         afin que les changements et autres commits apparaissent dans les
11         stats du profil GitHub.
12
13 Pour celà, il suffit de faire :
14         > git config --global user.email "votre.email@ducompteGitHub.ici"
15
16 ## Fonctions de base ##
17
18 # Récupérer le dépôt
19
20 Pour celà, on utilise :
21         > git clone <url_du_dépôt>
22 Git va alors créer un nouveau répertoire contenant tous les fichiers contenus
23         dans le projet.
24
25 # Ajouter un nouveau fichier
26
27 Il faut commencer par :
28         > git add <fichier>
29 Le fichier est alors ajouté pour la prochaine mise à jour du projet.
30 Une fois tous les nouveaux fichiers ajoutés, il est nécessaire d'officialiser
31         la prochaine mise à jour avec :
32         > git commit -m "message␣pour␣dire␣ce␣que␣j'ajoute␣ou␣je␣modifie"
33
34 Enfin, pour ajouter définitivevement la mise à jour au dépôt, on finit par un :
35         > git push origin master
36
37 /?\ : pour plus de détails sur la commande push, veuillez consulter les informations
38         à propos du systéme de branches de Git.
39
40 # Mettre à jour un fichier
41
42 Pour mettre à jour un fichier existant déjà dans le dépôt :
43         > git commit -a -m "message␣de␣mise␣à␣jour"
44 Par défaut, la commande commit ne met à jour que les nouveaux fichiers.
45 Avec l'option -a (all), Git met à jours TOUS les fichiers.
46 Enfin, il suffit à nouveau de "pusher" :
47         > git push origin master
48
```

```
# Récupérer les modifications

Pour ajouter les derniéres modifications au répertoire local :
        > git pull

# Supprimer un fichier

Pour supprimer un fichier :
        > git rm <chemin_du_fichier>
La commande supprime le fichier dans le dépôt.
Elle marche donc même si le fichier n'est plus présent dans votre répertoire local
        mais qu'il existe encore sur le dépôt distant.

## Bien utiliser Git ##

# Comment développer avec Git ?

Utiliser Git est trés simple : dés que l'on a effectué une modification un minimum
        conséquente, il faut effectuer un commit pour avoir une trace de nos
        modifications et en quoi elles consistent.
Par "conséquent", j'entends qu'il est inutile de commit à chaque modification d'une
        ligne ou autre. Le commit s'effectue aprés l'ajout d'une fonction, la
        correction d'un bug, l'ajout d'un répertoire de fichiers, etc.
Aprés une phase de travail (exemple : entre deux pauses), il faut push les commits
        que l'on a réalisés.

/!\ : il peut être nécessaire de push beaucoup plus souvent en fonction des
        besoins de vos collégues. Si l'un d'entre eux attend une fonction au plus
        vite, effectuer un push dés que le développement de cette fonction
        s'achéve est une excellent idée.

# origin ? master ?

origin est le serveur sur lequel le dépôt d'origine se situe.
        Dans notre cas, il s'agit du serveur de GitHub.
master est la branche principale du projet.
Le systéme de branches est un outil trés puissant.
Une branche est une version du programme. Par exemple, imaginons que j'implémente
        une nouvelle fonctionnalité complétement buggué du programme.
Pour éviter que l'intégralité du programme ne soit plus utilisable (car je viens de
        rajouter une mise à jour qui bug le programme), je vais créer une nouvelle
        branche. Par exemple, la branche dev.
Tant que ma fonctionnalité n'est pas finie et testée, je vais envoyer toutes mes
        modifications sur cette branche.
Une fois la fonctionnalité finie, la branche dev et la branche master va être
        fusionnée, et les utilisateurs vont directement passer du programme
        originel au programme avec la nouvelle fonctionnalité.
Pour mieux comprendre : http://fr.openclassrooms.com/informatique/cours/gerez-vos-codes-source-avec-

# Consulter les logs

        > git log

# Voir les différences entre le répertoire local et le dépôt

        > git diff

# Commit avec des fichiers spécifiques

        > git commit <fichier_1> <fichier_2> ... <fichier_n> -m "message"

# Annuler un commit publié et restaurer les fichiers avant le commit

Si un mauvais commit a été effectué sur le dépôt, il est nécessaire de créer
        un nouveau commit qui va annuler l'ancien.
Pour celà, il faut d'abord récupérer le numéro du commit avec :
```

```
115         > git log
116 /!\ Il n'est pas nécessaire de récupérer TOUTE l'id du commit. Seuls les 5 premiers
117         chiffres sont nécessaires s'ils sont uniques.
118
119 Une fois l'id récupérée, il suffit alors d'effectuer :
120         > git revert <id_du_commit_a_annuler>
121 Puis de pusher l'annulation :
122         > git push origin master
123
124 # Gérer un projet avec des branches
125
126 Pour créer une nouvelle branche, il suffit de faire :
127         > git branch <nom_de_la_nouvelle_branche>
128
129 Pour ensuite travailler dessus :
130         > git checkout <nom_de_la_branche>
131
132 Aprés un checkout, les nouveaux commit impacteront la nouvelle branche.
133 La branche master est à nouveau modifiable aprés un :
134         > git checkout master
135
136 L'ajout et la modification des fichiers fonctionnent exactement comme avec la branche
137         principale, excepté que pour pusher les changements de la nouvelle branche,
138         il faut faire :
139         > git push origin <nom_de_la_branche>
140
141 # Récupérer une branche depuis le serveur
142
143 Pour voir les branches que le serveur connaît :
144         > git branch -r
145
146 Une fois la branche que l'on souhaite récupérer trouvée, il suffit de faire :
147         > git branch --track <nom_de_la_branche_sur_votre_pc> <nom_branche_sur_serveur>
148
149 Une branche est alors créée sur votre ordinateur qui "suivra" la branche du serveur.
150 Pour passer sur la nouvelle branche, il suffit alors de faire :
151         > git checkout <nom_branche_sur_serveur>
152
153 /!\ PENSEZ A COMMIT VOTRE TRAVAIL AVANT DE CHANGER DE BRANCHE !
154
155 # Push sur une branche distante
156
157 Pour pusher sur une branche distante, il suffit de faire :
158         > git push origin <branche_locale>:<branche_sur_serveur>
159
160 Par exemple, si votre branche locale est "dev" et la branche du serveur est "FonctionInDev" :
161         > git push origin dev:FonctionInDev
162
163 + Fusionner des branches
164
165 Une fois que la fonctionnalité voulue est finalisée sur la nouvelle branche,
166         il faut fusionner ces changements avec la branche principale.
167 Celà va permettre d'ajouter la nouvelle fonctionnalité au projet tout en gardant
168         les éventuelles modifications de la branche master.
169 Tout d'abord, il faut se replacer sur la branche master :
170         > git checkout master
171 Puis il faut fusionner cette branche avec la branche <nouvelle_branche> :
172         > git merge <nouvelle_branche>
173
174 Une fois que la fusion a été effectuée, la branche de développement n'est plus
175         utile. Pour la supprimer, il faut utiliser :
176         > git branch -d <nouvelle_branche>
177
178 /!\ Pour supprimer une branche même si les modifications n'ont pas été ajoutées
179         à la branche principale, il faut utiliser l'option -D.
180
```

```
/!\ Avant de changer de branche, il est nécessaire de sauvegarder les changements.
        Au lieu de les sauvegarder avec un commit, il est possible de faire :
        > git stash
        avant le changement de branche, puis, une fois de retour sur la branche :
        > git stash apply

## POUR PLUS D'INFORMATIONS ##
http://fr.openclassrooms.com/informatique/cours/gerez-vos-codes-source-avec-git
```

Annexe B

TestMatrix.c

Listing B.1 – TestMatrix.c

```c
/* Cette fonction ne sert qu'a savoir si la fonction Matrix marche seul avec
 *une matrice de dimension 2, en effet, matrix est utilisé par les différentes
 *fonction pour générer des matrices à partir des test et des données de chaque
 *fonction de calcul de parcours.
 *
 *Par conséquent, une fois le test effectué,
 *la matrice générer par TestMatrix sera supprimé.
 *
 */
#include "Matrix.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

double tab[2][2] = {
        {4, 6},
        {7, 5}
};

int main(){
        bool success = false;
  // On créé une matrice de dimension 2
        Matrix m = createMatrix(2);
        // Valeur de test
        double testValue = 2.8;

        setMatrixValue(m, 0, 0, 1.5);
        setMatrixValue(m, 0, 1, testValue);
        // testvalue est mise dans la matrice

        // Si la matrice dans la position (m,0,1) est la même valeur que le testvalue,
        // cela signifie que la matrice marche
        if (getMatrixValue(m, 0, 1) == testValue)
                printf("-> Matrix : Setters et Getters -> Fonctionnement correct\n");
        else
                printf("-> Matrix : Setters et Getters -> Résultat non correct\n");

        setMatrixArray(m, (double*)tab, 2);
        // On effectue le même test que précédemment, sauf que c'est avec la matrice
        // statique que l'on fait le test
        if (getMatrixValue(m, 1, 1) == tab[1][1]){
                printf("-> Matrix : SetMatrixArray -> Fonctionnement correct\n");
                success = true;
        }
        else
                printf("-> Matrix : SetMatrixArray -> Résultat non correct\n");

        // On affiche la matrice
```

```
49          printf("Affichage␣de␣la␣matrice␣(showMatrix())␣:␣\n");
50          printf("\n");
51          showMatrix(m);
52
53          // Comme il s'agit d'une matrice de test, une fois les contrôles effectués,
54          // on peux la supprimer
55          destructMatrix(m);
56
57          if (success)
58                  return EXIT_SUCCESS;
59          return EXIT_FAILURE;
60  }
```

Annexe C

NearestNeighbour.c

Listing C.1 – NearestNeighbour.c

```c
#include <stdbool.h>
#include <malloc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include <Matrix.h>
#include <NearestNeighbour.h>

/*Descrption: Cette fonction va parcourrir une matrice, en partant de la 1ére ville, et en choisissa
condition que celle-ci ne soit pas une ville déjà parcourue. Les villes parcourues seront alors insc
Paramétre: Prend en paramétre une matrice de double, qui est la matrice des distances entre les ville
Retour: Elle renvoie un tableau de double,contenant l'ordre des villes à parcourrir.
*/
double* parcoursVoy(Matrix tab_distance){
  int length_tab_distance=getMatrixLength(tab_distance);
  double* tab_chemin= malloc(sizeof(double)*length_tab_distance);
  initMoinsUn(tab_chemin,length_tab_distance); //initialisation tableau des villes parcourues
  int i=0;
  int indice_tab_chemin=1;
  while (tab_chemin[length_tab_distance-1]==-1){ //tant que l'on a pas visité toutes les villes
    double distance_min=getMatrixMaximum(tab_distance);
    int indiceMin=0;
    for (int colonne=0; colonne<length_tab_distance; colonne++){  //pour toutes les villes d'une colo
      if (getMatrixValue(tab_distance,i,colonne)<distance_min && i!=colonne //si distance entre 2 vi
          && !testAppartenance(tab_chemin,colonne+1,indice_tab_chemin)){  //et si colonne+1 n'a pas
(colonne commence à 0, et les villes sont numérotées à partir de 1)
        distance_min=getMatrixValue(tab_distance,i,colonne);
        indiceMin=colonne;     //on sauvegarde l'indice de la ville
      }
    }
    tab_chemin[indice_tab_chemin]=indiceMin+1; //on rajoute la ville trouvée au tableau des vilels p
    indice_tab_chemin++;  //on avance d'un cran dans le tableau des villes
    i=indiceMin;  //on sauvegarde la derniére ville parcourue
  }
  return tab_chemin;     //retour du tableau contenant les villes parcourues
}

//Initialise un tableau de double à -1 (pour ne pas confondre avec la ville 0)
void initMoinsUn(double* t, int longueur_tableau ){
  t[0]=1;
  for (int i=1; i<longueur_tableau; i++) {
    t[i]=-1;
  }
}

//Teste l'apartenance d'un entier x, à un tableau de double t
```

```
46 bool testAppartenance(double *t, double x, int longueur_tableau){
47   for (int i=0; i<longueur_tableau+1; i++){
48     if (t[i]==x)
49       return true;
50   }
51   return false;
52 }
53
54 //Affiche, caractére pas caractére, un tableau de double
55 void afficher(double* t,int longueur_tableau){
56   printf("Route trouvée : ");
57   for (int i=0; i<longueur_tableau-1; i++){
58     printf("%lf - ", t[i]);
59   }
60   printf("%lf\n", t[longueur_tableau-1]);
61 }
62
63 //Teste l'égalite entre 2 tableaux de doubles
64 bool egalite(double *t1,double * t2){
65   int size1=sizeof(t1)/sizeof(double);
66   int size2=sizeof(t2)/sizeof(double);
67   if (size1==size2){
68     for (int i=0; i<=size1; i++){
69       if (t1[i]!=t2[i])
70         return false;
71     }
72     return true;
73   }
74   return false;
75 }
```

Annexe D

Extrait de Bruteforce.c

Listing D.1 – Bruteforce.c - Lignes 161 à 243

```
/** searchOptimalPath
* Effectue la recherche de la route optimale
* Renvoie un tableau de routes
*/
int** searchOptimalPath(Matrix m){
        // Initialisation
        currentMinWeight = 0x7FFFFFFF;
        weight = 0;
        nbPaths = 0;
        pathSize = getMatrixLength(m);
        initOptimalPaths();

        // On fournie un tableau défini ici pour éviter de recréer le tableau
        // à chaque appel de la fonction récursif = économie de mémoire
        int* path = malloc(sizeof(int)*pathSize);
        initPath(path);

        // Démarrage de l'algorithme
        path[0] = 0;
        calcPathByRecursivity(m, pathSize - 1, path);

        // On affiche le nombre de routes testées
        printf("Nombre de routes possibles testées : %d\n" , nbPaths);
        showOptimalPaths();

        free(path);
        return optimalPaths;
}

/** calcPathByRecursivity
* Recherche les routes optimales par récursivité
*/
void calcPathByRecursivity(Matrix m, int nbRemainingPoints, int* path){
        // Si il n'y a plus de path à trouver = si la route est complète
        if (nbRemainingPoints == 0){
                // On incrémente le compteur de routes testées
                nbPaths++;

                // On calcule le poids de la route
                weight = calcWeightForPath(m, path);

                // Si le poids trouvé est inférieur au poids minimal actuel
                if (weight < currentMinWeight){
                        // On supprime toutes les précédentes routes
                        removeAllOptimalPaths();
                        // On ajoute la nouvelle route
                        addOptimalPath(path);
                        // On modifie le poids minimal
```

```
49                          currentMinWeight = weight;
50
51                          //printf("Nouveau poids trouvé : %f pour la route ", weight);
52                          //showArray(path, pathSize);
53                  }
54                  // Sinon si on trouve une route avec un poids équivalent
55                  else if (weight == currentMinWeight){
56                          // On ajoute la route aux routes optimales
57                          addOptimalPath(path);
58
59                          //printf("Nouvelle route avec le poids %f : ", weight);
60                          //showArray(path, pathSize);
61                  }
62
63                  // Décommenter pour voir les routes créées
64                  // printf("Poids : %f pour route : ", weight);
65                  // showArray(path, pathSize);
66          }
67          else {
68                  // On teste si le tableau de path contient le point 0, puis le point 1, puis ...
69                  // Si le point n'est pas contenu dans le tableau, on l'ajoute et on appelle
70                  // la fonction récursive à nouveau
71                  // Exemple : 0 1 2 3 4 5 6 7 8 9 puis 0 1 2 3 4 5 6 7 9 8 puis 0 1 2 3 4 5 6 8 7 9 e
72                  for (int i=0; i<pathSize; i++){
73                          if (!isInArray(i, path, pathSize)){
74                                  path[pathSize - nbRemainingPoints] = i;
75                                  calcPathByRecursivity(m, nbRemainingPoints-1, path);
76
77                                  // On supprime le point du tableau aprés toutes les routes dans
78                                  //cette configuration trouvées
79                                  path[pathSize - nbRemainingPoints] = -1;
80                          }
81                  }
82          }
83  }
```