24/05/2014

Rapport du projet d'informatique

PROJET : LE TOUR DU MONDE LE MOINS CHER

Réalisé par : Robin Moussu & Jingbo Su

Table des matières

1.Introduction

Le but de ce projet informatique est d'appliquer nos connaissances acquises pendant cette année et de se familiariser avec les notions de programmation.

Donc pour notre sujet, le but est de réaliser le tour du monde le moins cher. C'est en fait le problème du voyageur de commerce qui doit visiter N villes en passant par chaque ville exactement une fois. Il commence par une ville quelconque et termine en retournant à la ville de départ. En connaissant le coût (temps, distance, ou autre chose...) des trajets possibles entre ces N villes, on doit trouver un circuit pour minimiser le coût du parcours.

Parmi les méthodes de résolution du TSP, on s'intéresse ici aux algorithmes de colonies de fourmis. Cette méthode est inspirée par le comportement de fourmis réelles dont l'action conjointe amène à trouver un chemin optimal entre deux points par la quantité de phéromone déposé sur le chemin.

Dans ce rapport, on va vous spécifier tout d'abord le structure de données, le rôle de chaque module, le prototype et rôle des fonctions essentielles, ainsi que les tests prévus. On détaillera aussi la répartition des tâches entre chaque membre ainsi que l'emploi du temps du projet.

Ensuite, on détaillera l'état du logiciel et des tests effectué. On vous expliquera comment le compiler et l'exécuter. On détaillera les optimisations et les extensions réalisées.

Nous allons également lister les problèmes rencontrés et les solutions proposées. On vous donnera le planning effectif, ce qu'on a appris de ce projet et ce qu'il reste à faire. On proposera aussi des suggestions d'amélioration du projet.

À la fin, la une conclusion donne une vue globale sur ce projet.

Nous avons également réalisé une documentation complète du projet situé dans le dossier « html ». Son utilisation est expliquée en introduction.

2. Documentation

Le projet a été entièrement documenté à l'aide de l'outil Doxygen. Il est disponible sous Windows, Linux et Mac. Cet outil permet de générer une documentation sous forme de pages web statiques. Ainsi un simple navigateur internet permet de la lire. Doxygen est uniquement nécessaire pour générer la documentation (ce qui qui a déjà été fait). Pour y accéder, la page d'accueil est le fichier « index.html » situé dans le dossier « html », mais un raccourci devrait normalement être présent dans le dossier principal du projet. Internet n'est pas nécessaire pour y accéder, vu que toutes les pages utilisées sont enregistrées avec le projet.

Cette documentation est faite pour être consultée sur un ordinateur. En effet, tous les fichiers, les fonctions, les typedefs, les structures, … sont présentés sous la forme d'hyperliens. Une description courte accompagne chacun des éléments documentés, et un bouton « plus de détail » permet d'accéder à une documentation plus complète. La page d'accueil présente le projet et donne les liens vers les éléments essentiels. Des raccourcis présents sur le haut de la page permettent de naviguer rapidement d'une partie à l'autre. Un champ de recherche est également présent. Nous vous invitons à la consulter lors de la lecture de ce rapport.

3.Spécifications

Avant de détailler le fonctionnement du programme, nous allons vous présenter son fonctionnement global.

Contrairement aux fourmis réelles, les fourmis virtuelles ont une mémoire qui leur permet de stocker le chemin parcouru à chaque cycle. Cette mémoire est « vide » au début de chaque cycle. Les fourmis virtuelles déposent une quantité d'information (phéromones) qui est proportionnelle à la qualité de la solution trouvée.

Elles ne mettent à jour les phéromones qu'une fois leur solution (ou chemin) complètement construite et évaluée, à la fin d'un cycle. L'évaporation est réalisée à chaque fin de chaque cycle, juste avant le dépôt des phéromones par les fourmis.

Lorsqu'elles sont sur une ville, les fourmis doivent décider sur quelle ville adjacente se déplacer. Alors que les fourmis réelles semblent utiliser le hasard et les phéromones, les fourmis virtuelles prennent en compte les phéromones, les villes déjà visitées et la « visibilité » des villes disponibles. La visibilité est définie comme l'inverse de la distance entre 2 villes. Au cours d'un cycle, une fourmi virtuelle ne repasse jamais par une ville déjà visitée. Si aucun trajet n'est trouvé (ce qui peut être le cas dans un graphe incomplet, le trajet est abandonné).

3.1 Données : description des structures de données

N.B : Toutes les structures de données, ainsi que les fonctions utilisées sont expliquées en détail dans la documentation annexe. Nous reprenons ici les informations essentielles.

On définit trois structures de données pour notre projet et on va vous décrire un par un dans cette partie.

3.1.1 Fourmis

On définit dans cette structure des données relative au parcours d'une fourmi.

```
typedef struct {
double L;
int nb_villes_deja_visite;
bool parcourt_valide;
Ville *tabu[];
} Fourmi;
```

La variable L représente la longueur d'un chemin, qui est la somme des longueurs de chaque arc constituant le chemin. On définit aussi un entier qui compte le nombre de villes déjà visitée. Le tableau tabu contient toutes les villes déjà parcours par la fourmi. Le booléen parcourt_valide reste vrai tant que le parcourt de la fourmi l'est.

3.1.2 Graphe

3.1.2.1 Sommet

La structure Sommet contient les données relatives à une ville.

```
typedef struct {
int id_ville;
double x,y;
char nom[64];
int nb_voisins;
Arc *voisins[];
} Sommet;
```

Pour implémenter cette structure, on définit tout d'abord l'identifiant de la ville qui nous permet de savoir son numéro, on définit aussi les coordonnées X et Y de la ville, le nom de la ville, nombre de villes qui relie avec notre ville, dernièrement, on crée une liste contenant des pointeurs vers les arcs sortant de cette ville.

3.1.2.2 Arc

On définit une structure d'arc qui contient tous les paramètres utilisé sur une arrête.

```
typedef struct {
double distance;
double pheromonesAB;
double pheromonesBA;
struct Ville *ville_A;
struct Ville *ville_B;
} Arc;
```

On définit tout d'abord la distance entre deux sommets, puis la phéromone couverte sur l'arc est spécifié. Dans cette structure, on crée aussi deux pointeurs l'un pour la ville de départ et l'autre pour la ville d'arrivée.

3.2 Modules : rôles de chaque module (couple de fichiers .c et .h)

N.B : Tous modules sont expliqués en détail dans la documentation. Nous reprenons ici les informations essentielles.

fourmi.c : Ce fichier contient toutes les fonctions concernant le parcours des fourmis.

fourmi.h : On définit dans ce fichier les prototypes des fonctions on va utiliser dans le fourmi.c.

graph.c : Contient les définitions des stuctures Sommet (Ville) et Arc, ainsi que les fonctions permettant de manipuler les arcs.

graph.h : On définit dans ce fichier les prototypes des fonctions on va utiliser dans le graph.c.

data.c : Contient les fonctions relatives à la lecture des données du graphe et à son affichage.

data.h : On définit dans ce fichier les prototypes des fonctions on va utiliser dans le data.c.

memory.c : Contient les fonctions relatives à la gestion de la mémoire.

memory.h : On définit dans ce fichier les prototypes des fonctions on va utiliser dans le fichier memory.c.

main.c : Il contient la fonction main du programme.

main.h : Il contient les paramètres de la simulation, ainsi que les options de tests. Il est inclus dans tous les fichiers du projet.

3.3 Fonctions : prototype et rôle des fonctions essentielles

N.B : De même, toutes les fonctions sont détaillées dans la documentation, et nous ne reprenons ici que les fonctions essentielles.

- void init_fourmi(Fourmi *f, Ville villes[], int nb_villes, bool deja_visite[]);

  Cette fonction initialise la structure fourmi.

- bool deja_visite(Ville *a_visiter, Ville *deja_visite[], int nb_villes_deja_visite);

  Cette fonction nous indique si la ville a déjà été visité par la fourmi.

- void ville_suivante(Fourmi *f, int alpha, int beta, double proba_ville[], bool deja_visite[]);

On déplace la fourmi dans la nouvelle ville, en fonction de sa visibilité et des phéromones sur l'arc. On calcule ici aussi la probabilité d'une ville à choisir. La ville suivante ne doit pas avoir été déjà visitée (c'est vérifié avec la fonction deja_visite).

- void parcourt(Fourmi *fourmi_actuelle, Ville villes[], int nb_villes, bool ville_visitees[], int alpha, int beta, double proba_ville[]);

  On détermine le parcourt de la fourmi actuelle, valide ce parcourt.
- bool parcourt_valide(Fourmi *f, int nb_villes, bool ville_visitees[]);

Valide le parcourt d'une fourmi en tenant compte le nombre de ville dans le fichier. Cette fonction n'est utilisé qu'en mode débug.

- void parcourt_update(Fourmi **fourmi_actuelle, Fourmi **meilleure_fourmi, int nb_villes, bool ville_visitees[], double evaporation, double depot_pheromones);

  Vérifie si le parcourt de la fourmi est valide (de manière exaustive en mode débug, et en se basant uniquement sur le booléen parcourt_valide en mode release) et met à jour le graph (évaporation + dépots des nouveaux phéromones).

- void affiche_parcourt(Fourmi *f, int nb_villes, bool ville_visitees[]);

  Affiche le parcourt d'une fourmi.

- void explore_graph(Ville villes[], Arc arcs[], Fourmi *(*fourmis[]), Fourmi *meilleure_fourmi, bool ville_visitees[], double proba_ville[], int nb_villes, int nb_fourmis, int max_cycle, double alpha, double beta, double evaporation, double depot_pheromones);

  Effectue la simulation. M fourmis vont parcourir N fois le graphe.

- Arc* get_arc(Ville *depart, Ville *arrivee);

  Renvoie l'arc qui permet de relier la ville départ avec la ville d'arrivée

- Ville* get_arrivee(Ville *depart, Arc *arc);

  Donne la ville d'arrivée, en partant de la ville départ, et en passant par l'arc.

- double* get_pheromones(Ville *depart, Arc *arc);

  Renvoie les phéromones de l'arc partant de la ville départ.

- Arc* get_in_arcs(Arc arcs[], int i);

  Permet d'accéder à l'arc numéro i.

- Ville* get_in_villes(Ville villes[], int i, int nb_villes);

  Permet d'accéder à la ville numéro i.

- void* memory_allocator(Ville *(villes[]), Arc *(arcs[]), Fourmi *(*fourmis[]), Fourmi **meilleure_fourmi, bool *(ville_visitees[]), double *(proba_ville[]), int nb_villes, int nb_arcs, int nb_fourmis);

Alloue la totalité de la mémoire nécessaire au programme en une seule fois, dans une seule zone contiguë. Cette fonction est expliquée très en détail dans la documentation du fichier « memory.h », ainsi que toutes les fonctions relatives à la manipulation mémoire qui n'ont pas été détaillées ici.

- void swap(void **p1, void **p2);

  Échange le contenu de deux pointeurs.

- void flush_line(FILE *fp);

  Passe à la ligne suivante.

- void read_villes(FILE *fp, Ville villes[], int nb_villes);

  Lit la liste des villes dans le fichier fp, et l'ajoute à la liste des villes.

- void read_arcs(FILE *fp, Arc arcs[], Ville villes[], int nb_villes, int nb_arcs);

  Lit la liste des arcs dans le fichier fp, et l'ajoute à la liste des arcs.

- void* creation_graph(const char *data_graph, Sommet *(villes[]), Arc *(arcs[]), Fourmi *(*fourmis[]), Fourmi **meilleure_fourmi, bool *(ville_visitees[]), double *(proba_ville[]), int *nb_villes, int *nb_arcs, int nb_fourmis);

  Initialise le graphe à partir des données contenu dans le fichier data_graph (et alloue la mémoire nécessaire)

- void print_arc(Arc *p_arc);

  Affiche les données d'un arc.

- void print_graph(Sommet villes[], Arc arcs[], int nb_villes, int nb_voisins);

  Affiche les données du graphe.

3.4 Tests : quels sont les tests prévus

*Teste l'ouverture des fichiers en vérifiant la valeur de retour de fopen.

*Affichage des graphes (permet de s'assurer que les données sont correctement lues).

*Teste la validité de l'allocation en vérifiant la valeur de retour de malloc (même si sous linux malloc renvoie toujours un pointeur valide).

*Teste si la ville prochaine peut être trouvé (à l'aide de la fonction ville suivante).

*Teste du meilleur chemin (à la fin du parcours de chaque fourmi, la distance parcourue est comparé à la distance parcourue par la meilleure fourmi).

*Teste si au moins un trajet a été trouvé à la fin de la simulation.

*Les fonctions sont testées à l'aide d'un débogueur, les allocations à l'aide de « valgrind » et le temps d'exécution à l'aide de la commande « time » (sous linux).

3.5 Répartition du travail et planning prévu

04/04/14(Première séance)-**17/04/14**

Commencer de comprendre le sujet en définissant les structures à utiliser et les fonctions à réaliser sans détailler.

18/04/14(Deuxième séance)-**08/05/14**

Commencer de faire des fonctions prévues et ainsi leurs prototypes.

09/05/14(Troisième séance)-**22/05/14**

Continuer d'écriture des fonctions tout en écrivant des tests et en faisant déboguer.

23/05/14(Quatrième séance)-**29/05/14**

Finir le projet complet et faire des améliorations pour des fonctions utilisées (au niveau de l'espace mémoire etc).

30/05/14 Date du rendu

4.Implantation

4.1 État du logiciel : ce qui fonctionne, ce qui ne fonctionne pas

Tous les buts sont réalisés. (Lire un fichier, créer un graph, parcourir dans les villes, chercher le meilleur chemin).

Tous les bogues actuellement rencontré ont été corrigés.

4.2 Tests effectués

Tous les tests prévus ont été réalisés.

4.3 Exemple d'exécution

Le projet peut-être compilé en mode « debug », où en mode « release » comme expliqué dans la documentation. En mode débug, tous les tests sont activés, et l'état actuel du programme est constamment notifié (il y a énormément de texte en sortie). En mode « release », le programme est compilé avec des options de compilations permettant d'avoir la vitesse d'exécution la plus élevée. Le makefile fournis permet de compiler directement avec les bons paramètres de compilation.

On peut exécuter un test en tapant :

```
cd voyageur_de_commerce
make install
./voyageur nom_du_graph_contenant_les_donnees.txt
```

La vitesse d'exécution de cet algorithme été particulièrement travaillé. Pour le tester vous pouvez la commande suivante :

```
make clean && make install && time ./voyageur
nom_du_graph_contenant_les_donnees.txt
```

Vous pouvez spécifier toutes les options de débug en modifiant les paramètres dans main.h, ou plus simplement toutes les activer à la compilation avec la commande suivante (attention, le programme devient alors très bavard !) :

```
make debug
./voyageur nom_du_graph_contenant_les_donnees.txt
```

4.4 Les optimisations et les extensions réalisées

Pendant la réalisation du projet, on a trouvé que suivant l'augmentation du nombre des villes, le temps d'exécution devenais trop long (notamment pour le graphe14.txt qui contient 100 villes), on donc optimisé le temps d'exécution en améliorant la gestion de la mémoire. On a donc créé les fichiers memory.c et memory.h qui contient les fonctions relatives sa gestion.

Son fonctionnement est détaillé dans la documentation du fichier memory.h. Tout d'abord, l'allocation de l'espace mémoire est faite en une seule fois dans une zone mémoire contiguë. De plus cette zone mémoire est allouée avec malloc (qui contrairement à calloc ne fait pas de memset sur la zone allouée). Les écritures mémoires ont étés limités au maximum. Par exemple, lorsqu'une fourmi à un parcourt plus intéressant que l'actuelle meilleure fourmi, au lieu de recopier les données, seule des pointeurs sont échangés.

De plus nous pouvons explorer des graphes incomplets (qui sont listés sur la page d'accueil de la documentation). Ils s'utilisent de la même manière que des graphes complets. Si aucune solution n'est trouvée, l'utilisateur est averti en fin de simulation.

5.Suivi

5.1 Problèmes rencontrés

Actuellement tous les bogues rencontrés ont été corrigés. Lors de la conception, les fonctions contenues dans memory.c ont dû être débogué à l'aide d'un débogueur puis testé avec Valgrind. De plus, l'ensemble des fonctions ont étés validé au fur et à mesure.

5.2 Planning effectif

Nous avons bien suivi le planning prévu et nous avons même eu un peu d'avance, ce qui nous a permis de réaliser l'optimisation du programme.

5.3 Qu'avons-nous appris ?

Nous avons appris à améliorer l'utilisation de la mémoire, la manipulation de pointeur, des structures, et à organiser le code d'un projet complet à plusieurs. Pour cela nous avons utilisés les outils git, valgrind, gdb, et time.

5.4 Suggestion et améliorations du projet

Lors de la lecture d'un fichier, on vérifie juste si le nom de du fichier correspond à ce qu'on attend mais pas la validité des données contenues. Par exemple, si dans le texte de graphe il est écrit qu'il a 50 villes mais en réalité il n'y a que 49, on ne va pas le vérifier.

Donc l'amélioration essentielle est de rajouter la vérification du contenu.

Afin d'améliorer la qualité des résultats fournis, les paramètres de la simulation ont été ajustés. Nous avons tenté d'implémenter un fonctionnement élitiste mais vu que la qualité des solutions trouvées était inférieure, nous l'avons retiré.

On peut aussi ajouter l'affichage graphique des villes et du chemin optimal qui peut apparaitre le résultat plus clairement.

6.Conclusion

Grâce aux recherches et à l'étude effectuée sur ce projet d'informatique, nous avons eu de la chance d'appliquer les connaissances acquises pendant cette année. De plus nous avons pu améliorer notre compréhension d'un sujet donné en l'implémentant sous la forme d'un programme en C.

Nous avons dû utilisés plusieurs outils pour notre projet : git pour la collaboration, un débogueur pour comprendre les bogues, valgrind pour vérifier la validité de la gestion de la mémoire, time pour mesurer le temps d'exécution Doxygen pour générer la documentation et enfin un éditeur de texte et gcc!

Comme ce projet était en binôme, nous avons vu l'importance de travailler en groupe afin de résoudre les problèmes. En effet la communication est très importante, de même que l'échange d'idées.