Rapport de projet d'informatique

Florian TAVARES – Jérémy FANGUEDE
PET-C 2011/2012

SOMMAIRE

Introduction

Spécifications

- Analyse initiale du projet
- Données : description des structures de données
- Fonctions : en-têtes et rôles des fonctions essentielles
- Tests prévus
- Répartition du travail et planning prévu

Implémentation

- Etat du logiciel
- Tests effectués
- Exemple d'éxécution
- Optimisations et extensions réalisées

Suivi

- Problèmes rencontrés
- Qu'avons-nous appris et que faudrait-il faire de plus ?
- Suggestions d'améliorations

Conclusion

Introduction

Le but de ce projet est de réaliser une application en langage C qui calcule et affiche l'itinéraire de métro le plus court entre deux stations choisies par un utilisateur. L'application s'appuie pour cela sur des fichiers CSV contenant la liste de toutes les stations parisiennes ainsi que les liaisons existant entre celles-ci.

D'un point de vue abstrait, l'ensemble des informations contenu dans ces fichiers peut être représenté sous la forme d'un graphe où chaque nœud représente une station et chaque arc un trajet possible entre deux stations. A chaque arc est associé un coût qui offre une bonne approximation de la durée nécessaire pour effectuer le trajet entre les deux stations concernées.

Spécifications

Analyse initiale du projet

Une première réflexion nous a permis de réaliser un découpage du projet entre trois grandes parties:

- une partie « Données » dont le rôle est d'assurer la lecture des informations contenues dans les fichiers CSV et leur stockage dans les structures de données choisies pour modéliser un graphe.
- une partie « Algorithmique » dont le but est de déterminer le plus court chemin du graphe ainsi créé.
- une partie « Interface utilisateur » ayant en charge aussi bien la saisie des stations de départ et d'arrivée que la sortie de l'itinéraire sous une forme simple.

A noter que si les deux premières parties semblent relativement fermées, la partie « Interface utilisateur » demeure ouverte et extensible dans le sens où elle peut aller d'une interface simple en ligne de commande à des interfaces plus évoluées comprenant des fonctionnalités plus avancées. On pense notamment ici à l'intégration d'un système permettant de diminuer l'effort nécessaire à la saisie des stations de départ et d'arrivée ou bien encore à la présence d'un rendu graphique en utilisant la bibliothèque SDL.

Données : description des structures de données

Nous avons décidé de gérer les données extraites du fichier CSV, comme cela était conseillé dans l'énoncé. Concrètement, nous avons créer une structure représentant une station qui contient son nom, son numéro (id), ses coordonnées, sa ligne et la liste des stations qui lui sont accessibles.

Nous avons donc créé une structure pour de liste pour stocker ces stations. Cette structure se nomme `Arc` et contient le numéro de la station, le coût pour y arriver et évidement un pointeur vers la structure suivante.

Voici le code C correspondant :

```c
struct Arc
{
        int num;
        double cout;
        struct Arc * next;
};
typedef struct Arc Arc;
typedef struct Arc * ListeArcs;

struct Station
{
        int num;
        double lat;
        double lon;
        char nom[TAILLE_NOM];
        char line[TAILLE_LINE];
        ListeArcs arcs;
};
typedef struct Station Station;
```

Pour représenter le graphe entier il nous suffira donc de créer un tableau de Station : `Station *plan = calloc(nbStation*sizeof(Station));`

Pour l'algorithme A star nous avons également créé des structures de données. Une première, `ListeData`, sous forme de liste, contient les données nécessaire à l'algorithme (coût depuis le départ, jusqu'à l'arrivée, ...). Une autre, `ListeSouv`, sert à retrouver le bon chemin une fois que l'algorithme est terminé, elle est aussi sous forme de liste et contient simplement le numéro de la station et le numéro de la station depuis laquelle il faut venir.

Il y a également une autre liste, `ListeRes`, qui sert à stocker le résultat et qui reprend toutes les données importante pour un affichage du résultat.

Voici le code C correspondant :

```c
struct Data
{
        int num;
        double a;
        double h;
        double c;
        struct Data * next;
};
typedef struct Data Data;
typedef struct Data * ListeData;
```

```c
struct Souvenir
{
        int num;
        int numPere;
        struct Souvenir * next;
};
typedef struct Souvenir * ListeSouv;

struct Res
{
        int num;
        double lat;
        double lon;
        char nom[TAILLE_NOM];
        char line[TAILLE_LINE];
        double coutIciToSuivant;

        struct Res * next;
};
typedef struct Res * ListeRes;
```

Une autre structure à été implémentée elle sert à faire la recherche par nom, elle contient la “ressemblance” avec le texte tapé par l’utilisateur le nom de la station quelle représente et son numéro. Cette structure est utilisé uniquement sous forme de tableau.

```c
struct SearchName
{
    int prob;
    char nom[TAILLE_NOM];
    int num;
};
typedef struct SearchName SearchName;
```

Pour réaliser un affichage propre en SDL nous avions besoin d’une autre structure qui représente un changement et non une station c’est pourquoi nous avons créer une liste `ListeChangement` qui représente cela.

```c
struct Changement
{
        char ligne[TAILLE_LIGNE];
        double cout;
        char nomDep[TAILLE_NOM];
        char nomArr[TAILLE_NOM];
        struct Changement * next;

};
typedef struct Changement * ListeChangement
```

Fonctions : En tète et rôle des fonctions essentielles

Fonctions nécessaires à la lecture du fichier (fichier.c) :

- `Station* lecture( char* nomFichier , int* nbStation);`

Cette fonction prend en argument le nom du fichier à ouvrir et le nombre de stations, qui est passé par adresse car il est utile dans la suite du programme.

Elle retourne le graphe complet des stations c'est à dire une `Station*`, un tableau de toute les stations.

- `void sePlacer( FILE * fichier, POSITION pos , int nbStation, int nbArcs);`

Cette fonction relativement simple permet une lecture aisée du fichier CSV en plaçant automatiquement le curseur au début de la liste des stations ou de la liste des arcs. Elle s'appuie sur le nombre de stations et le nombre d'arcs du fichier (fournis on le rappelle en première ligne)

Fonctions pour l'algorithme (astar.c) :

- `ListeRes aStar(int numDep, int numArr, Station * plan) ;`

C'est la fonction qui réalise l'algorithme de recherche de chemin A* ; pour cela elle se base sur le graphe des stations représenté par `Station* plan`. Elle prend également deux autres paramètres le numéro de la station de départ et le numéro de la station d'arrivée. Ces numéro sont calculés extérieurement à la fonction grâce à la fonction `nomToNum()`. La fonction `aStar()` utilise de nombreuses fonctions, notamment pour la gestion des différentes listes. Elle retourne le résultat sous la forme d'une liste de type `ListeRes` qui représente tout les "sommets" par lesquels il faut passer pour arriver à l'arrivée.

- `ListeRes reconstruire(ListeSouv souv, int numDep, int numArr, Station* plan) ;`

Cette fonction est utilisée par la fonction `aStar()`, qui l'utilise lorsqu'elle a trouvé le chemin. Elle utilise la liste `ListeSouv souv` générée par la

fonction `aStar` pour reconstruire le chemin et le mettre sous forme de `ListeRes`. Elle appelle également les fonctions `completer` et `simplifier` qui servent respectivement à compléter les données de la structure `ListeRes` et à supprimer les “correspondances” inutile au départ et à l’arrivée.

- `ListeData setData(ListeData l, int num , double a , double h) ;`

Cette fonction à deux utilités, d’abord celle standard d’ajout en tête d’une liste chaînée mais aussi celle de modifier les données de la structure de liste, si il y a déjà un maillon qui à le même numéro. Elle est utilisée pour charger des données de coût pour les listes ouvertes et fermées, dont les données peuvent être modifiées au fil des itérations.

- `int nomToNum( char * nom, Station * plan , int nbStation) ;`

C’est cette fonction qui réalise la recherche par nom, elle propose à l’utilisateur les cinq noms de stations les plus ressemblants à ce que l’utilisateur a tapé. Elle renvoie le premier numéro se trouvant dans `plan[]` correspondant au nom de station choisi par l’utilisateur. Par conséquent, lorsqu'il y a des correspondances à une station, l’algorithme A* va générer des sommets au nom identique, ils sont supprimés par la fonction `simplifier` (Cf ci-dessus). `nomToNum()` n’est pas sensible à la casse mais par contre les accents sont comptés. Pour réaliser la recherche des noms les plus ressemblant elle utilise la structure `SearchName` ainsi qu’une fonction qui effectue un tri à bulle sur un tableau de cette structure.

Fonctions de rendu Shell / SDL (affich.c) :

- `ListeChangement traitementAffichage(ListeRes resultat);`

Après réflexion, nous nous sommes rendus compte qu’il était plus clair d’obtenir un résultat dans lequel ne figure que les changements. Cette fonction transforme la liste de résultat sous forme de `ListeChangement`.

- `void afficherSDL(SDL_Surface** ecran, char** nomImages, ListeChangement l);`

Cette fonction parcourt la liste `ListeChangement l` et en plus d’afficher sur la console les stations à suivre elle les “affiche” aussi en SDL sur la

surface `SDL_Surface* ecran`, cette surface est passé par adresse car elle est modifiée par la fonction `SDL_BlitSurfaceSecure` (Cf ci-dessous). Le tableau `char ** nomImages` contient les noms des fichiers image (PNG) de toutes les lignes. Cette fonction affiche également le temps total nécéssire pour le trajet.

- `int SDL_BlitSurfaceSecure(SDL_Surface* src, SDL_Rect* srcrect, SDL_Surface **dst, SDL_Rect *dstrect);`

Cette fonction utilise en interne la fonction de la SDL dont le nom est emprunté. L'idée de créer cette fonction nous est venu en voyant que, dans la fonction SDL, lorsqu'on choisit une source de taille plus grande que la destination ou mal positionnée, une partie de celle-ci est perdue (pixels source hors zone destination). L'idée de cette fonction est, à partir des dimensions de la source, de celle la destination et de la position ou l'on veut insérer la source dans la destination, de créer un rectangle blanc dont la taille permet de contenir tous les éléments. Une fois cette surface créée, on utilise `SDL_BlitSurface` deux fois consécutives pour « coller » src et dst dessus, et on renvoie la surface ainsi créée.

Cette fonction se revèle à l'usage très pratique puisqu'elle permet d'obtenir un design qui s'adapte au contenu. Elle permet d'avoir une fenêtre dont la taille change en fonction du nombre de changements à afficher et qui est, à l'usage, très semblable, à un design à taille fixe.

Concernant la prototype, il s'agit de modifier l'adresse de dst fournie en paramètre et lui donner l'adresse la surface ainsi créée. Un passage par adresse de dst est donc impératif ! D'où le `SDL_Surface ** dst` !

Tests prévus

Dans un projet aussi imposant que celui-ci, il convient d’occuper une attention toute particulière aux tests. Il ne s’agit pas de tout coder, et de vérifier à la fin que tout marche ! Il convient d’effectuer un test des fonctions implémentées de façon individuelle (test unitaire) puis de les tester leur bonne intégration dans le ou les processus plus complexes qui l’utilise (tests d’intégration).

Le découpage du projet réalisé lors de la première séance nous a donné une meilleure vision des tests que nous aimerions effectuer.

Partie Données

Il s'agit de tester:

- la fonction `sePlacer` dans "fichier.c" pour vérifier qu'elle effectuait correctement son rôle quelque soit le fichier CSV en entrée

- les fonctions `lireStation` et `lireArc` avec des fonctions simples d'affichage de nos structures

- la bonne génération de la liste d'arc pour chaque station avec une fonction d'affichage de liste

- le tableau de ces listes d'arcs, représentation retenue pour modéliser notre graphe.

Partie Algo

Ici c'est les performances qui seront jugées. L'algorithme A* est connu pour sa rapidité mais il nous a semblé essentiel de tester par nous-mêmes qu'il était bien adapté au problème posé, au moins de manière qualitative.

Partie "Interface utilisateur"

Même si nous avons très rapidement choisi d'implémenter une interface graphique, une réalisation au préalable d'une sortie en mode console nous a semblé un bon test avant tout portage SDL.

Dans cette première version, les tests devaient essentiellement portés sur les fonctions de recherche de stations car une erreur à ce niveau empêcherait tout bonnement l'utilisateur de choisir son itinéraire et le programme perdrait tout son intérêt !

Pour la version graphique, les tests seraient plus tournés vers les fonctions offertes par SDL, afin de comprendre leur rôle et leur fonctionnement respectif. On pense ici aux tests des fonctions permettant de réaliser :

- la génération d'une fenêtre

- la génération de surfaces

- la génération de textes

- le chargement d’images

Tests finaux:

Les tests finaux devraient comporter des tests de fuites mémoire, et des tests d’utilisation poussées. Une étude comparative de nos résultats avec ceux du site ratp.fr a également été prévu, notamment afin de comparer notre algorithme A* avec celui utilisé par des professionnels et voir si les résultats diffèrent légèrement.

Répartion du travail et planning prévu

Sur ce point, il a été nécessaire de penser “la répartition du travail” afin qu’elle soit la plus efficace possible. Il fallait que chacun de nous puisse réaliser le plus de tests unitaires possible de façon indépendante afin qu’on puisse réduire le temps passé aux tests d’intégration. Il nous a semblé bon pour cela de reprendre le découpage du projet réalisé lors de la première séance. En effet, les trois parties qui en ressortent possèdent des fonctions assez différentes qui peuvent dans la plupart être testées de manière indépendante.

Données	Florian
Algorithmique	Jérémy
Interface utilisateur - Entrée Console (recherche des stations)	Jérémy
Interface utilisateur - Sortie Console (génération listeChangement)	Florian
Interface utilisateur - Portage graphique SDL	Jérémy (1/2) / Florian (1/2)

Nous avions prévu d'avoir un rendu finalisé en console en début de 3ème séance, afin de nous laisser deux séances complètes pour travailler sur la SDL.

Implémentation

État du logiciel

A l'heure actuelle, l'utilisation du logiciel n'a révélé aucun bug majeur, seulement quelques cas particuliers :

Le cas du Funiculaire de Montmartre

Montmartre Haut et Montmartre Bas sont deux stations du fichier CSV qui constituent a elle seule un graphe isolé du réseau parisien. Ansi, si l'utilisateur décide de se rendre d'une de ses deux stations à une autre station parisienne, l'algorithme A* ne marche pas (du moins il renvoie `NULL`), tout bonnement car aucun chemin du graphe n'existe entre les noeuds considérés! Nous avons choisi d'afficher un message d'erreur lorsque ce cas ce produisait. Toutefois, d'autres solutions auraient pu être envisagé. On pense notamment à une petite gestion d'exception lors de l'entrée des stations par l'utilisateur (par exemple si celui-ci choisi en station de départ Montmartre Bas alors le programme ne lui propose que la station Montmartre Haut). Une autre solution, la plus élégante et la plus logique il nous semble, est de modifier le fichier CSV en rajoutant un arc représentant une correspondance "à pied" avec une ou deux stations proches; elle nécessite d'avoir une estimation du temps nécessaire pour effectuer le trajet pour un piéton.

Le cas Station Départ = Station d'Arrivée

Le cas est géré, affiche un résultat cohérent mais pas très adapté à l'application. Pour l'instant l'itinéraire généré est un trajet de temps nul allant de la station à cette même choisi, une utilisant une des lignes qui desservent cette station. Là encore, empêcher l'utilisateur d'effectuer cette action (ou alors, sur un ton plus drôle, le laisser faire mais lui faire comprendre poliment qu'il est capable de trouver son itinéraire seul !) serait mieux adapté

Tests effectués

L'ensemble des tests initialement prévus ont été réalisé, on trouve dans le répertoire des fichiers test1.c, test2.c, testSDL.c qui contiennent

quelques un des codes de test effectués (pas tous car nous avons choisi d'écraser les codes de test une fois réussi par de nouveaux).

Exemple d'exécution

Lors du lancement du programme l'utilisateur est invité à choisir sa station de départ. Par exemple l'utilisateur veut partir de la station de métro *Bastille*, il peut donc taper dans le terminal `bas` (premières lettres de Bastille) puis appuyer sur Entrée. Le programme lui renverra la liste des cinq noms de station les plus ressemblants. L'utilisateur n'aura qu'a entré le numéro correspondant à *Bastille* le `0`.

```
Choix de la station de départ :
bas
0:  Bastille
1:  Basilique de Saint-Denis
2:  Argentine
3:  Charles de Gaulle, Étoile
4:  Champs Élysées, Clémenceau
0
Vous avez choisi Bastille comme station de départ
```

Puis l'utilisateur est invité de la même façon à choisir la station d'arrivée, par exemple il souhaite aller à l'aéroport d'Orly, il peut taper orly dans le terminal et choisir la proposition numéro 1 :

```
Choix de la station d'arrivée :
orly
0:  Orly-Ville
1:  Orly-Ouest, Aéroport d'Orly
2:  Orly-Sud
3:  Pont de Neuilly
4:  Porte Maillot
1
Vous avez choisi Orly-Ouest, Aéroport d'Orly comme station d'arrivée
```

Le programme va alors calculer le l'itinéraire le plus court et l'afficher dans le terminal.

```
Calcul aStar ...
Prendre la ligne M1 pendant 8 min, de Bastille à Châtelet Les Halles
Correspondance pour ligne B pendant 6 min
Prendre la ligne B pendant 12 min, de Châtelet Les Halles à Bourg-la-
Reine
Correspondance pour ligne B4 pendant 6 min
Prendre la ligne B4 pendant 4 min, de Bourg-la-Reine à Antony
Correspondance pour ligne Val pendant 6 min
Prendre la ligne Val pendant 6 min, de Antony à Orly-Ouest, Aéroport
d'Orly
```

Il affiche également le résultat dans une fenêtre grâce à la SDL.

Optimisations et extensions réalisées

Nous avons souhaité réaliser diverses améliorations nous avons donc implémenté deux extensions principales : la recherche par noms et l'affichage graphique. Nous n'avons malheureusement pas eu le temps d'implémenter d'autres optimisation notamment l'implémentation de listes triées pour l'algorithme A*. Toutefois, sur tous les itinéraires calculés, l'affichage est quasi immédiat si bien que cette absence ne nous a pas paru gênante.

Recherche par nom

Celle ci est implémenté dans la fonction `nomTonum`, elle propose à l'utilisateur les 5 meilleures nom correspondant.

Principe de fonctionnement : en fonction de ce qu'a tapé l'utilisateur la fonction calcule pour tout les nom de station la ressemblance grâce à la fonction `strcasecmp` et la stocke dans un tableau. Les cinq meilleurs ressemblances sont triées et proposées à l'utilisateur.

Affichage graphique

Nous avons également implémenter l'affichage graphique des résultats grâce à la bibliothèque SDL. Mais nous avons préféré afficher seulement

une liste des changements à effectuer plutôt que d’afficher les tracés sur une carte car cela nous paraissaient plus lisible. En effet, les 5% de stations qui se trouvent en périphérie lointaine restreignent les 95% de stations intra-muros à un gros pâté de points au centre de l’écran. Une solution intéressante aurait été d’implémenter un système de zoom mais cela nous a paru peu raisonnable au vu du nombre de séances accordées au projet.

Concernant la taille de la fenêtre, celle-ci s’adapte automatiquement à son contenu.

Suivi

Problèmes rencontrés

Mise à part les soucis classiques dû à la gestion des listes ou à celle de la SDL , nous n'avons pas rencontré de problèmes majeurs si ce n'est la gestion des entrées claviers. En effet nous avons dû utiliser la fonction `fgets` pour récupérer les données de l'utilisateur mais il s'est avéré que le buffer n'était pas vide et lors de la récupération suivante, nous obtenions des caractères non désirés. Nous avons donc utilisé une fonction `viderBuffer` (metro.c) afin de régler ce problème.

Qu'avons nous appris et que faudrait il de plus?

Ce projet nous a permis de nous rendre compte des difficultés liées à la programmation de gros projets.

D'un point de vue du code, on a pu se rendre compte que même si le langage C est très adapté pour les parties Algorithmique, Données et I/O Console, son utilisation dans la gestion d'une interface graphique (via notamment la bibliothèque SDL) relève du parcours du combattant. D'autres paradigmes de programmation semblent plus appropriés

Suggestion d'améliorations

De nombreuses améliorations sont possibles comme l'ajout de listes triées, l'affichage d'un plan, la gestion de l'entrée utilisateur en SDL.

Conclusion

En conclusion, nous avons réussir à fournir un programme conforme à ce que l'on imaginait, c'est-à-dire de fonctionnement assez proche de celui du site ratp.fr.

Si on compare les résultats fournies avec ce site avec ceux de notre programme, on constate quelques différences. Celles-ci concernent la durée des parcours et sur ce point, il serait peut être bon de revoir le coût des chemins ou bien la conversion cout ↔ temps. D'autre part le site prévoit également que l'usager puisse aller d'une station à l'autre à pied. Dans quelques cas, le résultat est semble-t'il plus rapide que de faire un changement de lignes au milieu du parcours. Il serait peut être intéressant de rajouter des arcs au graphe dans le fichier CSV pour nous aussi permettre à l'usager d'emprunter de tels itinéraires.