29/05/2021

Projet info S2 :

Plus court chemin dans un graphe

Hugo ROBERT et Clément TARDY

Objectif : Comparer deux méthodes pour trouver le plus court chemin et créer une interface graphique avec gestion des noms de stations.

Sommaire :

1. Implantation :

a) Etat du logiciel

Le logiciel que nous avons développé nous permet de trouver le plus court chemin entre deux sommets de tous les fichiers à notre disposition. La recherche des sommets peut se faire, soit par les numéros des sommets soit par les noms des stations avec un affichage graphique possible. Un test permet également de calculer le temps moyen nécessaire pour une recherche de plus court chemin dans un fichier donné.

Cependant, nous n'avons pas eu le temps de finir l'optimisation de la construction du graphe et des programmes de calcul du PCC.

```
DEBUT DIJKSTRA
Choisissez le nom de la station depart : Argentine
Choisissez le nom de la station arrivee : Bastille

resultat : 1
Chemin le plus court : ( 5 6 7 8 9 10 11 12 13 14 15 16 17 (Changement de ligne vers M8) 221 )
*fin*
```

figure 01 : Calcul d'un PCC

Le résultat donne les sommets que l'on doit prendre avec les changements de ligne.

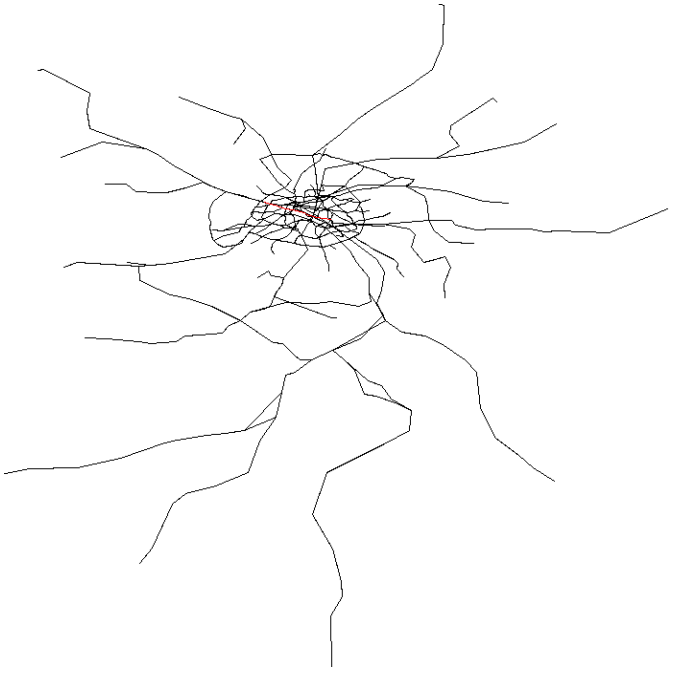

figure 02 : Résultat du projet

Voici ce que l'on obtient pour l'affichage de la carte de Paris. On remarque que le trajet en rouge correspond ici à Argentine vers Bastille.

b) Structures utilisées

Pour pouvoir développer notre programme nous avons dû créer différentes structures :

1. arcs

```c
// Type arc
typedef struct {
int arrivee; // L'indice du sommet d'arriv´ee de l'arc
double cout; // Le cout (distance) de l'arc
} edge_t;

// Type liste chaînée d'arcs
typedef struct maillon_edge {
edge_t val; // ou edge t* val, suivant votre choix : liste d'arcs ou de pointeurs d'arcs
struct maillon_edge * next;
}* listedge_t;
```

figure 03 : structure edge_t et listedge_t

Cette structure est définie dans le fichier arc.h, elle permet de créer les liaisons entre les sommets avec l'arrivée et le coût pour y aller. Il doit être utilisé avec listedge_t qui est une liste d'edge_t. La liste d'arcs est ici une liste triée dans l'ordre croissant des coûts de chaque arc.
La structure d'arc est utilisée car elle peut facilement être triée et l'accès à la plus petite donnée est donc très rapide (utile pour Dijkstra et A* car on choisit un sommet d'arrivée qui a le coût le plus faible pour un sommet de départ donné).

Fonctions implémentés : création d'un arc, compare 2 arcs, affiche un arc, création d'une liste d'arc, vérification si une liste d'arc est vide, ajout d'un arc à une liste d'arc triée, recherche du minimum d'une liste d'arc triée, calcul de la longueur d'une liste d'arcs, suppression du premier élément d'une liste d'arcs, suppression d'une liste d'arcs.

2. graphes

```c
// Type sommet
typedef struct {
int numero; // indice du sommet
char* nom; // nom donne au sommet
char* ligne; // nom de la ligne, utile uniquement pour le metro
double x,y; // coordonnees latitude et longitude du sommet
int sizeedges; //nombre d'arcs qui partent de ce sommet
listedge_t edges; // liste des arcs qui partent de ce sommet
double pcc; // valeur du "plus court chemin" entre le sommet de d´epart et ce sommet.
double cout;   //cout du sommet
int pere; //pere de la station numero (-1 si il y en a pas)
} vertex_t;

// Type graphe :
typedef struct {
int size_vertices;    // nombre de sommets
int size_egdes; // nombre d'arcs
vertex_t* data; // tableau des sommets, alloue dynamiquement
} graph_t;
```

figure 04 : structure graph_t

Cette structure est définie dans graph.h, un sommet (vertex_t) contient de nombreuses informations, ces informations sont soit lues dans un fichier texte comme pour les arcs soit calculées lors des différents algorithmes. Pour pouvoir retrouver le PCC nous avons choisi de mettre l'entier père en plus. Il nous permet à la fin de retrouver le chemin en partant de fils et en remontant de père en père jusqu'à l'arrivée.

Nous avons utilisé la structure graph_t comme proposé dans le sujet. Soit un tableau de sommets alloué dynamiquement avec le nombre d'arcs et de sommets.

Fonctions implémentées : création de graph, création de sommet, recherche de sommet, affichage de graph, affichage de sommet, suppression de graph, suppression de sommet.

```
// Type liste chaînée d'arcs
typedef struct maillon {
vertex_t * v;
struct maillon  *next;
}* list_t;
```

figure 05 : structure list_t de vertex

_____Cette structure est définie dans list.h, durant l'écriture de nos algorithmes nous avons eu besoin d'utiliser des listes de vertex_t donc nous avons développé tout un fichier permettant d'utiliser des listes de sommet. Cela sert dans nos deux algorithmes A* et Dijstra pour gérer les sommets à traiter et ceux déjà traités.

Fonctions implémentées : création de liste de sommets, test si une liste est vide, calcul de longueur d'une liste, ajout (trié ou non) d'un élément dans la liste, affichage d'une liste, recherche d'un numéro dans une liste, suppression du premier élément d'une liste, suppression d'une liste.

3. Pile d'entier

```
typedef struct _link {
  int val; /* un int de la liste*/
  struct _link *next; /* l'adresse du maillon suivant */
} link_t, *lifo_int_t;
```

figure 05 : structure de pile d'entier

Cette structure est définie dans lifo_int.h, à la fin de nos algorithmes nous avions la liste des sommets à parcourir de l'arrivée au départ. Or nous savons que les piles permettent de faire ça facilement. Donc nous avons coder les algorithmes dont nous avions besoin pour le faire.

Fonctions implémentées : création d'une pile, suppression du premier élément d'une pile, suppression d'une pile, test si une pile est vide, ajout d'un élément en tête, retourne l'élément de tête de pile, retire l'élément de tête de pile, affiche une pile

4. Fonctions de hachage

```c
typedef struct {
  keys_t key; //pour le nom de la station
  value_t value; //numero de la station
} element_hash_t;
```

Figure 06 : élément de hachage

```c
typedef struct _link1 {
  element_hash_t val; /* un élément de la liste*/
  struct _link1 *next; /* l'adresse du maillon suivant */
} link_hash_t, *list_hash_t;
```

Figure 07 : list d'élément de hachage

La table de hachage permet de retenir un numéro de sommet pour chaque station, ce qui nous permet de savoir après tout les arcs que l'on doit mettre à zéro ainsi que de connaître un sommet associé à la station que rentre l'utilisateur.

c) Tests effectués

Nous avons effectué de nombreux tests tout au long de notre programmation pour s'assurer que chaque partie fonctionnait. Voici la liste des différents tests :

Merci de déposer tous les fichiers .txt dans le dossier "text".
(Le fichier obj et text present ne doivent pas être push, il sont vide sur le depot gitlab est DOIVENT le rester)

! Pour des raisons d'optimisation, les algorithmes tel que grapheNewYork est trop long à traiter

* Test des fonctions implémenté :
  -> tester la creation & gestion de graph :
    make bin/test_graph
    ./bin/test_graph
  -> tester la creation & gestion de liste :
    make bin/test_list
    ./bin/test_list
  -> tester la creation & gestion de liste triée d'arcs:
    make bin/test_edge
    ./bin/test_edge
  -> tester la creation & gestion de la table de hashage :
    make bin/test_hashtable
    ./bin/test_hashtable
  -> tester l'ajout et la recherche d'espace dans une chaine de caractère (nécessaire pour la recherche de station par nom):
    make bin/test_space
    ./bin/test_space
  -> tester l'ouverture & la lecture d'un fichier .txt :
    make bin/test_readprint
    ./bin/test_readprint

* Test de l'algorithme final (avec le fichier graphe1.c ici, modifiable par d'autres fichiers):
  -> algorithme avec recherche par NUMERO de station et affichage sur TERMINAL
    make bin/test_algo
    ./bin/test_algo text/graphe1.txt
  -> algorithme avec recherche par NOM de station et affichage sur TERMINAL
    make bin/test_algohash
    ./bin/test_algohash text/graphe1.txt
  -> algorithme avec recherche par NOM de station et affichage GRAPHIQUE (par SDL)
    make bin/test_affichage
    ./bin/test_affichage text/graphe1.txt
  ->algorithme de test de performance par numero de station choisis au hasard
  make bin/test_algoTempo
  ./bin/test_algoTempo text/graphe1.txt

Tous les tests fonctionnent correctement, les prochaines figures sont les résultats des tests.

```
affichage vertex e1 :
1 3.000000 -1.000000 num0 nom num0
affichage graph rempli:
[
0 0.000000 0.000000 n nom n1 3.000000 -1.000000 num0 nom num02 -7.000000 0.000000 num1 nom num1]

numero de sommet de nom n (devrait afficher 0) : 0
numero de sommet de nom n0 (devrait afficher -1 qui signifie qu'il n'y en a pas) : -1
suppression graph...
*fin*
==3369==
==3369== HEAP SUMMARY:
==3369==     in use at exit: 0 bytes in 0 blocks
==3369==   total heap usage: 7 allocs, 7 frees, 1,008 bytes allocated
==3369==
==3369== All heap blocks were freed -- no leaks are possible
==3369==
==3369== For lists of detected and suppressed errors, rerun with: -s
==3369== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

figure 08 : test de graph

```
affichage liste initialise:

affichage liste avec e3:
( 0 0.000000 0.000000 n nom_n)

affichage liste avec e3 et e1 triee:
( 0 0.000000 0.000000 n nom_n ; 2 -7.000000 0.000000 num1 nom_num1)

affichage liste avec e1, e2 et e3 triee:
( 0 0.000000 0.000000 n nom_n ; 1 3.000000 -1.000000 num0 nom_num0 ; 2 -7.000000 0.000000 num1 nom_num1)

affichage taille liste (normalement 3): 3
affichage du nom de maillon de numero 2 (normalement num1) : num1
suppression de la liste
affichage liste supprimé (normalement rien):
*fin*
==3437==
==3437== HEAP SUMMARY:
==3437==     in use at exit: 0 bytes in 0 blocks
==3437==   total heap usage: 9 allocs, 9 frees, 816 bytes allocated
==3437==
==3437== All heap blocks were freed -- no leaks are possible
==3437==
==3437== For lists of detected and suppressed errors, rerun with: -s
==3437== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

figure 09 : test des listes de vertex

```
affichage arc e1 : 3,11.000000
affichage arc e2 : 1,1.000000
affichage arc e3 : 2,10.000000
affichage arc e4 : 2,0.000000
affichage liste vide :
affichage list (e1) : ( 3,11.000000 )
affichage list triee (e4,e2,e3,e1) :
( 2,0.000000 1,1.000000 2,10.000000 3,11.000000 )
minimum de la liste : 2,0.000000
taille de la liste : 4
Liberation de la liste...
taille de la liste liberee : 0
*fin*
==3480==
==3480== HEAP SUMMARY:
==3480==     in use at exit: 0 bytes in 0 blocks
==3480==   total heap usage: 4 allocs, 4 frees, 96 bytes allocated
==3480==
==3480== All heap blocks were freed -- no leaks are possible
==3480==
==3480== For lists of detected and suppressed errors, rerun with: -s
==3480== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[phelma@localhost tardyclroberthuprojetS2]$
```

figure 10 : test de la structure edge

```
affichage table de hachage au début :
[  ( station_3 : 4;  ) ( station_2 : 3;  station 1 : 2;  )   ]
ajout de 'station 1' de valeur 1 et de 'station_2' de valeur 5 :
[  ( station_3 : 4;  ) ( station_2 : 5;  station 1 : 1;  )   ]
affichage table de hachage finale :
[  ( station_3 : 4;  ) ( station_2 : 5;  station 1 : 1;  )   ]

la valeur associe a station 1 est : 1
la valeur associe a station_2 est : 5
la valeur associe a station_4 est : -1
==3521==
==3521== HEAP SUMMARY:
==3521==     in use at exit: 0 bytes in 0 blocks
==3521==   total heap usage: 11 allocs, 11 frees, 186 bytes allocated
==3521==
==3521== All heap blocks were freed -- no leaks are possible
==3521==
==3521== For lists of detected and suppressed errors, rerun with: -s
==3521== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[phelma@localhost tardyclroberthuprojetS2]$
```

figure 11 : test de la table de hachage

```
affichage du graph : [
0 0.500000 0.950000 M1  Aaa
1 0.100000 0.700000 M1  Baa
2 0.500000 0.700000 M1  Caa
3 0.900000 0.700000 M1  Daa
4 0.100000 0.350000 M1  Eaa
5 0.500000 0.350000 M1  Faa
6 0.900000 0.350000 M1  Gaa
7 0.100000 0.050000 M1  Haa
]

affichage de la liste des voisin de 2 : ( 5,10.000000 3,10.000000 )
suppression graph et liste...
*fin*
```

figure 12 : test de la lecture de fichier (ici graphe1.txt)

Nous pouvons voir que dans chaque cas nous obtenons le résultat voulu ce qui vérifie notre code. Maintenant pour aller plus loin il faut regarder les performances.

d)Analyse des performances

Nous avons choisis de caractériser la performance de nos programmes par deux critères, le premier est la rapidité et le second est la mémoire.

Pour la rapidité nous avons créé un test qui permet de faire plusieurs essais sur les algorithmes est de calculer la moyenne de temps pour effectuer chaque test, et le temps total pour effectuer l'ensemble des tests. Les résultats sont les suivants (certains fichiers étaient trop long à traiter) :

Nom du fichier	Temps construction graphe (s)	Temps recherche par Dijkstra (s)	Temps recherche par A* (s)
Graphe1 (8 sommets)	0.000036	0.000003	0.000003

Graphe2 (14 sommets)	0.000046	0.000006	0.000006
metroetu (732 sommets)	0.002212	0.000761	0.000787
grapheNewYork (264’346 sommets)	0.483477	92.960658	6.918825
grapheColorado (435’666 sommets)	0.820548	12.891988	1.095242
grapheFloride (1’070’376 sommets)	2.003613	-	-
grapheMonde (851’205 sommets)	2.481122	-	-
grapheGrandLacs (2’758’119 sommets)	5.469736	-	-
grapheUSAOuest (6’262’104 sommets)	13.890062	-	-
grapheUSACentral (14’081’816sommets)	36.305709	-	-

Remaque :

- Nous n’avons pas pu mesurer le temps nécessaire pour les cases avec un tiret
- il n’est pas aberrant que grapheColorado prenne plus de temps de recherche de pcc que grapheNewYork car Colorado a moins de sommets mais plus d’arcs (donc plus dense))

On remarque donc que de l’optimisation est encore nécessaire pour atteindre les mêmes performances que ceux donnés en figure 5 du sujet du projet. De Plus, pour des fichiers dense (grapheNewYork), c’est là que le programme A* est bien plus performant que Dijkstra, en revanche pour des fichiers peu dense (metroetu) Dijkstra est légèrement plus performant que A*.

Pour la mémoire et les erreurs nous avons utilisé la commande Valgrind qui permet de voir si l’espace mémoire alloué est correctement libéré. Dans la quasi totalité des tests il n’y a pas de fuite. Pour le test de temps, il y a des erreurs qui apparaissent avec valgrind uniquement quand on printf la variable moyenne_temps (test algoTempo). De même nous avons des erreurs avec la fonction add_space (test algohash). Ce sont les deux problèmes que nous n’arrivons pas à comprendre. Et pour finir il y a aussi les erreurs habituelles pour la SDL(test affichage). hormis ces erreurs, l’ensemble des fichiers test fonctionnent sans fuite mémoire, erreurs...

2. Suivi

a) Organisation de l'équipe

Ce projet demande une organisation complexe puisque 2 personnes travaillent sur le même code. Donc pour qu’il n’y ai pas de problème, nous avons choisi de faire une conversation sur messenger pour pouvoir parler facilement sur le

projet. Puis nous avons décidé de créer un drive google pour pouvoir partager des documents autre que du code. Nous avons commencé par créer un fichier qui regroupe notre avancée et toutes les tâches qui doivent être effectuées. Mais aussi toutes les structures de données qui doivent être implémentées. Pour se mettre d'accord nous avons fait une réunion et dans cette même réunion nous avons défini des message de commit pour se comprendre lorsque que l'on continue le travail de l'autre.

Nous nous sommes aussi mis d'accord sur le planning, le travail doit être presque terminé pour le 25/04. De là nous avons décidé aussi qui devait faire quoi. Mais si l'un d'entre nous voulait coder car nous aimons, il pouvait aussi prendre les parties de l'autre. L'objectif était ainsi de se répartir le travail par journée et par fichier pour éviter les problèmes de git. Les tests ont été implémentés au fur et à mesure de la création des différents fichiers .h pour s'assurer du bon fonctionnement de chacun des modules. Nous avons essayé de suivre une certaine logique d'indentation sur les fichier (surtout les .h) pour être compréhensible par le binôme. Hugo a fait les différents programmes pour calculer le PPC et les lectures de fichiers. Clément a fait la table de hachage, les différentes listes et piles. Les tests ont été fait à deux comme la SDL.

b) Outils de développement

Nous avons choisi d'utiliser Atom pour rédiger nos programmes car il est facile d'utilisation et GIT fonctionne correctement dessus. Nous avons codé principalement sur des machines Apple donc sous MACOS. Pour faire tous les tests nous sommes passés sur la machine virtuelle car la commande Valgrind y est présente.

Pour caractériser nos différents programmes et les tester nous avons donc utilisé la commande valgrind et le module time.

c) Problèmes rencontrés et solution

Nous avons rencontré plusieurs problèmes, nous allons les diviser en plusieurs sous parties.

1. Recherche par station :

Pour une recherche avec les noms des stations il faut prendre en compte que plusieurs sommets peuvent être à la même station. Dans un premier temps nous avons pensé à récupérer tous les sommets de départ et tous ceux d'arrivée pour ensuite faire un calcul du PCC entre chacune. Or cette méthode est longue donc

nous avons opté pour mettre tous les arcs à 0 entre les sommets de la station de départ et de même pour celle d'arrivée.

2. Lecture des noms de stations :

Nous nous sommes rendu compte que lorsque nous lisions le fichier pour créer le graphe il lisait aussi des espaces pour le nom de la station. Or lorsque l'on veut faire une recherche avec la table de hachage si nous ne mettons pas les espaces il ne trouvera pas la station. Donc nous avons choisi de créer tout un programme qui ajoute le bon nombre d'espace devant le mot, or il dépend de chaque fichier donc nous le calculons à chaque fois. Quand l'utilisateur rentre son mot nous ajoutons les espaces puis nous faisons la recherche sur la table.

3. Numéro de station :

Quelques heures avant la date limite de rendu du projet (lors de l'analyse des performances) nous nous sommes rendu compte que pour de gros fichiers, le numéro de station dépassait la valeur max que peut prendre un int. Par conséquent, le programme ne fonctionnait pas pour les plus gros fichiers à notre disposition. Il suffirait de remplacer les int par des long (et le %d par des %lu) malheureusement nous n'avons pas eu le temps de le faire.

d) Conclusion

Nous remarquons que les 2 programmes ont des comportements différents : A* est plus efficace sur des fichiers dense, et à l'inverse Dijkstra est plus efficace sur des fichiers de faible densité (c'est-à-dire avec un rapport nombre arcs/nombre sommet faible). De l'optimisation est encore nécessaire pour être efficace sur les plus gros fichiers, il faut aussi remplacer certaines variables int par des long. A part cela les différents tests fonctionnent avec de nombreuses options possible pour l'utilisateur : recherche par Dijkstra ou A*, recherche par nom ou numéro de station, avec affichage graphique ou non.

Ce projet nous a permis d'appréhender le travail d'équipe pour de la programmation. C'est une méthode de travail différente des autres projets (en électronique, projet de 1ere année…) car ici, il faut s'organiser pour travailler sur des fichiers différents et si possible à des moments différents pour s'assurer que chacun de nos fichiers puissent s'assembler et ainsi permettre de faire avancer l'autre grâce à des fonctions simple, claire et fiable. Cette organisation une fois acquise nous a permis de gagner beaucoup de temps pour la création des derniers tests. Ce projet nous a également permis d'être plus rigoureux sur la création de nos tests car sur des projets plus ambitieux que pour des BE, ceux ci prennent tous leurs sens pour comprendre en amont les problèmes de structure de programme avant de s'attaquer à des test qui utilisent de nombreuses fonctions et qui par conséquent doivent être "irréprochables". Enfin, ce projet fut l'occasion de découvrir de nouveaux modules (notamment le module time très utile).