Rapport du projet d'informatique

Arbre couvrant minimal

Notions: Graphes,gestion de projet développement logiciel

Semestre S2-PET 3
Guenoun Omar,Kazdaghli Meriem,Sabri Taha
2023-2024

Sommaire:

I/Introduction

Le sujet du plus court chemin dans un graphe qui vise à minimiser le coût du chemin qui relie les sommets d'un graphe est très abordé dans l'informatique et il a plusieurs variantes et algorithmes dont chacun a ses propres avantages et inconvénients. L'utilité du problème de plus court chemin se retrouve dans la vie réelle notamment dans le domaine des réseaux. En effet, notre projet s'inscrit dans le cadre de la planification de l'installation de la fibre optique tout en réduisant son coût. C'est pour cela que qu'on a choisi la notion de l'ACM(Arbre Couvrant minimal)pour répondre à notre enjeu.
La réalisation de l'ACM sera faite dans un premier lieu avec l'algorithme Prim et dans un deuxième lieu avec l'algorithme Kruskal,comme indiqué dans l'énoncé. Nous procéderons à différents tests pour s'assurer de leur fonctionnalité tout en justifiant les structures de données utilisées. Ensuite on passera à l'analyse des performances CPU et mémoire pour comparer l'efficacité des deux algorithmes de même que leur complexité théorique et expérimentale. Une autre partie du rapport va expliciter les outils de développement ainsi que l'organisation suivie par l'équipe pour bien mener le projet. Enfin on conclura sur ce qu'on a appris et sur une critique qu'on peut porter.

II/Implantation:

a)Etat du logiciel:

A la fin de ce projet, on constate que notre programme marche avec quelques petites différences entre l'algorithme de Kruskal et Prim qui seront explicitées dans ce qui suit. Ainsi, on a réussi à remplir notre cahier de charges pour ce projet de programmation.

b)Structure de données utilisées:

Algo:**Prim**	
Structure	Justification
tableau d'entiers alloué dynamiquement: int* pour l'ensemble **Atteint**	Pour pouvoir libérer la mémoire à la fin du programme puisque il y'a des graphes de très grandes tailles, De plus il s'est avérer qu'utiliser un tableau binaire est la façon la plus optimisée pour procéder
tableau d'entiers alloué dynamiquement:int* pour l'ensemble **Atraiter**	*
tableau d'arcs alloué dynamiquement: edge_t* pour **previous_edge**	même problème de mémoire + stocker les pères du sommets
tableau alloué dynamiquement:double* pour le **coût**	même problème de mémoire + stocker le coût des sommets
Algo:**Kruskal**	
Structure	Justification
tableau d'entiers pour Union find	Le tableau d'entier est très facile à manipuler et nécessite peu de mémoire

c)Fonctions utilisées:
On décrit dans le tableau les fonctions qu'on a utilisées tout en justifiant notre choix, mis à part les fonctions déjà fournies dans le sujet.

Fonction	Justification du choix
Algo:Prim	
`int atraiter_est_vide(int * atraiter, int n)`	Cette fonction retourne 1 si le tableau atraiter est vide. Elle permet de vérifier qu'on a parcouru tous les sommets ajoutés à atraiter.
`int min_cost_atraiter(int *atraiter,double *cost, int n)`	permet de retourner le plus faible coût d'un sommet dans la tableau atraiter
list_edge_t Prim(graph_t graph)	permet de retourner la liste d'arcs trouvés,c'est à dire le plus court chemin qui relie les sommets d'un graphe,par l'algo Prim.
vertex.c	
`int sommets_différents(vertex_t e1,vertex_t e2)`	compare deux sommets à partir de leurs

	champs et retourne faux si ils ne sont pas égaux
graph.c	
`int indice(vertex_t e, graph_t g)`	retourne la position d'un sommet dans un graphe
`vertex_t sommet_départ(graph_t g)`	renvoie le sommet de plus faible coût, donc qui sera le sommet de départ pour l'algo prim.C'est une façon d'optimiser l'exécution de Prim au lieu de commencer par un sommet aléatoire
`read_graph.c`	
`read_graph.c`	
`Algo:Kruskal`	
`list_edge_t Kruskal(graph_t g)`	permet de retourner la liste d'arcs trouvés,c'est à dire le plus court chemin qui relie les sommets d'un graphe,par l'algo Kruskal
`Union_find.c`	
`int* new_connexe(graph_t g)`	cette fonction nous permet d'initialiser notre structure Union-Find (tab[i]=i)
int recherche_representant(int i,int* tab);	recherche_representant nous renvoie l'indice du représentant du sommet d'indice i dans le tableau tab.
void fusion(int a, int b, int* tab);	cette fonction sert à fusionner deux connexes d'un tableau d'indices de sommets tab.
Dans ce qui suit , on est menés selon l'algorithme de Kruskal à trouver une méthode à complexité convenable pour pouvoir trier tous les arcs du graphe g (qu'on a mis dans une liste chainée à l'aide de prendre_edge). Et vu qu'on a opté pour une structure de liste chaînée on a pensé alors à utiliser un tri classique de complexité nlog(n) où n est le nombre d'arcs, il s'agit du Tri par fusion.	

`list_edge_t prendre_edge(graph_t g)`	*cette fonction a le rôle simple de mettre tous les arcs du graphe g dans une liste chaînée;*
list_edge_t fusionner(list_edge_t l1, list_edge_t l2);	*Ici on fusionne deux listes chaînées d'arcs en tenant compte de l'ordre croissant du coût de l'arc (en d'autres termes la valeur : l1→val.cost).*
*list_edge_t tri_fusion_edge(list_edge_t *l);*	*qui est implémentée selon l'algorithme de tri par fusion où on divise la liste chaque fois par deux et on fusionne les deux moitiés de façon ordonnée et récursive (d'où la complexité nlog(n) de ce tri.)*

3/Méthodologie des tests:

on était conformes aux consignes du sujet. on a pris soin de tester chacune des fonctions intermédiaires utilisées avant de passer au programme final. D'abord on a testé les fonctions déjà fournies avant de passer à celles qu'on a ajoutées

4/Analyse performances CPU et mémoire:

Une fois qu'on a testé les fonctions intermédiaires utilisées, la justesse des deux algo Prim et Kruskal, il est temps de voir le temps d'exécution de nos deux algos ainsi que l'espace mémoire consommé sur des exemples de graphes.

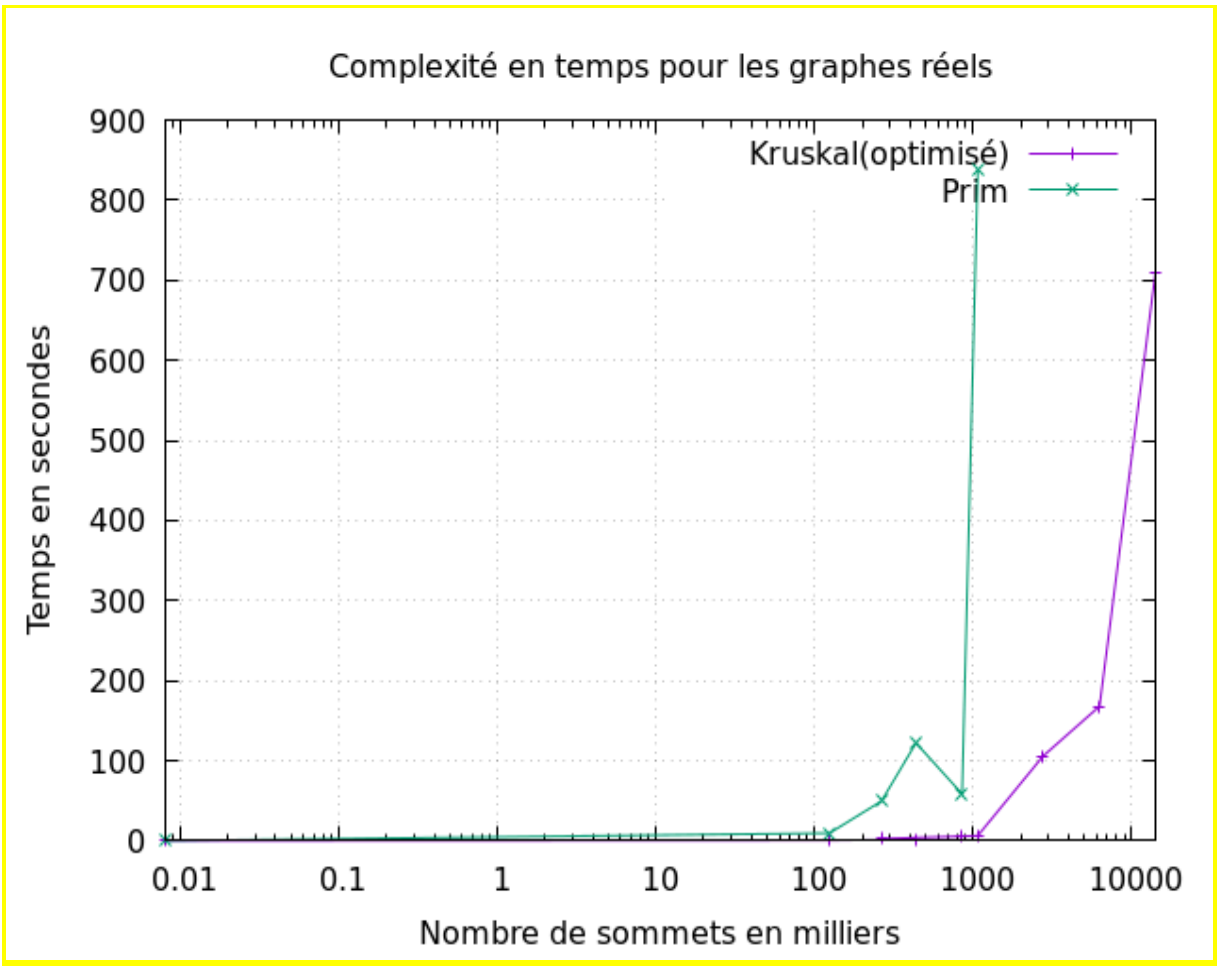
a)temps d'exécution

Grâce à la fonction clock fournie dans l'énoncé et qu'on a adapté pour notre programme, on a pu calculer le temps d'exécution en secondes, d'abord sur des petits graphes,ensuite sur des graphes routiers.

Graphes	**Prim**	**Kruskal(lecture des arêtes en un seul sens)**
graphe0.txt	0.000015	0.000017
graphe1.txt	0.000012	0.000033
graphe2.txt	0.000031	0.000031
grapheAmericas	9.37	0.54
grapheColorado	121.97	1.65
grapheFloride	839.24	5.31

grapheGrandlacs	N/A	105.65
grapheMonde	58.43	5.5
grapheNewYork	49.56	2.48
grapheUSAcentral	N/A	708
grapheUSA ouest	N/A	167.88

NB: pour les trois graphes restant vu leur volume très grand ainsi que la vitesse de notre algorithme Prim on n'a pas pu les traiter.



Dans cette partie, on a remarqué une propriété intéressante dans l'algorithme de Kruskal vis-à-vis de celui de Prim. Il s'agit de l'immunité contre l'orientation des arcs (edges), en fait , avec read_graph Kruskal n'a besoin que de lire les arêtes dans leur sens original décrit dans les fichiers .txt pour fonctionner tandis que Prim a nécessité de stocker deux arcs différents (sens inversés) dans les liste_edge associées au sommets ce qui implique ainsi un usage

plus exhaustif de la mémoire du PC ainsi qu'un temps d'exécution plus long. Cette hypothèse est confirmée par la mesure du temps d'exécution de Prim et Kruskal sur grapheColorado.txt :
lecture des arêtes dans un seul sens —>: Prim ne marche pas et Kruskal prend 1.65 secondes.
lecture des arêtes dans deux sens —>: Prim marche et prend 433 secondes vs Kruskal qui prend 13.5 secondes.

et donc on peut conclure le grand avantage de Kruskal par rapport à Prim surtout lorsque ça concerne des fichiers volumineux avec des millions de sommets et d'arêtes.

b)mémoire
En ce qui concerne la gestion de mémoire, on a vérifié par le biais de la commande valgrind que nos programmes ne génèrent pas de fuite de mémoire et ceci depuis les fonctions intermédiaires qu'on a utilisées. Ensuite, on a comparé l'espace mémoire consommé par l'algo Prim par rapport à celui de Kruskal
En s'appuyant sur des exemples de graphes,on a constaté que Prim utilise moins d'espace mémoire que Kruskal:

	Prim(MB)	Kruskal(MB)
graphe1.txt	0.007920	0.008272
graphe2.txt	0.009309	0.0010261
grapheAmericas	28.24	44.9
grapheColorado	56.79	135.3
grapheNewYork	86.35	90.96

c)complexité théorique

D'après notre recherche, on trouve que ces deux algorithmes doivent avoir une complexité O(n log(n)) avec n le nombre de sommets. Cette hypothèse est d'ailleurs confirmée par l'allure du graphe précédent pour Kruskal et presque de même pour Prim.

5/Optimisation:
Avant d'aboutir aux structures qu'on a utilisées dans l'algo Prim on a procédé différemment. En effet, on a créé une structure de liste de sommets pour les ensembles Atraiter et Atteint. Et ce qui nous a menés à écrire de nombreuses fonctions qui manipulent les listes:

```
list_vertex_t list_vertex_new()
int list_vertex_is_empty(list_vertex_t l)
list_vertex_t list_vertex_add_first(list_vertex_t l, vertex_t e)
list_vertex_t list_vertex_add_last(list_vertex_t l,vertex_t e )
list_vertex_t list_vertex_del_first( list_vertex_t l )
```

```
int non_dans_liste(list_vertex_t l,vertex_t e)
list_vertex_t orderedlist_vertex_add(list_vertex_t l,vertex_t e, double *
cost,graph_t g)
```

Puis on a procédé à la détermination de la complexité théorique de ces fonctions pour arriver à se décider sur le bon choix à faire entre tableaux et listes.
En effet,l'accès à un élément de la liste est un O(n) car il nécessite le parcours de la liste depuis le début jusqu'à l'élément souhaité.
L'ajout à la fin: O(1)
La suppression:O(1)

On a conclu que les tableaux sont plus efficaces dans l'accès direct à un élément spécifique.Mais ce qui a surtout tranché notre choix c'est la gestion de mémoire,(moins de fonctions utiles pour les tableaux) et aussi le temps d'exécution de l'algo Prim:
on a pris un graphe très simple(graphe0.txt fourni dans le git)et on mesuré le temps d'exécution par la fonction "…….." et on a trouvé:

En ce qui concerne l'algorithme de Kruskal, on a décidé de jouer sur la vitesse du tri vu qu'on a pensé au début à un tri par insertion de complexité n au carré au pire des cas ce qui risque de prendre beaucoup de temps.Et alors on a choisi d'opter pour le tri par fusion adapté au listes chaînées pour avoir la complexité de nlog(n) qui est d'ailleurs la complexité qu'on cherche à avoir pour notre algorithme de kruskal.
Faudrait-il ajouter qu'on a pensé aussi à changer la méthode de vérification du graphe qui est un parcours en boucle for du tableaux Union-find et donc d'une complexité n mais vu cette valeur alors on reste toujours dans un algorithme à complexité de O(nlog(n)).

II/Suivi

1/Outils de développement utilisés:

On utilise l'ensemble des outils suivants:

Éditeur: Vscode
Constructeur: Make
Compilateur: Clang
Gestion de mémoire : Valgrind
Debuggeur: Gdb

2/Organisation au sein de l'équipe:

Dès la première séance du projet, on a fait un planning qu'on a suivi pour bien mener le projet. Et on s'est fixé des délais pour chaque partie et objectif.

NUMÉRO	TITRE DE LA TÂCHE	ACTEURS	PCT DE LA TÂCHE TERMINÉ	Avril		Mai			
				1	2	3	4	5	6
1	**Définition et planification du projet**								
1	compréhension du projet	Groupe entier	100%						
2	identifier les modules,les fonctions et tests prévus	Groupe entier	100%						
3	répartition des tâches	Groupe entier	100%						
4	planning	Groupe entier	100%						
2									
1	fonctions sur les listes+tests	Kruskal : Taha Prim: Meriem+ Omar	100%						
2	Fonction de lecture	Taha	100%						
3	Implémentation de Prim+Test	Meriem+Omar	100%						
4	Implémentation de Kruskal+test	Taha	100%						
5			100%						
6									
7									
8									
9									
3	**Validation/tests du projet**								
1	vérification de l'ACM	Omar et Taha							
2	finaliser le progr	KAZDAGHLI Meriem							
3	tests sur des réseaux routiers	Omar et Taha							
4									
5									
6									
4	**Performance/suivi du projet**								
1	Analyse des performances+t aille mémoire	TAHA	100%						
2	rédaction du rapport	grande partie: Meriem et le reste: Taha+Omar							
3									

3/Notes proposées:
Vu notre répartition équitable du travail ainsi que l'implication et le sens de responsabilité de chacun d'entre nous, on propose de diviser les 30 points entre nous et par suite chacun aura 10 points.

III/Conclusion:

En termes de cahier de charges, il s'avère qu'on a réussi à achever tout les objectifs de notre projet sauf la vérification d'acm pour l'algorithme de Prim, à savoir l'implantation des deux algorithmes de Kruskal et de Prim ainsi que la comparaison de leurs performances CPU et mémoire.

Ce projet a aussi contribué d'une façon importante à notre capacité du travail au sein d'une équipe en améliorant notre communication, notre manière de planning et notre adaptabilité à différents obstacles tout au long de ce projet et par suite le mener au succès.
On tient aussi à remercier les professeurs pour leur encadrement et pour leur disponibilité pour donner de l'aide sans oublier leurs efforts menés pour faire des séances au début de ce projet.