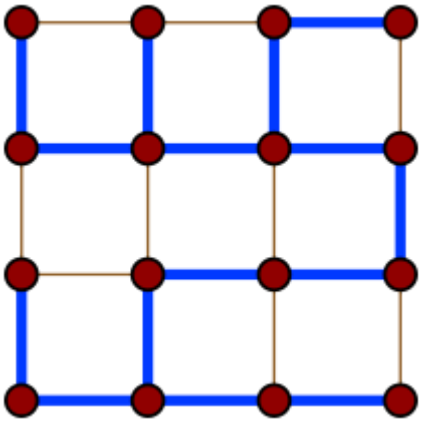
Rapport – Projet Info S6



*Recherche de l'**A**rbre **C**ouvrant **M**inimal*

PET 3 – Kenzo Cusinato, Ben Yates-Smith, Alice Lorido

Introduction : Recherche d'un **A**rbre **C**ouvrant **M**inimal

Lors de ce projet, nous devions coder deux algorithmes permettant de rechercher l'ACM dans un graphe quelconque. Pour mener à bien ce projet, nous devions donc correctement nous répartir les tâches en fonction de nos forces, nos faiblesses et nos envies. Il y a également plusieurs points importants à aborder : - il faut déterminer la complexité des algorithmes et la minimiser, étant donné qu'on travaille parfois sur des graphes immenses. (c.f partie 2.5) Il est également nécessaire de tester les deux algorithmes (Prim et Kruskal) avec des fonctions test adaptées. C'est pour celà qu'il y a un dossier /test qui permet de tester les différentes fonctionnalités de chaque algorithme.

Dans ce rapport, nous aborderons en premier lieu la méthode d'organisation mise en place par notre groupe de projet, puis nous verrons une partie sur l'algorithme de kruskal, sa réalisation, ses performances, et ses tests.
Puis, nous nous pencherons sur l'algorithme de Prim, sa complexité et ses tests. Enfin, nous conclurons.

— rapport : contient les documents tels que document initial de conception, rapport, fichier de suivi .Rédiger de manière claire et concise les r´esultats obtenus : le rapport est clair, bien écrit, concis et explicite tous les ´el´ements exprim´es dans le sujet/cahier des charges. 8.6.1 Rapport Vous ferez un rapport court (5 à 10 pages) au format PDF explicitant les points suivants : 1. Implantation (a) Etat du logiciel : ce qui fonctionne, ce qui ne fonctionne pas parmi les fonctionnalités demandées ; (b) Justification des structures de donn´ees que vous avez choisies (excepté celles donn´ees pour les graphes) ; (c) Methodologie utilisee pour les tests, commandes qui permettent de tester les fonctionnalités demandées ; (d) Analyse des performances CPU et mémoire, Complexité´e th´eorique et exp´erimentale ; 2. Suivi ; (a) Outils de développement utilis´es ; (b) Organisation au sein de l'équipe, Réunions et Planning effectif, qui a fait quoi ; (c) Notes proposées pour chacun des membres de l'équipe en les justifiant à l'aide du fichier de suivi. La somme des notes de l'´equipe doit ˆetre ´egale `a 30 points ; 3. Conclusion.

1 Outils Utilisés

1.1 Méthode d'organisation AGILE

Dans le cadre de notre projet d'informatique, il était impératif d'avoir une méthode d'organisation fiable et robuste pour être préparés à toute éventualité. C'est ainsi que nous avons décidé ensemble de la méthode d'organisation la plus adaptée à notre groupe et au projet. Avec l'aide du module d'accompagnement professionnel, (MAP) nous avons pris connaissance d'une méthode d'organisation plutôt efficace, la méthode AGILE. En voici le tableau final :

MODÈLE DE PLAN DE PROJET AGILE

NOM DU PROJET	CHEF DE PROJETS	DATE DE DÉBUT	AVANCEMENT GÉNÉRAL
Projet Informatique S2	Ben	15-avr.	étape 9/10

LIVRABLE DU PROJET	Calculer efficacement l'arbre couvrant minimal
ÉNONCÉ DU PÉRIMÈTRE	Professionel

RISQUÉ	NOM DE LA TÂCHE	TYPE DE FONCTIONNALITÉ	RESPONSABLE(S)	DÉBUT	FIN	DURÉE (EN JOURS)	STATUT	COMMENTAIRES
						0		
non	Structure de Données	Mise en place de la structure de données servant aux deux algorithmes	Alice	29-avr.	2-mai	3	Terminée	
non	Fonction sur les Listes	Création de fonctions de tri et autre pour Kruskal	Ben+Kenzo	3-mai	12-mai	9	Terminée	
oui	Lecture + Affichage Graphe	Affichage et chargement en mémoire d'un graphe à partir d'un fichier .txt	Alice	6-mai	13-mai	7	Terminée	
						14	Terminée	
non	Structure Union Find	Fonction tierce pour réaliser l'algorithme de Kruskal	Ben+Kenzo	17-mai	24-mai	7	Terminée	
non	Fonctions sur les Listes	Création de fonctions de tri et autre pour Prim	Alice	15-mai	18-mai	3	Terminée	
oui	Fonctions test Prim	Réalisation de fonctions vérifiant les exigences de Prim	Alice	18-mai	27-mai	9	Terminée	
oui	Fonctions test Kruskal	Réalisation de fonctions vérifiant les exigences de Kruskal	Ben	23-mai	28-mai	5	Terminée	
						13	Terminée	
non	Prim	Réalisation et optimisation de l'algorithme de Prim	Alice	17-mai	1-juin	15	En retard	
non	Kruskal	Réalisation et optimisation de l'algorithme de Kruskal	Kenzo + Ben	21-mai	29-mai	8	Terminée	
non	Rapport	Ecriture du rapport	Ben	24-mai	2-juin	9	Terminée	

STATUT CLÉ
Non commencée
En cours
Terminée
En attente
En retard

2 Algorithme de kruskal

2.1 Tri par ordre croissant

Pour appliquer l'algorithme de Kruskal, avant toutes choses, il est nécessaire de trier les arêtes par ordre de coût croissant. Ainsi, c'est par cette étape que nous avons débuté l'implémentation de l'algorithme de Kruskal avant de faire la structure Union-find.

Notre objectif initial était d'obtenir une structure fonctionnelle avant de nous plonger dans la complexité des différents types d'algorithmes de tri. Ainsi, nous avons créé un tri simple par ordre croissant. L'idée était de parcourir chaque liste chaînée de chaque sommet et, pour chaque maillon de la liste représentant une arête, de le placer au bon endroit dans une liste de tri par ordre croissant. De plus, afin d'éviter de refaire de l'allocation dynamique, nous avons décidé de laisser les listes préexistantes en place et de simplement modifier les valeurs des pointeurs de ces listes.

Bien que notre tri initial soit fonctionnel, sa complexité était de O(n^2), ce qui est très élevé, surtout lorsque les graphes deviennent très grands. Par conséquent, nous avons décidé d'utiliser un autre type d'algorithme de tri.

Tri par Fusion (Merge Sort)
Nous avons opté pour l'algorithme de tri par fusion, ou "merge sort" en anglais. Nous avons choisi cet algorithme en raison de sa complexité, qui est au pire de O(n log n) et au mieux de O(n log n), ce qui le rend plus efficace pour les grands graphes.

Le tri par fusion est un algorithme de tri basé sur la stratégie diviser-pour-régner. Il suit trois étapes principales : la division, la récursion et la fusion.

Division : L'algorithme commence par diviser la liste en deux moitiés. Cette étape de division est répétée récursivement sur chaque moitié jusqu'à ce que chaque sous-liste contient un seul élément ou soit vide.
Récursion: Une fois les listes divisées jusqu'à leur plus petite taille (un élément), l'algorithme applique récursivement le tri par fusion à chaque sous-liste. Cette phase de récursion permet de traiter et de trier les sous-listes de manière indépendante avant de les fusionner.

Fusion: La dernière étape consiste à fusionner les sous-listes triées pour former une seule liste triée par ordre croissant.

La raison pour laquelle le tri par fusion a une complexité en O(nlog(n)) s'explique par le fait que la liste est divisée en deux à chaque étape, ce qui donne un facteur de log(n), et que la fusion des sous-listes prend un temps linéaire O(n). Il est également important de noter que la complexité spatiale est O(n) car un espace supplémentaire est nécessaire pour stocker les sous-listes temporaires pendant la fusion.

Pour implémenter ce tri, nous avons décidé de ne pas utiliser cet algorithme sous forme de tableau, mais plutôt sous forme de liste. Cette décision a été prise car notre tri précédent fonctionnait avec des listes et nous voulions conserver cette même structure dans notre nouvel algorithme de tri. De plus, cela nous évite de devoir allouer de la mémoire pour créer un tableau.

Cependant, cette approche présente aussi des inconvénients. Par exemple, il est impossible d'accéder directement à un élément de la liste comme on pourrait le faire avec un tableau ; il faut accéder aux éléments successivement. Cela rend le code légèrement plus lent que la version utilisant des tableaux.

De plus, un autre problème auquel nous n'avions pas pensé est le risque de débordement de pile lorsque le graphe à traiter est très grand. Nous avons donc dû modifier la fonction pour que la partie récursive du programme se fasse de manière itérative. Cela nous a obligés à créer une fonction utilisant une boucle `while(1)` avec des `break`. Bien que ce ne soit pas la méthode la plus élégante, elle est tout à fait fonctionnelle.

2.2 Union-find

2.2.1 Implémentation

Une fois que nous avons une liste triée par ordre croissant, il est alors possible d'utiliser la structure union-find pour mener à bien l'algorithme de Kruskal. Pour implémenter cet algorithme, nous avons d'abord décidé de modifier la structure "vertex_t" en y ajoutant les champs suivants : int nb_sommet_dans_ensemble et "*struct vertex* parent*".

Le champ "*nb_sommet_dans_ensemble*" nous permet de connaître le nombre de sommets dans chaque ensemble, ce qui permet non seulement de savoir si tous les sommets ont été atteints, mais aussi de déterminer quel ensemble est plus grand qu'un autre.
Le champ "*parent*" nous permet de retrouver le représentant d'un ensemble.

Afin de mener à bien la structure Union-find nous l'avons séparé en 4 fonctions distinctes

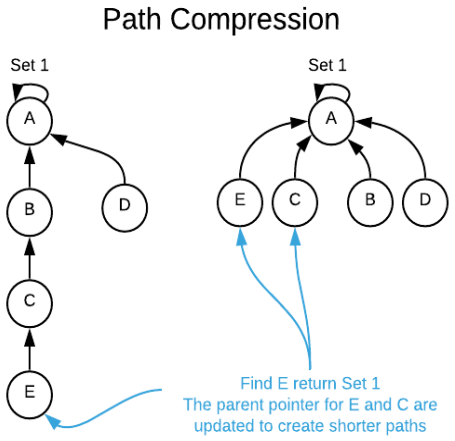
1. **vertex_t* find_parent(vertex_t* sommet):**
   Cette fonction permet de trouver le parent d'un sommet et de remonter jusqu'à ce que l'on trouve le représentant de l'ensemble.
2. **long int fusion_deux_ensembles(vertex_t* root_depart, vertex_t* root_arrival)** :
   Cette fonction permet de fusionner deux ensembles distincts s'ils n'ont pas le même représentant. Le représentant de l'ensemble le plus petit pointe vers le représentant de l'ensemble le plus grand, et on met à jour le nombre de sommets dans l'ensemble fusionné.

3. **list_edge_t union_find(graph_t graph, list_edge_t sorted_edges):**
   Cette fonction détermine les représentants des deux sommets d'une même arête. Si ces représentants sont différents, elle les fusionne en utilisant la fonction "fusion_deux_ensembles", puis ajoute l'arête correspondante à une nouvelle liste `acm` (Arbre de Couverture Minimum).

4. **list_edge_t kruskal(graph_t graph)** :
   Cette fonction principale implémente l'algorithme de Kruskal. Elle initialise chaque sommet en définissant "nb_sommet_dans_ensemble" à 1 et en faisant pointer `parent` sur lui-même. Ensuite, elle appelle la fonction "union_find" pour construire l'ACM. Enfin, elle libère la mémoire utilisée pour stocker les arêtes triées et retourne l'ACM.

2.2.2 Complexité de l'Union-find

Afin d'améliorer l'efficacité de l'algorithme Union-Find, nous avons utilisé la compression de chemin dans la fonction find_parent. La compression de chemin permet d'obtenir une très bonne complexité car chaque sommet pointe directement sur le représentant de son ensemble, évitant ainsi de devoir parcourir une longue chaîne de sommets pour trouver le représentant.



Pour mettre en place la compression de chemin, nous avons utilisé la récursivité. La compression de chemin fonctionne très bien de manière récursive, même avec des graphes de grande taille, car chaque appel récursif modifie la structure de l'arbre pour le rendre plus plat, ce qui réduit la profondeur des appels futurs. Ainsi, bien qu'il soit possible d'implémenter cette fonction de manière itérative, la version récursive est généralement suffisante et efficace.

L'algorithme Union-Find, lorsqu'il est optimisé avec la compression de chemin et l'union par rang, a une complexité presque constante. Cette complexité est souvent notée O(α(n)), où α(n) est la fonction d'Ackermann inverse. La fonction d'Ackermann inverse est une fonction mathématique qui croît extrêmement lentement. Pour toutes les applications pratiques, cette complexité peut être considérée comme O(1), c'est-à-dire constante. En d'autres termes, même pour des très grands ensembles, le temps nécessaire pour exécuter les opérations Union-Find reste très faible et ne dépend presque pas de la taille de l'ensemble.

2.3 Complexité totale

combiner les complexités des différentes étapes :

- Création de la liste des arêtes : O(V+E)O(V + E)O(V+E)
- Tri des arêtes : O(ElogE)
- Initialisation des ensembles : O(V)
- Construction de l'ACM avec Union-Find : O(E⋅α(V))O(E \cdot \alpha(V))O(E⋅α(V))

En combinant ces étapes, la complexité totale de l'algorithme de Kruskal est dominée par le tri des arêtes et la construction de l'ACM :

O(V+E)+O(ElogE)+O(V)+O(E·α(V)). Puisque α(V) est très petite et peut être considérée comme une constante dans les applications pratiques, la complexité totale simplifiée est :

O(ElogE+V) ce qui donne alors pour des graphes connectés **O(ElogE).**

2.4 Fichier test

Afin de tester le fonctionnement de l'algorithme de Kruskal, nous avons mis en place un fichier de test permettant de vérifier différentes parties de l'algorithme. Ce fichier de test comprend quatre fonctions de test et une fonction principale.

- **long int test_kruskal(graph_t graph_toTest)**
- **void test_sorted_ascending_graph_fusion(graph_t graph_toTest)**
- **void test_each_vertex_reached(list_edge_t acm, graph_t graph_toTest)**
- **void test_fusion_deux_ensembles(graph_t graph_toTest, long int id_vertex1, long int id_vertex2)`**

Dans toutes ces fonctions de test, nous utilisons `assert` pour vérifier que le code se comporte comme prévu et arrêter le programme si ce n'est pas le cas.

Parmi ces tests, la fonction test_each_vertex_reached est, selon nous, la plus importante, car elle permet de vérifier si l'algorithme de Kruskal a une chance d'avoir correctement trouvé un arbre couvrant minimal. Si chaque sommet n'est pas atteint, il est alors certain que l'ACM trouvé n'est pas correct, sauf si, bien sûr, tous les sommets ne sont pas connectés entre eux.

Pour vérifier que chaque sommet a été atteint, nous partons du dernier sommet et trouvons son représentant à l'aide de la fonction find_parent. Si la variable nb_sommet_dans_ensemble du représentant est égale au nombre total de sommets dans le graphe, nous pouvons être certains que tous les sommets ont été atteints et que notre algorithme a des chances de fonctionner correctement.

En résumé, ces tests permettent de s'assurer que :

1. Les arêtes du graphe sont correctement triées.

2. Chaque sommet du graphe est bien connecté dans l'ACM.

3. Les ensembles sont correctement fusionnés.

4. L'algorithme de Kruskal fonctionne comme prévu.

En utilisant ces tests, nous avons pu valider la robustesse et la validité de notre implémentation de l'algorithme de Kruskal.

2.5 Performances CPU et mémoire

Pour évaluer la qualité de notre code, on mesure la complexité pratique en traçant le temps de calcul de l'ACM en fonction du nombre de sommets de nos graphes.Les mesures ont été faites sur un ordinateur de référence MSIGL65156, processeur Intel Core I7.

Dans un premier temps relevé les temps de calcul pour certains graphes proposés, comme graph0.txt, graph1.txt, graph2.txt, grapheColorado.txt et enfin graphNewYork.txt. Dans un deuxième temps, on a généré des graphes aléatoires à l'aide du fichier programs/gen_graph_file.c qui contient un code C permettant de générer des graphes avec des nombres de sommets quelconques et un nombre de voisins donné pour chaque sommet. Il s'exécute avec la commande : bin/gen_graph_file *graine aleatoire seuil distance Nombre sommet Degre sommet Nom fichier*

Nous avons fixé des paramètres comme la graine aléatoire et le degré de sommet. De plus, le sujet nous indiquait que le seuil distance devait être de l'ordre de 3 à 10 pour des "petits graphes", et qu'il fallait le fixer à 10000 pour des "grands graphes" pour éviter des temps trop longs. On a donc cherché l'ordre de grandeur des graphes pour les réseaux routiers :

Un "grand graphe" pour les réseaux routiers peut être de l'ordre de :

- **Réseau urbain** : De 10,000 à 100,000 sommets.
- **Réseau régional** : De 50,000 à 500,000 sommets.
- **Réseau national** : De 500,000 à plusieurs millions de sommets.
- **Réseau continental/mondial** : De 5 millions à 50 millions de sommets ou plus.

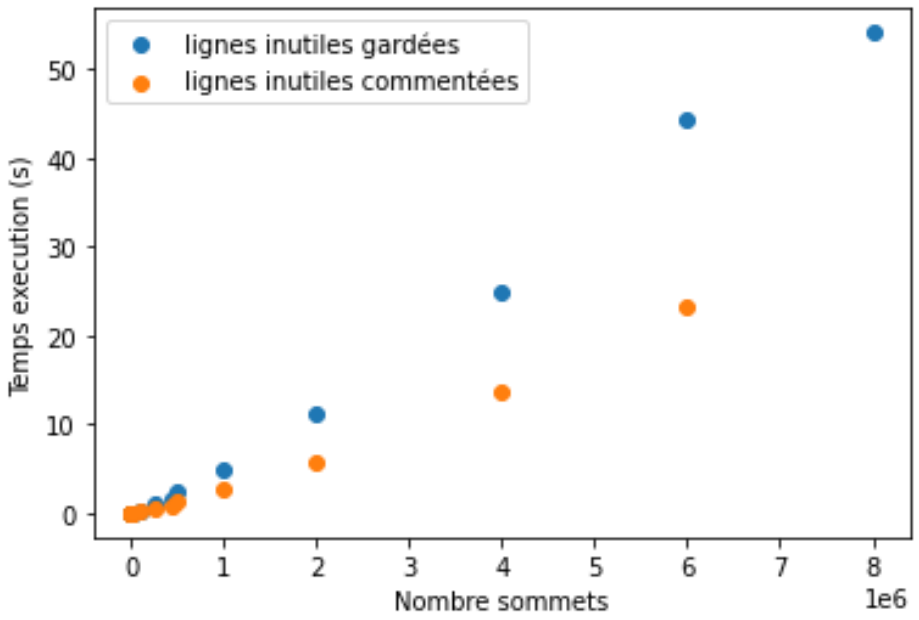
Dans le cas de notre sujet, on a décidé que les graphes étaient considérés de grande taille lorsqu'ils traduisent des réseaux routiers à l'échelle d'une région (ou état comme le Colorado, Floride…) : on a donc fixé à 10 le seuil distance pour un nombre de sommet inférieur à 100 000, et à 10000 pour des graphes avec plus de 100 000 sommets.

On a également implémenté le code pour mesurer le temps donné p.14 du sujet, dans le main de test_kruscal.c, en laissant toutes les lignes de code de la fonction test_kruscal(graph_t graph_toTest). Puis, on a décidé de commenter les lignes inutiles (comme les printfs…) pour voir si l'on pouvait gagner en rapidité de calcul, en ne gardant que l'appel de la fonction kruskal

Nombre sommets	Temps d'exécution (s) sans commenter	Temps d'exécution (s) en commentant	Nom fichier
8	0.000056	0.000008	graphe0.txt
8	0.000023	0.000009	graphe1.txt
14	0.000026	0.00001	graphe2.txt
264346	1.168244	0.469652	grapheNewYork
435666	1.568431	0.827234	grapheColorado

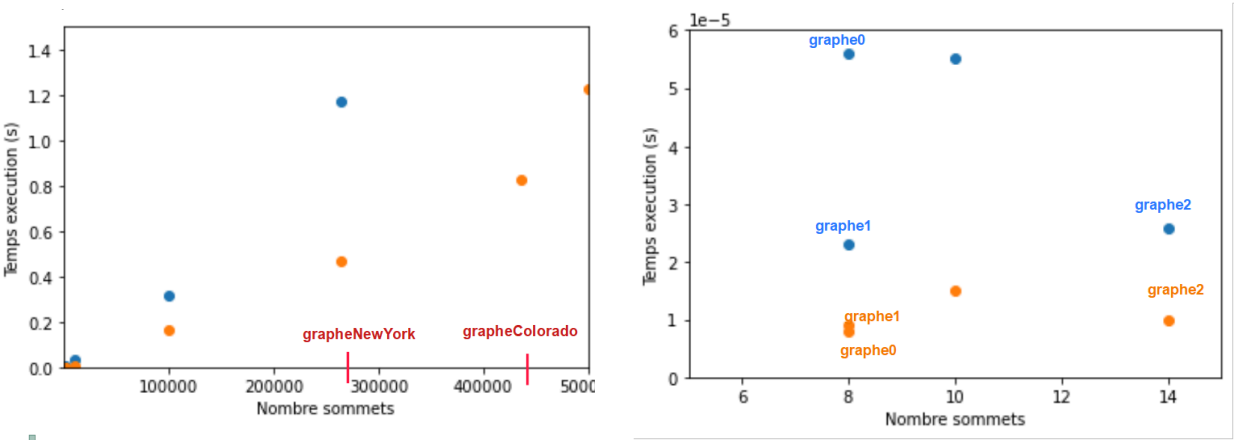
Nombre sommet	Temps d'exécution (s) Sans commenter	Temps d'exécution (s) En commentant	Graine aléatoire	Seuil Distance	Degré sommet	Nom Fichier
10	0.000055	0.000015	1	10	3	gen_graph_10vertex.txt
100	0.000193	0.000056	1	10	3	gen_graph_100vertex.txt
1000	0.002641	0.000951	1	10	3	gen_graph_1000vertex.txt
10000	0.031466	0.008283	1	10	3	gen_graph_10000vertex.txt
100000	0.316879	0.163302	1	10	3	gen_graph_100000vertex.txt
500000	2.309959	1.229752	1	10000	3	gen_graph_500000vertex.txt
1000000	5.033164	2.762247	1	10000	3	gen_graph_1000000vertex.txt
2000000	11.240752	5.798656	1	10000	3	gen_graph_2000000vertex.txt
4000000	24.767741	13.594285	1	10000	3	gen_graph_4000000vertex.txt
6000000	44.412381	23.180499	1	10000	3	gen_graph_6000000vertex.txt
8000000	54.119239		1	10000	3	gen_graph_8000000vertex.txt

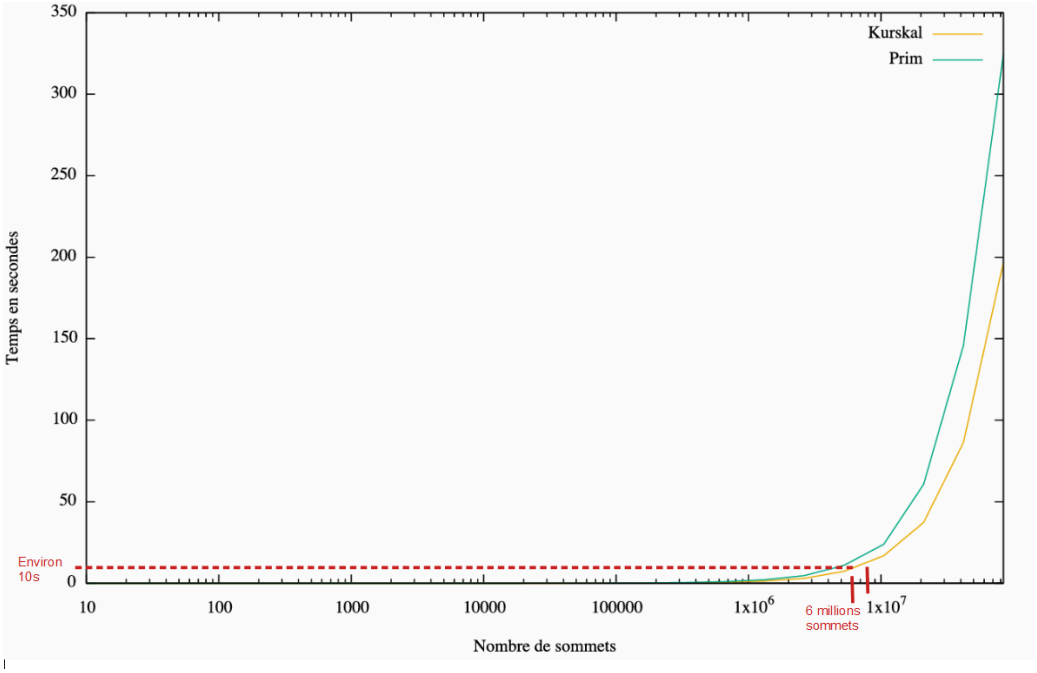
On obtient alors les courbes suivantes



Observation : on constate que le temps d'exécution a été divisé par 2 environ ! (non négligeable)

On réalise une zoom sur la fenêtre [0,500000] pour voir les graphes du Colorado et de Newyork; et [5,15] pour voir graphe0,graphe1 et graphe2:

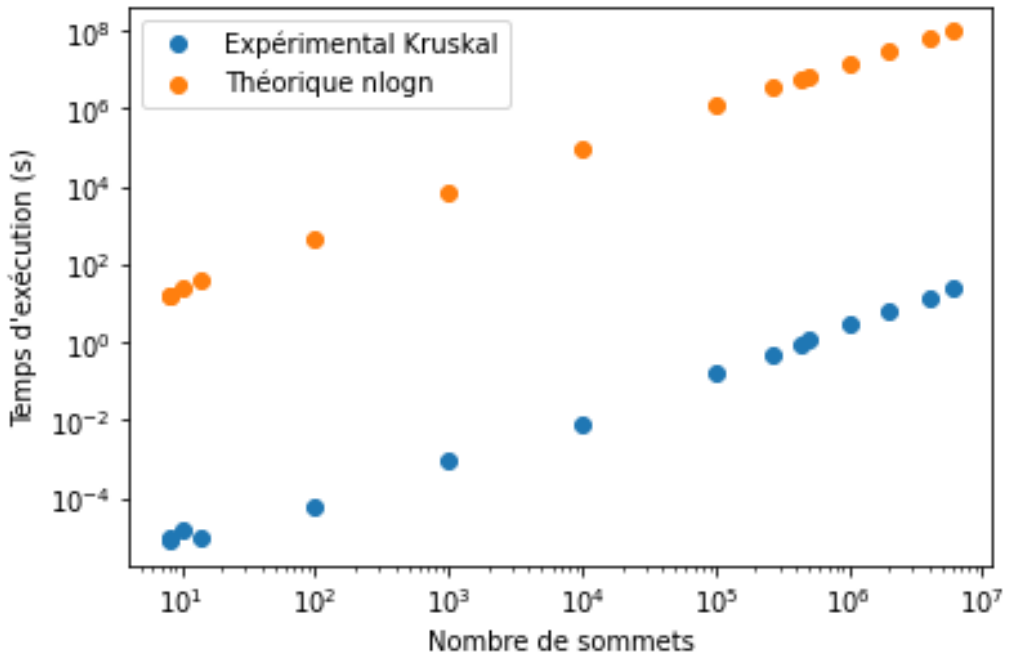




Le dernier graphe est celui donné par le sujet, pour un ACM de 3 voisins (comme nous car on a fixé degré sommet = 3). On voit que les temps d'exécutions sont relativement proches, par exemple, pour 6 millions de sommets, le graphe donné trouve environ 10s, et nous on trouve 23s.

Certes, il y a une différence, mais cela dépend de beaucoup de facteurs, comme la machine sur laquelle on travaille et ses performances (caractéristiques du matériel,charge du système, système d'exploitation), ainsi que la complexité temporelle des algorithmes.

En notant n le nombre de sommets, on obtient bien une complexité en nlog(n).



3. Algorithme de Prim

3.1 Choix des structures de données

Pour le bon fonctionnement de l'algorithme de Prim, il est nécessaire d'introduire des changements spécifiques aux structures déjà établies et d'en rajouter certaines.

sommet.h include/sommet.h

```c
#ifndef _SOMMET_H
#define _SOMMET_H

#include "list_edge.h"

// Type sommet : propriétés du sommet
typedef struct vertex {
    char* name; // nom donné au somme
    double x,y; // coordonnées latitu
    long int id, cout;
    edge_t previous_edge;
    list_edge_t edges; // liste des a
    //ce qui suit est pour kruscal
    long int nb_sommet_dans_ensemble;
    struct vertex* parent;
} vertex_t;
```

Ici, par exemple on a l'ajout de "id" pour mieux identifier sommets (visibilité dans l'écriture de Prim)

Et "cout", nécessaire la construction de l'ACM.

Il y a également l'introduction de la structure de liste de sommets, similaire à celle des arêtes dans son fonctionnement. Lors de l'algorithme De Prim, il faut en effet deux listes :
Atraiter et Atteint. Nous avons fait le choix d'introduire cette structure pour pouvoir avoir des fonctions sur les listes de sommets.

Dans la structure des sommets, on a également introduit un *edge_t* previous_edge, pour la construction de l'ACM dans la fin de l'algorithme De Prim.

list_sommet.h include/list_sommet.h

```c
#ifndef _LIST_SOMMET_H
#define _LIST_SOMMET_H

#include "sommet.h"



// Définitinon du type liste de sommets
typedef struct _link_vertex {
  vertex_t sommet;
  struct _link_vertex *next;
} * list_vertex_t;

// Création d'une liste vide
list_vertex_t list_sommet_new();

// Retourne true si la liste l est vide
int list_sommet_is_empty(list_vertex_t l);

// Ajoute le sommet s à la liste l
// et retourne la nouvelle liste,
// ou NULL en cas d'erreur
list_vertex_t list_sommet_add_first(list_vertex_t l, vertex_t s);

list_vertex_t list_sommet_add_last(list_vertex_t l, vertex_t s);

// recherche du sommet avec le cout minimal parmis une liste de s
// puis, retourne ce sommet
vertex_t list_sommet_rech_cout_min(list_vertex_t l);

// retourne 1 si le sommet v appartient a la liste l
int list_sommet_appartient(vertex_t v, list_vertex_t l);

#endif
```

3.1.1 Fonctions essentielles : list sommet

- *list_vertex_t* **list_sommet_add_first(list_vertex_t l, vertex_t s)**
- *list_vertex_t* **list_sommet_add_last(list_vertex_t l, vertex_t s)**
- *vertex_t* **list_sommet_rech_cout_min(list_vertex_t l)**
- *int* **list_sommet_appartient(vertex_t v, list_vertex_t l)**

Les deux premières fonctions ont pour but de gérer les additions des sommets dans Atraiter et Atteint. (listes de sommets dans l'algorithme de prim)

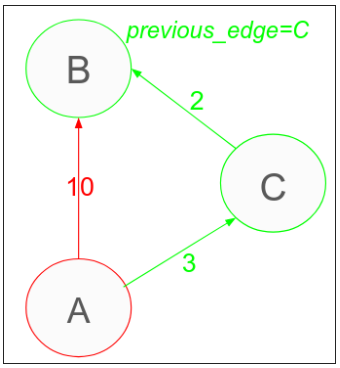
La troisième permet d'avoir le sommet de coût minimal parmis une liste de sommets, cette fonction sert à sélectionner le prochain sommet dans Atraiter.

Et la dernière fonction nous permet d'ajouter la condition que le sommet d'arrivée d'un arc ne soit pas déjà dans Atteint, le tout dans la boucle principale de l'algorithme de Prim.

3.2 Construction de la liste des previous_edge

Comme dit dans la partie 2.1, nous avons introduit previous_edge dans la structure des sommets pour nous permettre de construire la liste d'arêtes correspondant à l'ACM à la fin de l'algorithme de Prim.

Lorsqu'un nouveau chemin de coût minimal allant d'un Sommet A vers un Sommet B est trouvé. Le previous edge de B vaut C : le sommet intermédiaire par lequel il est moins coûteux de passer, en répétant celà pour tous les sommets on est en mesure d'obtenir la liste des arêtes de l'arbre couvrant minimal. En effet, c'est juste la liste des previous_edge des différents sommets.



3.3 Complexité de Prim

S=nb_sommets

Pour l'algorithme de Prim (acm_prim2), on utilise add_reverse_edges en complexité de O(S*max(graph->data[i].edges)) + O(S) + (dans le pire des cas) O(S*S)

Donc dans le pire des cas, $O(S^2)$.

3.4 Fichiers Test pour Prim

Nous avons introduit le fichier *test/05_acm_prim.c* pour tester l'algorithme de Prim sur les trois graphes qui nous ont été fournis. On a également un fichier *test/test_prim2.c* pour les tests sur le prim2, qui trouve l'ACM en ayant également l'arête de retour dans un graphe.

Dans ce fichier, on a une fonction qui teste de manière générale si l'algorithme de Prim se comporte correctement, puis on a également une fonction qui teste si tous les sommets du graphe sont atteints par la liste de l'ACM.

3.4.1 Sommet de départ

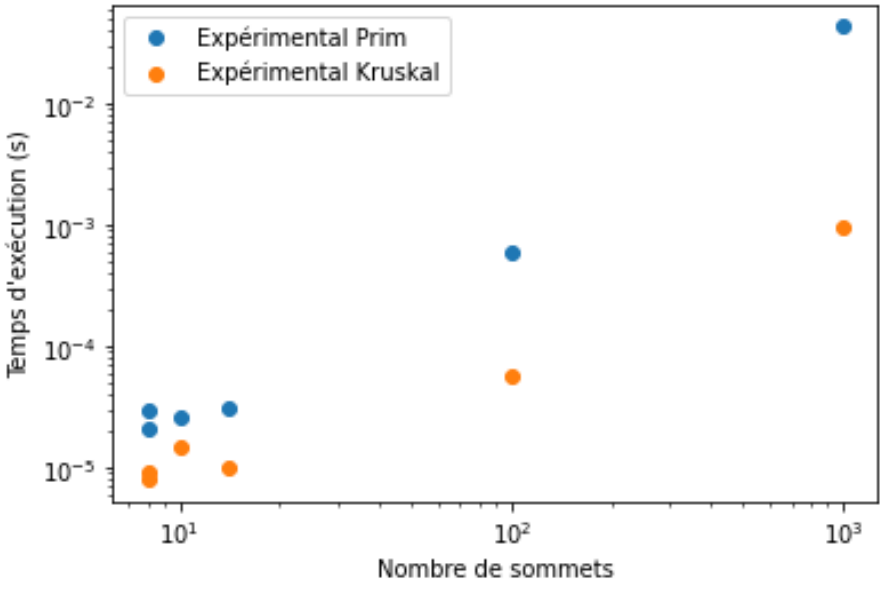
Il était nécessaire dans ce fichier d'avoir une méthode permettant de savoir le sommet par lequel on devait faire démarrer l'algorithme de Prim. En effet, sinon il y aurait un grave problème : des sommets de peuvent êtres atteints, en choisissant le mauvais sommet de départ, on est tributaire de la possibilité d'inexistence de l'acm. C'est pour celà qu'on se sert de la fonction ⇒ list_edge_t *sorted_ascending_graph_merge*(graph_t graph) ⇐

Car on peut accéder au sommet optimal facilement, c'est le sommet d'arrivée de la première arrête de la liste en return de la fonction précédente.

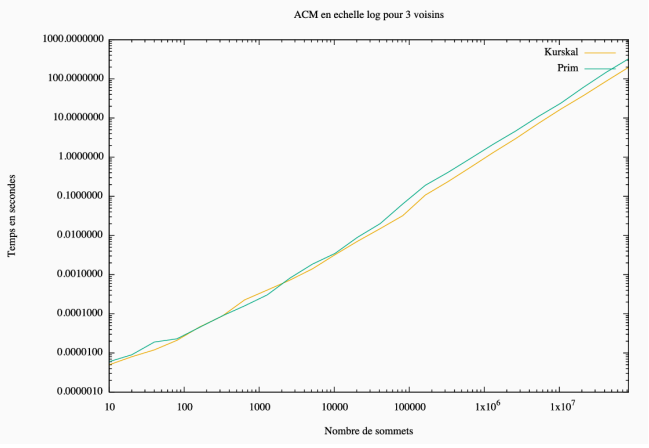
3.4.2 Mesure des performances du temps d'exécution

L'algorithme de Prim que nous avons implémenté fonctionne, mais pour des valeurs de graphes assez faibles. En effet, au-dessus de 10 000 sommets, nous rencontrons des problèmes de "heap overflow" et "edge new: invalid parameters". Cela rallonge de beaucoup le temps de calcul (pour 10 000 sommets, on avait environ 3s par rapport à 0.008283s !).

On compare alors les deux algorithmes pour des valeurs allant de 8 à 1000 sommets avec la même génération de graphe aléatoire que dans la partie 2.5 : (en échelle log)



On voit que globalement, l'allure est la même que celle proposée par le sujet, et que Prim reste moins efficace que Kruskal.



4. Bilan et conclusion

L'un des aspects les plus marquants de ce projet a été la collaboration entre les membres de l'équipe. Chacun d'entre nous a apporté une contribution unique et essentielle au bon fonctionnement du projet. Cette diversité de compétences a permis d'enrichir notre travail et d'atteindre nos objectifs plus rapidement et efficacement. La répartition des tâches a été cruciale pour s'assurer que chaque aspect du projet était couvert par les membres les plus qualifiés et les plus motivés pour ces tâches spécifiques.En travaillant ensemble, nous avons pu partager nos connaissances et apprendre les uns des autres. Ce partage constant a favorisé une ambiance de travail stimulante et motivante. Chaque membre de l'équipe a ainsi pu se sentir valorisé et important, contribuant à maintenir un haut niveau d'engagement et de performance.

Nous sommes fiers d'avoir produit un projet complet, qui a traité de plusieurs thématiques sur deux algorithmes différents, en comparant pour chacun comment il a fallu faire usage des structures de données vues lors du semestre 6.

Notes :

Benjamin : 12

Kenzo : 9

Alice : 9