Projet 3 - Groupe A1

Aydin Matya 37592100 | Dallemagne Brieuc 77122100 | Lebras Floriane 35022100 | Mounir Samy 46422100 | Van Hees Charles 35562100 | Verschuere Louise 33772100

I. Introduction

Dans le cadre du projet P3, il nous a été demandé d'implémenter en langage C, sur base d'un code Python existant, l'algorithme du plus court chemin de Bellman-Ford dans un graphe orienté pondéré. Dans un premier temps, cela a été réalisé en *single-threaded*. Nous présentons maintenant notre implémentation en *multi-threaded*.

Ce rapport contient tout d'abord un commentaire sur le programme que nous avons implémenté. Ensuite, nous comparons la consommation de temps, de mémoire et d'énergie de notre implémentation en langage C avec le code Python fourni. Après cette vision globale, nous présentons les améliorations faites à l'algorithme pour diminuer la consommation de mémoire instantanée et de temps. Avant de conclure, nous présentons notre dynamique de groupe.

II. Description du code C

A. Structure du code

Le code a été réalisé en utilisant quatre modules :

- `sp` : contient la fonction `main` ainsi que les fonctions et structures liées aux arguments pris par le programme ;
- `file` : contient les fonctions relatives à la lecture et à la fermeture de fichier ;
- `graph` : contient les fonctions liées à l'algorithme de Bellman-Ford ;
- `thread` : contient les fonctions relatives aux threads.

Les fichiers sources de ces modules sont contenus dans le répertoire `src` et les header correspondants dans le dossier `include`. Dans le dossier `include` se trouvent aussi trois autres headers : `portable_semaphore.h` et `portable_endian.h` sont nécessaires au fonctionnement du code sur Apple, et `include_and_struct.h`, importe les librairies et définit les structures utilisées dans plusieurs fichiers. Ce dernier header est importé dans les autres fichiers. Le dossier `tests` contient les graphes utilisés pour faire les tests ainsi que le code des tests dans le fichier `tests.c`.

B. Structure de données

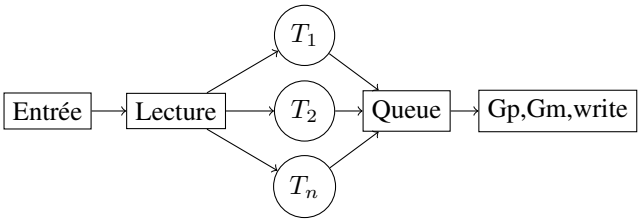
Étant donné que le langage C ne permet pas qu'une fonction renvoie plusieurs éléments, nous avons défini des structures dans lesquelles nous lions les résultats de la fonction. Par exemple, la fonction `bellman_ford` prend un pointeur vers une structure `result_bellman_ford_t` en argument, dont les variables sont `source`, `dist`, `path`, avec `dist` et `path` les résultats de l'algorithme de Bellman-Ford appliqué au noeud `source`. Nous avons procédé similairement pour `get_path` et `get_max`.

C. Multithreading

Nous étions initialement partis sur un modèle producteur-consommateur avec un buffer de taille fixe. Nous nous sommes finalement tournés vers un modèle qui utilise une structure de donnée chaînée de type *queue* car son fonctionnement coïncide avec l'aspect que nous avons voulu optimiser : la mémoire. Deux sémaphores sont utilisés pour faire le lien entre le producteur et le consommateur. Un premier fait attendre le consommateur jusqu'à ce qu'il y ait un élément dans la queue. Le second limite la taille de la queue pour limiter la mémoire instantanée, comme expliqué à la section amélioration du code. Si une erreur est détectée, tous les threads se terminent et la mémoire est libérée.

1) Producteur: Les threads "producteurs" exécutent l'algorithme de Bellman-Ford depuis chaque noeud source, stockent ce résultat dans un noeud et le placent dans la *queue*. Des mutex sont utilisés pour que tous les noeuds sources soient traités une unique fois et pour éviter des altercations lors des ajouts de noeuds à la queue.

2) Consommateur: Le thread consommateur exécute une fonction `dequeue` qui retire un noeud de la *queue*, applique les fonctions `get_path` et `get_max` au contenu du noeud et écrit les résultats dans le fichier de sortie. Les mutex permettent d'éviter des altercations lors de la prise d'un élément dans la queue, de savoir quand tous les noeuds sources ont été traités et d'éviter que les résultats depuis les différents noeuds ne se superposent lors de l'écriture du fichier de sortie.



L'écriture du fichier de sortie devant se faire de manière séquentielle, nous avons choisi de n'attribuer qu'un seul thread à cette fonction. Nous aurions pu mettre plusieurs threads pour exécuter les fonctions `get_path` et `get_max` mais ces deux fonctions ont un temps d'exécution minime par rapport au reste du code et le gain en temps d'exécution ne se fait pas ressentir. Le second mutex évitant les altercations

lors de l'écriture du fichier de sortie est donc inutile mais nous l'avons laissé pour pouvoir facilement augmenter le nombre de threads sur cette partie si désiré.

III. Tests unitaires et outils d'analyse

A. Tests unitaires

Nous avons utilisé quatre graphes pour vérifier le bon fonctionnement de notre code : un graphe normal (sans particularité), un graphe non connexe, un graphe avec un cycle négatif et un graphe avec tous les noeuds qui partent d'un même point. Ce dernier graphe permet de tester que c'est l'indice le plus faible qui est pris si plusieurs noeuds sont à la même plus grande plus petite distance et de tester que c'est l'arête de plus faible coût qui est choisie si deux arêtes ont les mêmes sources et destinations. Ces quatre graphes couvrent ensemble toutes les situations possibles. Pour certaines fonctions, il n'est même pas nécessaire de vérifier le bon fonctionnement sur tous les graphes.
Les quatres graphes sont utiles pour la fonction `bellman_ford`. Pour la fonction `get_path`, les graphes normaux et non connexes vérifient le résultat dans un cas standard et dans un cas où aucun noeud n'est accessible. Pour la fonction `get_max`, le graphe normal et celui dont toutes les arêtes partent du même noeud sont suffisants pour tester un cas normal, un cas où un noeud n'est pas accessible et un cas où plusieurs noeuds sont à la même plus grande distance. Pour les fonctions du module `file`, nous vérifions sur un graphe que la lecture et l'écriture des fichiers se font correctement. Ces opérations étant indépendantes du graphe, un seul test est suffisant. Tester le fichier de sortie permet aussi de tester que l'exécution entière du programme fonctionne.

B. Valgrind

L'outil Valgrind nous a permis de vérifier qu'il n'y a aucune fuite de mémoire lorsque nous lançons l'algorithme sur chacun des graphes, que le résultat soit affiché à l'écran ou enregistré dans un fichier de sortie. Nous avons également pu vérifier qu'en cas d'erreur lors d'un appel de fonction (malloc, fread...), il n'y a pas non plus de perte de mémoire.

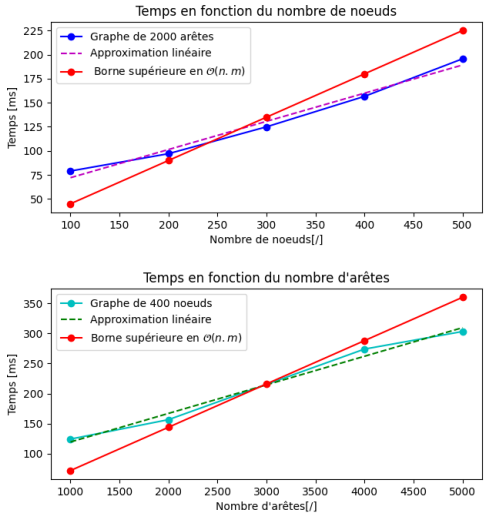
C. Cppcheck

Cppcheck pour nous a aidé à améliorer le code pour des variables qui seraient utilisées sans être toujours initialisées et pour réduire au maximum la portée des variables.

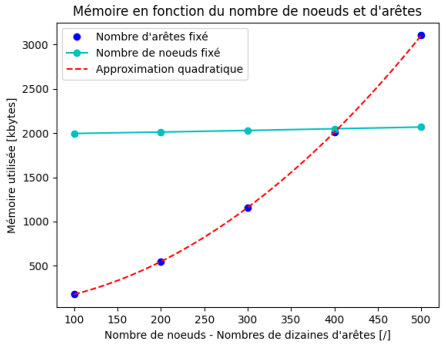
IV. Analyse des performances

A. Code C par rapport au code Python

1) Temps d'exécution : Pour un graphe de 400 noeuds et 2000 arêtes, les temps d'exécution sont respectivement de $57s$ pour le code Python et de $80ms$ pour le code C. Dans le cas d'un graphe de même dimension mais contenant un cycle négatif, les temps d'exécution sont de $66.32s$ et de $77ms$. L'exécution du code en C est donc nettement plus rapide.Ceci peut s'expliquer par le multithreading, par l'amélioration faite au code décrite dans la section amélioration du code et par la nature du langage C.
Notons que l'algorithme est en $\mathcal{O}(n.m)$ avec n le nombre de noeuds et m le nombre d'arêtes (dû aux itérations de l'algorithme de Bellman-Ford)



2) Mémoire consommée : Pour un graphe de 400 noeuds et 2000 arêtes, le code en C alloue $2020.59kB$ tandis que le code en Python utilise $343.96kB$. Le code C alloue donc beaucoup plus de mémoire. Cependant, comme expliqué à la section amélioration du code, nous avons décidé d'améliorer la mémoire instantanée du code. Nous aurions pu réduire la mémoire totale allouée en n'allouant par exemple qu'une fois de la mémoire à la structure `result_get_path_t` au lieu de l'allouer à chaque itération mais nous avons préféré cette seconde option pour atteindre notre objectif.

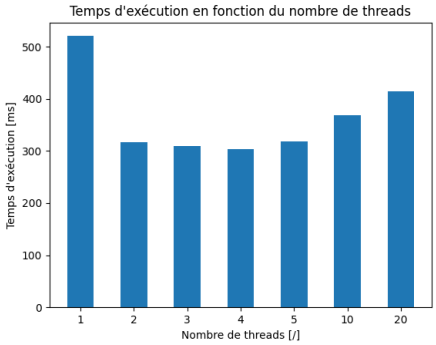


Notons que la mémoire évolue en $O(n^2)$ avec n le nombre de noeuds. Cette évolution quadratique est due au fait que l'algorithme de Bellman-Ford stocke ses résultats dans deux tableaux de taille n, et que cet algorithme est exécuté pour chaque noeud source (n fois). Les arêtes ne sont stockées que dans le graphe et font donc évoluer linéairement la mémoire.

3) Énergie consommée : Pour un graphe de 400 noeuds et 2000 arêtes sans cycle négatif, les puissances sont respectivement de $3.4W$ et $2.6W$ pour les codes en Python et en C.
Pour un graphe de même dimension mais avec un cycle négatif, les puissances sont de $3.4W$ et $2.65W$. L'exécution de l'algorithme en Python utilise donc environ 24.5% de puissance en plus.
L'exécution du code Python a duré 57 secondes et a donc consommé $57 * 3.4 = 193.8J$. L'exécution du code C n'a duré que 0.08 secondes et n'a dès lors consommé que $0.08 * 2.65 = 0.212J$. Le code C consomme donc beaucoup moins d'énergie car son temps d'exécution est plus court.

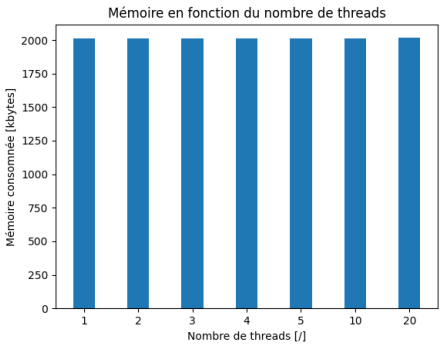
B. Évolution en fonction du nombre de threads

1) Temps d'exécution :



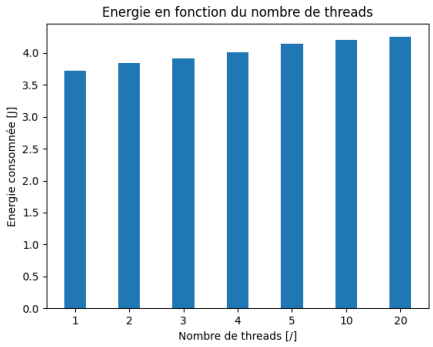
Passer de un à deux threads pour exécuter l'algorithme de Bellman-Ford rend l'exécution environ 1.72 fois plus rapide. Cependant, rajouter un trop grand nombre de threads diminue la vitesse d'exécution. En effet, le *Raspberry* n'a que 4 coeurs. Au-dessus de 4 threads, ceux-ci doivent se partager les coeurs à l'aide d'un scheduler non déterministe. Il y a donc des threads au repos qui attendent leur tour. Le temps alloué à la création d'un thread n'est pas "rentabilisé".

2) Mémoire consommée :



Comme visible sur le graphe ci-dessus, la mémoire totale allouée lors de l'exécution varie très peu avec le nombre de threads. En effet, si on ajoute un thread, seule la mémoire allouée à la création de ce thread vient s'ajouter à la mémoire totale. Cependant, nous remarquerons dans la section amélioration du code que la mémoire instantanée varie en fonction du nombre de threads.

3) Énergie consommée :



L'énergie consommée croît peu avec le nombre de threads. En effet, le temps d'exécution décroît mais la puissance augmente, ce qui stabilise le produit. Cependant, similairement au temps d'exécution, rajouter un trop grand nombre de threads engendre une surconsommation. La puissance augmente mais le temps d'exécution ne diminue plus.

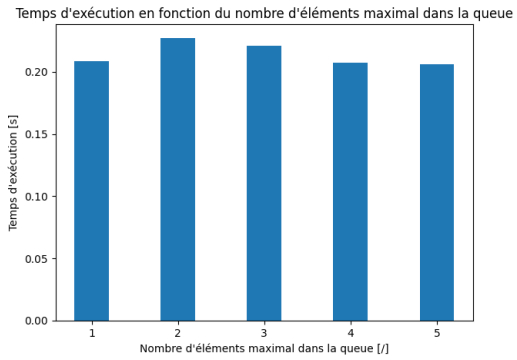
V. Amélioration du code

A. Mémoire instantanée allouée

Dans un premier temps, nous avons optimisé la mémoire instantanée allouée. Cela signifie que nous avons rendu la mémoire allouée par le code la plus faible possible à tout instant de l'exécution. Pour se faire, nous avons utilisé le plus de pointeurs possibles pour éviter de copier des données. En outre, nous n'allouons de la mémoire que lorsqu'elle devient nécessaire et la libérons dès qu'elle ne l'est plus. Cela ralentit légèrement le code par les nombreux appels systèmes mais c'était le prix à payer pour avoir la mémoire instantanée la plus optimale.
L'utilisation d'une queue entre le producteur et le consommateur permet d'avoir maximum $n + 1 + 1$ structures `result_bellman_ford_t` allouées en même temps (avec n le nombres de threads qui exécutent l'algorithme de bellman_ford). Les deux 1 correspondent à la structure `result_bellman_ford_t` stockée dans l'unique noeud de la queue et à la structure `result_bellman_ford_t` sur laquelle le thread qui s'occupe de `get_max`, `get_path` et l'écriture du fichier de sortie, est occupée.
Pour chaque structure `result_bellman_ford_t`, il peut aussi y avoir de la mémoire allouée à un noeud car la mémoire d'un noeud est allouée avant de savoir si de la place est disponible dans la queue et est libérée après avoir signalé qu'une place était disponible dans la queue. Un autre noeud peut donc être entré dans la queue avant que la mémoire du noeud précédent ne soit libérée.
À un instant donné, il est possible qu'il y ait également de la mémoire allouée à maximum une structure `result_get_max_t` et une structure

`result_get_path_t` car il n'y a qu'un seul thread qui s'occupe de cette partie du code.
De plus, de la mémoire est également allouée au graphe et aux threads, qui sont utilisés dans la majeure partie du code. Enfin, de la mémoire est parfois allouée durant un cours instant pour la lecture et l'écriture de fichiers.
La mémoire instantanée allouée évolue donc linéairement en fonction du nombre de threads.

Nous avons décidé de limiter la taille de la queue à un élément mais nous avons testé ce qu'il se passerait si nous autorisions plus de noeuds dans la queue. Comme visible sur la figure ci-dessous, le temps d'exécution est approximativement le même quelque soit la limite de taille de la queue. Cela signifie que, même si nous autorisons la queue à avoir une taille infinie, elle reste proche de un noeud. Il est possible que deux noeuds viennent dans la queue mais c'est rare. La quantité de mémoire décrite ci-dessus est bien une borne supérieure sur la mémoire instantanée et il est possible que cette borne ne soit pas toujours atteinte. La limite à un noeud dans la queue n'est donc pas une grande restriction mais permet d'avoir une idée de la mémoire allouée instantanément.

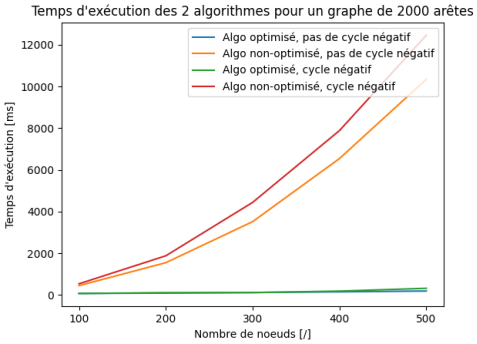


B. Temps d'exécution

Par la suite, nous avons décidé d'améliorer également le temps d'exécution de notre algorithme. Nous avons apporté deux modifications à l'algorithme de Bellman-Ford :

- Pour les graphes où il n'y a pas de cycle négatif, nous arrêtons les itérations de `bellman_ford` si, lors d'une itération, le coût minimal à chaque noeud n'a pas changé entre le début et la fin de l'itération.
- Pour les graphes avec cycle négatif, nous arrêtons les itérations de `bellman_ford` si la distance au noeud source devient négative à un moment. En effet, si elle est négative, c'est qu'il y a un cycle négatif.

La complexité de l'algorithme reste en O(n.m) mais ces deux modifications permettent de réduire le temps d'exécution dans de nombreux problèmes plus "simples". La complexité donne bien une borne supérieure sur le pire des cas.
La figure ci-dessous compare le temps d'exécution entre les algorithmes optimisés et non optimisés, dans le cas d'un graphe sans et avec cycle négatif.



VI. Conclusion

Ce projet nous a appris la programmation en C, en single et multi-thread, mais également l'utilisation d'outils comme Git ou encore la prise en main d'un *Raspberry Pi*.
Les résultats de nos mesures sont convenablement soutenus par la théorie, particulièrement en terme de mémoire consommée et de temps d'exécution. Notre implémentation de l'algorithme de Bellman-Ford en C est nettement plus rapide que celle en Python et nous avons optimisé la mémoire instantanée allouée.