Compte Rendu Algo TP0-5

Julien D’aboville - 22107316

2023

SOMMAIRE :

TP0 : Préliminaire

TP11

Explication du TP :

Dans ce TP, j'ai travaillé sur la création de structures de données pour représenter une forme polygonale à l'aide d'une liste doublement chaînée. J'ai utilisé deux structures principales, **point** pour stocker les coordonnées d'un point, et **cellule** pour construire la liste qui représente la forme polygonale.

Ressenti en Difficulté :

J'ai rencontré des difficultés au début dans la compréhension de la gestion des listes doublement chaînées avec les pointeurs et les allocations dynamiques de mémoire. C'est-à-dire comprendre comment chaque cellule était liée aux autres. La manipulation des pointeurs, en particulier lors de l'insertion et de la suppression, a été également compliquée. J'ai réussi à bien comprendre en représentant les listes chaînées avec leurs adresses mémoires sur une feuille, ensuite l'implémentation des méthodes a été beaucoup plus simple.

Conclusion :

Les codes compilent, s'exécutent et retournent des résultats corrects

TP12

Explication du TP :

Dans cette partie du TP, j'ai dû représenter un arbre binaire à l'aide de structures de données chaînées. La structure principale que j'ai créé est Noeud, qui stocke un caractère, un numéro de création automatiquement incrémenté, ainsi que des pointeurs vers les fils gauche et droit.

Ressenti en Difficulté :

J'ai trouvé le TP plus facile que le TP11 car j'ai commencé à mieux comprendre les pointeurs et allocations de mémoire. Les fonctions et structures à implémenter ne m'ont pas posé de problème.

Conclusion :

Les codes compilent, s'exécutent et retournent des résultats corrects

Questions :

Comment étendre la structure de données choisies pour un arbre générique ? Un graphe quelconque ?

Pour étendre la structure de données pour un graphe quelconque, on peut utiliser un tableau de pointeurs.

Structure en C :

```
typedef struct sommet{
        int value;
        struct sommet**voisins;
}sommet
```

Pour les arbres génériques cela va être relativement la même chose. Mis à part qu'il faudra éviter de faire des cycliques et il faudra conserver sa connexité car un arbre est connexe et acyclique.

TP1 Matrice d’adjacence

Explication du TP :

Dans cette partie du TP, j’ai créé un graphe à l'aide d’une matrice d’adjacence ainsi qu’une fonction pour marquer les voisins a partir d’un sommet, pour certains graphe elle ne marchait pas. J’ai donc créé une nouvelle version pour résoudre cela.

Ressenti en Difficulté :

Je n’ai pas trouvé l’algo très difficile à implémenter pour la fonction qui affiche pas tous les sommets. Pour celle qui affiche tous les sommets j’ai eu un peu plus de mal à l’implémenter.

Pour la transposer vers une version linéarisé je n’ai pas eu de problème.

Conclusion :

Les codes compilent, s’exécutent et retournent des résultats corrects

Questions :

Est-ce que cet algorithme marque tous les sommets ?

Non, il ne marque pas tous les sommets.’.

Quelle est la complexité calcul de cet algorithme ?

La complexité calcul est de O(n^2) car nous utilisons une matrice adjacente d’un graphe à n sommets de taille n*n.

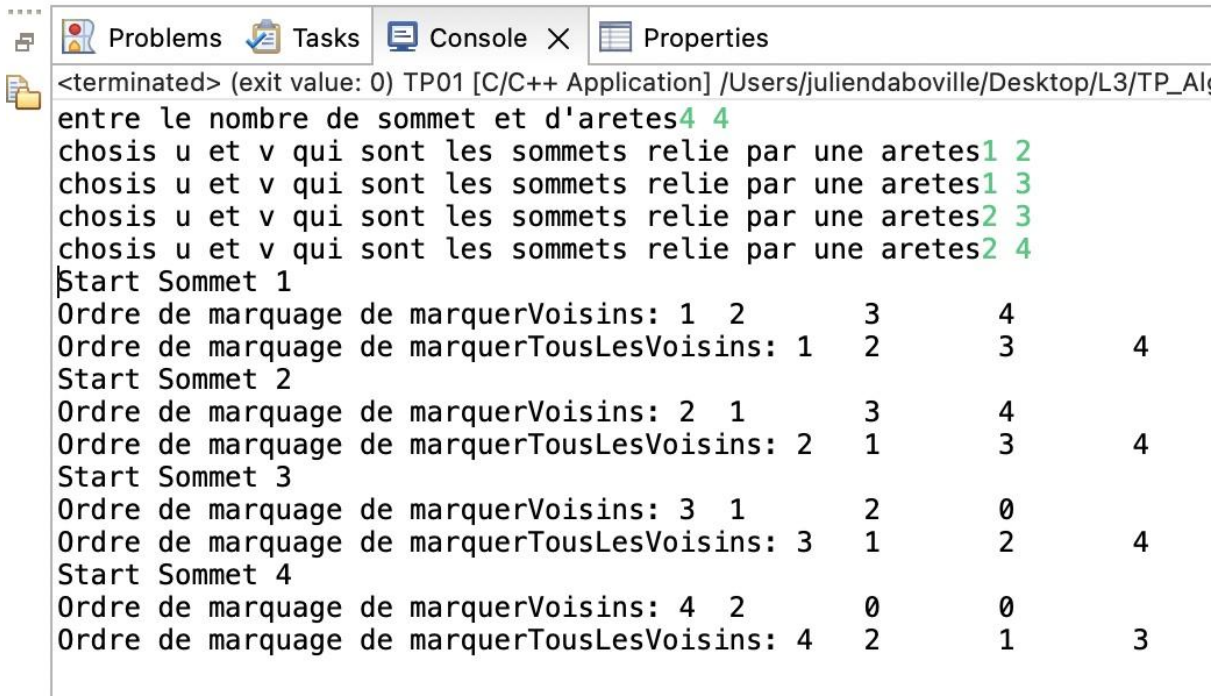
Quelle est la complexité mémoire ?

Elle sera également de O(n^2).

Peut-on faire mieux ?

Oui, si on utilise une liste d’adjacence par exemple la complexité serait de O(n).

Affichage de la fonction marquerVoisins et marquerTousLesVoisins



TP2 Connexion et chemins

Explication du TP :

Dans cette partie du TP, l'objectif principal était d'implémenter un algorithme pour trouver le plus court chemin dans un graphe non pondéré à partir d'un sommet de référence. Pour ce faire, j'ai utilisé une approche similaire à la recherche en largeur d'abord (BFS) en utilisant une file d'attente avec la structure cellule du tp11.

Ressenti en Difficulté :

Pour le PCC j'ai trouvé l'algo facile à comprendre et assez simple à implémenter. Je me suis aidé en utilisant la structure Cellule du TP11. Je me suis donc permis de faire l'activité supplémentaire.

Celle-ci m'a fait rencontrer des difficultés sur l'utilisation des fichiers et sur le fait de relier le code python avec les données du fichier csv contenant les complexité temporelle de le PCC. Je n'avais encore jamais fait ça mais j'ai tout de même réussi à le faire.

Conclusion :

Les codes compilent, s'exécutent et retournent des résultats corrects

Questions (Activités supplémentaire) :

1) Analyse de sa complexité mémoire(spatiale) et calcul(temporelle) :

La complexité de l'algo BFS est de l'ordre de O(E + V) dans le pire des cas.

Pour la complexité mémoire dû à l'utilisation d'une matrice d'adjacence pour le graphe, la complexité sera alors dominé par celle-ci donc de l'ordre de O(n^2).

Pour la complexité du calcul (temporelle) ?

La complexité serait de l'ordre de O(n^2) car dans le pire des cas, pour chaque sommet on doit vérifier tous les autres sommets pour vérifier s'ils sont adjacents ou pas.

2) Obtient-on un arbre couvrant avec l'algorithme précédent ? Pourquoi ?

Non ,l'algo ne construit pas explicitement un arbre couvrant, il aurait fallu qu'avec les informations stockés dans le tableau **pred** construire une structure de donnée représentant l'arbre couvrant.

3) Quel est le diamètre d'un graphe ? d'un arbre ?

Le diamètre d'un graphe est défini comme la plus grande longueur parmi les plus courts chemins possibles entre toutes les paires de sommets dans le graphe.

Le diamètre d'un arbre est la distance la plus grande entre deux nœuds de l'arbre

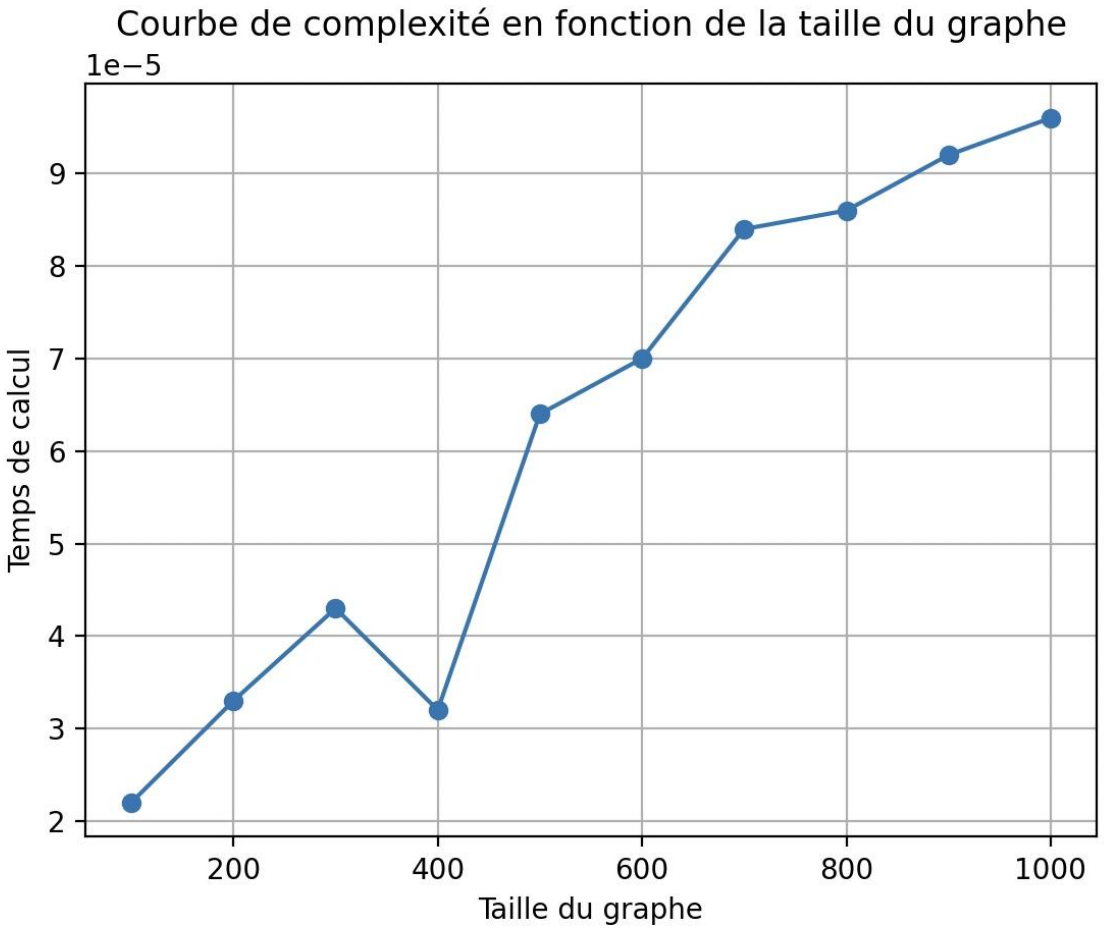
4) Obtient-on le diamètre de ce graphe avec l'algorithme précédent ?

Non, car il aurait fallu ajouter une variable supplémentaire diamètre pour suivre le maximum des longueurs minimum lm tout le long de l'algo.

5) Que faudrait-il faire pour l'obtenir si on imagine le BFS(G) comme un flot de propagation d'arêtes marquées partant d'un point source s ?

On peut commencer à partir d'un point central que l'on va appeler, s, comme le point de départ.
Chaque arête est marquée avec la distance minimale entre le point central s et le sommet actuel.
Pendant le parcours, on garde une trace de la distance maximale atteinte jusqu'à présent (diamètre)
À la fin, on parcourt le tableau **lm** pour trouver les deux points du graphe qui contribuent à cette distance maximale. Ces deux points sont les extrémités du "diamètre".

6) Courbe de complexité avec en abscisse la taille du problème et en ordonnée le temps de calcul obtenue avec mon code en python :


Courbe de complexité en fonction de la taille du graphe
1e−5
9
8
7
6
5
4
3
2
Temps de calcul
200
400
600
800
1000
Taille du graphe

TP 3 : Marche et Graphe Biparti

Explication du TP :

Le but de ce TP est de représenter les différents parcours possibles pour un arbre binaire (parcours préfixe, parcours infixe, parcours suffixe).

Ressenti en Difficulté :

Ce TP a été simple à implémenter car j'avais déjà essayé de le faire pour le Tp12.

Conclusion :

Les codes compilent, s'exécutent et les résultats sont corrects.

Questions :

Quel est le nombre chromatique d'un graphe tripartite ?

Le nombre chromatique d'un graphe tripartite est 3 car si il est tripartite, il sera alors partitionné en trois ensembles de sommets de telle sorte qu'aucune arête ne relie deux sommets dans le même ensemble.

TP 4 : Arbre Couvrant

Prim

Explication du TP :

Dans ce TP j'ai implémenté l'algorithme de Prim en langage C. L'algorithme de Prim est utilisé pour trouver l'arbre couvrant de poids minimum dans un graphe valué et non orienté. Le résultat de l'arbre couvrant s'est fait à l'aide d'une liste d'arête.

Ressenti en Difficulté :

La compréhension de l'algo de Prim ainsi que la création de la structure **t_arete** étaient assez simple. Enfin, en suivant le canevas je n'ai pas eu de soucis à l'implémenter.

Conclusion :

Les codes compilent, s'exécutent et retournent des résultats corrects

Kruskal (Activités supplémentaires)

Explication du TP :

Dans cette partie du TP j'ai implémenté l'algorithme de Kruskal en langage C. L'objectif de cet algorithme est de trouver l'arbre couvrant de poids minimum dans un graphe valué et non orienté. L'arbre couvrant ainsi obtenu est représenté sous forme d'un tableau d'arêtes.

L'algorithme de Kruskal fonctionne en triant les arêtes du graphe en ordre croissant de poids, puis en ajoutant progressivement les arêtes les plus légères à l'arbre couvrant, en évitant de former des cycles. L'ensemble des connexions entre les sommets est géré à l'aide d'un tableau **connexe.**

Ressenti en Difficulté :

Avec l'aide du canevas et des connaissances acquises lors de la mise en œuvre de l'algorithme de Prim précédemment, je n'ai pas rencontré de grandes difficultés. La seule tâche un peu complexe était de trier le graphe en fonction des poids de ses n arêtes dans l'ordre croissant.

Conclusion :

Les codes compilent, s'exécutent et retournent des résultats corrects

TP 5 : Arbre Couvrant

Partie B1

Explication du TP :

Le but du TP était de modifier le code de ChatGPT afin de faire fonctionner l'algorithme de Welsh-Powell.

Ressenti en Difficulté :

En ce qui concerne le code en C, je n'ai pas rencontré de grandes difficultés. J'ai commencé par analyser le code étape par étape en utilisant des instructions "printf" pour m'aider. Par la suite, j'ai compris que le tableau "color" n'était pas correctement initialisé avec des valeurs de -1. Une fois ce problème résolu, l'algorithme de Welsh-Powell ne donnait toujours pas le bon nombre chromatique que j'avais trouvé en effectuant les calculs sur mon cahier. J'ai donc décidé de recommencer le code en partant de zéro à partir du tableau "color", et j'ai ensuite trouvé le bon nombre chromatique.

En ce qui concerne le code Python, j'ai tenté de résoudre cette partie avec l'aide de ChatGPT. J'ai effectué plusieurs essais pour concevoir un algorithme qui calcule la complexité temporelle des graphes de taille N=10, 100 et 1000, puis qui les affiche.

Conclusion :

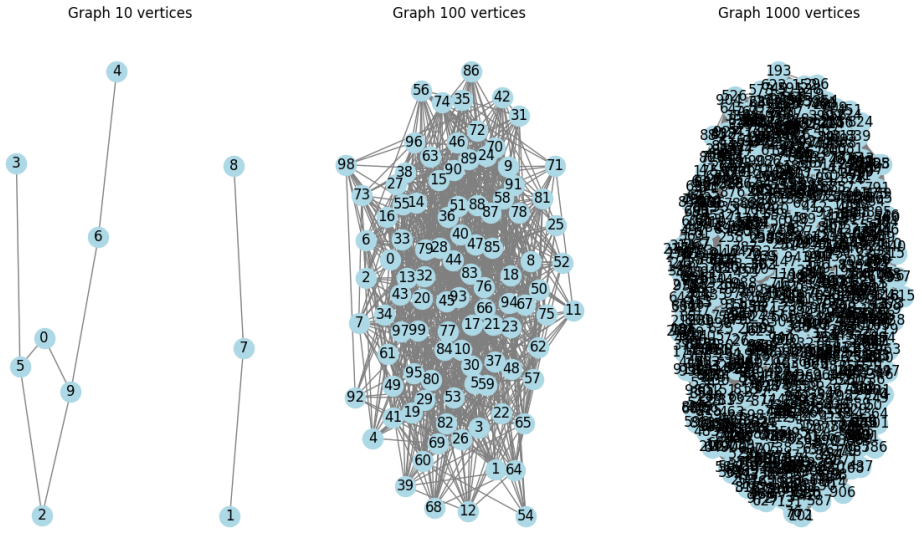
Les codes compilent, s’exécutent et retournent des résultats corrects.

Voici les temps que j’ai trouvé pour la complexité de la coloration d’un graphe:

Pour N =10 : temp d'exécution moyen=0.02s
Pour N =100 : temp d'exécution moyen=0.2s
Pour N =1000 : temp d'exécution moyen=2s

On peut voir que le temps d'exécution est environ 10 fois plus long lorsque N est multiplié par 10. Donc la complexité temporelle est linéaire donc polynomiale en N.

Voici des graphes générés en python avec des arêtes aléatoires :



Partie B2

Ensembles (Sets) :

- java.util.HashSet: Une implémentation de base d'un ensemble non ordonné.
- java.util.LinkedHashSet: Un ensemble ordonné qui conserve l'ordre d'insertion.
- java.util.TreeSet: Un ensemble ordonné qui utilise un arbre binaire pour le stockage.

Dictionnaires (Maps) :

- java.util.HashMap: Une implémentation de base d'un dictionnaire non ordonné.
- java.util.LinkedHashMap: Un dictionnaire ordonné qui conserve l'ordre d'insertion.
- java.util.TreeMap: Un dictionnaire ordonné qui utilise un arbre binaire pour le stockage.

Files d'attente (Queues) :

- java.util.LinkedList: Peut être utilisée comme une file d'attente (FIFO) en ajoutant des éléments à la fin et en retirant des éléments du début.
- java.util.PriorityQueue: Utilisée pour implémenter une file d'attente de priorité.

Piles (Stacks) :

- java.util.LinkedList pour simuler une pile (LIFO) en ajoutant et retirant des éléments au début.

En ce qui concerne les structures de recherche, quels tas (heap) sont disponibles nativement en Java ?

Tas (Heap) :

- Java dispose d'une classe appelée java.util.PriorityQueue qui peut être utilisée pour implémenter un tas binaire, qui est souvent utilisé pour la gestion des priorités dans les algorithmes de graphes

Partie C2

Explication du TP :

Dans cette partie de TP j’ai créé programme en Java pour générer tous les arbres binaires de taille n et afficher les résultats en utilisant des instructions "printf()"..

Ressenti en Difficulté :

Lors de la mise en œuvre du code en Java, la création de la structure de nœud n'a pas posé de problème majeur.

Cependant, j'ai rencontré des difficultés lors de la mise en place des boucles récursives pour générer les arbres binaires.

Conclusion :

Les codes compilent, s’exécutent et retournent des résultats corrects