Compte rendu Parties B en langage C

Fait par Remy XU, numéro etudiant: 22123345.

Preambule:

L'ensemble des TP de 0 à 5 ont été faits, sauf B2 la partie en Java du TP5. Chaque Tp a son propre dossier dans lequel il y a: un fichier .c avec le nom du programme avec son .h associé, et un fichier monNomFichier pour chaque .c qui contient le main des fichiers précédents. Il y a également un README.txt comportant quelques explications supplémentaires pour certains TDs, sinon les commentaires dans les codes peuvent aider à la compréhension des codes.

La plupart des programmes prennent en compte les exceptions (par exemple, supprimer un nœud lorsqu'il y en a pas dans un arbre, une liste, un graphe).

Pour le TP0 et TP3 et TP5, il y a des interfaces utilisateurs.

Note: il y a pour les programmes utilisant les arbres, un define PROFONDEUR. Les programmes ne l'utilisent pas et fonctionnent pour toute profondeur d'arbre. Les autres define sont tous utiles, notamment celui des graphes qu'il faut modifier si besoin (VOISINSMAX et SOMMETS MAX).

TP0:

Le dossier TP0 comporte les programmes pour:

- **Les arbres binaires** d'ints,
- **Les listes doublement chaînées** d'ints,
- **Les graphes** de char,
- Et une tentative de pseudo code pour les arbres quelconques de tri (n fils pour chaque nœud).

Les fonctionnalités de ces programmes sont: la création de la structure, l'ajout d'un élément dans la structure, la suppression des éléments, l'affichage de ces structures.

Il y a également une **interface utilisateur** pour ces 3 programmes, avec des lettres pour effectuer les actions (voir les mains).

(Note: Les programmes du TP0 ne suivent pas exactement la construction des TP en L2 car il y a des améliorations)

A) Liste doublement chaînées

On a deux structures, celle du nœud et celle de DListe qui pointe le premier nœud de la dliste et le dernier. Cette deuxième structure permet d'accéder également la fin de la liste ce qui accélère le processus d'ajout de nœud si on doit l'ajouter en fin de liste.
Ainsi pour l'insertion et pour la suppression de nœud, on parcourt la liste soit par la fin soit par le début en fonction de si l'indice se situe dans la première moitié ou de la deuxième pour un gain de calcul.
On peut également ajouter directement le nœud en début de liste ou en fin de liste

Sinon le fonctionnement est identique a une double liste classique.

B) Arbres binaires

Pour l'arbre binaire, nous pouvons précisément: initialiser l'arbre, remplir l'arbre, l'afficher, effacer un nœud.
Il ne s'agit pas d'un arbre binaire de recherche mais d'un arbre ou l'on peut stocker une valeur quelconque à l'ordre que l'on souhaite par parcours en BFS.
Par exemple, si on veut stocker l'arbre en ordre 0, il s'agit de la racine, 1 le fils gauche de la racine, 2 le fils droit de la racine et 3 le fils gauche de la racine puis son fils gauche à lui et ainsi de suite.

Remplir l'arbre binaire:
Donc pour remplir l'arbre j'ai utilisé plusieurs façons:
La première méthode que j'ai utilisé, est remplirArbre(). Elle remplit l'arbre en demandant à l'utilisateur de donner une chaîne de caractères de 0 et de 1.
Le str est l'arborescence de la ou on veut poser le nœud. On note 0 pour le fils gauche et 1 pour le fils droit.
Par exemple, en partant de la racine, on veut poser un nœud (une valeur) dans cet ordre (filsgauche->filsdroit->filsgauche). On note alors dans ce str "010".

Cette fonction n'est pas utilisée, on utilise une version améliorée: remplirArbreParOrdre().
Au lieu de demander à l'utilisateur l'arborescence en pseudo-binaire, on a une fonction intEnArbo() qui convertit l'ordre ou on veut placer l'entier, en arborescence de binaire.
Par exemple, on peut écrire :
`"remplirArbreParOrdre(0,intEnArbo(0),arbreE;".`

La fonction intEnArbo() a une explication détaillée dans le README.txt du TP0.

Dans le programme, on ne peut pas donner a une feuille un fils si le fils n'existe pas. Cela écrira un message. Normalement il est possible de remplacer un nœud existant, mais cela n'altère pas le programme (la taille de l'arbre sera cependant erronée).

Retirer un Noeud:
Lorsque l'on retire un nœud, on retire également ses fils. Pour calculer le nombre de noeuds qui vont être effacées, on utilise "`calculerNoeuds(noeud* node)`" ce qui permet de réduire la taille de l'arbre de ce nombre .

On note aussi que la valeur de la racine est initialisée à [-1000], pour pouvoir bien observer la suppression de la racine.

C) Graphes simples non orientés (ou orientés)

On reprend l'idée de la liste doublement chaînée mais au lieu de (noeud) pred et (noeud) suiv, on a un tableau de pointeur de nœuds (voisins). On peut alors créer une arête de A à B par exemple.
Le graphe est sous forme de tableau de sommets, et chaque sommet a une liste de voisins. Il s'agit d'une représentation en liste d'adjacence, et on choisit de l'afficher de cette manière.

On peut ajouter des sommets avec un nom en char, et créer des arêtes entre eux. On note que l'on peut aussi faire des boucles de sommet.

Transformation en non orienté:
Pour le transformer en non-orienté, on doit de mettre en commentaire la ligne 162 de graphe.c dans la méthode lierDeuxSommets (si le programme n'est pas modifié): "node2->voisins[node2->nbVoisins]=node1;".
Cela permet de construire des graphes orientés. La suppression de sommet pourrait être alors optimisée en une version moins coûteuse (car en non orienté il faut vérifier que tous les voisins d'une du sommet S que l'on va supprimer n'a pas S comme voisin. On peut passer d'une double boucle en une seule boucle).

On note que le retrait de la ligne 162 suffit amplement.

TP1:

L'algorithme modifié est dans marquerVoisins et chargeGraphe dans graphe2.c.

Le programme graphe2.c est similaire au TP0, mais il est moins complet que celui du TP0 (n'a pas la suppression des sommets et arêtes.) monGraphe2.c permet de transformer une matrice écrite par l'utilisateur en scanf, en graphe (liste d'adjacence).
Le programme MatriceAdjacence permet d'afficher la matrice a partir de saisie (scanf) de l'utilisateur.

QUESTIONS:
1) Cet algorithme marque tous les voisins du sommet s, puis leurs voisins à eux et ainsi de suite. Il s'agit d'un parcours en BFS pour afficher une composante connexe du graphe en partant du sommet s.
Cependant, il n'a pas l'air d'afficher tous les sommets du graphe.

Exemple:
[A[D],B[C],C[B,D],D[A,C]]
En faisant marquerVoisins(matrice, 4, 1), lors de la création de la liste marques on marque A, puis D lorsque x=0, puis aucun, puis aucun, puis c lorsque x=3. Il manque le sommet B alors que la composante est bien connexe.

(De plus, il ne prend pas en compte les arêtes allant vers lui même.)

J'ai donc modifié l'algorithme en deux méthodes:

- L'algorithme naïf en commentaire:

Il essaie de marquer tous les voisins de toutes les arêtes marques. Il fait ca plusieurs fois jusqu'à une itération dont le nombre de marquages est nul.

- L'algorithme utilisant le parcours en BFS.

Complexité calcul:
L'algorithme proposé par le tp a une complexité au pire de O(n^2).
L'algorithme naïf en commentaire, corrigé, a une complexité de O(m*n^2).
L'algorithme utilisant le parcours en BFS a une complexité de calcul en O(nm).

<u>Opérations remarquables:</u>
Lors du while: m car on parcourt au pire m éléments de la file
Lors du for: n car on parcourt n lignes de la matrice sur la colonne du sommet concernée.
Dans les boucles, on a des affectations et des opérations sur les structures de données.

Nous avons réussi à améliorer la complexité calcul de l'algorithme.

Complexité mémoire:
L'algorithme proposé par le tp a une complexité au pire de O(n^2).
L'algorithme naïf en commentaire, corrigé, a une complexité de O(n^2).

L'algorithme qui est très proche du BFS du cours a une complexité de mémoire en O(n^2), car il stocke une matrice d'adjacence en entrée et plusieurs listes lors de l'algorithme. (Comme la complexité de la matrice est dominante.)

Nous n'avons pas réussi à améliorer la complexité mémoire, mais nous pouvons l'améliorer en O(m) si nous décidons d'utiliser une liste d'adjacence au lieu de la matrice d'adjacence.

Concernant le fonctionnement de mon programme de la version linéarisée de la matrice d'adjacence, il s'agit exactement du même programme. Il suffit juste de changer adjacence[i][j] en adjacence[ordre*i+j] et changer la définition de int** adjacence en int*adjacence et si besoin en allouant ordre*ordre cases mémoires.

TP2:

La file est reprise de la liste doublement chaînée dans le TP0.
On peut l'initialiser, enfiler et défiler. Il n'y a pas de test de la fonction plusCourtChemin ici même si elle est corrigée.

Analyse du code:

On remarque que cet algorithme est très similaire au BFS: On a une file avec l'élément de départ et on développe ses voisins que l'on met dans la file et ainsi de suite. On ajoute également la longueur qu'on a parcouru pour atteindre l'arête concernée (longueur qui est ici de 1 pour un graphe non valué) dans un tableau.
On note également de prédécesseur de l'arête développée dans un autre tableau.

Il s'agit du code améliorée du TP1 marquerVoisins(). Ce programme marque également les voisins visité comme le TP1.

Complexité mémoire:
L'algorithme qui est très proche du BFS du cours a une complexité de mémoire en O(n^2), car il stocke une matrice d'adjacence en entrée et plusieurs listes lors de l'algorithme.

(Comme la complexité de la matrice est dominante.)

Complexité calcul:
L'algorithme utilisant le parcours en BFS a une complexité de calcul en O(nm).
Opérations remarquables:
Lors du while: m car on parcourt au pire m éléments de la file
Lors du for: n car on parcourt n lignes de la matrice sur la colonne du sommet concernée.
Dans les boucles, on a des affectations et des opérations sur les structures de données.

TP3:

Le programme arbre TP3 est identique à celui du TP0, il a seulement une option supplémentaire: Il faut choisir le mode d'affichage avant de faire les opérations sur l'arbre.

A pour préfixe, B pour infixe, et C pour postfixe.

L'affichage des arbres en l'un de ces trois parcours fonctionne par récurrence.

TP4:

Ce programme permet d'utiliser l'algorithme de Prim donné par l'énoncé. Il a été complété avec la fonction RemplirDListe().

Arbre sous forme de double liste:
L'arbre est renvoyé sous forme de liste d'arêtes avec les deux sommets et le poids de l'arête, avec la fonction remplirDListe(xmin,ymin,min, arbre).

Matrice d'adjacence avec poids:
Le programme récupère en scanf les valeurs des coordonnées de la matrice. Mais il est possible dans le main de decommenter un exemple avec ordre = 4.

TP5:

Ici, le programme de Welsh Powell a été entièrement corrigé en C, avec une interface utilisateur.

Liste des erreurs et améliorations:

1) Il ne s'agit pas tout a fait de l'algorithme de Welsh Powell

Lorsque le sommet n a un de ses voisins coloriés, il change de couleur mais pour les arretes suivantes il ne revient pas sur la couleur précédente qu'il pourrait potentiellement colorier. Il colorie certes toutes les arêtes, mais il pourrait colorier le graphe avec moins de couleurs.

Par exemple: Le sommet 0 a une couleur 0, le sommet 1 est un voisin du premier sommet. Comme ils sont voisins, le sommet 1 aura une couleur attribuée de 1. Cependant d'autres sommets peuvent avoir éventuellement la couleur 0.

2) Erreur de définition de la structure:

On définit struct Graph sans lui donner de nom. Ensuite on écrit struct Graph* en paramètre ce qui essaie de créer une autre structure dans les paramètres ce qu'on ne veut pas et qui est en plus syntaxiquement incorrect. (warning: 'struct Graph' declared inside parameter list will not be visible outside of this definition or declaration)

3) Le tableau color est mal initialisé. Il faut mettre tous les éléments du graphe a -1

4) Pour des raisons de performances et d'efficacité, on préfère l'utilisation de pointeurs pour afficher les matrices

Construction de la matrice pour le graphe de taille N:
La matrice générée aléatoirement a bien une diagonale de 0 (graphe non orienté), et la matrice est bien symétrique. Les valeurs de la matrice sont des 0 ou 1 de façon aléatoire. La commande "`afficherAdjacence(graph.vertices, graph.adjMatrix);`" permet d'observer la matrice générée aléatoirement. Une matrice d'adjacence avec N=50 en exemple sera affichée après avoir quitté le programme.

Remarques des tests des programmes avec N sommets générés aléatoirement:
Le programme tourne rapidement pour des valeurs de N basses, mais elle commence à prendre environ 52 secondes sur ma machine avec N=10000 et utilise 880 couleurs pour un seul test que j'ai effectué. Vous pouvez tester et cela devrait afficher des résultats différents. Pour N=100 000, le programme ne se termine pas en raison de ressources mémoires insuffisantes.