Algorithmique II
Structure de données
-
TP noté : rapport

Sommaire

Exercice 1 échauffement

Pour tous les exercices, on nomme V un ensemble de la forme {1, …, n} et n sa taille.

Question 1 : complexité en espace

- Pour les matrices d'adjacence : Effectivement, l'espace mémoire réservé à un tableau est représenté par une suite contigüe de cellules contenant chacune un unique élément de même type. En sachant que la complexité spatiale de chaque élément est équivalente à la complexité spatiale du type de l'objet contenu dans le tableau ; ici 0(1), il faut désormais connaître le nombre de case que contiendra notre tableau à 2 dimensions. Dans le pire des cas, toutes les cases seront remplies et dès lors, la complexité spatiale de la matrice est de l'ordre de $O(n^2)$. Cependant il est important de noter que dans tous les cas, un tableau à deux dimensions est alloué par le programme dès son initialisation (dimensions identiques, égales à la taille de l'ensemble V considéré). Donc la complexité en espace de matrices d'adjacence est de $O(n^2) = |V|^2$ de toute façon.
- Pour les listes d'adjacence : Dans le cas des listes adjacentes, il est important de considérer le nombre de relation possible avec les autres éléments de la liste. Le pire des cas est atteint lorsque tous les éléments sont en relation avec tous les autres. On a alors une complexité similaire à celle de la matrice. Sachant que chaque élément de la liste est composé de deux valeurs (la donnée pour la relation, et l'adresse de l'élément suivant), la complexité spatiale est alors de $O(n^2)$.

Question 2

- Dans la représentation de la matrice, on a ajouté une ligne en haut et une colonne à gauche pour étiqueter plus facilement les données, et avoir un affichage plus facilement lisible pour le debug. Elles ne sont pas indispensables, mais impactent sensiblement les réponses aux questions qui suivent.
- La liste d'adjacence quant à elle est représentée par une liste d'élément ; chaque élément étant composé de deux variables ; un entier qui permet le stockage de la valeur de l'élément et un lien vers un nouvelle élément défini comme suit :

```
struct Element *suivant;
```

Question 3

Pour les matrices

- Adjacence : La complexité est de l'ordre O(1), il suffit de retourner la valeur de la case x, y, où x et y sont les éléments de V.
- Voisins : La complexité en temps de Voisins est O(n), car il faut forcément parcourir toute la ligne/colonne du tableau bidimensionnel pour tester chaque relation et ainsi retourner toutes les relations telle que $\{y \in V | (x,y) \in R\}$.

Pour les listes

- Adjacence : Au maximum, O(n) si tous les éléments de V sont en relation avec l'élément x considéré.
- Voisins : Si tous les éléments de V sont en relation avec l'élément x considéré, la complexité est également de l'ordre de O(n). En réalité, la complexité de cette primitive est égale au nombre d'éléments dans la liste retournée par la fonction. Donc la complexité est de O(n), que l'on peut écrire aussi O(|voisinsListe(x)|).

Exercice 2 utilisation

Question 1

- Une relation indépendante est toujours un co-graphe, car elle est représentée par un point qui n'est relié à aucun autre point. Par ailleurs, une relation complète est également toujours un co-graphe car c'est un point qui est relié à tous les autres points.
- Imaginons que l'on ait une relation R totalement quelconque. On prend sa clôture transitive et on la note RT. Maintenant, on suppose que R soit une relation arborescente, alors la relation RT serait complète. On aurait donc par exemple, pour deux a et b donnés. aRTb et bRTa. Cependant, comme RT est transitive (cf plus haut), on aurait donc aRTa et bRTb. Cela indique donc que la relation RT n'est pas complète car une relation complète ne peut pas être réflexive si on se fie à l'énoncé. Contradiction -> R n'est pas arborescente. Donc la relation RT n'est pas arborescente

Question 2

Plusieurs fonctions de test des relations ont été réalisées, que ce soit pour les matrices ou les listes d'adjacences. Les prototypes sont énumérés ci-dessous :

- `int estIndependanteM(int taille, int **matrice)`
- `int estIndependanteL(int taille, Element* liste[taille])`
- `int estCompleteM(int taille, int **matrice)`
- `int estCompleteL(int taille, Element* liste[taille])`
- `int estArborescenteM(int taille, int **matrice)`
- `int estArborescenteL(int taille, Element* liste[taille])`
- `int estP4M(int taille, int **matrice, const int tabElements[4])`
- `int estP4L(int taille, Element* liste[taille], const int tabElements[4])`
- `int estCoGrapheM(int taille, int **matrice)`
- `int estCoGrapheL(int taille, Element* liste[taille])`

Les fonctions dont le nom se termine par « M » correspondent aux matrices et celles qui se terminent par « L » aux listes.

Il est important de noter que toute ces fonctions renvoient des entiers. En effet en C, le type booléen n'est pas explicitement défini (avant norme C99) ; par conséquent il est judicieux de réaliser des `DEFINE` afin de simuler l'existence des valeurs booléenne. Cela permet une meilleure compréhension du code.

- `#define TRUE 1`
- `#define FALSE 0`

Question 3

Pour les matrices

- Complexité de la relation d'indépendance : $O(n^2)$ car il faut parcourir tout le tableau pour tester chaque possibilité. Effectivement le parcours se fait sur deux boucles imbriquées. Sachant qu'une boucle dans ce code est de complexité O(n) ; il faut donc multiplier la complexité des deux boucles soit $O(n) * O(n) = O(n^2)$.
- Complexité de la relation de complétude : Cette fois-ci, seuls les éléments diagonaux sont parcourus. Ils sont au pire des cas au nombre de n ce qui signifie que la complexité en temps est O(n).
- Complexité de la relation d'arborescence : $O(n^3)$ car on peut appeler en fin de fonction la fonction de la fermeture transitive de l'exo 1 qui tourne en $O(n^3)$. Si on n'appelle pas cette fonction, la complexité est donc en $O(n^2)$.
- Complexité de la relation co-graphe : $O(n^4)$, puisqu'il faut générer tous les quadruplets d'éléments.

Pour les listes

- Complexité de la relation d'indépendance : O(n) : Dans le pire des cas, on parcourt tous les éléments de l'ensemble pour vérifier qu'ils n'ont pas de suivant et tester si cela correspond ou non à une relation d'indépendance.
- Complexité de la relation de complétude : $O(n^2)$ si tous les éléments sont en relation avec tous, car il faut parcourir toutes les listes en profondeur, pour détecter si un élément est en relation avec lui-même.
- Complexité de la relation d'arborescence : Comme pour l'arborescence des matrices d'adjacence, on est en $O(n^3)$ car on peut appeler en fin de fonction la fonction de la fermeture transitive de l'exo 1 qui tourne en $O(n^3)$. Si on n'appelle pas cette fonction, la complexité est donc aussi en $O(n^2)$.
- Complexité de la relation co-graphe : $O(n^5)$ La fonction est plutôt compliquée à comprendre (elle l'était aussi à coder). Ce n'est pas aussi simple que pour les matrices, où la fonction, bien que compliquée aussi, est plus aisée à s'approprier.

Exercice 3 Création de CO

Pour cette question deux structures ont été réalisés.

Structure Nœud

Une structure Nœud qui correspond à un Nœud contenu dans une structure Arbre. La structure Nœud contient un caractère nommé étiquette et qui peut avoir comme valeur '1','+' ou '*'. Cette structure Nœud possède également un numéro qui permet de savoir si ce nœud est une feuille (différent de 0). En outre la structure Nœud possède également un fils gauche (et un fils droit) noté filsG (et filsD) qui sont eux même des Nœud.

```
struct Noeud {

      char etiquette;
      int numero;
      struct Noeud *filsG ;
      struct Noeud *filsD;

};

typedef struct Noeud Noeud;
```

Précision : le `typedef` est juste un alias créé pour simplifier le code.

Structure Arbre

La structure Arbre quant à elle comporte un Nœud qui sera désigné comme racine ainsi qu'un nombre de feuille.

```
struct Arbre {

      Noeud *racine;
      int nbFeuilles;

};

typedef struct Arbre Arbre;
```

Fonctions

Plusieurs fonctions accompagnent ces structures :

- `Arbre* initArbre()` qui initialise un Arbre. On peut noter que l'allocation dynamique et l'usage de la fonction `creerRelation()` est nécessaire. Cette fonction retourne donc l'arbre formé.
- `Arbre* unionArbres(Arbre *arbre1, Arbre *arbre2, char operation)` qui réalise l'union de deux arbre précisés en paramètre et renvoie

l'arbre créé par cette union selon l'opération choisie en paramètre également ('+' ou '*').

- `void majArbre(Arbre *arbre)` qui met à jour le numéro des feuilles d'un arbre donné en paramètre.
- `void ArbreEnMatrice(int taille, Arbre *arbre, int **matrice)` qui transforme un Arbre en Matrice. La fonction prend en paramètre la taille de la matrice, un Arbre et une matrice. Il est important de noter que cette fonction utilise une fonction interne.
- `int* NoeudEnMatrice(Noeud *noeud, int **matrice)` qui s'occupe de la transition d'une nœud fourni en paramètre à une matrice (également passée en paramètre). Cette fonction prend en compte les différentes valeurs de l'étiquette de l'arbre et n'oublie pas de libérer la mémoire allouée par chaque nœud.
- `void ArbreEnListe(int taille, Arbre *arbre, Element *liste[taille])` qui transforme un arbre en liste en utilisant l'implémentation de la fonction précédente.
- `void creerRelation(Noeud *noeud)` qui crée une relation en initialisant chaque paramètre d'un nœud passé en paramètre.

Exercice 4 Manipulation

Question 1

La fonction `Noeud* NCA(Arbre *arbre, int x, int y)` prend en paramètre un arbre et deux numéros de feuille représentés par des entiers (ici x et y). Cette fonction retourne le plus petit ancêtre commun des deux nœuds x et y qui correspond à un nœud. L'implémentation de cette fonction consiste à vérifier si deux feuilles possèdent un ancêtre commun (cf. la fonction `int estAncetre(Noeud *test, int val)`) puis on renvoie le plus petit ancêtre commun des deux nœuds.

Question 2

- La fonction `int adjacentsA(Arbre *arbre, int x, int y)` prend en paramètre deux numéros de feuilles ainsi qu'un co-arbre et renvoie un entier qui vaut 1 si l'arbre est adjacent, 0 sinon. La fonction crée une matrice puis on transforme l'arbre en cette matrice ; ensuite on utilise la fonction `adjacentsM(taille, matrice, x, y)` qui teste si la matrice est adjacente et donc si le co-arbre est adjacent.
- La fonction `void voisinsA(Arbre *arbre, int x)` prend en paramètre un numéro de feuille ainsi qu'un co-arbre et affiche les voisins du co-arbre. La fonction crée une matrice puis transforme l'arbre en cette matrice ; ensuite on utilise la fonction `voisinsM(taille, matrice, x)` qui retourne toutes les relations telle que $\{y \in V | (x,y) \in R\}$, donc les voisins du co-arbre.

Question 3

Cette fonction réutilise de nombreuses fonctions déjà définies avant pour manipuler des co-arbres (cf. exercice 3).

Question 4

Les fonctions concernant les co-arbres sont fatalement plus longues que celles des matrices en temps d'exécution, puisqu'avant de tester l'adjacence ou le voisinage, une transformation en matrice est réalisée. Il y a donc création de matrice, remplissage par transformation du co-arbre, appel de la fonction consacrée à la matrice et libération de la mémoire.

Comparaison de la complexité en temps des 3 représentations			
Représentation considérée / **Action considérée**	**Matrice d'adjacence**	**Liste d'adjacence**	**Co-arbre**
Adjacence	O(1)	O(n)	$O(n^2)$
Voisinage	O(n)	O(n)	$O(n^2)$

Pour des tests sur l'adjacence, il vaut mieux considérer les matrices pour ces opérations, les autres étant plus longues pour donner le résultat.

Pour le voisinage, les fonctions sont équivalentes quelque soit le type de la représentation (sauf co-arbre car il y a toujours le passage par les matrices) car il faut dans tous les cas parcourir un certain nombre d'éléments, ne serait-ce que pour les retourner à la fin.

La complexité en espace des co-arbres est de 4 fois le nombre d'éléments de l'ensemble. Donc 4*n, donc O(n).

Comparaison de la complexité en espace des 3 représentations			
Représentation considérée	**Matrice d'adjacence**	**Liste d'adjacence**	**Co-arbre**
Complexité en espace	$O(n^2)$ dans tous les cas	$O(n^2)$ dans le pire des cas	O(n)

Finalement, si l'élément important est le stockage des données (pour une très grande relation par exemple), le co-arbre est le plus indiqué. En revanche, il est plus difficile à manipuler ensuite.

Il y a donc une réflexion à avoir avant de modéliser, stocker et traiter une relation, selon sa taille et selon ce que l'on veut en faire.