TP d'algorithmique & structures de données C. Bentz/D. Porumbel

Séance de TP no 5 du 05/12/2024 : Algorithmes de plus courts chemins

Cette séance de TP est dédiée à différents algorithmes de calcul de plus courts chemins dans des graphes (orientés ou non) munis de longueurs, sous différentes hypothèses : algorithme de parcours en largeur d'abord (pour le cas où les longueurs des arcs sont toutes égales au même entier positif, par exemple 1), algorithme de Dijkstra (pour le cas où les longueurs de tous les arcs sont positives ou nulles), avec ou sans utilisation d'un tas min, et algorithme de Floyd-Warshall (pour le cas général sans circuit absorbant).

Pour implémenter ces différents algorithmes en C, on utilisera un certain nombre de types structurés déjà abordés lors de séances précédentes. Pour faciliter la présentation de cette séance, les types représentant des *files* et des *graphes* (orientés ou non, et représentés par des matrices d'adjacence ou par des tableaux de listes d'adjacence) seront fournis, sous la forme de fichiers d'en-tête, intitulés `file.h` et `types_graphe.h` respectivement. En outre, toutes les procédure et fonctions définies sur les files qu'on pourra utiliser seront elles aussi fournies, sous la forme d'un fichier source intitulé `file.c`. De même, deux fonctions permettant de construire un graphe (pour chacune des deux représentations possibles) à partir d'un fichier de données seront fournies, sous la forme d'un fichier source intitulé `types_graphe.c`[1].

Exercice 1 : algorithme de parcours en largeur d'abord

On souhaite ici implémenter en C un algorithme de parcours en largeur d'abord d'un graphe (ou BFS, pour *Breadth-First Search*, en anglais), qui est un type particulier d'algorithme de parcours. La particularité du parcours effectué par cet algorithme est l'ordre dans lequel il permet d'explorer les sommets d'un graphe. En effet, cet ordre est fondamentalement lié à la distance (en nombre d'arcs) des sommets du graphe à un sommet de départ.

Un parcours en largeur d'abord d'un graphe s'implémente naturellement grâce à l'utilisation d'une file (d'entiers, ici). L'idée générale de l'algorithme est donc de progressivement marquer (pour signaler qu'ils ont déjà été visités, puisque aucun sommet n'est marqué initialement) puis stocker les sommets du graphe (en fait, leurs numéros) dans une file, et, à chaque itération, de sélectionner (puis de défiler) le sommet en tête de file, avant d'enfiler (ajouter en queue de file) ses successeurs non encore marqués (qui sont alors marqués).

[1] On compilera les fichiers `file.c` et `types_graphe.c`, puis on ne les utilisera plus.

Au moment où un nouveau sommet est marqué, on peut également calculer sa distance au sommet de départ, à l'aide de la distance du sommet qui a permis de marquer ce nouveau sommet. Informellement, l'algorithme de parcours en largeur d'abord peut donc être décrit de la façon suivante :

- Créer une file vide, et choisir un sommet de départ, noté `dep` ;
- Marquer et enfiler le sommet `dep`, et fixer sa distance à lui-même à 0 ;
- Tant que la file n'est pas vide faire :
  - Défiler le sommet `tete` qui est en tête de file ;
  - Pour chaque successeur `succ` de `tete` faire :
    - Si `succ` n'est pas marqué, alors le marquer, l'enfiler, et fixer sa distance (en nombre d'arcs) à 1 + celle de `tete` ;

Afin de savoir si un sommet donné est marqué, le plus simple est de définir et d'utiliser un tableau de booléens, dont la taille est égale au nombre de sommets du graphe, et qui sera initialisé avec la valeur `false` dans chaque case. Tant que le sommet correspondant n'a pas été marqué, cette valeur restera à `false`. En revanche, dès que le sommet correspondant sera marqué, on fixera cette valeur à `true`. En outre, la boucle sur les successeurs de `tete` devra évidemment être implémentée différemment selon la forme sous laquelle le graphe considéré sera donné (matrice d'adjacence, etc.).

Travail demandé :

1. Implémenter sous la forme d'une procédure écrite en C cet algorithme de parcours, dans le cas où le graphe est donné sous la forme d'une matrice d'adjacence. Cette procédure prendra en paramètres le graphe à explorer, le numéro du sommet de départ, et un tableau d'entiers qui sera destiné à contenir les distances du sommet de départ à tous les sommets du graphe. On supposera que, dans tous les graphes à n sommets que l'on considérera, les sommets seront numérotés de 1 à n.

2. Implémenter sous la forme d'une autre procédure écrite en C cet algorithme de parcours, dans le cas où le graphe est donné sous la forme d'un tableau de listes d'adjacence. Les paramètres de cette procédure seront les mêmes que ceux de la procédure de la question précédente.

3. Justifier en détails dans les commentaires du code le fait que, sous réserve que les opérations sur les files s'exécutent toutes en temps $O(1)$, ces deux procédures s'exécutent en temps $O(n^2)$ et $O(m)$ respectivement, où n est le nombre de sommets du graphe exploré, et m son nombre d'arcs (ou d'arêtes). Pour ce faire, on pourra notamment utiliser le fait que la somme des degrés d'un graphe est égale à $2m$.

4. Écrire, dans un fichier `.c` séparé, un `main` pour tester ces deux procédures sur les fichiers de données qui sont disponibles. On fournira en paramètres, au moment de l'appel au programme, le nom du fichier de données à utiliser, et une valeur indiquant si on souhaite créer à partir de ce fichier un graphe donné sous la forme d'une matrice, ou bien un graphe donné sous la forme d'un tableau de listes d'adjacence.

   Selon le cas, on appellera la méthode correspondante, et on effectuera un affichage rendant compte du résultat obtenu. On prendra soin, ici et tout au long de la séance, d'utiliser systématiquement un fichier *Makefile* pour compiler correctement tous les fichiers qui doivent l'être, et d'inclure tous les fichiers d'en-tête standards qui doivent l'être. Pour finir, on ne différenciera pas les graphes donnés sous forme de matrices d'adjacence (donc non valués) et les graphes donnés sous forme de matrices de distances. Dans les 2 cas, on définira une constante particulière prédéfinie et notée `PAS_ARETE`[2] (par exemple, on pourrait choisir 0 dans le cas d'une matrice d'adjacence), qui signifiera qu'il n'existe pas d'arête (ou d'arc) entre les 2 sommets concernés. Toute autre valeur signifiera l'existence d'une arête (ou d'un arc) entre ces 2 sommets, dont la longueur sera précisément cette valeur, le cas échéant.

Exercice 2 : algorithme de Dijkstra

L'algorithme de Dijkstra permet de déterminer, dans un graphe (orienté ou non) dont les arcs (ou arêtes) sont muni(e)s de longueurs positives ou nulles, tous les plus courts chemins issus d'un sommet donné. Le graphe considéré peut être donné sous la forme d'une matrice de distances ou d'un tableau de listes d'adjacence, mais la complexité au pire cas de cet algorithme est la même dans les deux cas. Pour cette raison, et par souci de simplicité, on ne l'implémentera que dans le cas où le graphe est donné sous la forme d'une matrice de distances. Informellement, cet algorithme peut être décrit ainsi :

- Initialement, aucun sommet n'est traité et n'a de précédesseur défini ;
- Choisir un sommet de départ, le définir comme son propre prédécesseur, et initialiser sa distance à lui-même à 0 ;
- Tant qu'il reste des sommets à traiter faire :
  - Parcourir l'ensemble des sommets non traités (et qui ont déjà un prédécesseur défini) pour trouver celui dont la distance **actuelle** au sommet de départ est la plus petite ;
  - Définir ce sommet `i` comme traité, et, pour chacun de ses successeurs (ou voisins) `succ` non traités, faire :

[2] Cela nécessite d'utiliser le mot-clé `extern` ; cf le fichier `types_graphe.h`.

* Si distance actuelle de `succ` > celle de `i` + longueur de l'arc (ou arête) entre `i` et `succ` (ou si `succ` n'a aucun prédécesseur défini), alors poser cette distance comme égale à celle de `i` + la longueur de l'arc (ou arête) entre `i` et `succ`, et définir `i` comme le prédécesseur actuel de `succ`.

À la fin de l'algorithme, la distance de chaque sommet sera bien égale à la longueur d'un plus court chemin du sommet de départ vers ce sommet, et le prédécesseur de chaque sommet sera bien égal au sommet précédant ce sommet dans un plus court chemin ayant pour origine le sommet de départ.

Travail demandé :

1. Implémenter cet algorithme en C, sous la forme d'une procédure qui prendra en paramètres un graphe donné sous la forme d'une matrice de distances, le numéro du sommet de départ, et deux tableaux d'entiers qui auront respectivement pour vocation de contenir, après exécution de l'algorithme, les numéros des prédécesseurs de tous les sommets, et leurs distances au sommet de départ. Pour déterminer si un sommet a déjà été traité, on pourra définir et utiliser un simple tableau de booléens, comme dans l'exercice précédent. En outre, on pourra définir une constante `ND` (par exemple, -1) qui, par convention, signifiera qu'un sommet donné n'a pas encore de numéro de prédécesseur défini.

2. Justifier en détail dans les commentaires du code le fait que cet algorithme s'exécute en temps $O(n^2)$ pour un graphe à n sommets, et que ceci est vrai à la fois dans le cas où ce graphe est donné sous la forme d'une matrice de distances, ou d'un tableau de listes d'adjacence.

3. Mettre à jour le `main` écrit à l'exercice précédent, de façon à appeler cette procédure lorsque le graphe est donné sous la forme d'une matrice.

Exercice 3 : algorithme de Dijkstra avec tas min

Dans cet exercice, on s'intéresse à une version améliorée (c'est-à-dire, plus rapide) de l'algorithme de Dijkstra implémenté dans l'exercice précédent. Plus précisément, on suppose ici que le graphe considéré est donné sous la forme d'un tableau de listes d'adjacence (sinon, les améliorations apportées ne permettront pas de diminuer la complexité au pire cas).

Cette version améliorée est fondée sur l'utilisation d'un tas min pour déterminer, à chaque itération, le sommet non traité dont la distance actuelle au sommet de départ est la plus petite. Néanmoins, cela nécessite deux modifications importantes dans l'implémentation d'un tel tas min.

D'abord, le tas min utilisé ne contiendra pas des entiers, mais des couples d'entiers *(id, valeur)*, et il vérifiera les propriétés d'un tas min standard vis-à-vis des valeurs uniquement (les *id* serviront d'*identifiants* aux valeurs).

Dans le cadre de l'implémentation de l'algorithme de Dijkstra, les *id* représenteront les numéros des sommets du graphe, et les valeurs les distances actuelles du sommet de départ à ces sommets. Au début de l'algorithme, toutes ces distances seront prises égales à un entier positif arbitraire (par exemple, `PAS_ARETE`), sauf celle associée au sommet de départ, qui sera prise égale à 0. Cela signifie qu'aucun sommet n'est initialement accessible à partir du sommet de départ, à part ce sommet lui-même. Tous ces couples seront insérés, un par un, dans un tas min, après création de ce dernier.

Ensuite, le tas min utilisé devra supporter une opération supplémentaire, qui permet de diminuer n'importe quelle valeur d'un des couples stockés dans le tas, à partir de son *id* associé. Pour cela, il sera nécessaire d'ajouter un champ au type structuré `tas` : un tableau d'entiers (initialisés à `ND`) dont le $i$ème élément donne la position (l'indice), dans le tableau représentant le tas min, du couple contenant l'*id* i et la valeur associée (on suppose donc que les *id* sont égaux à $0, \ldots, n-1$ lorsque le tas contient n éléments). Ce tableau, s'il est correctement mis à jour, permet alors d'avoir un accès direct, dans le tas min, à un couple donné (et donc à une valeur donnée), à partir de l'*id* contenu dans ce couple, et donc associé à cette valeur. Mettre à jour au cours de l'algorithme la distance du sommet de départ au sommet i revient alors à diminuer la valeur associée à l'*id* i dans le tas min.

Cette opération consistant à diminuer une valeur est en réalité très proche de l'opération d'insertion d'un nouvel élément : une fois cette valeur diminuée, on va la faire remonter progressivement dans l'arbre binaire, par échanges successifs, pour trouver la nouvelle place qui est la sienne dans le tas min. Néanmoins, pour mettre correctement à jour le tableau des positions dans le tas min, il est nécessaire de mettre à jour l'indice d'une valeur dans ce tas **à chaque fois** que cette valeur est déplacée (par exemple, lors d'un échange), dans toutes les opérations concernées (insertion d'un nouvel élément, diminution d'une valeur, et suppression de la plus petite valeur).

Travail demandé :

1. Implémenter en C cette nouvelle version des tas min : définition d'un type `element` pour représenter des couples *(id, valeur)*, ajout d'un tableau d'entiers (pour garder trace des positions) et d'une procédure permettant de diminuer la valeur associée à un *id* donné dans le tas, et modification des opérations d'insertion et de suppression de la plus petite valeur pour mettre à jour correctement le tableau des positions.

2. Implémenter en C, sous la forme d'une procédure, cette nouvelle version de l'algorithme de Dijkstra, le graphe considéré étant pour rappel donné sous la forme d'un tableau de listes d'adjacence. On pourra réutiliser les mêmes tableaux de prédécesseurs, de distances, et de sommets traités, que dans l'exercice précédent. Dès qu'un sommet sera traité, on n'oubliera pas de supprimer le couple qui le représente dans le tas.

3. Justifier dans les commentaires du code le fait que la procédure ajoutée à la première question de cette exercice, ainsi que les deux opérations modifiées, s'exécutent en temps $O(\log n)$ pour un tas ayant n éléments. Justifier ensuite, toujours dans les commentaires du code, le fait que cette nouvelle version de l'algorithme de Dijkstra s'exécute en temps $O(m \log n)$ pour un graphe à n sommets et m arcs (ou arêtes), et discuter de son intérêt par rapport à celle de l'exercice précédent.

4. Mettre à jour le `main` écrit à l'exercice précédent, de façon à appeler cette nouvelle implémentation de l'algorithme de Dijkstra lorsque le graphe est donné sous la forme d'un tableau de listes d'adjacence.

Exercice 4 : algorithme (matriciel) de Floyd-Warshall

L'algorithme de Floyd-Warshall est un algorithme matriciel permettant de déterminer les longueurs des plus courts chemins entre toutes les paires de sommets dans un graphe (orienté ou non) dont les arcs (ou les arêtes) sont valué(e)s par des longueurs arbitraires (positives, nulles, ou négatives). La seule condition pour que la solution renvoyée par l'algorithme soit correcte est qu'il n'existe pas dans ce graphe de circuit (ou de cycle) absorbant, c'est-à-dire, dans le cas des plus courts chemins, de circuit (ou de cycle) tel que la somme des longueurs de ses arcs (ou de ses arêtes) soit strictement négative.

Cet algorithme étant par nature matriciel, on ne l'implémentera que dans le cas où le graphe est donné sous la forme d'une matrice de distances (une généralisation des matrices d'adjacence) : pour rappel, la valeur sur la $i$ème ligne et $j$ème colonne est alors égale à la longueur de l'arc (ou de l'arête) (i, j) s'il (ou si elle) existe, et à une valeur par défaut (notée `PAS_ARETE` ici aussi) sinon. Voici une description informelle de cet algorithme très simple :

- Recopier la matrice des longueurs M du graphe à n sommets considéré dans une autre matrice M', et poser $M'_{ii} := 0$ pour tout $i \in \{1, \ldots, n\}$;
- Pour tout k allant de 1 à n, pour tout i allant de 1 à n, et pour tout j allant de 1 à n, poser $M'_{ij} \leftarrow \min(M'_{ij}, M'_{ik} + M'_{kj})$.

  (À la fin, on a : M'_{ij} = longueur d'un plus court chemin de i vers j.)

Travail demandé :

1. Implémenter cet algorithme en C, sous la forme d'une fonction qui prendra en paramètre un graphe donné sous la forme d'une matrice, et renverra le résultat obtenu sous la forme d'un graphe du même type.

2. Justifier dans les commentaires du code le fait qu'elle s'exécute en temps $O(n^3)$ pour un graphe à n sommets, puis mettre à jour le `main` obtenu à l'issue des exercices précédents, de façon à appeler cette fonction lorsque le graphe est donné sous la forme d'une matrice.