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

Séance de TP no 6 du 12/12/2024 : Algorithme de Kruskal pour les arbres couvrants de poids minimum

Cette séance de TP est dédiée à deux implémentations différentes de l'algorithme de Kruskal qui, étant donné un graphe connexe non orienté dont les arêtes sont pondérées (par des poids qu'on supposera entiers), calcule un arbre couvrant de poids minimum de ce graphe. Pour rappel, un arbre est un graphe connexe sans cycle, ou, de façon équivalente, un graphe connexe qui a une arête de moins que de sommets. Une troisième définition équivalente est qu'un arbre est un graphe sans cycle qui a une arête de moins que de sommets. Un arbre est dit *couvrant* s'il contient tous les sommets du graphe, et le *poids* d'un tel arbre est égal à la somme des poids de ses arêtes.

Exercice 1 : algorithme de Kruskal (version simple)

On souhaite ici implémenter en C l'algorithme de Kruskal, sans utiliser de structure de données particulière pour la deuxième phase de l'algorithme. La première phase, elle, sera effectuée à l'aide d'un tri par tas.

Informellement, l'algorithme de Kruskal appliqué à un graphe connexe G peut être décrit de la façon suivante :

- Initialiser un graphe G' contenant tous les sommets de G mais aucune arête, et trier les arêtes de G par ordre de poids croissants ;
- Pour chaque arête de G dans cet ordre faire :
  - Si les deux extrémités de l'arête courante ne sont pas dans la même composante connexe de G', alors ajouter cette arête à G' ;
  - Si G' contient une arête de moins que de sommets, alors STOP.

À la fin de l'algorithme, les arêtes de G' sont précisément les arêtes d'un arbre couvrant de poids minimum de G. En réalité, implémenter cet algorithme nécessite de prendre en compte deux difficultés : la première phase (le tri des arêtes), et la seconde phase (la sélection des arêtes de G', en fonction de l'évolution de ses composantes connexes).

Dans cet exercice, on considère la version la moins compliquée de cet algorithme, qui utilise un simple tableau d'entiers pour la seconde phase.

Il convient de noter que cette version est la moins efficace (en temps), et c'est notamment pour cela qu'on considèrera que le graphe est donné sous la forme d'une matrice (puisque cela ne change pas sa complexité au pire cas).

Pour effectuer la première phase de l'algorithme de Kruskal, on pourrait utiliser n'importe quel algorithme de tri efficace (pour trier les m arêtes en temps $O(m \log m)$), mais on fait le choix ici d'utiliser un tri par tas. La principale difficulté de cette phase est alors simplement de relier les entiers triés (les poids) aux arêtes auxquelles ils sont associés dans G.

Une solution est de stocker dans un tableau toutes les arêtes du graphe considéré (sous la forme de couples d'entiers, ou `element`, représentant les numéros des extrémités des arêtes). Ainsi, à chacune d'entre elles sera associé un identifiant (l'indice du tableau où l'arête est stockée), et on pourra alors insérer une par une les arêtes dans un tas min (qui sera initialement vide) similaire à ceux implémentés lors de la séance de TP précédente, en utilisant comme clé cet identifiant et comme valeur le poids de l'arête.

Comme dans le cas du tri par tas vu lors de la première séance de TP, on récupérera ensuite les éléments de ce tas min un par un, en accédant de façon itérative à celui de plus petite valeur. À chaque fois, on supprimera alors l'élément courant du tas min, et on vérifiera si l'ajout de l'arête associée crée ou non un cycle dans G'. Il ne reste donc plus qu'à déterminer comment on peut tester si l'ajout d'une arête crée ou non un cycle dans G'.

Dans cet exercice, on se contentera d'utiliser une solution simple : un tableau d'entiers `cc`, dont la taille est le nombre de sommets de G (et donc de G'). La valeur stockée à l'indice i de ce tableau représentera le numéro de la composante connexe de G' où se trouve le sommet $i+1$: initialement, G' ne contient aucune arête, donc chaque sommet est dans sa propre composante connexe. Lorsqu'on cherche à ajouter une arête (disons $(i+1, j+1)$) à G', on vérifie si les numéros des 2 composantes connexes de ses extrémités (c'est-à-dire `cc[`i`]` et `cc[`j`]`) sont les mêmes ou non. Si c'est le cas, alors ces 2 sommets sont déjà dans la même composante connexe, et l'ajout de cette arête créera donc un cycle ; cette arête est alors ignorée (elle sera absente de G'). Sinon, ajouter cette arête aura pour effet de fusionner les 2 composantes connexes de ses extrémités. On ajoute alors l'arête $(i+1, j+1)$ dans G', et on met à jour le tableau contenant les numéros des composantes connexes : on choisit le numéro de la composante connexe d'une des deux extrémités de l'arête (par exemple, `cc[`i`]`), et on place alors dans la composante connexe numéro `cc[`i`]` tous les sommets qui étaient anciennement dans la composante connexe de numéro `cc[`j`]`, en parcourant tout le tableau.

Pour finir, on pourra (mais ce n'est pas absolument nécessaire, puisque cela ne change pas la complexité au pire cas) implémenter la dernière ligne de l'algorithme de Kruskal (qui interrompt l'algorithme une fois qu'on a sélectionné suffisamment d'arêtes dans G'). Pour ce faire, on pourra utiliser soit une boucle `while`, soit une boucle `for`, puisque les 2 permettent d'intégrer une double condition en C (ce n'est pas vrai dans tous les langages).

Travail demandé :

1. Implémenter l'algorithme de Kruskal sous la forme d'une fonction écrite en C. Cette fonction prendra en paramètre un graphe (G) donné sous la forme d'une matrice, et renverra l'arbre résultant de l'application de cet algorithme sous la forme d'un autre graphe (G') du même type.

2. Justifier dans les commentaires du code le fait que le temps d'exécution de cette fonction est $O(m \log n + n^2)$ si G a n sommets et m arêtes.

3. Écrire, dans un fichier `.c` séparé, un `main` pour tester cette fonction sur les fichiers de données disponibles. On fournira en paramètre, lors de l'appel au programme, le nom du fichier de données à utiliser.

Exercice 2 : algorithme de Kruskal (version plus sophistiquée)

Cet exercice s'intéresse à une implémentation un peu plus sophistiquée de l'algorithme de Kruskal. Plus précisément, la seconde phase de l'algorithme sera réalisée via une structure "union-find" ("unir et trouver", en français).

Une telle structure est utilisée pour représenter des ensembles d'éléments (un représentant étant associé à chaque ensemble) et pour réaliser des unions de tels ensembles. Concrètement, dans une telle structure, 3 opérations de base sont disponibles : (*i*) une opération de création d'un singleton (ensemble constitué d'un seul élément), (*ii*) une opération (`trouver`) de recherche du représentant de l'ensemble où se trouve un élément donné, et (*iii*) une opération (`unir`) d'union de 2 ensembles pour n'en former qu'un seul.

Il existe différentes façons, plus ou moins simples à implémenter, de définir une telle structure. Le choix de telle ou telle façon de le faire a un impact direct sur les complexités des 3 opérations, et doit donc être pertinent. On présente ici une de ces façons, qui est à la fois suffisamment simple à implémenter, et suffisante pour garantir ensuite la meilleure complexité au pire cas possible en l'utilisant à l'intérieur de l'algorithme de Kruskal.

Dans la structure "union-find" considérée, tout ensemble sera représenté par un chaînage, constitué (comme dans le cas des listes chaînées) de maillons (ou cellules), chacun étant associé à un élément de cet ensemble. La principale différence avec une liste chaînée classique est le fait que, en plus de contenir une référence vers le maillon suivant, chaque maillon contient deux références supplémentaires, vers le premier et vers le dernier maillon de la liste. Chaque maillon contient également 2 autres informations : une donnée (entière), et un second entier qui représente le nombre de maillons dans la liste (c'est-à-dire le cardinal de l'ensemble associé). Dans le cas de l'algorithme de Kruskal, chaque élément représentera un sommet du graphe G' (et la donnée qui y est stockée sera le numéro de ce sommet), et chaque ensemble (liste) représentera une composante connexe de ce graphe.

Initialement, chaque sommet de G' (donc chaque élément) est dans sa propre composante connexe (son propre ensemble). L'opération (i), qui sert à créer de tels ensembles de taille 1, prendra donc la forme d'une fonction écrite en C dont l'unique paramètre sera le numéro du sommet concerné. Cette fonction réservera un emplacement mémoire de la taille requise pour un maillon, effectuera toutes les initialisations nécessaires pour ce maillon (la taille d'un singleton est 1, un maillon unique n'a pas de maillon suivant, etc.), puis renverra une référence vers le nouveau maillon ainsi créé.

En ce qui concerne l'opération (ii), qui a pour rôle de fournir le représentant de l'ensemble où se trouve un élément donné, elle prendra également la forme d'une fonction écrite en C. Plus précisément, on considérera systématiquement que le représentant de chaque ensemble (liste) est son premier élément (maillon). Cette fonction aura donc un seul paramètre (un maillon d'une liste, associé à un élément particulier d'un ensemble), et renverra une référence vers le premier maillon de la liste où se trouve le maillon considéré.

Enfin, en ce qui concerne l'opération (iii), qui a pour rôle de réaliser l'union de 2 ensembles, elle prendra elle aussi la forme d'une fonction écrite en C, dont les 2 paramètres seront 2 maillons qui se trouvent dans les 2 listes qu'on souhaite réunir, et qui renverra une référence vers le premier maillon de la liste résultant de cette union. Informellement, cette opération peut être décrite de la façon suivante :

- Trouver les représentants des 2 ensembles auxquels appartiennent les 2 éléments passés en paramètres, grâce à l'opération `trouver` ;
- Comparer les tailles des 2 listes associées à ces 2 ensembles, pour déterminer la plus grande, qu'on notera l_1, et qu'on mettra devant l'autre (qui, elle, s'appellera l_2), puis qu'on renverra à la fin de l'opération ;
- Mettre à jour la taille de l_1, uniquement pour le maillon situé en tête de l_1 (les autres maillons de l_1 n'auront pas nécessairement la bonne information, mais cette mise à jour leur serait de toute façon inutile), en fixant cette taille à la somme des tailles actuelles de l_1 et l_2 ;
- Raccrocher l_2 à l_1, en faisant du premier maillon de l_2 le maillon suivant du dernier maillon de l_1 ;
- Mettre à jour l'adresse du dernier maillon de l_1, uniquement pour le maillon qui se trouve en tête de l_1 (les autres maillons de l_1 n'auront pas nécessairement la bonne information, mais cette mise à jour leur serait de toute façon inutile) ;
- Parcourir les maillons de l_2, et modifier pour chacun la référence vers le 1er maillon de la liste en utilisant l'adresse du 1er maillon de l_1 ;
- Renvoyer l_1 comme liste résultant de l'union entre l_1 et l_2.

Afin de bénéficier du gain de complexité offert par l'utilisation d'une telle structure pour implémenter l'algorithme de Kruskal avec le meilleur temps d'exécution au pire cas possible, il est nécessaire de considérer que le graphe G est donné sous la forme d'un tableau de listes d'adjacence. Dans la suite de cet exercice, on supposera donc systématiquement que c'est le cas.

Travail demandé :

1. Implémenter en C la structure "union-find" décrite précédemment, sous la forme d'un type structuré (qui contiendra donc 5 champs), muni des opérations (i), (ii) et (iii). On écrira les 3 fonctions C associées aux 3 opérations dans un fichier `.c` séparé, et on déclarera le type structuré associé et les signatures de ces 3 fonctions C dans un fichier d'en-tête, qu'on n'oubliera pas d'inclure dans le fichier `.c`.

2. Justifier en détails dans les commentaires du code le fait qu'exécuter $O(m)$ opérations (en tout) de type création, `trouver` et `unir`, dont n opérations de création (ce qui signifie simplement que le nombre total d'éléments est n), nécessite dans le pire cas un temps $O(m + n \log n)$, si on utilise la structure "union-find" de la question précédente.

   Pour cela, on justifiera d'abord que chaque opération de type création ou `trouver` s'exécute en temps $O(1)$, et que c'est également le cas de toutes les étapes de chaque opération `unir`, **sauf** l'avant-dernière. On justifiera ensuite que le champ référençant le représentant de l'ensemble dans lequel un élément se trouve est mis à jour au plus $O(\log n)$ fois pour chaque élément, grâce au fait qu'à chaque fois qu'on réalise l'union de 2 listes, c'est la plus grande des 2 qui est mise devant.

3. Implémenter en C l'algorithme de Kruskal, à l'aide de la structure "union-find" précédente, en supposant que le graphe G est donné sous la forme d'un tableau de listes d'adjacence. L'algorithme prendra la forme d'une fonction écrite en C (dans le même fichier `.c` que celle de l'exercice précédent), dont le seul paramètre sera un graphe (G) donné sous la forme d'un tableau de listes d'adjacence, et qui renverra l'arbre résultant de l'application de cet algorithme sous la forme d'un autre graphe (G') du même type. Justifier ensuite en détails dans les commentaires du code le fait que cette version de l'algorithme de Kruskal s'exécute en temps $O(m \log n)$ dans le pire cas, et que c'est la meilleure complexité au pire cas qu'on puisse obtenir pour cet algorithme.

4. Mettre à jour le `main` écrit à l'exercice précédent, de façon à appeler cette nouvelle implémentation de l'algorithme de Kruskal lorsque le graphe est donné sous la forme d'un tableau de listes d'adjacence. Comme lors de séances de TP précédentes, on utilisera un `Makefile` pour compiler correctement tous les fichiers qui doivent l'être.

Au moment de l'appel au programme, on fournira alors en paramètres 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 de poids (auquel cas on appellera ensuite la fonction implémentée dans l'exercice précédent), ou bien un graphe donné sous la forme d'un tableau de listes d'adjacence (auquel cas on appellera ensuite la fonction implémentée dans cet exercice).