Travaux Pratiques n°10

Cours d'Informatique de Deuxième Année

—Licence MI L2.2—

Calcul d'arbre recouvrant de poids minimal par l'algorithme de Prim

Nous cherchons dans ce TP à implanter l'algorithme de Prim permettant de trouver l'arbre recouvrant (*spanning tree*) de poids minimal d'un graphe non orienté connexe valué.

Arbre recouvrant de poids minimal On considère un graphe G valué : on associe à chacune des arêtes du graphe un poids (entier ou réel). Un arbre recouvrant de G est un arbre comprenant tous les sommets de G (en supposant G connexe). On définit le poids d'un tel arbre par la somme des poids des arêtes dont il est constitué. Un arbre A recouvrant de G est dit minimal si tout autre arbre recouvrant de G possède un poids supérieur ou égal à A. On notera qu'il peut exister plusieurs arbres recouvrants minimaux pour un même graphe. Bien sûr, le problème de l'arbre recouvrant de poids maximal se déduit du problème de l'arbre recouvrant de poids minimal en utilisant des valeurs de poids opposées.

Quelques problèmes connexes Il existe de nombreuses variantes du problème de l'arbre recouvrant minimal, parmi elles nous pouvons citer :

- Le problème de l'arbre recouvrant minimal sur un espace euclidien. Dans cette variante, chaque sommet est un point d'un espace vectoriel euclidien. On cherche ensuite à calculer l'arbre recouvrant minimal : le poids de chaque arête est défini par la distance euclidienne séparant les deux sommets la constituant.
- Le problème de l'arbre recouvrant minimal sur k sommets. Soit un graphe G de n sommets : on cherche à trouver un arbre recouvrant minimal utilisant au moins k sommets définis ($k < n$). Contrairement au problème de l'arbre recouvrant minimum sans contrainte, ce problème est NP-complet.

Algorithmes de calcul d'arbres recouvrants Il existe deux algorithmes classiques de calcul d'arbre recouvrant de poids minimal d'un graphe connexe :

- L'algorithme de Kruskal proposé par *Joseph Kruskal* en 1956.
- L'algorithme de Prim proposé par *Robert Prim* en 1957.

Algorithme de Prim Nous décrivons ici l'algorithme de Prim (voir la figure 1) :

1. On choisit un sommet quelconque de démarrage. Le choix de ce sommet est uniquement important si le graphe manipulé n'est pas connexe (ce choix définit alors une sous-partie connexe du graphe pour le calcul de l'arbre recouvrant).
2. Lors de chaque itération de l'algorithme, on ajoute un nouveau sommet à l'arbre recouvrant en construction. On choisit ce sommet en sélectionnant l'arête de plus petit poids connectant un sommet déjà présent dans l'arbre construit avec un sommet non présent.

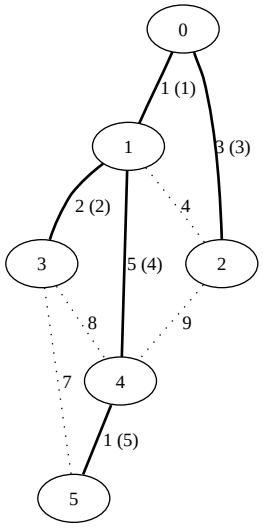

FIG. 1 – Exemple de graphe non orienté valué avec son arbre recouvrant de poids minimal (12) calculé avec l'algorithme de Prim. Les arêtes en gras appartiennent à l'arbre recouvrant de poids minimal (la valuation de chaque arête est suivie, entre parenthèses, par le numéro de l'itération ayant entraîné son ajout) ; les arêtes en pointillés sont les arêtes du graphe non-sélectionnées pour intégrer l'arbre recouvrant minimal.

Implantation naïve Une implantation naïve consiste à utiliser une liste chaînée de toutes les arêtes : à chaque itération, on reparcourt toute la liste pour trouver l'arête de poids minimal reliant un sommet présent dans l'arbre à un sommet non présent. Cette implantation est très coûteuse : pour un graphe de G de $|S|$ sommets et de $|A|$ arêtes, il est nécessaire pour les $n-1$ itérations consistant à ajouter $n-1$ sommets dans l'arbre (le sommet d'origine étant choisi préliminairement) de parcourir la liste des arêtes non encore utilisées : ce parcours est réalisé en $O(|A|)$. En conséquence la complexité temporelle de l'algorithme est $O(|S||A|)$.

▸ **Exercice 1.** ***Chargement du graphe*** *On exprime les graphes valués sous forme matricielle : la valeur (i, j) de la matrice est le poids associé à l'arête (i, j). On considère ici des graphes valués par des* int *non orientés (l'absence d'arête peut être symbolisée par une valeur* INT_MAX*) : en conséquence, les matrices manipulées sont symétriques.*
Écrire une structure de données permettant de manipuler un graphe valué non orienté sous forme matricielle. Ensuite rédiger une fonction permettant de charger un graphe à partir d'une matrice indiquée dans un fichier (la première ligne indiquant le nombre de sommets de la matrice).

▸ **Exercice 2.** ***Structure d'arbre*** *Nous cherchons maintenant à trouver une structure pour mémoriser l'arbre recouvrant minimal que nous souhaitons construire. Une structure envisageable est l'utilisation de deux pointeurs pour chaque sommet de l'arbre :*
- *Un pointeur vers le fils gauche (premier fils).*
- *Un pointeur vers le frère droit.*

Écrire les fonctions permettant d'initialiser une telle structure et d'ajouter un fils à un noeud de l'arbre en temps constant (on insérera le noeud en tête de la liste des frères)

▸ **Exercice 3.** ***Recherche d'un arête à ajouter*** *Écrire une fonction recherchant l'arête de plus petit poids reliant un sommet présent dans l'arbre recouvrant en construction à un sommet non présent dans l'arbre. Dans le cadre de l'implantation naïve, cette fonction doit parcourir l'ensemble de la matrice pour la sélection de l'arête ce qui est réalisé en temps* $O(|S|^2)$.
Afin de vérifier si un sommet est déjà présent dans l'arbre ainsi que pour trouver le pointeur vers ce sommet dans l'arbre, on maintient un tableau de $|S|$ *cellules : chacune de ces cellules comporte soit un pointeur* NULL *si le sommet n'est pas présent dans l'arbre, soit un pointeur vers le sommet dans l'arbre.*

▸ **Exercice 4.** ***Implantation de l'algorithme naïf*** *Implanter maintenant l'algorithme naïf. Il suffit pour cela de fixer un sommet de démarrage (par exemple le sommet* 0*) que l'on ajoute dans l'arbre (racine de l'arbre). Ensuite, pour chaque itération, on utilise la fonction rédigée précédemment pour l'exercice 3 afin de trouver l'arête à ajouter dans l'arbre : on ajoute le nouveau sommet dans l'arbre. La condition d'arrêt de l'algorithme est satisfaite lorsque l'on atteint une itération où l'on ne peut plus trouver d'arête reliant un sommet de l'arbre vers un sommet extérieur à l'arbre : si le graphe est connexe,* $|S|-1$ *itérations rajoutant* $|S|-1$ *sommets dans l'arbre seront nécessaires.*

Petite optimisation de l'algorithme naïf proposé Une petite optimisation de l'algorithme naïf consiste à utiliser une liste d'arêtes triées. Ceci permet d'arêter l'exploration de la liste des arêtes dès qu'une arête reliant un sommet de l'arbre vers un sommet non présent est trouvée : cette arête est de poids minimal en raison du tri préalable de la liste. La structure de matrice ne suffit donc plus : il est nécessaire d'introduire une liste chaïnée d'arêtes ; à chaque utilisation d'une arête, on peut la supprimer de la liste.

▸ **Exercice 5.** ***Implantation du tri de la liste des arêtes*** *Écrire une structure de données pour maintenir une liste d'arêtes (une arête étant définie comme deux indices entiers). Adapter l'algorithme précédemment écrit en triant la liste d'arêtes (la fonction* qsort*) de la bibliothèque standard pourra être utilisée) puis en travaillant sur la liste d'arêtes plutôt que sur la matrice.*

Vers un algorithme optimal

Afin d'améliorer la complexité, une idée consiste à maintenir un tas (file de priorité) contenant l'ensemble des sommets non encore rattachés à l'arbre et associant pour chacun de ces sommets l'arête de coût minimal vers un sommet extérieur à l'arbre en construction. À chaque itération, il suffit alors d'extraire le sommet associé à l'arête de coût minimal du tas : cette opération est réalisée en temps $O(\log |S|)$. D'autre part, lors de chaque itération, lorsqu'un sommet i est intégré au graphe, il est nécessaire d'examiner toutes les arêtes comprenant ce sommet i. On met ensuite à jour les arêtes associés à chaque sommet restant dans le tas : si pour un sommet j du tas, l'arête (i, j) est de meilleur coût que l'arête mémorisée, on met à jour l'arête associée. Cela nécessite de déplacer le sommet dans le tas conformément au nouveau coût.

Complexité de l'algorithme Analysons la complexité de l'algorithme :
– Le retrait de chaque sommet du tas est réalisé en $O(\log |S|)$. Il est nécessaire de retirer $|S| - 1$ sommets du tas lors des $|S| - 1$ itérations.

– Pour chaque sommet sélectionnée lors de chaque itération, on met à jour les sommets liés dans le tas. Lors du déroulement de l'algorithme, au plus $|A|$ mises à jours sont nécessaires. Pour un tas binaire d'implantation classique, chaque mise à jour est réalisée en $O(\log |S|)$.

En conclusion, la complexité temporelle de cet algorithme avec l'utilisation d'un tas est en $O(|S| \log |S| + |A| \log |S|)$.

▸ **Exercice 6.** ***Pour les courageux : utilisation d'un tas binaire*** *Implanter l'algorithme en utilisant un tas binaire. Il est bien évidemment possible de réutiliser le tas générique implanté lors d'un TP précédent (il sera sans doute nécessaire d'y ajouter une fonction de mise à jour de valeur dans le tas ainsi que le maintien d'une table indiquant la position de chaque sommet dans le tas pour la mise à jour).*

Faisons toujours mieux... Peut-on obtenir une meilleure complexité temporelle ? Oui, en utilisant une nouvelle structure de données (introduite par Michael Fredman et Robert Tarjan en 1984) : une file de priorité implantée par un tas de Fibonacci. Lorsqu'un tas de Fibonacci contient n éléments, le retrait de l'élément minimal est réalisé en temps $O(\log n)$ mais l'insertion ainsi que la mise à jour d'élément peut être réalisée en temps amorti constant ! La complexité temporelle de l'algorithme de Prim devient alors $O(|S| \log |S| + |A|)$, gain particulièrement appréciable pour les graphes denses.

▸ **Exercice 7.** ***Pour les super-hyper-méga motivés : utilisation d'un tas de Fibonacci*** *Implanter un tas de Fibonacci afin de pouvoir faire fonctionner l'algorithme de Prim en $O(|S| \log |S| + |A|)$.*

Nous voilà arrivés au terme des TP. Nous espérons que vous avez trouvé ceux-ci intéressants et enrichissants.
Bonne chance et bon courage pour l'examen.

```
L'Homme animé par l'esprit scientifique désire sans doute savoir,
mais c'est aussitôt pour mieux interroger.
        -+- Gaston Bachelard -+-
```