Algorithmes avancés sur les graphes

I Arbre couvrant de poids maximal dans un graphe pondéré

I.1 Arbre couvrant, poids d'un graphe

On rappelle la définition vue dans le chapitre précédent sur les graphes :

Définition 1

Soit $G = (S, A)$ un graphe non orienté connexe.
On appelle arbre couvrant de G un graphe de la forme $G' = (S, A')$ avec $A' \subset A$ tel que G' soit acyclique.

Autrement dit, un arbre couvrant de G est un sous-graphe de G obtenu en retirant des arêtes jusqu'à avoir un arbre, mais en retirant suffisamment peu d'arêtes pour que cet arbre passe par tous les sommets de G.

Évidemment, si G n'est pas connexe, il n'a pas d'arbre couvrant (on peut obtenir des arbres couvrants pour chacune de ses composantes connexes).

Exemple 1 : Un exemple avec un dessin.

Sauf qu'ici on va s'intéresser à des graphes **pondérés**.

Définition 2

Soit $G = (S, A)$ un graphe pondéré. On appelle poids de G la somme des poids des arêtes de G.

On s'intéresse dans la suite à la détermination d'un arbre couvrant de poids minimal.

Exemple 2 : Un exemple avec un dessin.

I.2 Algorithme de Kruskal

L'algorithme de Kruskal est un algorithme glouton permettant de déterminer un arbre couvrant de poids minimum dans un graphe pondéré. On construit l'ensemble des arêtes d'un tel arbre en rajoutant les arêtes une à une, en ajoutant à chaque itération une arête de poids minimal qui ne crée pas de cycle, et en s'arrêtant lorsque ce n'est plus possible.

Remarques 1 :

1. C'est un algorithme glouton : à chaque itération on choisit "la meilleure" arête à rajouter. Obtient-on bien globalement "le meilleur" arbre couvrant (à supposer qu'on obtienne un arbre couvrant) ? Il va falloir le montrer.
2. Il y a une autre stratégie gloutonne possible : à chaque itération, on pourrait choisir une arête de poids minimal qui conserve la connexité (c'est complètement dual : un arbre est un graphe acyclique et connexe !).
   Cette stratégie concurrente est celle de l'algorithme de Prim (mais il n'est pas explicitement au programme).

Pour mettre en œuvre l'algorithme de Kruskal il faut donc pouvoir à tout moment déterminer le plus rapidement possible une arête de poids minimal parmi les arêtes restantes.

1. Une solution possible est d'effectuer un prétraitement pour obtenir la liste triée toutes les arêtes : cela coûterait $O\big(a\lg(a)\big)$ en notant $a = |A|$ le nombre d'arêtes du graphe G, puis chaque détermination d'une arête de poids minimal coûterait $O(1)$. On examinerait donc toutes les arêtes en $O\big(a\lg(a)\big)$. On ne va pas choisir cette solution.
2. Une autre solution possible permet d'examiner toutes les arêtes en $O\big(a\lg(a)\big)$: laquelle ?

   .....

   On va choisir cette dernière solution.

I.3 Preuve de correction

Théorème 1

L'algorithme de Kruskal décrit plus haut, appliqué à un graphe connexe $G = (S, A)$, retourne bien un arbre couvrant de poids minimal.

Démonstration. Notons $T = (S, A')$ le graphe donné par l'algorithme de Kruskal.

1. Montrons que T est un arbre (c'est-à-dire acyclique et connexe) couvrant.
   • T est acyclique par construction : l'algorithme ne rajoute des arêtes à A' que lorsqu'elle ne crée pas de cycle.
   • Montrons que T est connexe et couvrant, c'est-à-dire que pour tout couple de sommets de S, il existe un chemin de l'un vers l'autre qui n'emprunte que des arêtes de A'.
   Soient u et v deux sommets de G. On veut montrer que u et v sont dans la même composante connexe dans T. Supposons par l'absurde que ce ne soit pas le cas, notons U la composante connexe de u dans T et V la composante connexe de v dans T. Il n'existe donc pas d'arête entre U et V dans T, mais il en existe une dans G car G est connexe. Cette arête aurait dû être ajoutée à A' puisque, seule, elle ne crée pas de cycle. C'est une contradiction !
2. Montrons que T est de poids minimal parmi les arbres couvrant. Soit $T_m = (S, A_m)$ un arbre couvrant de poids minimum : on veut montrer que T et T_m ont le même poids. Pour alléger les notations notons, pour tout graphe G, $p(G)$ le poids de G.

   • Si $T = T_m$, on a bien en particulier $p(T) = p(T_m)$.

   • Sinon il existe au moins une arête dans T_m et pas dans T. En effet, si ce n'était pas le cas, on aurait $A_m \subset A'$. Il existerait donc une arête dans T qui ne serait pas dans T_m, mais comme T_m est un arbre couvrant, cette arête créerait un cycle dans T, ce qui n'est pas possible.

Considérons une arête **de poids minimal** $e_m = u \to v$ qui soit dans T_m et pas dans T. Comme T est connexe, il existe un chemin dans T reliant u et v, et en particulier ce chemin comprend nécessairement une arête e qui n'est pas dans T_m, car T_m étant un arbre il ne contient pas de cycle. On a alors $p(e) = p(e_m)$ sinon Kruskal aurait ajouté e_m à A'.

Notons $T_2 = (S, A_2)$ avec $A_2 = \big(A' \backslash \{e\}\big) \cup \{e_m\}$. Par construction T_2 est un arbre couvrant et vérifie $p(T) \leqslant p(T_2)$, mais le nombre d'arêtes distinctes entre T_2 et T_m a diminué de 1 par rapport à celui qu'on avait entre T et T_m.

On répète le même processus sur T_2, ce qui nous donne T_3, puis sur T_3 ce qui nous donne T_4, etc. On finit nécessairement par obtenir T_m puisque le nombre d'arêtes distinctes diminue à chaque étape.

On a alors $p(T) \leqslant p(T_2) \leqslant p(T_3) \leqslant \cdots \leqslant p(T_m)$. Mais par définition de $p(T_m)$ on a aussi $p(T_m) \leqslant p(T)$ et donc $p(T) = p(T_m)$, CQFD. 

I.4 Implémentation naïve

On reprend l'implémentation impérative des files de prioriété par tas impératifs vu dans le chapitre 1 et déjà utilisée pour Dijkstra dans le chapitre 2 (dans le fichier *3.TasMin.ml* en ligne j'ai modifié pour obtenir des *tas min* au lieu de *tas max* histoire qu'on ne s'embête plus à tout opposer).

Écrire une fonction `stocker_aretes : graphe_p → (float,int*int) file_de_priorite` qui stocke les arêtes de son argument dans une file de priorité. Le graphe donné en argument est décrit par listes d'ajacences (`g.(i)` est la liste des couples `(j,w)` tel que $i \to j$ est une arête de `g` de poids `w`). La priorité d'une arête $i \to j$ dans la file est son poids `w` et sa valeur est le couple `(i,j)`.

On va maintenant vider cette file de priorité en se demandant pour chaque arête successive si elle crée un cycle ou pas : on l'ajoute à A' si et seulement si elle n'en crée pas.

Pour déterminer si l'arête crée un cycle ou pas, une méthode naïve est d'écrire un parcours de graphe, en largeur ou en profondeur (on sait encore faire ça?). On verra en TP une méthode beaucoup moins naïve, mais qui utilise une nouvelle structure de données.

Écrire une fonction `accessible : graphe_p → int → int → bool` telle que `accessible g u v` indique s'il existe un chemin de `u` vers `v` dans un graphe `g` (bien sûr, cette fonction sera appliquée au sous-graphe en cours de construction pendant l'algorithme et non pas au graphe dont on veut extraire un arbre couvrant de poids minimal).

Finalement, écrire une fonction `kruskal : graphe_p → graphe_p`.

II Couplage de cardinal maximal dans un graphe biparti

II.1 Graphes bipartis

On s'intéresse ici uniquement à des graphes non orientés. Ils pourraient être pondérés même si on ne considère pas de pondération dans la suite.

Définition 3

Un graphe non orienté $G = (S, A)$ est dit biparti lorsqu'il existe deux ensembles $B, R \subset S$ tels que $B \amalg R = S$ et toutes les arêtes du graphes ont pour extrémités un sommet de B et un sommet de R.

Exemples 3 : Quelques représentations de graphes bipartis.

Exemple 4 : On peut imaginer quelques exemples dans la vie courante : par exemple les sommets de B pourraient être les élèves admis à tous un concours de CPGE et ceux de R les écoles accessibles depuis ce concours, une arête étant présente entre un élève et une école si l'élève souhaite intégrer cette école.

Remarque 2 : Un graphe biparti n'est donc rien d'autre qu'un graphe .....

Pour le voir, il suffit de .....

Proposition 1

Un graphe est biparti si et seulement si il ne contient pas de cycle de longueur impaire.

Démonstration. $\boxed{\Rightarrow}$

$\boxed{\Leftarrow}$ Il suffit de traiter le cas d'un graphe connexe car .....

Application 1 : Ceci donne un algorithme pour déterminer si un graphe connexe (pour faire simple ici) est biparti.

Implémentons-le.

II.2 Couplage

Définition 4

On appelle couplage dans un graphe biparti $G = (B \amalg R, A)$ un sous-graphe de G pour lequel chaque sommet a pour degré au plus 1, ou de façon équivalente puisque le graphe est biparti, tel que chaque sommet de B et chaque sommet de R est l'extrémité d'au plus une arête.

Remarque 3 : Le premier point de cette définition permet de définir plus généralement les couplages dans un graphe non orienté quelconque, mais le programme demande de se limiter aux graphes bipartis, hors desquels un certain nombre d'interprétations de ce qu'est un couplage, et qu'on va faire ci-dessous, ne tiennent plus.

Exemples 5 : Reprenons les exemples précédents.

Qu'est-ce qu'un "meilleur" couplage ? Tout dépend du contexte (et plus encore quand le graphe est orienté), mais une bonne question est de **trouver un couplage de cardinal maximal**. Dans l'exemple des élèves et des écoles par exemple, c'est l'intérêt des écoles de remplir et c'est aussi l'intérêt des élèves d'intégrer une école qui les intéresse (cet exemple est simpliste car il ne tient pas compte des *préférences* des élèves ni de celles des écoles, mais il reste parlant).

II.3 Chemin augmentant

Définition 5

Soit C un couplage d'un graphe biparti $G = (B \amalg R, A)$ et s un sommet de G.
On dit que s est libre pour C si s n'est l'extrémité d'aucune une arête de C.

Un sommet libre pour un couplage est donc un sommet qui n'a pas été sélectionné par ce couplage.
Le pauvre. Il est tout triste. ☹

Définition 6

Soit C un couplage d'un graphe biparti $G = (B \amalg R, A)$ et s un sommet de G.
Un chemin $(x_1 \to y_1,\ y_1 \to x_2,\ x_2 \to y_2,\ \ldots,\ x_n \to y_n)$ est dit **augmentant** pour C lorsque :

- x_1 et y_n sont libres pour C ;
- deux arêtes successives de ce chemin sont l'une dans C, l'autre pas.

Exemples 6 : Reprenons les exemples précédents.

Théorème 2

En identifiant un couplage à l'ensemble de ses arêtes, et un chemin à l'ensemble de ses arêtes, on a :

1. Si C est un couplage et c un chemin augmentant pour C, alors $C\Delta c$ est aussi un couplage, de cardinal $|C|+1$.
2. Si C est un couplage et qu'il n'existe pas de chemin augmentant pour C, alors C est un couplage de cardinal maximal.

Exemple 7 : Reprenons un exemple précédents.

II.4 L'algorithme naïf de recherche d'un chemin augmentant

Initialisation :

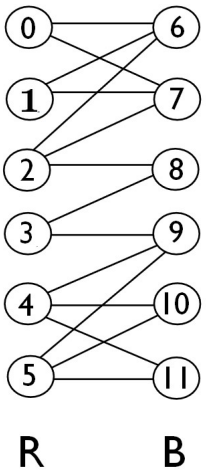
1. On oriente le graphe : les arcs vont désormais uniquement des sommets de R vers ceux de B.
2. On ajoute deux sommets s et t, avec un arc s vers chaque sommet de R et un arc de t vers chaque sommet de B.

Boucle principale : tant que c'est possible,

1. On cherche un chemin $s \to x_1 \to y_1 \to x_2 \to \ldots \to y_n \to t$, où aucun arc ne peut être emprunté deux fois.
2. On inverse tous les arcs traversés.
3. On met à jour le graphe : x_1 et y_n ne sont plus libres, on efface les arcs $s \to x_1$ et $y_n \to t$.

Fin : lorsqu'on ne trouve plus de tel chemin, on déduit un couplage maximal.

On peut détailler les premières étapes de l'algorithme sur le graphe suivant (tester aussi notre fonction de coloration) :



Pour l'implémentation on peut procéder par étapes :

1. Écrire une fonction qui initialise le graphe augmenté (on pourra prendre n pour s et $n+1$ pour t).
2. Écrire une fonction qui trouve un chemin dans le graphe augmenté (si c'est possible).
3. Écrire une fonction qui met à jour le graphe orienté une fois un chemin augmentant trouvé.
4. Écrire une fonction qui réalise la boucle principale.
5. Et enfin, écrire uen fonction qui reconstitue le couplage (ou modifier la fonction qui réalise la boucle principale en y ajoutant la reconstitution du couplage).