Informatique en MP(2)I

Marc de Falco

Algorithmique avancée des graphes

On présente ici trois algorithmes qui complètent les notions vue sen première année :

- le calcul d'un arbre couvrant de poids minimal dans un graphe pondéré
- le calcul des composantes fortement connexes dans un graphe orienté
- une notion de couplages maximale qui est un cas particulier d'un problème plus général de flot maximal

I Arbre couvrant minimal

I.1 Présentation du problème

Description informelle : on a un ensemble de maisons reliées par es routes, on cherche à poser des cables le long des routes pour que toute paire de maison soit connectée. Quelle est la longueur minimale de cable à utiliser ?

On comprend assez vite qu'il faut que les cables forment un graphe connexe et que pour des questions de réduction de coût, on peut supposer que les graphes sont acycliques. On cherche donc un arbre qui couvre chaque maison dont la somme des poids des arrêtes, les longueurs des cables, est minimale.

Définition I.1 Soit $G = (S, A)$ un graphe non orienté connexe on dit que $T \subset A$ est un arbre couvrant de G lorsque :

- $\forall x \in S, \exists a \in T, x \in a$: chaque sommet appartient à au moins une arête dans T
- (S, T) est un arbre, c'est-à-dire que c'est un sous-graphe acyclique et connexe

On notera ici $\mathcal{T}(G)$ l'ensemble des arbres couvrants de G.

Exemple FIXME

Une manière naturelle d'obtenir un arbre couvrant est de faire un parcours quelconque ou de calculer les composantes connexes avec une structure union-find. Dans ce dernier cas, comme le graphe est connexe, on obtiendra directement un unique arbre dans la forêt qui est un arbre couvrant. C'est l'ordre de traitement des arêtes qui va aiguiller vers un arbre de $\mathcal{T}(G)$ ou un autre.

Définition I.2 Soit $G = (S, A, w)$ un graphe non orienté connexe et pondéré par $w : A \to \mathbb{R}$, on note $w(T) = \sum_{a \in T} w(T)$ le poids d'un arbre couvant de G.

Comme $\mathcal{T}(G)$ est fini, il existe, au moins, un arbre T_0 tel que

$$w(T_0) = \min_{T \in \mathcal{T}(G)} w(T)$$

On dit que T_0 est un **arbre couvrant de poids minimal.**

Remarque En anglais, on parle de *minimum spanning tree*.

I.2 Algorithme de Kruskal

Pour calculer un arbre couvrant de poids minimal, on va considérer le calcul des composantes connexes avec une structure union-find mais en traitant les arêtes dans l'ordre croissant de leurs poids : c'est l'algorithme de Kruksal.

Algorithme - KRUSKAL
- Entrées :
  Un graphe non orienté pondéré connexe $G = (S, A, w)$.
-
  - ⋆ Pour chaque $x \in S$
    - ∘ `makeset(x)`
  - ⋆ On trie A par ordre croissant de poids.
  - ⋆ Pour chaque $\{x, y\} \in A$ trié
    - ∘ Si `find(x)` $\neq$ `find(y)`
    - ∘ Alors `union(x,y)`
  - ⋆ On renvoie l'unique arbre de la forêt.

I.3 Correction de l'algorithme de Krusal

On va montrer la correction d'une famille d'algorithme à laquelle Kruskal appartient.

Définition I.3 On dit que T est une forêt minimale de G s'il existe T' arbre couvrant de poids minimal de G tel que $T \subset T'$.

Remarque $\emptyset$ est ainsi une forêt minimale.

Définition I.4 Soit T une forêt minimale et $a \in A$. On dit que a est une arête **sûre** si $T \cup \{a\}$ est encore une forêt minmale.

Lemme I.1 Soit T une forêt minimale qui n'est pas un arbre couvrant, il existe une arête sûre a telle que $T \cup \{a\}$ soit encore une forêt minimale.

■ **Preuve**

Comme T est une forêt minimale, il existe T' arbre couvrant de poids minimal tel que $T \subset T'$. Comme T lui-même n'est pas un arbre couvrant, il existe $a \in T' \backslash T$. Ona alors $T \cup \{a\} \subset T'$ donc a est une arête sûre.

■

On en déduit un proto-algorithme de calcul d'un arbre couvrant de poids minimal :

Algorithme - ARBREMINQUELCONQUE
- Entrées :
  Un graphe non orienté pondéré connexe $G = (S, A, w)$.
-
  - ⋆ On pose $T = \emptyset$
  - ⋆ Tant que T n'est pas un arbre-couvrant
    - ∘ Déterminer une arête sûre $a \in E$
    - ∘ $T := T \cup \{a\}$
  - ⋆ Renvoyer T

Théorème I.2 Cet algorithme renvoie un arbre couvrant de poids minimal.

■ **Preuve**

Cet algorithme vérifie directement l'invariant suivant : T est une forêt minimale. En effet, le choix d'une arête sûre permet de prolonger l'invariant.

Cet algorithme termine car il n'existe qu'un nombre fini d'arêtes à ajouter et comme une arête sûre ne peut pas l'être une fois qu'on l'a rajoutée, on ne peut pas faire plus d'itérations que le nombre d'arêtes.

L'algorithme renvoie donc un arbre couvrant de poids minimal.

■

Remarque Il s'agit d'un proto-algorithme car la partie critique est de déterminer une arête sûre et c'est la partie qui n'est pas explicitée pour le moment.

Théorème I.3 Si F est une forêt minimale non couvrante et e l'arête de plus petit poids reliant deux arbres de F, alors e est sûre pour F.

■ **Preuve**

Comme F est une forêt minimale, il existe T arbre couvrant de poids minimal tel que $F \subset T$.

Soit $e \in T$, auquel cas $F \cup \{e\} \subset T$ est encore iune forêt minimale. Soit $e \notin T$ et comme T est un arbre, $T \cup \{e\}$ possède un cycle. On sait que $F \cup \{e\}$ est encore une forêt, donc ce cycle contient nécessairement une arête $e' \in T \backslash F$.

Par minimalité, $w(e') \geq w(e)$. Si on pose $T' = (T \backslash \{e'\}) \cup \{e\}$ alors T' est un arbre couvrant car comme l'ajout de e à T induit un cycle contenant e', les deux sommets de e' sont couverts par des arêtes dans ce cycle prive de e'. On en déduit de même que T' est connexe.

T' est nécessairement acyclique car on a cassé le seul cycle contenu dans $T \cup \{e\}$ en enlevant e'.

Reste que $w(T') = w(T) - w(e') + w(e) \leq w(T)$ donc T' est un arbre couvrant de poids minimal.

Ainsi $F \cup \{e\} \subset T'$ est une forêt minimale et e est sûre.

■

Comme $\emptyset$ est une forêt minimale, on vient de valider l'invariant pour Kruskal qui est que la forêt disjointe est une forêt minimale.

Corollaire I.4 Kruskal renvoie un arbre couvrant de poids minimal.

I.4 Complexité de l'algorithme de Kruskal

On peut décomposer l'algorithme :

- La création avec `makeset` est en $O(|S|)$
- Le tri des arêtes est en $O(|A| \log |A|)$.
- La boucle est en $O(|A|\alpha(|S|))$ avec $\alpha(|S|) = o(\log |A|)$

Comme G est connexe, on a $|A| \geq |S| - 1$ et ainsi $|S| = O(|A|)$. Ainsi la complexité global est en $O(|A| \log |A|)$.

Remarque Kruskal fait partie de ces algorithmes qui sont linéaires après avoir fait un tri. C'est un cas que l'on a déjà vu avec les algorithmes gloutons. D'ailleurs, on peut dire que Kruskal est un choix glouton d'arête sûre.

I.5 Prim ? TODO

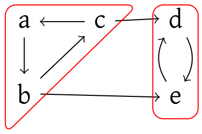
II Kosaraju et composantes fortement connexes

II.1 Rappel des définitions

Soit $G = (S, A)$ un graphe **orienté** on note, pour $x, y \in S$, $x \rightsquigarrow y$ quand il existe un chemin dans G reliant x à y. On dit que y est *accessible* depuis x.

On note $x \leftrightarrow y \iff x \rightsquigarrow y \wedge y \rightsquigarrow x$. C'est la restriction symétrique de $\rightsquigarrow$. Comme $\rightsquigarrow$ est réflexive et transitive, alors $\leftrightarrow$ est une relation d'équivalence. Les classes d'équivalences pour $\leftrightarrow$ sont appelées les **composantes fortement connexes** de G. On les note $CFC(G) = S/\leftrightarrow$.

Par exemple, sur le graphe :



On a les deux composantes fortement connexes $\{a, b, c\}$ et $\{d, e\}$.

On remarque une différence fondamentale avec les composantes connexes, c'est qu'il peut il y avoir des arrêtes entre deux composantes fortement connexes.

On cherche ici à déterminer un algorithme pour calculer les composanges fortement connexes d'un graphe.

II.2 Exemple

TODO voir cours

II.3 Rappels sur le parcours en profondeur

Comme on vient de le voir, le parcours en profondeur et ses temps d'entrée et de sorties sont très importants ici. On va donc faire des rappels sur ces notions.

On considère le programme suivant :

OCaml

```
type statut = Inconnu | EnTraitement | Traite

type etat_dfs = {
    statut : statut array;
    mutable clock : int;
    entree : int array;
    sortie : int array
}

let tick etat =
    let t = etat.clock in
    etat.clock <- t+1;
    t

let rec dfs ladj etat x =
    if etat.statut.(x) <> Traite
    then begin
        etat.statut.(x) <- EnTraitement;
        etat.entree.(x) <- tick etat;
        List.iter (fun y ->
            if etat.statut.(y) = Inconnu
            then dfs ladj etat y) ladj.(x);
        etat.sortie.(x) <- tick etat;
        etat.statut.(x) <- Traite
    end

let initialise_dfs ladj =
    let n = Array.length ladj in
    {
        statut = Array.make n Inconnu;
        clock = 1;
        entree = Array.make n 0;
        sortie = Array.make n 0
    }
```

Le temps d'entrée est le moment où commence à traiter un sommet et son temps de sortie est le moment où on a fini de le traiter car on a vu tous ses descendants. On notera ici $t_e(x)$ le temps d'entrée de x et $t_s(x)$ son temps de sortie.

Définition II.1 On considère un DFS d'un graphe $G = (S, A)$ et $x \in S$ découvert par ce parcours.

Soit $y \in S$, on dit que y est accesible par un chemin inconnu depuis x s'il existe un chemin de x à y ne passant que par des sommets de statut `Inconnu` au moment où on lance le DFS en x. On note $x \rightsquigarrow_I y$.

Théorème II.1 $x \rightsquigarrow_I y$ si et seulement si l'appel à DFS depuis x va appeler DFS sur y avant de se résoudre.

■ **Preuve**

- $\Rightarrow$) : On va raisonner par récurrence sur la longueur du chemin inconnu.
  - ⋆ Initialisation : si le chemin inconnu est vide, c'est direct.
  - ⋆ Hérédité : si $x \rightsquigarrow_I z \rightarrow y$ avec y inconnu et l'hypothèse de récurrence valide pour $x \rightsquigarrow_I z$ alors au moment de l'appel au DFS sur z, on va forcément faire un appel au DFS sur y, voisin inconnu de z.
- $\Leftarrow$) : si on considère la chaîne des appels qui ont mené jusqu'à y, vu la condition sur le statut, ce sont nécessairement tous des sommets inconnus et ils forment un chemin, qui est donc un chemin inconnu.

■

Théorème II.2 Soient $x, y \in S$ tels que $t_e(x) < t_e(y)$.
Si $x \rightsquigarrow_I y$, alors $t_f(y) < t_f(x)$.
Sinon, $t_f(x) < t_e(y)$.

Autrement dit, soit $[t_e(y); t_f(y)] \subset [t_e(x); t_f(x)]$, soit $[t_e(y); t_f(y)] \cap [t_e(x); t_f(x)] = \emptyset$.

On dit que les temps sont bien parenthésés. En effet, si on considère le mot sur S avec les lettres $(_x$ et $)_x$ pour chaque sommet x et tel qu'on écrive la lettre $(_x$ quand on note le temps d'entrée et $)_x$ quand on note le temps de sortie, alors ce mot est bien parenthésé.

Exemple Sur l'exemple du graphe précédent (TODO ref précise) on pourrait avoir le mot $(_a(_b(_c(_d(_e)_e)_d)_c)_b)_a$.

■ **Preuve**

Comme $t_e(x) < t_e(y)$, c'est qu'on a commencé le parcours en x avant de le commencer en y. i $x \rightsquigarrow_I y$ alors par le théorème précédent, on appelle le DFS sur y depuis l'appel du DFS sur x, donc le premier terminera avant le second et ainsi $t_f(y) < t_f(x)$.

Sinon, il n'est pas possible de rencontrer y en résolvant le DFS de x, donc on aura forcément fini de traiter x avant de commencer le parcours en y. Donc $t_f(x) < t_e(y)$.

■

Définition II.2 Soit $C \in CFC(G)$, on note

$$t_e(C) = \min\{\, t_e(x) \mid x \in C \,\}$$

$$t_f(C) = \max\{\, t_f(x) \mid x \in C \,\}$$

On a alors une propriété de parenthésage des temps sur les composantes fortement connexes elles-mêmes.

Théorème II.3 Soient $C, C' \in CFC(G)$.
S'il existe $x \in C, y \in C'$ avec $x \rightarrow y$, alors $t_f(C') < t_f(C)$.

■ **Preuve**

On suppose qu'il existe $x \in C, y \in C'$ avec $x \rightarrow y$.

Premier cas, $t_e(C) < t_e(C')$. On considère $u \in C$ tel que $t_e(C) = t_e(u)$. On a alors forcément $u \rightsquigarrow_I v$ pour tout $v \in C \cup C'$, en passant par $x \rightarrow y$. Ainsi x finit son DFS après tous les sommets dans $C \cup C'$ donc $t_f(C) = t_f(x) > t_f(C')$.

Second cas, $t_e(C) > t_e(C')$ si $u \in C'$ tel que $t_e(u) = t_e(C')$ alors on a visité tous les sommets de C' depuis u, donc $t_f(u) = t_f(C')$ et on n'a rencontré aucun sommet de c car $x \rightarrow y$ implique qu'il ne peut exister une arête de C' vers C. On a bien $t_f(C) > t_f(C')$.

■

II.4 Graphe miroir

Définition II.3 Soit $G = (S, A)$ un graphe orienté, on appelle **graphe miroir** de G le graphe $G^R = (S, A^R)$ où

$$\forall x, y \in S, (x, y) \in A \iff (y, x) \in A^R$$

Cela revient à renverser toutes les flèches du graphe G.

Théorème II.4 $CFC(G) = CFC(G^R)$

■ **Preuve**

On remarque que la relation $x \leftrightarrow_G y \iff x \leftrightarrow_{G^R} y$. Les deux relations ont donc *a fortiori* les mêmes classes d'équivalence.

■

II.5 Algorithme de Kosaraju

Algorithme - KOSARAJU

- Entrées :
  Un graphe orienté $G = (S, A)$
- ⋆ On initialise l'état d'un DFS pour G.
  ⋆ Tant qu'il y a des sommets inconnus, on lance un DFS depuis un sommet inconnu.
  ⋆ On trie S par **ordre décroissant** de temps de sortie.
  ⋆ On initialise l'état d'un DFS pour G^R.
  ⋆ $Comp \leftarrow \emptyset$
  ⋆ Tant qu'il y a un sommet inconnu x
    - On lance un DFS dans G^R à partir de x en notant les nouveaux sommets traités dans la liste C.
    - $Comp \leftarrow Comp \cup \{C\}$

  ⋆ On renvoie $Comp$.

Théorème II.5 $Comp = CFC(G)$.

■ **Preuve**

Il suffit de montrer l'invariant $Comp \subset CFC(G)$ pour la dernière boucle. Au départ, comme $Comp = \emptyset$ il est trivialement vérifié et à la fin, comme on aura traité tous les sommets, on aura nécessairement $Comp = CFC(G)$.

Supposons donc qu'on a $Comp \subset CFC(G)$ et qu'on relance un parcours dans G^R à partir de x. On sait que la composante $\overline{x}$ contenant x est forcément dans les sommets que l'on va traiter : $\overline{x} \subset C$. Si, par l'absurde, il existe un sommet $y \in C \backslash \overline{x}$, alors y est dans une autre composante $\overline{y}$. On a traité y depuis x, donc $t_e(x) < t_e(y)$. Comme $x \rightsquigarrow y$, on a par le théorème précédent $t_f(y) < t_f(x)$. On a donc traité la composante $\overline{y}$ dans un parcours précédent et donc y est traité. Contradiction.

■

Remarque La complexité de cet algorithme est dominée par les deux itérations de DFS, on est donc en $O(|S| + |A|)$.

III Couplage maximal dans un graphe bipartite

III.1 Problème

Définition III.1 Soit $G = (S, A)$ un graphe non orienté, on appelle **couplage** de G une partie $C \subset A$ telle que $\forall e, e' \in C, e \cap e' = \emptyset$.

On dit qu'un couplage est **maximal** pour G quand il est de cardinal maximal.

Rappel :

Définition III.2 Soit $G = (S, A)$ un graphe non orienté, on dit que G est **bipartite** lorsqu'il existe $S_1, S_2 \subset S$ avec $S_1 \cup S_2 = S$ et $S_1 \cap S_2 = \emptyset$ et toutes les arêtes relient un sommet de S_1 et un sommet de S_2.

On se pose alors la question de déterminer un couplage maximal dans un graphe bipartite. C'est un problème classique d'appariement. On peut ainsi citer le cas où on a des élèves et des écoles. On met une arête entre un élève et une école quand les deux veulent de l'autre. Un couplage maximal est alors une manière de placer le maximum d'élèves dans une école.

III.2 Chemin augmentant

Définition III.3 Soit C un couplage d'un graphe et x un sommet. On dit que x est libre **vis-à-vis** de C si x n'appartient pas à une arête de C.

Définition III.4 Soit $C \subset A$ un couplage d'un graphe.

Un chemin de x à y composé des arêtes $(e_1, \ldots, e_{2n+1})$ est dit **augmentant** si les arêtes pairs $e_{2i} \in C$, les arêtes impaires $e_{2i+1} \notin C$ et x et y sont libres pour C.

Lemme III.1 Soit C, C' des couplages de $G = (S, A)$. On considère $G' = (S, C \Delta C')$.

Les composantes connexes de G' sont :

- soit des sommets isolés
- soit des cycles **de longueur paire** alternant entre arêtes de C et C'
- soit des chemins alternant entre C et C' ayant des extremités distinctes.

■ **Preuve**

Il suffit de remarquer que les sommets de G' sont de degré au plus 2 car ils sont de degré au plus 1 dans (S, C) et (S, C').

De plus, comme les arêtes d'un couplage ne peuvent avoir des extrémités en commun un chemin devra forcément alterner entre arêtes de C et de C'. Les cycles sont donc nécessairement de longueur paire.

■

Théorème III.2 (Bergé 1957) C n'a pas de chemin augmentant, ssi C est un couplage maximal.

■ **Preuve**

On va montrer la contraposée : C non maximal ssi C a un chemin augmentant.

$\Leftarrow$) Supposons que G dispose d'un chemin augmentant φ, on considère C' différence symétrique de C et des arêtes dans φ. Ainsi, C' contient les arêtes de φ qui ne sont pas dans C, comme φ commence et finit avec des arêtes qui ne sont pas dans C, C' a une arête de plus que C.

De plus, comme φ est élémentaire et qu'ils relient deux sommets libres, on a l'assurance que C' est un couplage. Ainsi C n'est pas maximal.

$\Rightarrow$) Supposons que C non maximal, il existe C' tel que $|C'| > |C|$ et si on considère $G' = (S, C \Delta C')$, il a une composante qui contient au moins une arête de plus dans C' que dans C. Ça ne peut donc être un cycle et c'est un chemin qui est par construction augmentant pour C.

■

Remarque Il s'agit ici d'un cas particulier du théorème de Bergé.

III.3 Déterminer un chemin augmentant dans un graphe bipartite

On considère ici un graphe bipartite avec $S = S_1 \cup S_2$.

Pour déterminer un chemin augmentant pour C, on considère une orientation des arêtes $\{x, y\}$ ainsi :

- $x \to y$ si $x \in S_2, y \in S_1$ et $(x, y) \in C$.
- $y \to x$ sinon

On rajoute également deux sommets :

- un sommet source noté s avec $s \to x$ pour tout sommet **libre** dans S_1
- un sommet but noté t avec $x \to t$ pour tout sommet **libre** dans S_2.

On remarque qu'un sommet non libre y de S_2 est nécessairement de degré 1 et avec une arête $y \to x$ où x non libre et $\{x, y\} \in C$.

S'il existe un chemin de $s \rightsquigarrow t$ dans ce graphe orienté, alors il est de la forme :

$$s \to x_1 \to y_1 \cdots \to y_n \to t$$

avec :

- x_1 libre dans S_1
- y_n libre dans S_2
- tous les autre x_i et y_j sont non libres (ok) et deux à deux distincts (pas facile là !)
- $\{x_i, y_i\} \notin C$
- $\{y_i, x_{i+1}\} \in C$

Le chemin est donc augmentant

IV Exercices

Exercice 1 On considère un chemin entre deux sommets x et y dans un graphe non orienté pondéré. La largeur de ce chemin est le plus petit poids des arêtes présentes dans ce chemin. Le chemin vide de x à x est de largeur $+\infty$.

La distance de goulot d'étranglement entre x et y est la largeur maximale d'un chemin de x à y. S'il n'en existe pas, cette distance est $-\infty$.

1. Prouver que l'arbre couvrant de poids **maximal** contient les chemins les plus larges entre toute paire de sommets.
2. Décrire un algorithme pour résoudre en temps $O(|S|+|A|)$ le problème suivant : étant donné un graphe non orienté pondéré $G = (S, A)$, $x, y \in S$ et $W \in \mathbb{R}$, est-ce que le distance de goulot d'étranglement entre x et y est inférieure ou égale à W.
3. On suppose que la distance de goulot d'étranglement entre x et y est B.
   1. Prouver que la suppression d'une arête de poids inférieur à B ne change pas cette distance.
   2. Prouver que la contraction d'une arête de poids plus grand que B ne change pas cette distance. La contraction d'une arête (u, v) revient à identifier u et v, si cette contraction crée des arêtes parallèles, on ne conservera que l'arête de plus grand poids.

■ **Preuve**

1. On considère un arbre couvrant maximal T et deux sommets x, y. Supposons par l'absurde que T ne contienne pas le chemin le plus large de x à y. On considère alors le chemin dans T entre x et y, son arête de plus petit poids est $e = \{a, b\}$. On considère $T' = T \backslash \{e\}$ qui n'est plus connexe. Le chemin le plus large entre x et y contient ainsi forcément une arête $e' \notin T'$ et on peut considérer $T' \cup \{e'\} = T''$ qui est un arbre couvrant avec $w(T'') = w(T) + w(e') - w(e) > w(T)$ car $w(e') \geq largeur > w(e)$. Contradiction.

■