Chapitre 2

Arbres et arbres couvrants de poids minimal

Les algorithmes de ce chapitre concernent *a priori* des graphes non orientés ; cependant, comme observé au chapitre précédent, on peut les appliquer à des graphes orientés, il suffit pour cela “d’oublier” l’orientation et de remplacer les arcs par des arêtes, etc.

2.1 Arbres

Rappelons qu’un graphe non orienté $G = (X, U)$ est dit *connexe* si deux sommets distincts quelconques peuvent être reliés par une chaîne, donc par une chaîne simple (c’est-à-dire n’utilisant pas deux fois la même arête).

Si deux sommets peuvent être reliés par deux chaînes simples différentes, le graphe contient nécessairement un cycle.

Définition 2.1 (Graphe acyclique) *Un graphe non orienté est dit* acyclique *s’il ne comporte aucun cycle.*

Dans un tel graphe, deux sommets distincts peuvent être reliés par au plus une chaîne simple.

Définition 2.2 (Arbre) *Un* arbre *est un graphe connexe acyclique.*

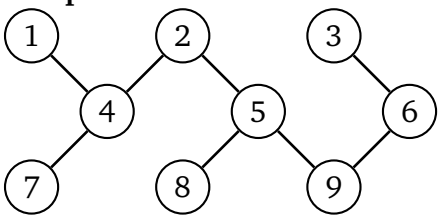
Propriété 2.1 (Propriétés caractéristiques) *Les propriétés suivantes de $G = (X, U)$ sont équivalentes et caractérisent un arbre (on note $n := |X|$ le nombre de sommets) :*

1. *G est connexe acyclique.*
2. *Deux sommets distincts de G sont reliés par une chaîne simple unique.*
3. *G est acyclique et a $(n-1)$ arêtes.*
4. *G est connexe et a $(n-1)$ arêtes.*

5. *G est acyclique et tout ajout d'une arête crée un cycle.*
6. *G est connexe et toute suppression d'une arête le rend non connexe.*

Remarque 7 *Le concept d'arbre est défini pour un graphe non orienté mais peut s'appliquer à un graphe orienté.*

Exemple 13

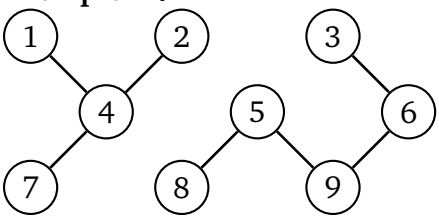


Si l'on ajoute l'arête (4,8) G n'est plus un arbre (cycle), si l'on supprime l'arête (5,9) non plus car 8 et 6 ne sont plus relié donc G n'est plus connexe.

Définition 2.3 (Forêt) *Un ensemble d'arbres est appelé un* forêt.

Tout graphe non orienté acyclique peut être vu comme une forêt : en effet, ses composantes connexes sont connexes et acycliques donc des arbres.

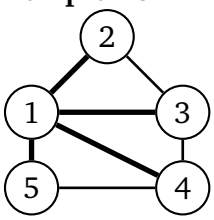
Exemple 14



2.2 Arbre partiel de poids minimum

Définition 2.4 (Arbre partiel) *Un* arbre partiel *(ou arbre couvrant) d'un graphe G (orienté ou non) est un graphe partiel de G connexe et sans cycle.*

Exemple 15



$G = (X, V)$ et $H = (X, \boldsymbol{V})$. H est bien un arbre connexe et sans cycle (connexe et 4 arêtes).

Théorème 2.1 *Un graphe admet un arbre partiel si et seulement s'il est connexe.*

Démonstration :
— Si G n'est pas connexe alors a fortiori aucun graphe partiel de *G* n'est connexe.

— Réciproque : Soit G un graphe connexe. Si quelle que soit l'arête qu'on enlève G n'est plus connexe alors G est un arbre (propriété 2.1 item 6) sinon on peut supprimer au moins une arête sans que G ne perde sa connexité. On enlève cette arête et l'on réitère le raisonnement, on arrive nécessairement à un graphe partiel connexe tel que quelle que soit l'arête qu'on enlève il ne l'est plus. Ce graphe partiel est donc un arbre.

□

Définition 2.5 (problème de l'arbre partiel de poids minimum)
*Soit $G = (X, U)$ un graphe connexe pondéré positivement par une fonction poids (ou coût) $p : U \to \mathbb{R}^+$. Le problème de l'*arbre partiel de poids minimum *ou a.p.m. consiste à trouver un graphe partiel $(X, U' \subseteq U)$ de G qui soit connexe et de poids $\sum_{u \in U'} p(u)$ minimum parmi tous les graphes partiels de G.*

Ce graphe partiel connexe de poids minimum est bien un arbre car il est par définition connexe et nécessairement *sans cycle* puisque sinon on pourrait diminuer son poids en supprimant une arête du cycle sans perdre la connexité du graphe.

Il existe deux algorithmes classiques pour la recherche de l'a.p.m. celui de PRIM 1957 [6] et celui de KRUSKAL 1956 [4] avec des complexités en $o(n^3)$ pour Prim et $o(m^2)$ pour Kruskal.

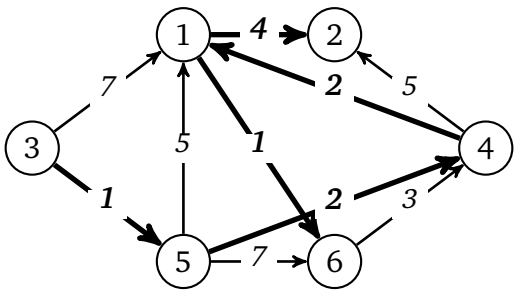
Définition 2.6 *Étant donné un ensemble non vide de sommets $A \subset X$, $A \neq \emptyset$ on désigne par $\omega(A)$ l'ensemble des arêtes reliant A à son complémentaire $X \setminus A$:*

$$\omega(A) := \{(x, y) \in U \mid x \in A \wedge y \notin A\}.$$

Si $\omega(A)$ est non vide il est appelé cocycle *de A.*

Algorithme de PRIM [6] :
— soit $G_0 = (X_0, \emptyset)$ où on place dans X_0 un seul sommet x_0 quelconque de X.
— étape courante : à partir de $G_k = (X_k, V_k)$ sélectionner l'arête $u = \{x, y\}$ du cocycle $\omega(X_k)$ de poids minimum, avec par exemple $x \in X_k$, $y \notin X_k$; on pose $x_k := x$ puis $G_{k+1} = (X_k \cup \{x_k\}, V_k \cup \{u\})$.
— fin de l'algorithme quand $k = n - 1$.

Exemple 16



L'application de Prim donne :

k	*arc selectionnés*	X_k	V_k	$P(G)$
0	*/*	$\{1\}$	$\emptyset$	*0*
1	*(1,6)*	$\{1,6\}$	$\{(1,6)\}$	*1*
2	*(4,1)*	$\{1,4,6\}$	$\{(1,6),(4,1)\}$	*3*
3	*(5,4)*	$\{1,4,5,6\}$	$\{(1,6),(4,1),(5,4)\}$	*5*
4	*(3,5)*	$\{1,3,4,5,6\}$	$\{(1,6),(4,1),(5,4),(3,5)\}$	*6*
5	*(1,2)*	$\{1,2,3,4,5,6\}$	$\{(1,6),(4,1),(5,4),(3,5),(1,2)\}$	*10*

Justification de l'algorithme : à chaque étape k :

1. $\omega(X_k) \neq \emptyset$ car $X_k \neq \emptyset$ et $X_k \neq X$ (tant que $k \neq n-1$) donc le choix d'une arête (x,y) du cocycle $\omega(X_k)$ peut s'effectuer
2. le graphe $G_k = (X_k, V_k)$ est sans cycle et connexe.

Algorithme de KRUSKAL [4] croissant : On classe préalablement les arêtes dans l'ordre de leurs poids croissants : $l(u_1) \leq l(u_2) \leq \ldots \leq l(u_m)$

— au départ $G_1 = (X, V_1)$, $V_1 = \{u_1\}$ où u_1 est l'arête de poids minimum dans G

— étape courante $G_{k+1} = (X, V_{k+1})$ obtenu à partir de G_k en ajoutant la première arête (dans l'ordre des poids) dont l'adjonction à V_k ne crée pas de cycle.

— fin de l'algorithme quand $|V_k| = n-1$

Justification : tant que $|V_k| < n-1$, le graphe (X_k, V_k) n'est pas connexe (moins de n-1 arêtes) et il existe parmi les arêtes non encore examinées une arête dont les deux extrémités appartiennent à deux composantes connexes différentes de (X_k, V_k).

Exemple 17 *L'application de Kruskal se base sur le classement suivant :* $(1,6)$*,* $(3,5)$*,* $(4,1)$*,* $(5,4)$*,* $(6,4)$*,* $(1,2)$*,* $(4,2)$*,* $(5,1)$*,* $(3,1)$*,* $(5,6)$*. Et donne le même résultat que Prim sur cet exemple.*

2.3 Kruskal : implémentation à l'aide de Union-Find

procedure Kruskal(G=(X,U,p) avec |X|=n et |U|=m)
  L = sort(U,p) ; ▷ complexité moyenne du tri : $O(m \log m)$
  V = $\emptyset$;
  while (|V| != n-1) {
    (x,y)=tete(L) ;
    If (X,$V \cup \{(x,y)\}$) ne contient pas de cycle
      V=V$\cup$ {(x,y)} ;
    L=L-{(x,y) ;
  }

Supposons que le test de création de cycle s'exécute en $f(n)$, vu que le corps de la boucle s'exécute $O(m)$ fois (dans le pire des cas on devra regarder tous les arcs pour en trouver $n-1$

sans cycle), on obtient une complexité globale en $O(m.f(n))$. Il est donc important que ce test soit réalisé de manière économique.

En fait, au fur et à mesure de l'exécution, on créé des CC (composantes connexes) (initialement une par sommet), dès lors que les extrémités de l'arête qu'on ajoute sont dans deux CC distinctes, on est sûr de ne pas créer de cycle, en revanche une fois cette arête ajoutée, il faut fusionner ces deux CC.
Le corps de la boucle devient alors :

```
while (|V| != n-1){
   (x,y)=tete(L) ;
   If (CC(x) != CC(y))
      {V=V∪{(x,y)} ; fusionner CC(x) et CC(y) ; }
   L=L-{(x,y)} ;
}
```

Les problème sont maintenant 1) de déterminer $CC(x)$ et 2) de fusionner $CC(x)$ et $CC(y)$ rapidement. La structure de données permettant de réaliser cela est appelé *Union-Find* (Find trouve la CC et Union fusionne les deux CC).

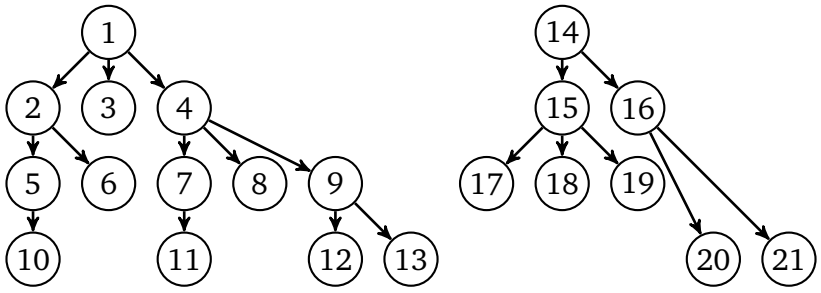
L'idée générale est de mémoriser chaque CC sous forme d'**arborescence**, une par CC.

Définition 2.7 (arborescence) *Une arborescence est un arbre avec une racine. Une racine est un sommet tel qu'il existe un chemin de ce sommet vers tous les autres sommets.*

Remarque 8 *Une arborescence est donc un graphe orienté.*

Un ensemble d'arbres est appelé *forêt*.

Notre forêt d'arborescences est représentée en mémoire sous forme d'un tableau, ainsi la forêt ci-dessous :



est mémorisée par :

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
p[x]	1	1	1	1	2	2	4	4	4	5	7	9	9	14	14	14	15	15	15	18	18

(par convention, les sommets racines ont eux-mêmes pour père).

Pour calculer CC(12), on procède ainsi CC(12) ⇒ CC(p[12])=CC(9) ⇒ CC(p[9])=CC(4) ⇒ CC(p[4])=1 ⇒ CC(p[1])=1

```
fonction Find(x)
  while (x != p[x])
    x=p[x];
  return x;
```

Définition 2.8 (hauteur) *La hauteur (ou profondeur) d'un sommet dans une arborescence est la longueur du chemin de la racine à ce sommet, et 0 pour la racine. La hauteur d'une arborescence est la longueur du plus long chemin de la racine vers un sommet.*

La complexité de Find(x) est linéaire par rapport à la hauteur de l'arborescence. La fusion des deux arborescences contenant respectivement les sommets x et y (qui ne sont pas nécessairement des racines) se faisant en ajoutant un lien de l'une des racines vers l'autre :

```
procedure Union(r,s)
  p[s]=r;
```

La complexité de Union est de l'ordre de 1 (coût d'une affectation).

Le problème qui subsiste étant d'éviter de se retrouver avec des arbres très déséquilibrés... car on peut, au pire, se retrouver avec des arbres en forme de listes. Pour éviter cela il est nécessaire de raffiner Union en UnionPondérée : mettre en racine de la fusion la racine de l'arbre de plus grande taille. On se dote d'un tableau taille[ ], qui sera initialisé à 1 pour chaque sommet, et qui contiendra le nombre de sommets de la composante. Le code de la fusion devient :

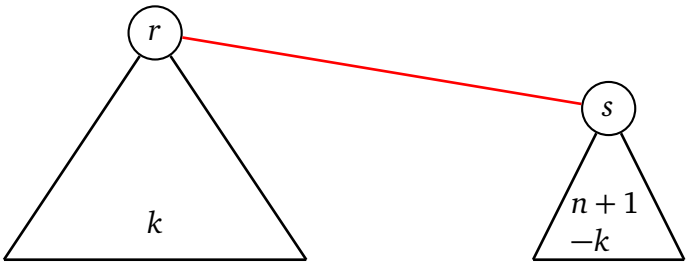
```
procedure UnionPonderée(r,s)
  if (taille[r] ≥ taille[s])          ▷ ça sera donc r la nouvelle racine
    {p[s]=r; taille[r]=taille[r]+taille(s);}
    else {p[r]=s; taille[s]=taille[s]+taille[r];}   ▷ là ce sera s la racine
```

Propriété 2.2 *La hauteur d'un arbre créé par UnionPonderée contenant n sommets ($n \geq 2$) est* $\leq 1 + \lfloor \log_2 n \rfloor$.

Démonstration : On le montre par récurrence sur n :

— C'est vrai pour $n = 2$ (un arbre créé par UnionPondérée contient au minimum 2 sommets), si $n = 2$ on a r et s de taille 1, après l'union, s devient le père de r la taille de s devient 2, sa hauteur est $1 \leq 1 + \lfloor \log_2 2 \rfloor$.

— Hypothèse de récurrence : l'abre créé par UnionPondérée de taille n est de hauteur $\leq 1 + \lfloor \log_2 n \rfloor$.
Soit un arbre de taille $n+1$ issu de l'union pondérée de r et de s, supposons que r est le plus grand (il contient k sommets) et s le plus petit (il contient $n+1-k$ sommets, et $n+1-k \leq (n+1)/2$). Alors après union pondérée et par hypothèse de récurrence, on obtient un arbre de hauteur égale au max de la hauteur de r et de la hauteur de s +1 (puisqu'on ajoute r à s).



Ce qui donne une hauteur égale à $\max(\lfloor 1+\log_2 k\rfloor, 2+\lfloor\log_2(n+1-k)\rfloor)$, or d'une part $\lfloor\log_2 k\rfloor \leq \lfloor\log_2(n+1)\rfloor$ (car log est croissante) et d'autre part $\lfloor\log_2(n+1-k)\rfloor \leq \lfloor\log_2(n+1)/2\rfloor = \lfloor(\log_2 n+1)-1\rfloor$. D'où la borne sur la hauteur de l'arbre final $1+\lfloor\log_2(n+1)\rfloor$.

— conclusion la formule est vraie pour tout $n \geq 2$

□

Dès lors, Find(x) est en $O(\log n)$ et UnionPondérée(r,s) en $O(1)$. La boucle interne de Kruskal est alors en $O(m.\log n)$, ce qui donne un coût de $O(m.(\log m + \log n))$ pour l'algorithme complet, et vu que $m \geq n-1$ (car le graphe est connexe), cela revient à $O(m.\log m)$.

On peut encore améliorer le tout (sans toutefois changer la complexité globale) grâce à la *compression des chemins* qui consiste lors du calcul de CC(x) à en profiter pour *aplatir* l'arborescence en mettant x et ses aïeux directement sous la racine. Ainsi *plus on va consulter CC(x), plus la hauteur des arborescences va diminuer*, et, asymptotiquement, le coût de CC(x) va devenir quasiment constant :

```
fonction FindAvecCompression(x) ;
    r = Find(x) ;
    while (x != r){
       y=p[x] ;
       p[x]=r ;
       x=y ;
    }
    return r ;
```

Plus précisément, l'auteur de la structure de données Union-Find, Tarjan[7], a démontré qu'une suite de n UnionPondérée et de m FindAvecCompression a un coût amorti [1] de $O(n+\log^*(n))$. Où $\log^*$ est le logarithme itéré : $\log^*(n)$ est le nombre de puissance de 2 ($2^{2^{2^{\cdots^2}}}$) qu'il faut empiler pour atteindre n, or pour $n < 2^{65536}$, le logarithme itéré vaut au plus 5 ! On peut donc considérer qu'une suite de n UnionPondérée et de m FindAvecCompression a un coût moyen de $O(n)$.

L'algorithme de Kruskal devient :

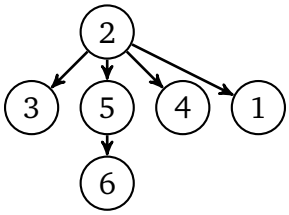
1. L'analyse amortie consiste à borner le temps d'exécution moyen d'un algorithme lorsqu'il est répété plusieurs fois de suite.

```
procedure Kruskal(G=(X,U,p) avec |X|=n et |U|=m)
   L = sort(U,p) ;
   V = ∅;
   while (|V| != n-1) {
      (x,y)=tete(L) ;
      r=FindAvecCompression(x) ;
      s=FindAvecCompression(y) ;
      If (r != s) {
         UnionPondérée(r,s) ;
         V=V∪ {(x,y)} ;
        }
      L=L-{(x,y)} ;
   }
```

Il nécessite, en plus du tri, $O(m)$ FindAvecCompression et $O(n)$ UnionPondérée, et a donc un coût amorti de $0((m.\log(m)) + O(n))$.
Puisque le graphe de départ est connexe, on a $m \geq n-1$, et donc le coût de l'algorithme est dominé par celui du tri initial. La complexité de l'algorithme de Kruskal est donc au final en $O(m.\log m)$.

Remarque 9 *L'arborescence créée n'a pas de rapport avec l'a.c.p.m. ni avec le graphe initial (elle peut comporter des arcs qui ne sont pas dans le graphe initial : un arc de l'arborescence représente simplement l'existence d'une chaîne dans le graphe initial (qui est supposé connexe donc ça n'apporte rien)).*

Exemple 18 *Sur l'exo II feuille 3, Kruskal-UnionPondérée-Find donne l'arborescence suivante :*



On peut détailler le déroulement de l'algo dans le tableau suivant qui contient les pères des sommets (avec en indice les tailles des CC des sommets au moment de l'ajout de l'arc) :

arcs selectionnés\ sommets	*1*	*2*	*3*	*4*	*5*	*6*
init	*1*	*2*	*3*	*4*	*5*	*6*
(5,6)	*1*	*2*	*3*	*4*	6_2	6_2
(2,3)	*1*	2_2	2_2	*4*	5_2	5_2
(3,5)	*1*	2_4	2_4	*4*	2_4	5_2
(4,5)	*1*	2_5	2_5	2_5	2_5	5_2
(1,6)	2_6	2_6	2_6	2_6	2_6	5_2

Ici avec FindCompression le 6 deviendrait fils de 2.