Arbre couvrant minimal
(Minimal spanning tree)

Option informatique
Lycée Louis-le-Grand, Paris

mars 1998

Résumé

Une compagnie de télécommunications est confrontée au problème suivant : étant donné un certain nombre de nœuds (les autocommutateurs) répartis de façon dispersée sur un territoire donné, il s'agit de les relier par un réseau de coût minimal. Il s'agit donc de dessiner un arbre (*id est* un graphe connexe sans cycle) qui passe par tous les sommets d'un graphe de distance (qui est donc un graphe complet) et dont la somme des longueurs des arêtes soit minimal.

Nous présentons ici les deux algorithmes classiques de recherche d'un arbre couvrant minimal d'un graphe connexe valué : les algorithmes de Prim et de Kruskal. Nous avons tenté de donner une présentation synthétique des algorithmes qui fasse ressortir les concepts qu'ils partagent.

Bien entendu, nous décrivons des programmes Caml pour une implémentation effective.

Table des matières

1 Considérations générales

1.1 Vocabulaire

Un **graphe** G est un couple (S, A) où S est un ensemble fini de **sommets**, et A est une partie de $\mathcal{P}_2(S)$, ensemble des parties à deux éléments de S. Une **arête** d'extrémités a et b est une paire $\{a, b\} \in A$. Un sous-graphe est un couple (S', A') où $S' \subset S$ et $A' \subset A \cap \mathcal{P}_2(S')$.
Un graphe **valué** est un graphe (S, A) muni d'une **fonction de coût** $c : A \to \mathbb{R}$.
Une **chaîne** est une suite finie $a_0, a_1, \ldots, a_p$ de sommets du graphe tels que pour $1 \leqslant i \leqslant p$ l'arête d'extrémité a_{i-1} et a_i fasse partie du graphe. Cette chaîne est dite **élémentaire** si les a_i sont deux à deux distincts. Sa longueur est alors égale à p.
Un graphe est dit **connexe** si deux sommets quelconques sont reliés par une chaîne.
Un **cycle** est une chaîne a_0, a_1, …, a_p telle que $a_0 = a_p$. Un tel cycle est **élementaire** si $a_i \neq a_j$ pour $0 \leqslant i < j \leqslant p - 1$.
Un **arbre** est un graphe connexe sans cycle.
Un sous-graphe (S', A') d'un graphe $G = (S, A)$ est dit **couvrant** si $S = S'$.
Le coût d'un sous-graphe (S', A') d'un graphe valué G est égal à $\sum_{\alpha \in A'} c(\alpha)$, c étant la fonction de coût du graphe.
On se pose le problème de déterminer un arbre couvrant de coût minimal d'un graphe valué connexe donné.
La figure 1 représente un exemple de graphe valué connexe que nous utiliserons dans la suite pour illustrer les algorithmes présentés.

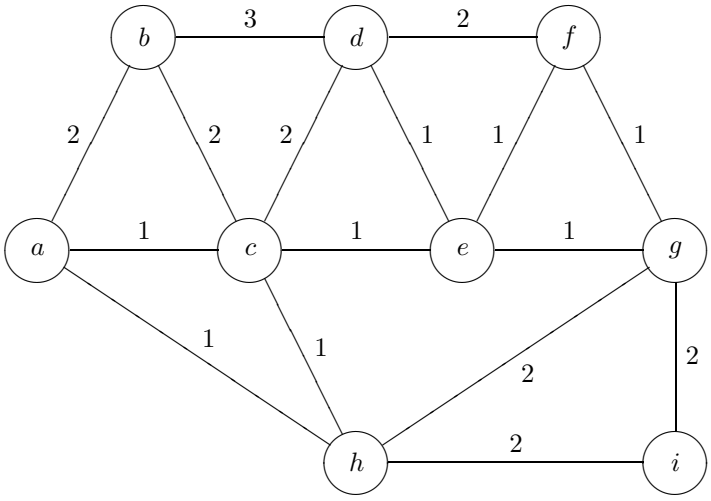


FIG. 1: un exemple de graphe valué

1.2 Cocycles et bordures

Soit $G = (S, A)$ un graphe et $X \subset S$ une partie de l'ensemble des sommets.
On appelle **cocycle** de X, et on note $\omega(X) = \{\{x, y\} \in A, x \in X, y \notin X\}$. Il s'agit de l'ensemble des arêtes qui relie un sommet de X à un sommet qui n'est pas dans X. On a $\omega(S) = \omega(\emptyset) = \emptyset$.
On appelle **bordure** de X, et on note $\beta(X) = \{y \in S \setminus X, \exists x \in X, \{x, y\} \in A\}$. Il s'agit de l'ensemble des sommets qui ne sont pas dans X mais **adjacents** à un sommet de X : on dit qu'un sommet y est adjacent à un sommet x (*resp.* à un ensemble X de sommets) si l'arête $\{x, y\}$ est dans A (*resp.* s'il existe $x \in X$ tel que l'arête $\{x, y\}$ est dans A). On dispose de $\beta(S) = \beta(\emptyset) = \emptyset$.

1.3 Partition approximante

Une partition[1] (O, R, L) de A est dite **approximante** s'il existe un arbre couvrant de coût minimal qui contient les arêtes de O mais aucune arête de R. Nous dirons que les arêtes de O sont *obligées*, celles de R *refusées*, et celles de L *libres*.
Notons que $(\emptyset, \emptyset, A)$ est une partition approximante de départ tout à fait convenable, qui nous servira de point de départ dans la suite. Notons enfin que résoudre notre problème de recherche d'un arbre couvrant minimal revient à déterminer une partition approximante de la forme (O, R, L) avec $L = \emptyset$.

1.4 Deux résultats fondamentaux

Nous allons montrer deux théorèmes qui permettent de passer d'une partition approximante (O, R, L) à une autre (O', R', L') vérifiant $O \subset O'$, $K \subset K'$ et $L \supsetneq L'$.

1.4.1 Utilisation de cycles candidats

Montrons le

Théorème 1
Soit (O, R, L) une partition approximante.
On suppose qu'il existe un cycle élémentaire γ ne contenant pas d'arête de R.
Soit alors α une arête de $\gamma \cap L$ de coût maximal.
$(O, R \cup \{\alpha\}, L \setminus \{\alpha\})$ est encore une partition approximante.

✧ Notons (S, T) un arbre couvrant minimal associé à notre partition approximante initiale, c'est-à-dire tel que $O \subset T$ et $T \cap R = \emptyset$.
Notons que γ contient au moins une arête libre (c'est-à-dire une arête de L). En effet, dans le cas contraire, les arêtes de γ seraient toutes dans O et donc dans T, et notre arbre contiendrait un cycle, ce qui est contradictoire avec la définition d'un arbre.
De deux choses l'une maintenant : ou bien notre arbre (S, T) ne contenait pas l'arête α choisie, et dans ce cas il est encore associé à $(O, R \cup \{\alpha\}, L \setminus \{\alpha\})$ et on a terminé.
Ou bien $\alpha \in T$. Mais alors $(S, T \setminus \{\alpha\})$ contient deux composantes connexes, qui sont des arbres disjoints. Mais parmi les arêtes de γ distinctes de α il existera une arête β (qui ne sera heureusement pas dans R) et qui reliera ces deux composantes connexes. Alors $(S, (T \cup \{\beta\}) \setminus \{\alpha\})$ est un arbre couvrant de coût plus petit ou égal que le coût de (S, T) qui était déjà minimal, donc lui aussi minimal. Et, bien sûr, $(S, T \cup \{\beta\} \setminus \{\alpha\})$ est lui associé à la nouvelle partition $(O, R \cup \{\alpha\}, L \setminus \{\alpha\})$. ✦

1.4.2 Utilisation de cocycles candidats

Montrons le

Théorème 2
Soit (O, R, L) une partition approximante.
On suppose qu'il existe un cocyle $\omega(X)$ d'une partie X de S ne contenant pas d'arête de O.
Soit alors α une arête de $\omega(X) \cap L$ de coût minimal.
$(O \cup \{\alpha\}, R, L \setminus \{\alpha\})$ est encore une partition approximante.

✧ Notons (S, T) un arbre couvrant minimal associé à notre partition approximante initiale, c'est-à-dire tel que $O \subset T$ et $T \cap R = \emptyset$.
Notons que $\omega(X)$ contient au moins une arête libre (c'est-à-dire une arête de L) : en effet, (S, T) étant couvrant, T rencontre $\omega(X)$. On conlut en rappelant que $\omega(X) \cap O = \emptyset$ et $T \cap R = \emptyset$.
Si $\alpha \in T$, l'arbre couvrant minimal (S, T) est encore associé à notre nouvelle partition approximante $(O \cup \{\alpha\}, R, L \setminus \{\alpha\})$ et on a terminé.

[1] O, R et L sont deux à deux disjoints et leur réunion vaut A tout entier.

Sinon, notons $\alpha = \{x, y\}$ avec $x \in X$ et $y \notin X$. Il existe dans (S, T) une unique chaîne élémentaire qui lie x à y. Elle devra quitter X à un moment ou à un autre : c'est dire que dans cette chaîne se trouve une arête (d'ailleurs unique) $\beta \in \omega(X)$. Soit alors $(S, (T \cup \{\alpha\}) \setminus \{\beta\})$ l'arbre obtenu en substituant α à l'arête β. Bien sûr, il est de coût moindre que (S, T), et il est toujours couvrant. $(O \cup \{\alpha\}, R, L \setminus \{\alpha\})$ est donc bien encore une partition approximante. ✥

1.5 Un algorithme générique

L'algorithme que l'on peut proposer est donc le suivant :

(i) on commence par la partition $(\emptyset, \emptyset, A)$;

(ii) tant qu'il existe un cycle ou un cocycle candidat, on remplace la partition approximante courante par celle obtenue en appliquant le théorème 1 ou le théorème 2.

Théorème 3
L'algorithme générique ainsi défini termine avec une partition courante $(O, R, \emptyset)$.
(S, O) est donc alors un arbre couvrant minimal.

✧ Il s'agit de prouver qu'à la fin $L = \emptyset$. Et en effet, supposons qu'il reste au moins une arête libre $\alpha \in L$, sans qu'on puisse pourtant appliquer aucun des théorèmes 1 et 2.
α ne peut relier deux composantes connexes disjointes du sous-graphe de G d'arêtes O : on pourrait sinon appliquer le théorème 2.
Mais alors c'est que α ferme un cycle constitué d'arêtes qui sont toutes dans $O \cup \{\alpha\}$: on pourrait appliquer le théorème 1, ce qui fournit la contradiction souhaitée. ✥

2 L'algorithme de Prim

2.1 Description de l'algorithme

Nous considérons un graphe connexe valué (S, A), comportant n sommets et p arêtes. Le principe de l'algorithme de Prim est d'appliquer $n-1$ fois le théorème 2 page 3, à des sous-ensembles emboîtés de sommets.
De façon informelle, l'algorithme de Prim peut s'écrire :

Initialisation On choisit au hasard un sommet $a \in S$, et on pose $P = \{a\}$ et $(O, R, L) = (\emptyset, \emptyset, A)$.

Boucle principale Pour i de 1 à $n-1$ faire ce qui suit :

(i) on choisit une arête $\alpha = \{x, y\} \in \omega(P)$ de coût minimal, avec $x \in P$ et $y \notin P$;

(ii) on itère avec $P \leftarrow P \cup \{y\}$ et $(O, R, L) \leftarrow (O \cup \{\alpha\}, R, L \setminus \{\alpha\})$.

Sur l'exemple du graphe de la figure 1 page 2, on obtient, en partant du sommet a, l'arbre couvrant minimal de la figure 2, et, en partant du sommet c, l'arbre couvrant minimal de la figure 3 page 7 (les arêtes de l'arbre obtenu sont en trait non pointillé). Notons toutefois qu'il n'y a pas unicité du résultat obtenu, même si l'on choisit le même sommet de départ. En revanche, bien entendu, les deux arbres ainsi dessinés ont le même coût, en l'occurrence 10.

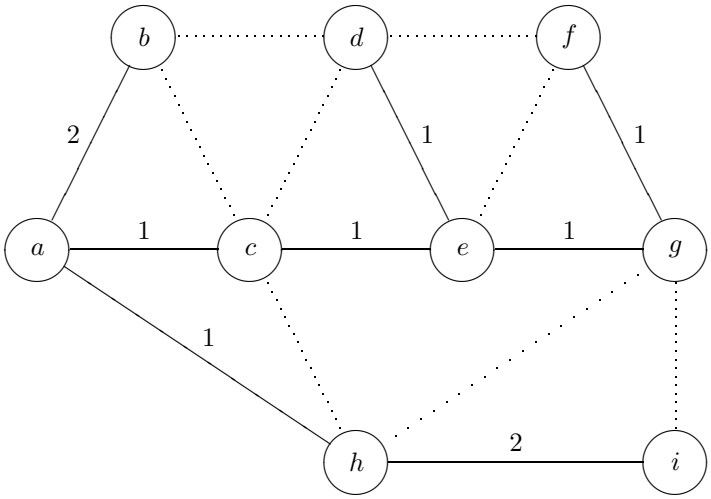


FIG. 2: un arbre calculé par l'algorithme de Prim, en partant du sommet a

2.2 Implémentation de l'algorithme de Prim

2.2.1 Représentation des graphes

Nous représenterons ici tout simplement un graphe valué par une matrice symétrique dont l'élément d'indices (i, j) représente le coût de l'arête qui lie les sommets numéros i et j. Il suffira de choisir un coût exorbitant pour les arêtes absentes.
Le programme 1 page suivante explicite notre représentation des graphes, et définit une fonction qui permet de calculer la matrice correspondante à une liste d'arêtes valuées.

2.2.2 Implémentation effective

Il ne reste plus qu'à écrire effectivement l'algorithme de Prim, ce dont est chargé le programme 2 page suivante.

Programme 1 gestion de base des graphes

```
let graph_of_edges_list n l =
    let g = make_matrix n n max_int
    in
    let rec ajoute_arêtes = function
        | (i,j,c) :: q -> g.(i).(j) <- c ; g.(j).(i) <- c ; ajoute_arêtes q
        | [] -> g
    in
    ajoute_arêtes l ;;

let g = graph_of_edges_list 9
[(0,1,2) ; (0,2,1) ; (0,7,1) ; (1,2,2) ; (1,3,3) ; (2,3,2) ; (2,4,1) ; (2,7,1) ;
 (3,4,1) ; (3,5,2) ; (4,5,1) ; (4,6,1) ; (5,6,1) ; (6,7,2) ; (6,8,2) ; (7,8,2)] ;;
```

Programme 2 algorithme de Prim

```
let rec intervalle i j =
    if i > j then []
    else i :: (intervalle (i+1) j) ;;

let prim g =
    let n = vect_length g
    in
    let vient_de = make_vect n 0
    and coût = init_vect n (function i -> g.(i).(0))
    in
    let rec trouve_mini c y = function
        | [] -> (c,y)
        | t :: q
         -> let c' = coût.(t)
            in
            if c' < c then trouve_mini c' t q
                        else trouve_mini c  y q
    in
    let mise_à_jour y =
        for i = 0 to n-1 do
            if coût.(i) > g.(i).(y) then
            begin
                coût.(i) <- g.(i).(y) ;
                vient_de.(i) <- y
            end
        done
    in
    let rec choisit arêtes = function
        | [] -> arêtes
        | non_couverts
            ->  let (_,y) = trouve_mini max_int 0 non_couverts
                in
                mise_à_jour y ;
                choisit ((vient_de.(y),y) :: arêtes) (except y non_couverts)
    in
    choisit [] (intervalle 1 (n-1)) ;;
```

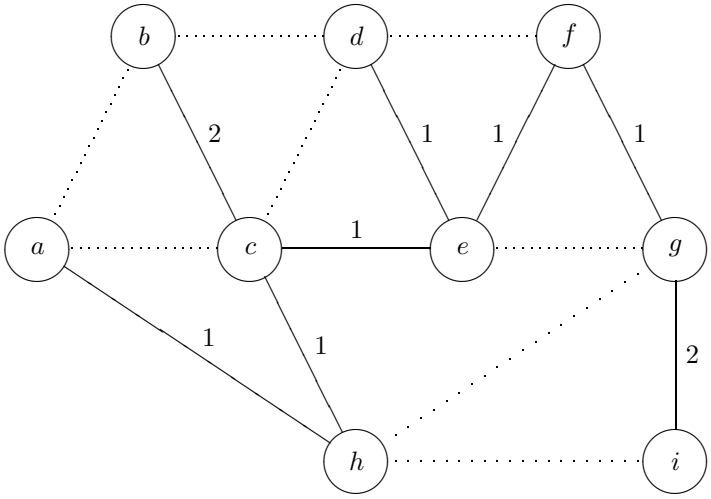


FIG. 3: un arbre calculé par l'algorithme de Prim, en partant du sommet c

On maintient tout le long de l'algorithme deux tableaux `vient_de` et `coût` qui, pour tout sommet $y \notin P$ contiennent respectivement l'élément $x \in P$ et le coût de l'arête (x, y), x étant choisi dans P pour réaliser justement un coût minimal parmi les arêtes de $\omega(P)$ qui arrivent sur y.
La fonction `trouve_mini`, définie en lignes 23–29, renvoie alors le sommet y qui réalise le coût minimal de $\omega(P)$.
La fonction `mise_à_jour`, définie en lignes 31–38, modifie alors éventuellement, pour tout élément $i \notin P$, les valeurs correspondantes des tableaux `vient_de` et `coût`, quand il devient plus avantageux, pour arriver au sommet i, de quitter $P \cup \{y\}$ avec une arête d'origine y.
Il n'y a plus qu'à itérer le processus jusqu'à ce que tous les sommets soient couverts par l'arbre construit petit à petit : on renvoie alors la liste des arêtes élues.

2.2.3 Évaluation

On vérifie très facilement que l'algorithme écrit tourne en un $O(n^2)$, où n représente le nombre de sommets du graphe considéré.

3 L'algorithme de Kruskal

3.1 Description de l'algorithme

Nous considérons toujours un graphe connexe valué (S, A), comportant n sommets et p arêtes. Le principe de l'algorithme de Kruskal est de trier les arêtes par ordre croissant de coût, puis de leur appliquer celui des deux théorèmes 1 ou 2 qui s'applique.
De façon informelle, l'algorithme de Kruskal peut s'écrire :

Initialisation On trie par ordre de coût croissant les arêtes, obtenant une liste $(\alpha_i)_{0 \leqslant i < p}$; on pose $(O, R, L) \leftarrow (\emptyset, \emptyset, A)$.

Boucle principale Pour i de 0 à $p-1$ faire ce qui suit :

(i) si α_i relie deux composantes connexes de O, itérer avec $(O, R, L) \leftarrow (O \cup \{\alpha_i\}, R, L \setminus \{\alpha_i\})$;

(ii) sinon, itérer avec $(O, R, L) \leftarrow (O, R \cup \{\alpha_i\}, L \setminus \{\alpha_i\})$.

Sur l'exemple du graphe de la figure 1 page 2, on obtient par exemple l'arbre couvrant minimal de la figure 4 (les arêtes de l'arbre obtenu sont en trait non pointillé). Notons toutefois qu'il n'y a pas unicité du résultat obtenu, car le tri donne des résultats différents si ne serait-ce que deux arêtes ont le même coût. En revanche, bien entendu, les arbres dessinés auront le même coût, en l'occurrence 10.

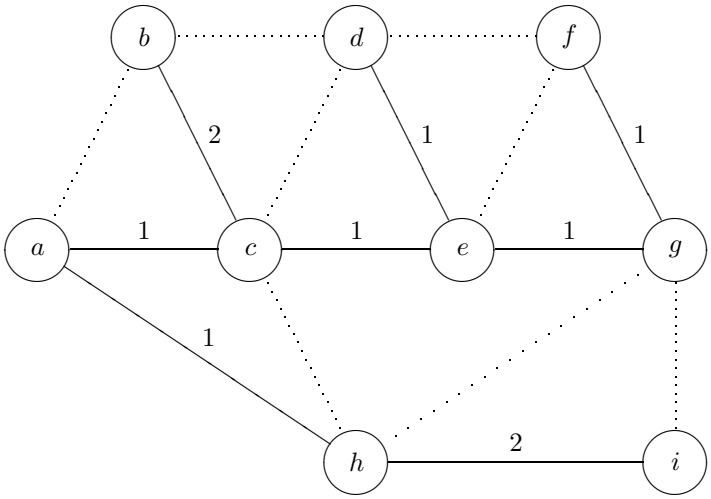


FIG. 4: un arbre calculé par l'algorithme de Kruskal

3.2 Une variante de l'algorithme de Kruskal

On peut également procéder de façon inverse :

Initialisation On trie par ordre de coût *décroissant* les arêtes, obtenant une liste $(\alpha_i)_{0 \leqslant i < p}$; on pose $(O, R, L) \leftarrow (\emptyset, \emptyset, A)$.

Boucle principale Pour i de 0 à $p-1$ faire ce qui suit :

(i) si $(S, (O \cup L) \setminus \{\alpha_i\})$ est encore connexe, itérer avec $(O, R, L) \leftarrow (O, R \cup \{\alpha_i\}, L \setminus \{\alpha_i\})$;

(ii) sinon, itérer avec $(O, R, L) \leftarrow (O \cup \{\alpha_i\}, R, L \setminus \{\alpha_i\})$.

Le test de connexité n'est pas aisé à réaliser, et c'est pourquoi dans la suite nous nous contenterons d'implémenter l'algorithme précédent.

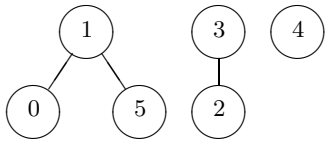
3.3 Kruskal et *union-find*

Tout le problème est bien sûr dans la gestion d'une table des composantes connexes relatives à l'ensemble O d'arêtes.
On utilise pour cela la structure de donéees habituelle pour la résolution du problème dit *union-find*. Il s'agit d'implémenter deux opérations sur une structure représentant dans son état initial une partition constituée de singletons.

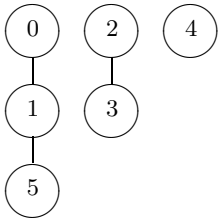
Question "find" dire si deux éléments sont dans la même classe d'équivalence ;

Question "union" regrouper en une seule classe les classes de deux éléments donnés.

Une partition est représentée par une forêt d'arbres n-aires. Par exemple la partition $\{0, 1, 5\}, \{2, 3\}, \{4\}$ pourra être représentée par l'une des forêts ci-dessous :



ou encore :



On représentera ce genre de forêt par un couple `(taille,père)`, où `taille` conservera le nombre de parties de la partition, c'est-à-dire le nombre d'arbres de la forêt, et `père.(x)` renverra `x` si il est la racine de son arbre, et la valeur de son père sinon.
La première forêt représentée ci-dessus sera donc représentée par le couple `(3,père)` où le tableau suivant décrit le vecteur `père` :

`x`	0	1	2	3	4	5
`père.(x)`	1	1	3	3	4	1

On cherche évidemment à limiter la profondeur des arbres de la forêt. Pour cela, on se propose de maintenir à jour un nouveau vecteur, `poids`, tel que `poids.(x)` contienne la taille du sous-arbre de racine `x`. Au départ ce vecteur est plein de 1.
Lors d'un regroupement, on fusionnera les deux arbres de telle sorte que l'on place à la racine du nouvel arbre la racine de poids le plus élevé des deux arbres.
On améliore encore l'efficacité de la structure en convenant d'utiliser le procédé dit de compression des chemins (ou *path compression*) : lors de la recherche de la classe d'un élément x, on remonte dans la structure jusqu'au père. On en profitera pour relier directement chaque nœud visité au père.
Supposons par exemple qu'on ait dans la forêt l'arbre de gauche de la figure 5 page 11. La recherche de l'élément 20 transforme le tableau `père` de telle sorte que le dessin de l'arbre est maintenant celui de droite.
On obtient alors l'implémentation en Caml du programme 3 page suivante.

3.4 Implémentation de l'algorithme de Kruskal

Nous nous donnerons cette fois un graphe par un couple constitué du nombre n de sommets (qui seront représentés par les entiers de 0 à $n-1$) et d'une liste de p arêtes, chacune d'elle étant constituée d'un triplet (x, y, c), x et y désignant les extrémités de l'arête et c son coût.
On obtient facilement le programme 4 page suivante.

Programme 3 gestion des partitions (problème *union-find*)

```
let crée_partition n = (n , init_vect n (function i -> i) , make_vect n 1) ;;

let find (nb_classes,père,poids) x =
    let rec cherche_père i = if père.(i) = i then i else cherche_père père.(i)
    in
    let racine = cherche_père x
    in
    let rec compression i =
        if i = racine then () else ( compression père.(i) ; père.(i) <- racine )
    in
    compression x ;
    racine ;;

let union ((nb_classes,père,poids) as partition) x y =
    let i = find partition x and j = find partition y
    in
    if i <> j then
    begin
        if poids.(i) < poids.(j)
            then ( poids.(j) <- poids.(i) + poids.(j) ; père.(j) <- i )
            else ( poids.(i) <- poids.(i) + poids.(j) ; père.(i) <- j ) ;
        (nb_classes - 1, père, poids)
    end
    else partition ;;
```

Programme 4 algorithme de Kruskal

```
let rec tri_fusion = function
    | [] -> []
    | [ a ] -> [ a ]
    | l ->  let (l1,l2) = découpe l
            in
            fusion (tri_fusion l1) (tri_fusion l2)
and découpe = function
    | [] -> [],[]
    | [ a ] -> [ a ],[]
    | a :: b :: q
        ->  let (l1,l2) = découpe q
            in
            (a :: l1) , (b :: l2)
and fusion l1 l2 = match l1,l2 with
    | [],_ -> l2
    | _,[] -> l1
    | ((x,y,c) as a)::q1, ((x',y',c') as b)::q2
        ->  if c < c' then a :: (fusion q1 l2)
            else b :: (fusion l1 q2) ;;

let kruskal nb_sommets arêtes =
    let la = tri_fusion arêtes
    in
    let partition = crée_partition nb_sommets
    in
    let rec itère partition la = match partition with
        | (1,_,_) -> []
        | (n,_,_)
            ->  let (x,y,c) = hd la
                in
                let ((n',_,_) as partition) = union partition x y
                in
                if n = n' then itère partition (tl la)
                else (x,y,c) :: (itère partition (tl la))
    in
    itère partition la ;;
```

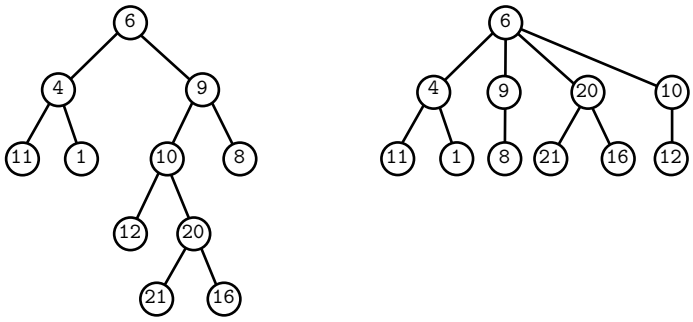


FIG. 5: le procédé de compression des chemins (on cherche la classe de 20)

3.5 Évaluation

On vérifie que le coût de l'algorithme de Kruskal est un $O(p \lg n)$.