Chapitre quatre

Arbres couvrants

MPI/MPI*, lycée Faidherbe

Résumé

Un exercice classique demande de montrer qu'un graphe non orienté connexe à n sommets admets au moins $n-1$ arêtes. Une démonstration consiste à partir des sommets sans arêtes, formé donc de n composantes connexes et de remarquer que l'ajout d'une arête peut, peut-être, joindre deux composantes connexes mais seulement 2, le nombre de composantes connexes diminue donc de 1 au plus pour chaque arête.

Les questions qu'on peut poser sont de savoir comment vérifier si une arête connecte deux composantes connexes distinctes et comment caractériser les composantes connexes que l'on construit.

Le type de donnée étudié dans ce chapitre permet de répondre à ces questions.

On l'utilise ensuite dans la recherche d'un arbre couvrant minimum.

I Structure `union-find`

I.1 Définition

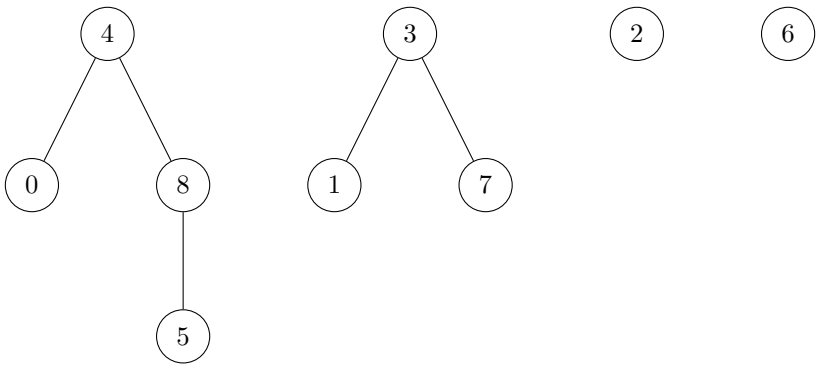
Si on veut définir les classes d'équivalence d'une relation, il y a deux procédés opposés :

- on peut découper, une par une, les classes d'équivalence, ; c'est ce que fait l'algorithme vu lors du parcours,
- on peut rassembler des morceaux d'une même classe jusqu'à l'obtention de classes entières.

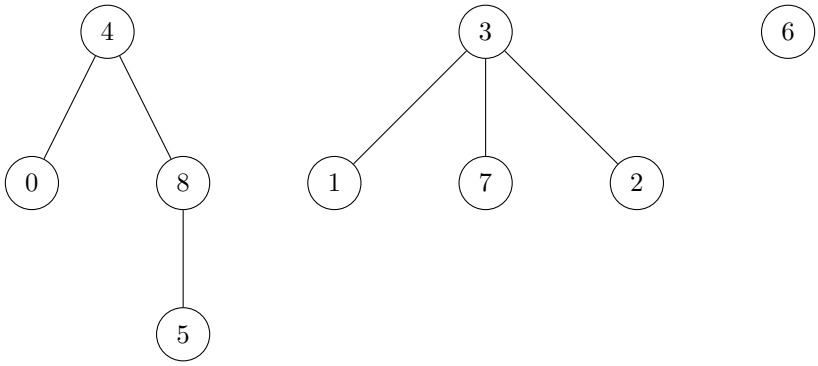
On s'intéresse ici à la seconde méthode.
On veut donc gérer des partitions d'un ensemble de plus en plus grossières. Pour cela on veut

- connaître à quelle classe d'équivalence appartient un élément d'un ensemble, `find`,
- assembler deux classes d"équivalence en une seule, `union`.

On propose de représenter les classes par des arbres : chaque classe admet alors la racine comme représentant privilégié. Par exemple pour la décomposition $\{0,4,5,8\},\ \{1,3,7\},\ \{2\},\ \{6\}$ de $[\![0;8]\!]$ on peut avoir la représentation



Trouver la classe d'équivalence revient alors à remonter jusqu'à la racine
et unir deux classe revient à ajouter un arbre parmi les fils de la racine d'un autre arbre.
Par exemple unir les classes d'équivalence de 1 (représentée par 3) et de 2 peut donner



En pratique on peut représenter ces arbres (une forêt) par un tableau `t` où `t[i]` est la valeur du père, sauf pour les racines pour lesquelles `t[i]` vaut i, elles sont leur propre père. La partition initiale est donc représentée par `[4, 3, 2, 3, 4, 8, 6, 3, 4]`.
L'opération d'union faite ci-dessus consiste juste à changer la valeur de `t[2]` qui devient 3, la valeur de la classe de 1, à la place de 2. On note que cette opération change le représentant de la classe de 2.
On remonte le long de l'arbre pour trouver la racine pour `trouver`.

Fonction de recherche du représentant

def `trouver`(*i : élément, t : tableau*) =
  si $t[i] = i$ **alors renvoyer** i
  sinon renvoyer `trouver`($t[i]$, t)

Pour `unir`, la racine d'une classe (ici la seconde) passe comme fils de la racine de l'autre classe d'équivalence.

Fonction de fusion

def `unir`(*i : élément, j : élément, t : tableau*) =
  $a \leftarrow$ `trouver`(i, t)
  $b \leftarrow$ `trouver`(j, t)
  $t[b] \leftarrow a$

Il faut créer la partition initiale, où toutes les classes d'équivalence sont des singletons.

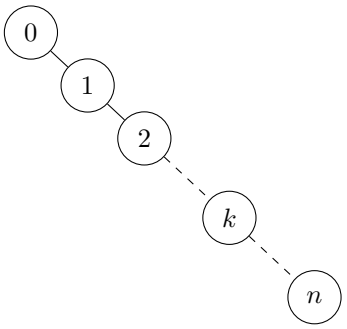
Fonction de création

def `creer_partition`(*taille : entier*) =
  $t \leftarrow$ tableau d'entiers de taille n
  pour $i = 0$ *à* $n - 1$ **faire** $t[i] \leftarrow i$
  renvoyer t

Exercice 1 **Solution page 9**
Écrire ces fonctions dans le langage C et dans le langage OCaml

I.2 Équilibrage

On peut remarquer que la complexité des fonctions peut être importante.
Par exemple si on l'appelle pour les paires $(n-1, n)$ puis $(n-2, n)$ et ainsi de suite jusqu'à $(1, n)$, on obtient un arbre dégénéré de hauteur $n-1$ et l'appel de `trouver(`n`, `t`)` aura une complexité en $\mathcal{O}(n)$.
La dernière union aura eu la même complexité.



Le problème est classique pour les arbres : il provient du déséquilibre possible entre les fils.
On veut donc optimiser faire le choix du représentant qui devient le parent d'un autre.
Comme c'est la hauteur qui est pénalisante, on choisit la racine de l'arbre de hauteur maximale comme père de l'autre. Cependant, on veut éviter de calculer la hauteur à chaque fusion, on va donc intégrer la hauteur dans la valeur de $t[r]$. On pourrait définir un couple (hauteur, père) pour chaque i mais on n'a pas besoin de la hauteur pour les fils et, pour les racines, on a juste besoin de savoir qu'elles n'ont pas de père.
On modifier la structure de donnée : une partition de E sera représentée par un tableau t de taille n tel que

- si i est le représentant de sa classe, alors $t[i] = -1 - h_i$ où h_i est la hauteur de l'arbre dont i est la racine, on ajoute 1 pour éviter la hauteur 0.
- sinon, $t[i]$ est égal au père de i dans la forêt.

Par exemple, la partition initiale est représentée par `[4, 3, -1, -2, -3, 8, -1, 3, 4]`.
Lors de la fusion de deux classes représentées par deux arbres, on ajoute celui des arbres qui sera placé comme fils dans l'autre en prenant celui de hauteur minimale.
Si on ajoute un arbre de hauteur h comme fils d'un arbre de hauteur h' avec $h \leqslant h'$ alors

- soit $h' = h$ et alors la hauteur de l'arbre hôte augmente de 1,
- soit $h' > h$ et alors la hauteur de l'arbre hôte est inchangée.

Algorithme 1 : structure union-find équilibrée

```
def creer_partition(taille : entier) =
    t ← tableau d'entiers de taille n
    pour i = 0 à n − 1 faire t[i] ← −1
    renvoyer t
def trouver(i : élément, t : tableau) =
    si t[i] < 0 alors renvoyer i
    sinon renvoyer trouver(t[i], t)
def unir(i : élément, j : élément, t : tableau) =
    a ← trouver(i, t)
    b ← trouver(j, t)
    si t[a] = t[b] alors
        t[b] ← a
        t[a] ← t[a] − 1
    sinon si t[a] < t[b] alors
        t[b] ← a
    sinon
        t[a] ← b
```

Exercice 2 **Solution page 9**

Écrire ces fonctions dans le langage C et dans le langage OCaml

Théorème 1

Les fonctions modifiées, `trouver` et `unir`, ont une complexité en $\mathcal{O}\big(\log(n)\big)$ pour une partition d'un ensemble de taille n.

Démonstration On montre que, si une classe d'équivalence admet p éléments, alors la hauteur h de l'arbre qui la représente vérifie $p \geqslant 2^h$.

- Si $p = 1$, alors la racine est le seul élément de la classe d'équivalence et la hauteur est nulle ; on a $p = 1 = 2^0 == 2^h$, l'inégalité est une égalité.
- On suppose que le résultat est valide pour toute classe d'équivalence de cardinal p au plus.
  On considère une classe d'équivalence C de cardinal $|C| = p + 1$.
  C est le résultat d'une fusion de deux classes C_1 et C_2 de taille inférieure.
  On note h_1 et h_2 les hauteurs des arbres représentant de C_1 et C_2.
  - Si $h_1 = h_2$ alors la hauteur de l'arbre représentant C est $h = h_1 + 1$.
    On a $p + 1 = |C| = |C_1| + |C_2| \geqslant 2^{h_1} + 2^{h_2} = 2^{h_1} + 2^{h_1} = 2^{h_1+1} = 2^h$.
  - Si $h_1 > h_2$ alors la hauteur de l'arbre représentant C est $h = h_1$.
    On a $p + 1 = |C| = |C_1| + |C_2| \geqslant |C_1| \geqslant 2^{h_1} 2^h$.
  - De même pour $h_1 < h_2$.

  Dans tous les cas on a bien $p + 1 \geqslant 2^h$.

La récurrence est prouvée.
Comme la complexité des opérations est proportionnelle à la hauteur des arbres rencontrés et que celle-ci est majorée par $\log_2\big(|C|\big) \leqslant \log_2(n)$, on a bien la majoration souhaitée. . . . . ■

I.3 Contraction des chemins

On peut remarquer que, lors du calcul de la fonction `trouver`, on calcule le représentant commun à toutes les valeurs d'une branche.Si on donne directement la valeur trouvée comme père de tous les nœuds, on rapproche les nœuds de la racine, l'arbre est applati. On évite des calculs futurs dans la fonction `trouver`, donc dans la fonction `unir` aussi.
Seule la fonction `trouver` est modifiée

Fonction trouver optimisée

```
def trouver(i : élément, t : tableau) =
    si t[i] < 0 alors renvoyer i
    sinon
        p ← trouver(t[i], t)
        t[i] ← p
        renvoyer p
```

Exercice 3 **Solution page 10**

Écrire cette nouvelle fonction `trouver` dans le langage C et dans le langage OCaml

Dans ce cas la valeur du tableau pour les racines n'indique plus qu'un majorant de la hauteur mais cela permet cependant de garder la complexité prouvée ci-dessus. En fait, on a mieux, on admet le résultat suivant.

Théorème 2

Avec cette modification, `trouver` et `unir`, ont une complexité amortie en $\mathcal{O}(\alpha(n))$ pour une partition d'un ensemble de taille n où α est une fonction croissante telle que $\alpha(n) \leq 4$ si n est inférieur au nombre de particules dans l'univers.

Exercice 4 - Application : composantes connexes **Solution page 11**

Écrire une fonction en OCaml qui renvoie un tableau associant, pour chaque sommet, un identifiant de la composante connexe le contenant.
Quelle est sa complexité ?

II Arbre couvrant minimum

II.1 Définition

On rappelle qu'un arbre (non enraciné) est un graphe non orienté connexe sans cycle.
Les résultats suivants on été vus en première année

Exercice 5 **Solution page 11**

Prouver que, pour un graphe non orienté $G = (S, A)$,

1. si G est connexe, alors $|A| \geqslant |S| - 1$,
2. si G est sans cycle, alors $|A| \leqslant |S| - 1$.

Exercice 6 **Solution page 12**

Prouver que, pour un graphe non orienté $G = (S, A)$, les 3 propriétés suivantes sont équivalentes :

1. G est un arbre ;
2. G est connexe avec $|S| - 1$ arêtes ;
3. G est sans cycle avec $|S| - 1$ arêtes.

Définition 1

Si $G = (S, A)$ est un graphe non orienté **connexe**, un *arbre couvrant* de G est un arbre dont l'ensemble des sommets est S et l'ensemble des arêtes est inclus dans A.

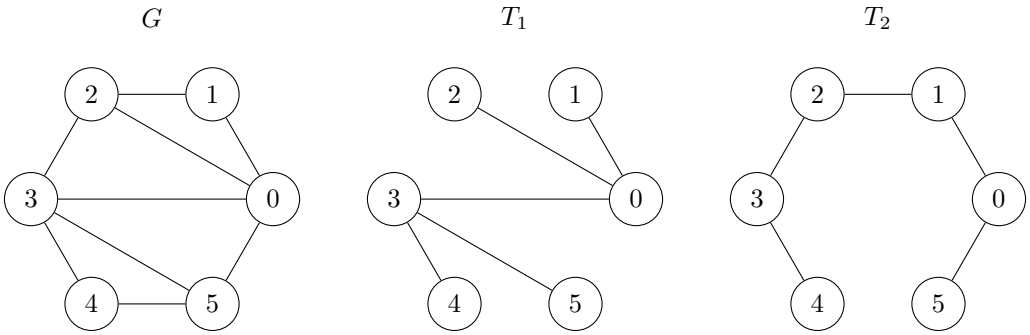
Exercice 7 **Solution page 12**

On considère K_n le graphe complet de taille n dont les arêtes sont coloriées en bleu ou rouge ; montrer qu'il existe un arbre couvrant monochrome.

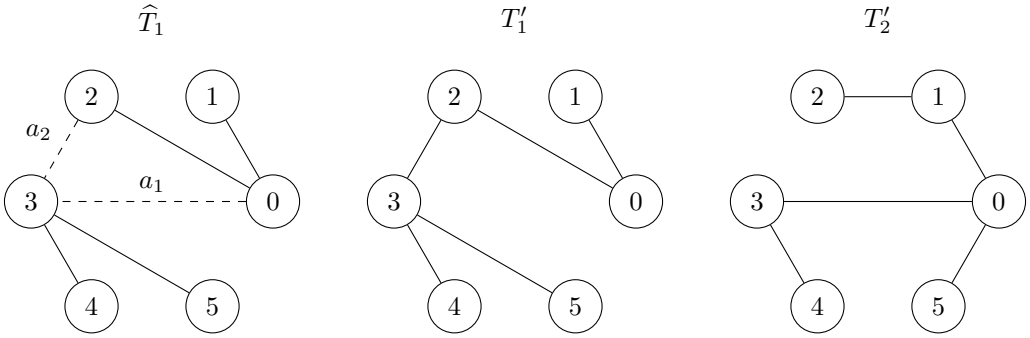
Théorème 3 Lemme d'échange

Si $T_1 = (S, A_1)$ et $T_2 = (S, A_2)$ sont des arbres couvrants du graphe connexe $G = (S, A)$ avec $A_1 \neq A_2$ alors, pour toute arête $a_1 \in A_1 \setminus A_2$, il existe une arête $a_2 \in A_2 \setminus A_1$ telle que, en posant $A'_1 = A_1 \cup \{a_2\} \setminus \{a_1\}$ et $A'_2 = A_2 \cup \{a_1\} \setminus \{a_2\}$,
(S, A'_1) et (S, A'_2) sont des arbres couvrants de G.

On peut donc échanger 2 arêtes entre T_1 et T_2 en conservant des arbres couvrants.



Démonstration On remarque que, comme A_1 et A_2 sont distincts mais de même cardinal alors $A_1 \setminus A_2$et $A_2 \setminus A_1$ sont non vides : on choisit $a_1 \in A_1 \setminus A_2$.
On pose $\widehat{A}_1 = A_1 \setminus \{a_1\}$ et $\widehat{T}_1 = (S, \widehat{A}_1)$.
$\widehat{T}_1$ admet $|S| - 2$ arêtes donc n'est pas connexe, cependant ajouter une seule arête, a_1, le rend connexe donc $\widehat{T}_1$ a deux composantes connexes C et C'.
Si s et s' sont les extrémités de a_1, on a, par exemple, $s \in S$ et $s' \in S'$.
Comme T_2 est connexe il existe un chemin dans T_2 entre s et s' ; ce chemin contient une arête dont les extrémités sont dans C pour l'une et C' pour l'autre. On la note a_2.



L'ajout de l'arête a_2 dans le graphe $\widehat{T}_1$ le transforme alors en un graphe connexe T'_1, comme il admet $|S| - 1$ arêtes, c'est un arbre couvrant.
Le chemin se s à s' augmenté de a_1 est un cycle contenant a_2 dans $(S, T_2 \cup \{a_1\})$ donc, en enlevant a_2 on garde la connexité et T'_2 est connexe avec $|S| - 1$ arêtes, c'est un arbre couvrant. . . . . ■

N.B. On peut remarquer que, comme T_2 est un arbre, le chemin entre s et s' dans T_2 est unique. De plus, comme T_2 est acyclique il existe une seule arête entre C et C'.

Définition 2

Un graphe *pondéré* est un triplet $G = (S, A, w)$ tel que (S, A) est un graphe et w est une fonction de A vers $\mathbb{R}$. $f(a)$ est le *poids* de l'arête a.

Les caractéristiques des graphes (orienté, connexe, ...) se transportent aux graphes pondérés.

Définition 3

Si $G = (S, A, w)$ est un graphe non orienté pondéré connexe et $T = (S, A')$ un arbre couvrant de G, le *poids* de T est $w(T) = \sum_{a \in A'} w(a)$.
On dit qu'un arbre couvrant de G, T, est un *arbre couvrant minimal* de G si son poids est minimal parmi tous les arbres couvrants de G.

Un arbre couvrant minimal existe toujours car il n'y a qu'un nombre fini d'arbres couvrant donc la borne inférieure des poids est atteinte.

Exercice 8 **Solution page 12**
Prouver que si les poids des arêtes sont distincts alors il existe un unique arbre couvrant minimal.

II.2 Algorithme de Kruskal

Pour calculer un arbre couvrant minimal, on va utiliser un algorithme glouton : on construit un graphe acyclique de plus en plus gros (en nombre d'arêtes) en ajoutant les arêtes les moins lourdes qui ne créent pas de cycle. On maintient une structure union-find qui reflète les composantes connexes du graphe en construction : elle permet de savoir si les deux extrémités d'une arête appartiennent à la même composante connexe et elle fusionne les deux composantes connexes si ce n'est pas le cas.

Algorithme 2 : algorithme de Kruskal

Entrées : Graphe pondéré $G = (S, A, f)$ non orienté connexe
Sorties : Un ensemble d'arêtes formant un arbre couvrant minimal
Début algorithme
  $n \leftarrow$ `ordre`(G)
  $A' \leftarrow \emptyset$
  $A_0 \leftarrow A$ trié par ordre croissant de poids
  `uf` $\leftarrow$ `creer_partition`(n)
(†) **pour chaque** (i, j) *dans* A_0 **faire**
    si `trouver(`i`, uf)` $\neq$ `trouver(`j`, uf)` **alors**
      $A' \leftarrow A' \cup \{(i, j)\}$
      `unir(`i`, `j`, uf)`
  renvoyer $T = (S, A')$

Analyse

- La terminaison est immédiate, on trie et on parcourt toutes les arêtes.
- On part d'un graphe sans arête et, à chaque étape, on n'ajoute une arête que si elle ne crée pas de cycle : par construction le graphe renvoyé est acyclique.
- Si T n'était pas connexe alors, comme G est connexe, il devrait exister une arête de G entre deux composantes connexes de T. Lors du traitement de cette arête par (†) dans l'algorithme, comme ses extrémités sont des composantes connexes distinctes de T donc du graphe à ce moment, cette arête aurait du être ajoutée. On aboutit à une contradiction : T est connexe.
- On considère, parmi les arbres couvrant minimums de G, un arbre $T^* = (S, T^*)$ ayant un nombre maximal d'arêtes en commun avec T.
  Pour simplifier l'écriture, si $\Gamma = (S, \widehat{A})$ est un graphe, on note $\Gamma_{+a} = (S, \widehat{A} \cup \{a\})$ et $\Gamma_{-a} = (S, \widehat{A} \setminus \{a\})$ les graphes obtenus en ajoutant ou en levant une arête.
  Si on a $T^* \neq T$ alors le lemme d'échange (théorème 3) donne l'existence d'arêtes $a' \in A' \setminus A^*$ et $a^* \in A^* \setminus A'$ telles que $T_{+a^*-a'}$ et $T^*_{-a^*+a'}$ sont des arbres couvrants.
  On a $w(T^*_{-a^*+a'}) = w(T^*) + w(a') - w(a^*)$.
  $T^*_{-a^*+a'}$ admet a' en commun avec T en plus alors que a^* n'est pas une arête de T ; $T^*_{-a^*+a'}$ admet une arête en commun avec T de plus que T^* donc ce n'est pas un arbre couvrant minimal ; ainsi $w(T^*_{-a^*+a'}) > w(T^*)$ d'où $w(a') > w(a^*)$: a^* est traité avant a' dans l'algorithme.
  Lors du traitement de a^* par l'algorithme, à la ligne (†), a^* n'est pas gardé car $a^* \notin A'$. Cela signifie que ses extrémités appartiennent à la même composante connexe de l'arbre T à ce

moment. Elles appartiennent donc à une même composante connexe de $T_{-a'}$ car a' n'a pas encore été ajouté.

Dans ce cas il est impossible que l'ajout de a^* à $T_{-a'}$ le rende connexe, ce qui contredit le fait que $T_{+a^*-a'}$ est un arbre couvrant.

La contradiction montre que $T = T^*$: T est minimal.

- Les 3 dernier points prouvent que T est un arbre couvrant minimal.
- La complexité de l'algorithme est celle du tri, en $\mathcal{O}\big(|A| \log_2(|A|)\big)$ ajouée au parcous de la liste des arêtes avec des opérations qu'on peut estimer en temps constant, en $\mathcal{O}\big(|A|\big)$. C'est la complexité du tri qui donne la complexité totale.

Solutions

Exercice 1

```c
int* creer_partition(int n){
    int* t = malloc(n * sizeof(int));
    for (int i = 0; i < n; i+=1){t[i] = i;}
    return t;
}

int trouver(int i, int* t){
    if (t[i] == i){return i;}
    else{return trouver(t[i], i);}
}

void unir(int i, int j, int* t){
    int a = trouver(i, t);
    int b = trouver(j, t);
    t[b] = a;
}
```

```
let creerUF n =
  Array.init n (fun i -> i)

let rec trouver i t =
  if t.(i) = i
  then i
  else find t.(i) t

let union i j t =
  let a = find i t in
  let b = find j t in
  t.(b) <- a
```

Exercice 2

```
int* creer_partition(int n){
    int* t = malloc(n * sizeof(int));
    for (int i = 0; i < n; i+=1){t[i] = -1;}
    return t;
}

int trouver(int i, int* t){
    if (t[i] < 0){return i;}
    else{return trouver(t[i], i);}
}

void unir(int i, int j, int* t){
    int a = trouver(i, t);
    int b = trouver(j, t);
    if(t[a] = t[b]){
         t[b] = a;
         t[a] = t[a] - 1;
    }
    else{
        if(t[a] < t[b]){t{b] = a;}
        else{t[a] = b;}
    }
}
```

```
let creer_partition n =
  Array.make n (-1)

let rec trouver i t =
  if t.(i) < 0
  then i
  else find t.(i) t

let union i j t =
  let a = find i t in
  let b = find j t in
  if t[a] = t[b]
  then {t[b] = a; t[a] = t[a] - 1}
  else if  t[a] < t[b]
    then t{b] = a
    else t[a] = b
```

Exercice 3

```
int trouver(int i, int* t){
    if (t[i] < 0){return i;}
    else{
         int p = trouver(t[i], i);
         t[i] = p;
         return p;
    }
}
```

```
let rec trouver i t =
  if t.(i) < 0
  then i
  else begin
    let p = trouver t.(i) t in
    t.(i) <- p;
    p
  end
```

Exercice 4
Le graphe est donné sous forme d'un tableau de listes de voisins.

```
let composantes  g =
  let n = Array.length g in
  let uf = creer_partition n in
  let traiter i j =
    if (trouver i uf) <> (trouver j uf)
    then unir i j uf in
  for i = 0 to (n-1) do
    List.iter (traiter i) g.(i) done;
  let cc = Array.make n 0 in
  for i = 0 to (n-1) do
    cc.(i) <- trouver i uf;
  cc
```

La recherche et la fusion peuvent être considérée comme se faisant en temps constant donc la fonction a une complexité linéaire, en $\mathcal{O}\big(|S| + |A|\big)$.

Exercice 5

1. Preuve de G est connexe implique $|A| \geqslant |S| - 1$.
   Si $|A| = 0$, alors G n'est connexe que s'il n'a qu'un seul sommet et alors $|A| = 0 = |S| - 1$.
   On suppose que le résultat est vrai pour tout graphe connexe dont le nombre d'arêtes est p.
   On considère un graphe connexe $G = (S, A)$ avec $|A| = p + 1$
   Comme G est connexe, tout sommet est de degré au moins 1.
   - Si tous les sommets sont de degré au moins 2, on peut définir un chemin infini qui doit donc contenir une boucle car le nombre de sommets est fini (†). Si on enlève une arête de cette boucle, on obtient alors un graphe $G' = (S, A')$ avec une arête de moins et qui est toujours connexe.
     D'après l'hypothèse de récurrence, $|A'| \geqslant |S'| - 1$ donc $|A| = |A'| + 1 \geqslant |A'| \geqslant |S'| - 1 = |S| - 1$.
   - S'il existe un sommets sont de degré 1, on l'enlève ainsi que son unique arête adjacente. On obtient un graphe qui reste connexe.
     D'après l'hypothèse de récurrence, $|A'| \geqslant |S'| - 1$ donc $|A| = |A'| + 1 \geqslant |S'| + 1 - 1 = |S| - 1$.

   Le résultat est prouvé par récurrence sur $|A|$.
2. Preuve de G est sans cycle implique $|A| \leqslant |S| - 1$.
   Si $|S| = 1$, alors $|A| = 0 = |S| - 1$.
   On suppose que le résultat est vrai pour tout graphe sans cycle d'ordre n.
   On considère un graphe acyclique $G = (S, A)$ avec $|S| = n + 1$
   Comme G ne possède pas de cycle, la contraposée du résultat (†) ci-dessus implique que G admet un sommet de degré 0 ou 1. En supprimant ce sommet, on obtient un graphe $G' = (S', A')$, qui est toujours acyclique, d'ordre n.

D'après l'hypothèse de récurrence, $|A'| \leqslant |S'|-1$ or $|S| = |S'|+1$ et $|A| = |A'|$ ou $|A| = |A'|+1$ donc $|A| \leqslant |A'| + 1 \leqslant |S'| + 1 - 1 = |S| - 1$.
Le résultat est prouvé par récurrence sur $|S|$.

Exercice 6

1. $\Rightarrow$ 3. Un arbre $T = (S, A)$ est connexe donc $|A| \geqslant |S| - 1$ et acyclique donc $|A| \leqslant |S| - 1$. Il est donc sans cycle et vérifie $|A| = |S| - 1$.
3. $\Rightarrow$ 2. Soit G un graphe sans cycle vérifiant $|A| = |S| - 1$.
   S'il n'était pas connexe, on pourrait ajouter une arête entre deux sommets appartenant à des composantes connexes distinctes. On obtiendrait un graphe $G' = (S, A')$ toujours acyclique avec une arête de plus donc $|A'| = |A| + 1 > |S| - 1$ ce qui contredit le résultat $|A'| \leqslant |S| - 1$ pour un graphe sans cycle.
   Ainsi G est connexe et vérifie toujours $|A| = |S| - 1$.
2. $\Rightarrow$ 1. Soit G un graphe connexe vérifiant $|A| = |S| - 1$.
   Si G admettait un cycle, on pourrait enlever une arête de ce cycle en conservant la connexité. On obtiendrait un graphe $G' = (S, A')$ toujours acyclique avec une arête de moins donc $|A'| = |A| - 1 < |S| - 1$ ce qui contredit le résultat $|A'| \geqslant |S| - 1$ pour un graphe connexe.
   Ainsi G est connexe et acyclique donc est un arbre.

Exercice 7
Le résultat est vrai pour $n = 1$ (il n'y a pas d'arête).
On suppose que le résultat est vrai pour K_n.
Soit $G = (S, A)$ un graphe complet d'ordre $n + 1$ dont les arêtes sont colorées en bleu ou rouge.
On isole un sommet s_0, le graphe induit en enlevant s_0, G', est un graphe complet à n sommet donc admet un arbre couvrant monochrome, par exemple bleu.

- S'il existe une arête bleue entre s_0 et un des autres sommets, il suffit de la rajouter à l'arbre couvrant de G' pour obtenir un arbre couvrant bleu de G.
- Sinon toutes les arêtes entre s_0 et les autres sommets sont rouges ; elles forment alors un arbre couvrant rouge de G.

Le résultat est prouvé par récurrence.

Exercice 8
On va montrer la contraposée : s'il existe deux arbres couvrants minimaux distincts alors il existe deux arêtes de même poids.
On suppose que $G = (S, A)$ admet $T_1 = (S, A_1)$ et $T_3 = (S, A_2)$ comme arbres couvrants minimaux avec $A_1 \neq A_2$. D'après l'exercice 6, $|A_1| = |A_2| = |S|$.
Ainsi, $A_1 \setminus A_2 \neq \emptyset$ et $A_2 \setminus A_1 \neq \emptyset$.
On considère une arête de poids minimal dans $A_1 \setminus A_2 \cup A_2 \setminus A_1$, par exemple $a \in A_1 \setminus A_2$.
D'après l'exercice 3, si on ajoute a à T_2, le graphe contient un cycle et si on échange a et une arête a' de ce cycle, on a toujours un arbre couvrant. Le poids de cet arbre couvrant est $w(T_2) + w(a) - w(a') \geqslant w(T_2)$ car T_2 est minimum. Ainsi $w(a) \geqslant w(a')$.
Parmi les arêtes du cycle, il y en a une, a', qui n'appartient pas à A_1, sinon T_1 contiendrait un cycle, ainsi $a' \in A_2 \setminus A_1$. D'après la définition de a on a $w(a) \leqslant w(a')$.
On en déduit l'égalité des poids.