Travaux pratiques 6

Arbres couvrants

Option informatique MP1, MP2 & MP3

Nous allons donner quelques algorithmes qui permettent d'extraire d'un graphe valué un squelette minimal.

I Arbres couvrants

À retenir

Dans cette partie les graphes sont non orientés et connexes mais ne sont pas valués.
n désigne la taille d'un graphe, le nombre de ses sommets.

I.1 Exemple des parcours

Dans les parcours vus dans le chapitre sur les graphe, on peut renvoyer, à la place du tableau des sommets vus (avec les hypothèses de cette partie, ils seront tous vus) mais la liste des arêtes utilisées pour parvenir aux sommets visités.

Arbre du parcours récursif

```
let parcours g s0 =
  let vus = Array.make (taille g) false in
  let arbre. = ref [] in
  let rec visiter s t = (* on visite s depuis t *)
    if not vus.(s)
    then begin vus.(s) <- true;
         if s <> t then arbre := (t, s) :: !arbre;
         List.iter visiter (voisins g s) s end
  in visiter s0;
  !arbre;;
```

Exercice 1 - Arbre du parcours en largeur
Écrire de même, une fonction qui renvoie la liste des arêtes utilisées dans le parcours en largeur.

On remarque que les arêtes forment un structure plus arborescente dans le cas du parcours en largeur.
On remarque aussi qu'on a obtenu, dans les deux cas, un graphe connexe et acyclique : un arbre. Les arbres ont été étudiés en exercices dans le chapitre 4. On a vu en particulier qu'un graphe (S', A') était un arbre si et seulement si il était connexe et vérifiait $|A'| = |S'| - 1$.

Définition 1 : Arbre couvrant

Si $G = (S, A)$ est un graphe non orienté connexe, un arbre couvrant de G est un graphe $G' = (S, A')$ avec $A' \subset A$ tel que G est connexe et $|A'| = |S| - 1$.

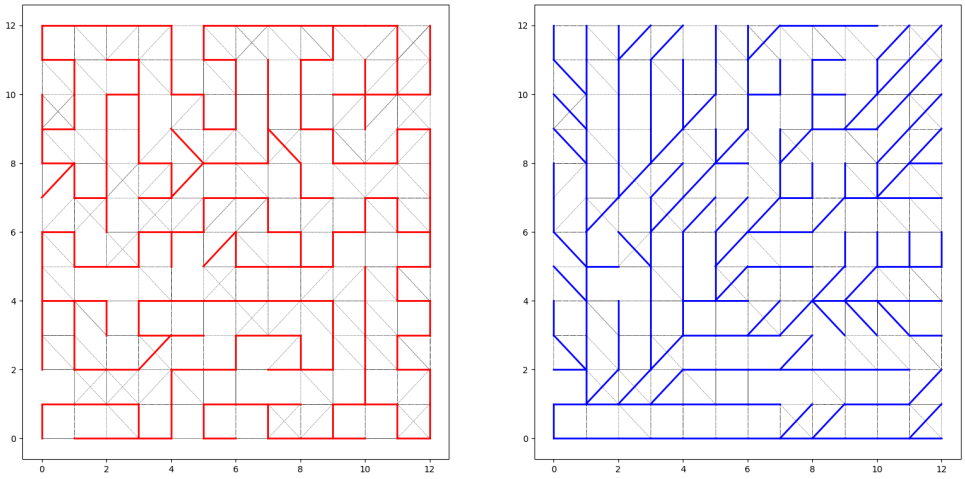
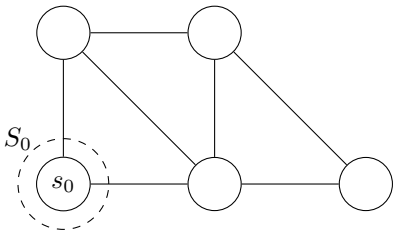


FIGURE 1 – Les arêtes des parcours (récursif en profondeur puis en largeur) d'un graphe

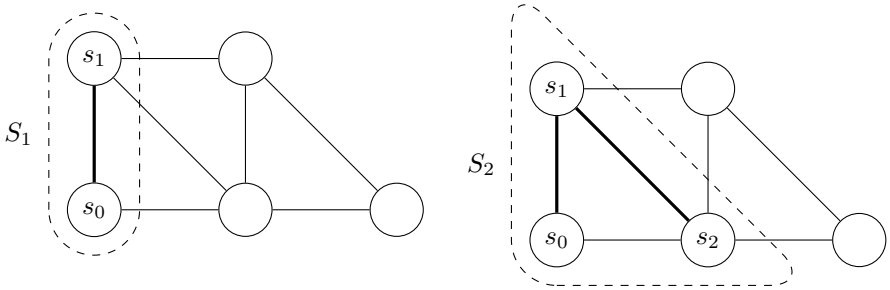
Un parcours permet de définir un arbre couvrant. Cela prouve l'existence de ceux-ci.
On va maintenant donner 2 moyens de construire des arbres couvrants.

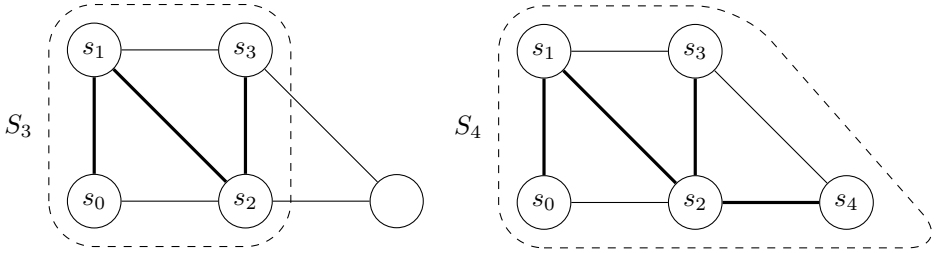
I.2 Coupures

Une première manière de définir un arbre couvrant est de faire grossir un arbre (S', A') avec $S' \subset S$ et $A' \subset A$ dans un graphe $G == (S, A)$.
On choisit un sommet de départ, $s_0 \in S$.
Si on pose $S_0 = \{s_0\}$, et $A_0 = \emptyset$, (S_0, A_0) est un arbre.



On suppose qu'on a définit un arbre (S_k, A_k) avec $S_k \subset S$, $|S_k| = k + 1$, $A_k \subset A$ et $|A_k| = k$.
On choisit alors une arête $a_k = (t_k, s_k)$ avec $t_k \in S_k$ et $s_k \notin S_k$.
On peut ajouter s_k et a_k : $S_{k+1} = S_k \cup \{s_k\}$ et $A_{k+1} = A_k \cup \{a_k\}$.
Si on fait cette opération $n - 1$ fois depuis (S_0, A_0) on obtient un arbre couvrant de G.





On voit apparaître une notion qui sera utile dans d'autres contextes.

Définition 2 : Coupure

Une coupure d'un graphe $G = (S, A)$ est une partition de S en deux parties non vides disjointes : $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$, $S_1 \neq \emptyset$ et $S_2 \neq \emptyset$.
Une **arête traversante** associée à une coupure (S_1, S_2) de $G = (S_A)$ est une arête $(a, b) \in A$ telle que $a \in S_1$ et $b \in S_2$.

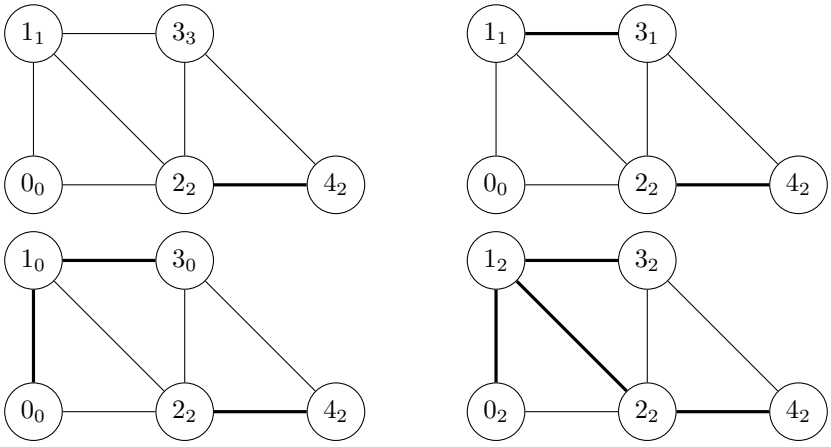
Exercice 2 - Propriétés des coupures
(S_1, S_2) est une coupure de $G = (S_A)$ connexe. $G' = (S, A')$ est un arbre couvrant de G.

1. Prouver qu'il existe une arête traversante pour la coupure.
2. Prouver que A' contient une arête traversante pour la coupure.
3. Prouver que si a est une arête traversante pour la coupure n'appartenant pas à A' alors A' contient une arête traversante a' telle que, pour $A'' = A' \cup \{a\} \setminus \{a'\}$, (S, A'') est aussi un arbre couvrant.

La question 1. montre que l'algorithme est possible.

I.3 Union-find

On a construit un arbre couvrant en augmentant un arbre. On peut aussi utiliser un principe dual : on ajoute des arêtes qui relient des composantes connexes distinctes.



Les indices dénotent les composantes connexes.

L'algorithme est simple :

- $A' = \emptyset$
- Pour chaque arête du graphe
- si les deux sommets qu'elle joint ne sont pas dans la même composante connexe
- l'ajouter à A'.

Exercice 3
Prouver que cet algorithme construit un arbre couvrant, quel que soit l'ordre des arêtes.

Il reste maintenant à tester quand deux sommets sont une même composante connexe.
On peut faire un parcours du graphe pour déterminer les composantes connexes : à chaque fois que l'on ajoute une arête dans A' on refait le calcul. Comme il y a au plus n arêtes, la complexité est un $\mathcal{O}(n)$.
On peut aussi remarquer que les composantes connexes obtenues ne sont pas indépendantes : chaque fois que l'on ajoute une arête, deux composantes sont réunies et les autres sont inchangées.
On peut définir une structure de données adaptée : UNION-FIND (UNIR-ET-TROUVER en français). Elle consiste à gérer des classes d'équivalence (par exemple les composantes connexes) sur un ensemble à n éléments représenté par $\{0, 1, 2, \ldots, n-1\}$. En voici le type abstrait.

- `unionFind`, le type à à définir
- `createUF : int -> unionFind`, `createUF` crée une structure sur $\{0, 1, 2, \ldots, n-1\}$ avec n classes d'équivalence, chacune réduite à un entier i.
- `find : int -> unionFind -> int`, `find a uf` renvoie l'entier représentant la classe d'équivalence de k dans la structure `uf`.
- `union : int -> int-> unionFind -> unit`, `union i j uf` modifie la structure en réunissant les classes d'équivalence contenant respectivement i et j.

Les axiomes qui doivent être vérifiés sont :

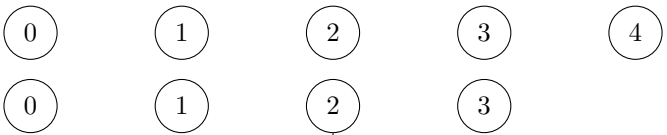
- Après l'initialisation, sans avoir effectué d'unions, on a `find i uf` qui renvoie i.
- Deux éléments a et b appartiennent à la même classe d'équivalence si et seulement si `find a uf = find b uf`.
- On suppose qu'on a `find a uf = k` et `find b uf = l`.
  On effectue `union i j uf`.
  - Si $k \notin \{i, j\}$ alors on a toujours `find a uf = k`
  - Si $k \in \{i, j\}$ et $k \in \{i, j\}$ alors `find a uf = find b uf`, on n'exige rien sur la valeur.

On commence par une implémentation simple.

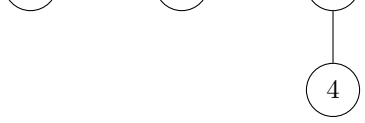
Exercice 4
Écrire les fonctions si le type est celui d'un tableau dont les valeurs sont les identifiants des composantes. Quelles sont les complexités ?

Nous allons définir une autre implémentation qui permettra une complexité meilleure.
L'idée est de représenter les classes d'équivalences par des arbres, la racine donne l'indice de la composante et l'union consiste à placer une des classes d'équivalence comme fils de l'autre. On commence par n arbres réduits à leur racine.
La construction du début de cette partie donne l'évolution qui suit.

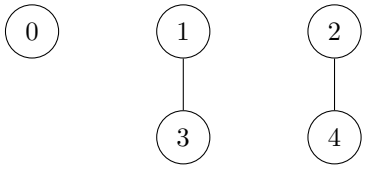
On part de n composantes



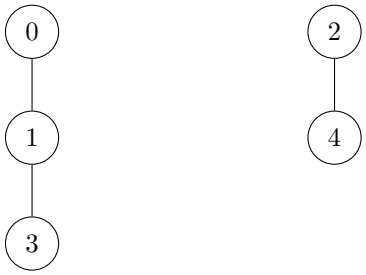
2 et 4 sont équivalents



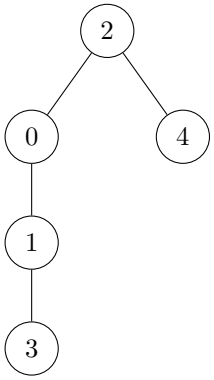
1 et 3 sont équivalents



0 et 1 sont équivalents



1 et 2 sont équivalents



En pratique, on continue à utiliser un tableau d'entiers.

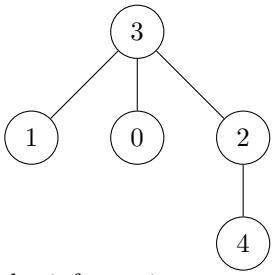
- Chaque valeur du tableau désigne le père de l'élément, sauf pour les racines, on a alors `t.(r) = r`.
- Pour trouver le représentant, on remonte les pères jusqu'à trouver un point fixe.
- Pour unir deux classes, la racine de la seconde prend la racine de la première comme père.

Dans l'exemple ci-dessus, les tableaux successifs sont

```
[| 0; 1; 2; 3; 4|]
[| 0; 1; 2; 3; 2|]
[| 0; 1; 2; 1; 2|]
[| 0; 0; 2; 1; 2|]
[| 0; 0; 0; 1; 2|]
```

Exercice 5
Écrire les fonctions. Quelles sont les complexités ?

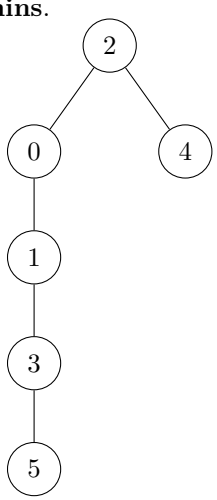
Dans l'exemple ci-dessus, on voit qu'on parvenait à un arbre moins haut en choisissant mieux le père dans les union, on pouvait obtenir l'arbre ci-contre.



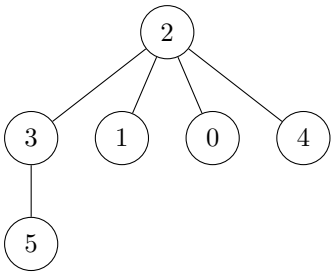
Pour obtenir de tels constructions, on dédouble les informations, par exemple avec un enregistrement de 2 tableaux. En plus du père, on ajoute la hauteur de l'arbre. Lors d'une union, on garde comme racine celle dont la hauteur est la plus grande.

Exercice 6
Écrire le type et les fonctions correspondantes.
Prouver que la complexité de `union` est logarithmique, celle de `find` restant constante.

Le résultat précédent donne une excellente complexité mais on peut faire mieux encore.
La structure est utilisée à de nombreuses reprises et on va appeler `find a uf` plusieurs fois. Or, à chaque fois, on remonte la chaîne des père pour retrouver l'identifiant de la composante. On peut mémoriser le résultat obtenu en attribuant comme père l'identifiant trouvé : c'est la **compression des chemins**.



Si on effectue `find 3 uf` dans l'arbre ci-contre, on obtient 2 et l'arbre va être transformé en



On admet qu'alors la complexité amortie des recherche est constante pour les tailles possibles.

Exercice 7
Modifier la fonction `find` pour inclure la compression des chemins.

II Arbre couvrant minimal

À retenir

Dans toute la suite on suppose que les graphes sont valués, non orientés et connexes.

On note $w(a)$ le poids d'une arête $a \in A$.

Définition 3 : Arbre couvrant minimal

Le **poids** d'un arbre couvrant est $w(G') = \sum_{a \in A'} w(a)$.

Un arbre couvrant est **minimal** si son poids est minimal parmi tous les arbres couvrants.

Exercice 8 - Existence
Prouver que tout graphe connexe admet un arbre couvrant minimal.

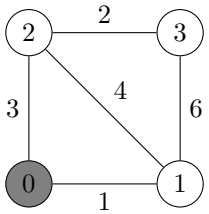
Nous allons utiliser les constructions ci-dessus pour calculer un arbre couvrant minimal.

II.1 Algorithme de Jarnik (dit aussi de Prim)

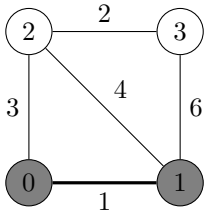
Le premier algorithme utilise la première construction en choisissant l'arête traversante de poids minimal. On retrouve un algorithme proche de l'algorithme de Dijkstra avec la même complexité.

- On part d'un sommet s0.
- On ajoute les arêtes une par une en choisissant celle qui sort de l'arbre déjà construit et qui est de poids minimum.

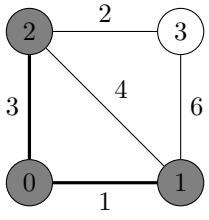
Initialisation, on part de 0



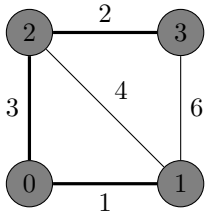
On ajoute $(0,1)$ de poids minimal parmi $(0,1)$ et $(0,2)$



Le poids minimal parmi $(0,2)$, $(1,2)$ et $(1,3)$ est $(0,2)$



Le poids minimal parmi $(1,3)$ et $(2,3)$ est $(2,3)$



Exercice 9 - Implémentation
Écrire une fonction `prim g s0` qui renvoie l'arbre couvrant minimal selon cette méthode.

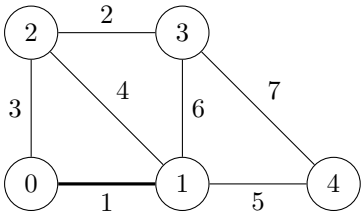
Exercice 10 - Preuve
Prouve que l'algorithme de Prim fournit bien un arbre couvrant minimal.

II.2 Algorithme de Kruskal

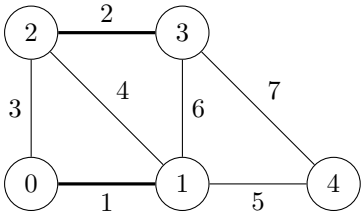
On peut aussi utiliser la seconde construction ci-dessus : pour cela on choisit de considérer les arêtes dans l'ordre croissant de poids.

- On trie les arêtes : $A = (u_1, u_2, \cdots, u_p)$ avec $w(u_i) \leqslant w(u_{i+1})$.
- On crée un ensemble d'arêtes vides, A'.
- pour i allant de 1 à p :
  1. $u_i = (a_i, b_i)$
  2. si a_i et b_i sont dans des composantes connexes distinctes
  3. ajouter u_i à A'
- (S, A') est un arbre couvrant minimal.

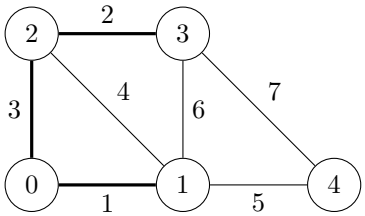
On commence par l'arête $(0, 1)$ de poids 1



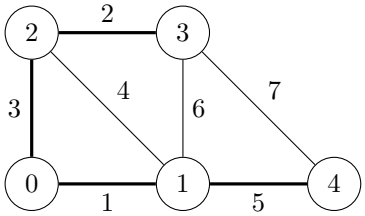
On ajoute l'arête $(2, 3)$ de poids 2



Ensuite l'arête $(0, 2)$ de poids 3



L'arête $(1, 2)$ de poids 4 créerait un cycle, on finit avec l'arête $(1, 4)$ de poids 5.



Exercice 11 - Implémentation
Écrire une fonction `kruskal g` qui renvoie l'arbre couvrant minimal selon cette méthode.

Exercice 12 - Preuve
Prouve que l'algorithme de Kruskal fournit bien un arbre couvrant minimal.

Solutions

Solution de l'exercice 1 - Arbre du parcours en largeur
Le tableau des pères permet de reconstituer la liste des arêtes.

```
let parcoursLargeur g s0 =
   let n = taille g in
   let pere = Array.make n None in
   let attente = Queue.create() in
   Queue.add s0 attente;
   pere.(s0) <- Some s0;
   while not (Queue.is_empty attente) do
      let s = Queue.take attente in
      let traiter t = if pere.(t) = None
                      then begin pere.(t) <- Some s;
                           Queue.add t attente end in
    List.iter traiter (voisins g s) done;
  let arbre = ref [] in
  for i = 0 to (n-1) do
     match pere.(i) with
     |Some k when k <> i -> arbre := (i, k) :: !arbre
     |_ -> ()  done;
  !arbre;;
```

Solution de l'exercice 2 - Propriétés des coupures

1. Si $u \in S_1$ et $v \in S_2$, on déduit de la connexité qu'il existe un chemin de a vers b dans (S, A) : $u = t_0 \to t_1 \to t_2 \to \cdots \to t_{q-1} \to t_q = v$. Si i est l'indice minimal tel que $t_i \in S_2$ on a (t_{i-1}, t_i) traversante.
2. C'est la même démonstration dans le graphe (toujours connexe) (S, A').
3. Si on choisit u et v $a = (u, v)$, la question précédente montre l'existence d'un chemin de u à v dans (S, A') qui contient une arête traversante $a' = (t_{i-1}, t_i) = (u', v')$.
   Tout chemin dans A' peut être transformé en un chemin dans A'' en remplaçant si besoin l'arête (u', v') par le chemin de A'', $u' = t_{i-1} \to t_{i-2} \to \cdots \to t_0 = u \to v = t_q \to t_{q-1} \to \cdots \to t_i v'$
   Ainsi (S, A'') est connexe et il contient aussi $|S| - 1$ arêtes, c'est un arbre couvrant.

Solution de l'exercice 3
Chaque arête ajoutée dans A' diminue le nombre de composantes connexes de 1.
On commence par n composantes connexes et on aboutit à au moins une donc A' contient au plus $n - 1$ arêtes.
On remarque que, à chaque étape, toutes les arêtes utilisées (qu'on les ait placées ou non dans A') joihnent des sommets dans la même composante connexe. À la fin on a utilisé toutes les arêtes donc on a reconstitué le graphe G, qui est connexe. On a donc placé $n - 1$ arêtes dans A'.

Solution de l'exercice 4

```
type unionFind = int array;;
```

```
let createUF n =
   Array.init n (fun k -> k);;
```

```
let find a uf =
   uf.(a);;
```

```
let unionF i j uf =
   let n = Array.length uf in
   for k = 0 to (n-1) do
      if uf.(k) = uf.(i) then uf.(k) <- uf.(j) done;;
```

`createUF` et `union` sont de complexité linéaire en la taille de l'ensemble.

Solution de l'exercice 5

```
type unionFind = int array;;
```

```
let createUF n =
   Array.init n (fun k -> k);;
```

```
let find a uf =
   let b = uf.(a) in
   if a = b then a else find b uf;;
```

```
let unionF i j uf =
   let ci = find i uf in
   let cj = find j uf in
   uf.(cj) <- ci done;;
```

`createUF` est de complexité linéaire en la taille de l'ensemble.
`find` peut être de complexité linéaire aussi si les unions ont crée un arbre linéaire.

Solution de l'exercice 6

```
type unionFind = {pere : int array;
                  haut : int array};;
```

```
let createUF n =
   {pere = Array.init n (fun k -> k);
    haut = Array.make n 0};;
```

```
let find a uf =
   let b = uf.pere.(a) in
   if a = b then a else find b uf;;
```

```
let union i j uf =
   let ci = find i uf in
   let cj = find j uf in
   if ci <> cj then begin
      let hi = uf.haut.(ci) in
      let hj = uf.haut.(cj) in
      if hi < hj
      then uf.pere.(ci) <- cj
      else begin uf.pere.(cj) <- ci;
           uf.haut.(ci) <- max hi (hj+1) end end;;
```

On montre l'invariant : la taille d'un arbre est supérieure ou égale à 2^h où h est la hauteur.

- C'est pour chaque arbre à l'initialisation : $1 \geqslant 2^0$.
- Lors d'une fusion on note t_i et t_j les tailles respectives et h_i et h_j les hauteurs respectives des arbres de racine i et j. On suppose $t_i \leqslant t_j$. Il y a trois cas :

— $h_i < h_j$, alors l'arbre en j garde sa hauteur et $2^{h_j} \leqslant tj \leqslant ti + tj$,

— $h_i > h_j$, alors l'arbre en i garde sa hauteur et $2^{h_i} \leqslant ti \leqslant ti + tj$,

— $h_i = h_j$, alors h_j est remplacé par $h_i + 1$ et $2^{h_i+1} = 2^{h_j} = h_i + h_j \leqslant t_i + t_j$.

Dans tous les cas l'inégalité est vérifiée.

Comme `find` consiste à remonter d'un élément jusqu'à sa racine, la complexité est linéaire en la hauteur qui est majorée âr $\log_2(t) \leqslant \log_2(n)$.

Solution de l'exercice 7

```
let find a uf =
   let b = uf.pere.(a) in
   if a = b
   then a
   else begin let c = find b uf in
              uf.pere.(a) <- c;
              c end;;
```

Solution de l'exercice 8 - Existence

Le nombre d'arbre couvrant possibles est fini car on doit choisir $|S|$ arêtes parmi $|A|$ donc le minimum du poids est atteint.

Solution de l'exercice 9 - Implémentation

Comme on a besoin de l'origine lorsqu'on intègre un sommet, on place un triplet dans la file d'attente.

```
let cle (s, t, w) = -w;;
```

```
let prim g =
  let n = taille g in
  let vus = Array.make n false in
  let fp = creeVide (0, 0, 0) in (* On part du sommet 0 *)
  let acm = ref [] in
  while not (estVide fp) do
    let (s, t, w) = extraire fp in
    if not vus.(t)
    then begin vus.(t) <- true;
               acm := (s,t) :: !acm;
               List.iter (fun (u, p) -> ajouter (t, u, p) fp)
                         (voisins g t) end done;
 !acm;;
```

Solution de l'exercice 10 - Preuve

On note $a'_1 = (s_0, s_1)$, $a'_2 = (t_2, s_2)$, …, $a'_{n-1} = (t_{n-1}, s_{n-1})$ les arête ajoutées, dans l'ordre par l'algorithme de Prim ; on a $t_i \in \{s_0, s_1, \ldots, s_{i-1}\} = A_i$ pour tout i ($t_1 = s_0$).

La propriété 2. de l'exercice 2 prouve que toutes les étapes de l'algorithme définissent bien une nouvelle arête a'_i, on a ainsi ajouté $n - 1$ sommets distincts en plus de s_0 et (G, A') est connexe, c'est bien un arbre couvrant.

On considère un arbre couvrant minimal $G_0 = (S, A^{(0)})$.

On va construire pas-à-pas des arbres couvrants minimaux $G_i = (S, A^{(i)})$ tels que $A^{(i)}$ contient les arêtes $a'_1, a'_2, \ldots, a'_i$.

On suppose défini G_{i-1} pour $1 \leqslant i < n$.

- Si $a'_i \in A^{(i-1)}$, on pose $G_i = G_{i-1}$.

- Si $a'_i \notin A^{(i-1)}$ on considère la coupure de premier ensemble A_i : a'_i est une arête traversante pour cette coupure. La propriété 3. de l'exercice 2 prouve qu'on peut remplacer une arête $a_i \in A^{(i-1)}$ traversante pour cette coupure pour obtenir un arbre couvrant $G_i = (S, A^{(i)})$ avec $A^{(i)} = A^{(i-1)} \cup \{a'_i\} \setminus \{a'_i\}$. a_i est différent des a'_k pour $k < i$ car ce sont des arêtes dans A_i donc $A^{(i)}$ contient bien les arêtes (t_k, s_k) pour $k \leqslant i$. De plus, l'algorithme de Prim a choisi une arête traversante de poids minimal donc $w(a'_i) \leqslant w(a_i)$ d'où $w(G_i) \leqslant w(G_{i-1})$. Comme G_{i-1} est minimal, on en déduit que G_i est minimal (on en déduit aussi $w(a'_i) = w(a_i)$).

La dernière étape fournit donc G_{n-1} minimal et égal à (S, A') : (S, A') est minimal.
On notera que si on suppose que les poids des arêtes sont distincts, la démonstration prouve qu'on doit avoir $a'_i = a_i$ pour tout i, l'arbre couvrant minimal est unique.

Solution de l'exercice 11 - Implémentation
On peut trier la liste des arêtes. On utilise la fonction `sort` qui a besoin d'une fonction de comparaison `compare` ; `compare x y` doit renvoyer un entier négatif si $x < y$, nul si $x = y$ et strictement positif si $x > y$.

```
let compare (s1, t1, w1) (s2, t2, w2) =
   if w1 < w2
   then -1
   else begin if w1 = w2
              then 0 else 1 end;;
```

```
let kruskal g =
   let n = taille g in
   let ar = List.sort compare (aretes g) in
   let acm = ref [] in
   let uf = createUF n in
   let traiter (s, t, w) =
      if find uf s <> find uf t
      then begin acm := (s, t) :: !acm;
                 union s t uf end in
   List.iter traiter ar;
   !acm;;
```

Solution de l'exercice 12 - Preuve
La démonstration est similaire au cas de l'algorithme de Prim.