Exercice 1. Implémentation de l'algorithme de Kruskal

Question 1 – On va utiliser un tableau d'entiers de taille `n`. Chaque composante connexe sera identifiée par un entier compris entre `0` et `n-1`. Pour tout sommet `v`, l'entier `cc.(v)` est le numéro de la composante connexe contenant `v`. Pour fusionner les composantes connexes de numéro `i` et `j`, il suffit de parcourir le tableau `cc` et d'y remplacer toutes les occurrences de `j` par `i`. Voici l'évolution du tableau lorsqu'on applique l'exemple de l'énoncé :

Après ligne 1 : `[|0; 1; 2; 3|]`
Après ligne 2 : `[|0; 1; 2; 0|]`
Après ligne 3 : `[|0; 2; 2; 0|]`
Après ligne 4 : `[|2; 2; 2; 2|]`

```
type compo_connexe = int array;;
```

```
let creer (n: int): compo_connexe =
  let f i = i in
  Array.init n f;;
```

```
let meme_compo (cc: compo_connexe) (u: int) (v: int) : bool =
  cc.(u) = cc.(v);;
```

```
(* Définir la variable i avant la boucle est obligatoire car cc.(v)
   va être modifiée lors de l'exécution de la boucle *)
let fusion (cc: compo_connexe) (u: int) (v: int): unit =
  if meme_compo cc u v then failwith "fusion: même composante connexe";
  let i = cc.(u) in
  let j = cc.(v) in
  for w = 0 to Array.length cc - 1 do
    if cc.(w) = j then cc.(w) <- i
  done;;
```

Question 2.a –

```
let liste_aretes (g: graphe): arete list =
  let li = ref [] in
  let n = Array.length g in
  for s1 = 0 to n-1 do
    for s2 = s1 to n-1 do
      if g.(s1).(s2) <> 0 then begin
          let a = {s1 = s1; s2 = s2; poids = g.(s1).(s2)} in
          li := a :: !li;
        end;
    done;
  done;
  !li;;
```

Question 2.b –

```
let comp (a1: arete) (a2: arete): int = a1.poids - a2.poids;;
```

Question 2.c –

```
let kruskal (g: graphe): graphe =
  let n = Array.length g in
  let t = Array.make_matrix n n 0 in
  let cc = creer n in
  let li = liste_aretes g in
  let li = List.sort comp li in

  let rec aux: arete list ->  unit = function
    | [] -> ()
    | a :: q ->
       if not (meme_compo cc a.s1 a.s2) then begin
           fusion cc a.s1 a.s2;
           t.(a.s1).(a.s2) <- a.poids;
           t.(a.s2).(a.s1) <- a.poids;
         end;
       aux q;
  in

  aux li;
  t;;
```

Exercice 2. Algorithme de Prim

Dans la fonction `maj_fp`, on suppose qu'on vient de supprimer un couple `(s0,Some t)` de `fp` et d'ajouter le sommet `s0` et l'arête `{s0, t}` au graphe T. Il s'agit maintenant de mettre à jour la file de priorité en parcourant tous ses éléments et en vérifiant si `s0` devient le sommet le plus proche pour certains des sommets de $S \setminus S'$.

```
let maj_fp (g: graphe) (s0: int) (fp: file_prio): unit =
  let rec aux = function
    | [] -> []
    | (s,t) :: q when g.(s).(s0) = 0 -> (s,t) :: aux q
    | (s,None) :: q -> (s,Some s0) :: aux q
    | (s, Some t) :: q when g.(s).(s0) < g.(s).(t) -> (s, Some s0) :: aux q
    | e :: q -> e :: aux q
  in
  fp := aux !fp;;
```

La fonction `creer` renvoie la file de priorité associée au graphe initial $T = (r, \varnothing)$.

```
let creer (g: graphe) (r: int): file_prio =
  let fp = ref [] in
  let n = Array.length g in
  for s = 0 to n-1 do
    if s <> r then
      fp := (s,None) :: !fp;
  done;
  maj_fp g r fp;
  fp;;
```

La fonction `pull` renvoie l'élément de la file de priorité correspondant à une arête dont le poids est minimum. Cet élément est supprimé de la file de priorité.

```
let pull (g: graphe) (fp: file_prio): int * voisin_plus_proche =
  let rec aux = function
    | [] -> failwith "File de priorité vide"
    | [e] -> e, []
    | (s1,None) :: q1 ->
       let m, q2 = aux q1 in
       m, (s1,None) :: q2
    | (s1, Some t1) :: q1 ->
       let m, q2 = aux q1 in
       begin
         match m with
         | s2, None -> (s1, Some t1), q1
         | s2, Some t2 when g.(s1).(t1) < g.(s2).(t2) -> (s1, Some t1), q1
         | _ -> m, (s1, Some t1) :: q2
       end
  in
  let m, fp2 = aux !fp in
  fp := fp2;
  m;;
```

```
(* Suppose que le graphe est connexe et contient au moins un sommet *)
let prim (g: graphe) =
  let n = Array.length g in
  let arbre_couvrant = Array.make_matrix n n 0 in
  let r = 0 in
  let fp = creer g r in
  for i = 0 to n-2 do
    let s,t = pull g fp in
    print_int s; print_string " ";
    match t with
    | None -> failwith "Le graphe n'est pas connexe"
    | Some t ->
       arbre_couvrant.(s).(t) <- g.(s).(t);
       arbre_couvrant.(t).(s) <- g.(t).(s);
       maj_fp g s fp;
  done;
  arbre_couvrant;;
```

Exercice 3. Preuve de la validité de l'algorithme de Kruskal

Question 1 –

Preuve du lemme 1. Soit :

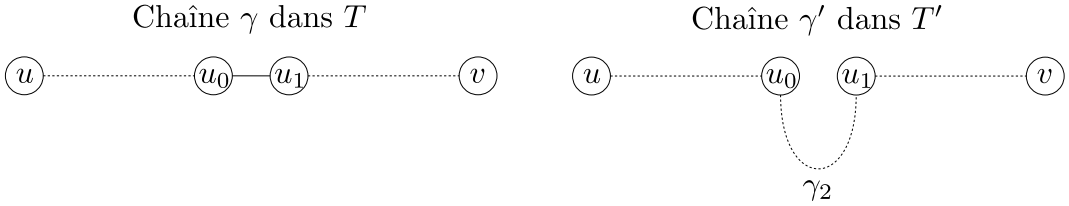
$$\mathcal{A} = \Big\{ G' = (S, A') : G' \text{ est un graphe connexe avec } A' \subset A \Big\}$$

On a $\mathcal{A} \neq \varnothing$ car $G \in \mathcal{A}$ et $\mathcal{A}$ est fini. On peut donc définir T l'élément de $\mathcal{A}$ dont le nombre d'arêtes est minimal. Montrons que T est un arbre couvrant de G. Comme $T \in \mathcal{A}$, il suffit de montrer que T ne contient pas de cycle. Par l'absurde, on suppose que T contient un cycle $\gamma = (u_0, u_1, \ldots, u_k)$ avec $u_k = u_0$ et $k > 0$. Soient :

$$\begin{cases} \gamma_1 = (u_1, u_2, \ldots, u_{k-1}, u_k) \\ \gamma_2 = (u_k, u_{k-1}, \ldots, u_2, u_1) \end{cases}$$

alors γ_1 (resp. γ_2) relie u_1 à u_0 (resp. u_0 à u_1) sans emprunter l'arête $\{u_0, u_1\}$ (rappelons que par définition, un cycle ne passe pas deux fois par la même arête). Soit T' le graphe obtenu lorsqu'on supprime de T l'arête $\{u_0, u_1\}$. Montrons que T' est connexe, ce qui contredirait la minimalité de T parmi les éléments de $\mathcal{A}$.

Soient u et v deux sommets. T est connexe donc il existe une chaîne γ de u à v dans T. On peut alors définir une chaîne γ' de u à v dans T' : à chaque fois que l'arête $\{u, v\}$ est empruntée dans γ, on la remplace par γ_1 ou γ_2 (en fonction du sens dans lequel l'arête est empruntée). La chaîne γ' obtenue relie u à v dans T'.



En conclusion, T' et connexe, donc $T' \in \mathcal{A}$ et cela contredit la définition de T (T est un élément de $\mathcal{A}$ dont le nombre d'arêtes est minimal). Finalement, T ne contient pas de cycle.

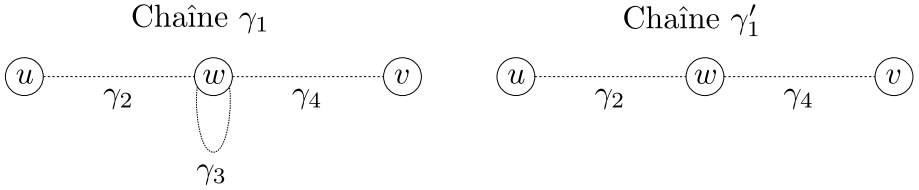
Preuve du lemme 3. Soit :

$$\mathcal{C} = \Big\{ \gamma : \gamma \text{ est une chaîne entre } u \text{ et } v \Big\}$$

On a $\gamma_0 \in \mathcal{C}$ donc $\mathcal{C} \neq \varnothing$ et $\mathcal{C}$ est fini. Soit $\gamma_1 \in \mathcal{C}$ dont le nombre d'arêtes noté $|\gamma_1|$ est minimal parmi les éléments de $\mathcal{C}$. Montrons que γ_1 est élémentaire. Supposons par l'absurde que γ_1 contient deux fois le même sommet noté w. Alors γ_1 est de la forme $\gamma_1 = \gamma_2 + \gamma_3 + \gamma_4$ où :

- $+$ désigne la concaténation entre chaînes.
- γ_2 est une chaîne de u à w.
- γ_3 est une chaîne de w à w.
- γ_4 est une chaîne de w à v.

Soit $\gamma_1' = \gamma_2 + \gamma_4$ alors $\gamma_1' \in \mathcal{C}$ et $|\gamma_1'| < |\gamma_1|$ ce qui contredit la minimalité de $|\gamma_1|$ parmi les éléments de $\mathcal{C}$.



On a une contradiction, donc γ_1 est élémentaire.

Preuve du lemme 4. Soit γ_1 une chaîne de u à v dans T. Par le lemme 3, il existe une chaîne élémentaire γ_2 de u à v dans T. En concaténant γ_2 avec l'arête $\{v, u\}$, on obtient un cycle de T. Notez en particulier que ce cycle ne passe pas deux fois pas la même arête comme le nécessite la définition de cycle.

Preuve du lemme 6. Soient $(w_1, w_2) \in S^2$ deux sommets. Il s'agit de montrer qu'il existe une chaîne entre w_1 et w_2 dans G_3. Par hypothèse, il existe une chaîne de w_1 à w_2 dans G_1. Par le lemme 3, il existe donc une chaîne élémentaire γ entre w_1 et w_2 dans G_1. Si γ ne passe pas par l'arête $\{u_1, v_1\}$ alors c'est une chaîne de G_2 et donc de G_3. Sinon, elle est de la forme $\gamma = \gamma_1 + \gamma_2 + \gamma_3$ où :

- $+$ désigne la concaténation de chaînes.
- γ_2 est une chaîne de longueur 1 qui consiste à emprunter l'arête $\{u, v\}$. Dans la suite, on suppose que cette arête est empruntée de u vers v (l'autre cas est symétrique).
- γ_1 est une chaîne de w_1 à u_1.
- γ_3 est une chaîne de v_1 à w_2.

Comme $u_2 \in CC_{G_2}(u_1)$ et $v_2 \in CC_{G_2}(c_1)$, il existe deux chaînes γ_4 et γ_5 de u_1 à u_2 et de v_2 à v_1 dans G_2. Alors $\gamma_1 + \gamma_4 + (u_2, v_2) + \gamma_5 + \gamma_3$ est une chaîne de w_1 à w_2 dans G_3.

Preuve du lemme 7. Par l'absurde, supposons que G_2 contient un cycle. Si ce cycle ne contient pas l'arête $\{u_1, v_1\}$ alors c'est aussi un cycle de G_1 ce qui est absurde. Sinon, ce cycle passe exactement une fois par l'arête $\{u_1, v_1\}$. En supprimant cette arête du cycle, on obtient une chaîne γ de u_1 à v_1 dans G_2 qui ne passe par l'arête $\{u_1, v_1\}$. γ est donc une chaîne de entre u_1 et v_1 dans G_1 ce qui contredit l'hypothèse.

Question 2.a – On a $A_0 = \varnothing$. Il suffit donc de montrer qu'il existe un arbre couvrant de poids minimal pour G. D'après le lemme 1, il existe au moins un arbre couvrant pour G. Puisque le nombre d'arbres couvrants est fini, il existe un arbre couvrant de poids minimal pour G.

Question 2.b – On suppose $(\mathcal{P}_i)$ et on montre $(\mathcal{P}_{i+1})$. Soit $T = (S, B)$ l'arbre couvrant donné par $(\mathcal{P}_i)$ et $\{u, v\}$ l'arête ajoutée pour passer de T_i à T_{i+1} :

$$A_{i+1} = A_i \cup \{\{u, v\}\}$$

Si l'arête $\{u, v\}$ appartient à B alors $A_{i+1} \subset B$ et $(\mathcal{P}_{i+1})$ est vraie. On se place maintenant dans le cas où $\{u, v\} \notin B$. D'après la description de l'algorithme, u et v ne sont pas dans la même composante connexe dans T_i. Comme T est connexe, par le lemme 3, il existe une chaîne simple γ de u à v dans T. Il existe donc au moins une arête $\{u', v'\}$ de γ tel que u' et v' ne sont pas dans la même composante connexe dans T_i. On définit $T' = (S, B')$ le graphe égal à T dans lequel on a supprimé l'arête $\{u', v'\}$:

$$B' = B \setminus \{\{u', v'\}\}$$

Alors, mis à part l'arête $\{u', v'\}$, toutes les arêtes de la chaîne simple γ apparaissent dans B'. On a donc :

$$u \in CC_{T'}(u') \qquad \text{et} \qquad v \in CC_{T'}(v')$$

On définit $T'' = (S, B'')$ le graphe égal à T' dans lequel on a ajouté l'arête $\{u, v\}$:

$$B'' = B' \cup \{\{u, v\}\}$$

Alors A_{i+1} est inclus dans B'. Il reste à montrer que T' est un arbre couvrant de poids minimal.

Le poids de T' noté $\omega(T')$ vérifie :

$$\omega(T') = \omega(T) + \omega(u, v) - \omega(u', v').$$

D'après la description de l'algorithme, $\{u, v\}$ est de poids minimal parmi les arêtes entre 2 sommets qui ne sont pas reliés dans T_i. Comme u' et v' ne sont pas reliés dans T_i, on a :

$$\omega(u, v) \leqslant \omega(u', v')$$

Donc :

$$\omega(T') \leqslant \omega(T)$$

On applique le lemme 6, avec :

$G_1 = T$ $\quad$ $u_1 = u'$ $\quad$ $v_1 = v'$ $\quad$ $G_2 = T'$
$u_2 = u$ $\quad$ $v_2 = v$ $\quad$ $G_3 = T''$

Le lemme 6 nous donne que T'' est connexe.

On peut appliquer le lemme 7, avec :

$G_1 = T'$ $\quad$ $u_1 = u$ $\quad$ $v_1 = v$ $\quad$ $G_2 = T''$

Le lemme 7 nous donne que T'' est connexe (notez que u et v ne sont pas dans la même composante connexe dans T' par le lemme 4).

Question 2.c – On se place après la boucle. On a donc $(\mathcal{P}_k)$.

Montrons que T_k est connexe. Par l'absurde, supposons qu'il existe deux sommets u, v qui ne sont pas reliés dans T_k. Comme G est connexe, il existe une chaîne γ de u à v dans G. Il existe donc une arête $\{u', v'\}$ de γ telle que u' et v' ne sont pas dans la même composante connexe dans T_k. On en déduit que pour tout i, u' et v' ne sont pas dans la même composante connexe dans T_i. En particulier, lorsque l'arête $\{u', v'\}$ a été considérée par l'algorithme de Kruskal, elle aurait dû être ajoutée à T ce qui constitue une contradiction.

Finalement, T_k est connexe et inclus dans un arbre couvrant de poids minimal de G noté T. Si $T \neq T_k$ alors par le lemme 4, T contient un cycle ce qui est absurde puisque T est un arbre.

En conclusion, $T = T_k$ et l'algorithme de Kruskal renvoie un arbre couvrant de poids minimal.