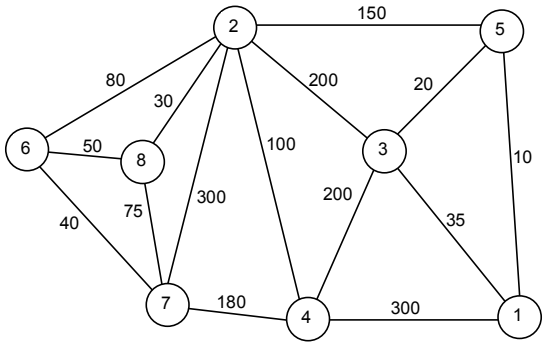
Corrigé E.D. Algorithmes et Structures de Données n° 6

Thème : Algorithme de Kruskal
(recherche d'un arbre couvrant de poids minimal dans un graphe connexe)

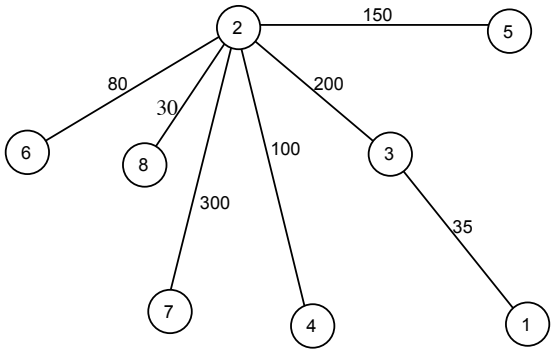
Exercice VI.1

Le but de cet exercice est d'étudier un algorithme de bout en bout, c'est-à-dire de sa conception en "pseudo-langage" jusqu'à un programme ADA.

Question 1 L'algorithme de Kruskal



Voici un exemple d'arbre couvrant (sous-ensemble de n-1 arêtes tel que le graphe reste connexe) qui n'est pas de poids minimal :



Il s'agit d'un arbre couvrant de poids 80+30+300+100+200+35+150=895.
Bien sûr, il existe des arbres couvrants de poids plus faible (il suffit par exemple de prendre l'arête [1,5] et supprimer [2,5] : on diminue le poids total de 140 (10 au lieu de 150).

Si on applique l'algorithme de Kruskal :

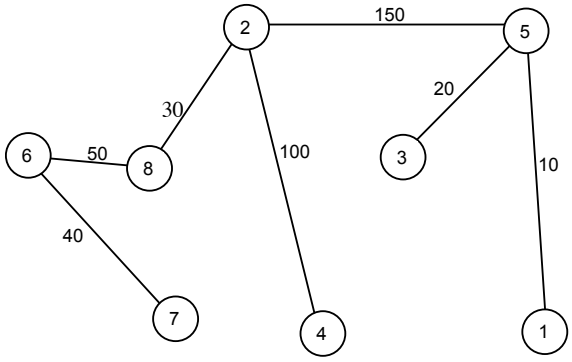
Tri des arêtes par poids croissant :

arête	[1,5]	[3,5]	[2,8]	[3,1]	[6,7]	[6,8]	[7,8]	[2,6]	[2,4]	[2,5]	[4,7]	[3,4]	[2,3]	[1,4]	[2,7]
poids	10	20	30	35	40	50	75	80	100	150	180	200	200	300	300

D'après l'algorithme :

On peut choisir [1,5] puis [3,5] puis [2,8] mais pas [3,1] (qui formerait un cycle avec [1,5] et [3,5]).
Puis on peut choisir [6,7] puis [6,8] mais pas [7,8] ni [2,6].
Puis on peut choisir [2,4] puis [2,5].

On s'arrête ici car le nombre d'arêtes choisies est 7 (c'est à dire n-1, puisque n = 8).



L'arbre obtenu est de poids **400**.

Il y a deux difficultés dans la programmation de l'algorithme de Kruskal :

- Le tri des arêtes : nous allons le faire par la méthode du "tri-rapide" (on pourrait utiliser un autre tri, mais celui-ci est bon)
- Savoir tester si une arête choisie ne forme pas de cycle avec les précédentes.

Question 2 Le tri rapide

2.1. Un exemple
Soit le tableau tab suivant donnant les arêtes du graphe et leur poids dans un ordre quelconque :

150	200	100	300	80	30	50	40	75	180	20	200	35	10	300
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Appliquer l'algorithme de tri-rapide pour trier les arêtes par ordre de poids croissants.

Au départ, tri-rapide(tab, 1,15)
pivot =150
i=1, j=15
avancer i et reculer j puis échanger leur contenu :

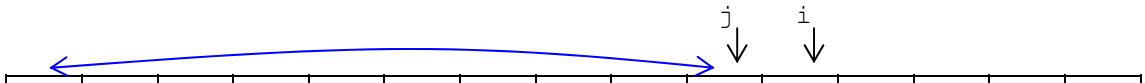
	i ↓												j ↓	
150	200	100	300	80	30	50	40	75	180	20	200	35	10	300
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

150	10	100	300	80	30	50	40	75	180	20	200	35	200	300
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Puis

150	10	100	35	80	30	50	40	75	180	20	200	300	200	300
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Puis



150	10	100	35	80	30	50	40	75	20	180	200	300	200	300
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

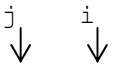
20	10	100	35	80	30	50	40	75	**150**	180	200	300	200	300
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

150 est maintenant à sa place définitive.

On va maintenant exécuter tri-rapide(tab, 1, 9) et tri-rapide(tab, 10, 15)

Voici tri-rapide(tab, 1, 9)

20	10	100	35	80	30	50	40	75
1	2	3	4	5	6	7	8	9



20	10	100	35	80	30	50	40	75
1	2	3	4	5	6	7	8	9

10	**20**	100	35	80	30	50	40	75
1	2	3	4	5	6	7	8	9

Exécute à son tour tri-rapide(tab, 1,1) et tri-rapide(tab, 3,9)

Etc …

2.2. L'algorithme

Ecrire en pseudo-langage l'algorithme du tri-rapide.

```
procedure tri_rapide (tab : in-out tableau de t_cle, g, d : indice)
      -- trie la partie g..d du tableau
      i, j : indice;
      pivot : t_cle;
debut
      si g<d alors
            i := g;
            j := d;
            pivot := tab(g);
            tant que (i<j) faire
                  i := i + 1;
                  tant que ((tab(i)<pivot) et (i<j)) faire i:= i+1; fait
                  tant que (tab(j) > pivot) faire j := j - 1; fait
                  si (i < j) alors echanger (tab, i, j); fin si
            fait
            echanger (g, j);
            tri_rapide (tab, g, j - 1);
            tri_rapide (tab, j + 1, d);
      fin si
fin
```

Question 3 Détection des cycles

3.1. Le principe

Comment peut-on détecter qu'une arête sélectionnée forme un cycle avec les précédentes ? (utiliser la notion de connexité).

Pour ce faire, on peut considérer qu'un sous-ensemble d'arêtes choisies induit une partition de l'ensemble des sommets du graphe :

- au départ : aucune arête n'est choisie : partition en singletons (on a n sous-ensembles, chaque sous-ensemble correspond à un sommet)
- puis, à chaque fois qu'une arête est choisie, on fusionne les sous-ensembles auxquels appartiennent les deux extrémités de l'arête choisie.

Au moment du choix d'une arête, on vérifie qu'elle relie des sommets appartenant à des ensembles différents, on est ainsi sûr que la nouvelle arête choisie ne formera pas de cycle.

3.2. L'algorithme

Les sous-ensembles (cc) seront représentés par des listes. A chaque sommet est associé une liste qui, au départ, contient uniquement ce sommet.

Pour fusionner deux ensembles, associés à 2 sommets i et j, on vide la liste de l'un des sommets, par exemple j et on ajoute l'ancienne liste de j à la liste de i. Il faut alors indiquer que j se trouve dans le même sous-ensemble que i. On le fait par l'intermédaire d'une variable associée à j (`meme_cc_que`).

Q1) Comment tester si 2 sommets x et y sont dans le même ensemble ?

On suppose que l'on dispose de deux fonctions sur les listes : `est_vide` qui retourne VRAI si et seulement si la liste passée en paramètre est vide et la fonction `appartient` qui teste si un élément appartient ou non à une liste.

D'abord, il faut trouver le sous-ensemble auquel appartient x. On regarde `tab_cc(x).cc`, si la liste est vide, on remplace x par `z = tab_cc(x).meme_cc_que`. Si z = y, alors la réponse est oui, sinon, on regarde `tab_cc(z).cc`, etc...

D'où :

```
z := x;
tant que (est_vide(tab_cc(z).cc)) et z • y faire
      z := tab_cc(z).meme_cc_que
fait
si y = z
      alors retourner VRAI;
      sinon retourner appartient (y, tab_cc(z).cc);
fin si;
```

Q2) Comment fusionner 2 ensembles (cc) associés à 2 sommets x et y ?
On suppose que l'on dispose d'une procédure `fusionner_liste(l1, l2)` qui ajoute à la première liste `l1` les éléments de la seconde `l2`, et d'une procédure `vider_liste(l)` qui vide la liste `l` passée en paramètre.

On trouve la liste où est x. Soit zx tel que `tab_cc(zx).cc` n'est pas vide et contient x. On trouve la liste où est y. Soit zy tel que `tab_cc(zy).cc` n'est pas vide et contient y. On ajoute `tab_cc(zy).cc` à `tab_cc(zx).cc`; on vide `tab_cc(zy).cc` et on indique que zy est dans la même cc que zx.

```
fonction trouver_chef_cc (x : entier; tab_cc : t_tab_comp_connexe)
return entier
      -- trouve le "chef" de la composante connexe à laquelle x
appartient
      z : entier
debut
      z := x;
      tant que (est_vide(tab_cc(z).cc)) faire
            z := tab_cc(z).meme_cc_que
      fait
fin trouver_chef_cc

procedure fusionner_2_cc (x, y : entier; tab_cc : t_tab_comp_connexe)
      zx, zy : entier;
debut
      zx := trouver_chef_cc(x, tab_cc);
      zy := trouver_chef_cc(y, tab_cc);
      fusionner_liste (tab_cc(zx).cc, tab_cc(zy).cc);
      vider_liste (tab_cc(y).cc);
      tab_cc(zy).meme_cc_que := zx;
fin fusionner_2_cc
```