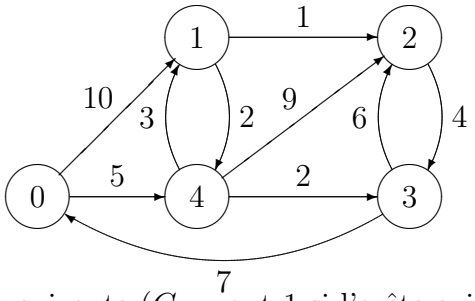
Graphes valués : corrigé — T.D.7.

Exercice 1



1. La matrice d'adjacence est la suivante ($G_{i,j}$ vaut 1 si l'arête existe et 0 sinon).

$$G = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

On définit en OCaml le graphe sous forme d'un tableau de listes d'adjacence en écrivant

```
let g=[| [1;4] ; [2;4] ; [3] ; [0;2] ; [1;2;3] |] ;;
```

2. Algorithme de Dijkstra mis en œuvre sur l'exemple.

- Etat initial : on est 0 et on ne voit que ses voisins directs.

Sommet	0	1	2	3	4
λ	$\underline{0}$	10	∞	∞	5

- Le minimum dans V est 5, on se déplace vers le sommet 4 et on met à jour le tableau pour ses voisins.

Sommet	0	1	2	3	4
λ	$\underline{0}$	8	14	7	$\underline{5}$

- Le minimum dans V est 7 et on se déplace vers le sommet 3.

Sommet	0	1	2	3	4
λ	$\underline{0}$	8	13	$\underline{7}$	$\underline{5}$

- Le minimum dans V est 8 et on se déplace vers le sommet 1.

Sommet	0	1	2	3	4
λ	$\underline{0}$	$\underline{8}$	9	$\underline{7}$	$\underline{5}$

- On n'a plus qu'un sommet qui ne peut rien améliorer. Notre table λ est correcte.

3. On gère une matrice de distances qui est initialement

$$\begin{pmatrix} 0 & 10 & \infty & \infty & 5 \\ \infty & 0 & 1 & \infty & 2 \\ \infty & \infty & 0 & 4 & \infty \\ 7 & \infty & 6 & 0 & \infty \\ \infty & 3 & 9 & 2 & 0 \end{pmatrix}$$

- On se permet de transiter par 0

$$\begin{pmatrix} 0 & 10 & \infty & \infty & 5 \\ \infty & 0 & 1 & \infty & 2 \\ \infty & \infty & 0 & 4 & \infty \\ 7 & 17 & 6 & 0 & 12 \\ \infty & 3 & 9 & 2 & 0 \end{pmatrix}$$

- On se permet de transiter par 1

$$\begin{pmatrix} 0 & 10 & 11 & \infty & 5 \\ \infty & 0 & 1 & \infty & 2 \\ \infty & \infty & 0 & 4 & \infty \\ 7 & 17 & 6 & 0 & 12 \\ \infty & 3 & 4 & 2 & 0 \end{pmatrix}$$

- On se permet de transiter par 2

$$\begin{pmatrix} 0 & 10 & 11 & 15 & 5 \\ \infty & 0 & 1 & 5 & 2 \\ \infty & \infty & 0 & 4 & \infty \\ 7 & 17 & 6 & 0 & 12 \\ \infty & 3 & 4 & 2 & 0 \end{pmatrix}$$

- On se permet de transiter par 3

$$\begin{pmatrix} 0 & 10 & 11 & 15 & 5 \\ 12 & 0 & 1 & 5 & 2 \\ 11 & 21 & 0 & 4 & 16 \\ 7 & 17 & 6 & 0 & 12 \\ 9 & 3 & 4 & 2 & 0 \end{pmatrix}$$

- On se permet de transiter par 4

$$\begin{pmatrix} 0 & 8 & 9 & 7 & 5 \\ 11 & 0 & 1 & 4 & 2 \\ 11 & 19 & 0 & 4 & 16 \\ 7 & 15 & 6 & 0 & 12 \\ 9 & 3 & 4 & 2 & 0 \end{pmatrix}$$

La matrice contient à l'intersection de la ligne i et de la colonne j la plus courte distance du sommet i au sommet j.

Exercice 2

a) Dans le tableau ci-dessous, on visualise l'évolution des valeurs du tableau d au cours de l'algorithme. Les différents tests de la deuxième partie de celui-ci reviennent à itérer une fois de plus les valeurs de d pour voir si l'une d'entre-elles est susceptible d'être encore modifiée ; voila pourquoi on applique la boucle une fois de plus dans le tableau.

k	d_1	d_2	d_3	d_4	d_5
	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
1	0	6	$+\infty$	$+\infty$	7
2	0	6	4	2	7
3	0	2	4	2	7
4	0	2	4	-2	7
	0	2	4	-2	7

Les valeurs du tableau d ne sont plus modifiée, l'algorithme se termine en renvoyant la valeur « Vrai ».
En revanche, si l'arête $4 \to 5$ est de poids 8, la dernière ligne de ce tableau est modifiée :

k	d_1	d_2	d_3	d_4	d_5
	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
1	0	6	$+\infty$	$+\infty$	7
2	0	6	4	2	7
3	0	2	4	2	7
4	0	2	4	-2	7
	0	2	4	-2	6

et l'algorithme renvoie donc la valeur « Faux ». On peut observer l'existence d'un cycle de poids négatif dans ce cas : $(2, 4, 5, 3, 2)$.

b) A l'instar de l'algorithme de Dijkstra, on démontre par récurrence qu'à l'étape k, d_u est égal au poids d'un chemin minimal allant de la source au sommet u sans passer par plus de k sommets. Sachant qu'un chemin minimal est de longueur au plus $n-1$, à la fin de la première partie de l'algorithme chaque valeur d_u est égale à $\delta(s,u)$ s'il n'y a pas de cycle de poids strictement négatif. Ainsi on aura pour tout u et v : $d_v \leqslant d_u + w(u,v)$ et la valeur retournée sera « Vrai ».

c) Supposons l'existence d'un cycle $(v_0, \dots, v_k)$ de poids strictement négatif avec $v_k = v_0$. Nous avons donc :

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0.$$

Raisonnons par l'absurde en supposant que la réponse retournée par l'algorithme soit « Vrai ». Alors pour tout $i \in [\![1,k]\!]$, $d_{v_i} \leqslant d_{v_{i-1}} + w(v_{i-1}, v_i)$. En sommant ces inégalités on obtient :

$$\sum_{i=1}^{k} d_{v_i} \leqslant \sum_{i=1}^{k} d_{vi-1} + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

soit en simplifiant : $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geqslant 0$, ce qui est absurde.

La réponse retournée dans le cas où il existe un cycle de poids strictement négatif est donc « Faux ».

d) Posons $n = |\mathrm{V}|$ et $p = |\mathrm{E}|$. L'initialisation du tableau d a un coût temporel en $\Theta(n)$; le corps principal de la fonction est en $\Theta(np)$ et la détermination de l'existence d'un cycle de poids strictement négatif est un $\mathrm{O}(p)$ donc le coût total de cet algorithme est un $\Theta(np)$.

Exercice 3

Raisonnons par l'absurde en supposant qu'un tel arbre n'existe pas, et considérons un arbre couvrant (V, B) tel que $|A \cap B|$ soit maximal. On considère alors une arête $(a, b) \in A \mid (a, b) \notin B$.
Puisque (V, B) est un arbre couvrant, il existe un chemin $a \rightsquigarrow b$ n'empruntant que des arêtes de B. Ce chemin contient au moins une arête (c, d) qui n'appartient pas à A, car dans le cas contraire A contiendrait un cycle. On pose alors $B' = B \setminus \{(c, d)\} \cup \{(a, b)\}$. (V, B') est toujours un arbre couvrant mais $|A \cap B'| > |A \cap B|$, ce qui est absurde.

Une autre solution consiste à affecter un poids à chacune des arêtes en posant $w(e) = 1$ si $e \in A$ et $w(e) = 2$ sinon, puis à appliquer l'algorithme de Kruskal. Celui-ci va sélectionner chacune des arêtes de A puis les compléter pour retourner un arbre couvrant (minimal pour cette répartition des poids).

Exercice 4 On parcourt le graphe en profondeur. Il admet un cycle si et seulement si, au moment d'ajouter un voisin on retombe sur un sommet déjà traité , autre que son prédécesseur. On garde en mémoire pour cela un tableau des prédécesseurs, mis à jour quand un voisin est ajouté.

```
let cycle g =
    let t = Array.make (Array.length g) false in
    t.(0) <- true;
    let pred = Array.make (Array.length g) (-1) in
    let bool = ref false in
    let rec traitement li = match li with
        | []     -> ()
        | x :: q -> traitement (voisins x g.(x) @ q)
    and voisins x li = match li with
        | []                -> []
        | y :: q when t.(y) -> if pred.(x) = y then voisins x q
                               else (bool := true; [])
        | y :: q            -> pred.(y) <- x; t.(y) <- true;
                               y :: voisins x q in
    traitement [0];
    !bool;;
```
