Graphes et algorithme de Dijkstra

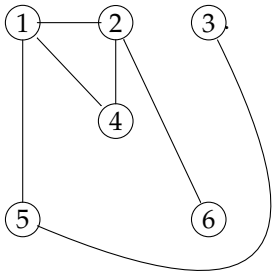
1 Définition

DÉFINITION

On appelle *graphe* (simple) non-orienté tout couple $G = (S, A)$ où S est un ensemble, appelé *ensemble des sommets de G*, et A un ensemble de paires d'éléments de A distincts, appelé *ensemble des arêtes de G*.

Les graphes seront représentés graphiquement, les sommets étant représentés par leurs noms, et les arêtes par des traits reliant les sommets.

EXEMPLE



DÉFINITION

Soit $G = (S, A)$ un graphe. Soit $a = \{x, y\}$ une arête de G.

- x et y sont appelés *extrémités de a*
- x et y sont dits *voisins* ou *adjacents*

On appelle alors *degré de x* le nombre de voisins du sommet x, noté $d(x)$ (ou $d_G(x)$ s'il peut y avoir une confusion).

DÉFINITION

Soit $G = (S, A)$ un graphe. On appelle *chemin de longueur k reliant x et y* toute suite finie $x_0 = x, x_1, \ldots, x_{k-1}, x_k = y$ de sommets tels que pour tout i, $\{x_i, x_{i+1}\} \in A$.
Si $x = y$, et que tous les autres x_i sont distincts, on dit que le chemin est un *cycle*.
Un chemin est dit *élémentaire* si tous les sommets le composant sont distincts.
Pour tous sommets x et y, on appelle *distance de x à y* la longueur du plus petit chemin reliant x et y s'il existe, et ∞ sinon ; on la note $d(x, y)$.
Un graphe tel que tout couple de sommets à une distance finie est dit *connexe*.

DÉFINITION

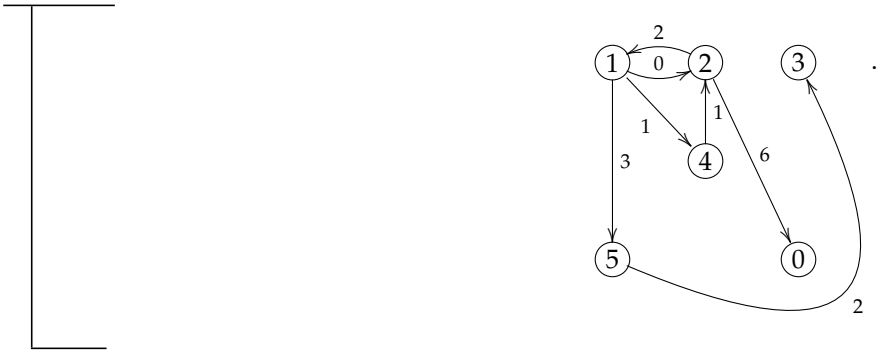
Un *graphe pondéré* est un triplet $G = (S, A, p)$ où (S, A) est un graphe, et p une fonction de A dans $\mathbb{R}$. L'image d'une arête par p est appelée son *poids*.

DÉFINITION

On appelle *graphe (simple) orienté* tout couple $G = (S, A)$ ou S est un ensemble et A un ensemble de couples d'éléments de S distincts.

On les représentera comme les graphes non orientés, mais en remplaçant les arêtes par des flèches.

EXEMPLE



2 Implémentation

Dans la suite, on supposera qu'on travaille avec des graphes orientés, pondérés et connexes.

On supposera toujours que les sommets des graphes considérés sont numérotés de 0 à n.

En Python, on pourra implémenter les graphes de deux façons :

- en utilisant la matrice d'adjacence du graphe, c'est-à-dire en construisant une matrice de $\mathcal{M}_{n+1}(\mathbb{R})$ telle que

$$m_{i,j} = \begin{cases} p(i,j) & \text{si } (i,j) \in A \\ -1 & \text{sinon} \end{cases}.$$

- en utilisant des listes d'adjacence, c'est-à-dire en construisant un dictionnaire `adj` de $n+1$ éléments, telle que la clef `i` de `adj` contienne la liste des couples `(s,p)` où `s` est voisin de `i` et `p` le poids de l'arête reliant `i` et `j`.

Exercice 1. Donner la représentation par matrice d'adjacence et par liste d'adjacence du graphe de l'exemple, puis les coder en Python.

Exercice 2. Écrire des fonctions `matriceVersListe` et `listeVersMatrice` passant d'une représentation à l'autre.

On travaillera maintenant avec des graphes sous forme de liste d'adjacence.

Exercice 3. Écrire une fonction `voisins` prenant en paramètre un graphe, un sommet du graphe, et renvoyant la liste des voisins de ce sommet.

Exercice 4. Écrire une fonction `cheminValide` prenant en paramètre un graphe et une liste de sommets `[s1,...,sp]` du graphe, et vérifiant que le chemin `s1`$\to \ldots \to$ `sp` est bien un chemin valide.

3 Parcours du graphe

Il existe deux manières de parcourir un graphe : les parcours en largeur, et en profondeur.

Exercice 5. Rappeler le code de ces deux types de parcours.

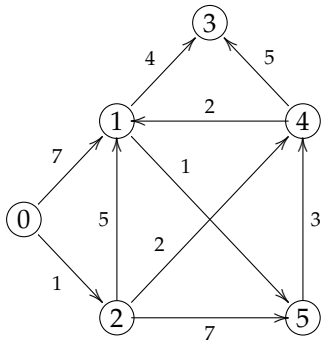
4 Algorithme de Dijkstra

Un problème important sur les graphes est de pouvoir trouver un chemin de poids minimal entre deux sommets donnés.

L'idée principale des algorithmes faisant cette recherche est le lemme suivant :

Lemme 4.1
S'il existe un plus court chemin entre a et b qui passe par c, alors les sous-chemins de a à c et de c à b sont aussi des plus courts chemins.

On travaillera sur l'exemple suivant :



L'algorithme de Dijkstra permet, en partant d'un sommet donné, de trouver les chemins de longueurs minimales entre le sommet de départ et *tous* les autres sommets. Il fonctionne en remplissant petit à petit une liste de distances à $n+1$ cases, qu'on notera d dans la suite. On suppose qu'on part du sommet 0.

- au départ, on a $d(0) = 0$ et pour tous les autres sommets s, $d(s) = \infty$.
- pour tous les voisins s de 0, on pose $d(s) = p(0, s)$.
  Attention, ce ne sont pas les distances minimales définitives, il faut pour cela attendre la fin de l'algorithme.
- on ajoute le sommet 0 à une liste des sommets déjà visites, v. On a donc $v = [0]$.
- on choisit alors dans la liste des sommets un sommet s non visite dont la distance est minimale : on est sûr que la distance de *ce* sommet est effectivement la distance minimale.
- on l'ajoute à la liste des sommets visites, et pour tous ses voisins t non visites, on pose
$$d(t) = \min(d(t), d(s) + p(s, t).$$
- on itère alors le processus jusqu'à ce que tous les sommets aient été visités.
- le tableau des distances contient alors a la liste des distances minimales en partant de 0.

On donne le code en pseudo-code :

```
fonction Dijkstra(graphe,s):
  Visite = {s}
  Pour tous v dans Sommets:
    d(v) <- poids(s,v)
  Tant que Visite != Sommets:
    Choisir u non visite de d(v) minimal
    Visite = Visite + {u}
    Pour tout v non visite:
    d(v) <- min(d(v), d(u)+poids(u,v))
```

Exercice 6. Dérouler l'algorithme de Dijkstra, en partant de 0, pour le graphe de l'exemple.

On pourra colorier les sommets visités.

Exercice 7. Implémenter l'algorithme de Dijkstra en Python.

Un problème dans cet algorithme est qu'on récupère effectivement la liste des distances minimales, mais pas les chemins correspondants. Une toute petite modification de l'algorithme permet cependant d'y remédier : en plus du tableau d, on ajoute un tableau o des origines.

- au départ, on a $o(s) = -1$ pour tous les sommets
- pour tous les voisins de 0, on pose $o(s) = 0$.
- le tableau o nous permet alors de savoir par quels sommets on est passés. On l'actualise en faisant, lors de la cinquième étape de l'algorithme
$$o(t) = \begin{cases} o(t) & \text{si } d(t) \leqslant d(s) + p(s,t) \\ s & \text{sinon} \end{cases}.$$

L'algorithme doit alors renvoyer les deux listes d et o. Pour trouver un chemin de longueur minimale entre 0 et s, il suffit alors de "remonter" les sommets du tableau o : $s \leftarrow o(s) \leftarrow o(o(s)) \leftarrow \dots$ jusqu'à retomber sur 0.

Exercice 8. Implémenter cette modification.

Exercice 9. Écrire enfin une fonction `distanceMin` prenant en paramètre un graphe et deux sommets du graphe, et renvoyant un chemin de longueur minimale entre i et j, ainsi que la dite longueur.