Théorie des graphes

Table des matières

1 Introduction **5**

1.1 Vocabulaire . . . 5

1.2 Problèmes liés aux graphes . . . 9

1.3 Représentation sur ordinateur . . . 15

1.3.1 Matrice d'adjacence . . . 15

1.3.2 Liste d'adjacence . . . 17

2 Plus court chemin (Problème 1) **19**

2.1 Quelques définitions . . . 19

2.2 Algorithme de Dijkstra . . . 24

2.2.1 Principe . . . 24

2.2.2 Correction . . . 25

2.2.3 Complexité . . . 25

2.3 Algorithmes de Bellman-Ford et de Floyd-Marshall . . . 26

Chapitre 1

Introduction

1.1 Vocabulaire

Définition 1.1 – Graphe simple

On appelle *graphe simple* la donnée d'un nombre fini d'objets S, appelés *sommets* et d'un sous-ensemble $A \subset \mathcal{P}(S)$, définissant les *arêtes*, où

$$\forall \alpha \in A,\ \mathrm{Card}(\alpha) = 2.$$

Exemple 1.1 – Graphe simple

Si on prend

$$S = \{a, b, c, d\} \text{ et } A = \big\{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{c, d\}\big\}$$

on obtient le premier graphe de la session 1.1, de la présente page.

Définition 1.2 – Graphe orienté

On appelle *graphe orienté* la donnée d'un nombre fini d'objets S, appelés *sommets* et d'un sous-ensemble $A \subset S \times S$, définissant les *arcs*, où

$$\forall (x, y) \in A,\ x \neq y.$$

Exemple 1.2 – Graphe orienté

Si on prend

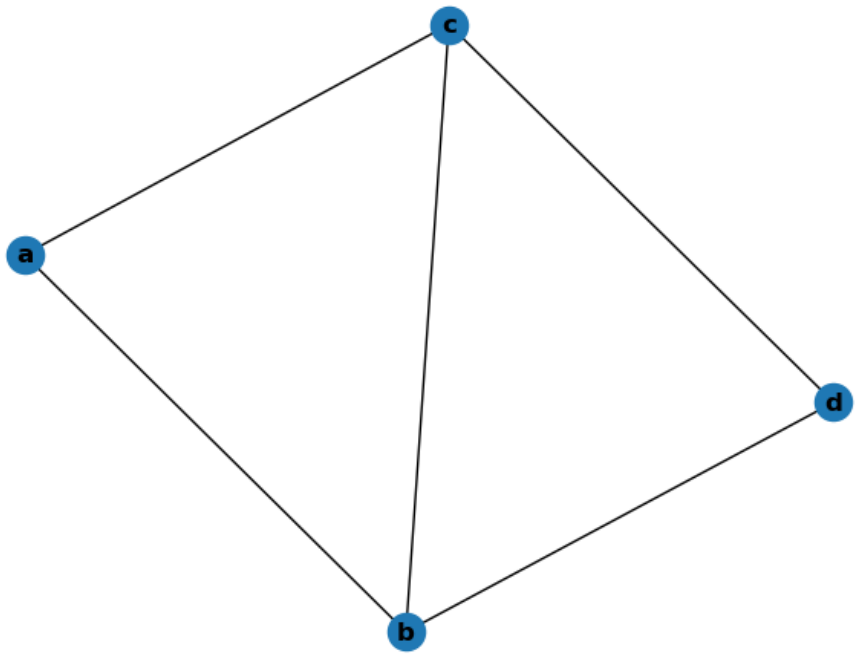
$$S = \{a, b, c\} \text{ et } A = \big\{(a, b), (b, a), (b, c), (c, a)\big\}$$

on obtient le deuxième graphe de la session 1.1, de la présente page.

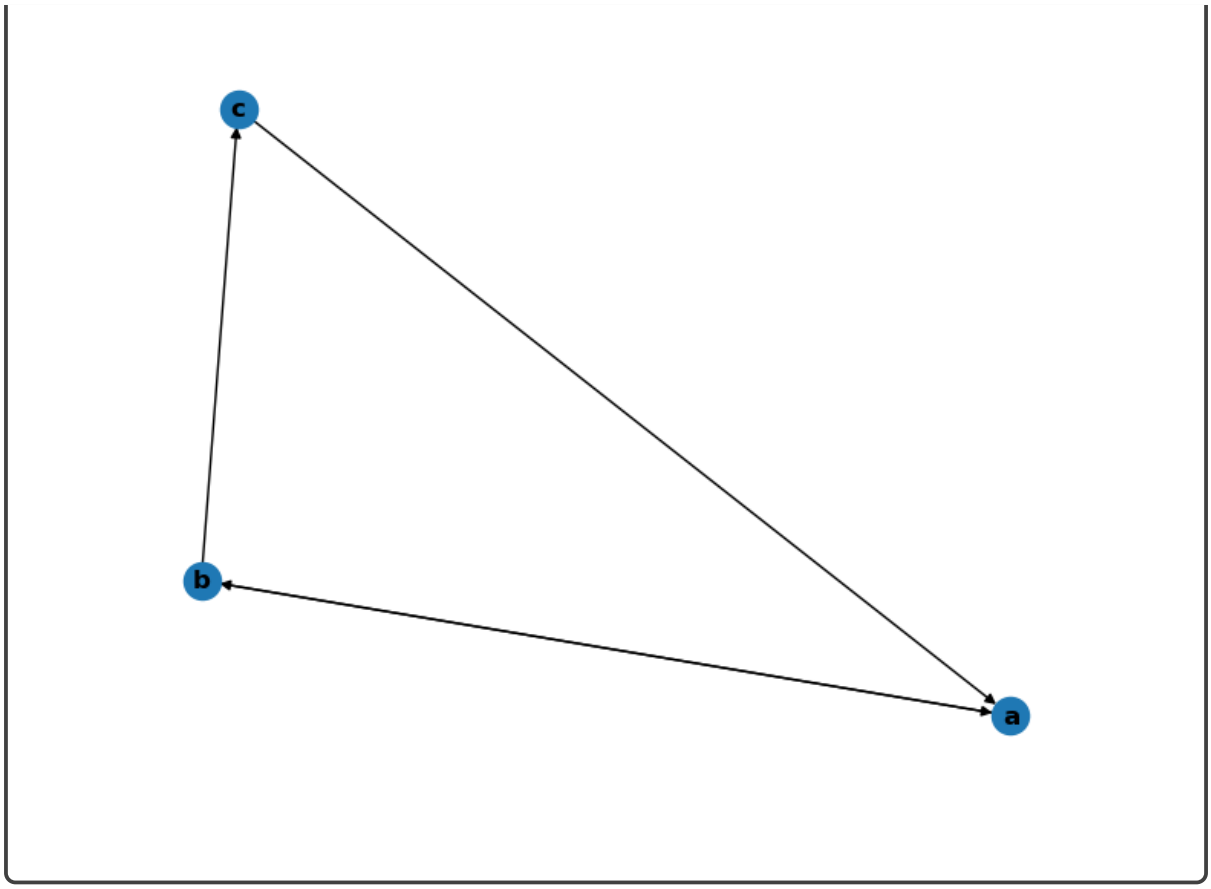
Session `Python` 1.1 – Graphe de l'exemple 1.1

```
import networkx as nx
```

```
    G = nx.Graph([("a", "b"), ("a", "c"), ("b", "c"), ("b", "d"), ("c",␣
↪"d")])
    nx.draw(G, with_labels = True, font_weight = 'bold')
```




```
    H = nx.DiGraph([("a","b"), ("b","a"), ("b","c"), ("c","a")])
    nx.draw(H, with_labels = True, font_weight = 'bold')
```



Définition 1.3 – Graphe pondéré

Un graphe (orienté ou non) peut être *pondéré*, c'est-à-dire qu'à chaque arête/arc sera associée un *poids* (valeur numérique) représentant la plupart du temps le *coût* d'utilisation de l'arête/arc.

Exemple 1.3 – Différents poids

Pour aller d'une ville à une autre, les poids peuvent être

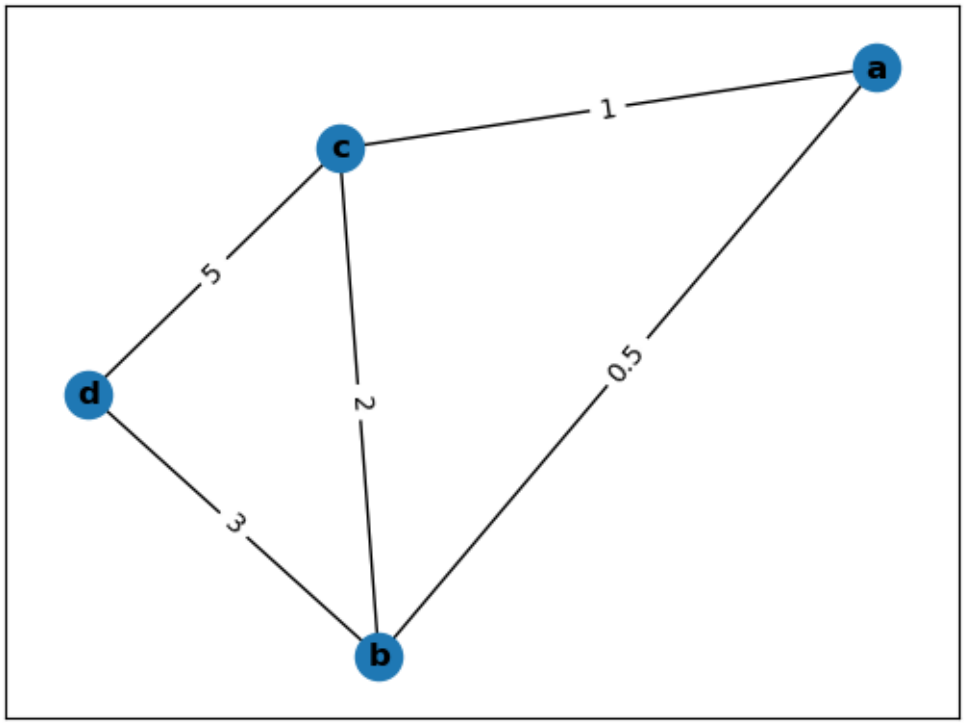
1. les kilomètres ;
2. le temps nécessaire au parcours ;
3. la consommation d'essence...

Exemple 1.4 – Graphe pondéré

Un exemple de code `Python` est donné (code 1.2, page suivante).

Session Python 1.2 – Exemple de graphe pondéré

```
    G = nx.Graph()
    G.add_weighted_edges_from([("a", "b", 0.5), ("a", "c", 1), ("b", "c",␣
↪2), ("b", "d", 3), ("c", "d", 5)])

    pos = nx.spring_layout(G)
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx(G, pos, with_labels = True, font_weight = 'bold')
    nx.draw_networkx_edge_labels(G, pos, edge_labels = labels)
```



Exemple 1.5 – Graphes complets

Le graphe complet à n sommets K_n est le graphe simple tel que

$$S = \{k \in \mathbb{N},\ 1 \leq k \leq n\} \text{ et } A = \big\{a \in \mathcal{P}(S),\ \text{Card}(a) = 2\big\}$$

.

Exemple 1.6 – Graphes linéaires

Le graphe linéaire à n sommets L_n est le graphe simple tel que

$$S = \{k \in \mathbb{N},\ 1 \leq k \leq n\} \text{ et } A = \big\{(k, k+1),\ 1 \leq k < n\big\}$$

.

Définition 1.4 – Sommets adjacents

Deux sommets a et b d'un graphe sont dit *adjacents* lorsque l'arête $\{a, b\}$ est dans le graphe. On dit aussi que a et b sont *voisins*.

Définition 1.5 – Dégré d'un sommet

Le *degré* d'un sommet a d'un graphe simple est le nombre d'arêtes qui contiennent a.

Définition 1.6 – Chemin

Dans un graphe simple, un *chemin* est, de façon equivalente :
- Une suite finie de sommets du graphes, tel que deux sommets successifs de la suite sont adjacents.
- Une suite finie d'arêtes du graphe, telle que deux arêtes successives de la suite partagent un sommet.

Dans un graphe orienté, un chemin devra parcourir les arêtes dans le bon sens.

Définition 1.7 – Composantes connexe

Dans un graphe simpe, la *composante connexe* d'un sommet a du graphe est l'ensemble des sommets b tels qu'il existe un chemin de a à b. Un graphe simple sera dit *connexe* lorsque tout ses sommets sont dans une unique composante connexe.

Définition 1.8 – Cycle et arbre

Un *cycle* est un chemin non trivial (qui passe au moins par une arête) dont le sommet de départ et le sommet d'arrivé sont identiques.
Un chemin ou un cycle est dit *simple* lorsqu'il passe au plus une fois par chaque arête.
Un *arbre* est un graphe simple connexe qui ne contient pas de cycle simple.

1.2 Problèmes liés aux graphes

Problème 1 – Plus court chemin

Dans un graphe pondéré, quel est le plus court chemin pour aller d'un sommet d (départ) à un sommet a (arrivée) ?

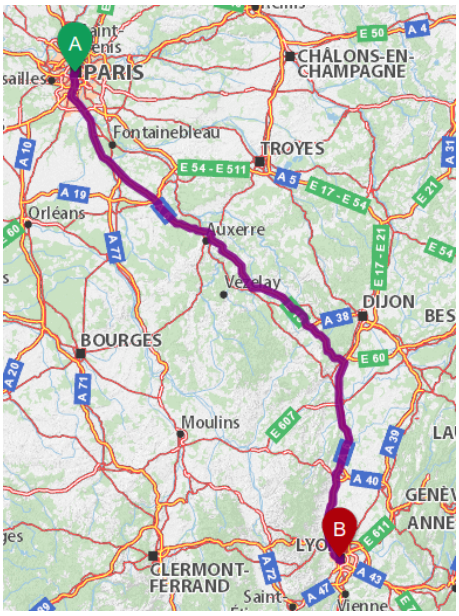
Exemple 1.7 – Plus court chemin

Voici ce que donne Michelin pour un voyage Paris/Lyon. (Figure 1.1, page suivante).

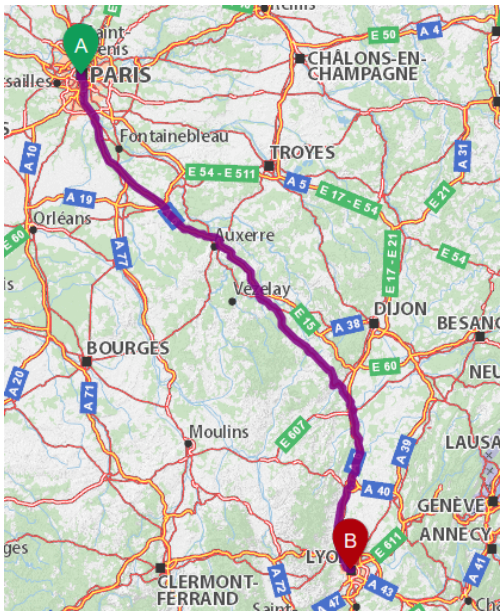
Remarque 1.1

On voit qu'il y a une différence dans Paris et après la ville de Vézelay.

Figure 1.1 – Itinéraire de Paris à Lyon



Itinéraire le plus rapide



Itinéraire le plus court

Problème 2 – Flot maximal

Dans un graphe pondéré, le poids représentant une capacité maximale, quelle est la capacité maximale entre un sommet d de départ et un sommet a d'arrivée ?

Exemple 1.8 – Flot maximal

Un réseau électrique (voir la figure 1.2, page ci-contre) peut fournir quelle quantité d'électricité au maximum. Quand ce n'est pas possible (souvent en hiver), on doit effectuer des délestages (certains endroits ne sont plus approvisionnés). Quels délestages choisir (il peut y avoir aussi des contraintes économiques, de sécurité, etc.) ?

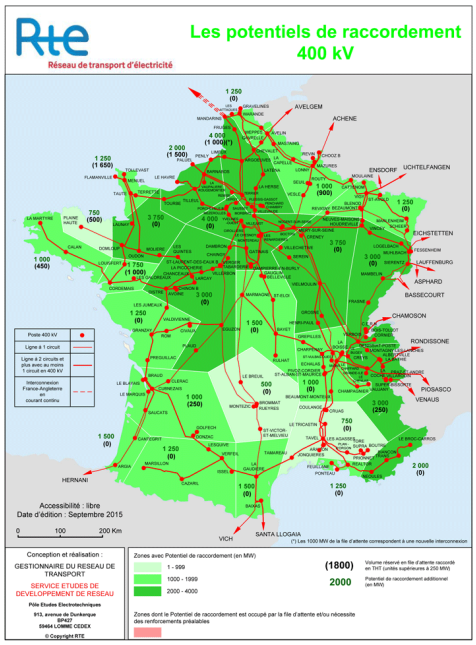
Problème 3 – Cycle ou circuit eulérien

Étant donné un graphe, comment s'organiser pour parcourir toutes les arêtes (problème dit du postier chinois) ?
Pour un cycle, il faut aussi revenir au sommet de départ...

Exemple 1.9 – Problème du postier chinois

Un postier doit distribuer le courrier dans une zone définie. Pour cela, il doit passer dans chaque rue au moins une fois et revenir à son point de départ. Est-il possible de ne passer qu'une seule fois dans chaque rue ? Et en ce cas, comment trouver le chemin ? Si ce n'est pas possible, comment trouver un parcours le plus court possible ? Voir le village de Taulignan et ses rues sur la figure 1.3, page 12. On suppose que le postier doit passer dans chaque rue.

Figure 1.2 – Réseau électrique français



Problème 4 – Cycle ou circuit hamiltonien

Étant donné un graphe, comment s'organiser pour passer par tous les sommets (problème dit du voyageur de commerce) ? Pour un cycle, il faut aussi revenir au sommet de départ.

Exemple 1.10 – Problème du voyageur de commerce

Un voyageur de commerce doit organiser ses voyages de manière à passer par chaque ville qu'il doit desservir. Il voyage en voiture et peut utiliser les routes existantes. Est-ce possible de ne passer qu'une seule fois par chaque ville ? Comment alors trouver le bon circuit à faire ? Et si ce n'est pas possible, comment optimiser son trajet ? Voir la figure 1.4, page 13.

Problème 5 – Couplage optimal

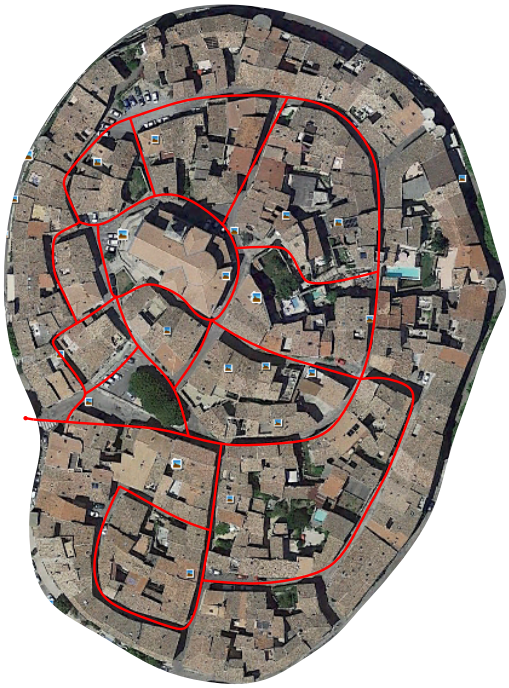
Étant donné un graphe (éventuellement pondéré), peut-on trouver un couplage (sous-ensemble d'arêtes ou d'arcs) de sommets disjoints, dont tous les sommets apparaissent dans les arêtes ou arcs sélectionnés ?

Exemple 1.11 – Affectation de tâches

On a 6 ordinateurs tels que

1. sur l'ordinateur 1, on peut faire tourner `R` et `Fortran` ;
2. sur l'ordinateur 2, on peut tout faire tourner sauf `Fortran` ;

Figure 1.3 – Village de Taulignan dans la Drôme



3. sur l'ordinateur 3, on peut faire tourner `Matlab` et `R` ;
4. sur l'ordinateur 4, on peut faire tourner `Matlab`, `R` et `C` ;
5. sur l'ordinateur 5, on peut faire tourner `Wxmaxima`, `C`, `Python` et `Fortran` ;
6. sur l'ordinateur 6, on peut faire tourner `Matlab` et `Fortran`.

Peut-on faire tourner tous les logiciels ? Si non, combien peut-on faire tourner au maximum ?

Problème 6 – Coloration des sommets d'un graphe

Étant donné un graphe simple, combien faut-il de couleurs au minimum de telle sorte que deux sommets reliés par une arête aient des couleurs différentes ? Et comment obtenir un tel coloriage ?

Exemple 1.12 – Stockage incompatible

Un industriel possède n produits chimiques qu'il veut stocker dans différents entrepôts. Malheureusement, certains produits chimiques ne peuvent pas être stockés dans le même entrepôt. Combien lui faut-il au minimum d'entrepôts ? Les impossibilités sont représentées par le graphe de la figure 1.5, page suivante.

Remarque 1.2

Un cas particulier est la coloration d'une carte avec 4 couleurs. Voir la figure 1.6, page 14

Figure 1.4 – Problème du voyageur de commerce

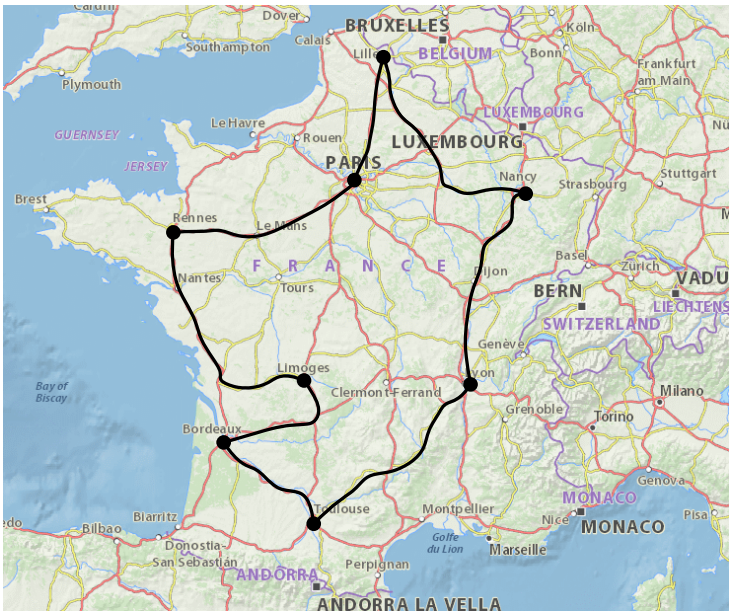
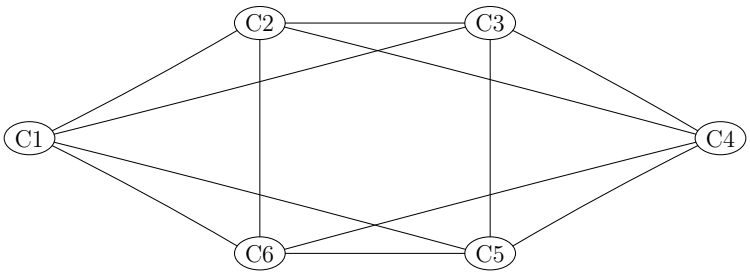


Figure 1.5 – Incompatibilités de stockage des produits chimiques



Problème 7 – Coloration des arêtes d'un graphe

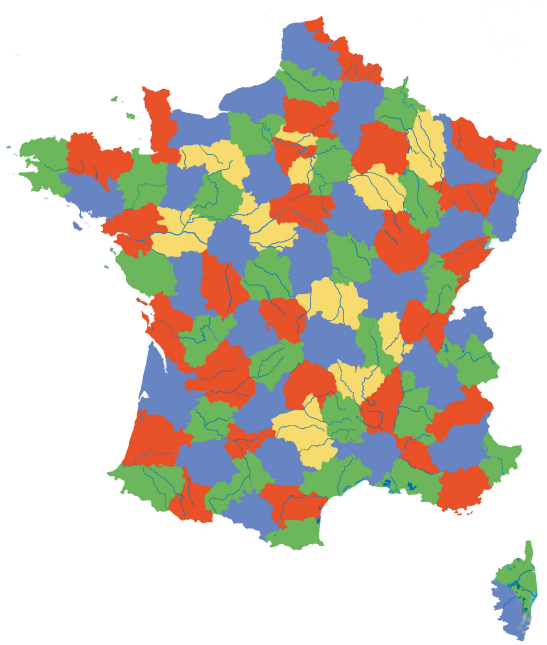
Étant donné un graphe simple, combien faut-il au minimum de couleurs pour colorer les arêtes de telle sorte que deux arêtes passant par un sommet sont de couleurs différentes ?

Exemple 1.13 – Organisation d'un tournoi

On veut organiser un tournoi de go où chaque joueur rencontre tous les autres joueurs une, et une seule fois (tournoi toutes rondes). Comment organiser ce tournoi ? Par exemple, pour 13 ou 14 joueurs, la table peut être comme donnée à la figure 1.7, page 15 [a].

a. Tiré de la page Wikipedia : `https://fr.wikipedia.org/wiki/Table_de_Berger`

Figure 1.6 – Carte coloriée avec 4 couleurs



Problème 8 – Arbre couvrant minimal

Étant donné un graphe, trouver un arbre (voir la définition **??**, page ??) de profondeur minimale qui passe par tous les sommets.

Exemple 1.14 – Réseau de distribution d'électricité, de gaz, ou informatique

On veut pouvoir distribuer de manière à minimiser les réseaux (voir, par exemple, la figure 1.8, page 16).

Problème 9 – Connexité d'un graphe

Étant donné un graphe, peut-on connecter deux sommets quelconques par un chemin ?

Exemple 1.15 – Résistance aux pannes

On a un réseau informatique, que se passe-t-il lorsqu'un ordinateur tombe en panne ? Combien peut-il y avoir de pannes ? Le réseau proposé à la figure 1.9, page 17 est très résistant aux pannes.

Figure 1.7 – Tournoi à 14 joueurs

Ronde							
1	1-14	2-13	3-12	4-11	5-10	6-9	7-8
2	14-8	9-7	10-6	11-5	12-4	13-3	1-2
3	2-14	3-1	4-13	5-12	6-11	7-10	8-9
4	14-9	10-8	11-7	12-6	13-5	1-4	2-3
5	3-14	4-2	5-1	6-13	7-12	8-11	9-10
6	14-10	11-9	12-8	13-7	1-6	2-5	3-4
7	4-14	5-3	6-2	7-1	8-13	9-12	10-11
8	14-11	12-10	13-9	1-8	2-7	3-6	4-5
9	5-14	6-4	7-3	8-2	9-1	10-13	11-12
10	14-12	13-11	1-10	2-9	3-8	4-7	5-6
11	6-14	7-5	8-4	9-3	10-2	11-1	12-13
12	14-13	1-12	2-11	3-10	4-9	5-8	6-7
13	7-14	8-6	9-5	10-4	11-3	12-2	13-1

Problème 10 – Cliques

Étant donné un graphe simple G, chercher les sous-graphes de G (sous-ensembles des sommets) tels que dans ces sous-graphes, tous les sommets sont reliés par des arêtes. (Voir la définition **??**, page ??).

Exemple 1.16 – Synonymie

La relation de synonymie (deux mots sont synonymes s'ils ont à peu près le même sens) peut être vue comme un graphe. Les cliques (voir la définition **??**, page ??) permettent de séparer les différents sens des mots. Dans la figure 1.10, page 18, on peut voir le résultat sur le mot *maison*. (Voir le site `http://dico.isc.cnrs.fr`).

1.3 Représentation sur ordinateur

1.3.1 Matrice d'adjacence

Définition 1.9 – Matrice d'adjacence

Soit G un graphe simple ayant n sommets, on appelle *matrice d'adjacence* de G la matrice de G définie par :

$$M_G = [g_{i,j}]_{(i,j)\in[\![1,n]\!]^2} \in M_n(\mathbb{Z}),\ \forall (i,j) \in [\![1,n]\!]^2,\ g_{i,j} = \begin{cases} 1 & \text{si l'arête } \{i,j\} \text{ appartient au graphe} \\ 0 & \text{sinon.} \end{cases}$$

Figure 1.8 – Exemple d'arbre couvrant du graphe des villes françaises

Remarquons que cette matrice est toujours symétrique !

Exemple 1.17

Le graphe donnée à l'exemple 1.1, page 5 a pour matrice d'adjacence

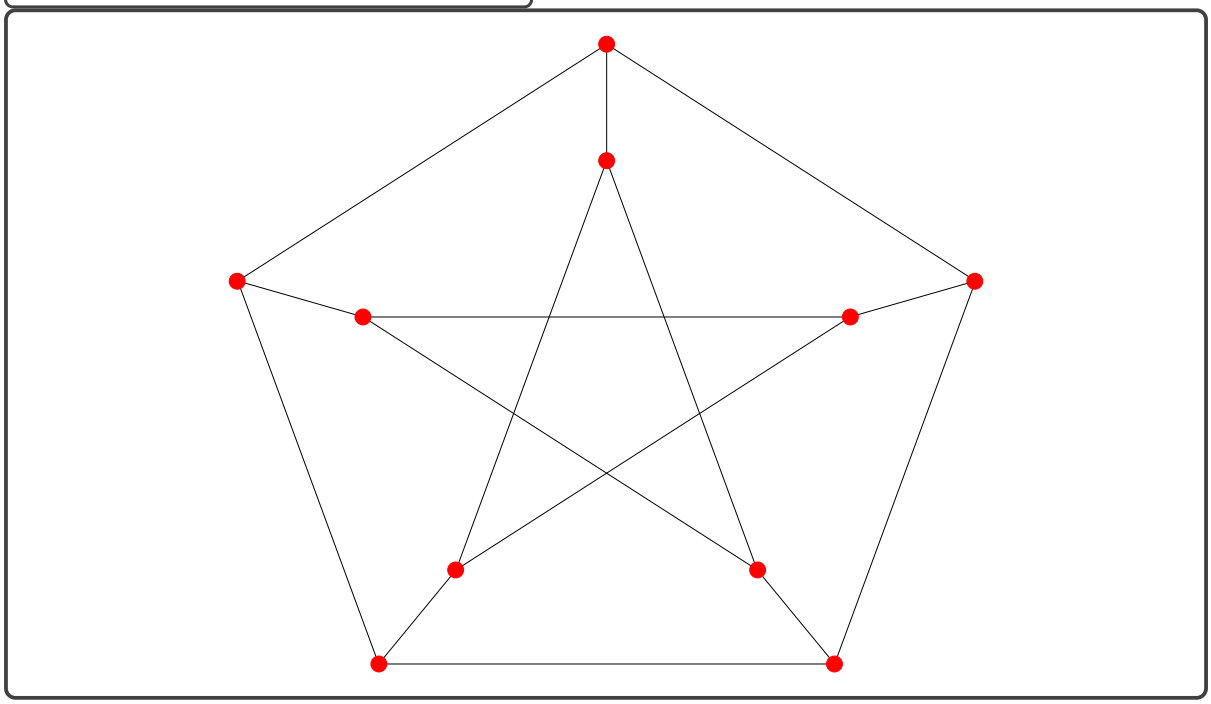
$$M = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Remarque 1.3

On peut, bien sûr, définir aussi les matrices d'adjacence pour les graphes orientés. On la définit comme la matrice

$$M_G = [g_{i,j}]_{(i,j)\in[\![1,n]\!]^2} \in M_n(\mathbb{Z}),\ \forall (i,j) \in [\![1,n]\!]^2\ ,\ g_{i,j} = \begin{cases} 1 & \text{si } \textit{l'arc } (i,j) \text{ appartient au graphe} \\ 0 & \text{sinon.} \end{cases}$$

Figure 1.9 – Réseau résistant aux pannes



1.3.2 Liste d'adjacence

Définition 1.10 – Liste d'adjacence

Soit G un graphe, on appelle *liste d'adjacence* une liste donc les éléments sont les listes des sommets adjacents (reliés par une arête ou un arc) à chaque sommet.

Exemple 1.18

Ainsi, le graphe G de l'exemple 1.1, page 5 a pour liste d'adjacence :

$$\big[[2,3],[1,3,4],[1,2,4],[2,3]\big]$$

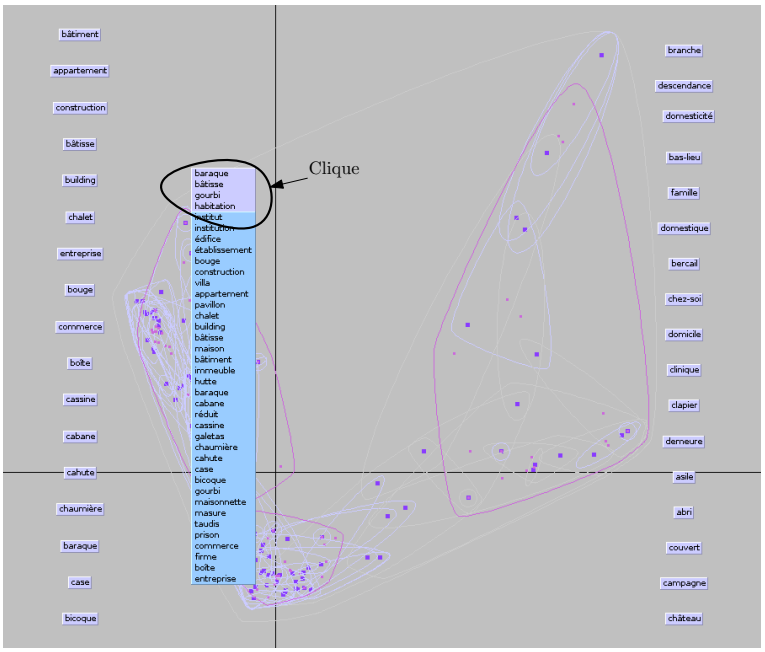
Et le graphe orienté de l'exemple **??**, page ?? a pour liste d'adjacence :

$$\big[[2,3],[1,3,4],[1],[1,2,3]\big]$$

Remarque 1.4

1. Souvent, le graphe est *creux*, c'est-à-dire qu'il y a peu d'arêtes (resp. arcs) par rapport au nombre total des arêtes (resp. arcs) possibles. La matrice d'adjacence est alors très grosse pour ne contenir que peu d'informations : on économise donc la place.
2. La liste d'adjacence étant une liste, cela peut être intéressant pour les situations *dynamiques* ou on ajoute ou retire des sommets.

Figure 1.10 – Clique du mot *maison* dans le graphe de synonymie



Exemple 1.19 – Évaluation des tailles

Étant donné un graphe simple G ayant n sommets et p arêtes. La matrice d'adjacence contiendra alors n^2 valeurs (mais ce sont des 0 ou des 1), alors que la liste d'adjacence contiendra $2\,p$ valeurs (mais ce sont des entiers).
En revanche, les temps de calculs peuvent être différents. En effet, savoir par exemple si une arête existe est facile sur une matrice d'adjacence et se réalise en temps constant, alors que sur une liste d'adjacence, cela va dépendre de la longueur de la liste que l'on est en train de lire...

Chapitre 2

Plus court chemin (Problème 1)

2.1 Quelques définitions

Définition 2.1 – Chemin

Soit G un graphe (orienté ou non), soit d (départ) et a (arrivée) deux sommets de G. On appelle *chemin de d à a* toute liste de sommets de G $(s_0, \dots, s_n)$ tels que

$$s_0 = d,\ s_n = a \text{ et } \forall k \in \mathbb{N}^*, \begin{cases} \{s_{k-1}, s_k\} & \text{est une arête de } G \\ & \text{(graphe simple)} \\ \text{ou} & \\ (s_{k-1}, s_k) & \text{est un arc de } G \\ & \text{(graphe orienté).} \end{cases}$$

n est alors appelé la *longueur* du chemin $(s_0, \dots, s_n)$.

Définition 2.2 – Plus court chemin

Parmi tous les chemins, allant de d à a, on appelle *plus court chemin* un chemin qui a une longueur minimale. Celui-ci n'est pas nécessairement unique.

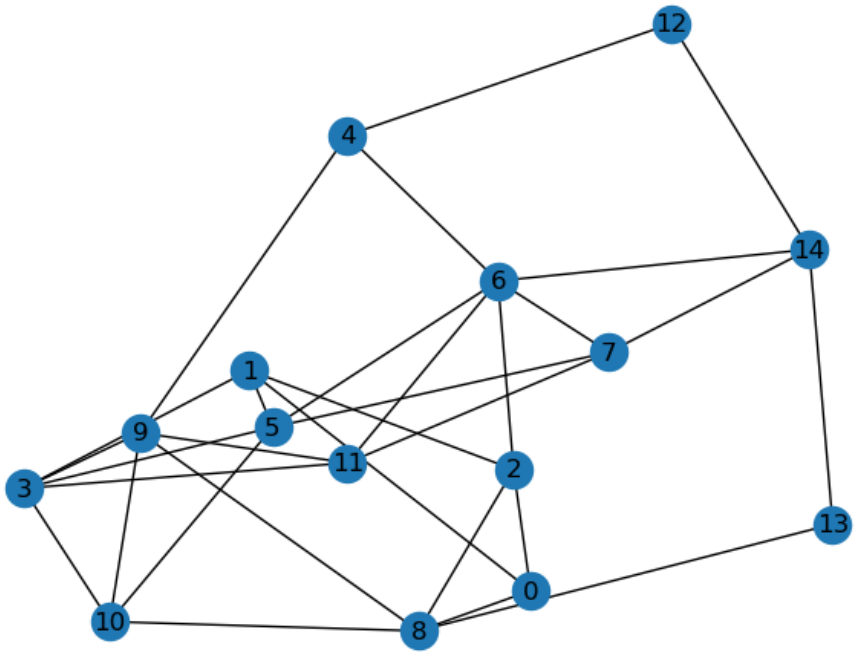
Exemple 2.1

Voici un plus court chemin calculé par `Python` (voir le code 2.1, de la présente page).

Session `Python` 2.1 – Plus court chemin

```
import networkx as nx
import matplotlib.pyplot as plt
```

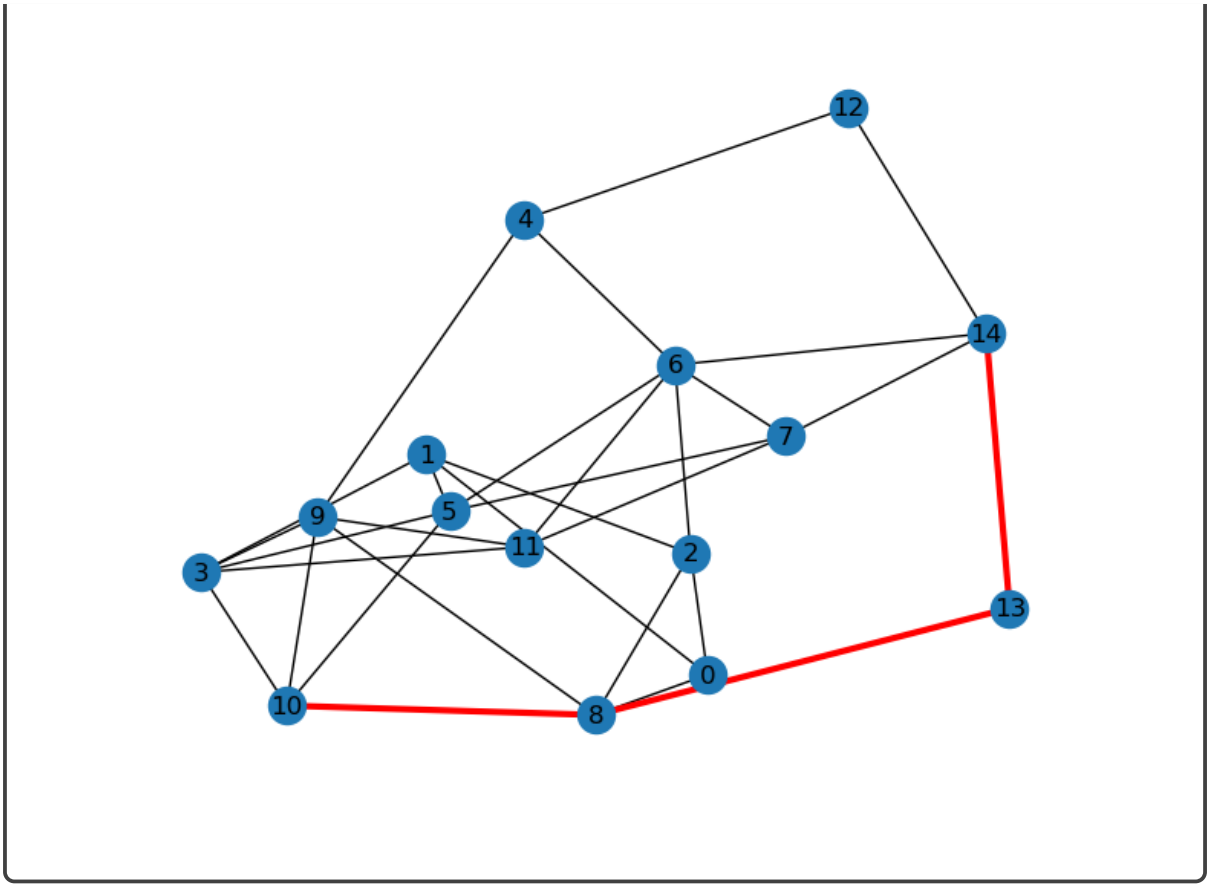
```
G = nx.connected_watts_strogatz_graph(15, 4, 0.5)
position = nx.spring_layout(G)
nx.draw(G, position, with_labels=True)
```




```
    P = nx.shortest_path(G, 14, 10, weight = 'weight')
    P
```

```
    [14, 13, 8, 10]
```

```
    chemin = [(P[i], P[i + 1]) for i in range(len(P) - 1)]

    nx.draw(G, position, with_labels=True)
    nx.draw_networkx_edges(G, position, edgelist = chemin, edge_color =␣
↪'red', width = 3)
    plt.show()
```



Définition 2.3 – Chemin de poids minimal

Soit G un graphe pondéré, d et a deux sommets de G, on appelle *chemin de poids minimal* tout chemin allant de d à a dont la somme des poids des arêtes/arcs parcouru(e)s est minimale.

Exemple 2.2

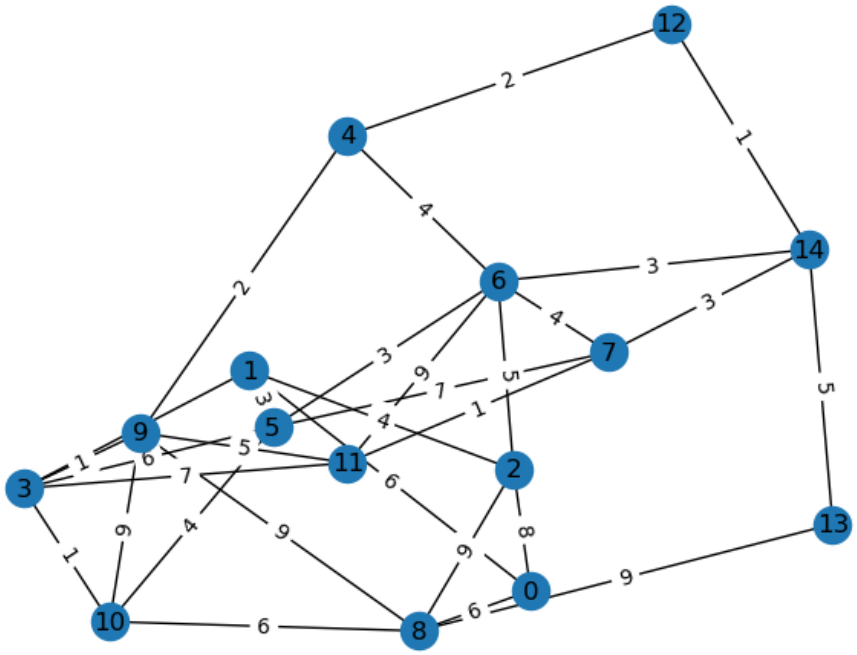
Le code `Python` 2.2, de la présente page est un exemple de calcul de chemin de poids minimal.

Session `Python` 2.2 – Chemin de poids minimal

```
from random import randrange
```

```
for e in G.edges:
G.edges[e]['weight'] = randrange(1, 10, 1)
```

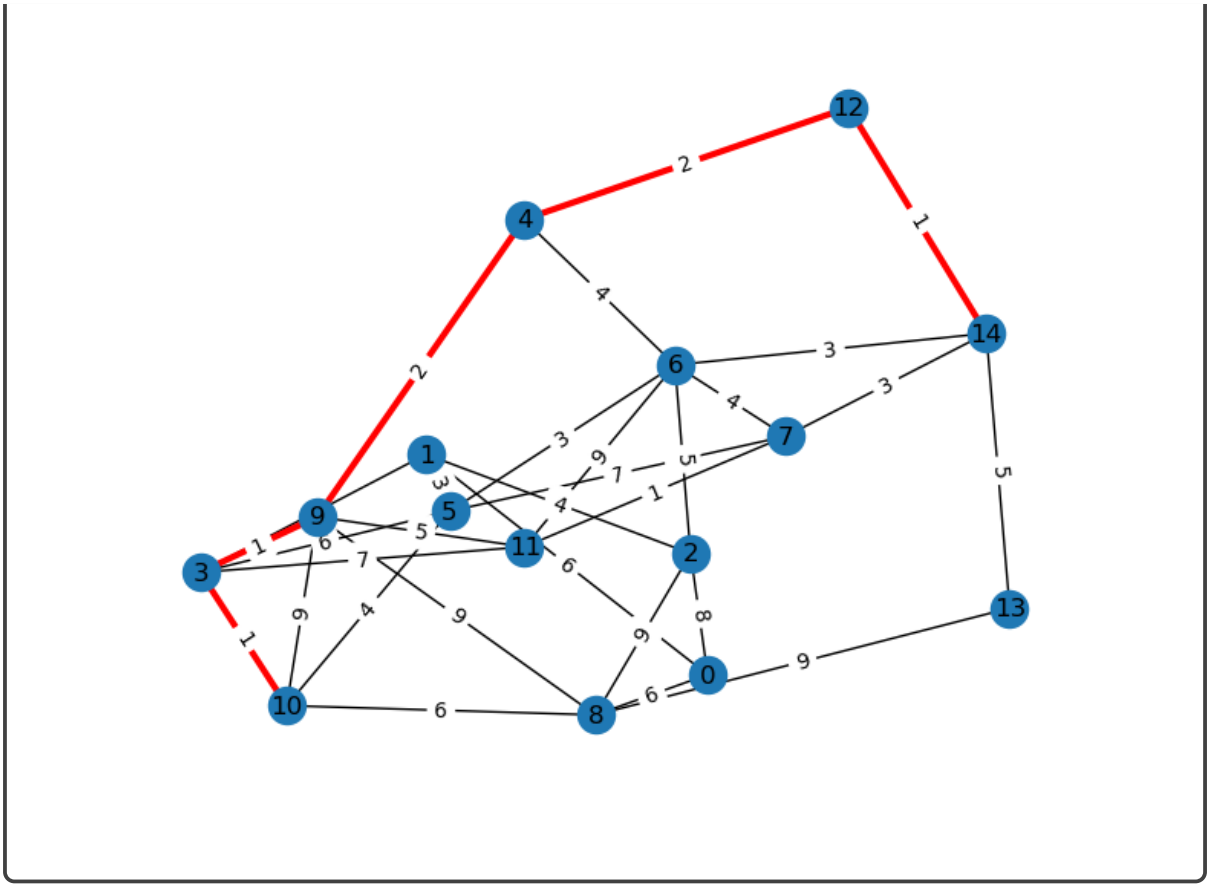
```
poids = nx.get_edge_attributes(G, 'weight')

nx.draw(G, position, with_labels=True)
nx.draw_networkx_edge_labels(G, position, edge_labels=poids)
plt.show()
```




```
P = nx.shortest_path(G, 14, 10, weight = 'weight')
P
```

```
[14, 12, 4, 9, 3, 10]
```

```
chemin = [(P[i], P[i + 1]) for i in range(len(P) - 1)]

nx.draw(G, position, with_labels=True)
nx.draw_networkx_edges(G, position, edgelist = chemin, edge_color =␣
↪'red', width = 3)
nx.draw_networkx_edge_labels(G, position, edge_labels = poids)
plt.show()
```

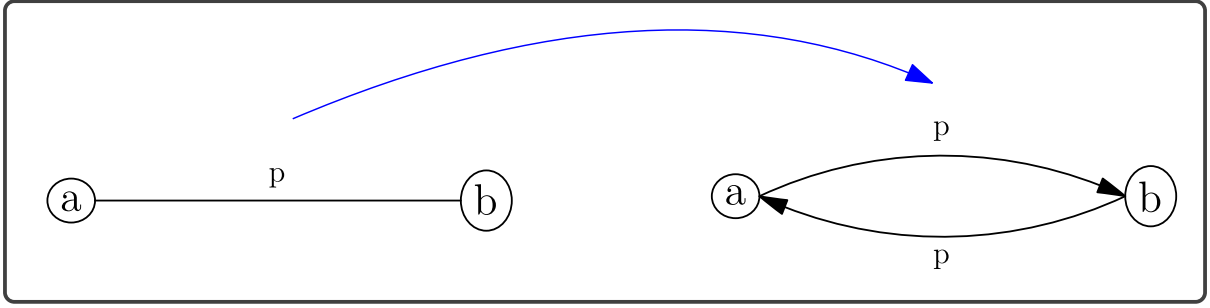


Remarque importante 2.1

On voit qu'il y a plusieurs situations.

1. *Le graphe n'est pas pondéré* : en ce cas, cela revient à dire que les poids de chaque arête/arc est pondéré par 1.
2. *Le graphe n'est pas orienté* : en ce cas, on peut considérer que chaque arête correspond à deux arcs suivant le dessin de la figure 2.1, de la présente page.

Figure 2.1 – Graphe orienté ou non ?



2.2 Algorithme de Dijkstra

Nous supposerons dans ce paragraphe que tous les graphes sont orientés et pondérés (suivant la remarque 2.1, page précédente). Nous appellerons *plus court chemin* tout chemin de poids minimal.

2.2.1 Principe

L'algorithme de Dijkstra ne fonctionne que lorsque les *poids sont tous positifs !*

Il s'agit de trouver un plus court chemin allant d'un sommet d à un sommet a.

Initialisation

Notation 2.1

Nous noterons le poids entre les sommets i et j par

$$\forall (i,j) \in A,\ \delta(i,j).$$

Notons, pour chaque sommet s du graphe G, $\pi(s)$ le poids total pour aller de d à s et $P(s)$ le prédécesseur dans le plus court chemin allant de d à s. On initialise par :

$$\forall s \in S \setminus \{d\},\ \pi(s) = \begin{cases} \delta(d,s) & \text{si } (d,s) \in A \\ \infty & \text{sinon} \end{cases} \text{ et } P(s) = \begin{cases} d & \text{si } (d,s) \in A \\ \infty & \text{sinon.} \end{cases}$$

On dispose de deux ensembles :

$$T = \{d\} \text{ ensemble des sommets traités}$$

et

$$N = S \setminus \{d\} \text{ ensemble des sommets non traités.}$$

Proposition 2.1 – Algorithme de Dijkstra

On trouve tous les chemins minimaux par l'algorithme suivant :

Tant que N est non vide

(a) choisir $i \in N$, tel que $\pi(i)$ soit minimal ;

(b) transformer T en $T \cup \{i\}$ et N en $N \setminus \{i\}$;

(c) pour chaque $j \in N$, $(i,j) \in A$, mettre à jour les poids par la relation

$$\text{si } \pi(j) > \pi(i) + \delta(i,j),\ \text{alors } \begin{cases} \pi(j) & \leftarrow \pi(i) + \delta(i,j) \\ P(j) & \leftarrow i. \end{cases}$$

Remarque 2.2

L'algorithme de Dijkstra choisit à chaque étape ce qui paraît être la solution optimale. Un tel algorithme est appelé *algorithme glouton*.

2.2.2 Correction

Quand on produit un algorithme pour résoudre un problème, la première question à se poser est : l'algorithme s'arrête-t-il ? C'est le problème de *l'arrêt*. La deuxième question est : trouve-t-on la/une bonne solution ? C'est le problème de la *correction* de l'algorithme.

Démonstration de l'algorithme de Dijkstra (proposition 2.1, page précédente)

1. *Arrêt.* À chaque étape, on augmente T et on diminue N, donc, il y a au plus $n-1$ étapes, où n est le nombre de sommets de G. Chaque étape met à jour un nombre fini de valeurs
$$\{\pi(s),\ s \in N\} \text{ et } \{P(s),\ s \in N\}.$$
2. *Correction.* Le fait que l'algorithme produit
   (a) la preuve de la non existence d'un chemin de d à a, si $P[a] = \infty$;
   (b) la production d'un chemin allant de d à a, si $P[a] \neq \infty$ est immédiate (c'est ce que fait la fonction `SP`).

   La seule chose à démontrer est que l'on a *un plus court chemin* allant de d à a. Pour cela nous allons raisonner par récurrence sur $q = \text{Card}(T)$ et montrer qu'à ce moment, les valeurs des $\pi(s)$ représentent les poids des plus courts chemins n'utilisant que les sommets de T.
   (a) (*Initialisation*) Le premier sommet s_1 que l'on ajoute à T est choisi tel que
   $$\pi(s_1) = \min\left(\{\pi(s) = \delta(1,s),\ s \in N\}\right).$$
   On a donc un plus court chemin de d à s_1, n'utilisant que $\{d, s_1\}$.
   (b) (*Hérédité*) Supposons que
   $$T = \{d, s_1, \ldots, s_{q-1}\}$$
   soit tel que

   $\forall s \in T\backslash\{d\},\ \pi(s)$ est le poids d'un plus court chemin de d à s n'utilisant que les sommets de T.

   On choisit alors
   $$s_q \in N,\ \pi(s_q) = \min\left(\{\pi(s),\ s \in N\}\right).$$
   La mise à jour des valeurs de π représentent la prise en compte des chemins utilisant les sommets de
   $$\{d, s_1, \ldots, s_{q-1}, s_q\},$$
   notamment dans le cas où il est avantageux de passer par s_q.

2.2.3 Complexité

Une chose utile aussi pour un algorithme est sa *complexité*.

Proposition 2.2

Soit G un graphe orienté pondéré ayant n sommets et m arcs, alors la complexité de l'algorithme de Dijkstra est, en pire cas, donné par la formule

$$C(n,m) = \Theta\left(n^2\right).$$

Démonstration

Il y a $n-1$ étapes. À chaque étape (q sommets dans T), la mise à jour (cas où le sommet est connecté à tous les autres) demande $\Theta(n-1)$ opérations (comparaisons/additions, etc.)

Remarque importante 2.3

On rappelle qu'un graphe simple peut être vu comme un graphe pondéré avec des poids 1. Lors de la mise à jour, il n'est pas utile de regarder une arête déjà évaluée... Si on modifie un peu l'algorithme, la complexité devient donc en

$$\Theta(m).$$

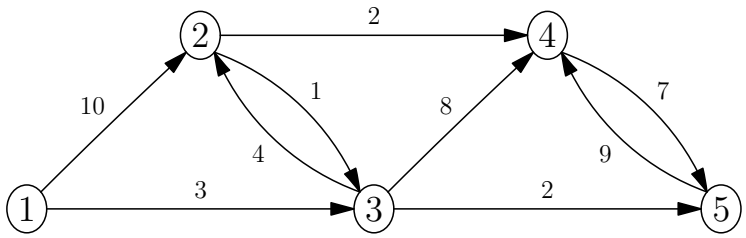
Exercice(s) 2.1

Soit le graphe donnée par la figure 2.2, de la présente page. On veut trouver tous les plus courts chemins partant du sommet 1.

(a) Appliquer l'algorithme de Dijkstra, en précisant toutes les étapes du calcul.

(b) Préciser le plus court chemin allant de 1 à 4 et donner son poids.

Figure 2.2 – Exercice sur Dijkstra



2.3 Algorithmes de Bellman-Ford et de Floyd-Marshall

L'algorithme de Dijkstra de recherche du plus court chemin a plusieurs défauts :

1. il suppose que tous les poids sont positifs, que faire quand certains poids deviennent négatifs ?
2. il donne les plus courts chemins à partir d'un point d, on veut parfois connaître des plus courts chemins entre toutes les paires de points. Avec l'algorithme de Dijkstra, il faut recommencer à chaque fois.

Remarque 2.4

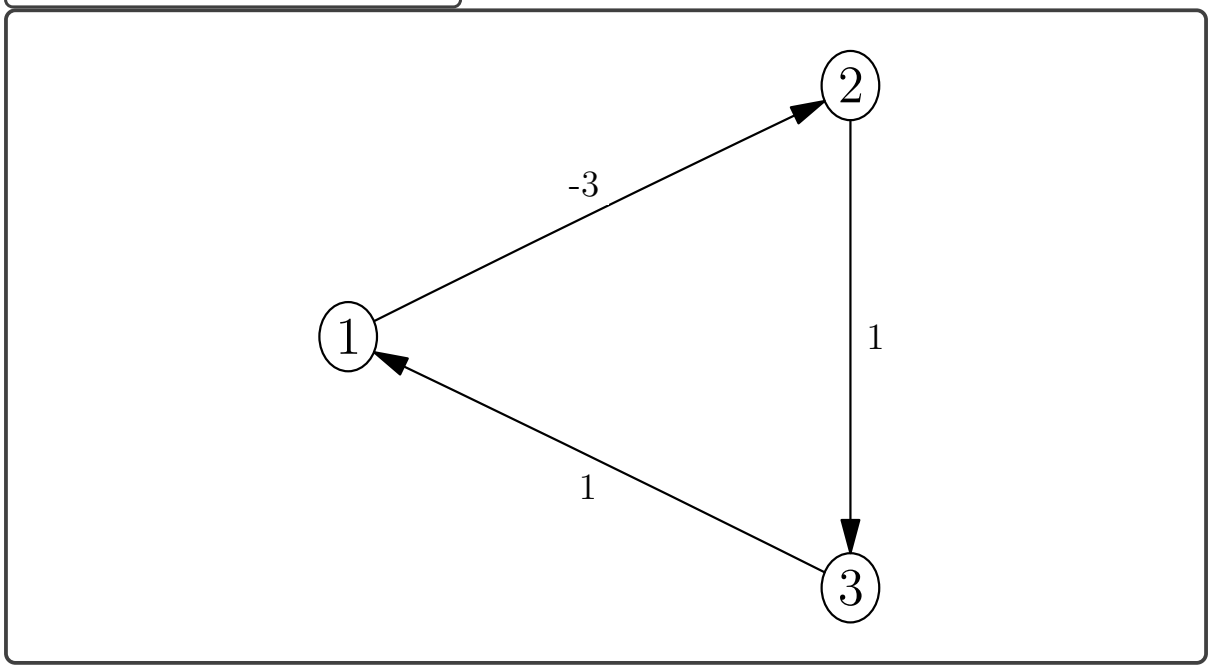
Les poids négatifs peuvent entraîner l'existence d'un cycle de poids négatif et, en ce cas, il n'y a pas existence d'un chemin de poids minimal. Voir la figure 2.3, page suivante.

Proposition 2.3 – Algorithme de Bellman – Ford

Le principe de l'algorithme de Bellman – Ford est le suivant. On a un graphe orienté pondéré (les poids pouvant être négatifs). Le poids de l'arc $(i,j) \in A$ est noté $\delta(i,j)$.
Notons pour $k \in [\![0, \mathrm{Card}(S)]\!]$ et $i \in S$,

$$\pi_k(d,i) = \textit{poids du plus court chemin de } d \textit{ à } i \textit{ en utilisant } k \textit{ arcs.}$$

Figure 2.3 – Cycle de poids négatif



$R_k(d,i)$ = *le premier sommet visité dans le plus cours chemin de* d *à* i *en utilisant* k *arcs*

On a alors :

1. Initialisation

$$\forall (d,i) \in S^2,\ \pi_0(d,i) = \begin{cases} 0 & si\ i = d \\ \infty & sinon. \end{cases}$$

$$\forall (d,i) \in S^2,\ R_0(d,i) = \begin{cases} d & si\ i = d \\ ? & sinon. \end{cases}$$

2. Hérédité

$$\forall k \in [\![1, \mathrm{Card}(S)]\!],\ \forall (d,i) \in S^2,\ \pi_k(i) = \min\left(\pi_{k-1}(d,i), \min_{u \in S,\ (d,u) \in A} \left(\pi_{k-1}(u,i) + \delta(d,u)\right)\right).$$

$$R_k(d,i) = \begin{cases} R_{k-1}(d,i) & si\ \pi_k(d,i) = \pi_{k-1}(d,i) \\ u & un\ sommet\ pour\ lequel\ le\ minimum \\ & est\ atteint\ dans\ le\ cas\ ou\ \pi_k(d,i) < \pi_{k-1}(d,i) \end{cases}$$

3. Arrêt $k = \mathrm{Card}(S) - 1$.
4. Test de cyclicité de poids négatif
   - *On calcule les* π_k *pour* $k = \mathrm{Card}(S)$.
   - *Si*

$$\forall i \in S,\ \pi_k(i) = \pi_{k-1}(i),$$

   il n'y a pas de cycle et les π_k *sont les poids minimaux cherchés.*
   - *Sinon, il y a un cycle de poids négatif !*

Exercice(s) 2.2

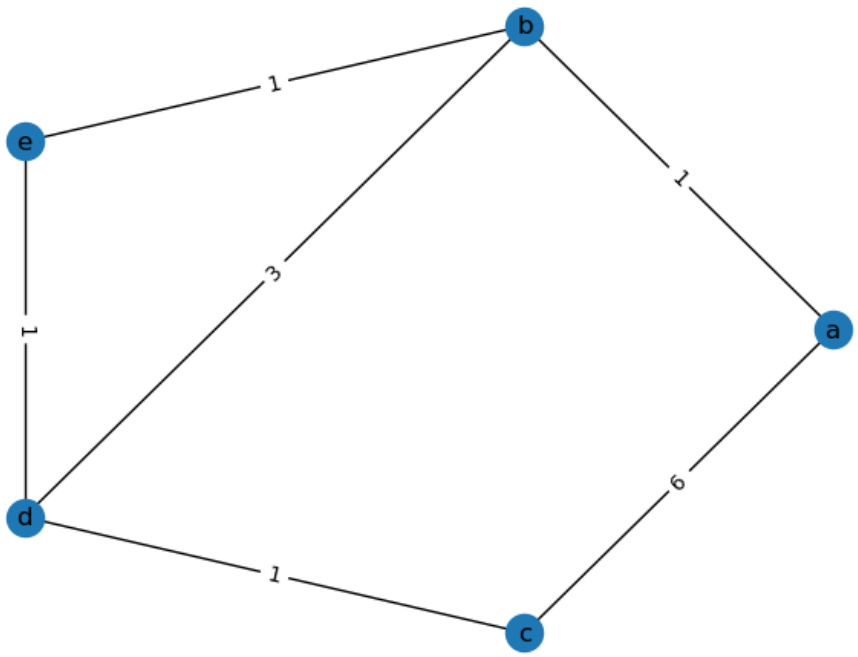
Montrer que l'algorithme s'arrête et produit les poids minimaux cherchés. Quelle est sa complexité ?

Session Python 2.3 – Algorithme de Bellman – Ford

```
import networkx as nx
import matplotlib.pyplot as plt
from math import inf
```

```
G = nx.Graph()
G.add_weighted_edges_from([('a', 'b', 1), ('a', 'c', 6), ('b', 'd', 3),
('c', 'd', 1), ('b', 'e', 1), ('d', 'e', 1)])
```

```
# Affichage du graphe G
position = nx.circular_layout(['a', 'b', 'e', 'd', 'c'])
poids = nx.get_edge_attributes(G, 'weight')

nx.draw(G, position, with_labels=True)
nx.draw_networkx_edge_labels(G, position, edge_labels=poids)
plt.show()
```



```
# Initialisation de l'algorithme
def bellmanford_init(G):
R = {}
for d in G.nodes:
R[d] = {}
for i in G.nodes:
R[d][i] = {'premier' : '?', 'weight' : inf}
R[d][d] = {'premier' : d, 'weight' : 0}
return R
```

```
R = bellmanford_init(G)
R
```

```
{'a': {'a': {'premier': 'a', 'weight': 0},
'b': {'premier': '?', 'weight': inf},
'c': {'premier': '?', 'weight': inf},
'd': {'premier': '?', 'weight': inf},
'e': {'premier': '?', 'weight': inf}},
'b': {'a': {'premier': '?', 'weight': inf},
'b': {'premier': 'b', 'weight': 0},
'c': {'premier': '?', 'weight': inf},
'd': {'premier': '?', 'weight': inf},
'e': {'premier': '?', 'weight': inf}},
'c': {'a': {'premier': '?', 'weight': inf},
'b': {'premier': '?', 'weight': inf},
'c': {'premier': 'c', 'weight': 0},
'd': {'premier': '?', 'weight': inf},
'e': {'premier': '?', 'weight': inf}},
'd': {'a': {'premier': '?', 'weight': inf},
'b': {'premier': '?', 'weight': inf},
'c': {'premier': '?', 'weight': inf},
'd': {'premier': 'd', 'weight': 0},
'e': {'premier': '?', 'weight': inf}},
'e': {'a': {'premier': '?', 'weight': inf},
'b': {'premier': '?', 'weight': inf},
'c': {'premier': '?', 'weight': inf},
'd': {'premier': '?', 'weight': inf},
'e': {'premier': 'e', 'weight': 0}}}
```

```
# Faire une étape de l'algorithme
def bellmanford_step(G, R):
R2 = R.copy()
for d in G.nodes:
for i in G.nodes:
for u in G.adj[d]:
if G[d][u]['weight'] + R[u][i]['weight'] < R2[d][i]['weight']:
R2[d][i]['weight'] = G[d][u]['weight'] + R[u][i]['weight']
R2[d][i]['premier'] = u
return R2
```

```
    # Calcul du plus court chemin **connu** pour la table de routage R
    def plus_court_chemin(R, d, i):
    if d == i:
    return [d]
    if R[d][i]['premier'] == '?':
    return ['?']
    u = R[d][i]['premier']
    return [d] + plus_court_chemin(R, u, i)
```

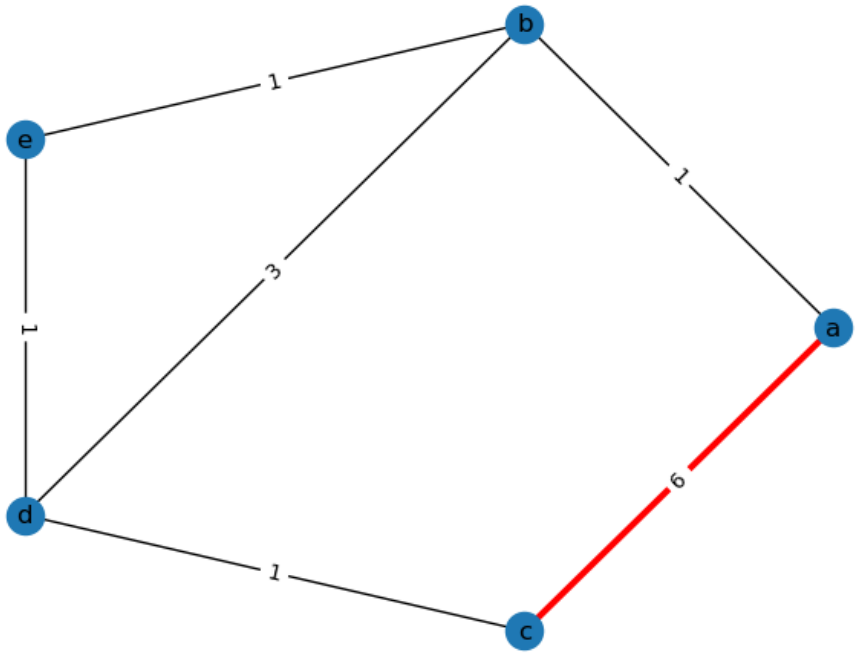
```
    # Trace le graphe avec le plus court chemin en rouge. Trace uniquement␣
↪le graphe si le chemin est inconnu.
    def afficher_chemin(c):
    global position, poids
    if not('?' in c):
    chemin = [(c[i], c[i + 1]) for i in range(len(c) - 1)]

    nx.draw(G, position, with_labels=True)
    nx.draw_networkx_edges(G, position, edgelist = chemin, edge_color =␣
↪'red', width = 3)
    nx.draw_networkx_edge_labels(G, position, edge_labels = poids)
    plt.show()
    else:
    nx.draw(G, position, with_labels=True)
    nx.draw_networkx_edge_labels(G, position, edge_labels=poids)
    plt.show()
```

```
    R = bellmanford_step(G, R)
    R
```

```
{'a': {'a': {'premier': 'a', 'weight': 0},
'b': {'premier': 'b', 'weight': 1},
'c': {'premier': 'c', 'weight': 6},
'd': {'premier': '?', 'weight': inf},
'e': {'premier': '?', 'weight': inf}},
'b': {'a': {'premier': 'a', 'weight': 1},
'b': {'premier': 'b', 'weight': 0},
'c': {'premier': 'a', 'weight': 7},
'd': {'premier': 'd', 'weight': 3},
'e': {'premier': 'e', 'weight': 1}},
'c': {'a': {'premier': 'a', 'weight': 6},
'b': {'premier': 'a', 'weight': 7},
'c': {'premier': 'c', 'weight': 0},
'd': {'premier': 'd', 'weight': 1},
'e': {'premier': '?', 'weight': inf}},
'd': {'a': {'premier': 'b', 'weight': 4},
'b': {'premier': 'b', 'weight': 3},
'c': {'premier': 'c', 'weight': 1},
'd': {'premier': 'd', 'weight': 0},
'e': {'premier': 'e', 'weight': 1}},
'e': {'a': {'premier': 'b', 'weight': 2},
'b': {'premier': 'b', 'weight': 1},
'c': {'premier': 'd', 'weight': 2},
'd': {'premier': 'd', 'weight': 1},
'e': {'premier': 'e', 'weight': 0}}}
```

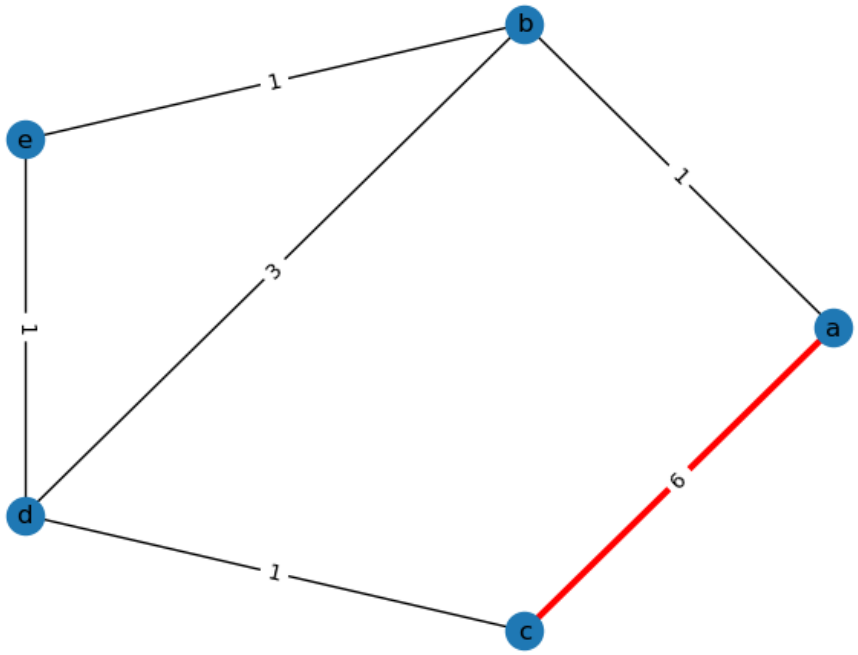
```
c = plus_court_chemin(R, 'a', 'c')
print(c)
afficher_chemin(c)
```

```
['a', 'c']
```

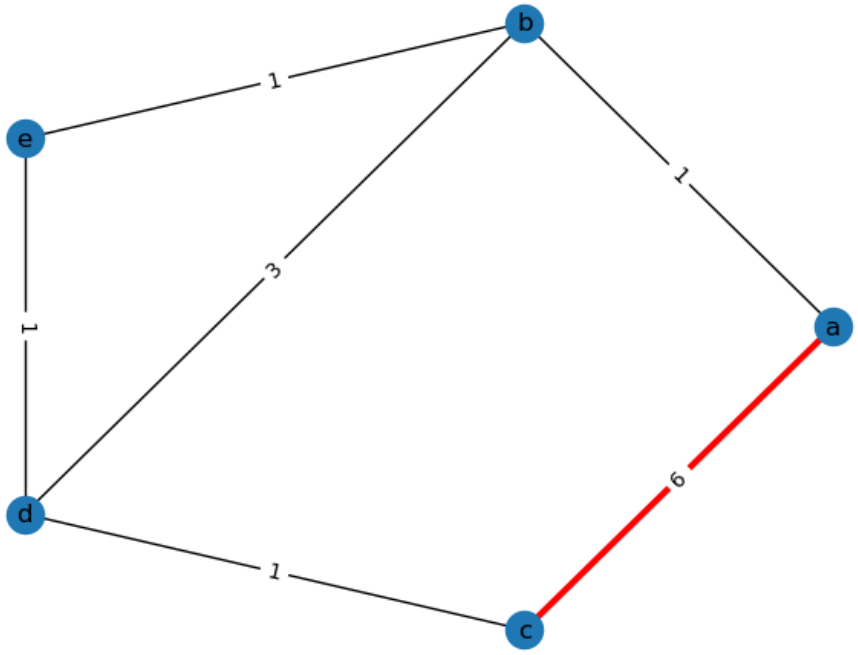



```
    # Faire toute les étapes de l'algorithme.
    R = bellmanford_init(G)
    for k in range(len(G.nodes) - 1):
    R = bellmanford_step(G, R)
    c = plus_court_chemin(R, 'a', 'c')
    print(c)
    afficher_chemin(c)
```

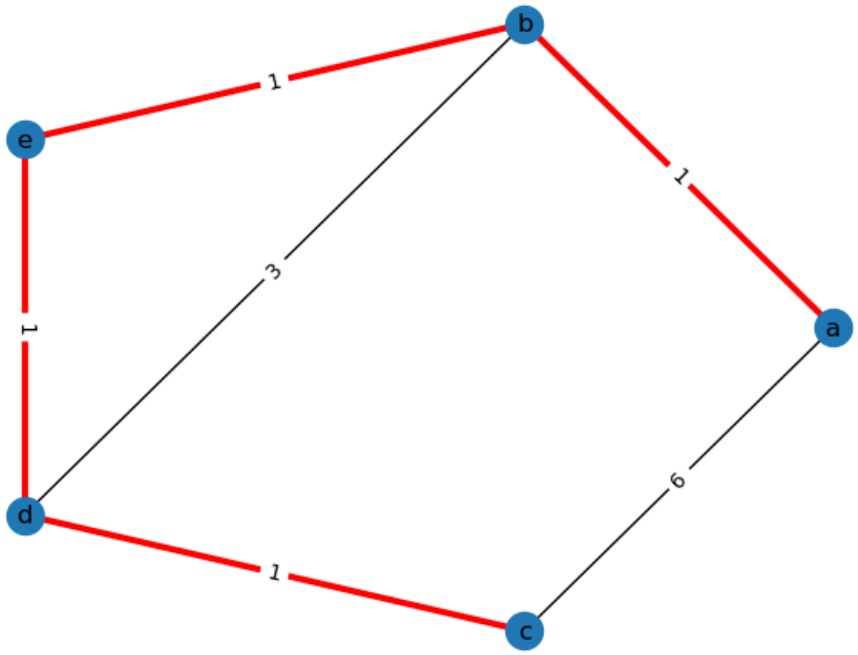
```
['a', 'c']
```



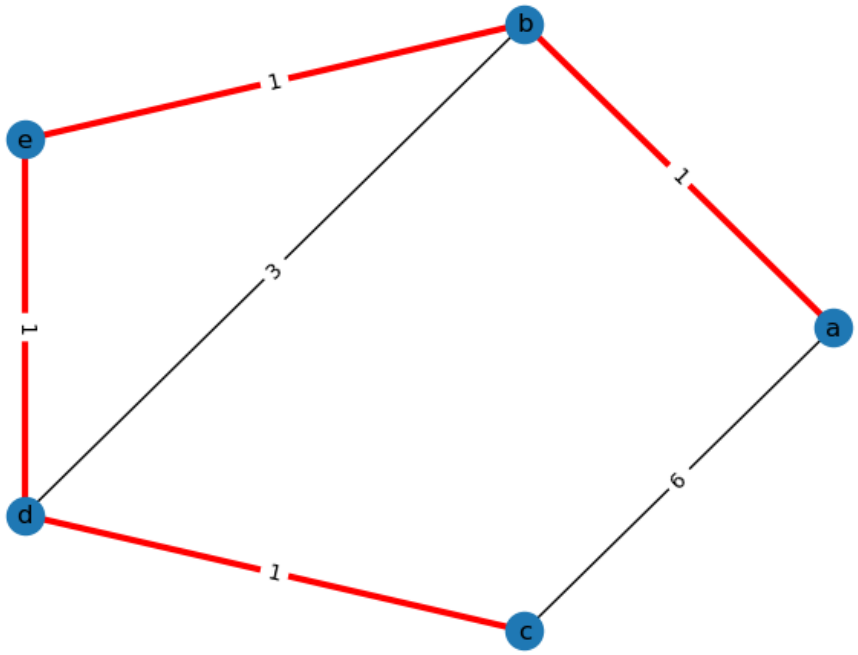

```
['a', 'c']
```



```
['a', 'b', 'e', 'd', 'c']
```

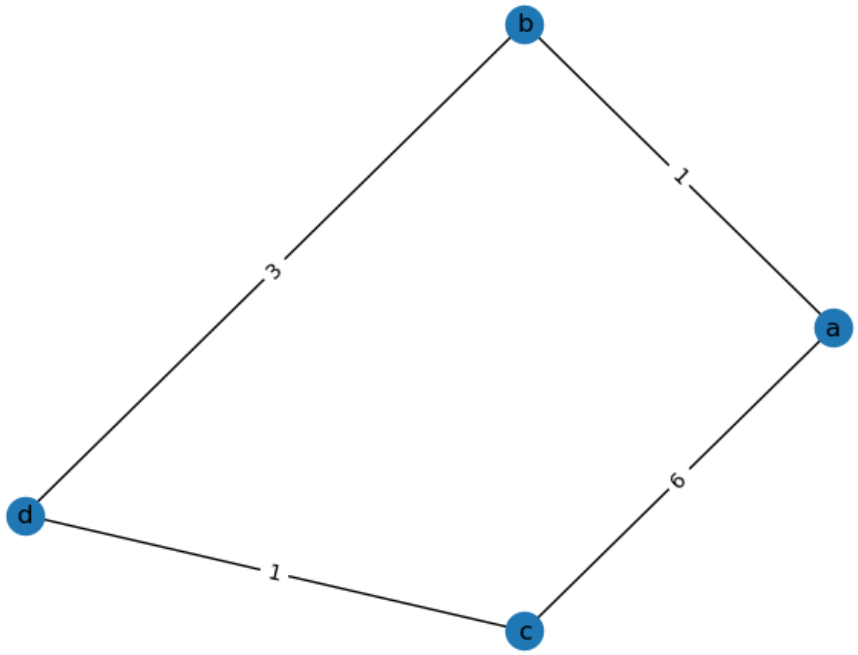



```
['a', 'b', 'e', 'd', 'c']
```




```
# Un ordinateur ne fonctionne plus !
G.remove_node('e')
```

```
poids = nx.get_edge_attributes(G, 'weight')

nx.draw(G, position, with_labels=True)
nx.draw_networkx_edge_labels(G, position, edge_labels=poids)
plt.show()
```



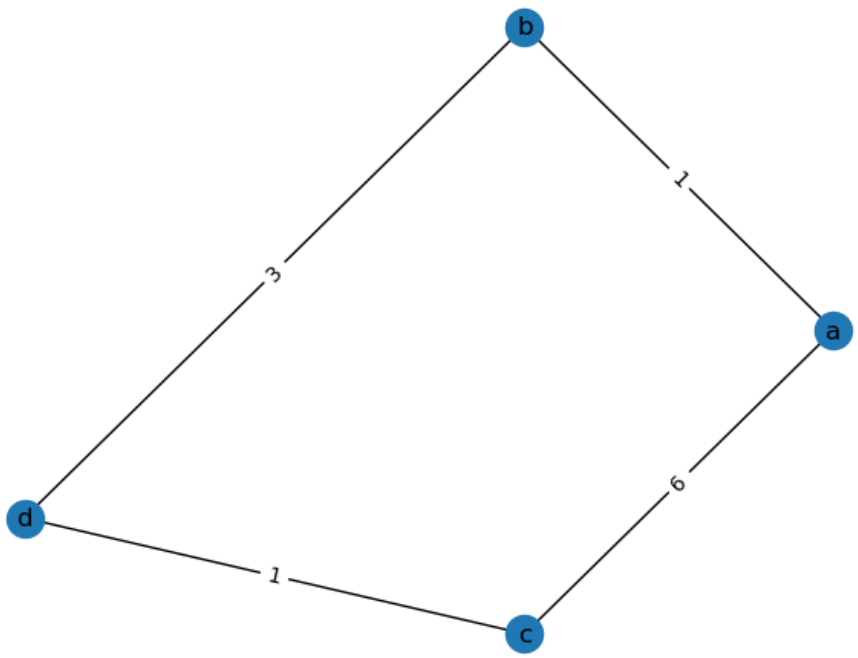

```
    # On nettoie toute la table de routage. Normalement, l'information que
    # l'ordinateur est inaccessible doit se propager dans le réseau de la
    # même façon que l'information sur les plus court chemins se propage.
    def pop_sommet(R, s):
    R.pop(s)
    for i in R.keys():
    R[i].pop(s)
    for j in R.keys():
    if R[i][j]['premier'] == s:
    R[i][j] = {'premier' : '?', 'weight' : inf}
    return R
```

```
    pop_sommet(R, 'e')
```

```
{'a': {'a': {'premier': 'a', 'weight': 0},
'b': {'premier': 'b', 'weight': 1},
'c': {'premier': 'b', 'weight': 4},
'd': {'premier': 'b', 'weight': 3}},
'b': {'a': {'premier': 'a', 'weight': 1},
'b': {'premier': 'b', 'weight': 0},
'c': {'premier': '?', 'weight': inf},
'd': {'premier': '?', 'weight': inf}},
'c': {'a': {'premier': 'd', 'weight': 4},
'b': {'premier': 'd', 'weight': 3},
'c': {'premier': 'c', 'weight': 0},
'd': {'premier': 'd', 'weight': 1}},
'd': {'a': {'premier': '?', 'weight': inf},
'b': {'premier': '?', 'weight': inf},
'c': {'premier': 'c', 'weight': 1},
'd': {'premier': 'd', 'weight': 0}}}
```

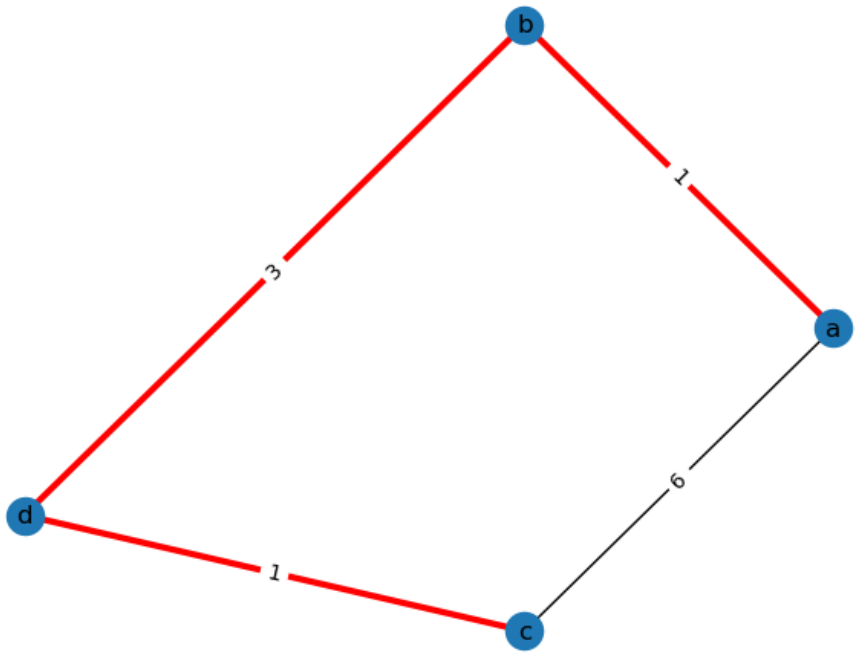
```
# 'a' ne sait plus comment joindre 'c'
c = plus_court_chemin(R, 'a', 'c')
print(c)
afficher_chemin(c)
```

```
['a', '?']
```



```
    # Après une étape de calcul, 'a' sait de nouveau joindre 'c' (et de␣
↪façon optimale !)
    R = bellmanford_step(G, R)
    c = plus_court_chemin(R, 'a', 'c')
    print(c)
    afficher_chemin(c)
```

```
['a', 'b', 'd', 'c']
```



Proposition 2.4 – Algorithme de Floyd – Warshall

Le principe de l'algorithme de Floyd – Warshall est le suivant.On a un graphe orienté pondéré (les poids pouvant être négatifs). Le poids de l'arc $(i,j) \in A$ *est noté* $\delta(i,j)$.
Notons $M^{(k)} = \left[m_{i,j}^{(k)}\right]$, $k \in [\![0, \text{Card}(S)]\!]$ *la matrice des poids minimaux en n'utilisant que les sommets intermédiaires (autres que* i *et* j*)* $\{1, \ldots, k\}$ *définie par :*

1. Initialisation
$$\forall (i,j) \in [\![1, \text{Card}(S)]\!]^2, \; m_{i,j}^{(0)} = \begin{cases} \delta(i,j) & si\ (i,j) \in A \\ \infty & sinon. \end{cases}$$
2. Hérédité
$$\forall k \in [\![1, \text{Card}(S)]\!], \; \forall (i,j) \in [\![1, \text{Card}(S)]\!]^2, \; m_{i,j}^{(k)} = \min\left(m_{i,j}^{(k-1)}, m_{i,k}^{(k-1)} + m_{k,j}^{(k-1)}\right).$$
3. Arrêt $k = \text{Card}(S)$.

4. Test de cyclicité *On refait l'algorithme une fois, si cela change, il y a un cycle de poids négatif.*

Exercice(s) 2.3

Montrer que l'algorithme s'arrête et produit les poids minimaux cherchés. Quelle est sa complexité ?