Plus courts chemins à source unique

Chaque jour partout dans le monde des millions de personnes utilisent des logiciels de calculs d'itinéraires pour se rendre d'un point A à un point B.

Ces logiciels sont basés sur des algorithmes de plus courts chemins dans des graphes.

1 Un exemple de graphe orienté pondéré

Un graphe orienté pondéré est un graphe orienté dont les arcs ont des poids (pas forcément positifs).

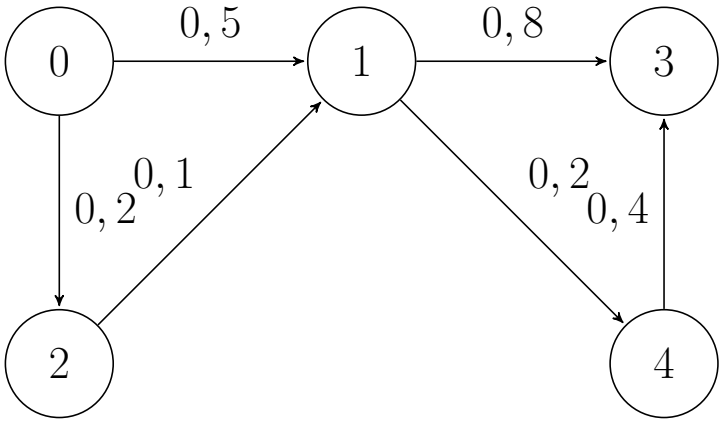
La **longueur d'un chemin** est la somme des poids des arcs composant ce chemin.

La seule contrainte demandée est qu'**il n'y a pas de cycle négatif dans le graphe orienté**.

Un cycle négatif est un cycle tel que la longueur du cycle est strictement négatif.

L'existence d'un cycle négatif ne permet pas de définir dans tous les cas de chemin le plus court d'une source vers un sommet.

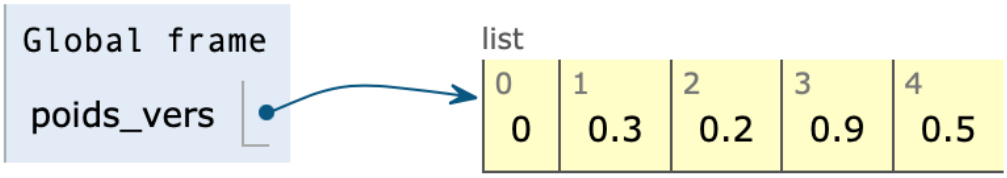
(Voir exercice)



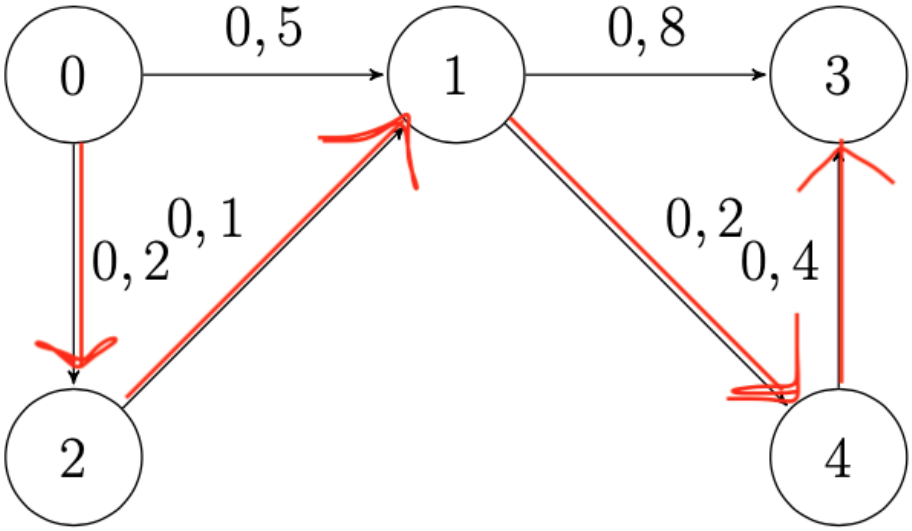
En partant de la source 0 les chemins les plus courts vers les autres sommets sont :

1. Pour aller vers 1 : le chemin 0 -> 2 -> 1 de poids 0,3
2. Pour aller vers 2 : le chemin 0 -> 2 de poids 0,2
3. Pour aller vers 3 : le chemin 0 -> 2 -> 1 -> 4 -> 3 de poids 0,9
4. Pour aller vers 4 : le chemin 0 -> 2 -> 1 -> 4 de poids 0,5

A la fin de la recherche des chemins les plus courts en partant de la source 0 on a un tableau `poids_vers` qui donne les poids de tous les chemins les plus courts de la source 0 vers un sommet i du graphe en exemple.



L'ensemble des chemins les plus courts de la source vers les autres sommets forme un **arbre**



Pour pouvoir travailler avec des arcs pondérés on va introduire une classe `Arc_P`

```
class Arc_P:
      def __init__(self,d,f,p):
        self.deb = d
        self.fin = f
        self.poids = p

    def origine(self):
        return self.deb

    def fin(self):
        return self.fin

    def poids(self):
        return self.poids
```

On implémente en Python une classe `Graphe_OP` pour graphe orienté pondéré

```
class Graphe_OP:
      def __init__(self,n):
```

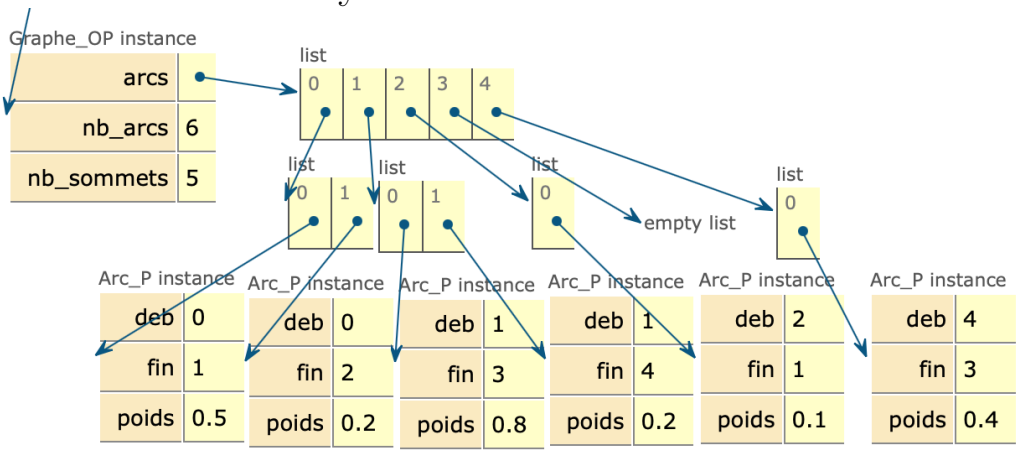
```
        self.nb_sommets = n
        self.nb_arcs = 0
        self.arcs =  \
        [[]for i in range(nb_sommets)]

    def ajoute_arc(self,i:int,arc_p:Arc_P):
        self.arcs[i] .append( arc_p)
        self.nb_arcs += 1

    def voisins(self,i):
        return self.arcs[i]

    def __str__(self):
        ch = str(self.nb_sommets)+\
        " sommets "+str(self.nb_arcs)+\
        " arcs \n"
        for i in range(self.nb_sommets):
            ch = ch + str(i) + " --> "
            for arc in self.arcs[i]:
                ch = ch + str(arc.fin)+\
                 "("+str(arc.poids)+")"+" "
            ch = ch +"\n"
        return ch
```

Voici l'instanciation du graphe en exemple ci-dessus

```
g = Graphe_OP(5)
g.ajoute_arc(0,Arc_P(0,1,0.5))
g.ajoute_arc(0,Arc_P(0,2,0.2))
g.ajoute_arc(1,Arc_P(1,3,0.8))
g.ajoute_arc(1,Arc_P(1,4,0.2))
g.ajoute_arc(2,Arc_P(2,1,0.1))
```

```
g.ajoute_arc(4,Arc_P(4,3,0.4))
```

On observe dans Python Tutor



On obtient dans la console lorsqu'on fait `print(g)`

```
5 sommets 6 arcs
0 --> 1(0.5) 2(0.2)
1 --> 3(0.8) 4(0.2)
2 --> 1(0.1)
3 -->
4 --> 3(0.4)
```

2 Algorithme de plus court chemin en partant d'une source

Définition 1. *Etant donné un graphe pondéré G et une source s de ce graphe.*

Un chemin de la source s vers un sommet i est une succession d'arcs permettant de passer de s à i.

$s \xrightarrow{p1} s1 \xrightarrow{p2} s2 \ldots. \xrightarrow{pn} i$

On notera un chemin de s vers i $s \rightsquigarrow i$

Le poids d'un chemin p est la somme $l(p)$ des poids des arcs composant ce chemin.

Le ***poids du plus court chemin*** *entre la source s et un sommet i de ce graphe est défini par :*

$\delta(s,i) = min(l(p) : s \rightsquigarrow i)$ *où* $l(p)$ *est la longueur d'un chemin* p,

Sinon $\delta(s,i) = \infty$

Un plus court chemin *de la source s vers un sommet i est un chemin quelconque p de longueur* $l(p)$ *tel que* $\delta(s,i) = l(p)$

Théorème 1. *(Conditions d'optimalité)*

G un graphe orienté pondéré sans cycle négatif et s le sommet source.

Le tableau `poids_vers` *donne le tableau des* ***poids des chemins les plus courts en partant de s*** *si et seulement si :*

1. `poids_vers[s] = 0`
2. *Pour tout sommets v et w , accessibles depuis la source s*

   `poids_vers[v]` $\leqslant$ `poids_vers[w]` $+$ `poids(w->v)`

Preuve 1. *Supposons que v et w sont accessibles depuis s avec :*

`poids_vers[v]` $>$ `poids_vers[w]` $+$ `poids(w->v)`*, dans ce cas le tableau ne contient pas les* ***poids des chemins les plus courts en partant de s*** *car* `poids_vers[v]` *ne contient pas le chemin de poids minimal en partant de s vers v car le chemin en partant de s vers v en passant par w a un poids* `poids_vers[w]` $+$ `poids(w->v)` *strictement inférieur à* `poids_vers[v]`

Définition 2. *Le relâchement d'un arc u -> v consiste à tester si l'on peut améliorer le plus court chemin de la source s à v en passant par u.*

Algorithme 1 : Relâchement d'un arc u-> v

```
relâchement (u->v)
début
    Données : Un arc pondéré u -> v, un tableau
              poids_vers
    Résultat : Rien
1   si poids_vers[v] > poids_vers[u] +poids(u->v) alors
2       poids_vers[v] ← poids_vers[u] +poids(u->v)
3       //fil d'ariane pour construire un chemin le plus
         court
4       predecesseur[v] ← u
5   fin
fin
```

D'où un algorithme très général pour rechercher les meilleurs chemins dans un graphe pondéré sans cycle négatif, en partant d'une source.

Algorithme 2 : Meilleurs chemins en partant de s

```
meilleurs_chemins (G,s)
début
    Données : Un graphe orienté pondéré, une source s
    Résultat : Rien
1   poids_vers[s] ← 0
2   Pour tout autre sommet poids_vers[w] ← inf
3   tant que conditions optimalité non vérifiées faire
4       Relâcher tous les arcs du graphe
5   fin
fin
```

Cet algorithme est très général au sens où il ne précise pas de méthode pour relâcher les arcs du graphe pondéré.

3 Algorithme de Dijkstra

On suppose les poids positifs.

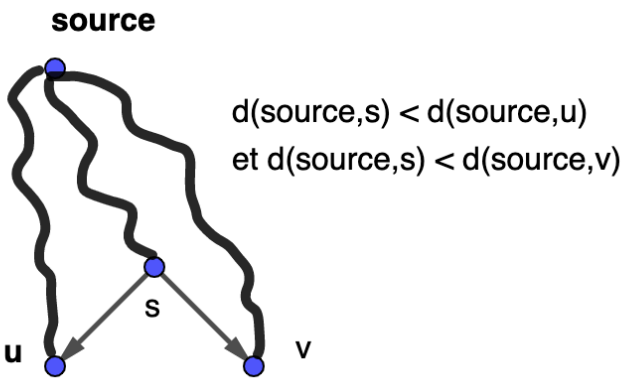
Rappelons le parcours en largeur dans un graphe orienté (ou non orienté).

La propriété intéressante du parcours en largeur d'un graphe orienté non pondéré, en partant d'un sommet particulier, appelé source, est la visite des sommets en partant de s en vagues successives, en premier les sommets voisins de s à distance 1, puis à distance 2 etc...

Algorithme 3 : Parcours en largeur d'un graphe à partir d'une source

```
   parcours_largeur (G,s)
   début
   |  Données : Un graphe G, une source s un entier
   |  Résultat : Rien
1  |  f = file_vide()
2  |  enfiler(f,s)
3  |  marquer(s)
4  |  tant que  non (est-vide(f)) faire
5  |  |  x = defiler(f)
6  |  |  afficher(x)
7  |  |  pour chaque voisin de x non marqué faire
8  |  |  |  enfiler(f,voisin)
9  |  |  |  marquer(voisin)
10 |  |  fin
11 |  fin
   fin
```

L'algorithme de Dijkstra, généralise le parcours en largeur, en utilisant une **file de priorité** à la place d'une file pour organiser le relâchement des arcs, **dans l'ordre des distances des origines des arcs à la source**



Autrement dit tous les arcs ne sont relâchés qu'une fois, et ils le sont dans l'ordre croissant des distances des origines des arcs à la source.

Qu'est ce qu'une file de priorité ?

C'est une file où chaque élément de la file a une clé.

Imaginons un instant une file à une caisse de supermarché réservé à des clients prioritaires comme femmes enceintes, personnes handicapées, pompiers,....

Si une de ces personnes prioritaires arrive dans la file à un moment donné, elle passe toute la file et sera servie en premier.

Dans notre problème la clé associé à un sommet s sera la distance de la source au sommet s, ainsi l'élément qui sortira de la file en priorité sera celui dont la clé est la plus petite possible.

On parle alors d'une file de priorité **minimale**.

1. On met dans une file de priorité minimale la source et comme clé poids_vers[source] = 0
2. Tant que la file n'est pas vide on prend l'élément prioritaire r de la file de priorité, on relâche tous les arcs d'origine r et on insère dans la file les extrémités de ces arcs avec leurs clés ou on met à jour les clés des éléments qui ont été concernés par le relâchement des arcs d'origine r.

Voici l'algorithme de Dijkstra :

Algorithme 4 : Algorithme de Dijkstra

```
dijkstra (G,s)
début
    Données : Un graphe orienté pondéré, une source s
    Résultat : Rien
1   poids_vers[s] ← 0
2   poids_vers[w] ← inf
3   file ← file_priorité_min()
4   insérer la source et sa clé 0 dans la file
5   tant que file non vide faire
6       r ← elt_prioritaire(file)
7       pour chaque voisin v de racine faire
8           //en relâchant on met à jour la file de priorité
9           //s'il y a eu relachement
10          //si v est dans la file on diminue sa clé
11          //sinon on met v dans la file
12          relâcher(r,v)
13      fin
14  fin
fin
```

On exécute l'algorithme de Dijkstra dans un tableau sur le graphe pondéré de la page 3 en prenant comme source le sommet 0.

1. **Initialisation**

   poids_vers = [0,∞,∞,∞,∞]

   file = [[0,0]]

2. **Premier tour de boucle**

   Est ce que la file est vide ?

   Non. L'élément prioritaire est le sommet 0.

On l'enlève de la file donc à ce stade file = []

Les voisins du sommet 0 c'est à dire les sommets accessibles par un arc à partir de 0 sont les sommets 1 et 2.

On va **relâcher les arcs 0->1 et 0->2**.

Relâcher l'arc 0->1 c'est comparer poids_vers[1] = ∞ à poids_vers[0] + poids(0->1) = 0 + 0.5 = 0.5

Puisque poids_vers[1] > poids_vers[0] + poids(0->1) on met à jour poids_vers[1].

poids_vers[1] = 0.5

Le sommet 1 n'est pas dans la file on le met avec sa clé 0.5 donc file = [[1,0.5]]

Relâcher l'arc 0->2 c'est comparer poids_vers[2] = ∞ à poids_vers[0] + poids(0->2) = 0 + 0.2 = 0.2

Puisque poids_vers[2] > poids_vers[0] + poids(0->2) on met à jour poids_vers[2].

poids_vers[2] = 0.2

Le sommet 2 n'est pas dans la file on le met avec sa clé 0.2 donc file = [[1,0.5],[[2,0.2]]

3. **Deuxième tour de boucle**

   Est ce que la file est vide ?

   Non. Il y a deux éléments dans la file.

   Le sommet prioritaire est 2 avec la clé minimale 0.2.

   On l'enlève de la file donc file = [[1,0.5]]

   Les voisins du sommet 2, c'est à dire les sommets accessibles par un arc à partir de 2, il n'y a qu'un seul c'est le sommet 1.

   On va **relâcher l'arc 2->1**.

   Relâcher l'arc 2->1 c'est comparer poids_vers[1] = 0.5 à poids_vers[2] + poids(2->1) = 0.2 + 0.1 = 0.3

Puisque poids_vers[1] > poids_vers[2] + poids(2->1) on met à jour poids_vers[1].

poids_vers[1] = 0.3

Le sommet 1 est déjà dans la file il faut mettre à jour la clé du sommet 1

Donc file = [[1,0.3]]

4. **Troisième tour de boucle**

   Est ce que la file est vide ?

   Non. Il y a un seul élément dans la file le sommet 1.

   On l'enlève de la file donc file = [].

   Ses voisins, c'est à dire les sommets accessibles par un arc à partir de 1, sont les sommets 3 et 4.

   On va **relâcher l'arc 1->3 et l'arc 1 >4**.

   Relâcher l'arc 1->3 c'est comparer poids_vers[3] = ∞ à poids_vers[1] + poids(1->3) = 0.3 + 0.8 = 1.1

   Puisque poids_vers[3] > poids_vers[1] + poids(1->3) on met à jour poids_vers[3].

   poids_vers[3] = 1.1

   Le sommet 3 n'est pas dans la file

   Donc file = [[3,1.1]]

   Relâcher l'arc 1->4 c'est comparer poids_vers[4] = ∞ à poids_vers[1] + poids(1->4) = 0.3 + 0.2 = 0.5

   Puisque poids_vers[4] > poids_vers[1] + poids(1->4) on met à jour poids_vers[4].

   poids_vers[4] = 0.5

   Le sommet 4 n'est pas dans la file

   Donc file = [[3,1.1], [4,0.5]]

5. **Quatrième tour de boucle**

Est ce que la file est vide ?

Il y a deux éléments dans la file [[3,1.1], [4,0.5]]

Le sommet prioritaire est 4 avec la clé minimale 0.5.

On l'enlève de la file donc file = [[3,1.1]]

Les voisins du sommet 4, c'est à dire les sommets accessibles par un arc à partir de 4, il n'y a qu'un seul c'est le sommet 3.

On va **relâcher l'arc 4->3**.

Relâcher l'arc 4->3 c'est comparer poids_vers[3] = 1.1 à poids_vers[4] + poids(4->3) = 0.5 + 0.4 = 0.9

Puisque poids_vers[3] > poids_vers[4] + poids(4->3) on met à jour poids_vers[3].

poids_vers[3] = 0.9

Le sommet 3 est déjà dans la file il faut mettre à jour la clé du sommet 3

Donc file = [[3,0.9]]

6. **Cinquième tour de boucle**

   Est ce que la file est vide ?

   Non file = [[3,0.9]]

   Le sommet prioritaire est 3 .

   On l'enlève de la file donc file = []

   Il n'a pas de voisins donc il n'y a pas d'arc à relâcher.

   A la fin de ce cinquième tour de boucle la file est vide.

   Donc l'algorithme s'arrête.

Première question : Cet algorithme s'arrête -t-il toujours ?

Oui, c'est un algorithme de type **glouton** :

Tous les sommets du graphe, accessibles depuis la source par un chemin, passeront dans la file de priorité et finiront par y

sortir car ils seront à un moment donné l'élément prioritaire de la file.

Donc à un moment la file sera vide et l'algorithme s'arrête.

Deuxième question : Comment être sûr qu'à la fin de l'algorithme on a dans le tableau poids_vers les poids minimaux des chemins de la source vers les autres sommets ?

Voici la propriété (invariant de boucle) qu'il faut prouver :

Lorsqu'on enlève de la file l'élément prioritaire s, sa clé est la distance minimale de la source à s = $\delta(source, s)$

Par récurrence :

1. Le premier élément prioritaire est la source et lorsqu'on l'enlève de la file poids_vers[source] = 0 = δ(source,source)
2. (Hérédité) Supposons que la propriété est vraie jusqu'à un certain nombre k de tours de boucle et montrons que cela reste vrai au $k+1$ ième tour de boucle.

   Autrement dit s_1 = source, ..., s_k sont passés dans la file et n'y sont plus par contre

   poids_vers[s_i] = $\delta(source, s_i)$ pour $1 \leqslant i \leqslant k$

   Montrons que la propriété sera vraie pour s_{k+1}.

   Pourquoi s_{k+1} est dans la file ?

   Parce qu'il existe **au moins un** sommet s_j passé avant lui dans la file de telle sorte qu'il y a eu relâchement de l'arc s_j->s_{k+1}

   Supposons qu'il y a eu deux tels sommetss_{j1} et s_{j2}, avec s_{j2} pour le dernier relâchement d'un arc pointant vers s_{k+1} et d'origine un des sommets dans la file avant s_{k+1}.

   Puisque s_{j2} est l'origine du relâchement du dernier arc alors

poids_vers[s_{k+1}] = poids_vers[s_{j2}] + poids(s_{j2}->s_{k+1}) = $\delta(source, s_{j2})$ + poids(s_{j2}->s_{k+1})

C'est le mieux qu'on puisse faire pour l'instant sur tous les chemins de la source à s_{k+1} et dont les sommets intermédiaires sont passés dans la file et y sont sortis avant s_{k+1}

Est il possible qu'un sommet s_l qui passe après s_{k+1} dans la file permette de construire un chemin source $\rightsquigarrow s_l \rightsquigarrow s_{k+1}$ de poids strictement inférieur à poids_vers[s_{k+1}] ?

Non car si s_l n'est pas sorti de la file avant s_{k+1} ou y entre après que s_{k+1} y est sorti c'est que :

La clé de s_l qui est poids_vers[s_l] est supérieure ou égale à la clé de s_{k+1} qui est poids_vers[s_{k+1}] or **les poids des arcs sont strictement positifs** donc le poids du chemin source $\rightsquigarrow s_l \rightsquigarrow s_{k+1}$ est :

poids_vers[s_l] + poids du chemin ($s_l \rightsquigarrow s_{k+1}$) > poids_vers[$s_{k+1}$]

Donc poids_vers[s_{k+1}] = $\delta(source, s_{k+1})$

Conclusion : A la fin de l'algorithme le tableau `poids_vers` contient bien les poids des chemins les plus courts de la source vers tous les autres sommets.

Quel est le coût de cet algorithme

La boucle principale (ligne 5) est exécutée n fois où n est le nombre de sommets accessibles à partir de la source (la source comprise).

Pour simplifier supposons que tous les sommets du graphe sont accessibles donc la boucle sera exécutée |S| fois où |S| est le nombre de sommets du graphe.

Dans la boucle ensuite il faut chercher l'élément prioritaire de la file de priorité et l'enlever de la file (ligne 6) puis relâcher tous les arcs issus de cet élément (ligne 12).

Or **chaque arc sera relâché une et une seule fois** donc le coût total du relâchement de tous les arcs sera |S|*|A| où |A| est le nombre d'arcs du graphe.

Le coût de l'algorithme de Dijkstra dépend donc du coût de la recherche de l'élément prioritaire et de l'extraction de cet élément dans la file de priorité.

Si pour rechercher la clé minimum il faut parcourir le tableau `poids_vers` à chaque fois alors le côut de l'algorithme de Dijkstra sera proportionnel à |S|*|S| + |S|*|A| = |S|*(|S| + |A|)

1. Si pour rechercher la clé minimum il faut parcourir le tableau `poids_vers` à chaque fois alors le côut de l'algorithme de Dijkstra sera proportionnel à |S|*|S| + |S|*|A| = |S|*(|S| + |A|)
2. On peut utiliser un tas min (module heapq de Python) pour implémenter la file de priorité et dans ce cas le coût de l'algorithme est proportionnel à ln(|S|)*(|S| + |A|)

4 Algorithme de Dijkstra dans un DAG

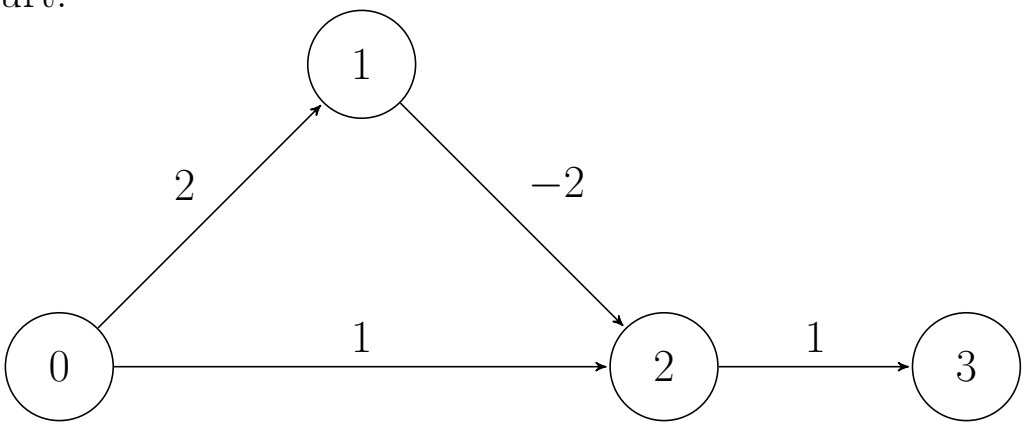
Si le graphe est un DAG l'algorithme de Dijkstra est plus efficace en relâchant les arcs selon leur ordre dans le tri topologique.

5 Algorithme de Bellman-Ford

Les poids peuvent être négatifs. Mais il n'y a pas de cycle négatif.

On observe sur le graphe suivant que l'algorithme de Dijkstra ne donne pas le chemin le plus court pour aller de 0 à 3.

L'algorithme de Dijkstra donne le chemin 0-2-3 avec un poids de 2 alors que le chemin 0-1-2-3 avec un poids de 1 est plus court.



Lemme 1. *(Propriété de relâchement de chemin)(admis)*

Si on a un chemin ***le plus court*** *de la source s vers un sommet i,*

$s = u_0 \to u_1 \to u_2 ..... \to u_k = i$

Si on relâche les arcs dans l'ordre $u_0 \to u_1$, $u_1 \to u_2$, ..., $u_{k-1} \to u_k$ *alors à la fin on obtient poids_vers[i] =* $\delta(s,i)$

L'Algorithme de Bellman-Ford est basée sur l'idée suivante :

S'il existe un chemin le plus court entre une source s et un sommet i, ce chemin ne contient pas de cycle (chemin simple) et par conséquent il a au plus n - 1 arcs où n est le nombre de sommets de ce graphe.

Supposons le pire des cas où il a n-1 arcs :

s $\to u_1 \to u_2 ..... \to u_{n-1} =$ i

Si on répète n-1 fois relâcher tous les arcs du graphe (dans n'importe quel ordre mais toujours le même) alors :

A la première itération on aura relâché forcément s $\to u_1$ parmi les autres arcs

A la deuxième $u_1 \to u_2$, donc pour l'instant on respecte l'ordre des relâchements s $\to u_1$ puis $u_1 \to u_2$ pour pouvoir

utiliser la propriété de relâchement de chemin

Et ainsi de suite jusqu'à la n-1 itération où on aura relâché $u_{n-2} \rightarrow u_{n-1}$

On aura fait cela pour tous les chemins de s vers i donc parmi un de ces chemins il y aura un chemin le plus court donc on est sûr d'après la propriété de relâchement de chemin d'avoir à la fin des n-1 itérations à la fois la distance la plus courte de s à i et aussi le prédécesseur de i permettant d'avoir un chemin le plus court de s à i.

Ceci est vrai pour tous les sommets i autre que la source.

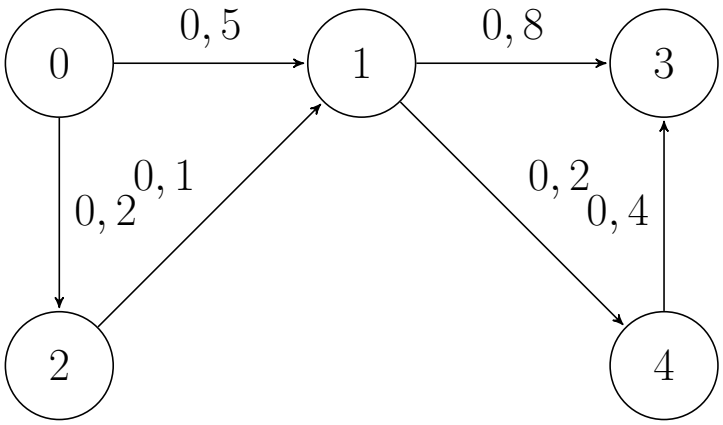
Voici l'algorithme de Bellman-Ford en pseudo-code :

Algorithme 5 : Algorithme de Bellman-Ford

```
Bellman-Ford (G,s)
début
    Données : Un graphe orienté pondéré de n sommets,
              une source s
    Résultat : Rien
1   poids_vers[s] ← 0
2   Pour tout autre sommet poids_vers[w] ← inf
3   pour i ← 1 jusqu'à n-1 faire
4       Relâcher tous les arcs du graphe dans n'importe
        quel ordre(mais toujours le même)
5   fin
fin
```

Appliquons l'algorithme de Bellman-Ford sur le graphe suivant en relâchant les arcs dans l'ordre arbitraire suivant :

(4,3) - (1,3) - (1,4) - (2,1) - (0,2) - (0-1)



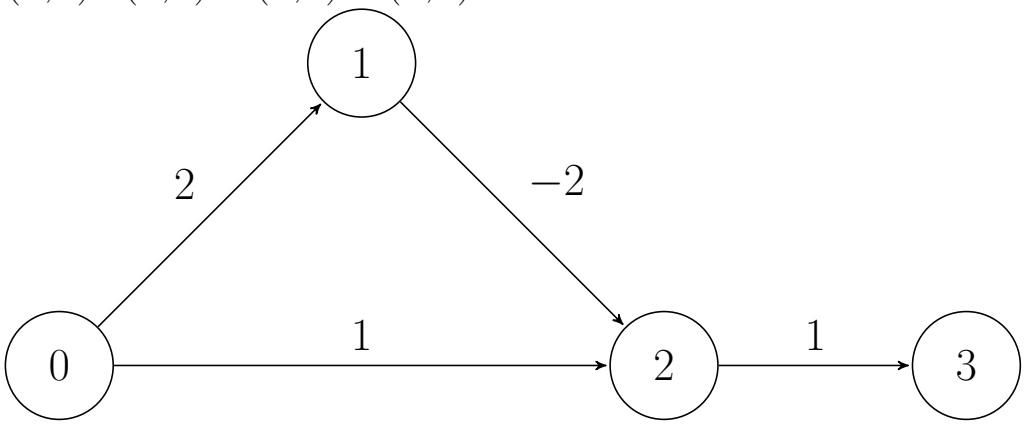
Il y a cinq sommets on va donc répéter 4 fois de l'étape 1 à l'étape 4

étape	0	1	2	3	4
0	0	∞	∞	∞	∞
1	0	0.5	0.2	∞	∞
2	0	0.3	0.2	1.3	0.7
3	0	0.3	0.2	1.1	0.5
4	0	0.3	0.2	0.9	0.5

On sait que dans le graphe suivant le chemin le plus court du sommet 0 au sommet 3 est constitué des arcs (0,1)- (1,2) - (2,3)

Appliquons l'algorithme de Bellman-Ford en relâchant les arcs dans l'ordre :

(2,3)- (1,2) - (0,1) - (0,2)



On obtient le tableau suivant :

étape	0	1	2	3
0	0	∞	∞	∞
1	0	2	1	∞
2	0	2	0	2
3	0	2	0	1

Exercices

Ex 1

Supposons que dans un graphe orienté pondéré il y ait un cycle négatif .

Etant donné une source s , et un sommet w tel qu'il existe un chemin de s vers w et tel que w appartient au cycle négatif, montrer qu'il n'existe pas de plus court chemin de s vers w.

Ex 2

Implémenter la classe `Arc_P` et la classe `Graphe_OP`.

Puis écrire une fonction `dijkstra(G:Graphe_OP,source:int)` et qui renvoie deux tableaux, celui contenant les poids des chemins les plus courts en partant de la source et les prédécesseurs de chaque sommet.

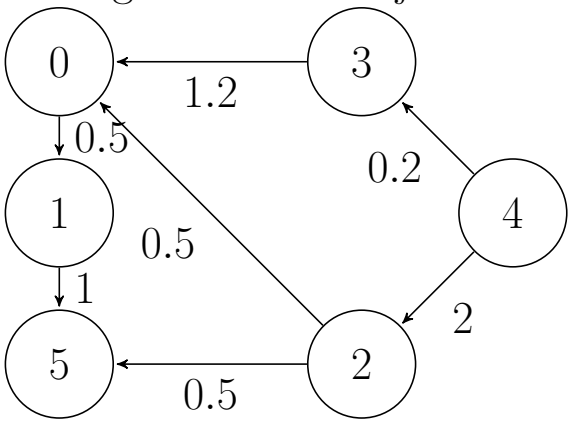
Ex 3

Découvrir le module heapq de Python, implémentant une file de priorité avec un tas min. Voir la documentation Python.

Ecrire une fonction `dijkstra2(G:Graphe_OP,source:int)` avec le module heapq et qui renvoie deux tableaux, celui contenant les poids des chemins les plus courts en partant de la source et les prédécesseurs de chaque sommet.

Ex 4

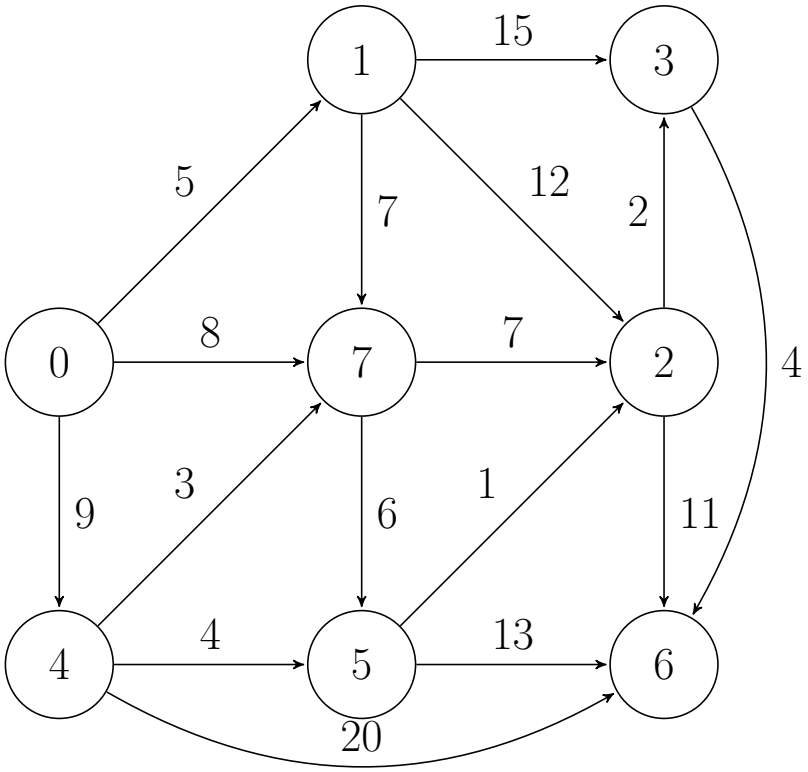
Faire le tri topologique du graphe orienté suivant puis appliquer l'algorithme de Dijkstra en prenant pour source s = 2



Ex 5

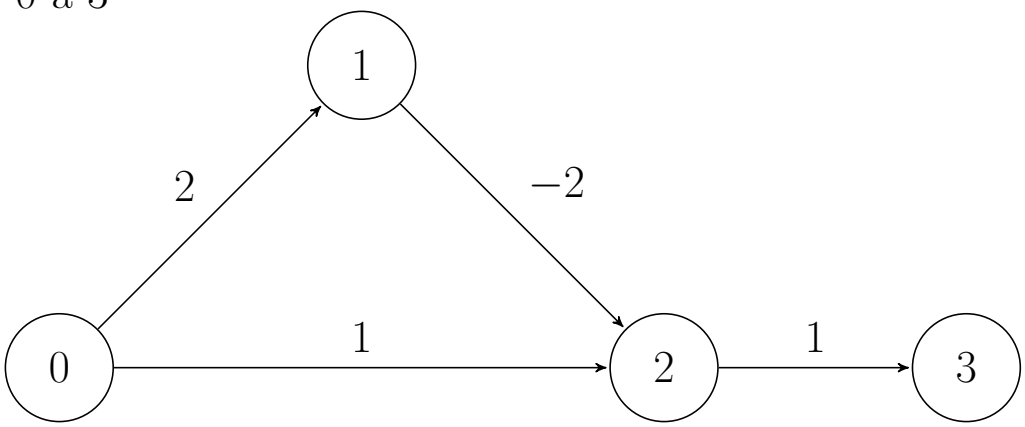
Appliquer l'algorithme de Dijkstra sur le graphe suivant en partant de la source :

1. s = 0
2. s = 4



Ex 6

Appliquer l'algorithme de Dijkstra sur le graphe suivant et observer qu'il ne donne pas le chemin le plus court pour aller de 0 à 3



Ex 7

Voici une implémentation de l'algorithme de Dijkstra dans le langage Java.

Bien observer la structure générale de l'algorithme.

```
public DijkstraUndirectedSP(EdgeWeightedGraph G, int s) {
    for (Edge e : G.edges()) {
        if (e.weight() < 0)
            throw new IllegalArgumentException("edge " + e + " has negative weight");
    }

    distTo = new double[G.V()];
    edgeTo = new Edge[G.V()];
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
    distTo[s] = 0.0;

    // relax vertices in order of distance from s
    pq = new IndexMinPQ<Double>(G.V());
    pq.insert(s, distTo[s]);
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        for (Edge e : G.adj(v))
            relax(e, v);
    }

    // check optimality conditions
    assert check(G, s);
}

// relax edge e and update pq if changed
private void relax(Edge e, int v) {
    int w = e.other(v);
    if (distTo[w] > distTo[v] + e.weight()) {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
        else                pq.insert(w, distTo[w]);
    }
}
```

Ex 8

Prouver le lemme sur la propriété du relâchement des arcs d'un chemin le plus court.

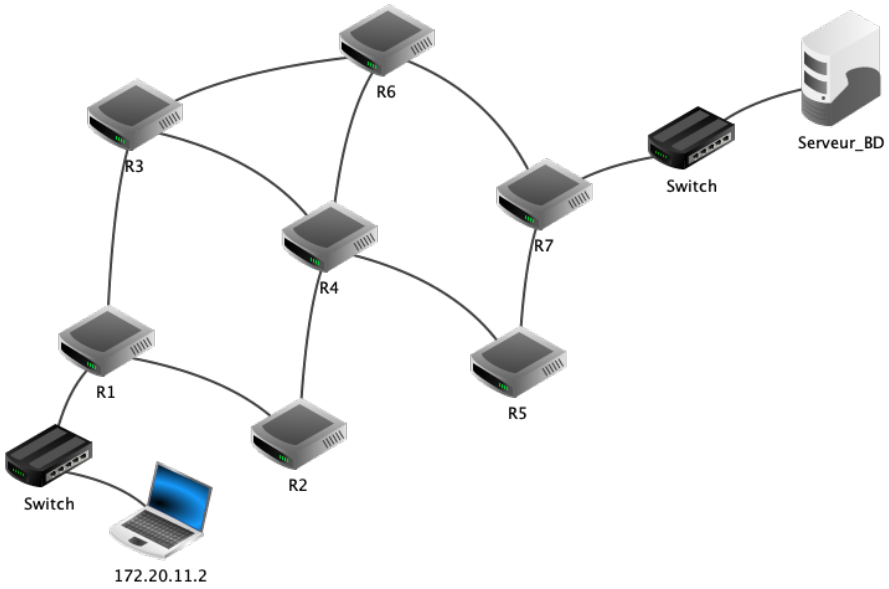
Ex 9

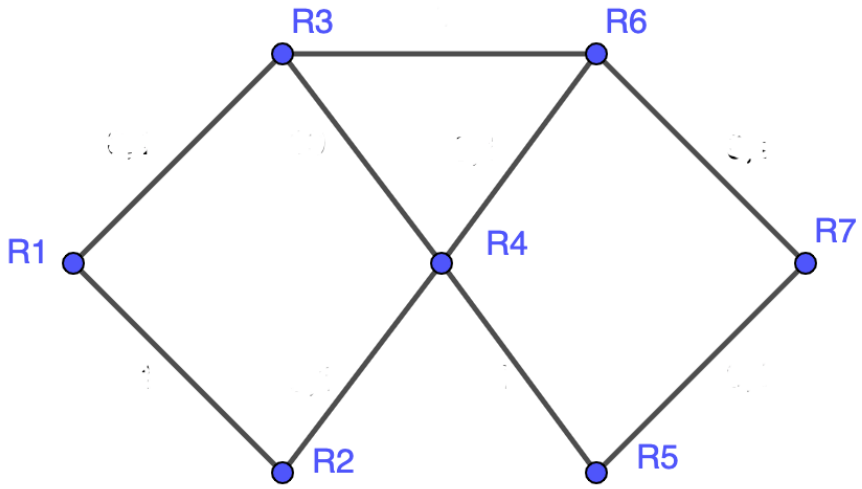
Ecrire une fonction `bellmann_ford(G:Graphe_OP,source:int)` et qui renvoie deux tableaux, celui contenant les poids des chemins les plus courts en partant de la source et les prédécesseurs de chaque sommet.

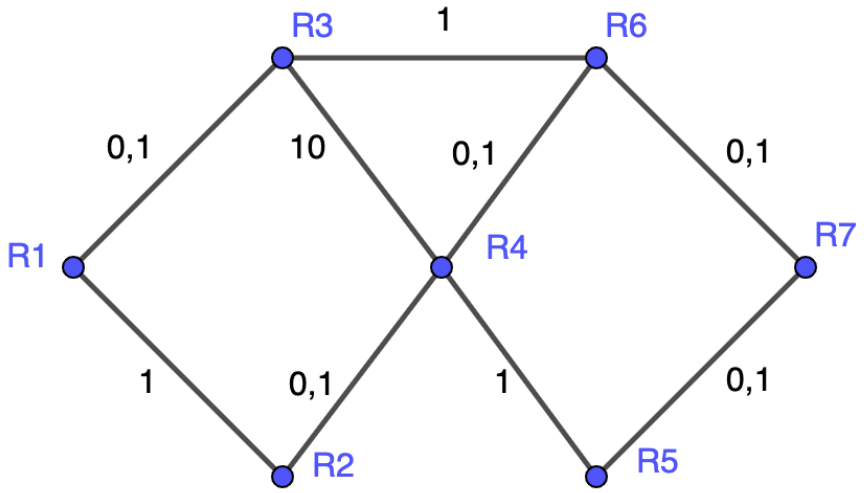
Ex 10

On a modélisé le réseau des routeurs sur un campus par un

graphe non orienté pondéré des coûts des connexions entre les routeurs.







Appliquer l'algorithme de Dijkstra en partant de la source R1, et en déduire le chemin le plus court entre les routeurs R1 et R7.