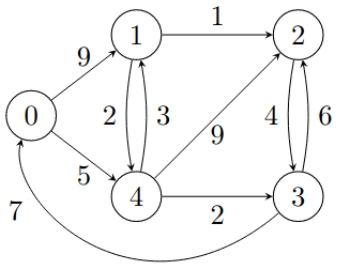
ITC - PC

Plus court chemin dans un graphe : Algorithme de Dijkstra

On s'interesse ici à la recherche d'un plus court chemin dans un graphe pondéré **avec des poids positifs**. Le chemin le plus court est celui de coût, c'est-à-dire la somme des poids des arêtes, le plus faible.

I - Implémentation d'un graphe pondéré orienté

La figure suivante présente un exemple de graphe pondéré orienté :



- Dans le cas d'une implémentation creuse, il suffit dans la liste d'adjacence d'un sommet u, de stocker des couples (v, p) à la place du seul sommet v.

```
>>> G = [[(1, 9), (4, 5)], [(2, 1), (4, 2)], [(3, 4)],
[(0, 7), (2, 6)], [(1, 3), (2, 9), (3, 2)]]
```

- Dans le cas d'une implémentation dense, on utilise une matrice d'adjacence est maintenant à valeurs dans $\mathbb{R} \cup \{+\infty\}$. Pour cela on rappelle que le type flottant possède une valeur $+\infty$ que l'on peut obtenir comme `float('inf')`.

```
>>> G = [[0, 9, inf, inf, 5],
         [inf, 0, 1, inf, 2],
         [inf, inf, 0, 4, inf],
         [7, inf, 6, 0, inf],
         [inf, 3, 9, 2, 0]]
```

> **Remarque**
> Attention à ne pas confondre la matrice d'adjacence dans les graphes pondérés et non pondérés : les zéros dans la matrice d'adjacence d'un graphe non pondéré deviennent des $+\infty$ dans la matrice d'un graphe pondéré, sauf sur la diagonale où ils restent des zéros.
> On exclura complètement la présence de boucles dans le contexte des graphes pondérés, la notion de coût pour aller d'un sommet à lui-même se devant d'être zéro !

II - Algorithme de Dijkstra

L'algorithme de Dijkstra que l'on va voir est une généralisation du parcours en largeur : il consiste également à traiter les sommets un par un, par poids depuis l'origine s croissants. C'est donc un algorithme glouton.

II.1 - Principe

Si le plus court chemin entre deux sommets D et A passe par un sommet I, alors la partie de ce chemin entre D et I est le plus court chemin de D à I, et la partie entre I et A est le plus court chemin entre I et A. À chaque étape, on effectue donc le meilleur choix possible.
L'algorithme est semblable à celui d'un parcours en largeur d'abord, mais au lieu d'utiliser une file pour les sommets en attente, on utilise une file de priorité. Cela signifie qu'on extrait le sommet ayant la priorité, dans ce cas c'est celui qui correspond à la distance minimale. :

II.2 - exemple

On considère le graphe représenté ci-dessus et on cherche le plus court chemin entre le sommet 0 et chacun des autres sommets du graphe. On affecte la valeur $+\infty$ à chaque sommet, sauf au sommet 0 de départ, à qui on affecte la valeur 0. À chaque étape :

1. On choisit le sommet dont la distance depuis A dans le tableau est minimale.
2. On regarde ses différents voisins encore accessibles (c'est-à-dire qui n'ont pas déjà été choisis).
3. On compare la distance avec laquelle on arrive aux différents voisins depuis ce sommet, à la distance avec laquelle on avait pu y arriver jusque là (l'infini, ou une autre distance par un autre chemin). On garde la distance minimale avec laquelle on peut arriver à ce voisin.

II.3 - Implémentation

Exercice 1

Ecrire une fonction `minimum(d)` prenant comme argument un dictionnaire dont les valeurs sont entières ou $+\infty$ et renvoyant une clé de valeur minimale.

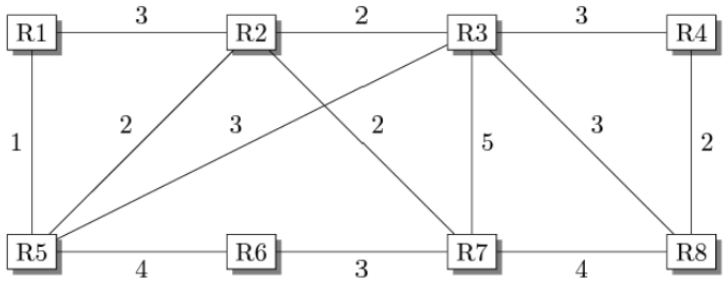
```
def Dijkstra (graphe , S) :
    """
    graphe : dictionnaire des listes d'adjacence
    sommet : sommet de départ
    """
    D ={} # dictionnaire solution
    dist = {cle : float('inf') for cle in graphe} # initialisation du dictionna
    # distances
    dist[S] = 0 # sommet S de départ, distance = 0
    while len(dist) > 0 : # tant que d n'est pas vide, il reste
    # des sommets a visiter
        smin = minimum(dist) # sommet de distance minimale
        D[smin] = dist[smin] # on copie le sommet smin et
        # sa distance dans le dictionnaire des résultats
        for k in range(len(graphe[smin])) : # parcours des voisins de smin
            v, d = graphe[smin][k]
            if v not in res : # on ne s'intéresse qu'aux voisins # non déjà chois
                dist [v]= min(dist[v], dist[smin] + d )
            del dist[smin] # on supprime smin de dist, qui contient les sommets
            # qu'il reste a visiter
    return res
```

Version sans commentaire

```
def Dijkstra (graphe , S) :
    """
    graphe : dictionnaire des listes d'adjacence
    sommet : sommet de départ
    """
    D ={}
    dist = {cle : float('inf') for cle in graphe}
    dist[S] = 0
    while len(dist) > 0 :
        smin = minimum(dist)
        D[smin] = dist[smin]
        for k in range(len(graphe[smin])) :
            v, d = graphe[smin][k]
            if v not in res :
                dist [v]= min(dist[v], dist[smin] + d)
            del dist[smin]
    return res
```

Exercice 2

Voici un réseau. On cherche les plus courtes distances entre chaque sommet et R8. Appliquer l'algorithme de Dijkstra "à la main" en construisant un tableau des distances.



Exercice 3 (*Version avec matrice d'adjacence*)

1. Ecrire une fonction `voisin(G, s)` déterminant la liste des voisins de `s` dans `G`.
2. En déduire une adaptation de la fonction `Djiskstra` pour qu'elle fonctionne avec un graphe définit par une matrice d'adjacence.