On rapelle la définition d'un graphe **valué** :

Définition 1. Un graphe $G = (V, E)$ est dit valué (ou pondéré) sur ses arêtes lorsqu'on lui associe une fonction de valuation (poids), $W : E \to \mathbb{R}$.
Ainsi, un graphe valué (sur ses arêtes) est une triplet $G = (V, E, W)$.

Adapter la classe "Graphe" et ses méthodes pour prendre en charge les graphes valués. On pourra pour ce faire représenter un graphe valué comme un dictionnaire de dictionnaires :

```
graphe = {"A " :{"C": 1},
          "B" : {"C" : 2, "E" : 3},
          "C" : {"A" : 1, "B" : 2, "D" : 4, "E" : 2},
          "D" : {"C" : 4},
          "E" : {"C" : 2, "B" : 3},
          "F" : {}
        }
```

On considère un graphe valué (avec des poids positifs) et sans boucles. Mettre en oeuvre et tester (sur un graphe orienté et le graphe non orienté correspondant)) l'algorithme de Dijkstra permettant de trouver le plus court chemin d'un sommet quelconques à tous ls autres :

Algorithme 1 : Algorithme de Dijkstra

Données : Un graphe orienté ou pas $G = (V, E)$ d'ordre $n > 0$, valué ($W > 0$) et sans boucles. Un sommet s d'où l'on débute le parcours. Les sommets sont numérotés de 1 à $n = |V|$, i.e. $V = \{1, 2, \dots, n\}$.

Résultat : Une liste D de distances telle que $D[v]$ soit la distance de s à v en suivant le plus court chemin, c'est donc la liste des plus petites distances de s à tous les sommets. Une liste P de sommets telle que $P[v]$ est le parent de v

$D \leftarrow [\infty, \infty, \dots, \infty]$; `/* distances initiales à` s `*/`
$D[s] \leftarrow 0$;
$P \leftarrow []$; `/* une liste ou un dictionnaire des "parents" */`
$Q \leftarrow V$; `/* file des sommets à visiter */`
tant que Q *est non vide* **faire**
  trouver $v \in Q$ tel que $D[v]$ est le minimum;
  $Q \leftarrow enlever(Q, v)$;
  pour $u \in \{liste\ des\ voisins\ de v\} \cap Q$ **faire**
    si $D[u] > D[v] + W(vu)$ **alors**
      $D[u] \leftarrow D[v] + W(vu)$;
      $P[u] \leftarrow v$
    fin
  fin
fin
retourner (D, P)

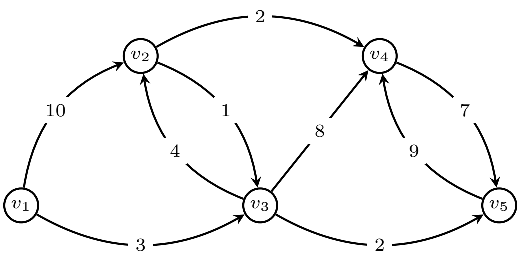
On proposera également une méthode qui renvoie le plus court chemin à un sommmet quelconque dans le bon ordre (et pas seulement la liste des parents), soit en utilisant une pile soit en utilisant la méthode *reverse* en python :
On obtient ainsi les déroulements suivant de l'algorithme pour un graphe orienté :

Algorithme 2 : Reconstruction du plus court chemin

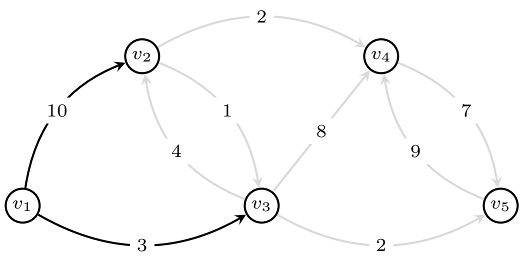
Données : Le sommet s d'où l'on a débuté le parcours, un sommet, d, destination, la liste des parents, P, obtenue grâce à l'algorithme de Dijkstra

Résultat : Le plus court chemin de s à d obtenu grâce à l'algorithme de Dijkstra (sous forme d'une liste, L, dans le bon ordre)

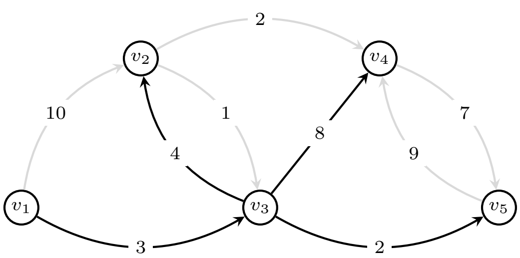
$L \leftarrow [d]$;
$v \leftarrow d$;
tant que $P[v]$ *est non vide* **faire**
  $L \leftarrow L + P[v]$;
  $v \leftarrow P[v]$;
fin
$L \leftarrow$ `"inverse de"` L;
retourner L



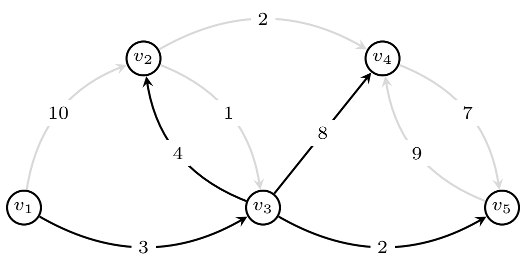
(a) Graphe orienté de départ



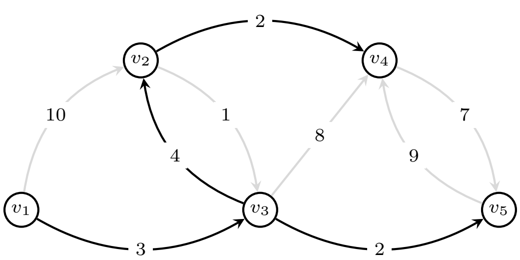
(b) Première itération



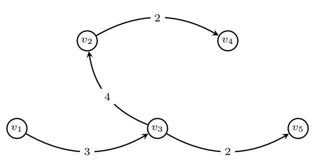
(c) Deuxième itération



(d) Troisième itération



(e) Quatrième itération



(f) Dernière itération

FIGURE 1 – Algorithme de Dijkstra pour un graphe orienté.

L'algorithme de Dijkstra ne prends pas en charge des pondérations négative. Pour cela on dispose de l'algorithme de Bellman-Ford ci-dessous. L'idée principale est de calculer la longueur des chemins de longeur k (contenants k arcs) de s aux autres sommets à partir des chemins de longeurs $k-1$. On peut remarquer que dans la boucle

---

Algorithme 3 : Algorithme de Bellman-Ford

Données : Un graphe orienté ou pas $G = (V, E)$ d'ordre $n > 0$, valué et sans boucles. Les pondérations peuvent être négatives. Un sommet s d'où l'on débute le parcours. Les sommets sont numérotés de 1 à $n = |V|$, i.e. $V = \{1, 2, \dots, n\}$.

Résultat : Une liste D de distances telle que $D[v]$ soit la distance de s à v en suivant le plus court chemin, c'est donc la liste des plus petites distances de s à tous les sommets. Une liste P de sommets telle que $P[v]$ est le parent de v. Si G a un cycle de poids négatif on retourne **False**

$D \leftarrow [\infty, \infty, \dots, \infty]$; `/* distances initiales à` s `*/`
$D[s] \leftarrow 0$;
$P \leftarrow []$; `/* une liste ou un dictionnaire des "parents" */`
pour $i \leftarrow 1, 2; \dots, n-1$ **faire**
    pour *chaque arc* $uv \in E$ **faire**
        si $D[v] > D[u] + W(uv)$ **alors**
            $D[v] \leftarrow D[u] + W(uv)$;
            $P[v] \leftarrow u$
        fin
    fin
fin
pour *chaque arc* $uv \in E$ **faire**
    si $D[v] > D[u] + W(uv)$ **alors**
        retourner False
    fin
fin
retourner (D, P)

"for" on met à jour, au plus $n-1$ fois la distance $D[v]$. Pour éviter les mises à jour inutiles on peut ajouter un test dans la boucle principale pour sortir si aucune mise à jour n'est effectuée lors de l'itération courante. Avec cette petite modification on obtient l'algorithme de Bellman-Ford suivant :

Algorithme 4 : Algorithme de Bellman-Ford avec test de redondance

Données : Un graphe orienté ou pas $G = (V, E)$ d'ordre $n > 0$, valué et sans boucles. Les pondérations peuvent être négatives. Un sommet s d'où l'on débute le parcours. Les sommets sont numérotés de 1 à $n = |V|$, i.e. $V = \{1, 2, \dots, n\}$.

Résultat : Une liste D de distances telle que $D[v]$ soit la distance de s à v en suivant le plus court chemin, c'est donc la liste des plus petites distances de s à tous les sommets. Une liste P de sommets telle que $P[v]$ est le parent de v. Si G a un cycle de poids négatif on retourne **False**

$D \leftarrow [\infty, \infty, \dots, \infty]$; `/* distances initiales à s */`
$D[s] \leftarrow 0$;
$P \leftarrow []$; `/* une liste ou un dictionnaire des "parents" */`
pour $i \leftarrow 1, 2; \dots, n-1$ **faire**
│ $misajour \leftarrow false$;
│ **pour** *chaque arc* $uv \in E$ **faire**
│ │ **si** $D[v] > D[u] + W(uv)$ **alors**
│ │ │ $D[v] \leftarrow D[u] + W(uv)$;
│ │ │ $P[v] \leftarrow u$;
│ │ │ $misajour \leftarrow true$
│ │ **fin**
│ **fin**
│ **si** $misajour = false$ **alors**
│ │ sortir de la boucle
│ **fin**
fin
pour *chaque arc* $uv \in E$ **faire**
│ **si** $D[v] > D[u] + W(uv)$ **alors**
│ │ **retourner** False
│ **fin**
fin
retourner (D, P)