TD : PROGRAMMATION DYNAMIQUE
== BELLMAN-FORD ==

Remarque : les rappels théoriques sont à la dernière page de ce sujet.

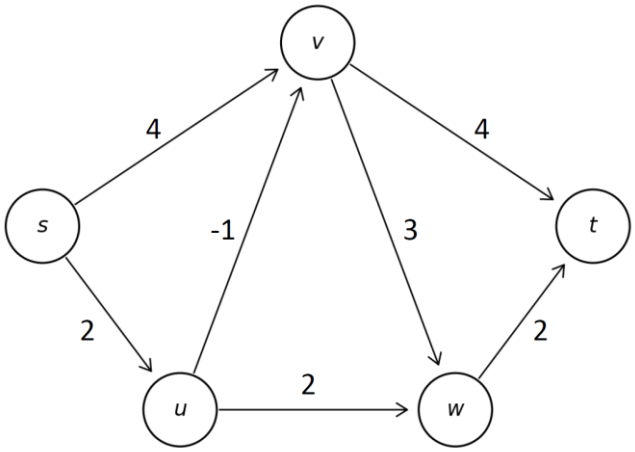
Le fichier source à utiliser pour ce TD est : « TD5 – BellmanFord.py »

Vous travaillez pour une entreprise de logistique qui doit optimiser ses livraisons. Le réseau de transport est modélisé par un graphe orienté où :

- Chaque sommet représente un entrepôt ou un point de livraison ;
- Chaque arête représente une route avec un coût de transport ;
- Certaines routes ont des coûts négatifs (subventions, remises partenaires).

L'objectif est de trouver le chemin de coût minimal depuis le dépôt central (source S) vers chaque destination, en utilisant l'algorithme de Bellman-Ford qui, contrairement à Dijkstra, fonctionne avec des poids négatifs.

Voici le graphe de transport que nous utiliserons :



Les données sont déjà définies dans le fichier source :

```
# Graphe représenté par un dictionnaire d'adjacence
# graphe[u] = [(v1, poids1), (v2, poids2), ...]
graphe = {
    'S': [('U', 2), ('V', 4)],
    'U': [('V', -1), ('W', 2)],
    'V': [('W', 3), ('T', 4)],
    'W': [('T', 2)],
    'T': []
}
source = 'S'
L = {}  # Dictionnaire de mémoïsation
```

I) APPROCHE BOTTOM-UP (TABULATION)

Dans cette partie, vous allez implémenter l'approche bottom-up qui remplit une table de tous les sous-problèmes, des plus petits aux plus grands. On utilisera un dictionnaire L pour stocker les valeurs L[(i, v)] qui représentent la distance minimale de la source S vers le sommet v en utilisant au plus i arêtes.

1. Écrire une fonction `initialiser_cas_de_base(G, S, L)` qui prend en paramètres le graphe G, le nom du sommet source S et le dictionnaire L et l'initialise puis retourne le dictionnaire L avec les cas de base (i = 0).

2. Écrire une fonction `obtenir_aretes_entrantes(G, v)` qui retourne la liste des arêtes entrantes dans le sommet v sous la forme [(u, poids), ...] où u est un prédécesseur de v.

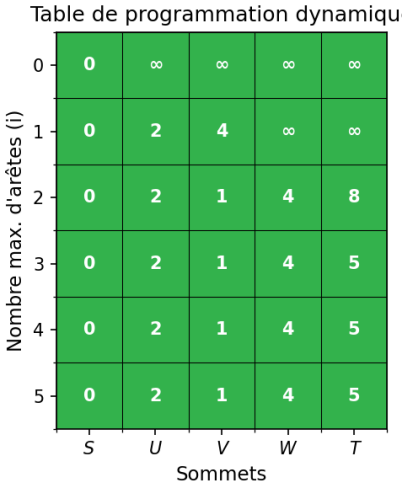
   Vérifier (l'ordre peut varier selon votre implémentation) :

```
>>> obtenir_aretes_entrantes(graphe, 'V')
[('S', 4), ('U', -1)]
>>> obtenir_aretes_entrantes(graphe, 'T')
[('V', 4), ('W', 2)]
>>> obtenir_aretes_entrantes(graphe, 'S')
[]
```

3. Écrire une fonction `remplir_table(G, L)` qui remplit entièrement la table L en utilisant l'équation de récurrence (voir rappels à la fin du sujet). L'ordre de parcours est : pour i allant de 1 à n (nombre de sommets), et pour chaque i, parcourir tous les sommets v.

Vérifier :

```
>>> L = {}
>>> L = initialiser_cas_de_base(graphe,source,L)
>>> L = remplir_table(graphe,L)
>>> AfficheTable(L,graphe)
```



4. Au regard de la table que vous obtenez, que pouvez-vous dire sur la présence ou l'absence d'un cycle négatif dans le graphe ?

5. Écrire une fonction `bellman_ford_bottomup(G, S)` où S est la source et qui utilise les fonctions précédentes pour calculer et retourner le dictionnaire L et un booléen indiquant s'il y a un cycle négatif.

Vérifier :

```
>>> bellman_ford_bottomup(graphe, source)
({(0, 'S'): 0, (0, 'U'): inf, (0, 'V'): inf, (0,
'W'): inf, (0, 'T'): inf, (1, 'S'): 0, (1, 'U'): 2,
(1, 'V'): 4, (1, 'W'): inf, (1, 'T'): inf, (2, 'S'):
0, (2, 'U'): 2, (2, 'V'): 1, (2, 'W'): 4, (2, 'T'):
8, (3, 'S'): 0, (3, 'U'): 2, (3, 'V'): 1, (3, 'W'):
4, (3, 'T'): 5, (4, 'S'): 0, (4, 'U'): 2, (4, 'V'):
1, (4, 'W'): 4, (4, 'T'): 5, (5, 'S'): 0, (5, 'U'):
2, (5, 'V'): 1, (5, 'W'): 4, (5, 'T'): 5}, False)
```

6. Modifier votre graphe afin de créer un cycle négatif atteignable depuis la source et vérifier qu'il est bien détectable :

Attention : si un cycle négatif est détecté, les valeurs retournées ne correspondent pas à des distances minimales (problème non borné) ; elles reflètent seulement l'état de la table après un nombre fini d'itérations.

```
# Graphe avec cycle négatif
graphe_neg = {
    'S': [('U', 2), ('V', 4)],
    'U': [('V', -1), ('W', 2)],
    'V': [('W', -3), ('T', -4)],
    'W': [('T', -2)],
    'T': [('V',-1)]
}
source = 'S'
L, cycle_negatif = bellman_ford_bottomup(graphe_neg, source)
AfficheTable(L,graphe_neg)
print(cycle_negatif) # <= Affiche True
```

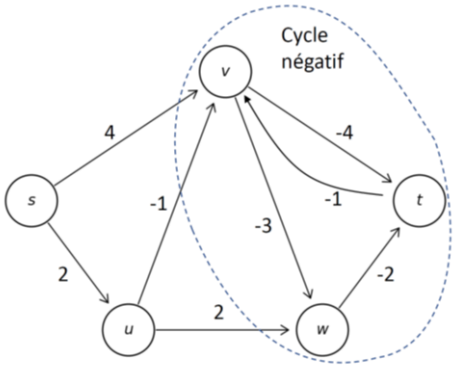


Table de programmation dynamique

Nombre max. d'arêtes (i)	S	U	V	W	T
0	0	∞	∞	∞	∞
1	0	2	4	∞	∞
2	0	2	1	1	0
3	0	2	-1	-2	-3
4	0	2	-4	-4	-5
5	0	2	-6	-7	-8

Sommets

7. Écrire une fonction `extraire_distances(L, graphe)` qui extrait les distances finales depuis le dictionnaire L et retourne un dictionnaire {sommet: distance}.

Vérifier :
```
>>> L, cycle_negatif = bellman_ford_bottomup(graphe, source)
>>> extraire_distances(L,graphe)
{'S': 0, 'U': 2, 'V': 1, 'W': 4, 'T': 5}
```

8. Combien de sous-problèmes sont calculés dans l'approche bottom-up pour un graphe à n sommets et m arêtes ? Quelle est la complexité temporelle et spatiale de cet algorithme ?

II) APPROCHE TOP-DOWN AVEC MÉMOÏSATION

Dans cette partie, vous allez implémenter l'algorithme récursif avec mémoïsation. L'idée est de partir du problème principal L[(n-1, v)] (si on ne cherche pas les cycles négatifs) et de le décomposer en sous-problèmes, en mémorisant les résultats pour éviter les calculs redondants.

On utilisera un dictionnaire défini dans le programme principal pour la mémoïsation : L = {}

1. Écrire une fonction récursive `rec_bellman_ford1(G, S, dest)` qui implémente la récurrence de Bellman-Ford avec mémoïsation. Cette fonction retourne la distance minimale de la source vers l'unique destination `dest` et ne gère pas la détection de cycle négatif.

   Vous pouvez vous aider du squelette ci-dessous :

```
def rec_bellman_ford1(G,S,dest):
    L = {}
    n = len(G)

    def f_rec(i,v):

        # Utilise la mémoisation
        if ...................:
            return .........

        # Cas de base
        if i == 0:
            if v == S:
                L[(i,v)] = .........
            else:
                L[(i,v)] = .........
            return L[(i,v)]

        # Cas n°1 : hériter de la valeur précédente
        val_opt = .........

        # Cas n°2 : Tester tous les prédécesseurs
        for (u,poids) in obtenir_aretes_entrantes(G,v):
            candidat = ..................
            val_opt = min(.........,.........)

        # Mémoisation
        L[(i,v)] = .........

        return .........

    distance = .........
    return distance, L
```

Table de programmation dynamique

Nombre max. d'arêtes (i)	S	U	V	W	T
0	0	∞	∞	∞	∞
1	0	2	4	∞	∞
2	0	2	1	4	8
3			1	4	5
4					5

Sommets

Vérifier :

```
>>> distance, L = rec_bellman_ford1(graphe,source,'T')
>>> print(distance)
5
>>> AfficheTable(L,graphe)
```

2. Écrire une fonction récursive `rec_bellman_ford2(G, S)` qui implémente la récurrence de Bellman-Ford avec mémoïsation. Cette fonction retourne maintenant toutes les distances minimales vers les différentes destinations sous forme d'un dictionnaire ainsi qu'un booléen précisant s'il existe un cycle négatif ou non dans le graphe.

```
Vérifier :  >>> rec_bellman_ford2(graphe,source)
            ({'S': 0, 'U': 2, 'V': 1, 'W': 4, 'T': 5}, False)
            >>> rec_bellman_ford2(graphe_neg,source)
            ({'S': 0, 'U': 2, 'V': -4, 'W': -4, 'T': -5}, True)
```

3. Quelle est la complexité temporelle de l'algorithme top-down avec mémoïsation dans le pire cas ? Quelle est la complexité spatiale (dictionnaire + pile d'appels) ?

III) RECONSTRUCTION DE LA SOLUTION

Maintenant que nous savons calculer les distances minimales, nous devons reconstruire le chemin optimal.

Pour cela, on « remonte » dans la table L depuis L[(n-1, destination)] jusqu'à L[(0, source)] pour déterminer, à chaque étape, quelle décision a été prise.

1. Modifier votre fonction `rec_bellman_ford2()` pour qu'elle renvoie, en plus du dictionnaire des distances et du booléen indiquant la présence ou l'absence d'un cycle négatif, la table L.

2. Écrire une fonction `determiner_choix(G, L, i, v)` qui détermine le choix optimal pour arriver à L[(i, v)]. Cette fonction retourne un tuple (choix, new_i, new_v) :
   - `choix` est une chaîne décrivant le choix : "HERITER" ou "ARETE u->v" ;
   - `new_i` et `new_v` sont les nouveaux indices pour continuer la reconstruction.

```
Vérifier :  >>> distances, cycle_negatif, L =
            rec_bellman_ford2(graphe,source)
            >>> determiner_choix(graphe,L,4,'T')
            ('HERITER', 3, 'T')
            >>> determiner_choix(graphe,L,3,'T')
            ('ARETE V->T', 2, 'V')
            >>> determiner_choix(graphe,L,2,'V')
            ('ARETE U->V', 1, 'U')
```

3. Écrire une fonction `reconstruire_chemin(G, L, source, destination)` qui retourne la liste des sommets du chemin optimal depuis la source jusqu'à la destination.

```
Vérifier :  >>> distances, cycle_negatif, L =
            rec_bellman_ford2(graphe,source)
            >>> reconstruire_chemin(graphe,L,source,'T')
            ['S', 'U', 'V', 'T']
            >>> reconstruire_chemin(graphe,L,source,'W')
            ['S', 'U', 'W']
```

RAPPELS THÉORIQUES

Formulation du problème

Soit un graphe orienté G = (V, E) avec n sommets et m arêtes, où chaque arête e possède une longueur réelle ℓ_e (possiblement négative). Étant donnée une source s ∈ V, on cherche à calculer pour chaque sommet v la distance minimale dist(s, v) depuis s.

Sous-problèmes et notation

On note $L_{i,v}$ la longueur minimale d'un chemin de s vers v utilisant au plus i arêtes. Si aucun tel chemin n'existe, $L_{i,v} = +\infty$.

Relation de récurrence

Pour tout i ≥ 1 et tout v ∈ V :

$$L_{i,v} = min \begin{cases} L_{i-1,v} & (cas\ n°1) \\ min_{(w,v)\in E}\{ L_{i-1,w} + \ell_{w,v}\} & (cas\ n°\ 2) \end{cases}$$

Cas n°1 : $L_{i-1,v}$ — le chemin optimal utilise (i-1) arêtes ou moins
Cas n°2 : $L_{i-1,w} + \ell_{w,v}$ — le chemin optimal utilise exactement i arêtes, la dernière étant (w, v)

Cas de base

Les cas de base sont (s est le sommet de départ) :

- $L_{0,s} = 0$ (il existe un chemin de s → s utilisant 0 arêtes – le chemin vide de longueur 0)
- $L_{0,v} = +\infty$ pour tout $v \neq s \in V$ (avec 0 arêtes, on ne peut atteindre aucun sommet autre que s)

Critère d'arrêt et détection de cycle négatif

Sans cycle négatif, les valeurs se stabilisent au plus tard à i = n-1 (un plus court chemin simple a au plus n-1 arêtes). Si $L_{n,v} < L_{n-1,v}$ pour au moins un sommet v, alors le graphe contient un cycle négatif atteignable depuis la source.

Algorithme de reconstruction

Une fois la table des valeurs optimales remplie, on reconstruit le chemin en « remontant » depuis le problème (n-1, destination) jusqu'à (0, source).

À chaque position (i, v), on détermine quelle décision a permis d'obtenir $L_{i,v}$:

- Si $L_{i,v} == L_{i-1,v}$ → Héritage (pas de nouvelle arête)
- Sinon, chercher w tel que $L_{i,v} == L_{i-1,w} + \ell_{w,v}$ → l'arête (w, v) fait partie du chemin optimal.