TD : PROGRAMMATION DYNAMIQUE
== FLOYD-WARSHALL ==

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

Le fichier source à utiliser pour ce TD est : « TD6 – FloydWarshall.py »

Vous travaillez pour une entreprise de transport ferroviaire qui doit optimiser les trajets entre différentes gares. Le réseau est modélisé par un graphe orienté où :

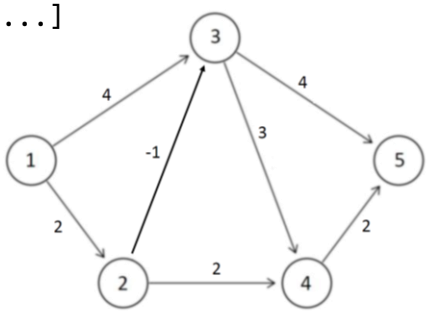
- Chaque sommet représente une gare ;
- Chaque arête représente une liaison directe avec un temps de trajet ;
- Certaines liaisons ont des temps négatifs (correspondances optimisées, bonus fidélité).

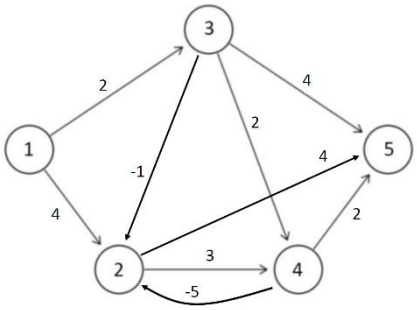
L'objectif est de calculer les plus courts chemins entre toutes les paires de gares, en utilisant l'algorithme de Floyd-Warshall.

On utilisera un dictionnaire L pour stocker les valeurs de programmation dynamique sous la forme L[(k, v, w)]. Le paramètre k désigne le plus grand sommet autorisé comme sommet intermédiaire.

Les données sont déjà définies dans le fichier source :

```
# Graphe représenté par un dictionnaire d'adjacence
# graphe[v] = [(w1, poids1), (w2, poids2), ...]
graphe = {
    1: [(2, 2), (3, 4)],
    2: [(3, -1), (4, 2)],
    3: [(4, 3), (5, 4)],
    4: [(5, 2)],
    5: []
}

# Variante avec cycle négatif (pour tests)
graphe_neg = {
    1: [(2, 4), (3, 2)],
    2: [(4, 3), (5, 4)],
    3: [(2, -1), (4, 2), (5,4)],
    4: [(2, -5), (5, 2)],
    5: []
}
```





I) APPROCHE BOTTOM-UP

Dans cette partie, vous implémentez l'approche bottom-up qui calcule systématiquement tous les sous-problèmes (k, v, w) dans l'ordre k = 0, 1, 2, ..., n.

1. Ecrire une fonction `poids_arete(G, u, v)`.
   Cette fonction retourne le poids de l'arête u $\rightarrow$ v si elle existe, sinon np.inf.

   Tester :
   ```
   >>> poids_arete(graphe,1,2)
   2
   >>> poids_arete(graphe,1,3)
   4
   >>> poids_arete(graphe,1,4)
   inf
   ```

2. Écrire une fonction `initialiser_L0(G)`.
   Cette fonction initialise et retourne le dictionnaire L contenant tous les cas de base k = 0 (voir les rappels théoriques à la fin du sujet).

   Tester :
   ```
   >>> L = initialiser_L0(graphe)
   >>> L[(0,1,1)]
   0
   >>> L[(0,1,3)]
   4
   >>> L[(0,1,5)]
   Inf
   ```

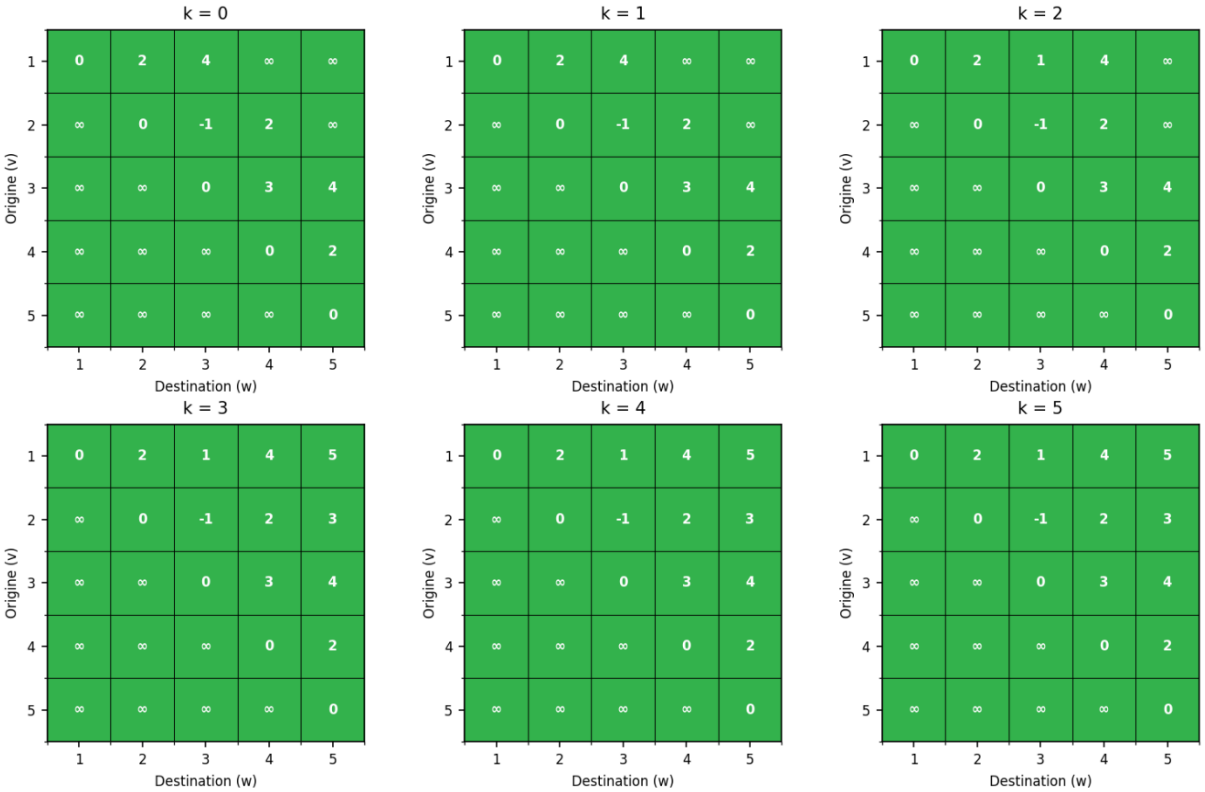
3. Écrire une fonction `floyd_warshall_bottomup(G)`.
   Cette fonction calcule toutes les valeurs L[(k, v, w)] pour v, w $\in$ V à partir des valeurs au niveau k = 1 , en appliquant la récurrence et détecte également un cycle négatif pendant les itérations.
   Elle retourne `(dist, False)` où `dist[(v, w)] = L[(n, v, w)]` si aucun cycle négatif n'est détecté, et retourne `(None, True)` dans le cas contraire.

   Tester :
   ```
   >>> L = initialiser_L0(graphe)
   >>> dist, cycle_negatif = floyd_warshall_bottomup(graphe,L)
   >>> dist[(1,5)]
   5
   >>> cycle_negatif
   False
   >>> L = initialiser_L0(graphe_neg)
   >>> dist, cycle_negatif = floyd_warshall_bottomup(graphe_neg,L)
   >>> dist
   >>> cycle_negatif
   True
   ```

4. Affichage (comparaison) : le fichier source fournit une fonction AfficheTable(L,G) qui affiche les tranches pour chaque valeur de k. En bottom-up, toutes les cases existent à chaque k.

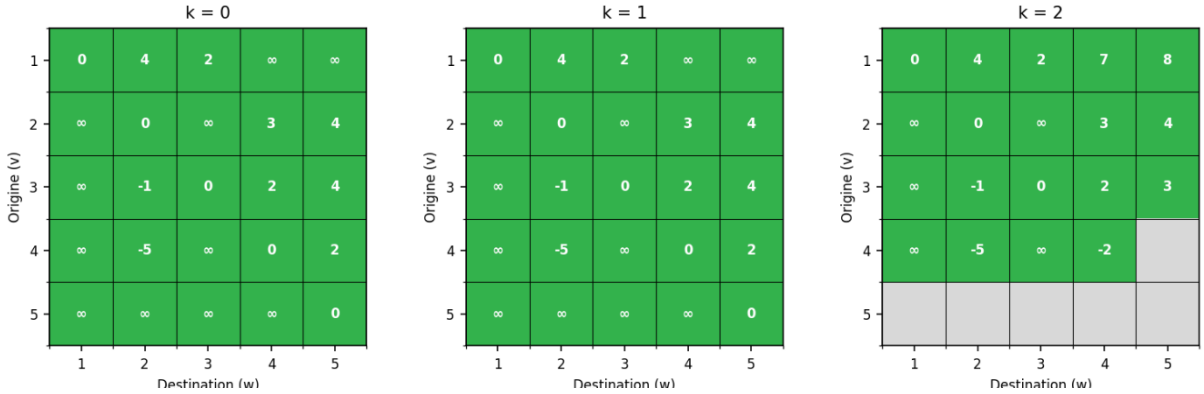
Graphe sans cycle négatif :

Table de programmation dynamique Floyd-Warshall



Graphe avec cycle négatif (détecté à k = 2) :

Table de programmation dynamique Floyd-Warshall



5. Questions théoriques (complexité).
   a) Combien de sous-problèmes (k,v,w) sont calculés pour un graphe à n sommets ?
   b) En déduire la complexité temporelle et la complexité spatiale.
   c) Le cours mentionne une version bottom-up optimisée en mémoire à $O(n^2)$. Donner l'équation de récurrence à utiliser dans ce cas.

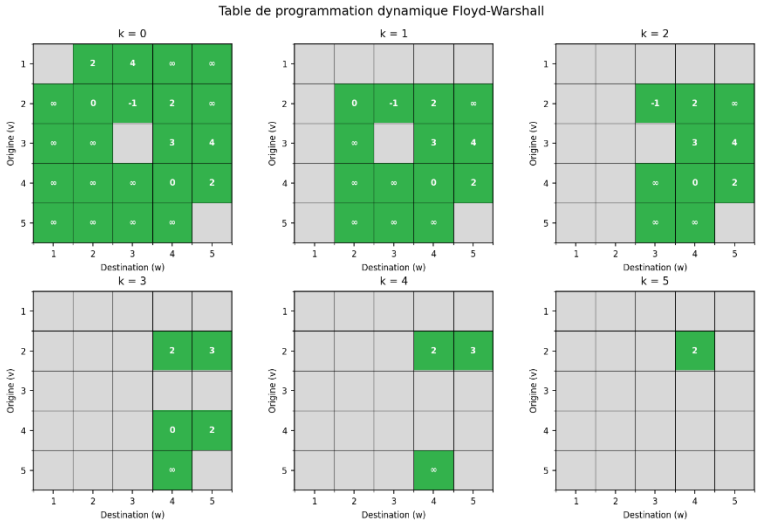
II) APPROCHE TOP-DOWN AVEC MÉMOÏSATION

Dans cette partie, vous implémentez la version récursive avec mémoïsation (top-down). L'idée est de calculer un sous-problème seulement lorsqu'il est demandé, et de mémoriser le résultat dans un dictionnaire.

1. Écrire une fonction `floyd_warshall_topdown_paire(G, v, w)`.
   Cette fonction calcule uniquement la distance optimale pour une paire (v, w) via une fonction récursive `f_rec(k, a, b)` et un dictionnaire L de mémoïsation.
   On inclut une détection précoce de cycle négatif : lever une exception « Cycle Négatif Détecté ».

```
def floyd_warshall_topdown_paire(G,v,w):
    L = {}

    def f_rec(k,a,b):
        # Mémoïsation
        ..................

        # Cas de base k == 0
        ..................

        # Récurrence
        ..................

        # Détection cycle_negatif
        if ............................... :
            raise ValueError("Cycle négatif détecté")

        return L[(k,a,b)]

    n = len(G)
    return L, f_rec(n,v,w)
```

```
Tester :
>>> L, dist = floyd_warshall_topdown_paire(graphe_neg,2,4)
TypeError: Cycle négatif détecté
>>> L, dist = floyd_warshall_topdown_paire(graphe,2,4)
>>> dist
>>> 5
>>> AfficherTable(L,graphe)
```

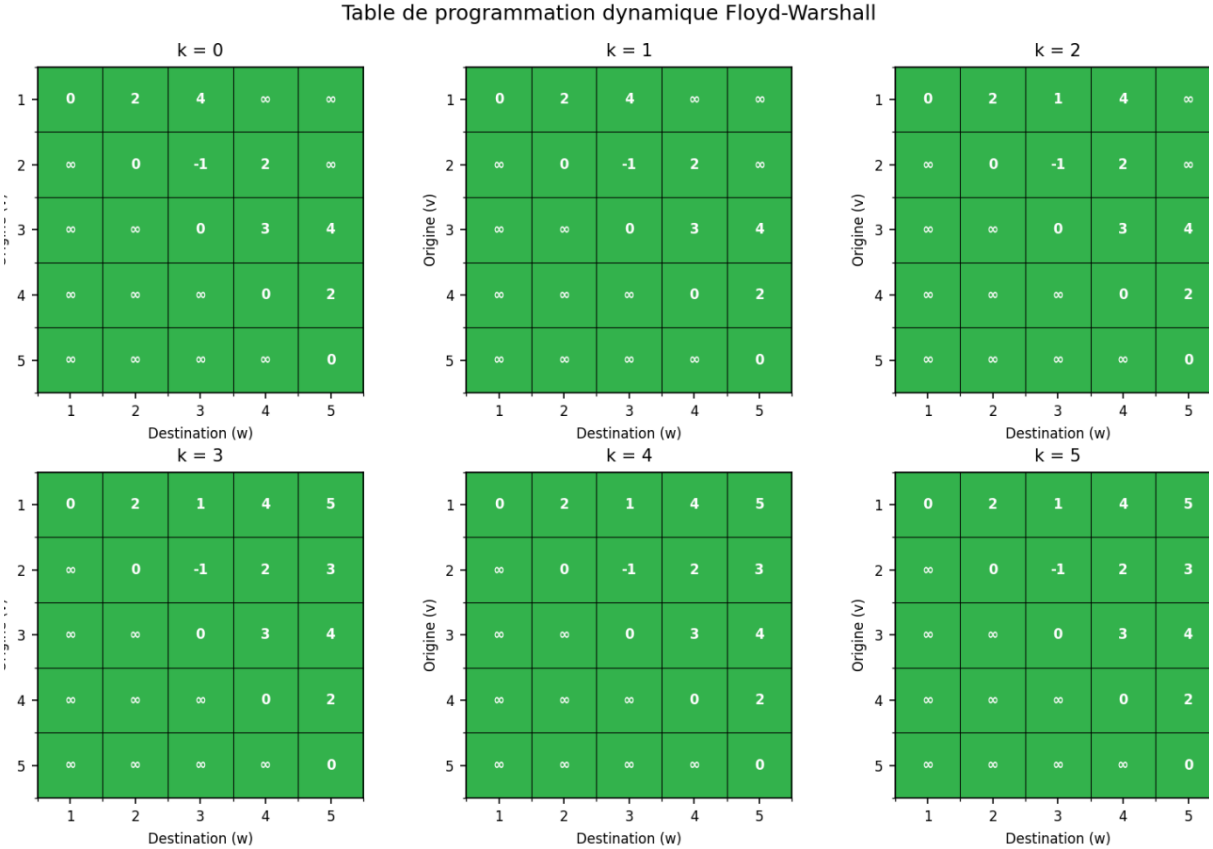


2. Écrire une fonction floyd_warshall_topdown_toutes_paires(G).
   Cette fonction calcule toutes les distances en réutilisant le même dictionnaire L pour tous les appels récursifs, puis retourne (dist, cycle_negatif, L). Si `cycle_negatif` est `True`, on retournera None pour les dictionnaires `dist` et L.

```
def floyd_warshall_topdown_toutes_paires(G):
    L = {}
    def f_rec(k,a,b):
        ......................
        ......................
        return L[(k,a,b)]

    n = len(G)
    dist = {}
    try:
        # Calcul des distances par récurrence top-down
        ......................
        ......................
        return (dist,False,L)
    except:
        return (None,True,None)
```

Tester :

```
>>> floyd_warshall_topdown_toutes_paires(graphe_neg)
>>> (None, True, None)
>>> dist,cycle_negatif,L = floyd_warshall_topdown_toutes_paires(graphe)
>>> cycle_negatif
False
>>> dist[(1,5)]
5
>>> AfficheTable(L,graphe)
```



3. Questions théoriques.
   a) Quelle est la complexité temporelle du top-down dans le pire des cas ?
   b) Pourquoi, lorsqu'on ne calcule qu'une seule paire, le nombre d'états mémorisés peut être strictement inférieur à $O(n^3)$?
   c) Le cours précise qu'on peut détecter un cycle négatif « au fur et à mesure » en top-down mais qu'on ne peut pas conclure « pas de cycle négatif » tant qu'on n'a pas forcé le calcul des diagonales pertinentes. Expliquer.

III) RECONSTRUCTION D'UN CHEMIN OPTIMAL

Une fois les valeurs L[(k,v,w)] calculées (par bottom-up ou top-down complet), on veut reconstruire un plus court chemin de v vers w. On suit le principe du cours : à partir de (n,v,w), on teste si la valeur « hérite » de k-1 ou si k est un sommet intermédiaire.

1. Écrire une fonction `decision_reconstruction(L, k, v, w)`.
   Cette fonction retourne une information sur la décision au niveau (k, v, w) :
   - soit « HERITER » si L[(k, v, w)] == L[(k-1, v, w)] ;
   - soit « DECOMPOSER » si L[(k, v, w)] == L[(k-1, v, k)] + L[(k-1, k, w)].

2. Écrire une fonction `rec_chemin(L, k, v, w)`.
   Fonction récursive de reconstruction :
   - Si L[(k, v, w)] == np.inf, retourner []
   - Si k == 0, renvoyer le chemin direct [v, w]
   - Si « HERITER », on descend à k-1
   - Si « DECOMPOSER », on reconstruit v $\rightarrow$ k puis k $\rightarrow$ w récursivement et on les concatène en évitant de dupliquer k.

```
Tester :    >>> rec_chemin(L,0,1,3)       # Chemin 1 → 3
            [1, 3]
            >>> rec_chemin(L,1,1,3)
            [1, 3]
            >>> rec_chemin(L,2,1,3)
            [1, 2, 3]
            >>> rec_chemin(L,3,1,3)
            [1, 2, 3]

            >>> rec_chemin(L,0,1,4)       # Chemin 1 → 4
            []
            >>> rec_chemin(L,1,1,4)
            []
            >>> rec_chemin(L,2,1,4)
            [1, 2, 4]
            >>> rec_chemin(L,3,1,4)
            [1, 2, 4]
            >>> rec_chemin(L,4,1,4)
            [1, 2, 4]
```

3. Questions théoriques.
   a) Pourquoi la reconstruction d'un chemin est en O(n) dans le pire cas ?
   b) Pourquoi reconstruire tous les chemins (toutes paires) peut coûter $O(n^3)$ au total ?

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). On cherche à calculer pour chaque paire de sommets (v, w) la distance minimale dist(v, w).

Sous-problèmes et notation

On note $L_{k,v,w}$ la longueur minimale d'un chemin sans cycle de v vers w utilisant uniquement les sommets {1, 2, ..., k} comme sommets intermédiaires. Si aucun tel chemin n'existe, on pose $L_{k,v,w} = +\infty$.

Relation de récurrence

Pour tout k ∈ {1, 2, …, n} et v,w ∈ V :

$$L_{k,v,w} = min \begin{cases} L_{k-1,v,w} & (cas\ n°1) \\ L_{k-1,v,k} + L_{k-1,k,w} & (cas\ n°\ 2) \end{cases}$$

Cas de base

Cas de base (k = 0) :

- $L_{0,v,v} = 0$ (chemin vide) ;
- $L_{0,v,w} = \ell_{v,w}$ si l'arête (v, w) existe ;
- $L_{0,v,w} = +\infty$ sinon

Détection de cycle négatif

Le graphe contient un cycle négatif si et seulement si, à la fin de l'algorithme, on a $L_{n,v,v} < 0$ pour un certain sommet v ∈ V.

Algorithme de reconstruction

Une fois la table des valeurs optimales remplie, on reconstruit le chemin en « remontant » depuis le problème (n, v, w) jusqu'à k = 0.

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

- Si $L_{k,v,w} == L_{k-1,v,w} \rightarrow$ Héritage (k non utilisé)
- Sinon, le sommet k fait partie du chemin optimal. On reconstruit v $\rightarrow$ k puis k $\rightarrow$ w récursivement et on concatène ces deux chemins.