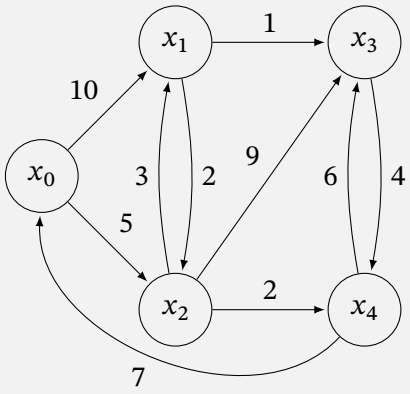
TP 18 : Algorithmes de plus court chemin — Floyd-Warshall

L'objectif des deux prochains TPs est d'implémenter des algorithmes de plus court chemin. La semaine prochaine, on parlera et implémentera efficacement l'algorithme de Dijkstra. Dans les deux TP, on se limitera aux cas des graphes orientés, et on considèrera des poids flottants.
Le fichier `tp18_exemple.ml` contient un exemple de graphe pondéré sur lequel on va travailler, identique à celui présenté en cours. Ce graphe est le suivant :

Exemple 1



Exercice 1 – Propriétés de la distance restreinte

Petit rappel :
Le problème que l'on cherche à résoudre ici est le problème du plus court chemin depuis tout sommet vers tout sommet : on prend en entrée un graphe orienté $\mathcal{G} = (S, A)$ et l'on cherche à déterminer $d(x, y)$ pour tout $x, y \in S$. On suppose que ce graphe ne contient pas de cycle de poids négatif.
L'algorithme de Floyd-Warshall est un algorithme de programmation dynamique. La décomposition en sous-structures optimales vient simplement de l'observation suivante : si c est un plus court chemin entre x et y, et que z est un sommet à l'intérieur de ce chemin, alors on peut découper c en un plus court chemin de x à z et de z à y.
Le problème de cette propriété de sous-structure optimale est qu'elle ne se traduit pas directement en une relation de récurrence correcte permettant d'implémenter une fonction récursive : par exemple, pour calculer la distance de x_0 à x_2, on va avoir besoin de la distance de x_0 à x_1, qui lui-même va avoir besoin de la distance de x_0 à x_2 !
Dans une optique de programmation dynamique *ascendante*, on va choisir un ordre dans lequel on cherche à calculer les plus courtes distances, à l'aide d'une astuce mathématique : on va considérer la notion de *distance restreinte*.

Définition 2 – Distance restreinte

On note $d_k(x_i, x_j)$ la distance de x_i à x_j dans lequel on ne considère que les chemins passant uniquement par les sommets $x_0 \dots x_{k-1}$ (sauf les extrémités), $+\infty$ si un tel chemin n'existe pas.
Autrement dit, c'est la distance de x_i à x_j dans le graphe induit par les sommets $\{x_0, \dots, x_{k-1}\} \cup \{x_i, x_j\}$.

▶ **Question 1** Déterminer $d_0(x_0, x_1)$, $d_4(x_0, x_3)$ et $d_5(x_0, x_3)$.

On rappelle que l'on définit la fonction de pondération $p : A \to \mathbb{R}$, et que l'on étend cette fonction à S^2 en fixant $p(a_x) = +\infty$ si $a_x \notin A$.

▶ **Question 2** Déterminer $d_k(x, y)$ pour tout $x, y \in S$.

▶ **Question 3** Soit $k \in \mathbb{N}$. Déterminer $d_{k+1}(\cdot, \cdot)$ en fonction de valeurs de $d_k(\cdot, \cdot)$.

▶ **Question 4** Justifier que $d(x, y) = d_n(x, y)$, avec $n = |S|$.

Et voilà ! Il suffit maintenant de déterminer les valeurs successives, pour $k \in \mathbb{N}$ de $d_k(x, y)$ et de renvoyer la matrice de ces valeurs pour $k = n$.

Exercice 2 – Première implémentation

On déduit des questions précédentes l'algorithme suivant, qui calcule et stocke dans des matrices les valeurs successives de $d_k(x, y)$. On considère que $\mathcal{G}$ est représenté par une matrice d'adjacence, où les poids sont des flottants et l'absence d'arc est représentée par le poids `infinity`.

Pseudo-code

```
1 : Fonction FloydWarshall(G)
2 :     M_0 ← CopieMatrice(G)
3 :     Pour k = 0 à n − 1 , faire
4 :         M_{k+1} ← CopieMatrice(M_k)
5 :         Pour i = 0 à n − 1 , faire
6 :             Pour j = 0 à n − 1 , faire
7 :                 M_{k+1}[i, j] ← min(M_k[i, j], M_k[i, k] + M_k[k, j])
8 :     retourne M_n
```

▶ **Question 1** Implémenter le pseudo-code précédent en une fonction `floyd_warshall : float array array -> float array array`.
On pourra utiliser les fonctions déjà présentes dans le fichier `tp18_exemple.ml` : `creer_matrices : int -> float array array array` tel que l'appel `creer_matrices n` renvoie un tableau de `n + 1` matrices de taille $n \times n$ remplies de `infinity`, et `copier_matrice_dans : float array array -> float array array -> unit` tel que l'appel `copier_matrice_dans m m'` copie toutes les valeurs de `m` dans `m'`.

▶ **Question 2** Déterminer sa complexité temporelle et spatiale.

La première étape pour améliorer ces complexités est qu'il suffit de se souvenir de la dernière matrice M_k pour déterminer la matrice M_{k+1}. Ainsi, à chaque instant on peut stocker seulement deux matrices : celle que l'on est en train de calculer et la précédente. En termes de pseudocode, cela donne :

Pseudo-code

```
1 : Fonction FloydWarshall2(G)
2 :     Précédent ← CopieMatrice(G)
3 :     Pour k = 0 à n − 1 , faire
4 :         Nouveau ← CopieMatrice(Précédent)
5 :         Pour i = 0 à n − 1 , faire
6 :             Pour j = 0 à n − 1 , faire
7 :                 Nouveau[i, j] ← min(Précédent[i, j], Précédent[i, k] + Précédent[k, j])
8 :         Précédent ← Nouveau
9 :     retourne Précédent
```

▶ **Question 3** Implémenter cet algorithme en une fonction `floyd_warshall2 : float array array -> float array array`.

▶ **Question 4** Déterminer sa complexité temporelle et spatiale.

▶ **Question 5** Il est encore possible d'améliorer sa complexité spatiale [a]. Comment ? L'implémenter en une fonction `floyd_warshall3`

a. Sans compter la mémoire utilisée par la sortie de l'algorithme.

Exercice 3 – Reconstruction de plus court chemin

En disposant seulement de la matrice FLOYDWARSHALL($\mathcal{G}$), il est difficile de reconstruire efficacement un plus court chemin entre deux sommets du graphe. On doit également noter, en même temps que la matrice des distances, le sommet utilisé pour atteindre cette distance : on doit donc écrire une fonction `floyd_warshall_avec_pred : float array array -> (float array array * int option array array)`, tel qu'un appel `floyd_warshall_avec_pred g` renvoie le couple :

- `m` la matrice des distances identique à celle renvoyée par `floyd_warshall`,
- `prochain` tel que `prochain.(i).(j) = None` si x_j n'est pas accessible depuis x_i, `prochain.(i).(j) = Some k` sinon avec x_k le sommet successeur de x_i dans un plus court chemin de x_i à x_j.

▶ **Question 1** Implémenter cette fonction `floyd_warshall_avec_pred`. *Sa complexité sera identique à la fonction* `floyd_warshall3`.

La connaissance de cette seconde matrice `prochain` permet donc de déterminer efficacement un plus court chemin entre deux sommets.

▶ **Question 2** Écrire une fonction `reconstruire : int option array array -> int -> int -> int list` tel que `reconstruire prochain x y` renvoie un plus court chemin entre `x` et `y`.

▶ **Question 3** Déterminer que la complexité de `reconstruire` linéaire en la taille du plus court chemin (sinon, recommencer).