ALGORITHMES DES GRAPHES

À la fin de ce chapitre, je sais :

- ☞ parcourir un graphe en largeur et en profondeur
- ☞ utiliser une file ou un pile pour parcourir un graphe
- ☞ énoncer le principe de l'algorithme de Dijkstra (plus court chemin)

A Parcours d'un graphe

Le parcours d'un graphe est une opération fondamentale et utilisée par de nombreux algorithmes, notamment Dijkstra et A*. On peut facilement mémoriser les différentes stratégies en observant les types d'ensemble qui sont utilisés pour stocker les sommets à parcourir au cours de l'algorithme :

1. Le parcours en **largeur** passe par tous les voisins d'un sommet avant de parcourir les descendants de ces voisins. Les sommets passent dans une **file** d'attente, c'est-à-dire structure de données de type First In First Out.
2. Le parcours en **profondeur** passe par tous les descendants d'un voisin d'un sommet avant de parcourir tous les autres voisins de ce sommet. L'implémentation la plus simple et la plus commune est récursive. Mais on peut aussi l'implémenter de manière itérative en utilisant une *pile* de sommets, c'est-à-dire une structure de données de type Last In First Out.

B Parcours en largeur

■ **Définition 1 — Parcours en largeur.** Parcourir en largeur un graphe signifie qu'on cherche à visiter tous les voisins d'un sommet avant de parcourir les autres sommets du graphe.

Vocabulary 1 — Breadth First Search ⇜⇝ Parcours en largeur

Le parcours en largeur d'un graphe (cf. algorithme 1) est un algorithme à la base de nombreux développements comme l'algorithme de Dijkstra et de Prim (cf. algorithmes 7 et **??**). Il utilise une file d'attente [1] afin de gérer la découverte des voisins dans l'ordre de la largeur du

1. structure de données de type First In Fisrt Out

graphe.

R Pour matérialiser le parcours en largeur dans l'algorithme 1, on opère en **marquant** les sommets du graphe à visiter et les sommets du graphes déjà visités. Lorsqu'un sommet est découvert, il intègre l'ensemble des éléments à visiter, c'est-à-dire la file d'attente F. Lorsque le sommet a été traité, il quitte la file. Il est donc également nécessaire de garder la trace du passage sur un sommet afin de ne pas traiter plusieurs fois un même sommet : si un sommet a été visité alors il intègre l'ensemble des éléments visités V.

Algorithme 1 Parcours en largeur d'un graphe

```
1 : Fonction PARCOURS_EN_LARGEUR(G, s)                    ▷ s est un sommet de G
2 :     F ← une file d'attente vide                        ▷ F comme file
3 :     E ← ∅                                              ▷ E ensemble des sommets explorés
4 :     P ← une liste vide                                 ▷ P comme parcours
5 :     ENFILER(F,s)
6 :     AJOUTER(E,s)
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         AJOUTER(P, v)                                  ▷ ou bien traiter le sommet en place
10 :        pour chaque voisin x de v dans G répéter
11 :            si x ∉ V alors                             ▷ x n'a pas encore été découvert
12 :                AJOUTER(E, x)
13 :                ENFILER(F, x)
14 :    renvoyer P          ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

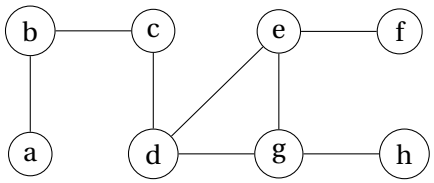

FIGURE 1 – Exemple de parcours en largeur au départ de a : a ⟶ b ⟶ c ⟶ d ⟶ e ⟶ g ⟶ f ⟶ h.

R La structure de donnée file permet de garantir la correction du parcours en largeur d'abord : on fait entrer en premier dans la file les voisins puis les descendants des voisins.

L'algorithme de parcours en largeur 1 est utilisé pour effectuer un traitement sur chaque sommet : il peut ne rien renvoyer, faire des modifications en place ; il peut aussi renvoyer une trace du parcours dans l'arbre dans une structure de type liste (P) qui enregistre le parcours pour un usage ultérieur.

a Terminaison du parcours en largeur

La terminaison du parcours en largeur peut être prouvée en considérant le variant de boucle

$$v = |F| + |\overline{V}| \tag{1}$$

c'est-à-dire la somme des éléments présents dans la file et du nombre de nœuds non visités.

Démonstration. À l'entrée de la boucle, si n est l'ordre du graphe, on a $|F| + |\overline{V}| = 1 + n - 1 = n$. Puis, à chaque itération, on retire un élément de la file et on ajoute ses p voisins en même temps qu'on marque les p voisins comme visités. L'évolution du variant au cours d'une itération s'écrit :

$$v = (|F|_d - 1 + p) + (|\overline{V}|_d - p) = |F|_d + |\overline{V}|_d - 1 \tag{2}$$

où l'on note $|F|_d$ et $|\overline{V}|_d$ les valeurs au début de l'itération de $|F|$ et $|\overline{V}|$.

À chaque tour de boucle, ce variant v décroit donc strictement de un et atteint nécessairement zéro au bout d'un certain nombre de tours. Lorsque le variant vaut zéro $|F| + |\overline{V}| = 0$, on a donc $|F| = 0$ et $|\overline{V}| = 0$. La file est nécessairement vide et tous les nœuds ont été visités. L'algorithme se termine puisque la condition de sortie est atteinte. ■

b Correction du parcours en largeur

Parcourir un graphe, à partir d'un sommet de départ s, cela veut dire trouver un chemin partant de s vers tous les sommets du graphe [2]. On remarque que tous les sommets du graphe sont à un moment ou un autre de l'algorithme **visités** et admis dans l'ensemble V : ceci vient du fait qu'on procède de proche en proche en ajoutant tous les voisins d'un sommet, sans distinction.

La correction peut se prouver en utilisant l'invariant de boucle $\mathcal{I}$:
«Pour chaque sommet v ajouté à V et enfilé dans F, il existe un chemin de s à v.»

Démonstration. On procède en montrant que l'invariant est vérifié à l'entrée de la boucle, conservé par les instructions de la boucle et donc vérifié à la sortie de la boucle.

- (**Initialisation**) : à l'entrée de la boucle, s est ajouté à V et est présent dans la file F. Le chemin de s à s existe trivialement.
- (**Conservation**) : on suppose que l'invariant est vérifié jusqu'à une certaine itération. On cherche à montrer qu'il l'est toujours à la fin de l'itération suivante. Lors de l'exécution de cette itération, un sommet v est défilé. Ce sommet faisait déjà parti de V et par hypothèse, comme l'invariant était vérifié jusqu'à présent, il existe un chemin de s à v. Puis, les voisins de v sont ajoutés à V et enfilés. Comme ils sont voisins, il existe donc un chemin de v à ces sommets et donc il existe un chemin de s à ces sommets. À la fin de l'itération, l'invariant est donc vérifié.
- (**Conclusion**) : à la fin de l'algorithme, il existe un chemin de s vers tous les sommets du graphe visités (ajoutés à V). Tous les sommets ont été parcourus.

■

2. On fait l'hypothèse que le graphe est connexe. S'il ne l'est pas, il suffit de recommencer la procédure avec un des sommets n'ayant pas été parcouru.

c Complexité du parcours en largeur

La complexité de cet algorithme est lié, comme toujours, aux structures de données utilisées. Pour qu'elle soit optimale, on considère donc que :

1. G, un graphe d'ordre n et de taille m, est implémenté par une **liste d'adjacence** g. Rechercher les voisins d'un sommet est alors une opération en $O(1)$.
2. la file d'attente F possède une complexité en $O(1)$ pour les opérations ENFILER et DÉFILER.
3. les listes P et V possèdent une complexité en $O(1)$ pour l'ajout en tête d'un élément AJOUTER.

Selon ces options choisies, lors du parcours en largeur, les instructions des boucles s'effectuent donc en un temps constant c.

$$C(n) = \sum_{k=0}^{n-1} \sum_{v \in g[k]} c \tag{3}$$

Dans le pire des cas, le graphe est complet : chaque sommet possède donc $n-1$ voisins et la boucle intérieur fait un nombre maximal d'itérations. La complexité dans le pire des cas est alors quadratique :

$$C(n) = \sum_{k=0}^{n-1} \sum_{v \in g[k]} c = c \sum_{k=0}^{n-1} \sum_{i=1}^{n-1} 1 = c(n-1) \sum_{k=0}^{n-1} 1 = cn(n-1) = O(n^2) \tag{4}$$

Si l'on considère un cas quelconque, c'est-à-dire un graphe non complet, on peut exprimer la complexité différemment. En déroulant les deux boucles, c'est-à-dire en écrivant les instructions les unes après les autres explicitement, on se rend compte que l'on parcours :

1. tous les sommets une fois et toutes les arêtes deux fois si le graphe n'est pas orienté,
2. tous les sommets une fois et toutes les arêtes une fois si le graphe est orienté.

C'est pourquoi on note généralement la complexité du parcours en largeur $O(n+m)$. C'est une expression qui montre que si le graphe est peu dense, alors la complexité du parcours n'est pas vraiment quadratique. Par contre, dans le pire des cas, le graphe est complet, $m = \frac{n(n-1)}{2}$ d'après le lemme des poignées de mains (cf. théorème 1) et on retrouve bien une complexité quadratique.

R Si on avait utilisé une matrice d'adjacence, on aurait été obligé de rechercher les voisins à chaque étape, c'est à dire de balayer une ligne de la matrice. Cette opération aurait eu un coût en $O(n)$. La complexité temporelle du parcours en serait impactée.

R Utiliser une liste Python pour implémenter une file d'attente n'est pas optimal. Si l'opération `append(x)` permet bien d'enfiler l'élément à la fin de la liste en une complexité $O(1)$, l'opération `pop(0)` qui permet de récupérer le premier élément de la liste (DÉFILER) est de com-

plexité $O(n)$, car il est nécessaire de réécrire tout le tableau dynamique sous-jacent.

Théorème 1 — Somme des degrés d'un graphe. Le nombre d'arêtes d'un graphe simple est égale à la moitié de la somme des degrés des sommets de ce graphe.

Plus formellement, soit $G = (S, A)$ un graphe simple alors on a :

$$2|A| = \sum_{s \in S} d(s) \tag{5}$$

R L'ensemble V n'est pas indispensable dans l'algorithme 1. On pourrait se servir de la liste qui enregistre le parcours. Néanmoins, son utilisation permet de bien découpler la sortie de l'algorithme (le parcours P) de son fonctionnement interne et ainsi de prouver la terminaison.

C Parcours en profondeur

■ **Définition 2 — Parcours en profondeur.** Parcourir en profondeur un graphe signifie qu'on cherche à visiter tous les voisins descendants d'un sommet qu'on découvre avant de parcourir les autres sommets découverts en même temps.

Vocabulary 2 — Depth First Search ⇜⇝ Parcours en profondeur

Le parcours en profondeur d'un graphe s'exprime naturellement récursivement (cf. algorithme 2). Il peut également s'exprimer de manière itérative (cf. algorithme 3) en utilisant une pile P afin gérer la découverte des voisins dans l'ordre de la profondeur du graphe.

Algorithme 2 Parcours en profondeur d'un graphe (version récursive)

```
1 : Fonction REC_PARCOURS_EN_PROFONDEUR(G, s, E)        ▷ s est un sommet de G
2 :     AJOUTER(E, s)                                    ▷ s est marqué exploré
3 :     pour chaque voisin x de s dans G répéter
4 :         si x ∉ V alors                               ▷ x n'a pas encore été découvert
5 :             REC_PARCOURS_EN_PROFONDEUR(G, x, E)
6 :     renvoyer E
```

La complexité de cet algorithme peut être calculé facilement si on considère son expression récursive (cf. algorithme 2). Soit un graphe $G = (S, A)$ possédant n sommets et m arêtes. On considère qu'en dehors des appels récursifs, la complexité de l'algorithme est constante. Par ailleurs, on effectue un nombre d'appels récursifs égal au nombre de voisin du sommet en cours d'exploration. C'est pourquoi, la complexité s'exprime :

$$T(S, A) = 1 + \nu_0 + T(S \setminus \{s_0\}, A \setminus \{a_0\}) \tag{6}$$

où s_0 désigne le sommet d'indice 0, ν_0 le nombre de voisins du sommet s_0 et a_0 les arêtes de s_0

Algorithme 3 Parcours en profondeur d'un graphe (version itérative)

```
1 : Fonction PARCOURS_EN_PROFONDEUR(G, s)                    ▷ s est un sommet de G
2 :     P ← une pile vide                                    ▷ P comme pile
3 :     E ← un ensemble vide                                 ▷ V comme explorés
4 :     C ← un liste vide                                    ▷ C pour le parcours
5 :     EMPILER(P,s)
6 :     tant que P n'est pas vide répéter
7 :         v ← DÉPILER(P)
8 :         AJOUTER(C, v)
9 :         si v ∉ E alors                                   ▷ v n'a pas encore été découvert
10 :            AJOUTER(E, x)
11 :            pour chaque voisin x de v dans G répéter
12 :                EMPILER(P, x)
13 :    renvoyer C                  ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

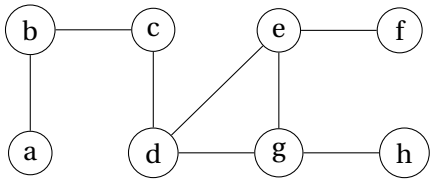

FIGURE 2 – Exemple de parcours en profondeur au départ de a : a ⟶ b ⟶ c ⟶ d ⟶ g ⟶ h ⟶ e ⟶ f

à ses voisins. On en déduit alors :

$$\begin{aligned} T(S,A) &= 1 + v_0 + 1 + v_1 + T(S \setminus \{s_0, s_1\}, A \setminus \{a_0, a_1\}) && (7)\\ &= 1 + 1 + 1 + \ldots + v_0 + v_1 + v_2 + \ldots + T(S \setminus \{s_0, s_1, s_2, \ldots\}, A \setminus \{a_0, a_1, a_2, \ldots\}) && (8)\\ &= n + \sum_{k=0}^{m-1} v_k && (9)\\ &= n + m && (10) \end{aligned}$$

La complexité de l'algorithme du parcours en profondeur est donc la même que celles du parcours en largeur et on la note $O(n+m)$.

D Trouver un chemin dans un graphe

On peut modifier l'algorithme 1 de parcours en largeur d'un graphe pour trouver un chemin reliant un sommet à un autre et connaître la longueur de la chaîne qui relie ces deux sommets. Il suffit pour cela de :

- garder la trace du prédécesseur (parent) du sommet visité sur le chemin,

- sortir du parcours dès qu'on a trouvé le sommet cherché (early exit),
- calculer le coût du chemin associé.

Le résultat est l'algorithme 4. Opérer cette recherche dans un graphe ainsi revient à chercher dans toutes les directions, c'est à dire sans tenir compte des distances déjà parcourues.

Algorithme 4 Longueur d'une chaîne via un parcours en largeur d'un graphe pondéré

```
1 : Fonction CD_PEL(G, a, b)                    ▷ Trouver un chemin de a à b et la distance associée
2 :     F ← une file d'attente vide                                          ▷ F comme file
3 :     V ← un ensemble vide                                                 ▷ V comme visités
4 :     P ← un dictionnaire vide                                             ▷ P comme parent
5 :     ENFILER(F,s)
6 :     AJOUTER(V,s)
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         si v est le sommet b alors                                       ▷ Objectif atteint, early exit
10 :            sortir de la boucle
11 :        pour chaque voisin x de v dans G répéter
12 :            si x ∉ V alors                                               ▷ x n'a pas encore été découvert
13 :                AJOUTER(V, x)
14 :                ENFILER(F, x)
15 :                P[x] ← v                                                 ▷ Garder la trace du parent
16 :    c ← 0                                                                ▷ Coût du chemin
17 :    s ← b
18 :    tant que s est différent de a répéter
19 :        c ← c + w(s, P[s])                   ▷ w est la fonction de valuation du graphe G
20 :        sommet ← P[s]                                                    ▷ On remonte à l'origine du chemin
21 :    renvoyer c
```

R Il faut noter néanmoins que le chemin trouvé et la distance associée issue de l'algorithme 4 n'est pas nécessairement la meilleure, notamment car on ne tient pas compte de la distance parcourue jusqu'au sommet recherché.

R Si le graphe n'est pas pondéré, il suffit de compter le nombre de sauts pour évaluer la distance, c'est-à-dire la fonction de valuation vaut toujours 1.

E Plus courts chemins dans les graphes pondérés

R Un graphe non pondéré peut-être vu comme un graphe pondéré dont la fonction de valuation vaut toujours 1. La distance entre deux sommets peut alors être interprétée comme

le nombre de sauts nécessaires pour atteindre un sommet.

Théorème 2 — Existence d'un plus court chemin. Dans un graphe pondéré **sans pondérations négatives**, il existe toujours un plus court chemin.

Démonstration. Un graphe pondéré possède un nombre fini de sommets et d'arêtes (cf. définition **??**). Il existe donc un nombre fini de chaînes entre les sommets du graphe. Comme les valuations du graphe ne sont pas négatives, c'est à dire que $\forall e \in E, w(e) \geqslant 0$, l'ensemble des longueurs de ces chaînes est une partie non vide de $\mathbb{N}$: elle possède donc un minimum. Parmi ces chaînes, il en existe donc nécessairement une dont la longueur est la plus petite, le plus court chemin. ■

■ **Définition 3 — Plus court chemin entre deux sommets d'un graphe.** Le plus court chemin entre deux sommets a et b d'un graphe G est une chaîne $\mathcal{C}_{ab}$ qui relie les deux sommets a et b et :

- qui comporte un minimum d'arêtes si G est un graphe non pondéré,
- dont le poids cumulé est le plus faible, c'est à dire $\min\limits_{\mathcal{C}_{ab} \in G} \left(\sum\limits_{e \in \mathcal{C}_{ab}} w(e) \right)$, dans le cas d'un graphe pondéré de fonction de valuation w.

■ **Définition 4 — Distance entre deux sommets.** La distance entre deux sommets d'un graphe est la longueur d'un plus court chemin entre ces deux sommets. Pour deux sommets a et b, on la note δ_{ab}. On a enfin :

$$\delta_{ab} = \min_{\mathcal{C}_{ab} \in G} \left(\sum_{e \in \mathcal{C}_{ab}} w(e) \right) \tag{11}$$

On se propose maintenant d'étudier les algorithmes les plus célèbres qui illustrent, dans différentes configurations, le concept de plus court chemin dans un graphe. La majorité de ces algorithmes reposent sur le principe d'optimalité de Bellman (cf. définition 5).

■ **Définition 5 — Principe d'optimalité de Bellman.** La solution optimale à un problème d'optimisation combinatoire présente la propriété suivante : quel que soit l'état initial et la décision initiale prise, les décisions qui restent à prendre pour construire une solution optimale forment une solution optimale par rapport à l'état qui résulte de la première décision [a].

a. PRINCIPLE OF OPTIMALITY. An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions[**bellman_theory_1954**].

Comme le montre la figure 3, on peut exprimer le plus court chemin récursivement en fonction du premier chemin choisi et du reste du graphe. Cela peut s'écrire formellement mathématiquement ou, plus simplement, comme suit :

Le plus court chemin est le chemin le plus court à choisir parmi :

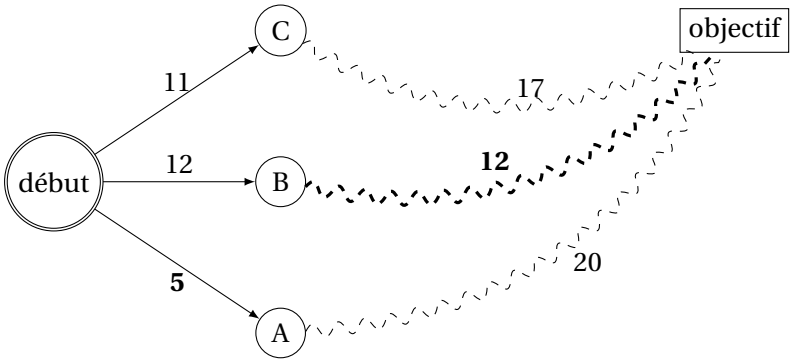

FIGURE 3 – Illustration du principe d'optimalité et de sous-structure optimale : trouver le plus court chemin dans ce graphe. Les nombres représentent la longueur d'un chemin. Les lignes droites sont des arrêtes. Les lignes ondulées indiquent les plus courts chemins dans le graphe : il faut imaginer qu'on n'a pas représenté tous les sommets.

- le chemin qui passe par A suivi par le chemin le plus court de A à l'objectif,
- le chemin qui passe par B suivi par le chemin le plus court de B à l'objectif,
- le chemin qui passe par C suivi par le chemin le plus court de C à l'objectif.

La résolution du problème amènera à choisir le chemin qui passe par B.

a Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford (cf. algorithme 5) calcule les plus courts chemins depuis un sommet de départ s_0. Cet algorithme repose sur la notion de saut, c'est-à-dire par combien de sommets dois-je passer pour que la distance soit la plus courte. Il prend en compte tous les chemins possibles et en choisit la distance minimale à chaque fois : en considérant les chemins à un saut, à deux sauts, ... jusqu'à $n-1$ sauts si n est l'ordre du graphe, alors on est sûr de trouver les plus courts chemin de s_0 vers les autres sommets du graphe.

Il est important de souligner que cet algorithme s'applique uniquement à des **graphes pondérés et orientés** dont les pondérations peuvent être négatives mais **sans cycles de longueur négative** [**bellman_routing_1958**, **ford_jr_network_1956**, **moore_shortest_1959**].

R Les poids négatifs peuvent représenter des transferts de flux (énergie ou chaleur en physique-chimie, argent en économie) et sont donc très courants.

R Les cycles de poids négatif ne peuvent pas permettre de définir une distance minimale : à chaque itération du cycle, la distance diminue.

R Cet algorithme ne s'applique pas à des graphes non orientés pour la raison suivante : les arêtes d'un graphe non orienté sont des cycles car la relation est dans les deux sens. Donc chaque arête de poids négatif est un cycle de poids négatif.

La complexité de l'algorithme 5 est en $O(nm)$ si n est l'ordre du graphe et m sa taille.

Algorithme 5 Algorithme de Bellman-Ford, plus courts chemins à partir d'un sommet donné

```
1 : Fonction BELLMAN_FORD(G = (S, A, w), s_0)
2 :     d ← dictionnaire vide                                   ▷ distances au sommet a
3 :     d[s] ← w(s_0, s)                          ▷ w(s_0, s) = +∞ si s n'est pas voisin de s_0, 0 si s = s_0
4 :     Π ← un dictionnaire vide     ▷ Π[s] est le parent de s dans le plus court chemin de s_0 à s
5 :     pour _ de 1 à |S| − 1 répéter                                ▷ Répéter n − 1 fois
6 :         pour (u, v) = a ∈ A répéter                    ▷ Pour toutes les arêtes du graphe
7 :             si d[v] > d[u] + w(u, v) alors   ▷ Si le chemin est plus court de s_0 à v passe par u
8 :                 d[v] ← d[u] + w(u, v)                  ▷ Mises à jour des distances des voisins
9 :                 Π[v] ← u                                ▷ Pour garder la tracer du chemin
10 :    renvoyer d, Π
```

■ **Exemple 1 — Application de l'algorithme de Bellman-Ford.** On se propose d'appliquer l'algorithme 5 au graphe pondéré et orienté représenté sur la figure 4. On note qu'il contient une pondération négative de b à f mais pas de cycle à pondération négative. Le tableau 1 représente les distances successivement trouvées à chaque itération.

On observe que le chemin de a à f emprunte bien l'arc de pondération négative.

Il faut noter que l'algorithme a convergé avant la fin de l'itération dans cas. C'est un des axes d'amélioration de cet algorithme.

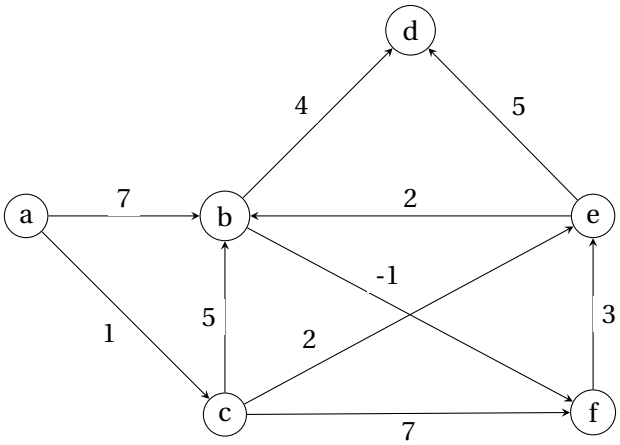

FIGURE 4 – Graphe pondéré et orienté à valeurs positives et négatives pour l'application de l'algorithme de Bellman-Ford.

N° d'itération	a	b	c	d	e	f
1	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
2	0	5	1	8	3	6
3	0	5	1	8	3	4
4	0	5	1	8	3	4
5	0	5	1	8	3	4

TABLE 1 – Tableau d des distances au sommet a successivement trouvées au cours de l'algorithme de Bellman-Ford appliqué au graphe de la figure 4

■ **Exemple 2 — Protocole de routage RIP.** Le protocole de routage RIP utilise l'algorithme de Bellman-Ford pour trouver les plus cours chemins dans un réseau de routeur. Il est moins adapté que OSPF pour les grands réseau à cause de sa complexité en $O(nm)$.

b Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall [**floyd_algorithm_1962**, **roy_transitivite_1959**, **warshall_theorem_1962**] est l'application de la programmation dynamique[3] à la recherche de **l'existence d'un chemin entre toutes les paires de sommets d'un graphe orienté et pondéré**. Les distances trouvées sont les plus courtes. **Les pondérations du graphe peuvent être négatives mais on exclue tout circuit de poids strictement négatif**.

Soit un graphe orienté et pondéré $G = (V, E, w)$. G peut être modélisé par une matrice d'adjacence M

$$\forall i, j \in [\![0, |V|-1]\!], M = \begin{cases} w(v_i, v_j) & \text{si } (v_i, v_j) \in E \\ +\infty & \text{si } (v_i, v_j) \notin E \\ 0 & \text{si } i = j \end{cases} \tag{12}$$

Un exemple de graphe associé à la matrice d'adjacence :

$$M_{\text{init}} = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & 9 & 0 \end{pmatrix} \tag{13}$$

est donné sur la figure 5. Sur cet exemple, le chemin le plus court de v_4 à v_3 vaut 3 et passe par v_2.

Pour trouver le plus court chemin entre deux sommets, on essaye tous les chemins de toutes les longueurs possibles et on ne garde que les plus courts. Chaque étape p de l'algorithme de Floyd-Warshall est donc constitué d'un allongement **éventuel** du chemin par le sommet v_p. À l'étape p, on associe une matrice M_p qui contient la longueur des chemins les plus courts d'un sommet à un autre passant par des sommets de l'ensemble $\{v_0, v_1, \ldots, v_p\}$. On

3. cf. programme de seconde année

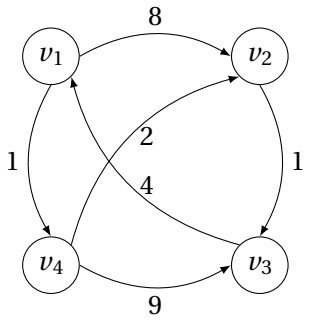

FIGURE 5 – Exemple de graphe orienté et pondéré pour expliquer le concept de matrice d'adjacence.

construit ainsi une suite de matrice finie $\left(M_p\right)_{p\in[\![0,n-1]\!]}$ et on initialise la matrice M avec avec M_{init}.

Supposons qu'on dispose de M_{p-1}. Considérons un chemin $\mathcal{C}$ entre v_i et v_j dont la longueur est minimale et dont les sommets intermédiaires sont dans $\{v_0, v_1, \ldots, v_{p-1}\}$, $p \leqslant n$. Pour un tel chemin :

- soit $\mathcal{C}$ passe par v_p. Dans ce cas, $\mathcal{C}$ est la réunion de deux chemins dont les sommets sont dans $\{v_0, v_1, \ldots, v_p\}$: celui de v_i à v_p et celui de v_p à v_j.
- soit $\mathcal{C}$ ne passe pas par v_p.

Entre ces deux chemins, on choisira le chemin le plus court.

On peut traduire notre explication ci-dessus par la relation de récurrence suivante :

$$\forall p \in [\![1,n]\!], \forall i,j \in [\![0,n-1]\!], M_p(i,j) = \min\left(M_{p-1}(i,j), M_{p-1}(i,p) + M_{p-1}(p,j)\right) \tag{14}$$

Pour $p = 0$, on pose $M_0 = M_{\text{init}}$.

L'algorithme de Floyd-Warshall 6 est un bel exemple de programmation dynamique. Sa complexité temporelle est en $O(n^3)$. Il peut être programmé en place.

Algorithme 6 Algorithme de Floyd-Warshall, plus courts chemins entre toutes les paires de sommet

```
1 : Fonction FLOYD_WARSHALL(G = (S, A, w))
2 :     M ← la matrice d'adjacence de G
3 :     pour p de 1 à |S| répéter
4 :         pour i de 0 à |S| − 1 répéter
5 :             pour i de 0 à |S| − 1 répéter
6 :                 M_p(i, j) = min(M_{p−1}(i, j), M_{p−1}(i, p) + M_{p−1}(p, j))
7 :     renvoyer M
```

R Cet algorithme effectue le même raisonnement que Bellman-Ford mais avec un vision globale, à l'échelle du graphe tout entier, pas uniquement par rapport à un sommet de départ.

■ **Exemple 3 — Application de l'algorithme de Floyd-Warshall.** Si on applique l'algorithme au graphe de la figure 5, alors on obtient la série de matrices suivantes :

$$M_0 = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & 9 & 0 \end{pmatrix} \tag{15}$$

$$M_1 = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & 12 & 0 & 5 \\ +\infty & 2 & 9 & 0 \end{pmatrix} \tag{16}$$

$$M_2 = \begin{pmatrix} 0 & 8 & 9 & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & 12 & 0 & 5 \\ +\infty & 2 & 3 & 0 \end{pmatrix} \tag{17}$$

$$M_3 = \begin{pmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \tag{18}$$

$$M_4 = \begin{pmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \tag{19}$$

c Algorithme de Dijkstra

L'algorithme de Dijkstra [4] [**dijkstra_note_1959**] s'applique à des **graphes pondérés** $G = (V, E, w)$ **dont la valuation est positive**, c'est à dire que $\forall e \in E, w(e) \geqslant 0$. C'est un algorithme **glouton optimal** qui trouve les plus courts chemins entre un sommet particulier $s_0 \in S$ et tous les autres sommets d'un graphe. Pour cela, l'algorithme classe les différents sommets par ordre croissant de leur distance minimale au sommet de départ : **c'est un parcours en largeur qui utilise une file de priorités** : lorsqu'on insère un nouvel élément dans cette file, celui-ci est placé d'après son niveau de priorité, le plus prioritaire en premier. Dans notre cas, la priorité est la distance la plus faible : la plus petite distance se situe en tête de la file.

Algorithme 7 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

```
1 : Fonction DIJKSTRA(G = (S, A, w), s_0)        ▷ Trouver les plus courts chemins à partir de a ∈ V
2 :     Δ ← s_0                                  ▷ Δ est le dictionnaire des sommets dont on connaît la distance à s_0
3 :     Π ← ∅                                    ▷ Π[s] est le parent de s dans le plus court chemin de s_0 à s
4 :     d ← ∅                                    ▷ l'ensemble des distances au sommet s_0
5 :     ∀ s ∈ V, d[s] ← w(s_0, s)                ▷ w(s_0, s) = +∞ si s n'est pas voisin de s_0, 0 si s = s_0
6 :     tant que Δ̄ n'est pas vide répéter        ▷ Δ̄ : sommets dont la distance n'est pas connue
7 :         Choisir u dans Δ̄ tel que d[u] = min(d[v], v ∈ Δ̄)        ▷ Choix glouton!
8 :         Δ = Δ ∪ {u}                          ▷ On prend la plus courte distance à s_0 dans Δ̄
9 :         pour x ∈ V_G(u) ∩ Δ̄ répéter          ▷ pour tous les voisins de u dans Δ̄
10 :            si d[x] > d[u] + w(u, x) alors
11 :                d[x] ← d[u] + w(u, x)        ▷ Mises à jour des distances des voisins
12 :                Π[x] ← u                     ▷ Pour garder la tracer du chemin le plus court
13 :    renvoyer d, Π
```

R Il faut remarquer que les boucles imbriquées de cet algorithme peuvent être comprises comme deux étapes successives de la manière suivante :

1. On choisit un nouveau sommet u de G à chaque tour de boucle *tant que* qui est tel que $d[u]$ est la plus petite des valeurs accessibles dans $\bar{\Delta}$. **C'est le voisin d'un sommet de $\bar{\Delta}$ le plus proche de s_0.** Ce sommet u est alors inséré dans l'ensemble Δ : c'est la **phase de transfert** de u de $\bar{\Delta}$ à Δ.
2. Lors de la boucle *pour*, on met à jour les distances des voisins de u qui n'ont pas encore été découverts. En effet, si x n'est pas un voisin de u, alors il n'existe par d'arête entre u et x et $w(u, x) = +\infty$. La mise à jour de la distance n'a donc pas lieu, on n'a pas trouvé une meilleure distance à x. C'est la **phase de mise à jour des distances** des voisins de u.

L'algorithme de Dijkstra procède donc de proche en proche.

4. à prononcer "Daillekstra"

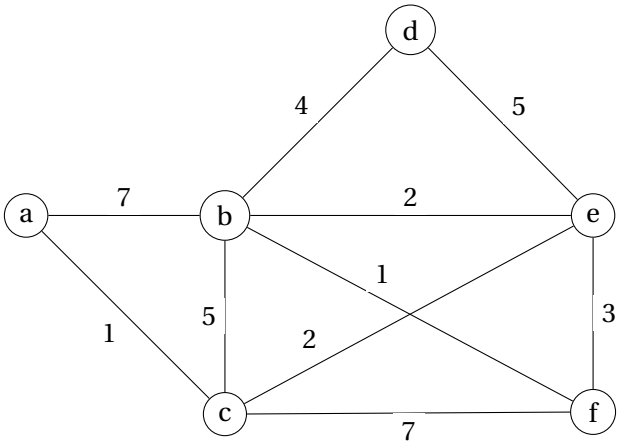

FIGURE 6 – Graphe pondéré à valeurs positives pour l'application de l'algorithme de Dijsktra.

■ **Exemple 4 — Application de l'algorithme de Dijkstra.** On se propose d'appliquer l'algorithme 7 au graphe représenté sur la figure 6. Le tableau 2 représente les distances successivement trouvées à chaque tour de boucle *tant que* de l'algorithme. En rouge figurent les distances les plus courtes à a à chaque tour. On observe également que certaines distances sont mises à jour sans pour autant que le sommet soit sélectionné au tour suivant.

À la fin de l'algorithme, on note donc que les distances les plus courtes de a à b, c, d, e, f sont $[5, 1, 8, 3, 6]$. Le chemin le plus court de a à b est donc $a \to c \to e \to b$. Le plus court de a à f est $a \to c \to e \to f$. C'est la structure de données Π qui garde en mémoire le prédécesseur (parent) d'un sommet sur le chemin le plus court qui permettra de reconstituer les chemins.

Δ	a	b	c	d	e	f	$\bar{\Delta}$
$\{\}$	0	7	1	$+\infty$	$+\infty$	$+\infty$	$\{a,b,c,d,e,f\}$
$\{a\}$	.	7	1	$+\infty$	$+\infty$	$+\infty$	$\{b,c,d,e,f\}$
$\{a,c\}$	.	6	.	$+\infty$	3	8	$\{b,d,e,f\}$
$\{a,c,e\}$	.	5	.	8	.	6	$\{b,d,f\}$
$\{a,c,e,b\}$	.	.	.	8	.	6	$\{d,f\}$
$\{a,c,e,b,f\}$	.	.	.	8	.	.	$\{d\}$
$\{a,c,e,b,f,d\}$	.	.	.	.	.	.	$\{\}$

TABLE 2 – Tableau d des distances au sommet a successivement trouvées au cours de l'algorithme de Dijkstra appliqué au graphe de la figure 6

Théorème 3 — L'algorithme de Dijkstra se termine et est correct.

Démonstration. **Correction de l'algorithme** : à chaque étape de cet algorithme, on peut distinguer deux ensembles de sommets : l'ensemble Δ est constitué des éléments dont on connaît la distance la plus courte à s_0 et l'ensemble complémentaire $\bar{\Delta}$ qui contient les autres sommets.

D'après le principe d'optimalité, tout chemin plus court vers un sommet de $\bar{\Delta}$ passera nécessairement par un sommet de Δ. Ceci s'écrit :

$$\forall u \in \bar{\Delta}, \delta_{s_0 u} = \min\left(\delta_{s_0 v} + w[v,u], v \in \Delta\right) \tag{20}$$

où l'on note $\delta_{s_0 u}$ la distance la plus courte de s_0 à u.

On souhaite montrer qu'à la fin de la boucle tant que (lignes 6-12), d contient les distances les plus courtes vers tous les sommets de Δ. On peut donc formuler un invariant de boucle $\mathcal{I}$ comme suit :

$$\forall u \in \Delta, d[u] = \delta_{s_0 u} \tag{21}$$

$$\forall x \in \bar{\Delta}, d[x] = \min\left(d[v] + w[v,x], v \in \Delta\right) \tag{22}$$

À l'entrée de la boucle, l'ensemble Δ ne contient que le sommet de départ s_0. On a $d[s_0] = 0$, ce qui est la distance minimale. Pour les autres sommets de $\bar{\Delta}$, d contient :

- une valeur infinie si ce sommet n'est pas un voisin de s_0, ce qui, à cette étape de l'algorithme est le mieux qu'on puisse trouver,
- le poids de l'arête venant de s_0 s'il s'agit d'un voisin, ce qui, à cette étape de l'algorithme est le mieux que l'on puisse trouver également.

On peut donc affirmer que d contient les distances entre s_0 et tous les sommets de Δ. L'invariant $\mathcal{I}$ est vérifié à l'entrée de la boucle.

On se place maintenant à une étape quelconque de la boucle et on cherche à vérifier que l'invariant n'est pas modifié par les instructions de la boucle. Notre hypothèse $\mathcal{H}$ est que toutes les itérations précédentes sont vérifient l'invariant. À l'entrée de la boucle on sélectionne un nouveau sommet u, le premier de la file de priorités.

u entre dans Δ, c'est à dire que on a choisi u tel que $u \in \bar{\Delta}$ et $\forall v \in \bar{\Delta}, d[u] \leqslant d[v]$. Ce choix glouton garantit que tous les autres chemins possibles de s_0 à u sont plus longs. On a donc nécessairement $d[u] = \delta_{s_0 u}$. La première partie de l'invariant est vérifiée.

Considérons un sommet x de $\bar{\Delta}$ voisin de u. Si passer par u conduit à une distance plus court pour aller à x, alors $d[x]$ est mise à jour. Comme $d[u] = \delta_{s_0 u}$, on a $d[u] + w(u,x) = \delta_{s_0 v} + w(u,x)$. Comme les pondérations du graphe sont positives, tout autre chemin conduisant de u à x ne peut être que plus long, puisque x est voisin de u. On est donc est certain que, à la fin de la boucle *pour*, $d[x] = \delta_{s_0 v} + w(u,x)$ est la distance minimale telle qu'on peut la connaître pour l'instant via les sommets de Δ : $\forall x \in \bar{\Delta}, d[x] = \min\left(d[v] + w[v,x], v \in \Delta\right)$.

Conclusion : L'invariant est vérifié à la fin de l'algorithme. d contient donc les distances vers tous les sommets de Δ. Comme $Delta = S$, on a donc $\forall u \in S, d[u] = \delta_{s_0 u}$

Terminaison de l'algorithme : avant la boucle *tant que*, $\bar{\Delta}$ possède $n-1$ éléments, si $n \in \mathbb{N}^\star$ est l'ordre du graphe. À chaque tour de boucle *tant que*, l'ensemble $\bar{\Delta}$ décroît strictement d'un élément et atteint donc nécessairement zéro. Le cardinal de $\bar{\Delta}$ est donc un variant de boucle. L'algorithme se termine lorsque le cardinal de $\bar{\Delta}$ atteint zéro. ■

La complexité de l'algorithme de Dijsktra dépend de l'ordre n du graphe considéré et de sa taille m. La boucle *tant que* effectue exactement $n-1$ tours. La boucle *pour* effectue à chaque fois un nombre de tour égal au nombre d'arêtes non découvertes qui partent du sommet u considéré et vont vers un sommet voisin de $\bar{\Delta}$. On ne découvre une arête qu'une seule fois, puisque le sommet u est transféré dans Δ au début de la boucle. Au final, on exécute donc la mise à jour des distances un nombre de fois égal à la taille m du graphe, c'est à dire son nombre d'arêtes. En notant la complexité du transfert c_t et la complexité de la mise à jour des distances c_d et en déroulant la boucle *tant que*, on peut écrire :

$$C(n,m) = (n-1)c_t + mc_d \tag{23}$$

Les complexités c_d et c_t dépendent naturellement des structures de données utilisées pour implémenter l'algorithme.

Si on choisit une implémentation de d par un tableau, alors on a besoin de rechercher le minimum des distances pour effectuer le transfert : cela s'effectue au prix d'un tri du tableau au minimum en $c_t = O(n\log n)$. Un accès aux éléments du tableau pour la mise à jour est en $c_d = O(1)$. On a donc $C(n) = (n-1)O(n\log n) + mO(1) = O(n^2 \log n)$.

Si d est implémentée par une file à priorités (un tas) comme le propose Johnson [**johnson_efficient_1977**], alors on a $c_t = O(\log n)$ et $c_d = O(\log n)$. La complexité est alors en $C(n) = (n+m)\log n$. Cependant, pour que le tas soit une implémentation pertinente, il est nécessaire que $m = O(\frac{n^2}{\log n})$, c'est à dire que le graphe ne soit pas complet, voire un peu creux!

■ **Exemple 5 — Usage de l'algorithme de Dijkstra .** Le protocole de routage OSPF implémente l'algorithme de Dijsktra. C'est un protocole qui permet d'automatiser le routage sur les réseaux internes des opérateurs de télécommunications. Les routeurs sont les sommets du graphe et les liaisons réseaux les arêtes. La pondération associée à une liaison entre deux routeurs est calculée à partir des performances en termes de débit de la liaison. Plus une liaison possède un débit élevé, plus la distance diminue.

OSPF est capable de relier des centaines de routeurs entre eux, chaque routeur relayant les paquets IP de proche en proche en utilisant le plus court chemin de son point de vue [a]. Le protocole garantit le routage des paquets par les plus courts chemins en temps réel. Chaque routeur calcule ses propres routes vers toutes les destinations, périodiquement. Si une liaison réseau s'effondre, le routeurs en sont informés et recalculent d'autres routes immédiatement. La puissance de calcul nécessaire pour exécuter l'algorithme sur un routeur, même dans le cas d'un réseau d'une centaine de routeur, est relativement faible car la plupart des réseaux de télécommunications sont des graphes relativement peu denses. Ce n'est pas rentable de créer des graphes de télécommunications complets, même si ce serait intéressant pour le consommateur et très robuste!

a. Cela fonctionne grâce au principe d'optimalité de Bellman!

d A*

L'algorithme A* [5] est un algorithme couteau suisse qui peut être considéré comme un algorithme de Dijkstra muni d'une heuristique : là où Dijkstra ne tient compte que du coût du chemin déjà parcouru, A* considère ce coût et une heuristique qui l'informe sur le reste du chemin à parcourir. Il faut bien remarquer que le chemin qu'il reste à parcourir n'est pas nécessairement déjà exploré : parfois il est même impossible d'explorer tout le graphe. Si l'heuristique pour évaluer le reste du chemin à parcourir est bien choisie, alors A* converge aussi vite voire plus vite que Dijkstra[**unswmechatronics_dijkstras_2013**].

■ **Définition 6 — Heuristique admissible.** Une heuristique $\mathcal{H}$ est admissible si pour tout sommet du graphe, $\mathcal{H}(s)$ est une borne inférieure de la plus courte distance séparant le sommet de départ du sommet d'arrivée.

■ **Définition 7 — Heuristique cohérente.** Une heuristique $\mathcal{H}$ est cohérente si pour tout arête (s,p) du graphe $G=(V,E,w)$, $\mathcal{H}(s) \leqslant \mathcal{H}(p) + w(s,p)$.

■ **Définition 8 — Heuristique monotone.** Une heuristique $\mathcal{H}$ est monotone si l'estimation du coût **total** du chemin ne décroît pas lors du passage d'un sommet à ses successeurs. Pour un chemin $(s_0, s_1, \ldots, s_n)$, on $\forall 0 \leqslant i < j \leqslant n, c(s_j) \geqslant c(s_i)$.

Soit $G=(V,E,w)$ un graphe orienté. Soit d la fonction de distance utilisée par l'algorithme de Dijkstra (cf. algorithme 7). A*, muni d'une fonction h permettant d'évaluer l'heuristique, calcule alors le coût total pour aller jusqu'à un sommet p comme suit :

$$c(p) = d(p) + h(p) \tag{24}$$

Le coût obtenu n'est pas nécessairement optimal, il dépend de l'heuristique.

Supposons que l'on cherche le chemin le plus court entre les sommets s_0 et p. Supposons que l'on connaisse un chemin optimal entre s_0 et un sommet s. Alors on peut écrire que le coût total vers le sommet p vaut :

$$\begin{aligned} c(p) &= d(p) + h(p) && (25)\\ &= d(s) + w(s,p) + h(p) && (26)\\ &= d(s) + h(s) + w(s,p) - h(s) + h(p) && (27)\\ &= c(s) + w(s,p) - h(s) + h(p) && (28) \end{aligned}$$

Ainsi, on peut voir l'algorithme A* comme un algorithme de Dijkstra muni :

- de la distance $\tilde{d} = c$,
- et de la pondération $\tilde{w}(s,p) = w(s,p) - h(s) + h(p)$.

L'algorithme 8 donne le détail de la procédure à suivre.

5. prononcer *A étoile* ou *A star*

Algorithme 8 A*

Fonction ASTAR($G = (V, E, w)$, a) ▷ Sommet de départ a
  $\Delta \leftarrow a$
  $\Pi \leftarrow$
  $\tilde{d} \leftarrow$ l'ensemble des distances au sommet a
  $\forall s \in V, \tilde{d}[s] \leftarrow \tilde{w}(a, s)$ ▷ Le graphe est partiel, l'heuristique fait le reste
  tant que $\bar{\Delta}$ n'est pas vide **répéter** ▷ $\bar{\Delta}$: sommets dont la distance n'est pas connue
    Choisir u dans $\bar{\Delta}$ tel que $\tilde{d}[u] = \min\left(\tilde{d}[v], v \in \bar{\Delta}\right)$
    $\Delta = \Delta \cup \{u\}$
    pour $x \in \bar{\Delta}$ **répéter** ▷ Ou bien $x \in \mathcal{N}_G(u) \cap \bar{\Delta}$, pour tous les voisins de u dans $\bar{\Delta}$
      si $\tilde{d}[x] > \tilde{d}[u] + \tilde{w}(u, x)$ **alors**
        $\tilde{d}[x] \leftarrow \tilde{d}[u] + \tilde{w}(u, x)$
        $\Pi[x] \leftarrow u$ ▷ Pour garder la tracer du chemin le plus court
  renvoyer $\tilde{d}$, Π