Graphes

Les graphes servent à la représentation de nombreux problèmes : comment choisir un chemin le plus court possible qui passe une et une seule fois par chaque ville d'un ensemble donné (**Problème du voyageur de commerce**) ? Comment créer un emploi du temps en respectant les contraintes physiques (un seul cours par salle et par heure !) et celles de chacun ?
Un graphe représente un ensemble avec ses connexions : réseau de communication, réseau routier, circuit électronique, relations sociales, etc.

Terminologie

Définition 1

Un **graphe** est un couple (S, A) où

- S est un ensemble fini d'éléments appelés **sommets** (ou **noeuds**)
- A un ensemble d'arêtes, c'est-à-dire de couples de sommets (les **extrémités** de l'arête). *On imposera ici que les sommets doivent être distincts.*

Exemple

Le problème des **ponts de Königsberg** (Euler, 1735) a préfiguré la topologie et la théorie des graphes. Il consistait à déterminer l'existence d'un chemin, partant d'un point et y revenant, passant une fois exactement par chacun des sept ponts de la ville.

- Une arête peut être **orientée**, avec un sommet de départ et un sommet d'arrivée, ou non, auquel cas le sens n'importe pas : le couple formant l'arête est non ordonné. Deux sommets **reliés par une arête**, c'est-à-dire qui en sont les extrémités, sont dits **adjacents**. On trouvera le terme **arc** pour désigner une arête orientée.
- L'**ordre** d'un graphe est le nombre de sommets de ce graphe. Le **degré** $d(s)$ d'un sommet s est le nombre de sommets qui lui sont adjacents, le **degré** d'un graphe est le degré maximum de ses sommets.
- Dans le cadre d'un graphe orienté, on peut distinguer les arêtes sortantes et entrantes. Le degré sortant $d_+(s)$ (pour un graphe orienté d'un sommet s est le nombre de sommets t tels que (s, t) est une arête du graphe. On définit de même le degré entrant $d_-(s)$. Le degré d'un sommet d'un graphe orienté est la somme de son degré entrant et de son degré sortant.
- Un graphe est **pondéré** lorsqu'à chaque arête est associé un nombre réel, le **poids**.
- Un graphe est **simple** lorsqu'aucun sommet n'est adjacent à lui-même. On n'étudie dans la suite de ce cours que des graphes simples. C'est un **multigraphe** lorsqu'il existe une arête reliant un sommet à lui-même ou plusieurs arêtes reliant les deux mêmes sommets.
- Un graphe est dit planaire, lorsqu'il possède une représentation pour laquelle ses arêtes ne se coupent pas.
- Un **sous-graphe** G' de $G = (S, A)$ est un couple (S', A'), avec $S' \subseteq S'$, et A' un ensemble d'arêtes dont les deux extrémités sont dans S'.
- Un graphe $G = (S, A)$ est **biparti** lorsqu'il existe (S_1, S_2) partition de S tels qu'aucune arête de A n'ait ses deux extrémités dans le même ensemble de (S_1, S_2)

Proposition

Si G est un graphe non orienté (et simple), on a $\sum_{s \in S} \deg(s) = 2|A|$

Proposition

Dans un graphe non orienté, $|A| \leq \frac{|S| \times (|S|-1)}{2}$. Dans un graphe orienté, $|A| \leq |S| \times (|S|-1)$.

Définition 2

Un **chemin** de longueur k reliant deux sommets a (son point de départ) et b (son point d'arrivée) est une suite finie $a = x_0, x_1, ..., x_k = b$ de sommets tels que pour tout $i \in [\![0, k-1]\!]$, (x_i, x_{i+1}) est une arête du graphe. La **distance** entre deux sommets a et b d'un graphe g est la longueur du plus petit chemin, s'il existe, qui relie a et b. Dans les graphes non orientés, un ensemble de sommets qui sont tous à distance finie les uns des autres est dit **connexe**. Tout graphe non orienté peut être décomposé en **composantes connexes**, qui sont des sous-graphes connexes maximaux.

Remarques :

- Tout sommet est accessible depuis lui-même par un chemin de longueur 0.
- S'il existe un chemin de a vers b, on dit également que b est **accessible** depuis a. Dans un graphe non orienté, la relation d'accessibilité est une relation d'équivalence.
- Dans le cadre des graphes non orientés, une composante connexe est une classe d'équivalence pour l'accessibilité.

Définition 3

Un graphe orienté est dit **fortement connexe** lorsque pour tout couple (i, j) de sommets, il existe un chemin de i vers j et un chemin de j vers i. Un graphe orienté peut être décomposé en **composantes fortement connexes**, lesquelles sont des sous-graphes fortement connexes maximaux.

Définition 4

Un chemin est dit **simple** s'il ne passe pas deux fois par la même arête. Un **cycle** est un chemin simple dont le point de départ et le point d'arrivée sont confondus.

Dans un graphe non orienté, un cycle est de longueur au moins 3. Un cycle est dit **élémentaire** si la seule répétition de sommets est celle de ses extrémités, donc lorsque il ne contient pas d'autre cycle.

Théorème

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Lemme

Si dans un graphe G tout sommet est de degré supérieur ou égal à 2, alors G possède au moins un cycle.

Proposition

Un graphe acyclique d'ordre n comporte au plus $n-1$ arêtes.

Arbres

Un arbre est un graphe connexe acyclique. *Rappel :* Un tel graphe à n sommets possède $n-1$ arêtes.

Théorème

Pour un graphe G d'ordre n, les assertions suivantes sont équivalentes :

1. G est un arbre ;
2. G est un graphe connexe à $n-1$ arêtes ;
3. G est un graphe acyclique à $n-1$ arêtes.

Remarque

Un arbre étant un graphe connexe acyclique non orienté, quels que soient deux de ses sommets, il existe un unique chemin reliant l'un à l'autre. On peut donc choisir de façon arbitraire tout sommet d'un arbre pour en faire sa racine, puis orienter les arêtes de façon à ce qu'il existe un chemin de la racine vers tout autre sommet. On obtient un **arbre enraciné**.

Un graphe non orienté acyclique est appelé une **forêt** : ses composantes connexes sont des arbres.
Exercice : Montrer qu'une forêt est un graphe biparti.
Montrer qu'un graphe est biparti si et seulement s'il ne contient aucun cycle de longueur impaire.

Implémentation : deux exemples

Les représentations proposées sont particulièrement adaptées au cas des graphes orientés, mais la représentation de graphes non orientés se fait en dédoublant toute arête non orientée (a, b) en deux arêtes orientées (a, b) et (b, a). Deux représentations principales sont exposées ici : par listes d'adjacence et par matrice d'adjacence.

Listes d'adjacence

Un graphe peut être représenté par une liste d'adjacence : liste qui associe à chaque sommet la liste de ses voisins. Cette représentation permet l'ajout ou la suppression facile de sommets et d'arêtes, mais l'accès à un sommet ou une arête en particulier est (au pire) linéaire en le nombre d'arêtes. Dans le cas d'un graphe non orienté, il faut penser à ajouter ou supprimer non seulement l'arête (a, b) mais également l'arête (b, a).
La quantité de mémoire utilisée est minimale, en $\Theta(n + p)$ où n est le nombre de sommets, et p le nombre d'arêtes.

Si on souhaite opérer sur des graphes statiques, sur lesquels on n'ajoute ni ne supprime de sommet ou d'arête, on peut considérer les sommets numérotés de 0 à $n - 1$, et les listes d'adjacences gardées en mémoire dans un tableau. On accède alors à la liste d'adjacence du sommet i en temps constant.
En Caml, on peut définir le type graphe ainsi :

```
type 'a sommet = {nom :'a; voisins:'a list};;
type 'a graphe = 'a sommet array;;
ou: type 'a graphe = 'a sommet list;;
```

Exercices : écrire des fonctions qui ajoutent et supprime une arête (de même pour un sommet) dans un graphe sous forme de listes d'adjacence, dans le cadre des graphes orientés et non orientés. En C, on utilisera des tableaux statiques, à deux dimensions, dont les lignes représentent chaque liste, soit avec une sentinelle indiquant qu'on a épuisé l'ensemble des arêtes partant d'un sommet donné, soit avec, en premier indice, un indicateur de taille de la liste d'adjacence de ce sommet.

Matrices d'adjacence

Si les sommets d'un graphe G sont ordonnées, et mis en bijection (par exemple, à l'aide d'un dictionnaire) avec des entiers consécutifs de 0 à $n-1$ (avec lesquels on choisit de les confondre), le graphe G peut être représenté par la matrice $M = (m_{i,j})_{i,j}$ où $m_{i,j}$ vaut 1 s'il existe une arête du sommet i vers le sommet j, et 0 sinon. Puisqu'on impose qu'une arête joigne deux sommets distincts, la diagonale est remplie de 0.
L'ajout et la suppression d'une arête se fait à coût constant, mais la représentation se fait en coût spatial quadratique contre un coût spatial en $O(|S| + |A|)$ pour les listes d'adjacence. Si le

graphe comporte peu d'arêtes et de nombreux sommets, cette représentation n'est pas optimale.

Exercices

1. Écrire une fonction qui détermine si un graphe représenté par une matrice d'adjacence est orienté ;
2. Écrire une fonction qui supprime le caractère orienté d'un graphe ;
3. Écrire deux fonctions passant d'une représentation d'un graphe à l'autre ;
4. Écrire une fonction déterminant si un cycle est élémentaire.
5. Écrire une fonction qui détermine si deux éléments a et b sont bien reliés par un chemin donné par une liste de sommets intermédiaires.

Parcours d'un graphe

Pour simplifier, on notera ici **successeur** de a un sommet b tel qu'il existe une arête de a à b.
Parcourir un graphe, c'est en énumérer les sommets accessibles par un chemin à partir d'un sommet donné, généralement pour calculer quelque chose à partir de ces sommets : taille du graphe, composantes connexes, etc. De mnaière générale, on doit garder en mémoire les sommets déjà visités, ainsi qu'une liste, qui croît et décroît au fur et à mesure du déroulement de l'algorithme, des sommets à traiter par la suite.
L'algorithme général suit la structure suivante, utilisant un graphe `G` et un de ses sommets `depart`

```
def parcoursLargeur(G,depart):
    àTraiter = [depart]
    dejaVus = [false for i in taille(G)]
        tant que (àTraiter!=[])
        sommet = àTraiter[0] ou à Traiter[-1]
        supprimer(àTraiter, sommet)
        traiter sommet
        pour (destination dans adjacent(sommet,G)):
             si destination non dans dejaVus alors
                 àTraiter.append(destination)
                 dejaVus[destination] = True
```

Complexité du parcours

On note $n = |S|$ et $m = |A|$.
Lorsqu'on dispose de la liste des successeurs d'un sommet s, chacun de ces successeurs est rangé dans la pile ou la file, selon qu'il s'agit d'un parcours en profondeur ou en largeur, et chacun sera traité à son tour. Pour chaque sommet, on vérifie s'il a déjà été visité. On ne regarde jamais deux fois la liste des successeurs d'un sommet. Ainsi, pour au plus chaque arête, on doit effectuer $O(1)$ opérations : le parcours, qu'il soit en largeur ou en profondeur est donc en temps $O(m)$.
En mémoire, on doit garder $O(n)$ informations : les sommets déjà traités (dans un tableau de la taille de S) et ceux en liste d'attente (nécessairement moins nombreux que ceux de S).

Parcours en largeur

Dans le cadre du parcours en largeur, ou BFS (*breadth first search*), la liste "d'attente" est gérée suivant le principe d'une file : un sommet inséré avant un autre sera traité en premier. Dès qu'on traite un sommet, on rajoute à la liste des sommets traités tous ceux qui lui sont adjacents. En

Python, si l'ajout d'un sommet se fait en fin de liste, on choisira pour le traiter le sommet en début de liste.
Si on observe la distance entre s et les sommets insérés dans la file, on observe qu'elle est croissante : d'abord s à distance 0 de lui-même, puis ses successeurs, à distance 1, les successeurs de ceux-ci, qui ne seront traités qu'ensuite, etc.

Parcours en profondeur

Le parcours en profondeur, ou DFS (*Depth First Search*) utilise non une file, mais une pile, c'est-à-dire une structure dans laquelle le dernier élément ajouté est celui que l'on enlève. Si on ajoute des éléments en fin de liste, c'est l'élément de fin de liste que l'on traitera en premier.
On peut également utiliser un algorithme récursif (l'utilisation d'une pile est un moyen de simuler plusieurs appels récursifs de fonctions, chaque appel non encore traité s'empilant, jusqu'à ce qu'une valeur soit renvoyée). La structure d'un tel algorithme reprend le principe "si le sommet observé n'a pas encore été traité : le traiter ; indiquer qu'il a été visité ; parcourir récursivement ses successeurs".

```
def parcoursRec(sommet, G, dejaVus):
    si sommet non dans dejaVus:
        dejaVus[sommet] = True
        traiter sommet
        pour chaque dest adjacent à s:
             parcoursRec(dest)
```

La fonction `parcoursRec` est ensuite appelée avec un sommet `sommet`, un graphe `G`, et un tableau `dejaVus` de taille n initialisé avec des `False`.
Une fonction récursive utilisant une **pile des appels** de taille maximale fixée, on peut obtenir une erreur s'il y a trop d'appels récursifs.

Applications

a Composante connexe

Si un graphe est orienté, le parcours en profondeur (comme le parcours en largeur) à partir d'un sommet `s` permettra de visiter tous les sommets accessibles depuis `s`. On peut le montrer par récurrence sur la distance entre s et le sommet cible c : si cette distance vaut 0, c'est vrai, puisque le sommet s, le seul à distance 0 de lui-même est visité au début de l'algorithme Pour une distance égale à 1, on observe que le sommet c est empilé dès traitement de s, donc du sommet à distance 0 qui mène à c, et sera visité lorsqu'il sera dépilé (ce qui arrive forcément, puisqu'un sommet n'est pas dépilé deux fois, qu'on dépile un sommet à chaque tour de boucle "while", et qu'il y a un nombre fini de sommets). Ce raisonnement mène à la compréhension de l'étape d'hérédité : on suppose que tous les sommets à distance $n-1$ sont visités, et que c est à distance n de s, donc qu'il existe au moins un sommet étape e à distance 1 de c et à distance $n-1$ de s. Par hypothèse de récurrence, le sommet e sera visité lors du déroulement de l'algorithme. Si c n'a alors pas encore été visité à ce moment, il sera empilé lors du traitement de e, puis traité lorsqu'il sera dépilé. Si un sommet n'est pas à distance finie de s, il n'existe pas de chemin le reliant à s, et l'algorithme procédant en suivant un certain chemin ne peut y accéder.

Puisque le parcours en profondeur permet **dans un graphe orienté** d'obtenir exactement les sommets accessibles **depuis** un sommet s, il permet également de le faire dans un graphe non orienté, où la relation d'accessibilité est une relation d'équivalence. L'ensemble des sommets traités est donc exactement ceux qui sont accessibles depuis s, c'est-à-dire sa composante connexe.

Le parcours d'un graphe **non orienté** à partir d'un sommet permet donc d'en parcourir la composante connexe contenant ce sommet. Si on veut obtenir toutes les composantes connexes du graphes, il faut itérer sur les sommets non encore parcourus. Si après un parcours à partir d'un unique sommet, on épuise tous les sommets d'un graphe, celui-ci est connexe. Si on modifie le tableau `dejaVus` en indiquant, non si on a vu un sommet ou non, mais la distance au sommet s, on peut calculer la distance à s de tous les sommet de sa composante connexe (cas non orienté), ou de tous les sommets accessibles depuis s (cas orienté).

b Plus court chemin

Un parcours en largeur d'un graphe à partir d'un sommet `depart` permet de trouver le plus court chemin entre ce sommet et un autre : dès que le sommet destination est atteint, on a trouvé un plus court chemin. On peut envisager d'utiliser un tableau `chemin`, tel que `chemin[i]` contient, soit une valeur sentinelle, soit un chemin minimal vers ce sommet. Ce tableau est mis à jour à chaque traitement de sommet. Cet algorithme marche-t-il si on utilise un parcours en profondeur ?

c Détection de cycles - graphe orienté

Le parcours en profondeur à partir de s dans un graphe orienté permet de détecter des cycles contenant un successeur du sommet s, mais nécessite pour cela une adaptation. En effet, retomber sur un sommet déjà vu ne signifie pas nécessairement qu'on a trouvé un cycle. Pour détecter un cycle, on sépare les sommets en trois types : jamais vu ; entièrement traité ; traitement en cours (c'est-à-dire qu'on traite un des sommets accessibles depuis lui-même lorsqu'on retombe sur ce sommet). Dans le troisième cas seulement, un cycle est détecté : le sommet est accessible depuis lui-même.
Pour détecter à coup sûr un cycle dans un graphe orienté, on doit lancer l'algorithme sur chacun de ces sommets, ce qui est coûteux, mais reste polynomial en la taille du graphe.

Exercices

1. Écrire une fonction qui imprime les sommets d'une composante connexe d'un graphe non orienté, présenté sous forme de liste d'adjacence, dans l'ordre du parcours en largeur (puis faire de même avec un parcours en profondeur)
2. Écrire une fonction qui détecte la présence de cycles dans un graphe *à l'aide d'un parcours bien choisi du graphe* et renvoie un tel cycle.
3. Écrire une fonction qui détermine la connexité ou non d'un graphe *à l'aide d'un parcours bien choisi du graphe*. Compter les composantes connexes.
4. Écrire une fonction qui détermine la distance entre deux sommets s et s' d'un graphe G et renvoie un chemin optimal.
5. Écrire une fonction qui détermine si chaque composante est un arbre (on considère ici que x est fils de y s'il existe une arête de x vers y, et aucune de y vers x).

Tri topologique

Le **tri topologique** d'un graphe orienté acyclique consiste à ordonner ses sommets de telle manière que s'il existe un chemin de a vers b dans le graphe, alors $a < b$. Un tel tri peut ne pas être unique ; la relation "il existe un chemin de a vers b" est une relation d'ordre partiel. Le tri topologique prolonge cet ordre en un ordre total.
Ce type de tri intervient par exemple dans des tâches d'ordonnancement : un sommet a correspond à une tâche représentée par le sommet b lorsque $a < b$.
L'algorithme de tri topologique prend, tant que cela est possible, un sommet non déjà traité, puis effectue un parcours en profondeur depuis ce sommet. Lorsque le parcours en profondeur est achevé, le sommet choisi est inséré en tête d'une liste chaînée.

Exercice : Montrer que le tri topologique classe les sommets par ordre croissant. L'implémenter.

Plus court chemin

Dans un graphe non pondéré, un parcours en largeur suffit à déterminer la distance entre deux noeuds, et un plus court chemin de l'un à l'autre. Dans le cadre d'un graphe **pondéré**, dont les arêtes sont porteuses d'un **poids**, nombre réel quelconque, la longueur d'un chemin n'est plus le nombre d'arêtes séparant deux sommets, mais la somme des poids rencontrés sur le chemin. La distance entre deux sommets n'est donc plus nécessairement indiquée par un chemin entre ces deux sommets comportant un nombre minimal d'arêtes. Ce faisant, on permet de modéliser plus fidèlement des situations concrètes comme l'étude de chemin suivant un réseau autoroutier.

On confond dans ce qui suit les sommets avec l'indice qui les désigne.

Définition 5

Une **pondération** est une application $w : E \mapsto \mathbb{R}$. Le **poids** d'une arête e est $w(e)$. On prolonge l'application w sur l'ensemble des couples de S^2 en posant $w(a,b) = w(e)$ s'il existe une arête de a vers tb, et $w(a,b) = +\infty$ sinon.
Le **poids** d'un chemin est la somme des poids des arêtes qui le composent. On note $\delta(a,b)$ le poids du plus court chemin de a vers v s'il existe ; dans le cas contraire, on pose $\delta(a,b) = +\infty$

Un poids étant un nombre réel quelconque, on accepte aussi qu'il puisse être négatif. S'il existe un cycle de poids négatif, on posera $\delta(a,b) = -\infty$ pour chaque couple de sommets de ce cycle.
Il existe trois problèmes de plus courts chemins :

1. calculer le chemin de poids minimal entre une source a et une destination b
2. calculer les chemins de poids minimal entre une source a et tout autre sommet du graphe
3. calculer tous les chemins de poids minimal entre deux sommets quelconques du graphe

Les algorithmes que l'on présente sont basés sur le résultat suivant :

Lemme

(Principe de sous-optimalité) Si un chemin de a vers b est un plus court chemin qui passe par c, alors le sous-chemin qui part de a et arrive en c et celui qui part de c et arrive en b sont également des plus courts chemins.

Algorithme de Floyd-Warshall

La matrice d'adjacence M d'un graphe $G = (S, A)$ pondéré est donnée par $m_{i,j} = w(i,j)$. On observe que dans le cas d'un graphe dont chaque arête est pondérée par 1, on se rapproche de la définition précédemment donnée des matrices d'adjacence. Poser qu'une arête non existante est de poids $+\infty$ permet de la distinguer d'une arête de poids nul.
L'algorithme de Floyd-Warshall consiste à calculer une suite finie récurrente de matrices, dont la $i$ème indique la taille du plus petit chemin entre deux sommets quelconques passant par les i premiers sommets intermédiaires. Ainsi M_0 est la matrice d'adjacence du graphe, telle que présentée ci-dessus, puis, pour tout $k < n$, pour tout couple de sommets (i,j), $m_{i,j}^{(k+1)} = \min(m_{i,j}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)})$

Théorème

Si G ne contient pas de cycle de poids strictement négatif, alors $m_{i,j}^{(k)}$ est égal au poids du chemin minimal reliant les sommets i et j de sommets intermédiaires dans $[\![1, k]\!]$.

Exercice : Démontrer le théorème !
La complexité de l'algorithme de Floyd-Warshall est cubique en l'ordre du graphe. Sa complexité spatiale est quadratique.

Si on souhaite déterminer, non seulement une distance entre les couples de sommets, mais également un chemin minimal, on peut adapter l'algorithme précédent pour retenir les chemins les plus courts à chaque étape.

Algorithme de Dijkstra

L'algorithme de Dijkstra répond au problème des plus courts chemins à partir d'un sommet précis, appelé **source** dans un graphe pondérés avec des poids **positifs**.
Il consiste en la mise à jour d'une partition des sommets et d'un tableau des distance de s aux autres sommets.
Au début de l'algorithme, les sommets sont divisés en deux sous-ensembles. L'ensemble S contient le sommet source ; l'ensemble S' contient les autres sommets. On écrit dans le tableau les distances entre s et chaque sommet, en ne tenant compte que des arêtes entre s et les sommets de S', autrement dit, on ne considère que les sommets adjacents à s, les autres étant placés à distance $+\infty$ de s.
Puis, jusqu'à épuisement des sommets de S', on sélectionne le sommet t qui est le plus proche de s, et il rejoint les sommets de S. On met à jour ensuite les distances des sommets de S' en choisissant la plus petite des distances entre celle qui est enregistrée dans le tableau, et la somme de la distance de s à t et de la distance de t à ses sommets adjacents.

On note $d(u)$ la distance de s à u présente dans le tableau au moment où on l'observe. On rappelle qu'il n'y a pas d'arêtes pondérées négativement.
La correction de l'algorithme vient de ce que lorsqu'un sommet t est choisi dans S' et rejoint S, $d(t) = \delta(s,t)$, et qu'avant que ce soit le cas, $d(t) = \min\limits_{u\in S}(\delta(s+u) + w(u,t))$.

- Lorsque s est placé dans S, $d(s) = \delta(s,s) = 0$. La valeur $d(s)$ n'évolue plus ensuite.
- On se trouve à une étape quelconque de l'algorithme, où un sommet t est choisi dans S' pour rejoindre S. Il faut montrer que $d(t) = \delta(s,t)$. Par minimalité de la plus courte distance, $d(t) \geq \delta(s,t)$. Supposons qu'il existe un autre chemin, plus court. Il passe nécessairement par au moins un sommet v de S' distinct de t, puisque $d(t)$ dépend des chemins trouvés avec pour sommets étapes ceux de S, sans quoi un tel autre chemin aurait déjà permis de modifier $d(t)$. Soit v le premier sommet de S' dans un tel plus court chemin entre s et t. Puisque les poids sont positifs, $\delta(s,t) \geq \delta(s,v)$. Par ailleurs, à cette étape, $d(v) = \min\limits_{u\in S}(\delta(s,u) + w(u,v)) \leq \min\limits_{u\in S}(\delta(s,u) + w(u,t))$, puisque t est choisir à cette étape. On en déduit que $d(t) \leq d(v)$, ce qui permet de montrer que $d(t) = \delta(s,t)$. Les sommets déjà dans S ne voient pas leur valeur modifiée, et elle vaut leur distance à s ; un sommet v qui reste dans S' voit éventuellement sa valeur modifiée à la baisse, et vaut (encore) $\min\limits_{u\in S}(\delta(s,u) + w(u,v))$.

Ainsi, lorsque tous les sommets du graphe appartiennent à S, la valeur dans le tableau indique leur distance à la source.

La complexité de cet algorithme dépend de la manière dont on choisit de représenter les données. La boucle principale comptera n étapes, la première étant triviale, et à chaque étape on doit mettre à jour des distances de sommets, autant que le sommet choisi précédemment a de sommets adjacents, puis il faut faire une recherche d'élément minimal. On peut ainsi envisager un coût quadratique.

Dans le cadre d'un graphe complet, avec le nombre maximal d'arêtes, le coût est exactement quadratique : chaque actualisation doit se faire sur l'ensemble des sommets restants, et la recherche du minimum n'est que du même ordre de grandeur. Mais si le graphe est creux, avec peu d'arêtes, on peut envisager d'utiliser une file de priorité pour limiter l'impact de cette recherche d'élément minimal. Un tas-min permet la récupération de l'élément de priorité minimale en temps logarithmique en fonction de sa taille, qui correspond au nombre de sommets. La mise

à jour du tableau, suite au calcul d'une nouvelle distance temporaire $d(v)$, se fait également en temps $O(\log(n))$. Chaque arête ne donne lieu qu'à une modification dans le tableau et amène donc $O(p \log n)$ opérations. L'ensemble est donc de complexité $O((n+p)\log(n)$.
Il devient plus intéressant d'utiliser une file de priorité, et une structure de tas-min, lorsque $(n+p)\log(n) = O(n^2)$, soit lorsque $p = O(\frac{n^2}{\log(n)})$.
Pour trouver un plus court chemin entre deux sommets a et b, on applique l'algorithme de Dijkstra en partant du sommet a. On peut stopper la recherche dès que le sommet b rentre dans l'ensemble S. Le chemin peut être gardé en mémoire dans un tableau annexe, comme il avait été fait dans le cadre de l'algorithme de Floyd-Warshall : ici, on propose de garder en mémoire en case j le dernier sommet visité lors d'un plus court chemin de la source à j. Le chemin peut ensuite être reconstitué en temps linéaire en la taille du chemin (donc de la taille du graphe).