COURS : ALGORITHME DE DIJKSTRA AVEC LES TAS

L'objectif de ce cours est d'introduire l'algorithme de Dijkstra en utilisant les tas comme structure de données. Cela permettra d'accélérer le temps d'exécution de l'algorithme puisque les tas sont une structure de données possédant une opération de recherche du minimum s'exécutant en O(log n), alors que la recherche d'un minimum avec les listes ou les dictionnaires se fait en O(n).

I) RAPPELS SUR L'ALGORITHME DE DIJKSTRA

L'algorithme de Dijkstra est l'un des algorithmes les plus célèbres en informatique, notamment pour la recherche du plus court chemin dans un graphe pondéré. Cet algorithme permet de trouver le plus court chemin entre un nœud source et tous les autres nœuds d'un graphe, à condition que les poids des arêtes soient positifs. C'est Edsger W. Dijkstra, un informaticien néerlandais, qui a découvert cet algorithme en 1956.

Cet algorithme fonctionne dans tout graphe orienté dont les arêtes ont des longueurs non négatives, et il calcule les longueurs des plus courts chemins à partir d'un sommet de départ vers tous les autres sommets.

Dans ce chapitre, nous allons faire quelques rappels sur la mise en œuvre et les propriétés de cet algorithme lorsqu'il est implémenté avec des structures usuelles. Puis, dans le chapitre suivant, nous verrons une implémentation extrêmement rapide de l'algorithme qui tire parti de la structure de tas.

I.1. Le problème du plus court chemin à source unique

Le problème du plus court chemin à source unique consiste à déterminer, dans un graphe pondéré, les plus courts chemins partant d'un sommet donné vers tous les autres sommets :

Problème : plus court chemin à source unique

Entrée : Un graphe orienté G = (V, E), un sommet de départ s ∈ V, et une longueur non négative ℓ_e pour chaque arête e ∈ E.

Sortie : dist(s, v), pour chaque sommet v ∈ V.

Rappelons que la notation dist(s, v) désigne la longueur d'un plus court chemin de s vers v (s'il n'existe aucun chemin de s vers v, alors dist(s, v) vaut $+\infty$). Par « longueur d'un chemin », on entend la somme des longueurs de ses arêtes. Par exemple, dans un graphe où chaque arête a une longueur égale à 1, la longueur d'un chemin est simplement le nombre d'arêtes qui le composent. Un plus court chemin d'un sommet v vers un sommet w est un chemin de longueur minimale (parmi tous les chemins de v à w).

Par exemple, si le graphe représente un réseau routier et si la longueur de chaque arête représente le temps de trajet attendu d'une extrémité à l'autre, alors le problème du plus court chemin à source unique consiste à calculer les temps de conduite depuis une origine (le sommet de départ) vers toutes les destinations possibles.

Considérons l'entrée suivante du problème du plus court chemin à source unique, avec le sommet de départ s, et où chaque arête est étiquetée par sa longueur :

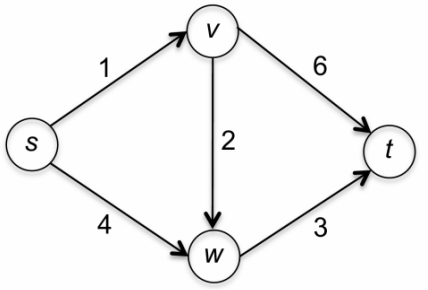


Figure 1: Exemple d'un graph orienté

Dans cet exemple, la distance du plus court chemin de s vers lui-même est 0, et celle de s vers v est 1. Le sommet w est plus intéressant. Un chemin de s à w est l'arête directe (s, w), dont la longueur est 4. Mais utiliser davantage d'arêtes peut réduire la longueur totale : le chemin s $\rightarrow$ v $\rightarrow$ w a une longueur de seulement 1 + 2 = 3, et c'est le plus court chemin de s à w. De même, chacun des chemins à deux étapes de s à t a une longueur de 7, tandis que le chemin « en zigzag » a une longueur de seulement 1 + 2 + 3 = 6.

I.2. Hypothèses

Nous supposons que le graphe d'entrée est orienté. L'algorithme de Dijkstra s'applique cependant tout aussi bien aux graphes non orientés, après quelques modifications.

Nous supposons également que la longueur de chaque arête est non négative. Dans de nombreuses applications, comme le calcul d'itinéraires routiers, les longueurs des arêtes sont naturellement non négatives, et il n'y a donc rien à craindre.

Mais les chemins dans un graphe peuvent représenter des séquences abstraites de décisions. Par exemple, on pourrait vouloir calculer une suite d'opérations financières rentables, impliquant à la fois des achats et des ventes. Ce problème correspond à la recherche d'un plus court chemin dans un graphe dont certaines arêtes ont des longueurs positives et d'autres négatives. On ne peut pas utiliser l'algorithme de Dijkstra dans des applications comportant des longueurs d'arêtes négatives.

Cependant, nous aborderons cette année (dans le thème traitant de la programmation dynamique) des algorithmes efficaces pour le problème plus général du plus court chemin dans lequel les longueurs d'arêtes négatives sont autorisées. Nous étudierons notamment l'algorithme de Bellman-Ford pour les chemins à source unique, ainsi qu'une version plus générale pour les chemins à sources multiples, l'algorithme de Floyd-Warshall.

I.3. Pourquoi ne pas utiliser l'algorithme de recherche en largeur (BFS) ?

Vous avez vu en première année que l'une des applications phares de la recherche en largeur dans les graphes (Breadth-First Search) est le calcul des distances de plus court chemin à partir d'un sommet de départ. Alors, pourquoi avons-nous besoin d'un autre algorithme de plus court chemin ?

L'algorithme de recherche en largeur calcule le nombre minimal d'arêtes dans un chemin allant du sommet de départ à chaque autre sommet. Il s'agit d'un cas particulier du problème du plus court chemin à source unique, dans lequel chaque arête a une longueur égale à 1. Or, nous avons vu dans l'exemple précédent que lorsque les longueurs des arêtes sont généralement non négatives, un plus court chemin n'est pas nécessairement celui comportant le plus petit nombre d'arêtes.

De nombreuses applications du calcul de plus courts chemins, comme le calcul d'itinéraires routiers ou la planification d'une suite de transactions financières, impliquent inévitablement des arêtes de longueurs différentes.

Mais peut-être êtes-vous en train de vous demander : ne pourrions-nous pas simplement considérer une arête plus longue comme un chemin composé de plusieurs arêtes, chacune de longueur 1, comme l'illustre le schéma ci-dessous :

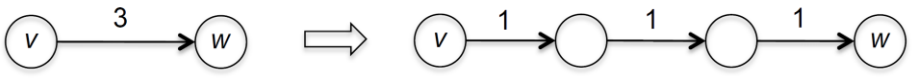


Figure 2 : Transformation d'un chemin de valeur non unitaire en multiples arêtes unitaires

En principe, on peut résoudre le problème du plus court chemin à source unique en remplaçant chaque arête par un chemin d'arêtes de longueur 1, puis en appliquant la recherche en largeur (BFS) sur le graphe ainsi étendu. C'est un exemple de réduction d'un problème à un autre.

Un problème A se réduit à un problème B si un algorithme qui résout B peut être facilement transformé en un algorithme qui résout A. Les réductions sont importantes dans l'étude des algorithmes et de leurs limites, et elles peuvent également avoir une grande utilité pratique. Par exemple, le problème du calcul de la médiane d'un tableau se réduit au problème du tri de ce tableau.

Le principal problème de la réduction proposée précédemment est qu'elle fait exploser la taille du graphe. Cette explosion n'est pas trop gênante si toutes les longueurs des arêtes sont de petits entiers, mais ce n'est pas toujours le cas dans les applications. La longueur d'une arête peut même être beaucoup plus grande que le nombre total de sommets et d'arêtes du graphe d'origine !

La recherche en largeur s'exécuterait en un temps linéaire par rapport à la taille du graphe étendu, mais ce temps n'est pas forcément proche du temps linéaire par rapport à la taille du graphe original.

L'algorithme de Dijkstra peut être vu comme une simulation astucieuse de la recherche en largeur sur le graphe étendu, tout en ne travaillant que sur le graphe original et en s'exécutant en temps quasi linéaire.

I.4. Pseudocode de l'algorithme de Dijkstra

La structure générale de l'algorithme de Dijkstra ressemble à celle des algorithmes de parcours de graphe. À chaque itération de sa boucle principale, un nouveau sommet est traité. La sophistication de l'algorithme réside dans sa règle ingénieuse pour choisir le prochain sommet à traiter : il sélectionne le sommet encore non traité qui semble le plus proche du sommet de départ.

Dijkstra

Entrée : Un graphe orienté G = (V, E), un sommet de départ s ∈ V, une longueur non négative ℓ_e pour chaque arête e ∈ E.

Remarque : len(x) représente la longueur de l'arête (s,x)

```
    // Initialisation
1   X := {s}
2   len(s) :=0, len(v) := +∝ pour tous les v ≠ s

    // Boucle principale
3   TANT QUE il existe une arête (v, w) avec v ∈ X, w ∉ X alors :
4           (v*, w*) := l'arête qui minimise len(v) + ℓ_{v,w}
5           ajouter w* à X
6           len(w*) := len(v*) + ℓ_{v*,w*}
```

L'ensemble X contient les sommets que l'algorithme a déjà traités. Au départ, X ne contient que le sommet initial (et bien sûr, len(s) = 0), puis cet ensemble s'étend progressivement, jusqu'à recouvrir tous les sommets accessibles depuis s.

L'algorithme attribue une valeur finie à len(v) (la longueur estimée du plus court chemin vers v) au moment même où il ajoute le sommet à X. Chaque itération de la boucle principale agrandit X d'un nouveau sommet, la tête d'une certaine arête (v, w) qui relie X à {V – X} (s'il n'existe aucune telle arête, l'algorithme s'arête, en laissant len(v) = $+\infty$ pour tout v $\notin$ X).

Il peut y avoir plusieurs arêtes de ce type ; l'algorithme de Dijkstra en choisit une seule (notée (v^*, w^*)) qui minimise le score de Dijkstra, défini par : len(v) + $\ell_{v,w}$
Remarque : les scores de Dijkstra sont définis sur les arêtes et un sommet w $\notin$ X peut être la tête de plusieurs arêtes différentes reliant X à {V – X}. Ces arêtes auront généralement des scores de Dijkstra différents.

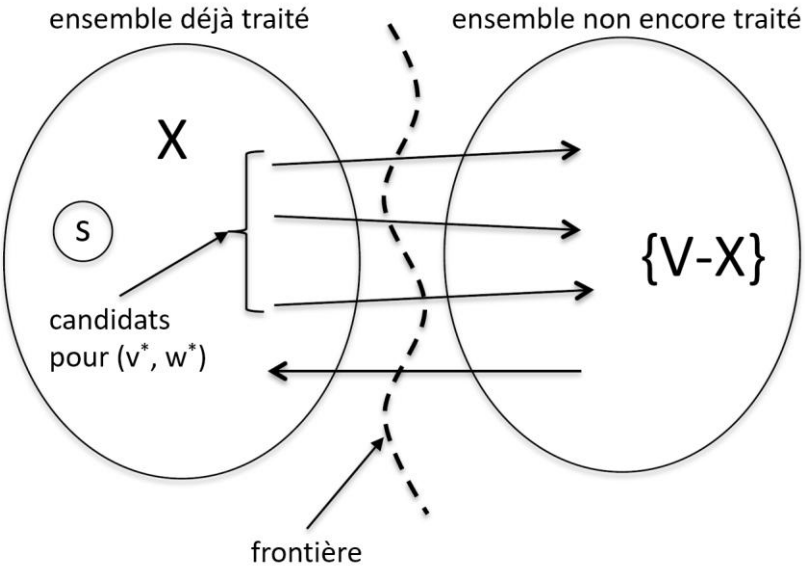


Figure 3 : Chaque itération de l'algorithme de Dijkstra traite un nouveau sommet, la tête d'une arête reliant X à V – X.

On peut associer le score de Dijkstra d'une arête (v, w), avec v $\in$ X et w $\notin$ X, au fait que le plus court chemin de s à w est constitué d'un plus court chemin de s à v, suivi de l'arête (v, w) de longueur $\ell_{v,w}$.

Ainsi, l'algorithme de Dijkstra choisit d'ajouter le sommet encore non traité qui semble le plus proche de s, en se basant sur les distances de plus court chemin déjà calculées et sur les longueurs des arêtes reliant X à {V – X}. Lorsqu'il ajoute w^* à X, l'algorithme attribue à len(w^*) sa distance supposée de plus court chemin depuis s, c'est-à-dire le score de Dijkstra de l'arête (v^*, w^*), soit len(v^*) + ℓ_{v^*,w^*}.

I.5. Exemple

Prenons l'exemple déjà étudié en introduction de ce chapitre :

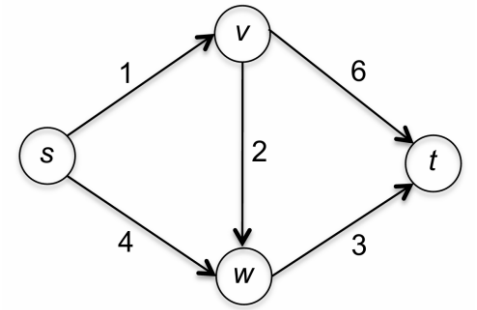


Figure 4: Exemple d'un graph orienté

Au départ, l'ensemble X contient uniquement s, et len(s) = 0.

Lors de la première itération de la boucle principale, il y a deux arêtes reliant X à {V – X} (et qui peuvent donc jouer le rôle de (v^*, w^*)) : les arêtes (s, v) et (s, w). Les scores de Dijkstra pour ces deux arêtes sont :

- $len(s) + \ell_{s,v} = 0 + 1 = 1$
- $len(s) + \ell_{s,w} = 0 + 4 = 4$

Comme la première arête possède le score le plus faible, sa tête v est ajoutée à X, et len(v) reçoit la valeur du score de Dijkstra de l'arête (s, v), soit 1.

Lors de la deuxième itération, avec X = {s, v}, il y a trois arêtes à considérer pour le rôle de (v^*, w^*) : (s, w), (v, w) et (v, t). Leurs scores de Dijkstra respectifs sont :

- $len(s) + \ell_{s,w} = 0 + 4 = 4$
- $len(v) + \ell_{v,w} = 1 + 2 = 3$
- $len(v) + \ell_{v,t} = 1 + 6 = 7$

Comme (v, w) a le score le plus faible, w est ajouté à X, et len(w) reçoit la valeur 3 (le score de Dijkstra de (v, w)).

Lors de la troisième itération, avec X = {s, v, w}, il y a deux arêtes à considérer pour le rôle de (w*, t*) : (v, t) et (w, t). Leurs scores de Dijkstra respectifs sont :

- $len(v) + \ell_{v,t} = 1 + 6 = 7$
- $len(w) + \ell_{w,t} = 3 + 3 = 6$

Comme il ne reste plus que le sommet t, il est ajouté à X, et len(t) est fixé à la plus petite valeur, soit 6. L'ensemble X contient maintenant tous les sommets, donc aucune arête ne relie plus X à V − X, et l'algorithme s'arrête.

Les valeurs finales sont : len(s) = 0, len(v) = 1, len(w) = 3, et len(t) = 6. Elles correspondent exactement aux véritables distances de plus court chemin que nous avions identifiées dans l'exemple d'introduction.

I.6. Graphes avec des chemins de longueurs négatives

L'algorithme de Dijkstra ne garantit pas toujours le calcul exact des distances de plus court chemin lorsque les arêtes peuvent avoir des longueurs négatives.

Vous vous demandez peut-être pourquoi il est si important que les arêtes aient ou non des longueurs négatives. Ne pourrait-on pas simplement rendre toutes les longueurs d'arêtes non négatives en ajoutant une grande valeur à chacune d'elles ?

Hélas, il est impossible de réduire le problème du plus court chemin à source unique avec des longueurs d'arêtes quelconques au cas particulier des longueurs non négatives de cette manière. Le problème vient du fait que différents chemins entre deux sommets peuvent ne pas avoir le même nombre d'arêtes.

Si l'on ajoute une même constante à la longueur de chaque arête, alors les longueurs totales des chemins augmentent de façons différentes selon le nombre d'arêtes qu'ils contiennent, et le plus court chemin dans le nouveau graphe pourrait être différent de celui du graphe original.

Voici un exemple simple :

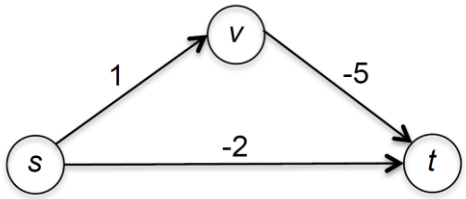


Figure 5 : Exemple de graphe avec des longueurs négatives

Il existe deux chemins allant de s à t : le chemin direct, dont la longueur est −2, et le chemin en deux étapes s $\rightarrow$ v $\rightarrow$ t, dont la longueur est 1 + (−5) = −4. Le second chemin a une longueur plus petite et constitue donc le plus court chemin de s à t. Pour obliger le graphe à n'avoir que des longueurs d'arêtes non négatives, on pourrait ajouter 5 à la longueur de chaque arête :

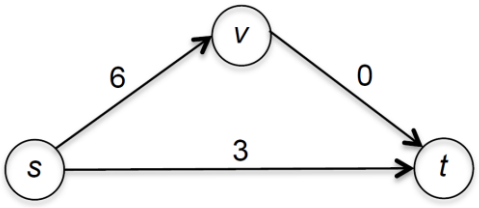


Figure 6 : Graphe corrigé

Le plus court chemin de s à t a changé : c'est maintenant l'arête directe s$\rightarrow$t (dont la longueur est 3, meilleure que l'autre option, 6). Ainsi, exécuter un algorithme de plus court chemin sur le graphe transformé ne donnerait pas la bonne réponse pour le graphe original.

Que se passe-t-il si l'on exécute directement l'algorithme de Dijkstra sur un graphe contenant des arêtes de longueur négative, comme celui ci-dessus ?

Comme toujours, au départ X = {s} et len(s) = 0, ce qui est parfaitement normal.

Cependant, lors de la première itération de la boucle principale, l'algorithme calcule les scores de Dijkstra des arêtes (s, v) et (s, t), et comme la seconde arête a le score le plus faible, l'algorithme ajoute le sommet t à X et assigne len(t) = -2.

Or, comme nous l'avons déjà remarqué, le véritable plus court chemin de s à t a une longueur de −4.

I.7. Implémentation et temps d'exécution

L'algorithme de plus court chemin de Dijkstra rappelle les algorithmes de parcours de graphe en profondeur (DFS) et en largeur (BFS) en temps linéaire. Une raison essentielle pour laquelle ces parcours s'exécutent en temps linéaire est qu'ils ne passent qu'un temps constant à décider quel sommet explorer ensuite, en retirant simplement un sommet du début d'une file (BFS) ou du haut d'une pile (DFS). Or, dans l'algorithme de Dijkstra, chaque itération doit identifier l'arête traversant la frontière ayant le plus petit score de Dijkstra.

Supposons que les symboles n et m désignent respectivement le nombre de sommets et le nombre d'arêtes du graphe d'entrée. Dans une implémentation simple, à chaque itération l'algorithme effectue une recherche exhaustive sur toutes les arêtes, calcule le score de Dijkstra pour chaque arête dont la queue est dans X et la tête est hors de X (ce calcul prenant un temps constant par arête), puis sélectionne l'arête franchissant la frontière ayant le plus petit score (ou détermine correctement qu'il n'existe plus d'arêtes franchissant la frontière). Pour savoir quels sommets appartiennent à X, l'algorithme leur associe une variable booléenne.

Après au plus n - 1 itérations, l'algorithme de Dijkstra n'a plus de nouveaux sommets à ajouter à son ensemble X. Comme le nombre d'itérations est O(n) et que chacune prend un temps O(m), le temps d'exécution total de cette version est O(mn).

Le temps d'exécution de l'implémentation simple est bon, mais pas exceptionnel. Elle fonctionnerait correctement pour des graphes comportant quelques centaines ou quelques milliers de sommets, mais échouerait face à des graphes beaucoup plus grands. Le « graal » en conception d'algorithmes est d'obtenir un algorithme en temps linéaire (ou s'en approchant), et c'est précisément ce que nous souhaitons pour le problème du plus court chemin à source unique. Un tel algorithme pourrait traiter des graphes contenant des millions de sommets sur un ordinateur portable ordinaire.

Nous n'avons pas besoin d'un nouvel algorithme pour obtenir une solution en temps quasi linéaire à ce problème. Il nous faut simplement une meilleure implémentation de l'algorithme de Dijkstra.

Les structures de données (comme les files et les piles) ont joué un rôle crucial dans les implémentations en temps linéaire des algorithmes de parcours en largeur et en profondeur.

De façon analogue, l'algorithme de Dijkstra peut être implémenté en temps quasi linéaire à condition d'utiliser la bonne structure de données, capable de faciliter les calculs répétés de minimums effectués dans sa boucle principale. Vous connaissez maintenant cette structure de données : le tas.

II) L'ALGORITHME DE DIJKSTRA RAPIDE

Nous avons vu que l'implémentation simple de l'algorithme de Dijkstra nécessite un temps O(mn), où m est le nombre d'arêtes et n le nombre de sommets. C'est suffisamment rapide pour traiter des graphes de taille moyenne (comportant quelques milliers de sommets et d'arêtes), mais pas assez pour des graphes de grande taille (comportant des millions de sommets et d'arêtes).

Les tas (heaps) permettent une implémentation extrêmement rapide, en temps quasi linéaire, de l'algorithme de Dijkstra. En effet, l'implémentation de l'algorithme de Dijkstra basée sur un tas s'exécute en O((m + n) log n).
Bien que pas tout à fait aussi rapide que les algorithmes de parcours de graphe en temps linéaire, un temps d'exécution de O((m + n) log n) reste excellent, comparable à celui des meilleurs algorithmes de tri, et suffisamment efficace pour être considéré comme une opération élémentaire « gratuite ».

À chaque itération de la boucle principale, l'algorithme de Dijkstra réalise un calcul de minimum sur les scores de Dijkstra des arêtes franchissant la frontière.

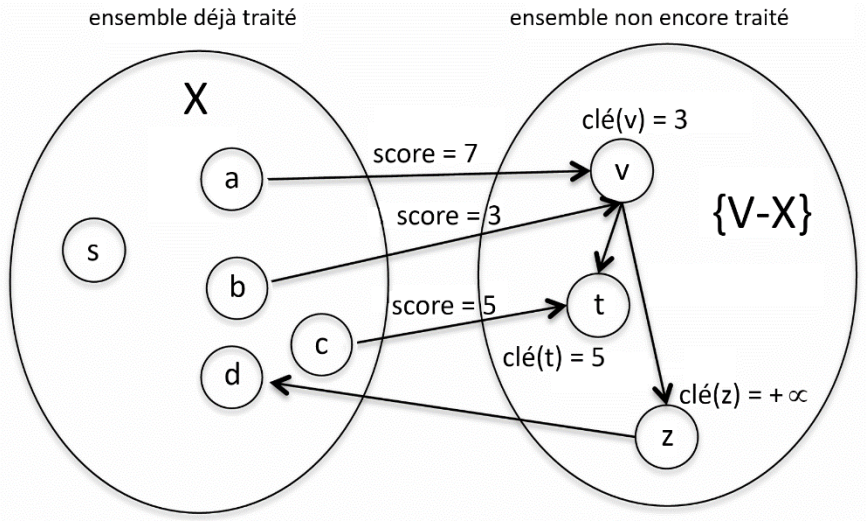
L'implémentation simple effectue ces calculs de minimum par recherche exhaustive. Or, le but même des tas est de remplacer les recherches linéaires par des recherches logarithmiques pour ce type de calculs.

II.1. Principe de fonctionnement

L'idée consiste à stocker les sommets encore non traités (ceux de {V – X}) dans un tas, tout en leur attribuant la clé qui remplit la condition suivante :

Pour un graphe G = (V, E), la clé d'un sommet w ∈ {V - X} est le plus petit score de Dijkstra parmi les arêtes ayant leur queue v ∈ X, ou $+\infty$ s'il n'existe aucune arête de ce type :

$$\text{clé}(w) = \min_{(v,w)\in E,\ v\in X} \text{len}(v) + \ell_{v,w}$$



Pour identifier l'arête (v, w), avec $v \in X$ et $w \notin X$, ayant le plus petit score de Dijkstra, on utilise la procédure suivante :

1. Dans un premier temps, pour chaque sommet $w \in \{V - X\}$, on cherche l'arête (v, w) dont la queue $v \in X$ et la tête est w qui a le plus petit score de Dijkstra (s'il en existe une). Dans l'exemple ci-dessus, on obtiendrait les arêtes : (b, v), (c, t) et (d, z).
2. Ensuite, on sélectionne celle qui possède le plus petit score de Dijkstra. Dans l'exemple, on obtiendrait (b, v) = 3.

C'est exactement l'arête que l'on aurait trouvée avec une recherche exhaustive.

La valeur de la clé d'un sommet $w \in \{V - X\}$ correspond exactement au score gagnant de la première étape. Ainsi, on modélise implicitement toutes les compétitions de la première étape.

Ensuite, extraire le sommet ayant la clé minimale revient à exécuter la seconde étape : on obtient alors le prochain sommet à traiter, c'est-à-dire la tête de l'arête traversant la frontière avec le plus petit score de Dijkstra.

Voici une première ébauche de l'algorithme qui prend en compte la deuxième étape, qui consiste à extraire le sommet ayant la clé minimale :

Dijkstra avec les tas (première partie de l'algorithme)

Entrée : Un graphe orienté G = (V, E), un sommet de départ $s \in V$, une longueur non négative ℓ_e pour chaque arête $e \in E$.

Remarque : len(x) représente la longueur de l'arête (s,x) ; H est un tas

```
    // Initialisation
1   X := ∅ ; H := ∅
2   clé(s) := 0
3   POUR chaque v ≠ s :
4           clé(v) := +∞
5   POUR chaque v ∈ V :
6           Insérer v dans le tas H                 // Ou utiliser l'insertion multiple

    // Boucle principale
7   TANT QUE H est non vide alors :
8           w* := Extraction du minimum (H)         // La première itération garantit
9           ajouter w* à X                          // l'extraction du vertex de départ
10          len(w*) := clé(w*)
            // Mise à jour du tas en maintenant la condition sur la clé
11          (à suivre…)
```

Il reste maintenant à étudier comment implémenter l'enregistrement de la clé en respectant la condition décrite précédemment.

II.2. Enregistrement de la clé

Il faut maintenant expliquer comment maintenir l'enregistrement de la clé en respectant la condition sans effectuer un travail excessif.

À chaque itération de l'algorithme, un sommet v passe de {V - X} à X, ce qui modifie la frontière. Les arêtes partant de sommets de X vers v sont aspirées dans X et ne traversent plus la frontière. Plus problématiques sont les arêtes partant de v vers d'autres sommets de {V - X} : elles ne se trouvent plus entièrement dans {V - X}, mais deviennent désormais des arêtes franchissant la frontière entre X et {V - X} :

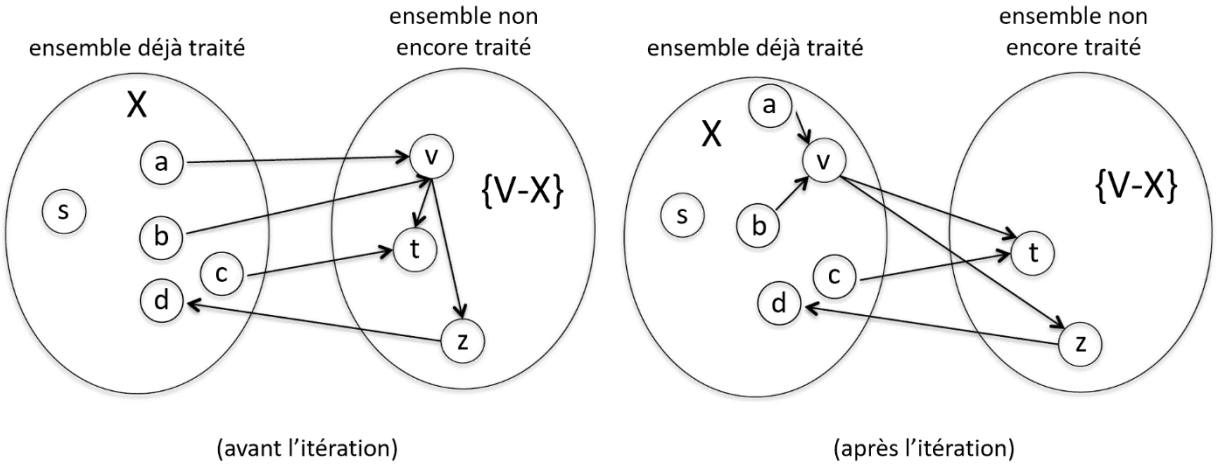


Figure 7 : Illustration de l'évolution des ensembles X et {V - X} après une itération de l'algorithme

C'est un problème parce que la condition sur les clés exige que, pour chaque sommet w ∈ {V - X}, la clé de w soit le plus petit score de Dijkstra parmi les arêtes franchissant la frontière et aboutissant en w. Or, l'apparition de nouvelles arêtes franchissant la frontière introduit de nouveaux candidats pour ce plus petit score car dans la relation permettant de calculer la clé du sommet w :

$$\text{clé}(w) = \min_{(v,w)\in E,\, v\in X} \text{len}(v) + \ell_{v,w}$$

… la valeur $\ell_{v,w}$ peut avoir diminuée pour certains sommets w.

Chaque fois qu'un sommet w^* est extrait du tas, c'est-à-dire qu'il est déplacé de {V - X} vers X, il se peut que la clé de certains sommets de {V - X} doit être changée afin de prendre en compte les nouvelles arêtes franchissant la frontière.

Comme toutes les nouvelles arêtes franchissant la frontière partent de w^*, il suffit de parcourir la liste des arêtes sortantes de w^* et de vérifier les sommets $y \in \{V - X\}$ reliés par une arête (w^*, y).

Pour chaque sommet y de ce type, il existe alors deux candidats au rôle de vainqueur de la première étape de sélection :

- Soit le même sommet qu'avant,
- Soit le nouveau participant, c'est-à-dire l'arête (w^*, y)

Ainsi, la nouvelle valeur de la clé de y doit être le minimum entre son ancienne valeur, et le score de Dijkstra de la nouvelle arête qui franchie la frontière, soit $len(w^*) + \ell_{w^*,y}$.

Une méthode simple pour diminuer la valeur de la clé d'un élément dans le tas consiste à le supprimer à l'aide de l'opération de suppression, à mettre à jour sa clé, puis à le réinsérer dans le tas avec l'opération d'insertion.

Cela achève l'implémentation de l'algorithme de Dijkstra basée sur un tas :

Dijkstra avec les tas (algorithme complet)

Entrée : Un graphe orienté G = (V, E), un sommet de départ $s \in V$, une longueur non négative ℓ_e pour chaque arête $e \in E$.

Remarque : len(x) représente la longueur de l'arête (s,x) ; H est un tas

```
    // Initialisation
1   X := ∅ ; H := ∅
2   clé(s) := 0
3   POUR chaque v ≠ s :
4           clé(v) := +∞
5   POUR chaque v ∈ V :
6           Insérer v dans le tas H                 // Ou utiliser l'insertion multiple

    // Boucle principale
7   TANT QUE H est non vide alors :
8           w* := Extraction du minimum (H)         // La première itération garantit
9           ajouter w* à X                          // l'extraction du vertex de départ
10          len(w*) := clé(w*)

            // Mise à jour du tas en maintenant la condition sur la clé
11          POUR chaque arête (w*, y) :
12                  supprimer l'élément y du tas H
13                  clé(y) := min {clé(y), len(w*) + ℓ_{w*,y}}
14                  Insérer y dans le tas H
```

II.3. Temps d'exécution

La quasi-totalité du travail effectué par l'implémentation de Dijkstra basée sur un tas consiste en des opérations sur le tas. Chacune de ces opérations prend un temps O(log n), où n est le nombre de sommets (le tas ne contient jamais plus de n − 1 éléments).

Il y a (n – 1) opérations à chacune des lignes 6 et 8 : une par sommet autre que le sommet de départ s.

Concernant les lignes 12 et 14, au cours d'une seule itération de la boucle principale, ces deux lignes peuvent être exécutées jusqu'à (n – 1) fois : une par arête sortante du sommet w^*. Comme il y a (n – 1) itérations, cela semble conduire à un nombre quadratique d'opérations sur le tas.

Cette borne est correcte pour les graphes denses, mais en général, on peut faire mieux.

Attribuons la responsabilité de ces opérations sur le tas non plus aux sommets, mais aux arêtes. Chaque arête (v, w) du graphe n'apparaît au plus qu'une seule fois à la ligne 11, lorsque v est extrait du tas et déplacé de (V – X) vers X. Ainsi, les lignes 12 et 14 sont chacune exécutées au plus une fois par arête, soit un total de 2m opérations, où m est le nombre d'arêtes du graphe.

Cela montre que l'implémentation de l'algorithme de Dijkstra basée sur un tas effectue O(m + n) opérations sur le tas, chacune prenant un temps O(log n). Ainsi, le temps d'exécution total est de O((m + n) log n).