L'algorithme de Dijkstra en partant de zéro

Youssef Benjelloun

21 février 2022

Table des matières

Introduction

En théorie des graphes, l'algorithme de Dijkstra sert à résoudre le problème du plus court chemin. Il permet, par exemple, de déterminer un plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région. Plus précisément, il calcule des plus courts chemins à partir d'une source vers tous les autres sommets dans un graphe orienté pondéré par des réels positifs. On peut aussi l'utiliser pour calculer un plus court chemin entre un sommet de départ et un sommet d'arrivée.

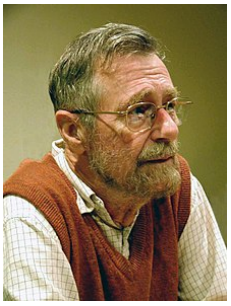

FIGURE 1 – Edsger Wybe DIJKSTRA

1 Biographie

Edsger Wybe DIJKSTRA né à Rotterdam le 11 mai 1930 et mort Nuenen le 6 août 2002, est un mathématicien et informaticien néerlandais du XX^{e} siécle. Il reçoit en 1972 le prix Turing pour ses contributions sur la science et l'art des langages de programmation et au langage Algol. Juste avant sa mort, en 2002, il reçoit le prix PoDC de l'article influent, pour ses travaux sur l'autostabilisation. L'année suivant sa mort, le prix sera renommé en son honneur prix Dijkstra.

Dijkstra avait une écriture manuscrite très lisible et a toujours refusé d'utiliser un traitement de texte, malgré son domaine d'activité, préférant la lettre manuscrite photocopiée. Luca CARDELLI a créé une fonte « Dijkstra » en son honneur, qui imite son écriture règuliére. Dijkstra référençait toutes ses lettres par EWD suivi d'un nombre, la dernière étant la lettre EWD 1318. Pour aller plus loin sur la vie de DIJKSTRA [5].

2 Théorie des Graphes

2.1 Introduction

La théorie des graphes s'est développée au cours du XX^{e} siécle. La génèse de la théorie des graphes semble être une étude de Léonard EULER, un très célèbre mathématicien du XVIII^{e}. Dans un article publié en 1736, il traite un problème devenu classique, illustré par la devinette : peut on faire une promenade passant une fois par chacun des sept ponts de la ville de Koenigsberg ? Regardez la figure 2.

FIGURE 2 – Koenigsberg

2.2 Les bases pour comprendre algorithme de Dijkstra

Définition 2.1. Un graphe $\Gamma(S, A)$ est la donnée de deux ensembles finis : un ensemble de sommets S et un ensemble d'arêtes A. Une arête est une paire de sommets, ce sont les extrémités de l'arête. Une arête $\{ x, y \}$ est notée xy, et les sommets sont dits adjacents.

Définition 2.2. Un chemin de longueur n est une suite de sommets $x_0, x_1, \ldots, x_n$ tels que :

$$\forall i,\, 0 \leq i < n \implies x_i, x_{i+1} \in A$$

Les sommets x_0 et x_n sont les extrémités du chemin, x_0 est l'origine et x_n le sommet terminal. On parle de cycle quand $x_0 = x_n$. On dit que x est connecté à y, et on note $x \to y$ quand il existe un chemin d'extrémité x et y. Il s'agit là d'une relation symétrique et transitive qu'il convient de prolonger par réflexivité. Un chemin est dit simple quand il ne passe jamais plus d'une fois par une même arête. Un chemin élémentaire ne passe pas deux fois par un même sommet.

Définition 2.3. L'ensemble des sommets voisins d'un sommet x :

$$\text{voisin}(x) = \{ y \in S \mid xy \in A \}, \quad \deg(x) = \#\text{voisin}(x)$$

le degré d'un sommet est égal au nombre d'arêtes incidentes. Un sommet de degré pair est dit pair, un sommet de degré impair est dit impair.

Nous avons l'amusante relation des paires et de l'impair

Lemme 2.4. *(parité et impairs). Dans un graphe, $\Gamma(S, A)$:*

$$\sum_{x \in S} deg(x) = 2|A|$$

en particulier, le nombre des sommets impairs est toujours pair.

Démonstration. Notons A_s les arêtes incidentes au sommet s,

$$A = \bigcup_{s \in S} A(s).$$

Une triple intersection des A_s est vide, une double intersection non vide est une arête du graphe. Le principe d'inclusion et d'exclusion s'applique sans difficulté et donne :

$$|A| = \sum_{s \in S} \deg(s) - |A|$$

□

Exercice 2.1. Prouver qu'un graphe d'ordre supérieur à 1, possède deux sommets de degré identique.

Exercice 2.2. De tout parcours, on peut extraire un parcours élémentaire ayant les mêmes extrémités. Détaillez cette affirmation !

Exercice 2.3. Dans un graphe acyclique ayant au moins une arête, il existe un sommet de degré 1. Expliquez !

Pour creuser plus sur l'intéresant théorie des graphes [3].

3 L'algorithme de Dijkstra

3.1 Le principe

L'algorithme prend en entrée un graphe pondéré par des réels positifs et un sommet source. Il s'agit de construire progressivement un sous-graphe dans lequel sont classés les différents sommets par ordre croissant de leur distance minimale au sommet de départ. La distance correspond à la somme des poids des arcs empruntés.

Au départ, on considère que les distances de chaque sommet au sommet de départ sont infinies, sauf pour le sommet de départ pour lequel la distance est nulle. Le sous-graphe de départ est l'ensemble vide.

Au cours de chaque itération, on choisit en dehors du sous-graphe un sommet de distance minimale et on l'ajoute au sous-graphe. Ensuite, on met à jour les distances des sommets voisins de celui ajouté. La mise à jour s'opère comme suit : la nouvelle distance du sommet voisin est le minimum entre la distance existante et celle obtenue en ajoutant le poids de l'arc entre sommet voisin et sommet ajouté à la distance du sommet ajouté. On continue ainsi jusqu'à épuisement des sommets (ou jusqu'à sélection du sommet d'arrivée).

3.2 Schéma de l'algorithme

Le graphe est noté $G = (S, A)$ où :
- l'ensemble S est l'ensemble fini des sommets du graphe G ;
- l'ensemble A est l'ensemble des arcs de G tel que : si (s_1, s_2) est dans A, alors il existe un arc depuis le nœud s_1 vers le nœud s_2 ;
- on définit la fonction *poids* définie sur $S \times S$ dans $\mathbb{R}^+ \cup \{+\infty\}$ qui à un couple (s_1, s_2) associe le poids positif $poids(s_1, s_2)$ de l'arc reliant s_1 à s_2 (et $+\infty$ s'il n'y a pas d'arc reliant s_1 à s_2).

Le poids du chemin entre deux sommets est la somme des poids des arcs qui le composent. Pour une paire donnée de sommets s_{deb} (le sommet du départ) s_{fin} (le sommet d'arrivée) appartenant à S, l'algorithme trouve un chemin depuis s_{deb} vers s_{fin} de moindre poids (autrement dit un chemin le plus léger ou encore le plus court).

L'algorithme fonctionne en construisant un sous-graphe P de manière que la distance entre un sommet s de P depuis s_{deb} soit connue et soit un minimum dans G. Initialement, P contient simplement le nœud s_{deb} isolé, et la distance de s_{deb} à lui-même vaut zéro. Des arcs sont ajoutés à P à chaque étape :

1. en identifiant tous les arcs $a_i = (s_{i1}, s_{i2})$ dans $P \times G$;
2. en choisissant l'arc $a_j = (s_{j1}, s_{j2})$ dans $P \times G$ qui donne la distance minimum depuis S_{deb} à s_{j2} en passant tous les chemins créés menant à ce nœud.

L'algorithme se termine soit quand P devient un arbre couvrant de G, soit quand tous les nœuds d'intérêt sont dans P. Information tiré de [4].

3.3 L'algorithme de Dijkstra en C++

On voit à la figure 3 une implementation du code en C++

```cpp
#include <iostream>
using namespace std;
#include <limits.h>

// Nombre des sommets
#define V 9

// Renvoi la distance minimal
int minDistance(int dist[], bool sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;
    }
    return min_index;
}

void dijkstra(int graph[V][V], int src)
{
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX, sptSet[i] = false;
    }
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;
        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
        }
    }
}
```

FIGURE 3 – L'algorithme de DIJKSTRA en C++

3.4 Exemple graphique

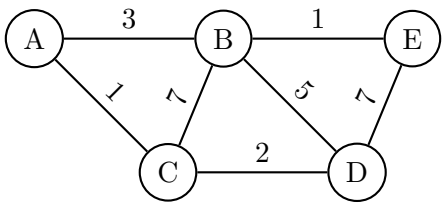

FIGURE 4 – Graphe d'exemple

Supposons le graphe pondéré de la figure 4.

Pendant l'exécution de l'algorithme, nous allons marquer chaque nœud avec sa distance minimale au nœud C (notre nœud sélectionné). Pour le noeud C, cette distance est 0. Pour le reste des noeuds, comme nous ne connaissons pas encore cette distance minimale, elle commence à l'infinie (∞). Nous aurons également un noeud courant. Initialement, nous le définissons sur C (notre nœud sélectionné). Dans l'image, nous marquons le nœud actuel avec un point rouge. Regardez la figure 5.

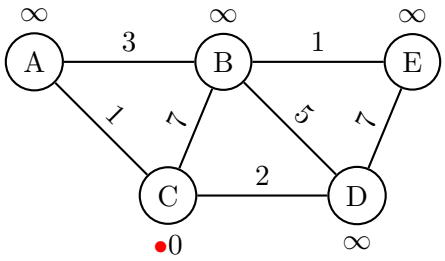

FIGURE 5 – Etape 1

Maintenant, vérifions les voisins de notre nœud actuel (A, B et D) sans ordre particulier. Commençons par B. Nous ajoutons la distance minimale du nœud actuel (dans ce cas, 0) au poids de l'arête qui relie notre nœud actuel à B (dans ce cas, 7), et nous obtenons $0 + 7 = 7$. Nous comparons cette valeur avec la distance minimale de B (infini) ; la valeur la plus petie est celle qui reste comme distance minimale de B (dans ce cas, 7 est plus pétit que l'infini) et ainsi de suite pour les autres voisins.

Nous avons vérifié tous les voisins de C. Pour cette raison, on le marque comme visité.

Nous devons maintenant choisir un nouveau noeud courant. Ce noeud doit être le noeud non visité avec la plus petite distance minimale (donc, le noeud avec le plus petit nombre et sans marque de contrôle). C'est A. Marquons-le d'un point rouge. Regardez la figure 6.

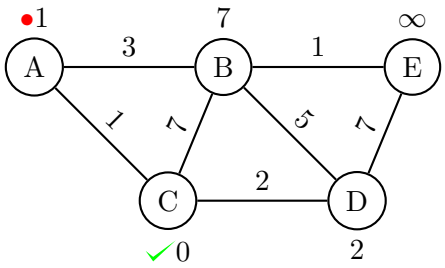

FIGURE 6 – Etape 2

Et maintenant nous répétons l'algorithme. Nous vérifions les voisins de notre nœud actuel, en ignorant les nœuds visités. Cela signifie que nous ne vérifions que B.

Pour B, nous ajoutons 1 (la distance minimale de A, notre noeud actuel) à 3 (le poids

de l'arête reliant A et B) pour obtenir 4. Nous comparons ce 4 avec la distance minimale de B (7) et nous laissons la plus petite valeur : 4. Regardez la figure 7.

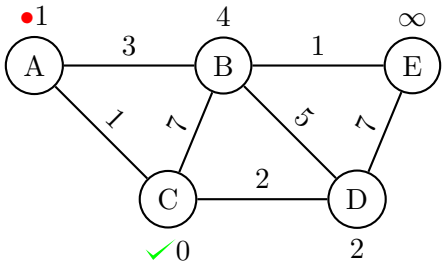

FIGURE 7 – Etape 3

Ensuite, nous marquons A comme visité et choisissons un nouveau noeud courant : D, qui est le nœud non visité avec la plus petite distance actuelle. Regardez la figure 8. Nous répétons à nouveau l'algorithme. Cette fois, nous vérifions B et E.

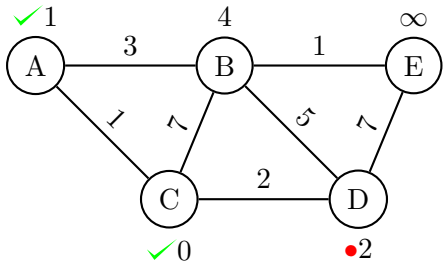

FIGURE 8 – Etape 4

Pour B, nous obtenons $2 + 5 = 7$. Nous comparons cette valeur avec la distance minimale de B (4) et laissons la plus petite valeur (4). Pour E, nous obtenons $2 + 7 = 9$, nous la comparons avec la distance minimale de E (infini) et nous laissons la plus petite (9).

Nous marquons D comme visité et mettons notre noeud actuel à B. Regardez la figure 9.

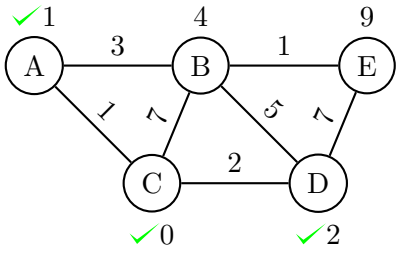

FIGURE 9 – Etape 5

Nous répétons l'algorithme pour chaque noeud et à la fin nous obtenons le graphe de la figure 10. Pour plus d'information [1].

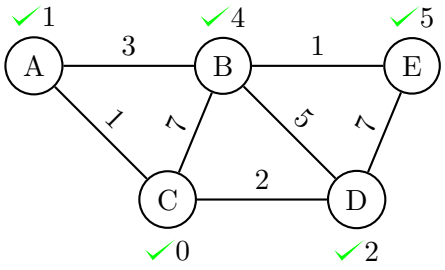

FIGURE 10 – Graphe finale

4 Complexité

4.1 Rappels sur la complexité

Définition 4.1. Soit g une fonction de $\mathfrak{F}$. On définit respectivement les notations grand omicron et grand oméga :

$$O(g) := \{\, f \in \mathfrak{F} \mid \exists c > 0 \quad \exists N \in \mathbb{N} \quad \forall n \geq N \quad 0 \leq f(n) \leq cg(n) \,\}$$

$$\Omega(g) := \{\, f \in \mathfrak{F} \mid \exists c > 0 \quad \exists N \in \mathbb{N} \quad \forall n \geq N \quad 0 \leq cg(n) \leq f(n) \,\}$$

Définition 4.2. Soit g une fonction de $\mathfrak{F}$. On définit la notation grand thêta :

$$\Theta(g) := \{\, f \in \mathfrak{F} \mid \exists a > 0 \quad \exists b > 0 \quad \exists N \in \mathbb{N} \quad \forall n \geq N \quad 0 \leq ag(n) \leq f(n) \leq bg(n) \,\}$$

Proposition 4.3. *Soient f et g deux fonctions de $\mathfrak{F}$. On a*

$$f = \Theta(g) \iff f = O(g) \quad \textit{et} \quad f = \Omega(g)$$

L'expression algébrique d'une fonction et quelques connaissances sur la croissance des fonctions réelles suffisent parfois à mettre en évidence qu'une majoration est grossière, par exemple quand on affirme que la fonction $n \to 2n + 1$ est en $O(n^3)$. Ce n'est pas la définition de la notation O qui nous permet d'affirmer que cette estimation est grossière mais la connaissance de la croissance des fonctions monomiales. Ainsi l'écriture $f = O(g)$ ne précise pas si f croit beaucoup moins vite que toute constante fois g asymptotiquement ou si f s'en approche. La notation petit o répond partiellement à cette question, quand on écrit$f = o(g)$, on signifie que :

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

taille de l'entrée	complexité requise
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(nlog(n))$ ou $O(n)$
$n > 10^6$	$O(1)$ ou $O(log(n))$

FIGURE 11 – Temps d'execution par taille de l'entrée

Définition 4.4. Soit g une fonction de $\mathfrak{F}$. On définit respectivement les notations petit omicron et petit oméga :

$$o(g) := \{ f \in \mathfrak{F} \mid \forall c > 0 \quad \exists N \in \mathbb{N} \quad \forall n \geq N \quad 0 \leq f(n) \leq cg(n) \}$$

$$\omega(g) := \{ f \in \mathfrak{F} \mid \forall c > 0 \quad \exists N \in \mathbb{N} \quad \forall n \geq N \quad 0 \leq cg(n) \leq f(n) \}$$

Pour plus d'information sur la théorie de l'infomation [6]

4.1.1 Estimation de l'efficacité

En calculant la complexité en temps d'un algorithme, il est possible de vérifier, avant de le mettre en œuvre, si'il est suffisamment efficace pour le problème. Nous partons pour les estimations qu'un ordinateur moderne peut effectuer quelques centaines de millions d'opérations en une seconde.

Par exemple, supposons que le temps limite pour un problème est d'une seconde et que la taille de l'entrée est $n = 10^5$. Si la complexité est $O(n^2)$, l'algorithme effectuera environ $(10^5)^2 = 10^{10}$ operations. Cela devrait prendre au moins quelques dizaines de secondes. Donc l'algorithme semble donc trop lent pour résoudre le problème. D'un autre côté, étant donné la taille de l'entrée, nous pouvons essayer de deviner la complexité requise pour l'algorithme. Le tableau 11 contient quelques estimations utiles en supposant une limite de temps d'une seconde. Informations utiles sur la complexité [2, p. 20].

4.2 Complexité de l'algorithme de Dijkstra

L'efficacité de l'algorithme de Dijkstra repose sur une mise en œuvre efficace de la recherche du minimum. L'ensemble $\mathcal{Q}$ est implémenté par une file à priorités. Si le graphe possède $|A|$ arêtes et $|S|$ sommets, qu'il est représenté par des listes d'adjacence et si on implémente la file à priorités par un tas binaire (en supposant que les comparaisons des poids des arêtes soient en temps constant), alors la complexité en temps de l'algorithme est $O((|A| + |S|) \times \log(|S|))$. En revanche, si on implémente la file à priorités avec un tas de Fibonacci, l'algorithme est en $O(|A| + |S| \times \log(|S|))$.

5 Applications

L'algorithme de Dijkstra est appliqué pour le calcul d'itinéraires routiers comme sur les sites Web de cartographie comme *MapQuest* ou *Google Maps*. Le poids des arcs pouvant être la distance (pour le trajet le plus court), le temps estimé (pour le trajet le plus rapide), la consommation de carburant et le prix des péages (pour le trajet le plus économique).

Une application courante de l'algorithme de Dijkstra apparaît dans les protocoles de routage interne « à état de liens », tels que *Open Shortest Path First* ou encore *PNNI* sur les réseaux *ATM*, qui permettent un routage internet très efficace des informations en cherchant le parcours le plus efficace.

Références

[1] Dijkstra's algorithm. `https://www.codingame.com/playgrounds/1608/shortest-paths-with-dijkstras-algorithm/dijkstras-algorithm`. Visité : 21/02/2022.

[2] Antti Laaksonen. Competitive programmer's handbook. Visité : 21/02/2022, 2018.

[3] Philippe Langevin. Theorie et algorithmique des graphes. Visité : 21/02/2022, 2021.

[4] Algorithme de dijkstra. `https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra`. Visité : 21/02/2022.

[5] Edsger dijkstra. `https://fr.wikipedia.org/wiki/Edsger_Dijkstra`. Visité : 21/02/2022.

[6] Jean Pierre Zanotti. Notations asymtotiques. `http://zanotti.univ-tln.fr/ALGO/II/Asymptotiques.html`. Visité : 21/02/2022.