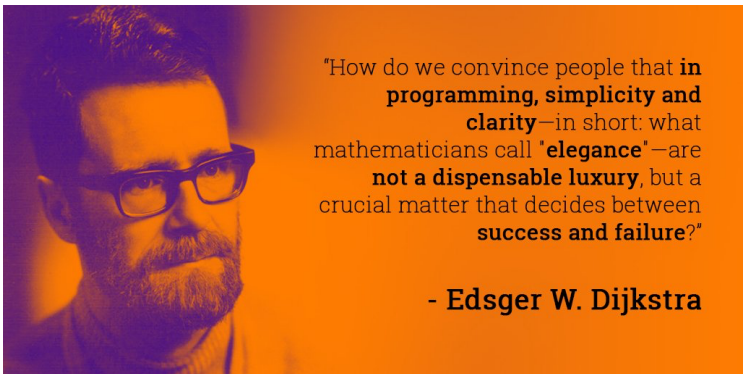
Cours 23 – Bases des graphes – Partie 3 – Algorithme de DIJKSTRA

Edsger Wybe Dijkstra né à Rotterdam le 11 mai 1930 et mort à Nuenen le 6 août 2002, est un mathématicien et informaticien néerlandais du xx^{e} siècle. 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.

Sources : https://fr.wikipedia.org/wiki/Edsger_Dijkstra

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, dans un graphe orienté pondéré par des réels positifs, ***il calcule des plus courts chemins à partir d'une source vers tous les autres sommets***. On peut aussi l'utiliser pour calculer un plus court chemin entre un sommet de départ et un sommet d'arrivée.

Sources :https://fr.wikipedia.org/wiki/Problème_de_plus_court_chemin



Soit un graphe $G = (S, A)$, orienté ou non, dont les arêtes ou arcs sont **pondérés** par des poids positifs, c'est-à-dire qu'il existe une application surjective $w: \begin{cases} A \to \mathbb{R}_+ \\ (S_D \to S_V) \to w(S_D \to S_V) \end{cases}$ qui à tout arc ou arête $(S_D \to S_V)$ menant d'un sommet de départ S_D vers un de ses voisins S_V, associe le poids (positif) qui pondère cet arc.

L'algorithme de **Dijkstra (1959)** permet de générer, **au départ d'un nœud initial** $N_D \in S$ du graphe, une (voire deux) application(s) surjectives :

- $D: \begin{cases} S \to \mathbb{R}_+ \\ S_i \to w(N_D, S_i) \end{cases}$ qui à tout sommet S_i du graphe associe la distance du plus court chemin (c'est-à-dire la somme des pondérations du chemin minimal) permettant d'aller du nœud de départ N_D au sommet S_i.

- $P: \begin{cases} S \backslash \{N_D\} \to S \\ S_i \to P(S_i) \end{cases}$ qui à tout sommet S_i du graphe associe le dernier sommet $P(S_i)$ visité (dit sommet parent, ou père de S_i) lorsque l'on parcoure le chemin le plus court menant du nœud de départ N_D au sommet S_i.

Dans cet élément de cours, nous nous focaliserons sur *D* et ignorerons *P*, qui sera introduit en TD.

A / Principe de l'algorithme

On part d'un des sommets N_D du graphe.

A.1 / Initialisation

On initialise en affectant :

→ à chaque sommet autre que le sommet initial N_D une étiquette de valeur ∞

→ et au sommet initial N_D l'étiquette 0.

Lors des itérations de l'algorithme, l'étiquette liée à chaque sommet S_i représentera la distance $D(S_i)$ du plus court chemin le séparant du sommet de départ N_D et qui soit connu à cette itération de l'algorithme.

Remarque : Avant d'avoir parcouru le graphe au départ de N_D, tout ce qu'on sait c'est que N_D est à distance 0 de lui-même, et que les autres sommets, n'ayant pas encore été explorés, sont à distance infinie.

A.2 / Processus itératif

On débute alors un parcours **en largeur** du graphe, c'est-à-dire un parcours où à chaque itération on choisit un sommet de départ selon un certain critère, et on balaie tous ses voisins directs avant de changer de sommet de départ à l'itération suivante.

- On choisit comme sommet de départ S_{pp} celui dont la distance $D(S_{\text{pp}})$ au sommet initial N_D est la plus petite (le plus proche)

  - Comme dans l'algorithme de parcours en largeur, on tient à jour une liste des sommets `deja_traites`

    à laquelle on ajoute le sommet S_{pp}, c'est-à-dire qu'on valide que la distance $D(S_{\text{pp}})$ alors connue est bien celle du plus court chemin.

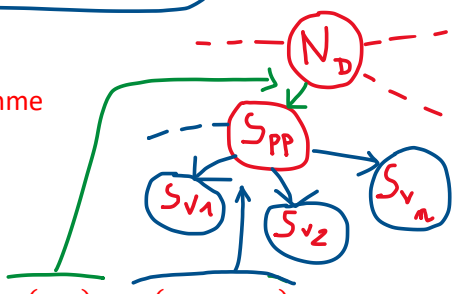
    En effet, il ne peut pas y avoir de chemin plus court y menant (puisque tous les chemins arrivant aux autres nœuds du graphe sont déjà plus longs, et que tous les poids sont positifs, inutile de chercher un chemin passant par un autre nœud).

  - Puis on va **relâcher** (c'est le terme technique) un par un chacun de ses voisins sortants S_V. C'est-à-dire que l'on teste chaque voisins sortants S_V adjacent de S_{pp}.
    Pour rappel, on note :

    $D(S_V)$ la distance entre N_D et S_V alors connue à cette itération de l'algorithme

    $D(S_{\text{pp}})$ la distance entre N_D et S_{pp} définitive (on vient de la valider)

    $w(S_{pp} \to S_V)$ le poids de l'arc ou de l'arête menant de S_{pp} vers S_V.

    On découvre un nouveau chemin menant de N_D à S_V, de longueur $D_{\text{new}} = D(S_{\text{pp}}) + w(S_{pp} \to S_V)$

    - On teste alors si $D(S_V) > D_{\text{new}}$

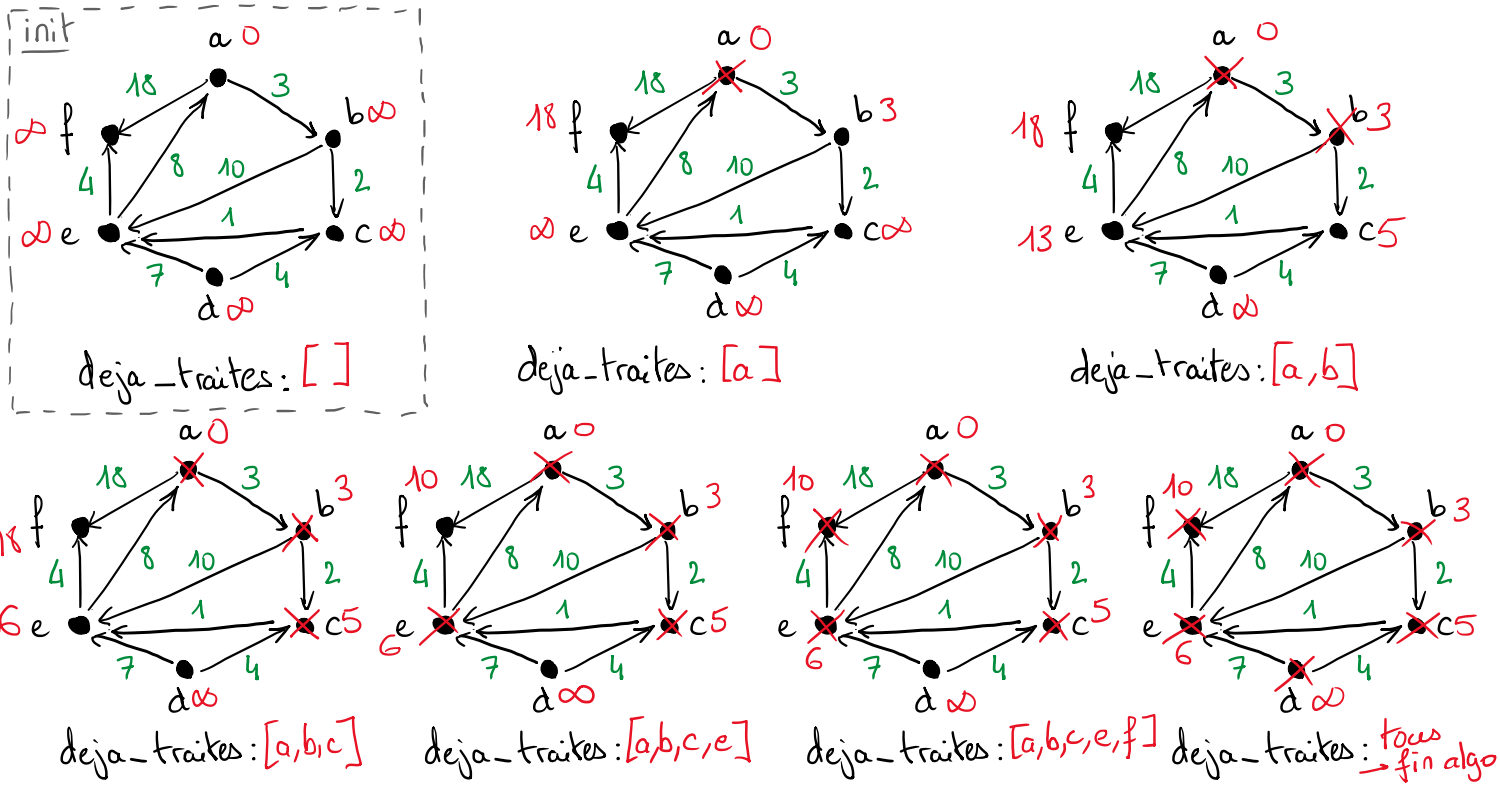
      → si c'est le cas : c'est que le nouveau chemin découvert est plus court que le dernier qu'on avait en mémoire.

      → il faut donc écraser $D(S_V) \longleftarrow D_{\text{new}}$

    - Si ce n'est pas le cas, on ne change rien, le chemin actuellement connu est, pour l'instant, le plus efficace

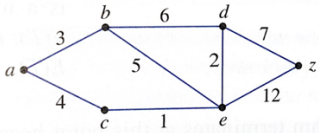
On continue itérativement jusqu'à ce que tous les sommets du graphe soient dans `deja_traites` ou que le sommet d'arrivée désiré soit dans `deja_traites`.

B / Illustration avec un exemple de graphe orienté – tous les plus courts chemins depuis *a*



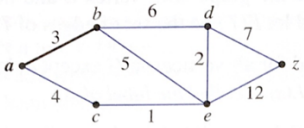
C / Autre exemple : le plus courts chemins de *a* juqu'à *z* (source : Discrete Mathematics with Applications, 5ème édition, Susanna S. Epp)

Show the steps in the execution of Dijkstra's shortest path algorithm for the graph shown below with starting vertex a and ending vertex z.



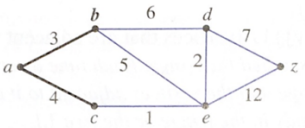
Solution

Step 1: Going into the **while** loop: $V(T) = \{a\}$, $E(T) = \varnothing$, and $F = \{a\}$



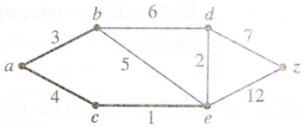
During iteration:
$F = \{b, c\}$, $L(b) = 3$, $L(c) = 4$.
Since $L(b) < L(c)$, b is added to $V(T)$, $D(b) = a$, and $\{a, b\}$ is added to $E(T)$.

Step 2: Going into the **while** loop: $V(T) = \{a, b\}$, $E(T) = \{\{a, b\}\}$



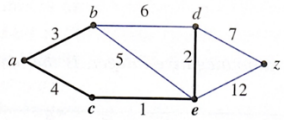
During iteration:
$F = \{c, d, e\}$, $L(c) = 4$, $L(d) = 9$, $L(e) = 8$.
Since $L(c) < L(d)$ and $L(c) < L(e)$, c is added to $V(T)$, $D(c) = a$, and $\{a, c\}$ is added to $E(T)$.

Step 3: Going into the **while** loop: $V(T) = \{a, b, c\}$, $E(T) = \{\{a, b\}, \{a, c\}\}$



During iteration:
$F = \{d, e\}$, $L(d) = 9$, $L(e) = 5$
$L(e)$ becomes 5 because ace, which has length 5, is a shorter path to e than abe, which has length 8.
Since $L(e) < L(d)$, e is added to $V(T)$, $D(e) = c$, and $\{c, e\}$ is added to $E(T)$.

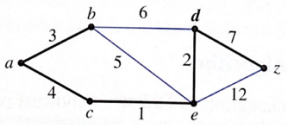
Step 4: Going into the **while** loop: $V(T) = \{a, b, c, e\}$, $E(T) = \{\{a, b\}, \{a, c\}, \{c, e\}\}$



During iteration:
$F = \{d, z\}$, $L(d) = 7$, $L(z) = 17$
$L(d)$ becomes 7 because *aced*, which has length 7, is a shorter path to d than *abd*, which has length 9.

Since $L(d) < L(z)$, d is added to $V(T)$, $D(d) = e$, and $\{e, d\}$ is added to $E(T)$.

Step 5: Going into the **while** loop: $V(T) = \{a, b, c, e, d\}$, $E(T) = \{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}\}$



During iteration:
$F = \{z\}$, $L(z) = 14$
$L(z)$ becomes 14 because *acedz*, which has length 14, is a shorter path to d than *abdz*, which has length 17.

Since z is the only vertex in F, its label is a minimum, and so z is added to $V(T)$, $D(z) = d$, and $\{d, z\}$ is added to $E(T)$.

Execution of the algorithm terminates at this point because $z \in V(T)$. The shortest path from a to z has length $L(z) = 14$.

Keeping track of the steps in a table is a convenient way to show the action of the algorithm. Table 10.6.1 does this for the graph in Example 10.6.5.

TABLE 10.6.1

Step	$V(T)$	$E(T)$	F	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(z)$
0	$\{a\}$	$\varnothing$	$\{a\}$	**0**	∞	∞	∞	∞	∞
1	$\{a\}$	$\varnothing$	$\{b, c\}$	0	**3**	4	∞	∞	∞
2	$\{a, b\}$	$\{\{a, b\}\}$	$\{c, d, e\}$	0	3	**4**	9	8	∞
3	$\{a, b, c\}$	$\{\{a, b\}, \{a, c\}\}$	$\{d, e\}$	0	3	4	9	**5**	∞
4	$\{a, b, c, e\}$	$\{\{a, b\}, \{a, c\}, \{c, e\}\}$	$\{d, z\}$	0	3	4	**7**	5	17
5	$\{a, b, c, e, d\}$	$\{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}\}$	$\{z\}$	0	3	4	7	5	**14**
6	$\{a, b, c, e, d, z\}$	$\{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}, \{e, z\}\}$							

In step 1, $D(b) = a$; in step 2, $D(c) = a$; in step 3, $D(e) = c$; in step 4, $D(d) = e$; and in step 5, $D(z) = e$. Working backward gives the vertices in the shortest path. Because $D(z) = d$, $D(d) = e$, $D(e) = c$, and $D(c) = a$, the shortest path from a to z is *acedz*. ■