Algorithme de Dijkstra

Problématique :

Un problème fréquent qui traverse les disciplines et notre quotidien consiste à chercher à déterminer le plus court chemin pour passer d'un point à un autre au sein d'un réseau ou d'un graphe en sélectionnant, au choix, des critères tels que le temps de parcours, la distance, le coût, etc...
Plusieurs solutions ont été imaginées mais l'algorithme de Dijkstra, créé en 1959 par E. W. Dijkstra, est une façon élégante et économique en calculs de résoudre ce problème.

Partie I : Introduction sur les graphes

I.1 : Quelques définitions

Qu'est-ce qu'un **graphe** ?

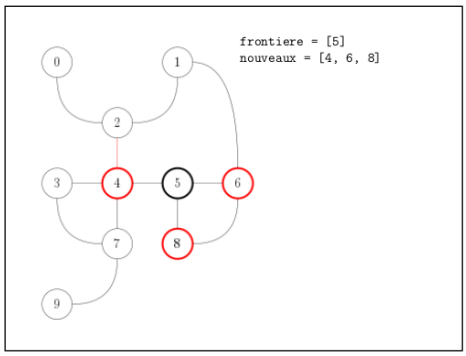
— D'abord un ensemble de nS objets désignés sous le terme de **sommets** ou **noeuds** (ce peut être des arrêts de bus, des stations de métro, des villes au sein d'un réseau ferré, etc.). Parmi ces sommets, deux d'entre eux peuvent être considérés comme le point de départ et d'arrivée du trajet à parcourir, peu importe lesquels. Ω désignera l'ensemble des sommets possibles.

— Une liste **d'arêtes** qui relient ces sommets entre eux en sachant qu'une arête entre les sommets i et j n'existe que si ces sommets sont adjacents au sein du graphe. Nous supposerons par la suite que toute arête peut être empruntée dans les deux sens mais ça n'a rien d'indispensable et l'algorithme de Dijkstra décrit par la suite s'applique aussi bien lorsque certains chemins sont à sens unique (on parlera dans ce cas d'« arc »).
Ce n'est pas le cas de tous les graphes, mais dans le cas particulier de l'algorithme de Dijkstra, à chaque arête est associé un **poids** noté $\tau(i, j)$ qui dépend du problème posé (ce poids peut désigner les distances qui séparent chaque sommet, les temps de parcours, le nombre de feux de circulation, etc...). On dira alors que le graphe est « pondérés »(et si on pense aux pixels voisins d'une image qui sont reliés par une arête à condition d'être de la même couleur, on voit bien que cette notion de pondération n'a pas toujours de sens...)

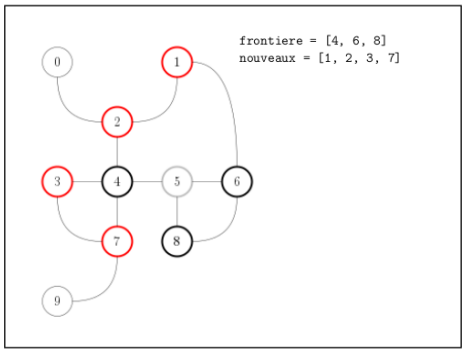
I.2 : Le cas des graphes non pondérés

Parcourir un graphe non pondéré à partir d'un de ses sommets, c'est passer par tous les noeuds que l'on peut atteindre à partir de ce sommets. Il y a plusieurs façons de le faire mais comme l'algorithme de Dijkstra est une adaptation du parcours dit *en largeur* au cas des graphes pondérés, nous allons présenter rapidement ce type de parcours : On suppose partir d'un sommet s qui a des voisins $v_1, \cdots, v_p$ et on cherche à explorer tour à tour chacun de leurs voisins qui viendront s'ajouter aux sommets vus à condition de ne pas y être encore passé. Au fur et à mesure du traitement, les sommets à distance k de s sont traités et génèrent de nouveaux sommets à distance $k+1$. Quand on a fini de traiter les sommets à distance k, on entame les sommets à distance $k+1$ qui vont donner des sommets à distance $k+2$. On parlera à chaque étape de « frontière ».

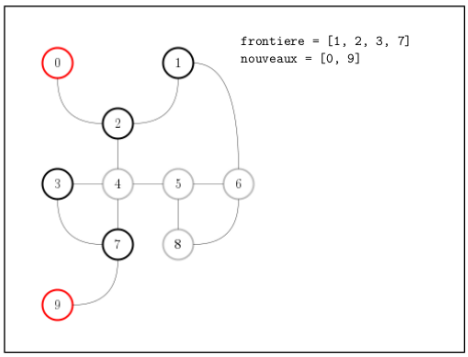
Pour plus de clarté, suivons l'exemple ci-dessous dans lequel les frontières sont en noire et les voisins non encore vus en rouge. Il s'agit de parcourir le graphe à partir du sommet $s = 5$ qui initialise la frontière à l'étape 0. Ses voisins, 4, 6 et 8 formant la frontière à l'étape 1, soit les sommets accessibles en un pas de temps. Etc.



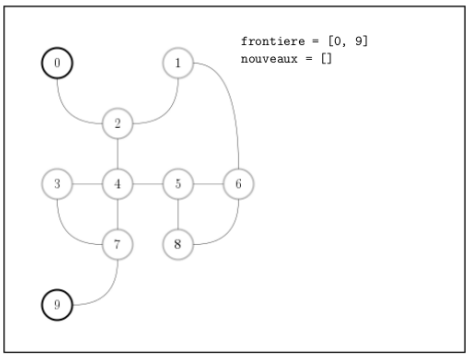
étape 0



étape 1



étape 2

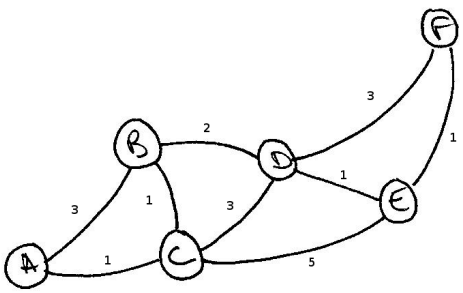


étape 3

En trois pas de temps, l'ensemble des sommets reliés au sommet 5 ont été atteints. Vous pouvez maintenant travailler sur une application plus « concrète » dans la première partie du Notebook consacrée à la recherche des composantes connexes sur une image.

II : Le cas des graphes pondérés - Algorithme de Dijkstra

Considérons comme support à nos explications le cas simple où les distances (dans l'unité de votre choix) sont indiquées sur chaque arête comme dans le schéma ci-dessous :



Un trajet entre deux sommets S_i et S_j correspond à un chemin dans le graphe qui part de S_i et arrive en S_j. Le poids d'un tel trajet dont les sommets auraient pour indices : $i = i_0 \to i_1 \to \cdots \to i_n = j$ et qui utilise n arêtes est la somme des poids de chacune des arêtes. Ou encore :

$$\text{poids}(S_i = S_{i_0} \to S_{i_1} \to \cdots \to S_{i_n} = S_j) = \sum_{k=0}^{n-1} \tau(i_k, i_{k+1})$$

Le principe fondamental de l'algorithme repose sur l'idée que, si le trajet du sommet S_i au sommet S_j est optimal, alors chaque trajet intermédiaire jusqu'à l'un quelconque des sommet S_k de ce parcours est lui-même optimal (ce qui semble aller de soi car s'il existait un meilleur chemin de S_i à S_k que le sous-chemin $i_0 \to i_1 \to \cdots \to i_k$ au sein du chemin de S_i à S_j, alors on pourrait remplacer ce sous-chemin par un autre qui ferait diminuer le poids total. Ceci contredirait l'optimalité du chemin de départ).

En partant de S_i considéré comme sommet de départ, il s'agit donc pas à pas, sommet après sommet, de construire le trajet qui minimise le « poids » associé à son parcours.

✐ Notons qu'il est possible que le trajet recherché ne soit pas unique mais en aucun cas nous ne réclamons dans ce contexte l'unicité de la solution...

✐ Avant de vous plonger dans l'approche théorique, je vous suggère une présentation vidéo, pas à pas, de l'algorithme, destinée à des élèves de terminale. Utilisez pour ça le qrcode suivant :

II.1. Mise en place de l'algorithme.

Supposons que nous nous intéressions aux distances des trajets qui pondèrent le graphe. Il s'agira pour nous de déterminer le trajet de distance minimale entre un sommet initial désigné comme étant la **source** et un sommet final désigné comme le **puits**.

1) Outils et initialisation :

— L'essentiel consiste d'abord à savoir déterminer les voisins d'un sommet donné. Nous choisissons de décrire le graphe par une liste dite « d'adjacence ». Ça signifie que pour **chaque** sommet, on stocke la liste de ses *voisins / successeurs* et on met toutes ces listes dans une grande liste. Et comme notre graphe est pondéré, on décide de stocker les voisins sous forme d'une liste de couples. Sur notre exemple page précédente, pour le voisin B cette liste sera `[('A', 3), ('C', 1), ('D', 2)]` puisque ce sommet est relié au sommet `'A'` avec un poids de 3, au sommet `'C'` avec un poids de 1 et au sommet `'D'` avec un poids de 2...

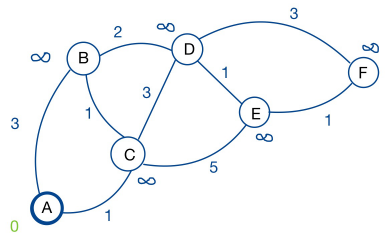
— Dans l'algorithme de Dijkstra, on souhaite toujours parcourir les sommets par distance croissante au noeud initial. On créé donc une liste `Distances` qui stockera les distances mises à jour à chaque pas de temps pour atteindre chacun des sommets depuis la source S_0 que nous supposerons pour l'instant être A (mais qui doit pouvoir être n'importe lequel des sommets). A l'étape 0, la distance pour atteindre A est naturellement fixée à 0 tandis qu'on affectera **l'infini** à chacun des autres sommets, non encore atteints.

— La seconde liste utilisée appelée `S_selec`, contient les sommets accessibles à l'étape k dont nous pourrons extraire le sommet qui minimise le trajet depuis l'origine (du moins à cette étape).

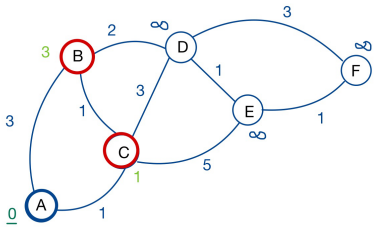
Lors de l'initialisation, `S_selec` sera réduit au seul sommet A, départ imposé de notre parcours, assorti de sa distance à l'origine. Soit `S_selec = [(A, 0)]` (il faut bien l'initialiser !). Par la suite, cette liste va s'accroître de tous les sommets candidats. A chaque étape, on sélectionne dans cette liste, celui qui est le plus proche de l'origine (tout sommet étant renseigné sous la forme d'un couple `(sommet, distance_a_l_origine)`). Le trajet passera désormais par ce sommet. On ne souhaite pas qu'il puisse être à nouveau sélectionné car on ne repasse pas deux fois par le même sommet. On fera appel à la méthode `L.remove()` pour le supprimer.

— Enfin une dernière liste, `Parents`, indique pour chaque sommet le sommet d'où l'on vient (en indice dans la vidéo). Au départ, n'étant pas encore parti, nous mettrons un A pour le sommet A (on était en A avant d'être en A...) et un `None` (comme 'vide') pour chacun des autres sommets qui, n'ayant pas été parcouru, n'ont évidemment pas de sommet « parent »... Ce tableau est primordial car, *in fine*, c'est lui qui nous permettra de reconstruire, « à reculons », le plus court chemin...
✎ On insiste sur le fait que `Parents['A'] = 'A'` et que ce sera le seul sommet à vérifier cette propriété...

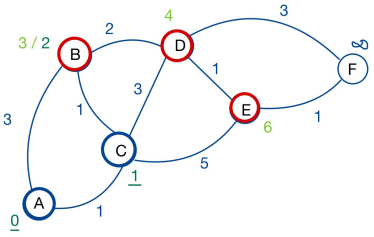
2) Une approche par l'exemple :



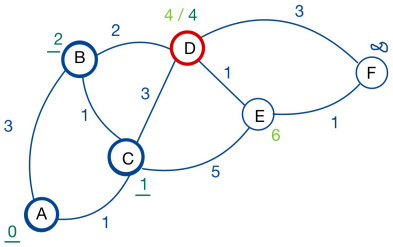
étape 0



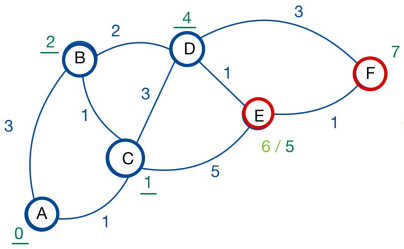
étape 1



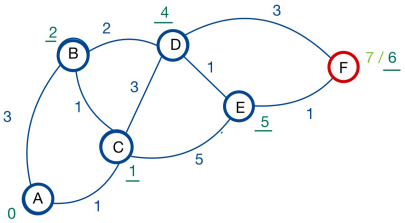
étape 2



étape 3



étape 4



étape 5

① ***Étape 0 :*** On part de A. Dans `Distances`, la distance depuis A est nulle et tous les autres sommets sont initialisés à une distance infinie. Seul le sommet A est sélectionné. Soit :

```
Distance = [0, inf, inf, inf, inf, inf]
Parents = [A, -, -, -, -, -]
```

② ***Étape 1 :*** Depuis A, le sommet B est à 3 km et le sommet C est à 1 km. Soit :

```
Distance = [0, 3, 1, inf, inf, inf]
Parents = [A, A, A, -, -, -]
S_selec = [(B,3), (C,1)]
```

Le plus court trajet parmi les sommets sélectionnés est de 1 km et aboutit en C : On sélectionne donc le sommet C qu'on supprime de la liste des sommets sélectionnés

③ ***Étape 2 :*** Partant de A via C, on va :
- en B avec un trajet total de longueur : 1 km $+d(C, B) = 1 + 1 = 2$ km. C'est mieux que 3 km. On indique dans `Parents` que pour aller en B on passe désormais par C avec une distance égale à 2.
- en D avec un trajet total de longueur : 1 km $+d(C, D) = 1 + 3 = 4$ km. C'est mieux qu'une distance infinie. On met à jour le tableau `Parents` en indiquant qu'on passe par C avec une distance égale à 4.
- en E avec un trajet total de longueur : 1 km $+d(C, E) = 1 + 5 = 6$ km. C'est également mieux qu'une distance infinie. On met également à jour la liste `Distance`.

Soit :

```
Distance = [0, 2, 1, 4, 6, inf]
Parents = [A, C, A, C, C, -]
S_selec = [(B,2), (D,4), (E, 6)]
```

A l'étape 2, le plus court trajet est de longueur 2 ; c'est celui qui mène en B : On sélectionne le sommet B et on le supprime de `s_selec`.

④ ***Étape 3 :*** Depuis **B** (à une distance de 2 kilomètres) en venant de C, les voisins sont : `[(A, 2+3), (C, 2+1), (D, 2+2)]`
Aucun des voisins n'est à une distance inférieure à celle déjà obtenue. On ne touche pas à la liste `Distance` qui reste inchangée.
`S_selec = [(D,4), (E, 6]` A cette étape le plus court trajet est celui qui mène en D (4 km) : On sélectionne le sommet D et on le supprime de `S_selec`.

⑤ ***Etape 4 :*** Depuis **D** (à une distance de 4 km), les voisins sont : `[(B, 4+2), (C, 4+3), (E, 4+1), (F, 4+3)]`.
Au regard de la liste `Distance`, seuls les voisins E et F sont désormais plus proche de A en passant par D. On met donc à jour nos listes pour ces deux sommets :

```
Distance = [0, 2, 1, 4, 5, 7]
Parents = [A, C, A, C, D, D]
S_selec = [(E,5), (F,7)]
```

A cette étape le plus court trajet est celui qui mène en E (5 km) : On sélectionne le sommet E et on le supprime de `S_selec`.

⑥ ***Etape 5 :*** Depuis **E** (5 km), les voisins sont `[(C, 5+5), (D, 5+1), (F, 5+1)]`.
Seul le voisin F est désormais plus proche de A. On met à jour nos liste pour ce sommet. Soit :

```
Distance = [0, 2, 1, 4, 5, 6]
Parents = [A, C, A, C, D, E]
S_selec = [(F,6)]
```

A cette étape le plus court trajet est celui qui mène en F (6 km). C'est fini !

A l'issue de l'algorithme, la liste des parents est donc :

`Parents=[A,C,A,C,D,E]`

Et si on se souvient que les éléments de cette liste représentent les parents des sommets du graphe qui sont `[A,B,C,D,E,F]`, on obtient que le dernier sommet parcouru pour rejoindre F est E, puis grâce à la colonne du sommet E, on obtient que son prédécesseur est D dont le prédécesseur est C (colonne du sommet D), lui-même précédé du sommet A (colonne du C) avec `Parents[A] = A`. Nous sommes arrivés à la source. On s'arrête et on réécrit tout ça dans le bon sens...

Conclusion : Le plus court chemin est $A - C - D - E - F$ et il fait 6 km

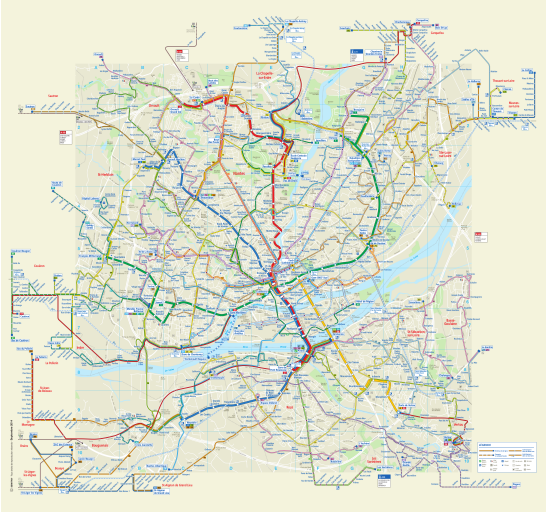
Partie III : Programmation

Exercice 1 : Mise en place de l'algorithme de `dijkstra` :

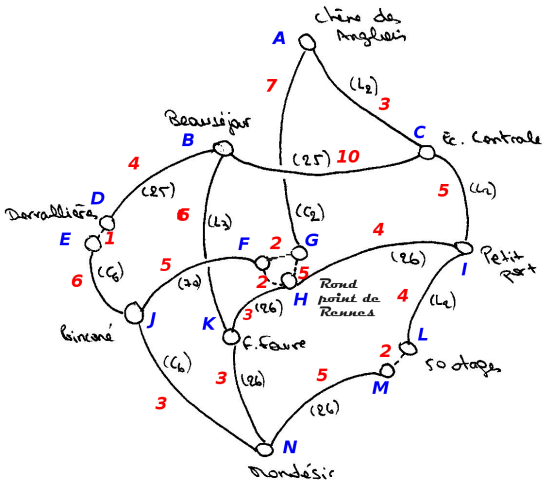
Reportez-vous au notebook intitulé `devoir_maison_Dijkstra` et complétez chacune des fonctions afin de pouvoir mettre en pratique l'algorithme de Dijkstra sur le parcours de A à F proposé dans ce polycopié.
Vérifiez à cette occasion que vous pouvez choisir n'importe lequel des sommets comme étant la « source » et le « puits » et vérifiez les solutions proposées.

Exercice 2 : Un trajet en bus :

On souhaite, en empruntant le réseau de transport de la TAN reproduit ci-dessous, se rendre du Chêne des Anglais à l'arrêt Mondésir, autrement dit du point A au point N dans une version simplifiée réduite aux lignes les plus rapides du réseau comme les lignes de tramway L_2 et L_3 et les lignes C_2 et C_6.
Les temps entre les stations sont indiqués en rouge. Ce sont ceux fournis par la TAN et sont donnés en minutes. Les temps de changement de lignes, quant à eux, sont le fruit de pures spéculations...



Plan de la TAN



Graphe simplifié (temps en minutes)

① Le graphe étant à la fois défini par l'ensemble des stations marquées d'une lettre de A à N et par l'ensemble des arêtes qui les relient, vérifier que '`Graphe2` fourni en fin de Notebook est bien sa liste d'adjacence.

② Appliquer vos fonctions `dijkstra()` et `meilleur_trajet()` et fournir le meilleur trajet pour arriver en moins de 20 mn au lycée en partant du Chêne des anglais.