Informatique

11

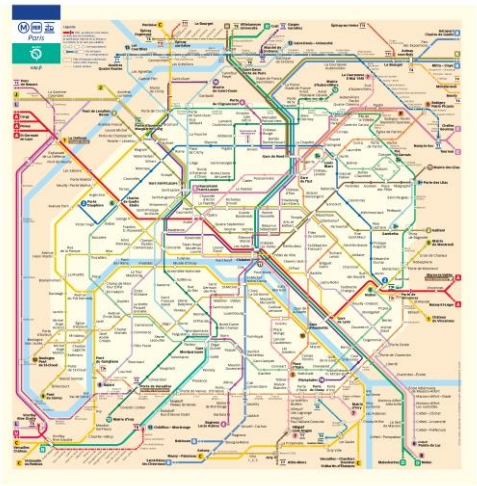
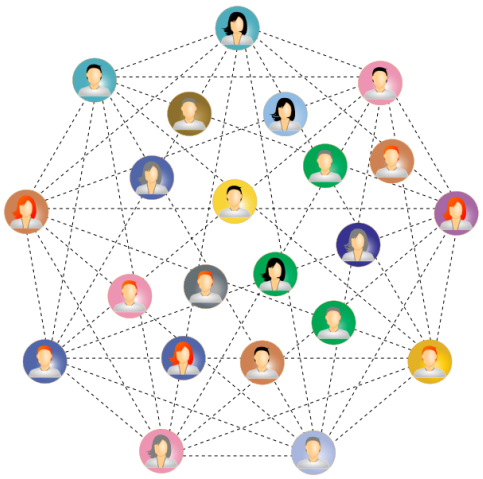
Bases des graphes

Cours

Bases des graphes

1.I. Introduction

On utilise les graphes pour représenter un ensemble d'éléments (sommets) et de liens (arêtes, arcs) entre ceux-ci. Par exemple :

Réseaux de transport (personnes, marchandises, eau, électricité, gaz, internet…)	Réseaux sociaux
	

1.II. Graphes

1.II.1 Définitions

Un **graphe non orienté d'ordre n** est un ensemble de n points ou **sommets** reliés entre eux ou non, par des lignes appelées **arêtes**.

Un **graphe orienté d'ordre n** contient des flèches et non des arêtes, appelées **arcs**. Un sommet peut être relié à lui-même par une boucle.

Deux sommets sont dits **adjacents** s'ils sont reliés par une arête ou un arc. Un sommet adjacent à aucun autre est dit **isolé**.

Le **degré** d'un sommet $d(s)$ est le nombre d'arêtes ou d'arcs (quel que soit leur sens) auxquels il est relié (une boucle compte deux fois).

Pour un graphe orienté, le degré entrant d'un sommet $d_-(s)$ est le nombre d'arcs dirigés vers ce sommet, et le **degré sortant** $d_+(s)$ le nombre d'arcs partant de ce sommet. On a alors $d(s) = d_-(s) + d_+(s)$.

Dans un graphe (orienté ou non), la somme des degrés de chaque sommet est égale au double du nombre total d'arêtes/arcs du graphe N : $\sum d(s_i) = 2N$. En effet, il y a forcément un départ et une arrivée par arête/arc.

Un **graphe complet** est un graphe dans lequel tous les sommets sont adjacents entre eux.

Dans un graphe non orienté, une **chaine** de longueur n est une succession de n arêtes telles que l'extrémité de chacune est l'origine de la suivante, sauf pour la dernière.

Dans un graphe orienté, on parle de **chemin**, et il suit le sens des flèches.

Un graphe non orienté est **connexe** s'il y a une chaine entre n'importe quelle paire de sommets distincts du graphe (pour les graphes orientés, on parle de faiblement et fortement connexe, mais cette notion n'est à priori pas au programme).

Lorsqu'une chaine/un chemin possède le même sommet de départ et d'arrivée, on dit qu'elle/il est **fermé**(e).

Lorsqu'une chaine/un chemin fermé(e) est composé(e) d'arêtes/arcs distincts, on parle de **cycle/circuit**.

Bilan	Graphe non orienté	Graphe orienté
Lien entre sommets	Arêtes	Arcs
Succession de sommets …	Chaine	Chemin
Chaine/chemin fermé d'arêtes/arcs distincts	Cycle	Circuit

On appelle **graphe pondéré**, un graphe pour lequel chaque arête/arc possède un poids.

1.II.2 Notations

Soit S l'ensemble des sommets d'un graphe G.

On appelle A l'ensemble :

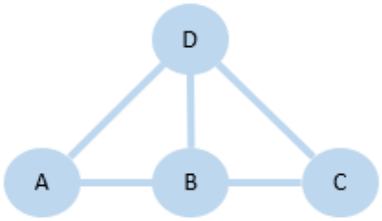
- Des arêtes d'un graphe non orienté tel que $A = \left\{\{s_i, s_j\}, s_i \in S, s_j \in S\right\}$
- Des arcs d'un graphe non orienté tel que $A = \left\{(s_i, s_j), s_i \in S, s_j \in S\right\}$

On note $G = (S, A)$ le graphe composé de ces sommets et arêtes/arcs.

On note $G = (S, A, \omega)$ un graphe pondéré tel que chaque arc/arête a un poids $\omega(s_i, s_j)$.

1.II.3 Exemples

1.II.3.a Graphe non orienté

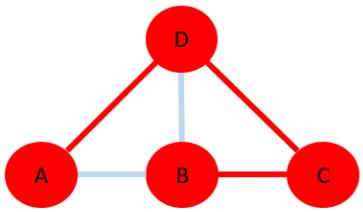


Etude de ce graphe :

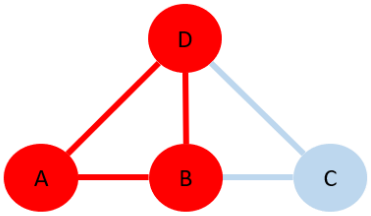
- Graphe non orienté d'ordre 4 (4 sommets A, B, C et D)
- Liste des sommets adjacents : AB, AD, BD, BC, CD
- Table des degrés des sommets :

Sommet s_i	A	B	C	D
$d(s_i)$	2	3	2	3

- Somme des degrés : On a $\sum d(s_i) = 10$, soit 5 arêtes.
- Exemple d'une chaine (ADCB) et d'un cycle (ADBA) :



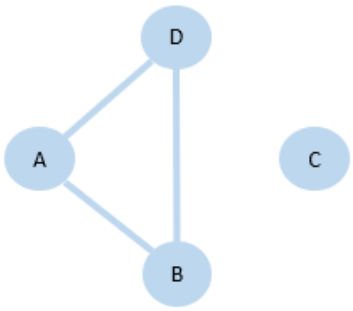
Chaine ADCB



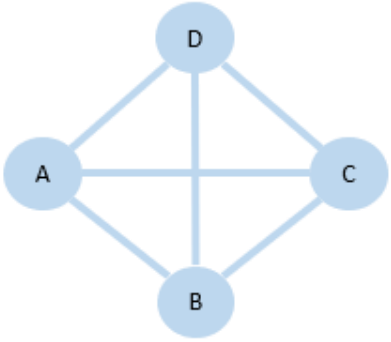
Cycle ADBA

- Le graphe est connexe puisqu'il existe une chaine entre tous les sommets pris deux à deux

Dans le graphe ci-contre, « C » est un sommet isolé :

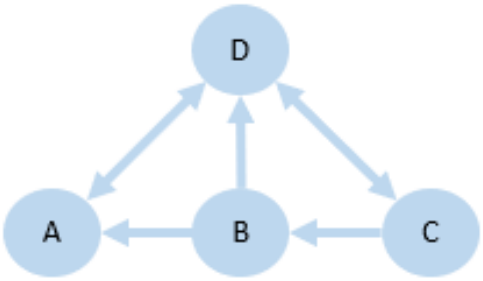


Voici enfin un graphe non orienté complet :



1.II.3.b Graphe orienté

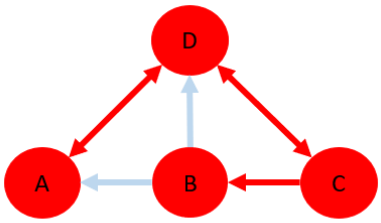
Etude de ce graphe :



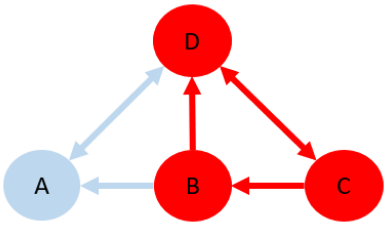
- Graphe orienté d'ordre 4 (4 sommets A, B, C et D)
- Liste des sommets adjacents : AB, AD, BD, BC, CD
- Table des degrés des sommets :

Sommet s_i	A	B	C	D
$d_-(s)$	2	1	1	3
$d_+(s)$	1	2	2	2
$d(s)$	3	3	3	5

- Somme des degrés : On a $\sum d(s_i) = 14$, soit 7 arcs (une double flèche représente 2 arcs).
- Exemple d'un chemin (ADCB) et d'un circuit (BDCB) :



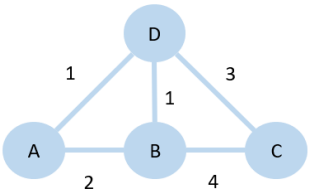
Chemin ADCB

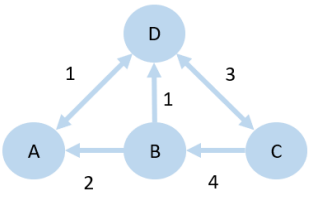


Circuit BDCB

1.II.3.c Graphe pondéré

On s'intéresse maintenant à des graphes dont les arcs représentent une valeur utile au problème étudié (distances ou temps entre deux points d'un réseau de transport par exemple).





Ainsi, dans l'exemple proposé :

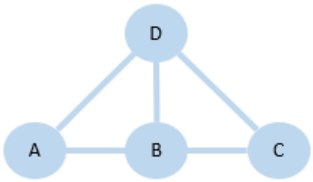
- Sur le graphe non orienté, le trajet de A à B et de B à A prendrait 2 minutes.
- Sur le graphe orienté, il est possible d'aller de B vers A en 2 minutes, mais impossible d'aller de A à B.

1.II.4 Matrices d'adjacence

1.II.4.a Définition

A tout graphe G d'ordre p, de sommets notés s_i, on peut associer une matrice carrée d'ordre p : $M = (m_{ij})$ où m_{ij} est le nombre d'arcs/arêtes reliant les sommets s_i à s_j. Cette matrice est appelée matrice d'adjacence associée à G.

Exemples :

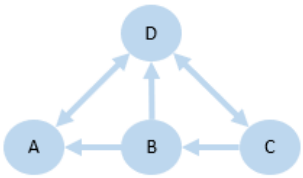


Destination

PROVENANCE

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

;



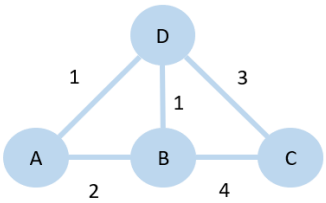
Destination

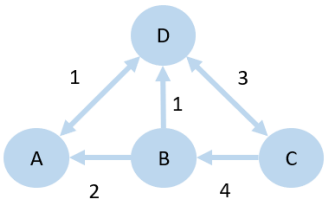
PROVENANCE

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

1.II.4.b Matrice des distances

Reprenons les deux exemples précédents :





Pour représenter les pondérations, on propose d'utiliser une nouvelle matrice des distances sur le principe de la matrice d'adjacence, contenant les pondérations à la place des 0 et 1 :

Destination

PROVENANCE

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 4 & 1 \\ 0 & 4 & 0 & 3 \\ 1 & 1 & 3 & 0 \end{bmatrix} \end{array}$$

;

Destination

PROVENANCE

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 0 & 4 & 0 & 3 \\ 1 & 0 & 3 & 0 \end{bmatrix} \end{array}$$

On introduit alors la notion de **poids d'une chaine**, comme somme de toutes les pondérations des arêtes/arcs composant la chaine. Exemple : dans le graphe orienté, le poids de la chaine ADCB vaut 8.

1.II.5 Liste d'adjacence

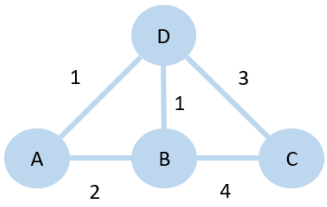
Une liste d'adjacence est :

- Pour un graphe non orienté : la liste des voisins de chaque sommet
- Pour un graphe orienté : liste des sommets en bout des arcs pour chaque sommet

Sous Python, on pourra stocker la liste d'adjacence sous deux formes :

- Une liste
- Un dictionnaire

Exemple :



	Liste	Dictionnaire
Initialisation	`>>> S = ["A","B","C","D"]` `>>> L = [["B","D"],["A","C","D"],["B","D"],["A","B","C"]]`	`>>> D = {"A":["B","D"],"B":["A","C","D"],"C":["B","D"],"D":["A","B","C"]}`

Utilisation	`>>> s = "B"`
	`>>> i = S.index(s)`
	`>>> i`
	`1`
	`>>> L[i]`
	`['A', 'C', 'D']`
		`>>> D["B"]`
		`['A', 'C', 'D']`

Remarque	Il est nécessaire de passer par une liste des sommets pour associer un indice à chaque sommet donnant alors ses voisins dans L	Le dictionnaire est bien plus pratique 😊

On pourra par ailleurs ajouter dans ce dictionnaire les pondérations, par exemple :

```
>>> D["B"]
[['A', 2], ['C', 4], ['D', 1]]
```

1.III. Parcours d'un graphe

1.III.1 Les files et piles

1.III.1.a Piles

Une pile (**stack** en anglais) est un ensemble de données dans lequel on ajoute ou l'on supprime un élément à partir de la même extrémité appelée sommet de la pile (**top** en anglais).

On effectue donc les opérations suivantes :

- Empiler (**push** ou append) : Ajouter un élément sur le sommet de la pile
- Dépiler ou désempiler (**pop**) : Supprimer un élément sur le sommet de la pile

Le principe des opérations repose sur la règle « dernier arrivé, premier sorti » ou LIFO en anglais : « last in, first out ».

Soit une pile composée de $n+1$ objets : $Pile = \begin{bmatrix} objet\_n \\ \vdots \\ objet\_1 \\ objet\_0 \end{bmatrix}$

1.III.1.a.i Empiler

Soit New_Objet un objet.

Nous souhaitons empiler New_Objet dans $Pile$. Nous créons donc une fonction $empiler(Pile, New\_Objet)$ qui va inclure l'objet New_Objet dans la pile $Pile$

Point de départ	$\rightarrow New\_Pile = empiler(Pile, New\_Objet) \rightarrow$	Résultat
$Pile = \begin{bmatrix} objet\_n \\ \vdots \\ objet\_1 \\ objet\_0 \end{bmatrix}$ $x = New\_Objet$	$New\_Pile = \begin{bmatrix} New\_Objet \\ objet\_n \\ \vdots \\ objet\_1 \\ objet\_0 \end{bmatrix}$	

1.III.1.a.ii Dépiler

Nous souhaitons dépiler la pile d'un élément, soit le dernier inséré $Objet\_n$. Nous créons donc une fonction $depiler(Pile)$:

Point de départ	$\rightarrow New\_Pile, Objet = depiler(Pile) \rightarrow$	Résultat
$Pile = \begin{bmatrix} objet\_n \\ \vdots \\ objet\_1 \\ objet\_0 \end{bmatrix}$	$New\_Pile = \begin{bmatrix} objet\_n - 1 \\ \vdots \\ objet\_1 \\ objet\_0 \end{bmatrix}$ $Objet = objet\_n$	

1.III.1.a.iii Les piles sous Python

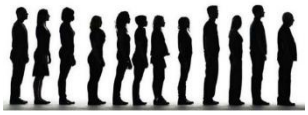
Attention : nous proposons de gérer des piles sous forme de listes. MAIS on peut utiliser n'importe quel autre outil, comme un fichier texte dans lequel empiler consiste à ajouter une ligne, une base de données, ou encore une image dans laquelle on stockerait des entiers de 0 à 255 dans les bits de ses pixels… Nous verrons dans le cadre du TP sur les piles ce que cela engendre.

Il est simple d'implémenter les piles sous Python avec des listes car les fonctions existent :

```
L = []              # Création d'une liste vide
L.append(x)         # Ajoute l'objet x à la liste L
x = L.pop()         # Enlève le dernier élément de la liste L et l'affecte à
la variable x
```

Cette implémentation est intéressante car les fonctions append(x) et pop() sont de complexité O(1) quelle que soit la taille de la liste.

1.III.1.b Files



Une file (**queue** en anglais) est un ensemble de données dans lequel

- On ajoute les éléments toujours du même côté appelé fin, arrière ou queue (**back** ou **tail** en anglais)
- On supprime les éléments toujours du même côté appelé début, avant ou tête (**front** ou **head** en anglais), côté opposé à l'arrière

On effectue donc les opérations suivantes :

- Enfiler (**enqueue**) : Ajouter un élément à la queue de la file
- Défiler (**dequeue**) : Supprimer un élément à la tête de la file

Le principe des opérations repose sur la règle « premier arrivé, premier sorti » ou FIFO en anglais : « first in, first out ».

Soit une file composée de $n+1$ objets : $File = \begin{bmatrix} objet\_0 \\ objet\_1 \\ \vdots \\ objet\_n \end{bmatrix}$

1.III.1.b.i Enfiler

Soit New_Objet un objet. Nous souhaitons enfiler New_Objet dans $File$. Nous créons donc une fonction $enfiler(File, New\_Objet)$ qui va inclure l'objet New_Objet dans la file $File$

Point de départ	$\rightarrow New\_File = enfiler(File, New\_Objet) \rightarrow$	Résultat
$File = \begin{bmatrix} objet\_0 \\ objet\_1 \\ \vdots \\ objet\_n \end{bmatrix}$ $x = New\_Objet$	$New\_File = \begin{bmatrix} objet\_0 \\ objet\_1 \\ \vdots \\ objet\_n \\ New\_Objet \end{bmatrix}$	

1.III.1.b.ii Défiler

Nous souhaitons dépiler la file d'un élément, soit le dernier inséré $Objet\_n$. Nous créons donc une fonction $defiler(File)$:

Point de départ	$\rightarrow New\_File, Objet = defiler(File) \rightarrow$	Résultat
$File = \begin{bmatrix} objet\_0 \\ objet\_1 \\ \vdots \\ objet\_n \end{bmatrix}$	$New\_File = \begin{bmatrix} objet\_1 \\ objet\_2 \\ \vdots \\ objet\_n \end{bmatrix}$ $Objet = objet\_0$	

1.III.1.b.iii Les files sous Python

Il est simple d'implémenter les piles sous Python avec des listes car les fonctions existent :

```
L = []              # Création d'une liste vide
L.append(x)         # Ajoute l'objet x à la liste L
x = L.pop(0)        # Enlève le premier élément de la liste L, l'affecte à
la variable x et la liste L se retrouve avec cet élément en moins
```

Toutefois, si la liste L possède n termes, pop(0) est de complexité $O(n)$ ☹

Il faut donc utiliser le module « collections » et son sous module « deque ». Ce module implémente des types de données de conteneurs spécialisés permettant un ajout/retrait rapide d'éléments de chaque côté de l'objet (file ou pile ici) utilisé.

```
from collections import deque

f = deque()         # Création une « dèque » vide
f.append(x)         # Ajoute l'objet x à droite
f.appendleft(x)     # Ajoute l'objet x à gauche
x = f.pop()         # Enlève l'élément à droite
x = f.popleft()     # Enlève l'élément à gauche
t = len(f)          # Nombre d'éléments de f
```

Toutes ces fonctions sont de complexité O(1).

1.III.2 Parcours en largeur

1.III.2.a Principe

A partir d'un sommet, l'algorithme explore tous ses voisins. Puis, pour chaque voisin, il explore tous ses voisins, et ainsi de suite, jusqu'à un cul de sac ou un sommet déjà exploré. Dans le cas particulier des arbres, il est inutile de marquer les sommets déjà explorés (cela ne peut arriver).

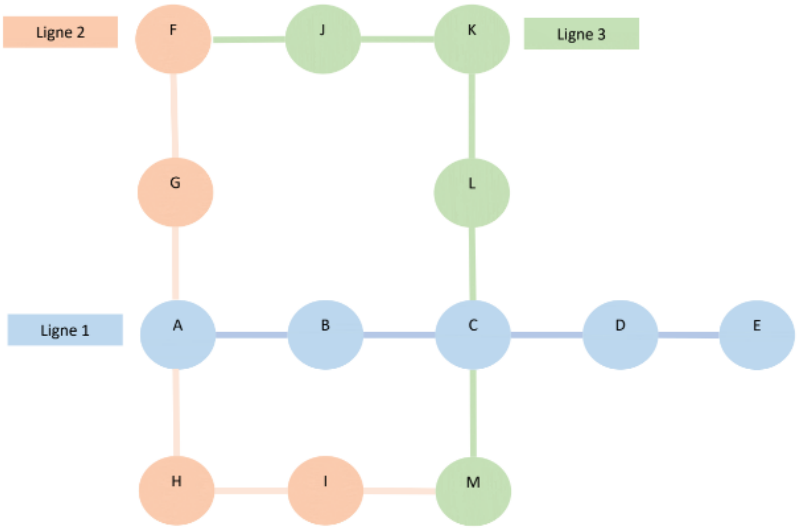
1.III.2.b Algorithme

L'algorithme se déroule donc ainsi :

- Mettre le nœud de départ dans une file et le marquer comme visité
- Tant que la file n'est pas vide :
    - Retirer le premier sommet de la file pour le traiter
    - Mettre tous ses voisins non visités à la fin de la file et les marquer visités

1.III.2.c Exemple

Soit le réseau de stations suivant :



L'exécution de cet algorithme au départ de F ajoute à la liste des stations visitées les stations dans leur ordre de distances (nombre de stations) depuis le départ :

F G J A K B H L C I D M E

On trouve donc tous les sommets accessibles depuis le départ.

Si, pour chaque station visitée, on stocke la distance (nombre de stations) $d[s]+1$, on obtient à la fin le nombre de stations à parcourir pour toutes les stations du graphe depuis le départ.

Visite: F G J A K B H L C I D M E

Distance: 0 1 1 2 2 3 3 4 4 5 5 6

Vous trouverez derrière ce lien une animation (pdf à ouvrir en plein écran) de cette résolution.

1.III.3 Parcours en profondeur

1.III.3.a Principe

Le parcours d'un graphe est réalisé en allant au plus loin dans chaque chemin possible jusqu'à arriver à un cul de sac ou un sommet déjà visité. On revient alors au dernier sommet qui permettait de suivre un autre chemin qui est alors exploré, et ainsi de suite. L'exploration s'arrête lorsque tous les sommets ont été visités. Cette procédure est réalisée de manière récursive.

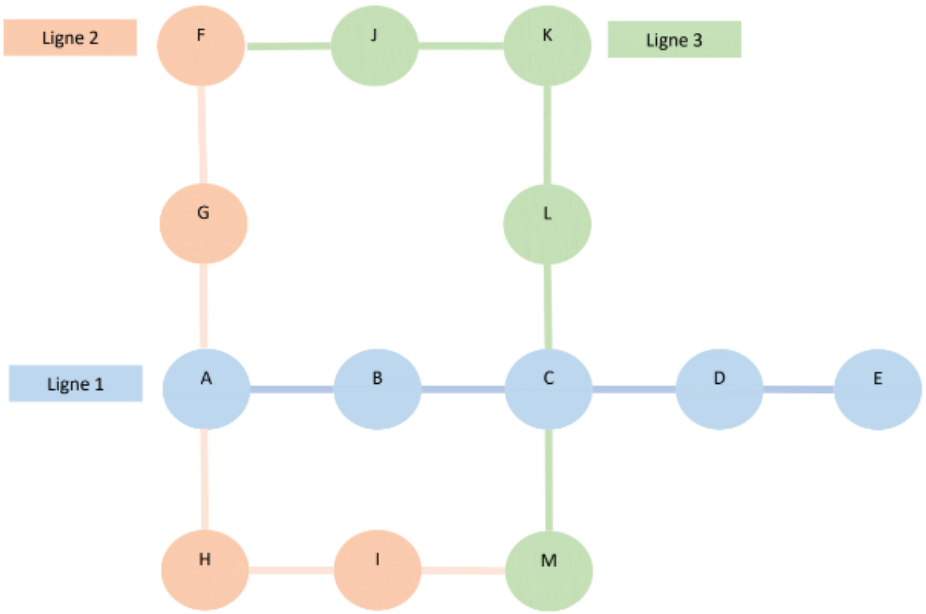
1.III.3.b Algorithme

L'algorithme se déroule donc ainsi :

- Fonction récursive d'exploration Explorer(s) :
    - Marquer le sommet s comme visité
        - Pour tout voisin v de s non marqué :
            - Explorer(v)
- Le parcours total est alors réalisé ainsi :
    - Pour tout sommet s non marqué du graphe :
        - Explorer(s)

1.III.3.c Application

Soit le réseau de stations suivant :



L'exécution de cet algorithme en traitant les stations dans leur ordre d'apparition dans les lignes (ABCDEFGHIMJKL), en partant de A donc, ajoute à la liste des stations visitées les stations ainsi :

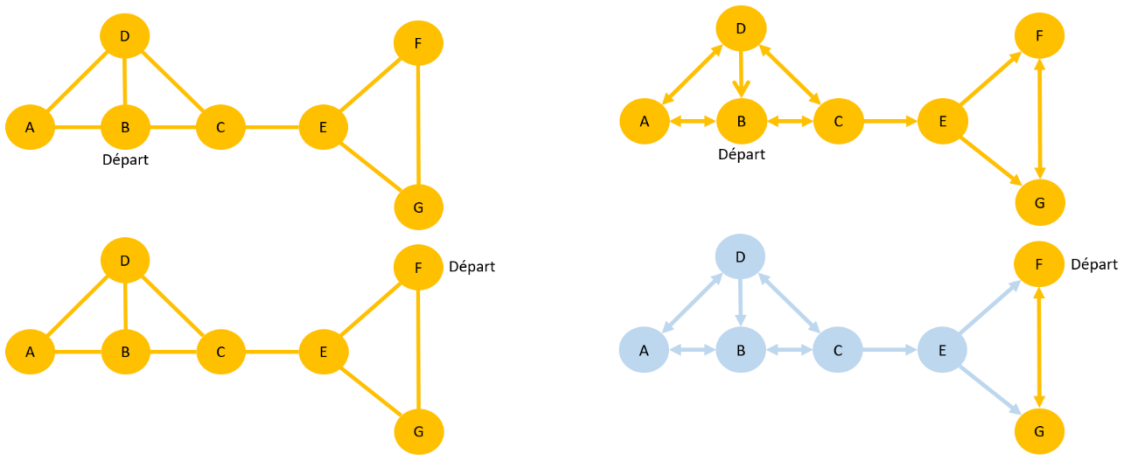
A D B C E M I H L K J F G

Vous trouverez derrière ce lien une animation (pdf à ouvrir en plein écran) de cette résolution (un seul appel de Explorer est illustré, il suffit dans ce cas).

1.III.4 Remarques

1.III.4.a Stations accessibles depuis le départ

Que le graphe soit orienté ou non, le parcours en largeur au départ de s / une seule exécution de la fonction Explorer(s) du parcours en profondeur permet de trouver tous les sommets accessibles depuis s (sommets en orange ci-dessous).

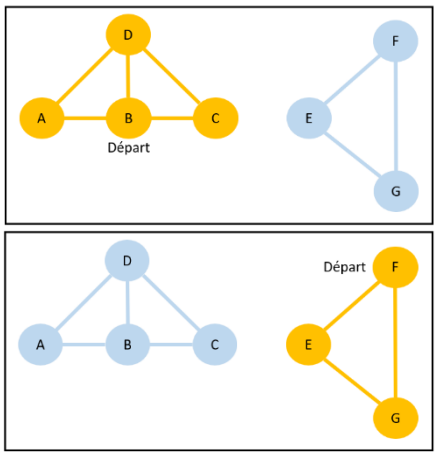


Pour un graphe non orienté connexe, on obtient alors tous les sommets de G.

1.III.4.b Composantes connexes de sous graphes

Un parcours en largeur depuis s / une seule exécution de Explorer(s) pour un graphe non orienté permet de déterminer les composantes connexes (sous graphe connexe), c'est-à-dire toutes les stations reliées ensemble s.

Si le résultat du parcours ne contient pas toutes les stations du graphe, le graphe n'est pas connexe.



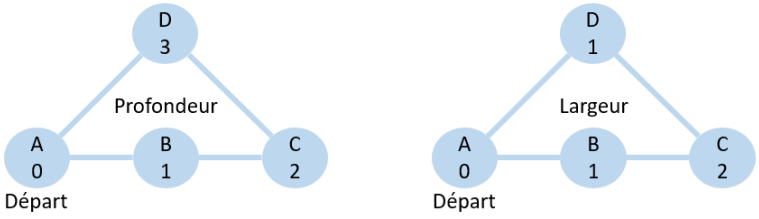
1.III.4.c Choix dans l'algorithme

Selon l'ordre de sélection des voisins, les parcours peuvent se comporter quelque peu différemment.

1.III.4.d Chemin le plus court en nombre de sommets

Dans le cas du parcours en largeur, comme les sommets sont explorés par distance croissante au sommet de départ, on trouve le plus court chemin au sens « nombre de sommets » depuis le départ. L'algorithme de Dijkstra abordé plus tard dans ce cours peut être vu comme une extension de cette méthode pour les graphes pondérés.

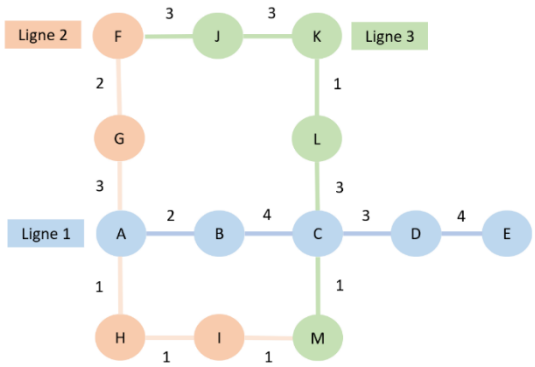
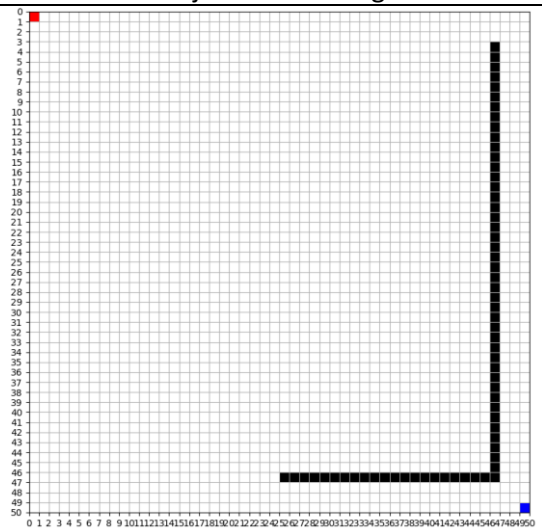
Attention : ce n'est pas vrai pour un parcours en profondeur, exemple sur un cycle :



1.IV. Plus cours chemin d'un graphe pondéré

1.IV.1 Contexte

Nous ferons deux applications de recherche du plus court chemin en TD :

Réseau de transport	Trajet sur une image
	
Dans ce réseau, on souhaite trouver le plus court chemin d'une station de départ (F) à une station d'arrivée (C)	Dans cette image, tout pixel blanc est accessible, tout pixel noir est bloqué, et on souhaite déterminer le plus court chemin du point rouge au point bleu avec les seuls mouvements ←↙↓↘→↗↑↖

1.IV.2 Algorithme de Dijkstra

1.IV.2.a Principe

Soient un graphe pondéré $G(S, A, \omega)$ et les sommets de début s_{deb} et de fin s_{fin}. L'algorithme se déroule ainsi :

- $P = \emptyset$ **ou** $Q = S$
- Initialisation de $Predecesseur$
- $d[s] = +\infty \; \forall s \in S$
- $d[s_{deb}] = 0$
- Tant qu'il existe un sommet hors de P **ou** un sommet dans Q
    - Choisir s_i hors de P tel que $d[s_i] = \min(d[s_k]), s_k \in \bar{P}$
    - $P += s_i$ **ou** $Q -= s_i$
    - Pour chaque sommet $v_j \in \bar{P}$ **ou** $v_j \in Q$ voisin de s_i $(ie. \omega(s_i, v_j) \neq 0)$
        - Si $d[s_i] + \omega(s_i, v_j) < d[v_j]$
            - $d[v_j] = d[s_i] + \omega(s_i, v_j)$
            - $Predecesseur[v_j] = s_i$

      Fin Pour

  Fin Tant que
- Si $d[s_{fin}] \neq 0$
    - Le plus court chemin vaut $d[s_{fin}]$
    - Remonter le chemin à l'envers par succession des prédécesseurs de s_{fin} à s_{deb}

1.IV.2.b Remarques

1.IV.2.b.i Arrêt quand s_{fin} est trouvé

Si l'on souhaite trouver le plus court chemin de s_{deb} à s_{fin} le plus rapidement possible et une seule fois, il est intéressant de stopper l'algorithme dès lors que le sommet s_i choisi est s_{fin}. On pourra alors ajouter la condition $s_i \neq s_{fin}$ à la boucle Tant que.

Si l'on souhaite faire tourner une seule fois l'algorithme au départ de s_{deb} et être ensuite capable de trouver les plus courts chemins de s_{deb} à tous les autres sommets sans le refaire tourner, il faut alors effectivement attendre qu'il n'y ait plus de sommets hors de P / dans Q puis post-traiter les résultats.

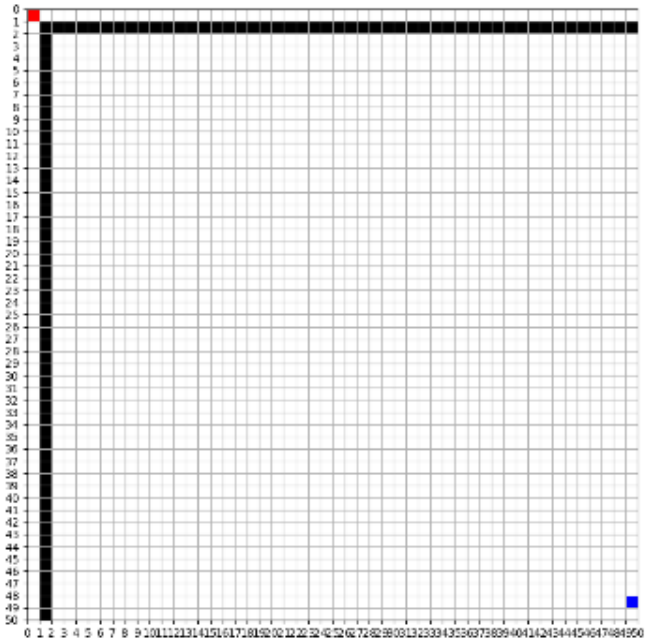
1.IV.2.b.ii Algorithme avec P ou Q

Il peut être légèrement plus intéressant de travailler avec P (se rempli) qu'avec Q (se vide) :

- Si on va jusqu'à la condition « tant qu'il existe des sommets hors de P » ou « tant qu'il existe des sommets dans Q », les temps seront équivalents.
- Si on met en place une condition d'arrêt quand on trouve s_{fin}, il devient intéressant de chercher dans P dès le départ.

1.IV.2.b.iii Arrivée non accessible

Il est possible d'améliorer légèrement cet algorithme dans le cas où s_{fin} n'est pas accessible depuis s_{deb} (ex. ci-dessous). En effet, si au cours de la résolution, $d[s_i] = \infty$, c'est que parmi tous les sommets s hors de P / dans Q restants, aucun n'a une distance finie, et n'est donc pas voisin des sommets déjà traités. L'algorithme va passer en revue inutilement tous les sommets restants, tous inaccessibles. On pourra donc ajouter la condition $d[s_i] \neq \infty$ à la boucle Tant que.



1.IV.2.b.iv Recherche des voisins

La recherche des voisins d'un sommet dans $\bar{P}$ ou dans Q peut être réalisée au préalable en créant une liste d'adjacente, ou alors dans la boucle de résolution à l'aide d'une matrice d'adjacence. Quel que soit l'endroit où elle est placée, elle présentera un même cout de calculs.

1.IV.2.b.v Influence de la programmation

Selon la programmation, l'algorithme peut choisir différents chemins de même distance. Cela se passe à deux endroits :

- Choix de s : selon le minimum retenu (premier ou dernier minimum)
- Inégalité stricte ou non dans le test $d[s_i] + \omega(s_i, v_j) < d[v_j]$

1.IV.2.b.vi Version avec une pile

Il est possible de réaliser une version de l'algorithme de Dijkstra qui améliore son temps d'exécution avec un objet (pile, liste, file) contenant :

- Aucun sommet traité
- Tous les sommets voisins non traités des sommets déjà traités (pour résumer : les prochains sommets potentiellement choisis)

Cela évite de rechercher le minimum parmi tous les sommets dans $\bar{P}$ ou dans Q et limite la recherche à un objet de petite taille.

L'algorithme prend alors cette tournure :

- Initialisation de la file $F = \emptyset$
- Initialisation des sommets traités $T = \emptyset$
- Initialisation de $Predecesseur$
- $F \mathrel{+}= s_{deb}$
- $d[s] = +\infty \; \forall s \in S$
- $d[s_{deb}] = 0$
- Tant que $F \neq \emptyset$
    - Choisir s_i dans F tel que $d[s_i] = \min(d[s_k])\,, s_k \in F$
    - $F \mathrel{-}= s_i$
    - Pour chaque sommet $v_j \in S$ voisin de s_i $\left(ie. \omega(s_i, v_j) \neq 0\right)$
        - Si $v_j \notin T$: **(*)**
            - Si $v_j \notin F$:
                - $F \mathrel{+}= v_j$
            - Si $d[s_i] + \omega(s_i, v_j) < d[v_j]$
                - $d[v_j] = d[s_i] + \omega(s_i, v_j)$
                - $Predecesseur[v_j] = s_i$

      Fin Pour

  Fin Tant que
- Si $d[s_{fin}] \neq 0$
    - Le plus court chemin vaut $d[s_{fin}]$
    - Remonter le chemin à l'envers par succession des prédécesseurs de s_{fin} à s_{deb}

Il reste toutefois la recherche des sommets voisins $\underline{v_j}$ de $\underline{s_i}$ non encore traités ($\underline{v_j \in \bar{P}}$ **ou** $\underline{v_j \in Q}$), qui coûte **(*)**. Et là, il est encore possible de gagner en temps, en n'excluant pas les voisins traités (en traitant donc tous les voisins $\underline{v_j \text{ de } s_i}$). En effet, lors du traitement de $\underline{s_i}$, tout sommet déjà traité s_j avec l'algorithme de Dijkstra possède une distance $d[s_j]$ plus faible que $d[s_i]$, et donc plus faible que $d[s_i] + \omega(s_i, s_j)$. Sa distance ne sera donc jamais mise à jour.

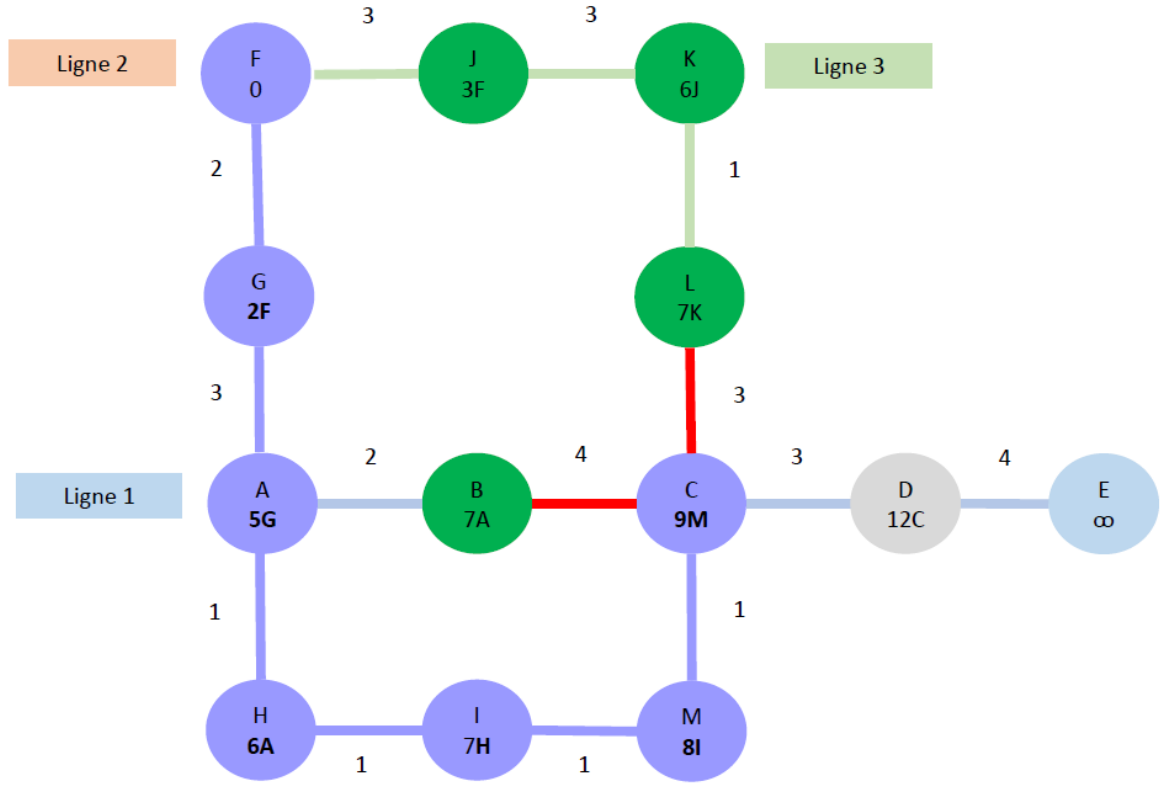
1.IV.2.c Illustration

Voici un lien vers une vidéo et un lien vers un document pdf (à utiliser en mode plein écran) qui illustrent l'exécution de l'algorithme de Dijkstra sur l'exemple du réseau de transport proposé en introduction de cette partie.

Remarques :

- J'ai intégré l'arrêt de l'algorithme quand l'arrivée C est trouvée
- Lorsqu'il y a des sommets exæquo dans le choix de s, c'est le premier par ordre d'apparition dans la liste des stations [ABCDEFGHIMJKL] qui est retenu (premier minimum)
- Je garde l'inégalité stricte dans le test $d[s_i] + \omega(s_i, v_j) < d[v_j]$

Voici le résultat :



FGAHIMC

A	B	C	D	E	F	G	H	I	M	J	K	L
5	7	9	12	∞	0	2	6	7	8	3	6	7
G	A	M	C			F	A	H	I	F	J	K

1.IV.3 Algorithme A star

1.IV.3.a Heuristique

L'algorithme A*, A étoile ou A star est une évolution de l'algorithme de Dijkstra où on introduit une fonction telle que :

$$f(s) = g(s) + h(s)$$

Avec :

- $g(s)$ le coût réel du chemin optimal partant du sommet initial jusqu'à s
- $h(s)$ le coût estimé du reste du chemin partant de n jusqu'à un état satisfaisant du but

$h(s)$ est une heuristique.

1.IV.3.b Principe

Dans des applications de recherche de chemin sur une image par exemple, il est possible de connaître pour chaque sommet la distance « à vol d'oiseau » de chaque sommet à l'arrivée. On utilise donc cette heuristique et, au lieu de choisir le sommet ayant la plus petite distance depuis le départ parmi les sommets hors de P / dans Q, on utilise la fonction suivante :

$$f(s) = d(s) + d'(s, s_{fin})$$

Avec :

- $d(s)$ la distance depuis le départ actuellement déterminée par l'algorithme pour le sommet s
- $d'(s, s_{fin})$ la distance à vol d'oiseau entre s et s_{fin}.

On va alors dans l'algorithme de Dijkstra remplacer la ligne :

- Choisir s hors de P tel que $d[s] = \min(\boldsymbol{d}[s_i]), s_i \in \bar{P}$

Par :

- Choisir s hors de P tel que $d[s] = \min(\boldsymbol{f}[s_i]), s_i \in \bar{P}$

Lorsque le chemin direct entre s et s_{fin} présente peu d'obstacle, A* est très intéressant.

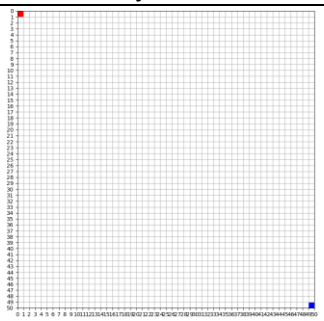
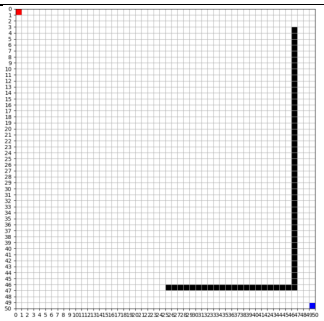
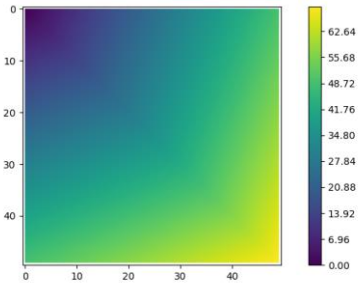
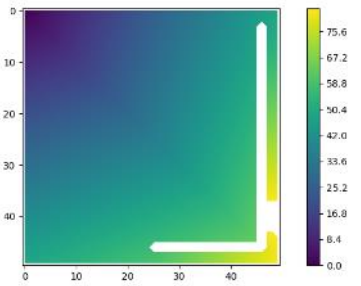
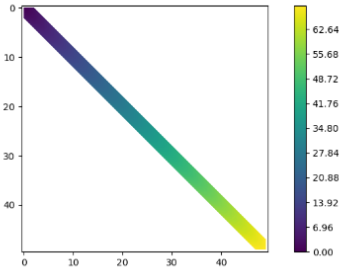
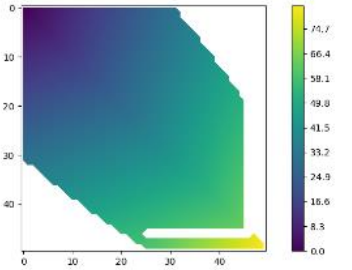
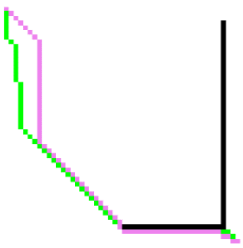
1.IV.3.c Remarque

Comme pour Dijkstra, cet algorithme peut utiliser $\bar{P}$, Q ou une file F.

Je ne peux garantir que l'inclusion des voisins de la version avec file de Dijkstra (cf **(*)**) fonctionne toujours selon l'heuristique choisie, mais pour l'avoir essayée dans le TD sur la recherche de distance sur une image comme proposé ci-dessus, elle est très efficace (quel que soit le cas ?).

1.IV.4 Comparaison Dijkstra/Astar

Voici deux exemples de résolution du plus court chemin dans une image où l'on affiche dans les vidéos en rouge les pixels/cases choisis comme s par l'algorithme :

	Trajet libre	Obstacle
Grille de départ 50x50 *Départ rouge* *Arrivée bleue*		
Dijkstra *Echelle de couleur : distance depuis le départ des sommets calculés*	Vidéo  Itérations : 2500	Vidéo  Itérations : 2411
A* *Echelle de couleur : distance depuis le départ des sommets calculés*	Vidéo  Itérations : 50	Vidéo  Itérations : 1819
Trajets **Dijkstra A***	 Distance : 69.29	 Distance : 82.18

Remarque : Pour ces deux algorithmes, les distances sont comptées comme somme de segments horizontaux, verticaux et en diagonale. Ainsi, on le voit bien dans la vidéo de Dijkstra, l'avancée à plus courte distance du départ ne se produit pas en arc de cercles, mais en octogones, c'est malheureusement normal quand on considère que les seuls mouvements possibles sont ←↙↓↘→↗↑↖

1.V. En deuxième année

J'ai illustré tout ce cours sur des exemples de simples, avec des graphes dont la matrice d'adjacence est facile à réaliser. En pratique, on ne connait les graphes à l'avance, et l'on procédé généralement de la sorte :

- Fonction qui détermine touts les successeurs (voisins) d'un sommet
- Mise en place d'un parcours en largeur ou profondeur afin d'explorer tout le graphe

En deuxième année, nous utiliserons cette procédure afin d'étudier les jeux. Voici deux exemples de TD que vous réaliserez probablement :

Jeu « Le compte est bon »	Jeu de babylon