éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

PLUS COURT CHEMIN DANS UN GRAPHE

- ▷ Histoire de l'informatique
- ▷ Structures de données
- ▷ Bases de données
- ▷ Architectures matérielles et systèmes d'exploitation
- ▷ Langages et programmation
- ▷ Algorithmique

Ce document s'appuie largement sur la page `http://mpechaud.fr/scripts/parcours/index.html` qui a permis en particulier de générer certaines figures de ce document. L'exploration du site `http://mpechaud.fr` est recommandée de façon plus générale : c'est une mine !
Ce document présente trois approches de la recherche du plus court chemin d'une source à un but dans un graphe : il se veut un support pour le professeur mais ne l'enjoint pas à présenter systématiquement les trois approches en classe.

1. Généralités

Dans tout ce document, G désigne un graphe **non orienté, connexe** (c'est-à-dire tel que deux sommets quelconques sont reliés par un chemin), et **pondéré** (c'est-à-dire que les arêtes sont accompagnées d'un *poids*, entier ou flottant **positif**). En particulier, on pourra considérer les graphes qui simulent un réseau routier, les poids étant les distances kilométriques des trajets entre les différents points du réseau. On considère spécialement des *graphes euclidiens*, c'est-à-dire des graphes où les poids des arêtes sont les distances (euclidiennes) entre les points, à vol d'oiseau.
Dans tous les cas, il importe ici que tous les poids soient positifs [1].
Dans un tel graphe, le poids d'un chemin est la somme des poids des arêtes qui le constituent.
On fixe un point de départ et un point d'arrivée parmi les sommets du graphe G et on cherche un chemin de poids minimal qui les relie, qu'on appelle un *plus court chemin*, par exemple pour les graphes euclidiens. Il n'y a en général pas unicité de la solution : pour s'en convaincre, il suffit de considérer un graphe euclidien dont les sommets et les arêtes forment un carré $ABCD$, pour lequel il y a clairement deux plus courts chemins entre deux sommets diagonalement opposés. Néanmoins, on rencontre souvent (et on pardonne volontiers) l'abus de langage consistant à parler *du* plus court chemin, comme s'il n'y en avait qu'un.

1.1. Notations

Dans la suite, on considère un graphe non orienté connexe G possédant n sommets et p arêtes. On numérote les sommets de 0 à $n-1$ avec la convention que 0 est le numéro du sommet de départ et $n-1$ celui du sommet d'arrivée pour lesquels on cherche un plus court chemin. Par abus de langage, on dira simplement *le sommet* k pour *le sommet de numéro* k.
On représente le graphe par une liste de listes d'adjacences `adj` telle que `adj[k]` est la liste des sommets voisins du sommet `k`.

1. sinon, la notion de plus court chemin risque de n'avoir plus aucun sens, le parcours répété d'un cycle de poids négatif permettant de réduire indéfiniment le coût total d'un chemin

Les poids (ou longueurs) des arêtes sont les éléments d'une matrice `poids` telle que `poids[i][j]` est le poids de l'arête qui lie les sommets `i` et `j`. Cette matrice est symétrique : `poids[i][j] == poids[j][i]` puisqu'il s'agit de la même arête. Dans le cas où les sommets `i` et `j` ne sont pas voisins, on utilisera `poids[i][j] = inf` qui est la valeur constante représentant l'infini qu'on peut importer du module math par la commande `from math import inf`.
Dans le cadre d'un graphe euclidien, on fournira les coordonnées cartésiennes des sommets par l'intermédiaire d'une liste XY de couples de réels (x_i, y_i). On pourra alors initialiser le tableau des poids de la façon suivante.

```
from math import inf,sqrt

# distance euclidienne entre deux points représentés par des couples de coordonnées
def d(p1, p2):
    return sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

# calcul de la matrice des poids des arêtes pour un graphe euclidien
def calcule_poids(adj, XY):
    assert len(adj) == len(XY)
    n = len(adj)
    poids = [[inf] * n for i in range(n)]
    for i in range(n):
        poids[i][i] = 0
    for i in range(n - 1):
        for j in adj[i]:
            # on ne considère que les arêtes présentes dans le graphe
            s1 = XY[i]
            s2 = XY[j]
            poids[i][j] = poids[j][i] = d(s1, s2)
    return poids
```

1.2. Différentes façons d'évaluer le poids d'un chemin

La figure 1 représente un graphe euclidien. On a tracé une grille dont les carreaux sont de côté unité pour permettre d'estimer les longueurs. Chaque arête est étiquetée par un poids entier (qui n'est pas la distance euclidienne).
On vérifie que, pour ce graphe, le chemin le plus court est :

- le trajet rouge pour la distance euclidienne ;
- le trajet vert pour la somme des poids des arêtes ;
- le trajet bleu si l'on compte le nombre d'étapes dans le trajet.

Ainsi, pour la distance euclidienne, l'ordre (du plus court au plus long) des trajets est : rouge (6,24), vert (7,82), bleu (8,08) ; pour les poids marqués : vert (5), bleu (6), rouge (8) ; pour le nombre d'étapes : bleu (2), rouge (3), vert (4).

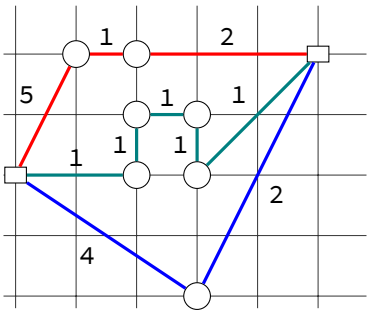

FIGURE 1 – trois évaluations des chemins sur un même graphe

2. Quelques exemples de graphes

On décrit ici quelques graphes qui serviront dans la suite de ce document

Un premier graphe euclidien

Ce premier graphe comporte 12 sommets et 17 arêtes.

```
n = 12

# on donne un nom à chaque sommet
etiquettes = [chr(ord('A') + i) for i in range(n)]

# listes d'adjacence
adj = [[1, 6], [0, 2, 3], [1, 3, 9], [1, 2, 4], [3, 5], [4, 11],
       [0, 7, 8], [6, 8], [6, 7, 9, 10], [2, 8, 10, 11],
       [8, 9, 11], [5, 9, 10]]

# coordonnées des sommets
XY = [(0, 2), (0, 3), (1, 2), (2, 3), (2, 4), (4, 4),
      (0, 1), (1, 0), (2, 0), (3, 2), (4, 1), (4, 3)]

poids = calcule_poids(adj, XY)
```

L'objectif est de déterminer les chemins du sommet A au sommet L.

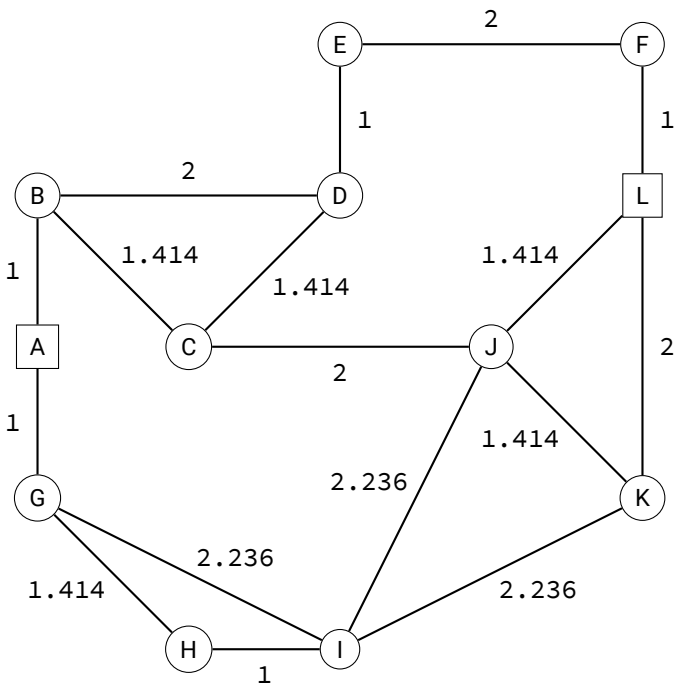

FIGURE 2 – un premier graphe

Un deuxième graphe euclidien

Ce deuxième graphe comporte 15 sommets et 15 arêtes.

```
n = 15

# on donne un nom à chaque sommet
etiquettes = [chr(ord('A') + i) for i in range(n)]

# listes d'adjacence
adj = [[1, 12], [0, 2], [1, 3], [2, 4], [3, 5], [4, 6], [5, 7], [6, 8],
       [7, 9], [8, 10], [9, 11], [10, 14], [0, 13], [12, 14], [11, 13]]

# coordonnées des sommets
XY = [(0 ,1), (1, 1), (1, 0), (2, 0), (2, 1), (3, 1), (3, 0), (4, 0), (4, 1),
    (5, 1), (5, 0), (6, 0), (0, 3), (6, 3), (6, 1)]

poids = calcule_poids(adj, XY)
```

L'objectif est de déterminer les chemins du sommet A au sommet O.

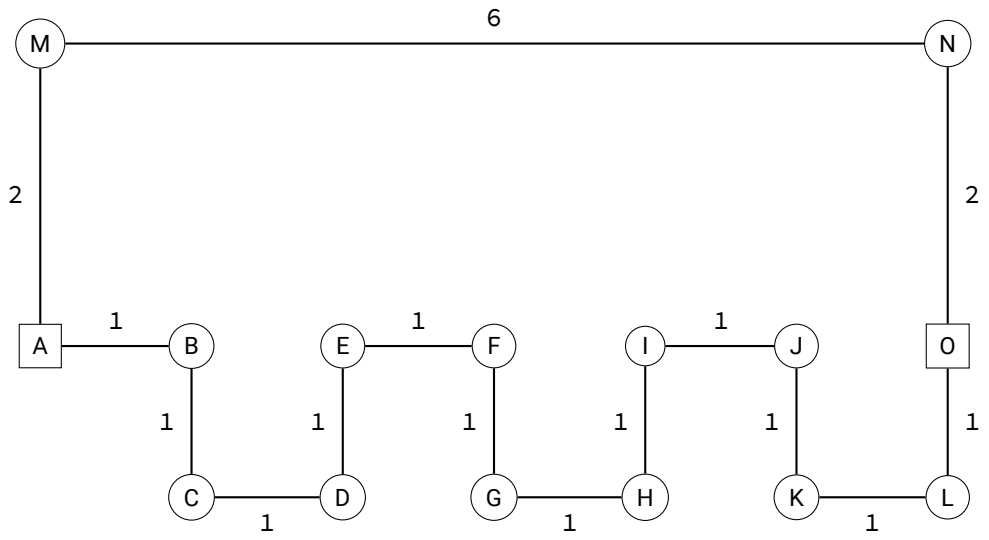

FIGURE 3 – un graphe à créneaux

Un troisième graphe euclidien

Ce troisième graphe comporte 9 sommets et 9 arêtes.

```
n = 9

# on donne un nom à chaque sommet
etiquettes = [chr(ord('A') + i) for i in range(n)]

# listes d'adjacence
adj = [[1, 5], [0, 2], [1, 3], [2, 4], [3, 8], [0, 6], [5, 7], [6, 8], [4, 7]]

# coordonnées des sommets
XY = [(1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (0, 0), (0, 1), (6, 1), (6, 0)]

poids = calcule_poids(adj, XY)
```

L'objectif est de déterminer les chemins du sommet A au sommet L.

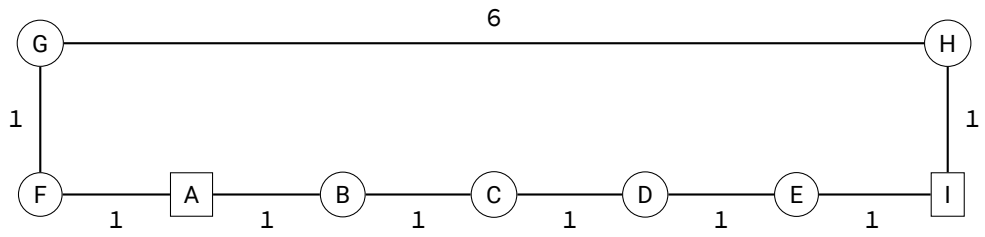

FIGURE 4 – un troisième graphe

3. Parcours en largeur

Dans un parcours en largeur, on visite d'abord le sommet origine, puis la génération des sommets qui sont à distance 1 (en nombre d'arêtes), puis la génération des sommets qui sont à distance 2, etc. Quant on visite le point d'arrivée, le numéro de sa génération représente le nombre d'arêtes qui le séparent du point de départ.
Autrement dit, un parcours en largeur à partir du sommet d'origine permet de trouver un plus court chemin vers le sommet d'arrivée en nombre d'étapes (c'est-à-dire d'arêtes traversées).
Si on prend la précaution de noter (dans un tableau `provenance`) le père de chaque génération de sommets, il suffit de remonter de père en père pour reconstituer le chemin d'accès du point d'arrivée.

```
# reconstitution du chemin à partir du tableau des provenances
def decrit_chemin(etiquettes, provenance):
    assert len(etiquettes) == len(provenance)
    n = len(etiquettes)
    s = ""
    k = n - 1
    while k != 0:
        s = ' ' + etiquettes[k] + s
        k = provenance[k]
    return etiquettes[0] + s
```

On va effectuer un parcours en largeur du graphe, en colorant les sommets au passage : les sommets non explorés en blanc, les sommets explorés en rouge et les prochains sommets à explorer en bleu.
On renvoie un tableau de provenance : pour chaque sommet ajouté à la file (en bleu), on retient ainsi le sommet père, permettant de reconstituer le chemin.

```
BLANC = 0   # couleur des sommets non explorés
BLEU = 1    # couleur des sommets à explorer (la frontière)
ROUGE = 2   # couleur des sommets explorés

# parcours en largeur
from collections import deque

def parcours_largeur(adj):
    n = len(adj)
    provenance = [0] * n
    couleur = [BLANC] * n
    couleur[0] = BLEU
    file = deque([0])
    while couleur[n - 1] != ROUGE:
        k = file.popleft()
        couleur[k] = ROUGE
        for j in adj[k]:
            if couleur[j] == BLANC:
                couleur[j] = BLEU
                provenance[j] = k
                file.append(j)
    return provenance
```

On a utilisé le module `deque` pour implémenter une file d'attente, qui répond à la sémantique FIFO *(firt in first out)* : ce sont les éléments ajoutés en premier qui sont retirés. On assure ainsi qu'on termine de visiter tous les voisins d'un sommet avant d'envisager un autre sommet. C'est bien la sémantique d'un parcours en largeur.
Observons ce que renvoie l'appel `decrit_chemin(etiquettes, parcours_largeur(adj))` sur nos trois exemples de graphe :

- ▷ pour le graphe de la figure 2, l'appel renvoie `A B C J L`, qui se trouve être également le chemin le plus court (au sens de la distance euclidienne) ;
- ▷ pour le graphe de la figure 3, l'appel renvoie `A M N O`, qui se trouve être également le chemin le plus court (au sens de la distance euclidienne) ;
- ▷ pour le graphe de la figure 4, l'appel renvoie `A F G H I`, qui comporte certes moins d'étapes que le chemin le plus court.

4. Parcours best-first

L'algorithme *best-first* n'utilise pas une file d'attente comme l'algorithme de parcours en largeur. Parmi les sommets bleus, il choisit non pas le plus anciennement introduit dans la file, mais celui qui est le plus proche, à vol d'oiseau, du sommet d'arrivée.
On parle d'algorithme informé car il utilise une heuristique simple, reposant sur le calcul de distance pour effectuer ses choix.
Pour ce faire, on a écrit une fonction `extrait_plus_proche` qui réalise une recherche de minimum dans la liste des sommets bleus. Une fois trouvé ce minimum, il le supprime de la liste avant de renvoyer le numéro du sommet bleu correspondant.

```
def extrait_plus_proche(cible, liste, XY):
    xy_cible = XY[cible]
    k = liste[0]
    mini = d(XY[k], xy_cible)
    for j in liste[1:]:
        dist = d(XY[j], xy_cible)
        if dist < mini:
            k = j
            mini = dist
    liste.remove(k)
    return k

def best_first(adj, XY):
    assert len(adj) == len(XY)
    n = len(adj)
    provenance = [0] * n
    couleur = [BLANC] * n
    couleur[0] = BLEU
    bleus = [0]
    while couleur[n - 1] != ROUGE:
        k = extrait_plus_proche(n - 1, bleus, XY)
        couleur[k] = ROUGE
        for j in adj[k]:
            if couleur[j] == BLANC:
                couleur[j] = BLEU
                provenance[j] = k
                bleus.append(j)
    return provenance
```

Observons ce que renvoie l'appel `decrit_chemin(etiquettes, best_first(adj, XY))` sur nos trois exemples de graphe :

- pour le graphe de la figure 2, l'appel renvoie `A B D E F L` qui n'est pas optimal pour la distance euclidienne ;
- pour le graphe de la figure 3, l'appel renvoie `A B C D E F G H I J K L O` : l'algorithme s'est laissé piéger par la configuration particulière du graphe ;
- pour le graphe de la figure 4, l'appel renvoie `A B C D E I`, qui est bien le chemin le plus court pour la distance euclidienne.

L'algorithme *best-first* est très rapide, car il s'appuie sur une heuristique simple. Cependant, il échoue à trouver un chemin optimal au sens de la distance euclidienne.
Dans le cas d'un réseau routier dense (c'est le cas du réseau français) il reste très efficace, s'approcher le plus vite possible de la cible étant souvent la meilleure stratégie, et la configuration particulière de notre deuxième exemple de graphe n'a pas vraiment d'analogue dans la réalité.

5. Algorithme de Dijkstra

5.1. Présentation de l'algorithme

L'algorithme de Dijkstra opère sur un graphe connexe pondéré, pas nécessairement euclidien. Nous en détaillons le fonctionnement sur l'exemple volontairement simple du graphe de la figure 5.

L'algorithme met à jour une table des poids estimés des plus courts chemins entre chaque sommet et le sommet de départ. Les sommets que nous colorions en bleu, c'est-à-dire les sommets de la frontière de notre parcours de graphe, sont ici rangés dans une file de priorité, qui est une structure de données d'où l'on peut extraire l'élément de poids minimal.

La table qui suit la figure 5 montre les états successifs de la table et de la file de priorité au fur et à mesure du déroulement de l'algorithme. L'élément souligné de la file de priorité est celui, de poids minimal, qui en est extrait.

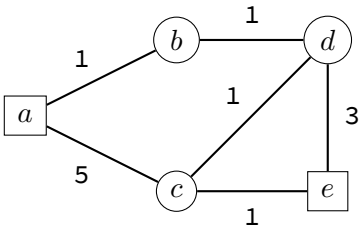

FIGURE 5 – un petit graphe illustrant l'algorithme de Dijkstra

a	b	c	d	e	file de priorité
0	∞	∞	∞	∞	$\underline{a}$
0	1	5	∞	∞	$\underline{b}\,c$
0	1	5	2	∞	$c\,\underline{d}$
0	1	3	2	5	$\underline{c}\,e$
0	1	3	2	4	$\underline{e}$

TABLE 1 – les étapes successives

On démarre avec des poids estimés infinis sauf, bien sûr, pour le sommet de départ (le chemin de poids minimal de a vers a est évidemment de poids nul).

On extrait donc a de la file de priorité, ses voisins sont b et c (qui sont ajoutés à la file de priorité). On en profite pour mettre à jour les poids estimés de ces deux sommets : pour l'instant on retient le poids des arêtes ab et ac, à savoir 1 et 5. On va donc ensuite extraire b de la file de priorité, puisque c'est le sommet de la file de poids minimal.

À l'étape suivante, on examine les arêtes ba et bd. L'arête ba n'améliore pas le trajet de a à a (le chemin aba est plus coûteux que le fait de rester sur a), l'arête bd permet d'estimer le poids du trajet vers d à $1 + 1 = 2$ (poids estimé du trajet vers b augmenté du poids de l'arête bd). On a ajouté d à la file, avec un poids estimé égal à 2, on n'a pas modifié le poids de a, et on extrait maintenant d, de poids minimal dans la file.

On examine maintenant les arêtes db, dc et de. L'arête db n'améliore pas le poids estimé vers b (le trajet $abdb$ est plus coûteux que le trajet abd), l'arête de fournit une première estimation, égale à $2 + 3 = 5$ du poids du trajet vers e. En revanche l'arête dc améliore le coût estimé d'un trajet vers c, qu'on ramène à $2 + 1 = 3$ au lieu de 5. Ce sera maintenant au tour de c, de poids minimal 3, à être extrait de la file.

On examine donc les arêtes ca, cd et ce. On vérifie que seul le poids estimé de e doit être mis à jour à la valeur $3 + 1 = 4$, meilleure que le poids estimé précédent.

L'extraction de e ne modifie plus rien, la file est vide, le déroulement de l'algorithme se termine.

5.2. Implémentation

On retient l'information sur les poids estimés des plus courts trajets dans un tableau `poids_est` qui est initialisé en lignes 7 et 8 avec une valeur infinie sauf bien sûr pour le sommet initial, d'indice 0, dont le poids estimé vaut 0.
Une structure qui permet l'ajout d'un élément et l'extraction de l'élément de valeur minimale s'appelle une file de priorité. Ici nous avons besoin en outre de pouvoir modifier la valeur associée à un élément présent dans la file : c'est pourquoi nous utilisons le module `heapdict`[2] car le module `heapq`, d'un usage pourtant plus fréquent, ne permet pas la mise à jour de la valeur associée à une clé.
Cette file de priorité `poids_fp` est initialisée en ligne 9.
Notons que la ligne 10, en renseignant `poids_fp[0]`, ajoute *de facto* le sommet 0 et la valeur associée 0 à la file initialement vide.
En ligne 12, on récupère le couple `(k,v)` (k est le numéro du sommet dont la valeur v est minimale) dont on ne conserve que la première coordonnée, c'est-à-dire l'indice k du sommet.
En ligne 15, on vérifie si une amélioration de l'estimation du chemin optimal est possible.
Enfin, la ligne 18 réalise deux opérations différentes :

- si le sommet d'indice `j` n'est pas présent dans la file de priorité, on l'y insère avec comme valeur associée le poids estimé ;
- sinon, la même instruction réalise une mise à jour de la valeur d'un sommet déjà présent.

On remarque qu'on garantit ainsi que la valeur associée à un sommet `j` présent dans la file est toujours égale à `poids_est[j]`.

```
1  import heapdict
2
3  def dijkstra(adj, poids):
4      assert len(adj) == len(poids)
5      n = len(adj)
6      provenance = [0] * n                          # couleur = [BLANC] * n
7      poids_est = [inf] * n
8      poids_est[0] = 0
9      poids_fp = heapdict.heapdict()
10     poids_fp[0] = 0                               # couleur[0] = BLEU
11     while len(poids_fp) > 0:
12         k = poids_fp.popitem()[0]                 # couleur[k] = ROUGE
13         for j in adj[k]:
14             d = poids_est[k] + poids[k][j]
15             if d < poids_est[j]:
16                 provenance[j] = k
17                 poids_est[j] = d
18                 poids_fp[j] = d                   # couleur[j] = BLEU
19     return provenance
```

Remarque : on n'a pas utilisé les couleurs, la gestion de la file de priorité faisant le travail, mais on a laissé en commentaires ce qui aurait pu être fait (qui ne change rien à l'algorithme), ce qui sera utile pour l'étude de l'algorithme.

Observons ce que renvoie l'appel `decrit_chemin(etiquettes, dijkstra(adj, poids))` sur nos trois exemples de graphe :

- pour le graphe de la figure 2, l'appel renvoie un chemin optimal : `A B C J L` ;
- pour le graphe de la figure 3, l'appel renvoie un chemin optimal : `A M N O` ;
- pour le graphe de la figure 4, l'appel renvoie un chemin optimal : `A B C D E I`.

2. on peut l'installer via la commande `pip install heapdict`

5.3. Correction de l'algorithme

La preuve de correction est plutôt difficile, elle ne constitue pas un attendu du programme.
Notons, pour tout sommet k, $\lambda(k)$ le poids d'un plus court chemin de l'origine à ce sommet.
Donnons en une idée, en utilisant la coloration des sommets que nous avons explicitée en commentaires.
Il s'agit de démontrer que, pour les sommets k colorés en rouge, `poids_est[k]` est égal au poids du plus court chemin de l'origine au sommet considéré, c'est-à-dire à $\lambda(k)$.
Autrement dit, il s'agit des sommets extraits tour à tour de la file de priorité, pour lesquels l'analyse est donc complète.
La connexité du graphe entraîne que tous les sommets arriveront tôt ou tard dans la file de priorité. La boucle principale commence par l'extraction d'un sommet de la file de priorité. Ainsi peut-on affirmer qu'au bout du compte tous les sommets auront été colorés en rouge. La terminaison est garantie car le nombre de sommets rouges croît strictement à chaque passage dans la boucle (et est bien sûr borné par le nombre total de sommets).

Toute la difficulté consiste donc à montrer que, pour un sommet rouge k, `poids_est[k]` est égal à $\lambda(k)$.
Considérons le moment où on extrait un sommet k de la file de priorité et considérons alors un chemin de poids minimal de l'origine à k, dont le poids total vaut donc $\lambda(k)$, et qui passe par différents sommets. Soit a le dernier sommet de ce chemin à être coloré en rouge, et b son voisin : l'arête (a, b) est l'arête qui traverse la frontière. Notons x son poids.
On $\lambda(k) = \lambda(a) + x + ?$, le point d'interrogation désignant le poids (positif) de la partie du chemin entre b et k. Retenons simplement que $\lambda(k) \geqslant \lambda(a) + x$.
Comme a est coloré en rouge, ses voisins ont été visités dans la boucle, dont b en particulier. Au moment où on a extrait a de la file de priorité, on avait `poids_est[a]` égal à $\lambda(a)$ et donc `poids_est[b]` est en particulier inférieur ou égal à la somme $\lambda(a) + x$ puisqu'il a été modifié si besoin par la conditionnelle de la ligne 15.
Mais k n'est pas quelconque : parmi les éléments de la file, il est de poids estimé minimal. Donc `poids_est[b]` est supérieur ou égal à `poids_est[k]`.
Finalement `poids_est[k]` est inférieur ou égal à $\lambda(a) + x$ lui-même plus petit que $\lambda(k)$, comme on l'a dit plus haut.
Mais cela signifie qu'on a égalité, d'après la définition de $\lambda(k)$. Autrement dit on a bien prouvé que `poids_est[k]` est égal à $\lambda(k)$ au moment où on colorie k en rouge. [3]

5.4. Complexité

Le module `heapdict` assure que les opérations élémentaires sur la file de priorité (extraction de la clé minimale, ajout ou mise à jour) ont un coût de l'ordre de la quantité $\log(n)$ où n est le nombre de sommets.
On montre alors que la complexité de l'algorithme de Dijkstra, pour un graphe de n sommets et p arêtes, est de l'ordre de $(p + n)\log(n)$.

3. On peut même en déduire qu'en réalité $b = k$.

6. L'algorithme A*

L'algorithme A* s'inspire à la fois des algorithmes de Dijkstra et best-first.
En s'appuyant sur l'algorithme de Dijkstra il garantit l'optimalité du chemin trouvé, ce que ne sait pas faire best-first.
En s'appuyant sur best-first, il fait une recherche *informée*, pour accélérer la recherche du sommet cible.
Alors que la réponse de l'algorithme de Dijkstra permet de trouver un plus court chemin de l'origine à n'importe quel sommet, l'algorithme A* ne s'intéresse qu'à un plus court chemin de l'origine à un seul sommet cible.

```
1  import heapdict
2
3  def Astar(adj, XY):
4      assert len(adj) == len(XY)
5      n = len(adj)
6      poids = calcule_poids(adj, XY)
7      provenance = [0] * n
8      poids_est = [inf] * n
9      poids_est[0] = 0
10     poids_fp = heapdict.heapdict()
11     poids_fp[0] = d(XY[0], XY[n - 1])
12     while len(poids_fp) > 0:
13         k = poids_fp.popitem()[0]
14         if k == n - 1:
15             break
16         for j in adj[k]:
17             d1 = poids_est[k] + poids[k][j]
18             d2 = d1 + d(XY[j], XY[n - 1])
19             if d1 < poids_est[j]:
20                 provenance[j] = k
21                 poids_est[j] = d1
22                 poids_fp[j] = d2
23     return provenance
```

La table `poids_est` continue à retenir une estimation du coût d'un chemin depuis l'origine jusqu'au sommet considéré, mais la file de priorité retient une estimation du coût d'un chemin passant par le sommet considéré k et allant jusqu'au sommet cible, en ajoutant la distance euclidienne au sommet cible.
La différence avec Dijkstra se lit

- en ligne 11, où la valeur associée au sommet de départ dans la file de priorité est sa distance euclidienne au sommet cible ;
- en ligne 15, où l'on quitte la boucle dès qu'on extrait de la file de priorité le sommet cible ;
- en lignes 17 et 18, où l'on guide la recherche par une heuristique consistant à évaluer le poids de la fin du chemin optimal reliant le sommet considéré `j` au sommet cible par sa distance euclidienne.

La preuve de correction de cet algorithme est difficile, nous ne la présentons pas.
Sa complexité est identique à celle de l'algorithme de Dijkstra.

Observons ce que renvoie l'appel `decrit_chemin(etiquettes, Astar(adj, XY))` sur nos trois exemples de graphe :

- pour le graphe de la figure 2, l'appel renvoie un chemin optimal : `A B C J L` ;
- pour le graphe de la figure 3, l'appel renvoie un chemin optimal : `A M N O` ;
- pour le graphe de la figure 4, l'appel renvoie un chemin optimal : `A B C D E I`.

On obtient bien les mêmes chemins optimaux qu'avec l'algorithme de Dijkstra, ce qui n'a rien d'étonnant.
Nous ne cherchons pas ici à prouver la correction de l'algorithme ni à évaluer sa complexité, qui sont au-delà des attendus de la spécialité NSI.

7. Exemples plus réalistes

Le site http://mpechaud.fr/scripts/parcours/index.html, déjà cité, propose des applets qui permettent d'observer l'exécution des différents algorithmes sur des graphes complexes et réalistes.

En figure 6, on compare l'exécution des algorithmes *best-first* et A* sur un même graphe euclidien. On observe à la fois que le chemin trouvé (en rouge) par *best-first* n'est pas optimal, mais en même temps que cet algorithme explore moins d'arêtes que A* (les arêtes explorées sont colorées en vert).

La figure 7 montre l'effet, sur le même graphe, de l'algorithme de Dijkstra : on retrouve bien le chemin optimal trouvé par A*, mais à un coût plus élevé en termes du nombre d'arêtes visitées.

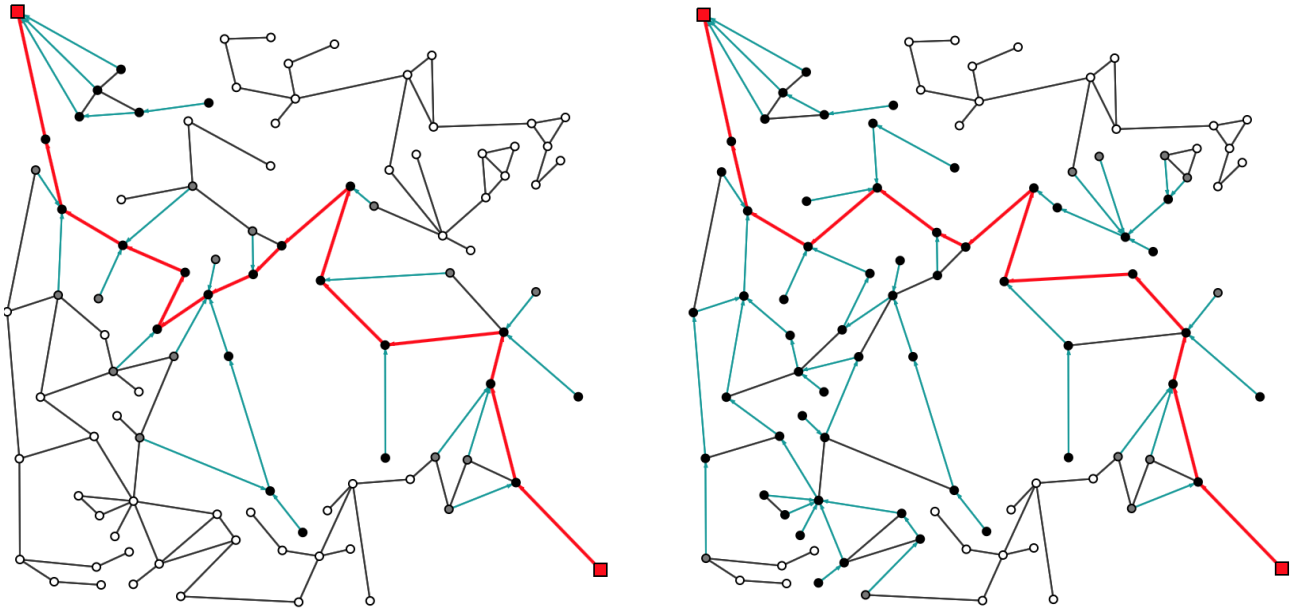

FIGURE 6 – algorithmes Best-First et A*

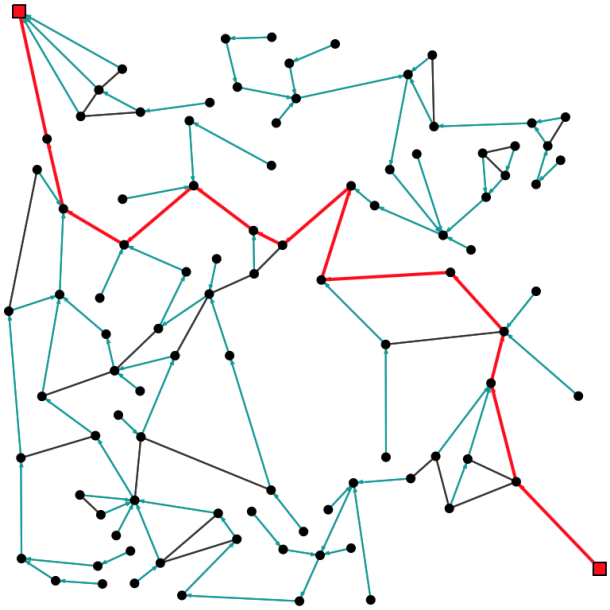

FIGURE 7 – algorithme de Dijkstra