Graphes

Les graphes servent à la représentation de nombreux problèmes : comment choisir un chemin le plus court possible qui passe une et une seule fois par chaque ville d'un ensemble donné (**Problème du voyageur de commerce**) ? Comment créer un emploi du temps en respectant les contraintes physiques (un seul cours par salle et par heure !) et celles de chacun ?
Un graphe représente un ensemble avec ses connexions : réseau de communication, réseau routier, circuit électronique, relations sociales, etc.

Terminologie

Définition 1

Un **graphe** est un couple (S, A) où

- S est un ensemble fini d'éléments appelés **sommets** (ou **noeuds**)
- A un ensemble d'arêtes, c'est-à-dire de couples de sommets (les **extrémités** de l'arête). *On imposera ici que les sommets doivent être distincts.*

Exemple

Le problème des **ponts de Königsberg** (Euler, 1735) a préfiguré la topologie et la théorie des graphes. Il consistait à déterminer l'existence d'un chemin, partant d'un point et y revenant, passant une fois exactement par chacun des sept ponts de la ville.

- Une arête peut être **orientée**, avec un sommet de départ et un sommet d'arrivée, ou non, auquel cas le sens n'importe pas : le couple formant l'arête est non ordonné. Deux sommets **reliés par une arête**, c'est-à-dire qui en sont les extrémités, sont dits **adjacents**, et sont **incident** à l'arête - l'arête est incidente à ces sommets. On parlera parfois d'**arc** pour désigner une arête orientée.
- ⚠On ne mélange pas au sein d'un même graphe arêtes orientées et non orientées !
- L'**ordre** d'un graphe est le nombre de sommets de ce graphe. Le **degré** $d(s)$ d'un sommet s est le nombre de sommets qui lui sont adjacents, le **degré** d'un graphe est le degré maximum de ses sommets.
- Dans le cadre d'un graphe orienté, on peut distinguer les arêtes sortantes et entrantes. Le degré sortant $d_+(s)$ (pour un graphe orienté d'un sommet s est le nombre de sommets t tels que (s,t) est une arête du graphe. On définit de même le degré entrant $d_-(s)$. Le degré d'un sommet d'un graphe orienté est la somme de son degré entrant et de son degré sortant.
- Un graphe est **pondéré** lorsqu'à chaque arête est associé un nombre réel, le **poids**.

Proposition

Dans un graphe non orienté, $|A| \leqslant \dfrac{|S| \times |S| - 1}{2}$. Dans un graphe orienté, $|A| \leqslant |S| \times |S| - 1$.

Un graphe admet une représentation graphique (exemples au tableau).

Définition 2

Un **chemin** de longueur k reliant deux sommets a (son point de départ) et b (son point d'arrivée) est une suite finie $a = x_0, x_1, ..., x_k = b$ de sommets tels que pour tout $i \in [\![0, k-1]\!]$, (x_i, x_{i+1}) est une arête du graphe. La **distance** entre deux sommets a et b d'un graphe g est la longueur du plus petit chemin, s'il existe, qui relie a et b. Dans les graphes non orientés, un ensemble de sommets qui sont tous à distance finie les uns des autres est dit **connexe**. Tout graphe non orienté peut être décomposé en **composantes connexes**, qui sont des sous-graphes connexes maximaux.

Remarques :

- Tout sommet est accessible depuis lui-même par un chemin de longueur 0.
- S'il existe un chemin de a vers b, on dit également que b est **accessible** depuis a. Dans un graphe non orienté, la relation d'accessibilité est une relation d'équivalence.
- Dans le cadre des graphes non orientés, une composante connexe est une classe d'équivalence pour l'accessibilité.

Définition 3

Un chemin est dit **simple** s'il ne passe pas deux fois par la même arête. Un **cycle** est un chemin simple dont le point de départ et le point d'arrivée sont confondus.

Dans un graphe non orienté, un cycle est de longueur au moins 3. Un cycle est dit **élémentaire** si la seule répétition de sommets est celle de ses extrémités, donc lorsque il ne contient pas d'autre cycle.

Implémentation : deux exemples

Les représentations proposées sont particulièrement adaptées au cas des graphes orientés, mais la représentation de graphes non orientés se fait en dédoublant toute arête non orientée (a, b) en deux arêtes orientées (a, b) et (b, a).

Listes d'adjacence

Un graphe peut être représenté par une liste d'adjacence : liste qui associe à chaque sommet la liste de ses voisins. Cette représentation permet l'ajout ou la suppression rapide de sommets et d'arêtes, mais l'accès à un sommet ou une arête en particulier est au pire linéaire en le nombre d'arêtes. Dans le cas d'un graphe non orienté, il faut penser à ajouter ou supprimer non seulement l'arête (a, b) mais également l'arête (b, a).
La quantité de mémoire utilisée est minimale, en $\Theta(n + p)$ où n est le nombre de sommets, et p le nombre d'arêtes.

Si on souhaite opérer sur des graphes statiques, sur lesquels on n'ajoute ni ne supprime de sommet ou d'arête, on peut considérer les sommets numérotés de 0 à $n - 1$, et les listes d'adjacences gardées en mémoire dans un tableau. On accède alors à la liste d'adjacence du sommet i en temps constant.

Matrices d'adjacence

Si les sommets d'un graphe G sont ordonnées, et mis en bijection (par exemple, à l'aide d'un dictionnaire) avec des entiers consécutifs de 1 à n (avec lesquels on choisit de les confondre), le graphe G peut être représenté par la matrice $M = (m_{i,j})_{i,j}$ où $m_{i,j}$ vaut 1 s'il existe une arête du sommet i vers le sommet j, et 0 sinon. Puisqu'on impose qu'une arête va d'un sommet vers un sommet distinct du premier, la diagonale est remplie de 0.
L'ajout et la suppression d'une arête se fait à coût constant, mais la représentation se fait en coût spatial quadratique. Si le graphe comporte peu d'arêtes et de nombreux sommets, cette représentation n'est pas optimale.

Exercices

1. Écrire une fonction qui détermine si un graphe représenté par une matrice d'adjacence est orienté ;

2. Écrire une fonction qui supprime le caractère orienté d'un graphe ;
3. Écrire deux fonctions passant d'une représentation d'un graphe à l'autre ;
4. Écrire une fonction déterminant si un cycle est élémentaire.
5. Écrire une fonction qui détermine si deux éléments a et b sont bien reliés par un chemin dont on donne les sommets intermédiaires.
6. Écrire une fonction qui crée un graphe aléatoire à n sommets. On considère que les sommets sont numérotés de 0 à $n-1$. Quels choix faites-vous ? Le nombre d'arêtes est-il passé en paramètre ou obéit-il à une loi aléatoire ? Quelle représentation choisir ?
7. *Écrire une fonction qui, connaissant deux sommets d'un graphe, calcule un chemin (soit quelconque, soit le plus court possible) reliant le premier au second, si un tel chemin existe.*
8. *Écrire une fonction qui détermine les composantes connexes d'un graphe.*

```
#Ecrire une fonction qui détermine si un graphe,
sous forme de matrice d'adacence est orienté


def estOriente(M):
    n = len(M) # la matrice est carrée
    for i in range (n):
        for j in range(i,n): # on peut ne traiter que la partie triangulaire supérieure
            if M[i][j]!=M[j][i]:
                return True
    return False

M1 = [[1,0,1],[0,0,1],[1,0,1]]#orientée
M2 = [[1,0,1],[0,1,0],[1,0,1]]#non orientée

print(estOriente(M1))
print(estOriente(M2))

#fonction qui désoriente. version liste d'adjacences
# on suppose que les sommets sont numérotés de 0 à (n-1)
def desoriente(L):
    for i in range(len(L)):
        for e in L[i]:
            # si i n'est PAS dans l[e], on l'ajoute.
            if not(i in L[e]):
                L[e].append(i)
    return


L1 = [[1,2],[0,4],[],[3],[2,1]]
desoriente(L1)
print(L1)


#Écrire deux fonctions passant d'une représentation d'un graphe à l'autre;

def listeVersMatrice(L):
    n = len(L)
    M = [[0 for i in range(n)] for j in range(n)]
    for i in range(n):
        for e in L[i]:
            M[i][e]=1
    return M

def matriceVersListe(M):
    n = len(M)
    L = [[] for i in range (n)]
    for i in range (n):
        for j in range(n):
            if M[i][j]==1:
                L[i].append(j)
    return L

print("graphe, liste : " ,L1)
```

```
M3 = listeVersMatrice(L1)
print(M3)
print(matriceVersListe(M3))

#  Écrire une fonction déterminant si un cycle est élémentaire.
def estElementaire(cycle):
    n = len(cycle)
    # un cycle est sous la forme d'une liste de n+1 sommets, de a vers a
    for i in range(n-1):
        for j in range(i+1,n-1):
            #si on trouve à deux indices distincts le même élément
            if cycle[i]==cycle[j]:
                return False
    return True


print(estElementaire([1,2,3,1,4,1]))   #non, le point de départ et d'arrivée est
print(estElementaire([1,2,3,12,2,1]))
print(estElementaire([1,2,3,12,4,1]))


# Écrire une fonction qui détermine si deux éléments $a$ et $b$ sont
bien reliés par un chemin dont on donne les sommets intermédiaires.

#j'utilise une représentation matricielle du graphe
def estUnChemin(a,b,chemin,graphe):
    sommetCourant = a
    for e in chemin:
        if graphe[a][e]!=1:
            #on a détecté une arêt inexistante
            return false
        #on se déplace vers le sommet suivant
        sommetCourant=e
    return True


# Écrire une fonction qui crée un graphe aléatoire à n sommets
#plusieurs choix sont possibles et également valables, l'important étant
de s'adapter au contexte (qui est ici, inexistant)
# Je choisis la représentation matricielle, et je met une arête entre deux
sommets de manière aléatoire, avec probabilité uniforme choisie ici
arbitrairement égale à 0.3
# attention, ce graphe aléatoire pourra avoir des artes d'un sommet vers lui-meme :
comment modifier cela?
import random
def grapheAleatoire(n):
    M = []
    for i in range(n):
        ligne = []
        for j in range(n):
            arete = 0
            tirage = random.random()
            if tirage<=0.3:
                arete = 1
```

```
            ligne.append(arete)
        M.append(ligne)
    return M

M = grapheAleatoire(100)
print(M)
L = matriceVersListe(M)
print(L)
```

Parcours d'un graphe

Pour simplifier, on notera ici **successeur** de a un sommet b tel qu'il existe une arête de a à b.
Parcourir un graphe, c'est en énumérer les sommets accessibles par un chemin à partir d'un sommet donné, généralement pour calculer quelque chose à partir de ces sommets : taille du graphe, composantes connexes, etc. De mnaière générale, on doit garder en mémoire les sommets déjà visités, ainsi qu'une liste, qui croît et décroît au fur et à mesure du déroulement de l'algorithme, des sommets à traiter par la suite.
L'algorithme général suit la structure suivante, utilisant un graphe `G` et un de ses sommets `depart`

```
def parcoursLargeur(G,depart):
    àTraiter = [depart]
    dejaVus = [false for i in taille(G)]
        tant que (àTraiter!=[])
        sommet = àTraiter[0] ou à Traiter[-1]
        supprimer(àTraiter, sommet)
        traiter sommet
        pour (destination dans adjacent(sommet,G)):
             si destination non dans dejaVus alors
                 àTraiter.append(destination)
                 dejaVus[destination] = True
```

Complexité du parcours

On note $n = |S|$ et $m = |A|$.
Lorsqu'on dispose de la liste des successeurs d'un sommet s, chacun de ces successeurs est rangé dans la pile ou la file, selon qu'il s'agit d'un parcours en profondeur ou en largeur, et chacun sera traité à son tour. Pour chaque sommet, on vérifie s'il a déjà été visité. On ne regarde jamais deux fois la liste des successeurs d'un sommet. Ainsi, pour au plus chaque arête, on doit effectuer $O(1)$ opérations : le parcours, qu'il soit en largeur ou en profondeur est donc en temps $O(m)$.
En mémoire, on doit garder $O(n)$ informations : les sommets déjà traités (dans un tableau de la taille de S) et ceux en liste d'attente (nécessairement moins nombreux que ceux de S).

Parcours en largeur

Dans le cadre du parcours en largeur, ou BFS (*breadth first search*), la liste "d'attente" est gérée suivant le principe d'une file : un sommet inséré avant un autre sera traité en premier. Dès qu'on traite un sommet, on rajoute à la liste des sommets traités tous ceux qui lui sont adjacents. En Python, si l'ajout d'un sommet se fait en fin de liste, on choisira pour le traiter le sommet en début de liste.
Si on observe la distance entre s et les sommets insérés dans la file, on observe qu'elle est croissante : d'abord s à distance 0 de lui-même, puis ses successeurs, à distance 1, les successeurs de ceux-ci, qui ne seront traités qu'ensuite, etc.
En python, on peut simuler une file à l'aide d'un liste en ajoutant un élément en fin de liste, puis en enlevant le premier élément de la liste, qui correspond au premier élément de la file. La suppression d'un élément dans un tableau nécessite la recopie des éléments suivants, tous décalés d'un cran. La suppression du premier élément, ou un ajout en première position (`pop(0)`, `insert(0,e)`) est donc linéaire en la taille de la liste. Dans ces conditions, il devient intéressant de pouvoir utiliser un module dédié :
le module `collections` implémente plusieurs types de données, dont en particulier les conteneurs `deque` qui permettent l'utilisation de liste avec ajouts et retraits rapides en début et en fin.

Parcours en profondeur

Le parcours en profondeur, ou DFS (*Depth First Search*) utilise non une file, mais une pile, c'est-à-dire une structure dans laquelle le dernier élément ajouté est celui que l'on enlève. Si on ajoute des éléments en fin de liste, c'est l'élément de fin de liste que l'on traitera en premier.
On peut également utiliser un algorithme récursif (l'utilisation d'une pile est un moyen de simuler plusieurs appels récursifs de fonctions, chaque appel non encore traité s'empilant, jusqu'à ce qu'une valeur soit renvoyée). La structure d'un tel algorithme reprend le principe "si le sommet observé n'a pas encore été traité : le traiter ; indiquer qu'il a été visité ; parcourir récursivement ses successeurs".

```
def parcoursRec(sommet, G, dejaVus):
    si sommet non dans dejaVus:
        dejaVus[sommet] = True
        traiter sommet
        pour chaque dest adjacent à s:
             parcoursRec(dest)
```

La fonction `parcoursRec` est ensuite appelée avec un sommet `sommet, un graphe \verbG'`, et un tableau `dejaVus` de taille n initialisé avec des `False`.
Une fonction récursive utilisant une **pile des appels** de taille maximale fixée, on peut obtenir une erreur s'il y a trop d'appels récursifs.

Applications

a Composante connexe

Si un graphe est orienté, le parcours en profondeur à partir d'un sommet `s` permettra de visiter tous les sommets accessibles depuis `s`. On peut le montrer par récurrence sur la distance entre s et le sommet cible c : si cette distance vaut 1, le sommet c est empilé dès traitement de s, et sera visité lorsqu'il sera dépilé. Si tous les sommets à distance $n-1$ sont visités, et si c est à distance n de s, alors il existe au moins un sommet étape e à distance 1 de c et à distance $n-1$ de s. Par hypothèse de récurrence, le sommet e sera visité lors du déroulement de l'algorithme. Si c n'a alors pas encore été visité, il sera empilé lors du traitement de e, puis traité lorsqu'il sera dépilé. Si un sommet n'est pas à distance finie de s, il n'existe pas de chemin le reliant à s, et l'algorithme procédant en suivant un certain chemin ne peut y accéder.

Puisque le parcours en profondeur permet **dans un graphe orienté** d'obtenir exactement les sommets accessibles **depuis** un sommet s, il permet également de le faire dans un graphe non orienté, où la relation d'accessibilité est une relation d'équivalence. L'ensemble des sommets traités est donc exactement ceux qui sont accessibles depuis s, c'est-à-dire sa composante connexe.

☞ Comment montrer la même chose lorsqu'on utilise un parcours en largeur au lieu d'un parcours en profondeur ?

Le parcours d'un graphe **non orienté** à partir d'un sommet permet donc d'en parcourir la composante connexe contenant ce sommet. Si on veut obtenir toutes les composantes connexes du graphes, il faut itérer sur les sommets non encore parcourus. Si après un parcours à partir d'un unique sommet, on épuise tous les sommets d'un graphe, celui-ci est connexe. Si on modifie le tableau `dejaVus` en indiquant, non si on a vu un sommet ou non, mais la distance au sommet s, on peut calculer la distance à s de tous les sommet de sa composante connexe (cas non orienté), ou de tous les sommets accessibles depuis s (cas orienté).

b Plus court chemin

Un parcours en largeur d'un graphe à partir d'un sommet `depart` permet de trouver le plus court chemin entre ce sommet et un autre : dès que le sommet destination est atteint, on a trouvé un plus court chemin. On peut envisager d'utiliser un tableau `chemin`, tel que `chemin[i]` contient, soit une valeur sentinelle, soit un chemin minimal vers ce sommet. Ce tableau est mis à jour à chaque traitement de sommet. Cet algorithme marche-t-il si on utilise un parcours en profondeur ?

c Détection de cycles - graphe orienté

Le parcours en profondeur à partir de s permet de détecter des cycles contenant le sommet s, mais nécessite pour cela une adaptation. En effet, retomber sur un sommet déjà vu ne signifie pas nécessairement qu'on a trouvé un cycle. Pour détecter un cycle, on sépare les sommets en trois types : jamais vu ; entièrement traité ; traitement en cours (c'est-à-dire qu'on traite un des sommets accessibles depuis lui-même lorsqu'on retombe sur ce sommet). Dans le troisième cas seulement, un cycle est détecté : le sommet est accessible depuis lui-même.

Exercices

1. **Tester** le module `collections` et les fonctions `append(e)`, `appendleft(e)`, `clear()`, `count()`, `pop()`, `popleft()` (Créer une liste ou une file avec `file = collections.deque()`).
2. Écrire une fonction qui imprime les sommets d'une composante connexe d'un graphe non orienté, présenté sous forme de liste d'adjacence, dans l'ordre du parcours en largeur (puis faire de même avec un parcours en profondeur)
3. Écrire une fonction qui détecte la présence de cycles dans un graphe *à l'aide d'un parcours bien choisi du graphe* et renvoie un tel cycle.
4. Écrire une fonction qui détermine la connexité ou non d'un graphe *à l'aide d'un parcours bien choisi du graphe*. Compter les composantes connexes.
5. Écrire une fonction qui détermine la distance entre deux sommets s et s' d'un graphe G et renvoie un chemin optimal.
6. Écrire une fonction qui détermine si chaque composante est un arbre (on considère ici que x est fils de y s'il existe une arête de x vers y, et aucune de y vers x).
7. Trouver l'ordre de grandeur de la taille maximale de la pile d'appel, c'est-à-dire du nombre maximal d'appels récursifs avant d'obtenir l'erreur :

   ```
   RecursionError: maximum recursion depth exceeded while calling a Python object
   ```

```
# module collections
import collections
file = collections.deque()
file.append(1)
file.append(2)
file.appendleft(3)
file.append(4)
file.appendleft(5)
print(file)
#enlever le dernier élément et le stocker dans une variable
element = file.pop()
print(element, file)
#enlever le premier élément et le stocker dans une variable
element = file.popleft()
print(element, file)
# pour savoir si la pile/file est vide, on peut regarder sa longueur
print(len(file))
# Trois opérations suffisent (pile ou file) : ajout, suppression
(avec lecture de l'élément), test pour savoir si la pile/file est vide.

#Écrire une fonction qui imprime les sommets d'une composante connexe d'un graphe
 non orienté, présenté sous forme de liste d'adjacence, dans l'ordre du parcours
 en largeur (puis faire de même avec un parcours en profondeur)
# Le graphe est donné ici sous forme de liste d'adjacence
import collections
def parcoursProfondeur(graphe, sommet):
    #le traitement d'un sommet consiste ici à le mettre en fin de liste
    liste = []
    n = len(graphe)#n est le nombre de sommets
    dejaVus = [False for i in range(n)]
    dejaVus[sommet]=True
    pile= collections.deque([sommet])
    while (len(pile)>0):
        sommetEnTraitement = pile.pop()
        liste.append(sommetEnTraitement)
        successeurs = graphe[sommetEnTraitement]
        for e in successeurs:
            if not (dejaVus[e]):
                dejaVus[e]=True
                pile.append(e)
    return liste

graphe = [[1, 4], [2, 3], [0, 7], [5], [2, 4, 7], [4, 5], [2, 3], [1, 2, 4, 6]]
desoriente(graphe)
print(graphe)
print(parcoursProfondeur(graphe,0))

#le Parcours en largeur s'obtient en considérant non une pile mais une file,
#on peut par exemple modifier pile.pop() en file.popleft()

#On peut implémenter un parcours en profondeur récursif, qui suit le principe :
# on traite un noeud, puis récursivement, on traite chacun de ses successeurs.
#attention, on n'obtiendra pas le même parcours : les listes sont traitées dans
le sens inverse, et l'indication "dejaVu" intervient à un moment différent par rapport
```

```
au précédent algorithme
def parcoursProfondeurRecursif(graphe, sommet, dejaVus):
    print(sommet, end =',')
    dejaVus[sommet]=True
    for e in graphe[sommet]):
        if not dejaVus[e]:
            parcoursProfondeurRecursif(graphe, e, dejaVus)


print(parcoursProfondeurRecursif(graphe,0, [False for i in range(8)]))
```

Plus court chemin

Algorithme de Dijkstra

L'algorithme de Dijkstra répond au problème des plus courts chemins à partir d'un sommet précis, appelé **source** dans un graphe pondérés avec des poids **positifs** (à chaque arête du graphe est associée un poids, qui est un nombre réel, ici, positif).
Il consiste en la mise à jour d'une partition des sommets et d'un tableau des distance de s aux autres sommets.
Au début de l'algorithme, les sommets sont divisés en deux sous-ensembles. L'ensemble S contient le sommet source ; l'ensemble S' contient les autres sommets. On écrit dans le tableau les distances entre s et chaque sommet, en ne tenant compte que des arêtes entre s et les sommets de S', autrement dit, on ne considère que les sommets adjacents à s, les autres étant placés à distance $+\infty$ de s.
Puis, jusqu'à épuisement des sommets de S', on sélectionne le sommet t qui est le plus proche de s, et il rejoint les sommets de S. On met à jour ensuite les distances des sommets de S' en choisissant la plus petite des distances entre celle qui est enregistrée dans le tableau, et la somme de la distance de s à t et de la distance de t à ses sommets adjacents.

La structure de l'algorithme est la suivante :

```
S = (s)
S' : ensemble de tous les autres sommets
initialiser un tableau des distances à s
tant que S' est non vide:
    trouver le sommet t le plus proche de s
    enlever t de S'
    mettre t dans S
    actualiser le tableau des distances en ne prenant en compte que les arêtes issues de t
renvoyer le tableau des distances
```

Si la recherche de l'élément le plus proche de s dans S' n'est pas optimisée, elle nécessite $O(k)$ opérations élémentaires, avec k la taille de S'. La mise à jour du tableau est de l'ordre de grandeur de la taille de la liste d'adjacence du dernier sommet ajouté à S. Ces deux opérations sont des $O(n)$. L'algorithme entier est alors de complexité $O(n^2)$, borne supérieure atteinte dans le cas de graphes complets, possédant le nombre maximal d'arêtes.
Si on dispose d'une file de priorité, structure de données permettant de trouver l'élément minimal en temps constant, puis de mettre à jour la structure elle-même en temps $O(\log(n))$, la complexité de l'algorithme est en $O(n+p)\log n$: les mises à jour des distances se font une fois exactement pour chaque arête, et la file de priorité doit être mise à jour à chaque fois ; la recherche de l'élément minimal se fait n fois, avec réorganisation de la file ensuite. Cette organisation est

préférable lorsque les graphes sont **creux**, c'est-à-dire lorsqu'ils n'ont pas beaucoup d'arêtes, et en particulier lorsque $(n+p)\log(n) = O(n^2)$, soit lorsque $p = O(\frac{n^2}{\log(n)})$.

Exercice : Implémenter l'algorithme de Dijkstra.

Algorithme A*

L'algorithme A* permet de répondre, rapidement mais approximativement, à la question d'un plus court chemin entre deux sommets d'un graphe : il renverra toujours une solution, si une solution existe, en revanche, il pourra ne pas renvoyer la meilleure solution, c'est-à-dire le plus court chemin. Cet algorithme utilise une heuristique, c'est-à-dire une fonction calculant rapidement une solution non forcément optimale, et une file de priorité. Dans l'algorithme de Dijkstra, le sommet retiré des sommets en attente - tant qu'on n'a pas trouvé le sommet cible - est celui qui a la distance (temporaire) à la source s la plus petite. Ici, c'est la somme de cette distance et de la valeur renvoyée par l'heuristique que l'on souhaite minimiser. D'autres heuristiques peuvent être la distance "à vol d'oiseau" (distance euclidienne), la distance de Manhattan (somme du décalage suivant deux axes perpendiculaires de la source à la cible)n, ou autre. Si l'heuristique est l'heuristique nulle, qui affecte 0 à tout élément, alors on retrouve l'algorithme de Dijkstra.
L'idée se base sur la réponse intuitive au problème du plus court chemin : autant aller *dans la direction* de la cible. L'heuristique donnera une plus petite valeur (c'est la plus petitte valeur qui nous intéresse !) à un sommet qui semble être un bon candidat pour se trouver sur le chemin vers la cible. Cet algorithme permet, par exemple, de trouver un chemin entre deux points sur un terrain encombrés d'obstacles, et il vient, historiquement, de la recherche en robotique.