13 Recherche de chemins

- Algorithmes de Dijkstra et A⋆.

13.1 Algorithme de Dijkstra

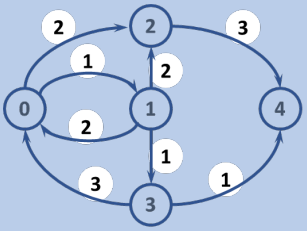
Sources

- Cours de Quentin Fortier.
- Cours de Jules Svartz, Lycée Masséna.
- https://www.youtube.com/watch?v=GC-nBgi9r0U

L'algorithme de Dijkstra est un parcours en largeur d'un graphe **pondéré (poids positifs)** et orienté. Il permet de calculer l'ensemble des plus courts chemins entre un sommet vers tous les autres sommets du graphe.

Pour modéliser le graphe, on utilisera une matrice d'adjacence M pour laquelle $M_{ij} = w(i,j)$ et $w(i,j)$ représente le poids de l'arête de i vers j. Lorsqu'il n'y a pas d'arc entre deux sommets, on aura $M_{ij} = \infty$.

Exemple –



Colonne j, sommet d'arrivée

Ligne i, sommet de départ

$$\begin{bmatrix} 0 & 1 & 2 & \infty & \infty \\ 2 & 0 & 2 & 1 & \infty \\ \infty & \infty & 0 & \infty & 3 \\ 3 & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

Définition – Poids d'un chemin

Soit un un graphe pondéré $G = (V, E, w)$ où V désigne l'ensemble des sommets, E l'ensemble des arêtes et w, la fonction poids définie par $w : E \to \mathbb{R}$ ($w(u,v)$ est le poids de l'arête de u vers v).
On appelle poids du chemin C et on note $w(C)$ la somme des poids des arêtes du chemin.
Un chemin de $u \in V$ à $v \in V$ est un plus court chemin s'il n'existe pas de chemin de poids plus petit.

Exemple –

Pour le chemin $C = 0, 1, 2, 4$, on a $w(C) = 6$.
Pour le chemin $C' = 0, 1, 3, 4$, on a $w(C') = 3$.
C' est un plus court chemin.

Définition – Distance

La distance $d(u,v)$ est le poids d'un plus court chemin de u à v. On peut alors noter $d(u,v) = \inf\{w(C) | C \text{ est un chemin de } u \text{ à } v\}$.
Si v n'est pas atteignable depuis u on poser $d(u,v) = \infty$.

Exemple –

Dans le cas précédent, $d(0,4) = 3$, $d(2,4) = 3$ et $d(4,1) = \infty$

Propriété – Sous-optimalité

Soit C un plus court chemin de u à v ainsi que u' et v' deux sommets de C. Alors le sous-chemin de C de u' à v' est aussi un plus court chemin.

Exemple –

$C' = 0,1,3,4$ est un plus court chemin ; donc $C' = 1,3,4$ ou $C' = 0,1,3$ aussi.

Objectif

Soit un graphe pondéré $G = (V,E,w)$ où V désigne l'ensemble des sommets, E l'ensemble des arêtes et w, la fonction poids.
Soit s un sommet de V. L'objectif est de déterminer la liste de l'ensemble des distances entre s et l'ensemble des sommets de V.

Pour répondre à l'objectif, on peut formuler l'algorithme de Dijkstra ainsi.

Entrées : un graphe pondéré donné par liste ou marrice d'adjacence, un sommet `s` du graphe
Sortie : `D` liste des distances entre `s` et chacun des sommets
Initialisation de `D` : `D = n`$\times[\infty]$
Initialisation de `D` : `D[s] =`$[0]$
Initialisation de `T` : `T = n x [False]` liste des sommets traités
Initialisation d'une file de priorité avec le sommet de départ `F = {s}`
Tant que `F` n'est pas vide :
    Recherche du sommet `u` tel que `d[u]` minimal parmi les sommets de `F`
    Pour tout voisin v de u **faire** :
        Si v n'est ni dans T ni dans F **alors**
            Ajouter v à F
        `D`=min $(d[v], d[u] + w(u,v))$
    `T[u] = True`
Renvoyer `D`

Une des étapes qui diffère avec le parcours en largeur notamment, est l'utilisation d'une file de priorité et la recherche du sommet vérifiant $d[u]$ minimal. Cela signifie que lorsqu'on partira d'un sommet `s`, on déterminera alors l'ensemble des distances permettant d'atteindre les voisins de `s`. À l'itération suivante, on visitera alors le sommet ayant la distance la plus faible.

Définition – File de priorité

Une file de priorité est une structure de données sur laquelle on peut effectuer les opérations suivantes :

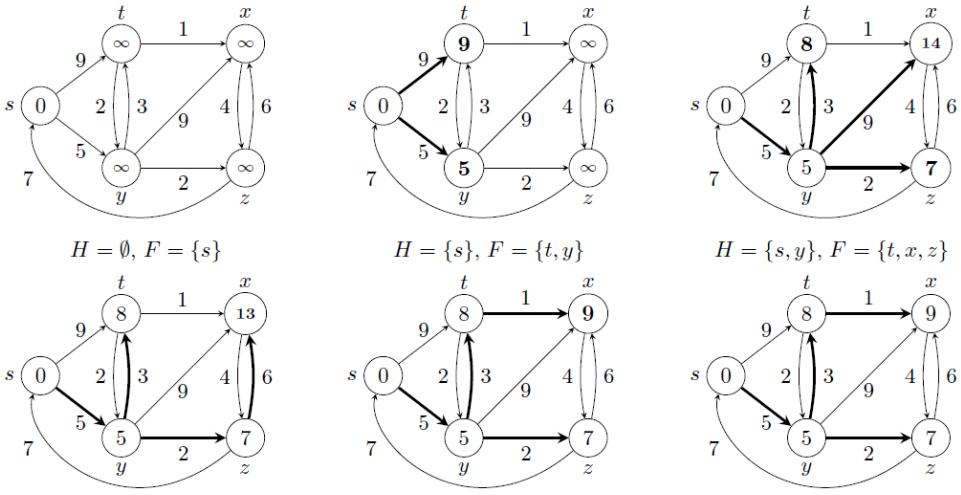
- insérer un élement ;
- extraire l'élément ayant, dans notre cas, la plus petite valeur ;
- tester si la file de priorité est vide ou pas.

Exemple –

Soit une file de priorité comprenant les éléments suivants : `file = [12,1,4,5]`. La file de priorité est dotée d'une méthode `pop` permettant d'extraire la plus petite valeur. Ainsi, `file.pop()` renvoie `1` et la file contient alors les éléments `[12,4,5]`. En réitérant `file.pop()` renverra la valeur `4` et la file contient désormais les éléments `[12,5]`.
Soit une file de priorité comprenant les éléments suivants : `file = [(1,2),(2,5),(0,1)]`. La méthode `pop` permetttra d'extraire le couple pour lequel la première valeur est la plus petite.
Ainsi, `file.pop()` renvoie `(0,1)` et la file contient alors les éléments `[(1,2),(2,5)]` etc.

Exemple –

[Jules Svartz] La figure suivante représente le déroulement de l'algorithme de Dijkstra sur un graphe à 5 sommets, depuis la source s. Pour chaque sommet u on a fait figurer la valeur $d[u]$ à l'intérieur du cercle. Les arcs en gras représentent l'évolution de la liste des prédecesseurs.



On peut donc commencer par implémenter une fonction `cherche_min` permettant de trouver le sommet `i` vérifiant `d[i]` minimal parmi les sommets n'ayant pas été traités.

```
1 def cherche_min(d, traites):
2     """ Renvoie le sommet i vérifiant d[i] minimal et traites[i] faux, s'il
      existe un tel sommet
3     tel que d[i] != inf. Sinon, renvoie -1 """
```

```python
    n=len(d)
    x=-1
    for i in range(n):
        if not traites[i] and d[i] != float('inf') and (x==-1 or d[x]>d[i]):
            x=i
    return x
```

On donne alors l'algorithme de Dijkstra.

```python
def dijkstra_mat(G,s):
    """
    G donné par matrice d'adjacence. Renvoie les poids chemins de plus petits
    poids depuis s.
    """
    n=len(G)
    d = [float('inf')]*n
    d[s]=0
    traites = [False]*n
    while True:
        x=cherche_min(d,traites)
        if x==-1:
            return d
        for i in range(n):
            d[i]=min(d[i], d[x]+G[x][i])
        traites[x]=True
```

Propriété –

Pour n sommets et a arcs, il existe un algorithme de Dijkstra telle de complexité $\mathcal{O}\left(a + n \log n\right)$.

Exemple –

Reprendre le graphe précédent et utiliser l'algorithme de Dijkstra en partant du sommet t.

13.2 Algorithme A⋆ – Problématique

Références :

- https://khayyam.developpez.com/articles/algo/astar/
- https://www.youtube.com/watch?v=-L-WgKMFuhE

Figure 13.1 – Exemple de chemin

Lorsqu'on cherche le plus court chemin entre deux sommets, départ et arrivée, le plus rapidement possible et en évitant les obstacles éventuels, l'algorithme A⋆ (prononcer A star) est fait pour ça ! Cet algorithme est par exemple utilisé dans les jeux vidéos. C'est un algorithme de recherche de chemin dans un graphe. C'est l'un des plus efficaces en la matière. Il ne donne pas toujours la solution optimale mais il donne très rapidement une bonne solution.

Au premier abord, on pourrait se dire que pour trouver un chemin d'un point à un autre il faut commencer par se diriger vers la destination. Et bien... c'est justement cette idée qu'utilise l'algorithme A*! À chaque itération, on va tenter de se rapprocher de la destination. Pour cela, on va donc privilégier les possibilités directement les plus proches de la destination, en mettant de côté toutes les autres. Toutes les possibilités ne permettant pas de se rapprocher de la destination sont mises de côté, mais pas supprimées. Elles sont simplement mises dans une liste de possibilités à explorer si jamais la solution en cours s'avère mauvaise. En effet, on ne peut pas savoir à l'avance si un chemin va aboutir ou s'il sera le plus court. Il suffit que ce chemin amène à une

impasse pour que cette solution devienne inexploitable.
L'algorithme va donc d'abord se diriger vers les chemins les plus directs. Et si ces chemins n'aboutissent pas ou bien s'avèrent mauvais par la suite, il examinera les solutions mises de côté. C'est ce retour en arrière pour examiner les solutions mises de côté qui nous garantit que l'algorithme nous trouvera toujours une solution (si tenté qu'elle existe, bien sûr).

13.3 Comment fonctionne cet algorithme

Il est basé sur l'algorithme de `Dijkstra` auquel est ajouté une `heuristique`.

Définition – Heuristique

En algorithmique, une **heuristique** est une méthode de calcul qui fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte, pour un problème d'optimisation difficile (wikipédia).
Cette heuristique est définie dans la fonction : $f(s) = g(s) + h(s)$.
Avec :

- $g(s)$ est le coût du chemin optimal partant du sommet initial jusqu'au sommet s ;
- $h(s)$ le coût estimé du reste du chemin partant de s jusqu'à un état satisfaisant de l'arrivée. $h(s)$ est une heuristique.

13.3.1 Principe

Propriété – Principe de fonctionnement

Dans des applications de recherche de chemin sur une image ou un quadrillage par exemple, il est possible de connaître pour chaque sommet la distance « à vol d'oiseau » ou « à taxi-distance » de chaque sommet à l'arrivée. On peut utiliser une de ces heuristiques et, au lieu de choisir le sommet ayant la plus petite distance depuis le départ parmi les sommets visités, on peut choisir la relation :

$$f(s) = d(\text{depart}, s) + d'(s, \text{fin})$$

On choisit alors la valeur `min` des f(s) des sommets visités.

Remarque

- La valeur calculée peut être mémorisée sous forme d'entier.
- Vous pouvez calculer ces distances de la manière que vous voulez, distance euclidienne, distance de `Manhattan` ou autre, elles peuvent convenir.

Définition – Distance de Manhattan

- La distance de `Manhattan`, appelée aussi *taxi-distance*, est la distance entre deux points parcourue par un taxi lorsqu'il se déplace dans une ville où les rues sont agencées selon un réseau ou quadrillage (figure 13.2).
- Un **taxi-chemin** est le trajet fait par un taxi lorsqu'il se déplace d'un sommet du réseau à un autre en utilisant les déplacements horizontaux et verticaux

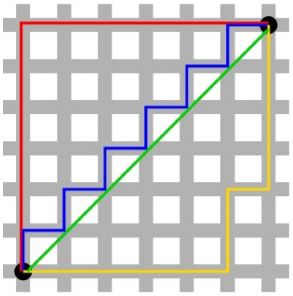
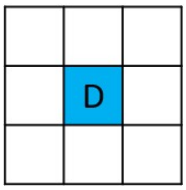


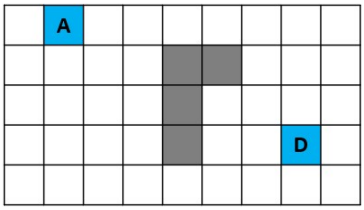
Figure 13.2 – Distance de Manhattan (rouge, bleu, jaune contre distance euclidienne en vert)

du réseau.

13.3.2 Exemple de construction sur un quadrillage



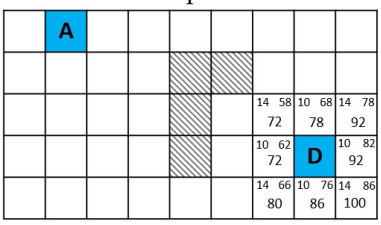
Toutes les cases du quadrillage sont des sommets et les sommets voisins sont les cases ayant un côté ou un angle commun.
Chaque case voisine par le côté est à une distance 10 soit 1×10 et les cases en diagonale à une distance 14 soit $\sqrt{2} \times 10$ arrondi à la valeur entière.



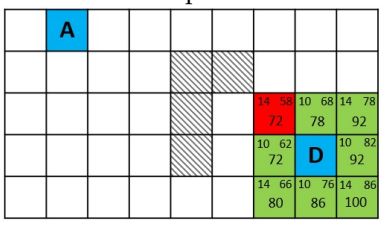
On cherche à déterminer le plus court chemin entre D et A. Sur le quadrillage ci-contre, on peut reporter les distance de Manhattan de chaque case à **A**.

Ensuite, on évalue la distance à l'origine, la distance de la fin et le coût global (étape 1). On choisit le sommet dont le coût global est le plus faible. En cas d'égalité, on choisit le sommet le plus proche de l'arrivée (étape 2).

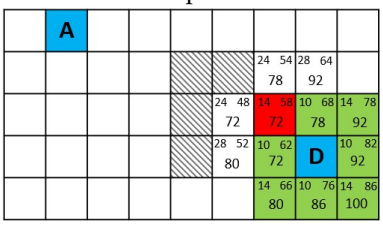
Etape 1



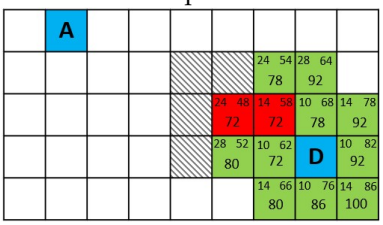
Etape 2



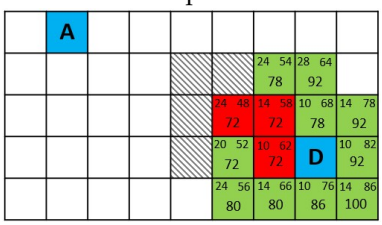
Etape 3



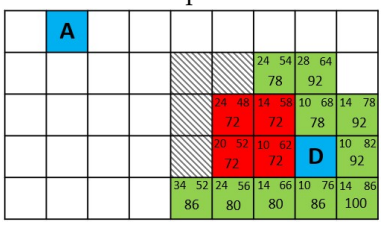
Etape 4



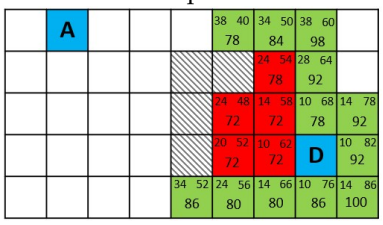
Etape 5



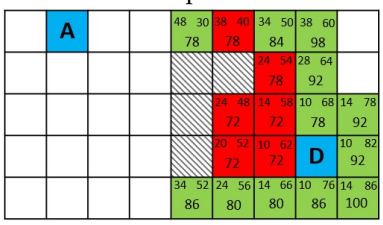
Etape 6



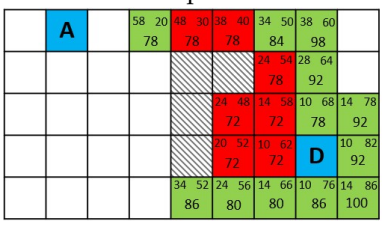
Etape 7



Etape 8



Etape 9



Finalement le chemin le plus court est donné figure 13.3.

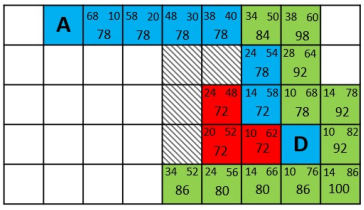


Figure 13.3 – Chemin le plus court

.

13.4 L'algorithme A*

13.4.1 Structures de données

Un sommet du graphe peut être représenté sous forme d'une liste dans un dictionnaire avec :

- la valeur de `G` (de type `int`), c'est la distance pour aller du point de départ au sommet considéré ;

- la valeur de `H` (de type `int`), c'est la distance pour aller du sommet considéré au point d'arrivée ;
- la valeur de `F` (de type `int`), c'est la somme des deux précédents mémorisée pour ne pas la recalculer ;
- le parent, représenté par ses coordonnées.

Initialement, un sommet non visité prend comme valeurs : `{(l,c) :[inf, None, inf, None]}`.

Si le point $O(0,0)$ est le point de départ et $A(5,9)$ est le point d'arrivée, alors le sommet $B(0,1)$ voisin de O prend comme valeurs : `{(0,1) :[10, 106, 116, (0,0)]}` On peut aussi choisir l'écriture suivante, `{(0,1) :{'G' :10,'H' :106,'F' :116,'pred' :(0,0)}}`

13.4.2 L'heuristique

```
def heuristique(pt1:tuple,A:tuple)->int:
    """Calcul de la valeur de l'heuristique avec la méthode du cours, 14 pour
    un déplacement diagonal et 10 pour un déplacement sur le côté entrées :
    pt1, tuple ou list des coordonnées et A, tuple ou list des coordonnées
    sortie : H : integer, valeur de l'heuristique. """
    a = abs(pt1[0]-A[0])
    b = abs(pt1[1]-A[1])
    if a <= b:
        return (a*14+(b-a)*10)
    else:
        return (b*14+(a-b)*10)
```

13.4.3 Le graphe pondéré

```
G = np.ones((5,9))
G[1,4] = -10
G[1,5] = -10
G[2,4] = -10
G[3,4] = -10
M = {}
ligne,colonne=G.shape
for i in range(ligne):
    for j in range(colonne):
        if not G[i,j] == -10:
            M[i,j] = [np.inf,None,np.inf,None]
```

13.4.4 L'algorithme

```
### les voisins sont les cases a côté du point considéré en ligne droite ou en
     diagonale
def estVoisin(M:dict,pt1:tuple):
    l,c = pt1
    voisins = []
    for i in [-1,0,1]:
        for j in [-1,0,1]:
            if (l+i,c+j) in M:
                voisins.append((l+i,c+j))
    voisins.remove(pt1)
    return voisins
```

```

# >>> estVoisin(G,[1,2])
# [[0, 1], [0, 2], [0, 3], [1, 1], [1, 3], [2, 1], [2, 2], [2, 3]]
```

```
### on peut calculer la distance entre deux points voisins en dehors de Astar
def distance(pt1:tuple,pt2:tuple):
    '''pt1 et pt2 doivent être voisin en ligne ou en diagonale'''
    ligne = 0
    colonne = 0
    i,j = pt1
    l,c = pt2
    if not i == l:
        ligne=1
    if not j == c:
        colonne = 1
    if colonne == ligne:
        return 14
    else:
        return 10

# >>> distance([1,2],[0, 3])
# 14
```

```
def Astar(M:dict,depart,fin,visited=[]):
    '''calcul le plus court chemin en partant de départ pour atteindre arrivée
     par l'algorithme de Astar avec un heuristique de distance la plus courte
    entrées :
    M : dict, dictionnaire dont chaque sommet est un couple de coordonnées et
    sa valeur une liste de 4 éléments : G, H, F et le prédécesseur
    départ : un sommet de M dont on connait les coordonnées
    fin : un sommet de M dont on connait les coordonnées
    sortie : None (ne renvoie rien)'''
    if depart not in list(M.keys()):
        raise TypeError('Le sommet de départ n est pas dans le graphe')
    if fin not in list(M.keys()):
        raise TypeError('Le sommet d arrivée n est pas dans le graphe')
    # condition de sortie de la boucle récursive
    if depart==fin:
        # on construit le plus court chemin et on l'affiche
        chemin=[]
        pred=fin
        while pred != None:
            chemin.append(M[pred][-1])
            pred=M[pred][-1]
        chemin.reverse() # on retourne la liste dans le bon ordre
        print ('chemin le plus court :'+str(chemin)+' cout='+str(M[fin][2]))
    else : # au premier passage, on initialise le cout à 0
        if visited==[] :
            M[depart][0]=0 # changement
        # on visite les successeurs de depart
        voisins=estVoisin(M,depart) # changement
        for voisin in voisins:
            if voisin not in visited:
                # heuristique
                G = M[depart][0] + distance(depart,voisin)
                H = heuristique(voisin,fin)
                F = G + H
                if F < M[voisin][2]:
                    # les distances calculées
```

```
34                      M[voisin][0] = G
35                      M[voisin][1] = H
36                      M[voisin][2] = F
37                      # le prédécesseur
38                      M[voisin][3] = depart
39          # On marque comme "visited"
40          visited.append(depart)
41          # Une fois que tous les successeurs ont été visités : récursivité
42          # On choisit les sommets non visités avec la distance globale la plus
    courte
43          # On ré-exécute récursivement Astar en prenant depart='x'
44          not_visited={}
45          for k in M.keys():
46              if not k in visited :
47                  not_visited[k] = M[k][2]
48          x=min(not_visited, key=not_visited.get)
49          Astar(M,x,fin,visited)
```