TP : Dijskstra et A* par files de priorités

December 19, 2024

Deux exemples de graphes donnés par liste de voisins et poids des arcs.

Les exemples ci-dessous représentent des graphes implémentés comme des listes de listes d'adjacence avec pondérations. La présence de (j, p) dans la i-ème liste d'adjacence signifie que j est un voisin de
i et que le poids de l'ac (i, j) est p.

[1]:
```
#graphe 1
ex1 = [[(1,1),(2,3)],[(0,1),(2,1),(3,2)],[(0,3),(1,1),(4,4)],\
[(1,2),(4,2),(5,6)],[(2,4),(3,2),(5,2)],[(3,6),(4,2)]]
```

On lit que $(0, 2)$ est un arc et que son poids est 2, que $(3, 5)$ est un autre arc et son poids est 6 etc.

[2]:
```
#graphe 2
ex2 = [[(1,6),(2,9)],[(0,6),(2,5),(3,8),(6,6)],[(0,9),(1,5),(3,4),(4,8),(5,7)],\
[(1,8),(2,4),(5,4),(6,5)],[(2,8),(5,9),(7,4)],[(2,7),(3,4),(4,9),(6,3),(7,10)],\
[(1,6),(3,5),(5,3),(7,6)],[(4,4),(5,10),(6,6)]]
```

[3]:
```
# un graphe vu en cours
cours = [[(1,12),(4,5)],[(2,2)],[(0,1)],\
         [(2,2),(1,3)],[(1,1),(3,2),(2,4)]]
```

Les dessiner.

1 Dijskstra

1.1 Files de priorité

Se documenter sur les files de priorité sur Wikipedia ou bien consulter le chapitre 2 sur les tas de ce cours.

Le module `heapq` de Python implémente la notion de tas. On gère ensuite ce tas comme une *file de priorité*.

Dans la documentation officielle en Français, on trouve :

"Les *tas* sont des arbres binaires pour lesquels chaque valeur portée par un nœud est inférieure ou égale à celle de ses deux fils. L'implémentation des tas par `heapq` utilise des tableaux pour lesquels `tas[k] <= tas[2*k+1]` et `tas[k] <= tas[2*k+2]` pour tout k, en commençant la numérotation à zéro. Pour contenter l'opérateur de comparaison, les éléments inexistants sont considérés comme

porteur d'une valeur infinie. L'intérêt du tas est que son plus petit élément est toujours la racine, `tas[0]`."

Bref les tas sont en Python des arbres représentés par un tableau. Un noeud père en position k a deux fils aux positions $2k+1$ et $2k+2$. Un père est toujours plus petit que ses fils.

Il faut bien comprendre qu'il n'y a pas de classe `tas`. Ce sont les listes qui font office de tas. On trouve dans `heapq` un certain nombre d'outils qui permettent de considérer une liste comme un tas.

Dans le principe : - on crée une liste vide `h`

- on ajoute à `h` autant d'éléments qu'on veut en utilisant la fonction `heappush`. Cette fonction réorganise la liste pour qu'on puisse toujours la considérer comme un tas.
- à tout moment, on peut retirer l'élément le plus petit de `h` par `x=heappop(h)`. Cette fonction réorganise la liste pour qu'on puisse toujours la considérer comme un tas.

1.1.1 Tuto

cette section est donnée à titre informatif.

[4]:
```
import heapq as hq #importer le module
```

[5]:
```
t = [1, 3, 5, 11, 13, 2, 4, 6, 8, -1]# des éléments à ajouter
h=[]
for e in t:
    hq.heappush(h,e)
h
```

[5]:
```
[-1, 1, 2, 6, 3, 5, 4, 11, 8, 13]
```

Affichons le contenu de `h` par niveau de profondeur dans l'arbre binaire.

[6]:
```
#*****************
i = 0
exp = 1
try:
    while i < len(h):
        for j in range(exp):
            print("{}".format(h[i]), end=" ")
            i+=1
        print()
        exp*=2
except IndexError:
    print()
```

```
-1
1 2
6 3 5 4
11 8 13
```

Dans le retour précédent, on observe que :

- le minimum est bien le premier élément de `h`.
- Les fils sont plus grands que leur père. Par exemple `t[3]` (qui vaut 6) a deux fils en positions 7 et 8 : `t[7]` et `t[8]` qui valent 11 et 8.

1.1.2 Tas comme files de priorités

Dans l'algorithme de Dijkstra, il faut extraire de la file de priorité le sommet de plus petite distance à la source. Un moyen simple est de stocker les sommets avec leurq distances à la source dans un tuple `(distance,sommet)`. L'ordre lexicographique garantit que les tuples seront rangés prioritairement par leur distance.

1.2 Retour à Dijkstra

Un problème est que nous avons stockés nos sommets dans les listes d'adjacence sous la forme `(sommet, poids de l'arc)`. Il faut s'en souvenir au moment du codage.

Un problème plus préoccupant est que nous avons besoin de modifier régulièrement les distances des sommets dans la file. Or aucune fonction de `heapq` ne nous le permet. Nous allons donc autoriser un sommet à figurer en plusieurs exemplaires dans la file mais stocké avec des distances différentes.

La solution ci-dessous est proposée par François Fayard.

- On commence par insérer la source dans la file avec une distance de 0
- Tant que la file de priorité n'est pas vide :
  - On récupère l'élément x ayant la plus faible distance δ.
  - Si x n'a pas encore été visité :
    - On le marque comme visité. La distance δ est alors la distance entre la source et x.
    - Pour tous ses voisins y, on les insère dans la file de priorité avec la distance $\delta + \rho(x \to y)$.

Un même sommet peut maintenant se retrouver plusieurs fois dans la file de priorité, avec des distances différentes. L'algorithme de parcours ne traitant que les sommets de la file qui n'en sont pas encore sortis, si l'on insère un sommet x avec une distance δ' et qu'il est déjà présent dans la file avec une distance $\delta \leq \delta'$, cette insertion n'affecte pas le déroulement de notre programme. Si par contre $\delta' < \delta$, c'est l'ancien élément présent dans la file qui est ignoré. **Tout se passe donc comme si la distance δ du sommet x était mise à jour à** $\min(\delta, \delta')$.

Q1

Ecrire la fonction `dijkstrafp (g , s )` qui prend en paramètres un graphe `g` donné comme liste de listes d'ajacence avec pondération et un sommet source `s`. La fonction renvoie la liste des distances à la source des sommets du graphe avec la convention que `None` est infini.

On n'utilise ici que deux couleurs `visité` ou `non visité` (vrai ou faux).

[8]:
```
dijkstrafp(ex2,0)
```

[8]:
```
[0, 6, 9, 13, 17, 15, 12, 18]
```

Q2

On veut maintenant retourner, outre le tableau des distances, la liste des prédecesseurs de chaque sommet x dans un PCC depuis la source s. Des adaptations sont nécessaires dans le code précédent.

'Ecrire

```
def dijkstrafp2(g , s ) :
```

qui réalise ces modifications.

[10]:
```
dijkstrafp2(ex2,0)
```

[10]:
```
([0, 6, 9, 13, 17, 15, 12, 18], [0, 0, 0, 2, 2, 6, 1, 6])
```

[11]:
```
dijkstrafp2(ex2,0)
```

[11]:
```
([0, 6, 9, 13, 17, 15, 12, 18], [0, 0, 0, 2, 2, 6, 1, 6])
```

Q3

Ecrire la fonction `chemin(s,b,peres)` qui prend en paramètre le tableau des prédecesseurz tel que calculé par `dijkstrafp2(G, s)` et renvoie un PCC de s à b (b pour but)

[13]:
```
d,p = dijkstrafp2(cours,0)
chemin(0,1,p)
```

[13]:
```
[0, 4, 1]
```

2 Algorithme A*

Dans toute la suite on veut aller de la source s vers le but b.

L'algorithme A* gère un tableau d des distances, un tableau p pour les prédecesseurs (comme Dijkstra) et un nouveau tableau : le tableau des scores.

Si $d[n]$ représente la distance de s à b, alors le score de n, $f[n]$, est donné par la formule :

$$f[n] = d[n] + h(n)$$

L'algorithme A* stocke les sommets à étudier dans la structure verte. Ici, c'est une file de priorité.

La première différence avec Dijkstra est alors qu'un sommet peut sortir de la file (donc devenir rouge) puis y retourner (donc redevenir vert).

La seconde différence est que le sommet qu'on retire de la file verte est celui de plus petit score et pas celui de plus petite distance à la source.

Dans cette section les labyrinthes sont représentés par des grilles carrées de 0 et de 1. Une case 0 est interdité d'accès, une case 1 est autorisée.

Pour nos tests, donnons nous le labyrinthe suivant :

[15]:
```
maze = [\
 [1, 0, 1, 1, 1, 1, 1, 0, 1],\
 [1, 0, 1, 0, 0, 0, 1, 0, 1],\
 [1, 0, 1, 1, 1, 0, 1, 1, 1],\
 [1, 0, 1, 0, 1, 0, 0, 0, 0],\
 [1, 0, 1, 0, 1, 1, 1, 1, 1],\
 [1, 0, 0, 0, 0, 0, 0, 0, 1],\
 [1, 1, 1, 1, 1, 0, 1, 1, 1],\
 [0, 0, 0, 0, 1, 0, 1, 0, 1],\
 [1, 1, 1, 1, 1, 1, 1, 0, 1]]
```

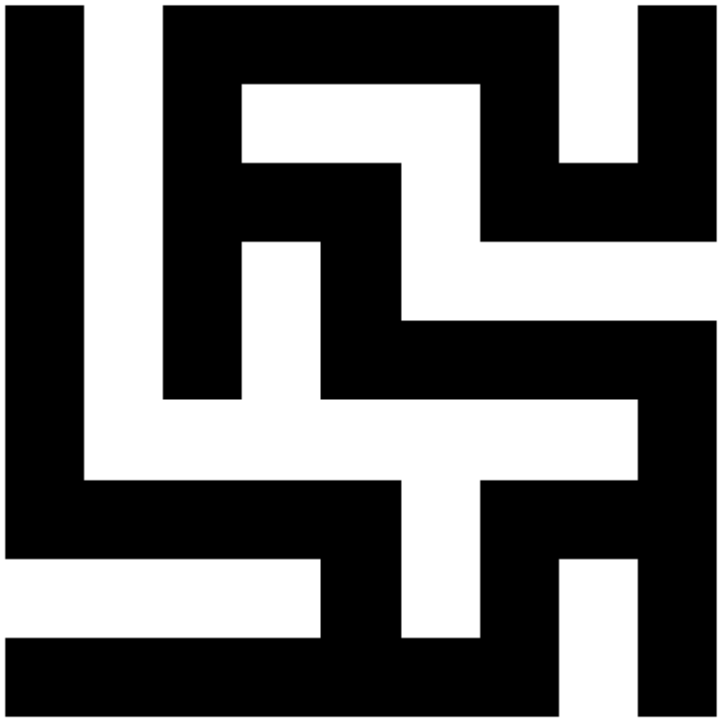
On le comprend ainsi : un individu ne peut se trouver que sur une case marquée 1. Il peut se rendre sur une case voisine marquée 1 en se déplaçant horizontalement ou verticalement mais pas verticalement.

Affichons notre labyrinthe avec `matplolib` :

[16]:
```
import matplotlib.pyplot as plt

def plot_labyrinth(labyrinth):
    plt.imshow(labyrinth, cmap='binary')
    plt.axis('off')
    plt.show()

plot_labyrinth(maze)
```



Il s'agit d'aller de l'entrée (par exemple en case (0,0) vers la sortie (par exemple (0,8)) en n'empruntant que des cases noires, jamais de case blanche.

On décide de considérer le labyrinthe comme un graphe mono-pondéré (les poids des arêtes valent tous 1). Une case marquée 0 n'a pas de voisin. Une case marquée 1 a pour voisins les (au plus) 4 cases à une distance 1 si elles ne sont pas marquées 0.

Ainsi, le sommet $(6, 4)$ a pour voisins $(7, 4)$ et $(6, 3)$ et les arêtes qui les relient ont un pids de 1.

Par nature, les sommets de ce graphe sont donc des cases vues comme des tuples de coordonnées (i, j).

Remarque

Dans la section précédente sur Dijkstra, nos sommets étaient des entiers. Il y certes un moyen simple de numéroter les cases en associant au tuple (i, j) le nombre $Ni + j$ où N est la longueur du labyrinthe. Cette association est en effet bijective. Ce n'est pas ce que nous faisons ici : nos sommets sont bien considérés comme des tuples.

Q1

Ecrire la fonction `voisins (maze,case)` qui retourne les voisins d'une case. Le résultat est une liste de tuples $((i, j), w)$ où (i, j) représentent les coordonnées d'un voisin et w est le poids de l'arête reliant la case en paramètre à celles de coordonnées (i, j).

Remarque

Nous avons décidé que tous les arcs avaient un poids de 1, ce qui fait qu'on peut fort bien se passer du poids w. Toutefois, dans l'objectif futur d'appliquer l'algorithme A* à des graphes dont les arêtes peuvent avoir des valuations différentes, nous maintenons la contrainte de retourner une liste de tuples (coordonnées,poids). Notons que dans le cas d'un graphe à valuations différentes, il suffirait juste de changer la fonction `voisins` et pas celles qui suivent.

[18]:
```
voisins(maze,(0,1)), voisins(maze,(6,4)),voisins(maze,(6,0))
```

[18]:
```
([], [((7, 4), 1), ((6, 3), 1)], [((5, 0), 1), ((6, 1), 1)])
```

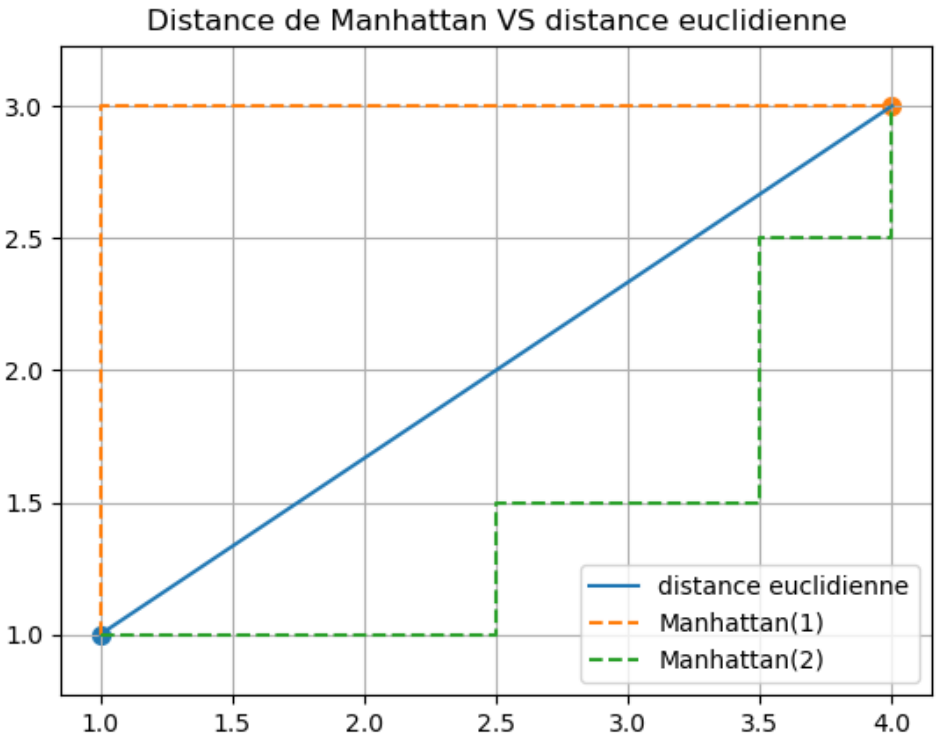
Q2

L'origine du nom de la *distance Manhattan* (ou distance de taxi) vient de la structure en quadrillage régulier des rues du quartier de Manhattan (New York).

C'est la distance entre deux points dans un environnement basé sur une grille (avec uniquement des mouvements horizontaux et verticaux).

La distance de Manhattan est la simple somme des mouvements horizontaux et verticaux, alors que la distance diagonale ou "à vol d'oiseau" peut être calculée en appliquant le théorème de Pythagore.

Une figure avec deux façons différentes de calculer la distance de Manhattan :



1. Ecrire la fonction `manhattan(c1,c2)` qui calcule la distance de Manhattan entre deux cases `c1,c2` données comme des tuples de coordonnées.

[20]:
```
manhattan((3,4),((7,2)))
```

[20]: 6

2. Implémenter la fonction `heur(b)` qui prend en paramètre un sommet donné b (le *but*) et retourne une fonction qui calcule la distance de Manhatan entre son argument et le but. Donc `heur(b)` renvoie la fonction :

$$h : x \mapsto \text{distance de Manhattan de } x \text{ à } b.$$

Q3

Ecrire la fonction `astar(g , s , b ,h)` qui prend en paramètre un graphe pondéré dont les sommets sont des tuples (i, j), une source, un but et une heuristique. La fonction calcule un plus court chemin entre s et b selon l'algorithme A* et renvoie deux dictionnaires :

- le premier associe à un tuples (i, j) sa distance à la source;
- le second associe au même tuples (i, j) son prédecesseur dans un plus court chemin depuis la source.

Notons qu'il peut y avoir des cases non reneignées puisque A* ne passe que par des sommets jugés à un moment utiles dans un chemin de la source au but.

[24]:
```
s = (0,0)
b = (8,8)
h = heur(b)
distance, parent = astar(maze,s,b,h)
distance[b], parent[b]
```

[24]: (20, (7, 8))

[25]:
```
distance[(8,0)]
```

[26]:
```
path = chemin(s,b,parent)
for e in path:
    print(e,end= ", ")
```

```
(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (6, 1), (6, 2), (6, 3),
(6, 4), (7, 4), (8, 4), (8, 5), (8, 6), (7, 6), (6, 6), (6, 7), (6, 8), (7, 8),
(8, 8),
```

[27]:
```
len(path) # correct
```

[27]: 21

On voit bien que le parcours n'aborde que les sommest "utiles", beaucoup sont laissés de côté !

[28]:
```
len([k for k in distance if distance[k] is None and maze[k[0]][k[1]]==1])
```

[28]: 24