MPSI – PTSI

Découverte de l'algorithmique et de la programmation | Informatique

Cours

Chapitre 14
Algorithme A⋆

P. Bessonnat – Xavier Pessoles
MPSI – PTSI

1 Problématique

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.

FIGURE 1 – Exemple de chemin

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).

2 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.

2.1 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.

R

- La valeur calculée peut être mémorisée sous forme d'entier (voir exemple).
- 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 (Fig. 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 du réseau.

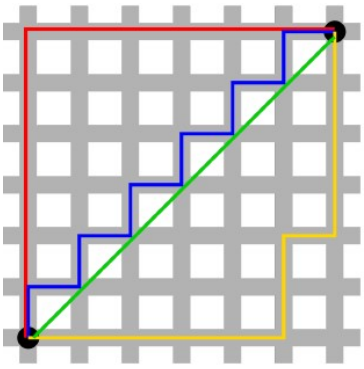
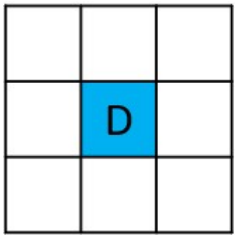


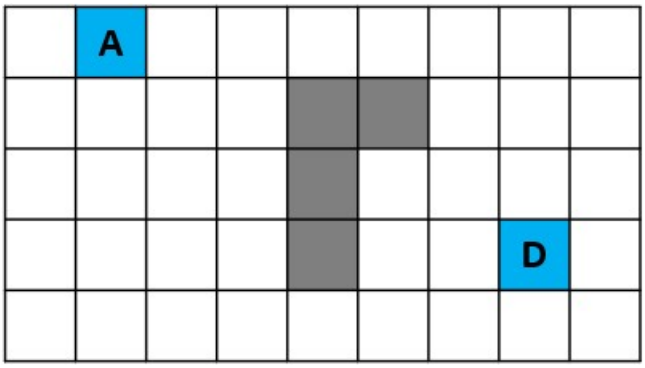
FIGURE 2 – Distance de Manhattan (rouge, bleu, jaune contre distance euclidienne en vert)

2.2 Exemple de construction sur un quadrillage

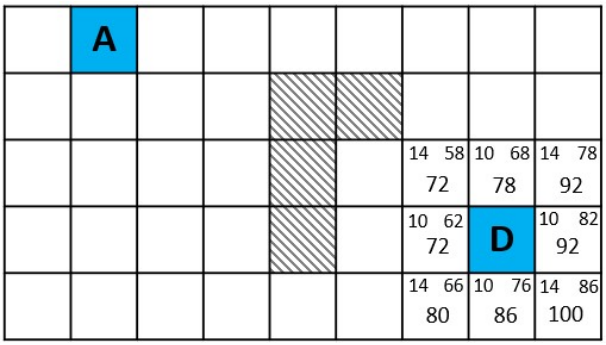
Toutes les cases du quadrillage sont des sommets et les sommets voisins sont les cases ayant un côté ou un angle commun.
Chaque case voisine par le côté est à une distance 10 soit 1×10 et les cases en diagonale à une distance 14 soit $\sqrt{2} \times 10$ arrondi à la valeur entière.



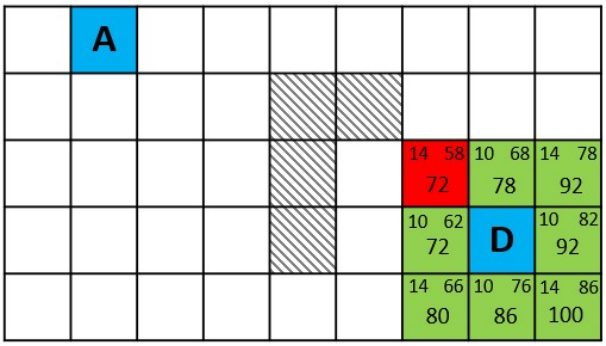
On cherche à déterminer le plus court chemin entre D et A. Sur le quadrillage ci-dessous, on peut reporter les distance de Manhattan de chaque case à **A** :



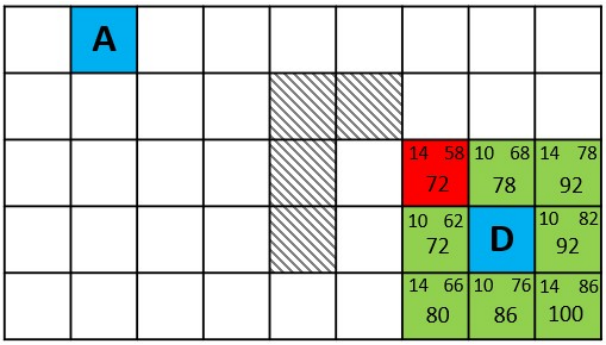
Étape 1 : On évalue la distance à l'origine, la distance de la fin et le coût global



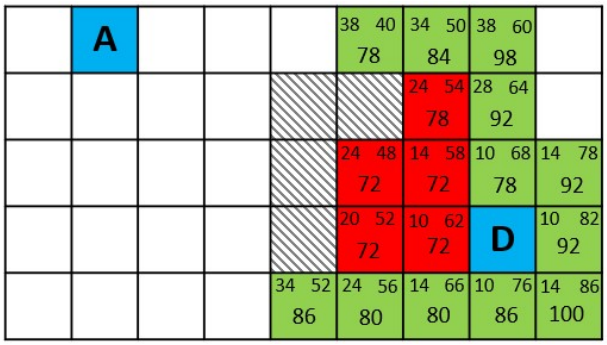
Étape 2 : On choisit le sommet dont le coût global est le plus faible. En cas d'égalité, on choisit le sommet le plus proche de l'arrivée.



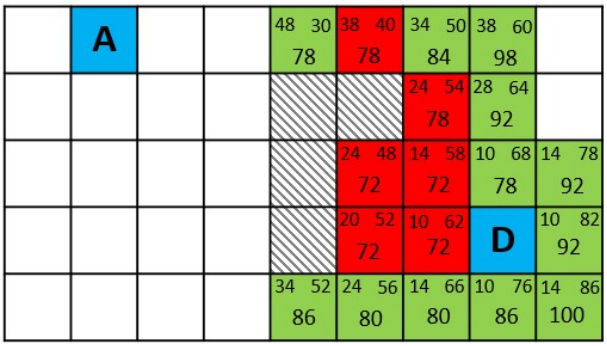
Étapes 3, 4, 5 et 6



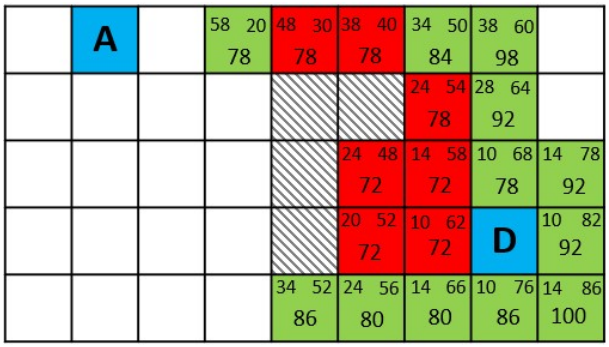
Étape 7



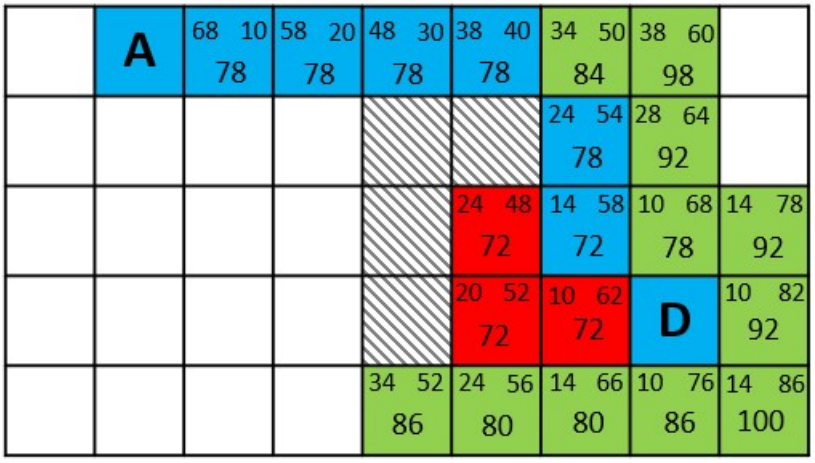
Étape 8



Étape 9 :



Finalement le chemin le plus court :



3 L'algorithme A*

```
### 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 à0
        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
                    M[voisin][0] = G
                    M[voisin][1] = H
                    M[voisin][2] = F
                    # le prédécesseur
                    M[voisin][3] = depart
        # On marque comme "visited"
        visited.append(depart)
        # Une fois que tous les successeurs ont été visités : récursivité
        # On choisit les sommets non visités avec la distance globale la plus courte
        # On ré-exécute récursivement Astar en prenant depart='x'
        not_visited={}
        for k in M.keys():
            if not k in visited :
```

```
            not_visited[k] = M[k][2]
        x=min(not_visited, key=not_visited.get)
        Astar(M,x,fin,visited)
```

Références :

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