|  |  |  |
| --- | --- | --- |
|  | Algorithme A\* | 21/10/2020 |

**I - Introduction :**

**1 • Présentation :**

L’algorithme A\* est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un nœud final. Cet algorithme a été présenté la première fois par Peter E. Hart, Nils John Nilson et Bertram Raphael en 1968 afin d’améliorer les déplacements du robot Shakey.

Il s’agit en fait d’une extension de l’algorithme de Dijkstra.

**2 • Description de l’algorithme :**

On considère qu’il existe deux catégories de nœud : les nœuds à visiter et ceux déjà visités. A chaque itération on sélectionne le meilleur nœud à étudier, c’est-à-dire celui avec le meilleur cout. La définition du cout peut varier : on peut par exemple prendre la somme des distances entre le nœud final et le nœud de départ. Etudier un nœud signifie étudier ses voisins et ainsi ajouter dans la catégorie « a visité » ceux ayant un meilleur cout. Lorsque l’on étudie le nœud qui est le nœud d’arrivé, on remonte les parents de chaque nœud jusqu’à retrouver le nœud de départ.

**II – Implémentation en Python :**

**1 • Choix d’implémentation :**

On choisit de choisir comme graphe un tableau de tableaux remplit des valeurs :

* ‘#’ : Obstacle
* ‘.’ : Passage libre
* ‘@’ : départ
* ‘$’ : arrivée

On choisit de définir le cout d’un nœud comme la somme de sa distance au nœud de départ et de sa distance au nœud d’arrivé.

On choisit de considérer les distances comme des distance de Manhattan afin d’uniquement manipuler des entiers, c’est-à-dire que la distance D entre (i, j) et (k, l) est :

On choisit d’utiliser un dictionnaire afin de stocker les valeurs du graphe.

**2 • Implémentation des nœuds :**

On choisit de définir un nœud par :

* Sa position
* Son parent
* Sa distance au nœud de départ
* Sa distance au nœud d’arrivé
* Son cout

De plus, on dit que deux nœuds sont égaux lorsqu’ils ont la même position et qu’un nœud est supérieur à un autre lorsque son cout est inférieur au cout de l’autre

On obtient alors en python la classe Nœud :

|  |
| --- |
| **class** **Noeud:**  **def** \_\_init\_\_**(**self**,** position**:(),** parent**:()):**  self**.**position **=** position  self**.**parent **=** parent  self**.**g **=** 0 # Distance au Noeud de départ  self**.**h **=** 0 # Distance au Noeud de fin  self**.**f **=** 0 # Cout total  **def** \_\_eq\_\_**(**self**,** other**):**  **return** self**.**position **==** other**.**position  **def** \_\_lt\_\_**(**self**,** other**):**  **return** self**.**f **<** other**.**f |

**3 • Fonctions intermédiaires :**

On peut réaliser une fonction « add\_to\_open » qui indique si un nœud doit etre ajouter a la liste ouverte c’est-à-dire si il n’est pas déjà dans la liste ouverte et si son cout est inférieur au cout des autres nœuds de la liste. On a alors :

|  |
| --- |
| **def** add\_to\_open**(**listeOuverte**,** voisin**):**  **for** noeud **in** listeOuverte**:**  **if** **(**voisin **==** noeud **and** voisin**.**f **>=** noeud**.**f**):**  **return** **False**  **return** **True** |

On réalise aussi une fonction qui calcul la distance de Manhattan entre deux nœuds. On a :

|  |
| --- |
| **def** Distance\_Manhattan**(**a**,**b**):**  **return** abs**(**a**.**position**[**0**]** **-** b**.**position**[**0**])** **+** abs**(**a**.**position**[**1**]** **-** b**.**position**[**1**])** |

Enfin, on réalise une fonction qui prend en entrée un tableau de tableaux et qui renvoie les parametres initiaux pour l’algorithme final. On a :

|  |
| --- |
| **def** carteFromMatrix**(**M**):**  largeur **=** len**(**M**[**0**])**  hauteur **=** len**(**M**)**  map **=** **{}**  deb**,**fin **=** **(**0**,**0**),(**0**,**0**)**  **for** j **in** range **(**hauteur**):**  **for** i **in** range **(**largeur**):**  map**[(**i**,** j**)]** **=** M**[**j**][**i**]**  **if** M**[**j**][**i**]==**'@'**:**  deb **=** **(**i**,**j**)**  **if** M**[**j**][**i**]==**'$'**:**  fin **=** **(**i**,**j**)**  **return** map**,**deb**,**fin**,**largeur**,**hauteur |

**4 • Première implémentation avec des listes :**

Dans cette première implémentation, on utilisera deux List pour stocker les nœuds déjà visité (liste fermée) et les nœuds à visiter (liste ouverte). Pour simplifier la sélection du meilleur nœud, on triera la liste ouverte.

|  |
| --- |
| **def** A\_étoile**(**map**,** deb**,** fin**):**  listeOuverte **=** **[]**  listeFermée **=** **[]**  start\_node **=** Noeud**(**deb**,** **None)**  goal\_node **=** Noeud**(**fin**,** **None)**  listeOuverte**.**append**(**start\_node**)**  **while** len**(**listeOuverte**)** **>** 0**:**  listeOuverte**.**sort**()**  current\_node **=** listeOuverte**.**pop**(**0**)**  listeFermée**.**append**(**current\_node**)**  **if** current\_node **==** goal\_node**:**  path **=** **[]**  **while** current\_node **!=** start\_node**:**  path**.**append**(**current\_node**.**position**)**  current\_node **=** current\_node**.**parent  **return** path  **(**x**,** y**)** **=** current\_node**.**position  Voisins **=** **[(**x**-**1**,** y**),** **(**x**+**1**,** y**),** **(**x**,** y**-**1**),** **(**x**,** y**+**1**)]**  **for** next **in** Voisins**:**  map\_value **=** map**.**get**(**next**)**  **if(**map\_value **==** '#'**):**  **continue**  voisin **=** Noeud**(**next**,** current\_node**)**  **if(**voisin **in** listeFermée**):**  **continue**  voisin**.**g **=** Distance\_Manhattan**(**voisin**,**goal\_node**)**  voisin**.**h **=** Distance\_Manhattan**(**voisin**,**start\_node**)**  voisin**.**f **=** voisin**.**g **+** voisin**.**h  **if(**add\_to\_open**(**listeOuverte**,** voisin**)** **==** **True):**  listeOuverte**.**append**(**voisin**)**  **return** **None** |

Critique : on tri la liste à chaque fois

**5 • Amélioration avec une liste et un tas :**

Au lieu de trier la liste ouverte, on peut choisir d’utiliser une structure triée comme liste ouverte. On propose alors d’utiliser un tas. On utilisera alors l’implémentation du module « headq ».

|  |
| --- |
| **def** A\_étoile\_Tas**(**map**,** deb**,** fin**):**  listeOuverte **=** **[]**  listeFermée **=** **[]**  start\_node **=** Noeud**(**deb**,** **None)**  goal\_node **=** Noeud**(**fin**,** **None)**  heapq**.**heappush**(**listeOuverte**,**start\_node**)**  **while** len**(**listeOuverte**)** **>** 0**:**  current\_node **=** heapq**.**heappop**(**listeOuverte**)**  listeFermée**.**append**(**current\_node**)**  **if** current\_node **==** goal\_node**:**  path **=** **[]**  **while** current\_node **!=** start\_node**:**  path**.**append**(**current\_node**.**position**)**  current\_node **=** current\_node**.**parent  **return** path  **(**x**,** y**)** **=** current\_node**.**position  Voisins **=** **[(**x**-**1**,** y**),** **(**x**+**1**,** y**),** **(**x**,** y**-**1**),** **(**x**,** y**+**1**)]**  **for** next **in** Voisins**:**  map\_value **=** map**.**get**(**next**)**  **if(**map\_value **==** '#'**):**  **continue**  voisin **=** Noeud**(**next**,** current\_node**)**  **if(**voisin **in** listeFermée**):**  **continue**  voisin**.**g **=** Distance\_Manhattan**(**voisin**,**goal\_node**)**  voisin**.**h **=** Distance\_Manhattan**(**voisin**,**start\_node**)**  voisin**.**f **=** voisin**.**g **+** voisin**.**h  **if(**add\_to\_open**(**listeOuverte**,** voisin**)** **==** **True):**  heapq**.**heappush**(**listeOuverte**,**voisin**)**  **return** **None** |

**6 • Comparaison :**

On se propose de réaliser un programme qui compare le temps d’exécution des deux algorithmes précédents. De plus on réalisera une fonction qui créer des tableaux de tableaux aléatoirement.

|  |
| --- |
| **def** matriceAléa**(**n**,**p**):**  res**=[]**  ligne0**=[]**  lignefinal**=[]**  **for** i **in** range**(**n**):**  ligne0**.**append**(**'#'**)**  res**.**append**(**ligne0**)**  **def** nvligne**(**n**):**  res**=[]**  **for** i **in** range**(**n**):**  **if** i**==**0 **or** i**==**n**-**1**:**  res**.**append**(**'#'**)**  **elif** random**.**randint**(**0**,**100**)>=**20**:**  res**.**append**(**'.'**)**  **else:**  res**.**append**(**'#'**)**  **return** res  **for** i **in** range**(**p**-**2**):**  res**.**append**(**nvligne**(**n**))**  **for** i **in** range**(**n**):**  lignefinal**.**append**(**'#'**)**  res**.**append**(**lignefinal**)**  res**[**1**][**1**]=**'$'  res**[**p**-**2**][**n**-**2**]=**'@'  **return** res  **def** chrono**(**f**,**g**,**n**):**  m**=**carteFromMatrix**(**matriceAléa**(**n**,**n**))**  debf **=** time**.**time**()**  f**(**m**)**  finf **=** time**.**time**()**  debg **=** time**.**time**()**  g**(**m**)**  fing **=** time**.**time**()**    **return** finf**-**debf**,**fing**-**debg |

On obtient :

|  |  |
| --- | --- |
|  | On constate que la version utilisant un tas est plus efficace que la version utilisant une liste.  Cependant, on peut se rendre compte que la complexité sera toujours polynomiale (quadratique ou cubique par exemple) |

**II – Etude mathématique :**

**1 • Complexité temporelle :**

On se propose de calculer la complexité de l’algorithme. On note n le nombre de ligne et de colonne du graphe.

Calcul de la complexité temporelle de « add\_to\_open » :

Dans le pire cas, chaque voisin ou presque a sa place dans la liste ouverte, alors la longueur de la liste est n².

On a alors une complexité temporelle en O(n²)

Calcul de la complexité :

On remarque que le nombre de tour de boucle est de l’ordre de grandeur de n.

Alors on a :

La complexité temporelle est alors cubique