Implémentation de l'algorithme de Dijkstra et sa variante A* en Python 3 et Tkinter

Boisnier Thomas

4 avril 2018

Sommaire

Introduction **2**

1 Algorithmes **3**
1.1 Principe . . . 3
1.2 Heuristiques . . . 4
1.3 Implémentation . . . 5

2 Historique **6**
2.1 Principe . . . 6
2.2 Implémentation . . . 6

3 Tkinter **8**
3.1 Fenêtre principale . . . 8
3.2 Grille . . . 8
3.3 Menu . . . 9
3.3.1 Animation . . . 9
3.3.2 Résolution . . . 9
3.4 Informations . . . 9
3.5 Configuration . . . 9

Conclusion **10**

Introduction

Au cours de ce document, nous allons examiner comment nous avons implémenter l'algorithme de Dijkstra, ainsi que sa variante A* [Wik] [Pat18] afin d'avoir un rendu graphique à la fois simple et ludique. Nous utiliserons pour cela le langage Python 3 [Fou18] avec Tkinter [Lun05] pour tout ce que est de l'interface utilisateur. Le tout sera basé sur l'implémentation effectué par Xueqiao (Joe) Xu [Xu18].

Chapitre 1

Algorithmes

1.1 Principe

L'algorithme de Dijkstra (de son inventeur Edsger Dijkstra) ainsi que sa variante, le A*, sont utilisés (à l'origine) pour rechercher le plus court chemin entre 2 points (itinéraires routiers, etc).

Pour se faire, les algorithmes utilisent plusieurs éléments :

- un point de départ
- un point d'arrivé
- un calcul de distance
- une heuristique (uniquement pour A*)
- une liste qui contiendra les points visités
- une file à priorité qui contiendra les points accessibles (tri croissant du coût global : distance + heuristique)

On commence au point de départ choisi. On estime ensuite la distance qui sépare ce point du point d'arrivé voulu ainsi que son heuristique, puis on l'ajoute à la file à priorité.

Les algorithmes retirent le premier point de la file d'attente prioritaire (dû au fonctionnement d'une file d'attente, le point au coût global le plus bas est retiré en premier). Si la file d'attente est vide, il n'y a aucun chemin du point de départ au point d'arrivé, ce qui interrompt l'algorithme. Si le point retenu est le point d'arrivée, les algorithmes reconstruisent le chemin complet et s'arrête.

Si le point retenu n'est pas le point d'arrivé, on ajoute tous les points accessibles à la file a priorité (en calculant leurs coût global à partir du point retenu).

Les algorithmes maintiennent également la liste de points qui ont été visités (couramment appelée "closed list"). Si un point nouvellement produit est déjà dans cette liste avec un coût égal ou inférieur, aucune opération n'est faite sur ce point.

Après, l'évaluation du coût du nouveau point au point d'arrivé, ce nouveau point est alors ajouté à la liste d'attente prioritaire, à moins qu'un point identique dans cette liste ne possède déjà une coût inférieure ou égale.

Une fois les trois étapes ci-dessus réalisées pour chaque nouveau point accessible, le point original pris de la file d'attente prioritaire est ajouté à la liste des points vérifiés. Le prochain point est alors retiré de la file d'attente prioritaire et le processus recommence.

1.2 Heuristiques

L'algorithme A* nécessite une heuristique, on va donc en définir 4, de sorte que *node* soit le point actuel, *target* soit le point d'arrivée, et m_x, m_y les coordonées en x et en y des points.

- Manhattan :

```
1  def manhattan(node, target):
2      dx = abs(node.m_x - target.m_x)
3      dy = abs(node.m_y - target.m_y)
4
5      return dx + dy
```

- Euclidien :

```
1  def euclidean(node, target):
2      dx = node.m_x - target.m_x
3      dy = node.m_y - target.m_y
4
5      return sqrt(dx * dx + dy * dy)
```

- Chebyshev :

```
1  def chebyshev(node, target):
2      dx = abs(node.m_x - target.m_x)
3      dy = abs(node.m_y - target.m_y)
4
5      return (dx + dy) - min(dx, dy)
```

- Octile :

```
1  def octile(node, target):
2      dx = abs(node.m_x - target.m_x)
3      dy = abs(node.m_y - target.m_y)
4
5      return (dx + dy) + (sqrt(2) - 2) * min(dx, dy)
```

1.3 Implémentation

L'algorithme de Dijkstra et sa variante, le A*, fonctionnant tous les deux sur le même principe, nous pouvons faire un implémentation unique et ne jouer que sur les heuristiques pour définir quel algorithme nous voulons utiliser.

Fonction principale du déroulement des algorithmes :

```
1  def run(self, start, end):
2      """
3      Recherche du chemin le plus court
4
5      :param start: Noeud de depart
6      :param end: Noeud d'arrivee
7
8      :type start: Node
9      :type end: Node
10
11     :return: list(Node) - Liste ordonee des noeuds du chemin obtenu
12
13     :except ExceptionPathNotFound: Erreur levee en cas de chemin impossible
14     """
15     closed_set = []
16     open_set = []
17
18     current = start
19     current.set_cost_h(self.m_heuristic(start, end))
20     heappush(open_set, current)
21
22     while open_set:
23         current = heappop(open_set)
24
25         if id(current) == id(end):
26             return self.path(current)
27
28         closed_set.append(current)
29
30         for node in self.neighbors(current):
31             if node in closed_set:
32                 continue
33
34             tmp_cost_g = current.m_cost_g + self.distance(current, node)
35
36             if node in open_set:
37                 if node.m_cost_g > tmp_cost_g:
38                     node.set_cost_g(tmp_cost_g)
39                     node.set_parent(current)
40             else:
41                 node.set_cost_g(tmp_cost_g)
42                 node.set_cost_h(self.m_heuristic(node, end))
43                 node.set_parent(current)
44                 heappush(open_set, node)
45
46     raise ExceptionPathNotFound()
```

Chapitre 2

Historique

2.1 Principe

Un des objectifs de cette implémentation étant le rendu du déroulement des algorithmes, il a fallu trouver un solution afin d'avoir quelque chose de fluide et de paramétrable afin d'avoir un contrôle total sur la vitesse. L'idée a donc été de créer un "historique" des actions importante effectué par l'algorithme, que l'on rendra graphiquement une fois tous les calculs terminées.

Nous avons donc défini 4 types d'action dites "importantes" :

- Le choix du noeud actuel (représenté en orange)
- L'ajout d'un noeud dans la file à priorité (représenté en vert)
- L'ajout d'un noeud dans la liste des noeuds visités (représenté en bleu)
- L'ajout d'un noeud au chemin final (représenté en rose)

2.2 Implémentation

L'historique n'étant qu'une liste des actions effectuées, nous avons simplement hérité notre classe de *list* déjà présent en python. Cependant, afin d'avoir plus d'information sur le déroulement de l'algorithme, nous définissons aussi une classe Action qui contiendra les informations que l'on souhaite avoir sur l'action en cours. Nous obtenons donc une liste d'Action, qui ne reste plus qu'à ajouter les actions au moment clé de l'algorithme, puis de parcourir pour retrouvé le déroulement de l'algorithme.

Historique contenant les actions :

```
1  class History(list):
2
3      def __init__(self):
4          super().__init__()
5
6      def add(self, action_type, element, passed_time):
7          """
8          Ajoute une action
9
10         :param action_type: Type d'action effectuee
11         :param element: Element sur lequel l'action a ete effectue
12         :param passed_time: Temps ecoule avant execution de l'action
13
14         :type action_type: ActionType
15         :type element: object
16         :type passed_time: double
17         """
18         self.append(Action(action_type, element, passed_time))
```

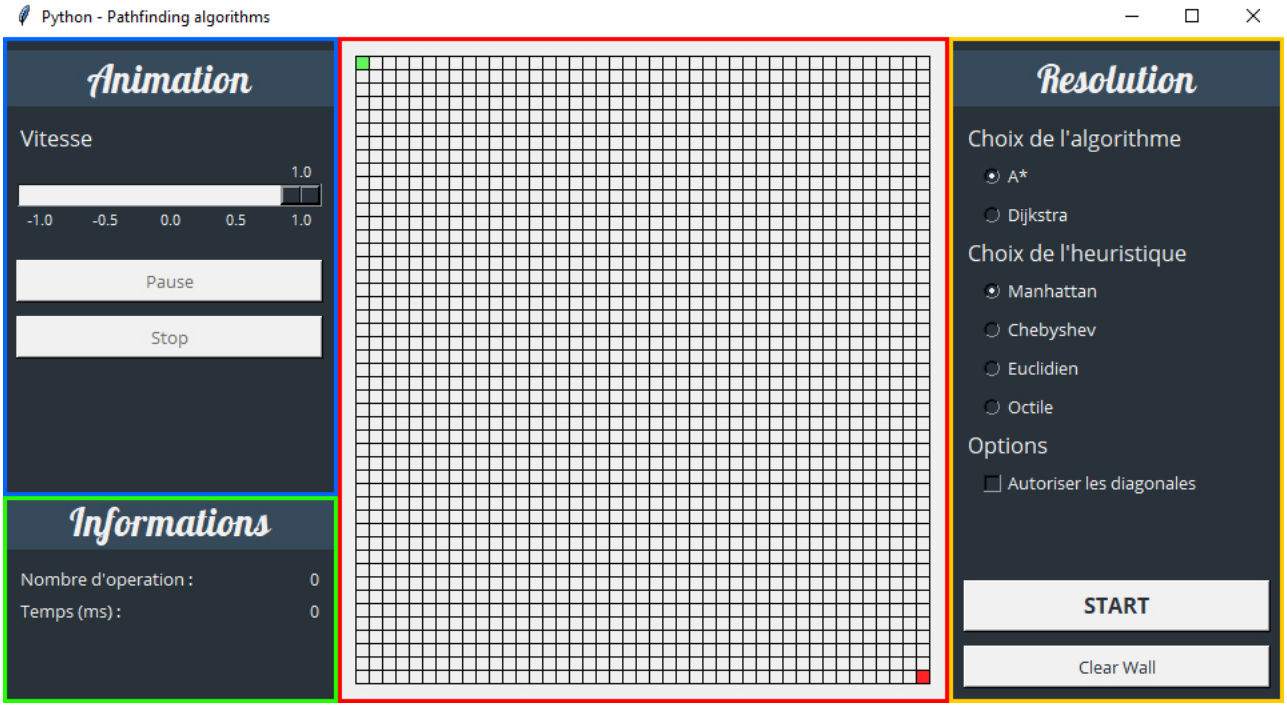
Action contenant les informations :

```
class Action:

    def __init__(self, action_type, element, passed_time):
        """
        Constructeur

        :param action_type: Type d'action effectuee
        :param element: Element sur lequel l'action a ete effectue
        :param passed_time: Temps ecoule avant execution de l'action

        :type action_type: ActionType
        :type element: object
        :type passed_time: double
        """
        self.m_type = action_type
        self.m_element = element
        self.m_passed_time = passed_time
```

Chapitre 3

Tkinter

3.1 Fenêtre principale



Comme nous pouvons le voir, l'interface se divise en 4 parties :

- Le menu de gestion de l'animation
- Le panneau d'affichage des informations
- La grille éditable
- Le menu du choix de l'algorithme et des options

3.2 Grille

Afin d'avoir une grille dont l'édition soit aussi intuitive que possible, toute son ergonomie a été tournée autour du système de drag & drop. Nous pouvons ainsi déplacer les cases de départ, d'arrivée et créer/supprimer des obstacles très simplement.

Pour cela, l'implémentation de la grille sera divisé en 3 parties :

- Les cellules
- Les événements
- La grille

Chaque cellule a 2 états possible :

- vide : représente un déplacement possible
- pleine : représente un obstacle

Les événements permettrons d'interpréter les actions de l'utilisateur.
Et enfin la grille permettra le liens entre tous ces éléments.

3.3 Menu

3.3.1 Animation

L'ordinateur étant bien plus rapide que nous, si l'on impose pas de délai lors de l'animation, celle-ci serai instantané et perdrai tout son intérêt.
Nous avons donc ajouté un curseur pour pouvoir régler la vitesse d'animation. Par ailleurs, une vitesse négative a également été autorisé afin de pouvoir remonter l'animation. Le réglage du délai d'attente va suivre une fonction de façon à ce que celui-ci soit plus fin lorsque l'on réduit la vitesse.

3.3.2 Résolution

Pour avoir le choix de l'algorithme, de l'heuristique et des options que l'on doit utilisé pour la résolution, le menu de résolution sera composé de 3 éléments : 2 listes de boutons radio et des cases à cocher pour les options. Un bouton pour supprimer tous les obstacles de la grille a également été ajouté par simple soucis d'ergonomie.

Lorsque l'on clique sur le bouton "Start", alors la résolution se lance :

1. Activation des boutons pour l'animation
2. Récupération des choix de l'utilisateur
3. Récupération de la position des cases de départ / d'arrivée
4. Lancement de l'algorithme choisi
5. Lancement de l'animation si une solution a été trouvé, sinon affichage d'un message d'erreur

3.4 Informations

Un des objectifs de cette implémentation étant le but pédagogique, l'affichage de certaines informations est important pour avoir des éléments de comparaison entre les différents algorithmes / heuristiques. Actuellement, 2 informations sont récupérées : le temps d'exécution, et le nombre d'opérations. Chaque information est récupérée depuis l'historique des actions, et affiché lors du déroulement de l'animation finale.

3.5 Configuration

Afin de pouvoir personnaliser l'interface facilement, un fichier de configuration a été mis en place. Que ce soit les couleurs, les polices d'écriture, ou les dimensions des différents éléments qui composent l'interface, une variable statique y est associé.

Conclusion

De nombreuses améliorations sont encore possible, comme par exemple :

- Pouvoir changer les dimensions de la grille a partir d'une fenêtre modale ou de la molette de la souris.
- Pouvoir changer la résolution de la fenêtre au démarrage
- Supporter l'ajout d'obstacle en cours de résolution (nécessite l'animation en temps réel)

Cependant, au final, on se rend compte que l'algorithme ne fait pas tout. En effet, même si l'algorithme A* semble plus performant, il dépend encore de l'heuristique que l'on choisi. De plus, même le choix de l'heuristique change la charge de calcul, et donc le temps de résolution, mais plus important encore, la grille sur laquelle on lance la résolution à un impact.

Bibliographie

[Fou18] Python Software Foundation. Python 3.6.5 documentation. https ://docs.python.org/3/, 2001-2018.

[Lun05] Fredrik Lundh. An introduction to tkinter. http ://effbot.org/tkinterbook/, 2005.

[Pat18] Amit Patel. Amit's a* pages. http ://theory.stanford.edu/ amitp/GameProgramming/, 1997-2018.

[Wik] Wikipedia. A* search algorithm. https ://en.wikipedia.org/wiki/A*_search_algorithm/.

[Xu18] Xueqiao (Joe) Xu. Pathfinding. https ://qiao.github.io/PathFinding.js/visual/, 2011-2018.