ALGORITHME GLOUTON ET LABYRINTHE

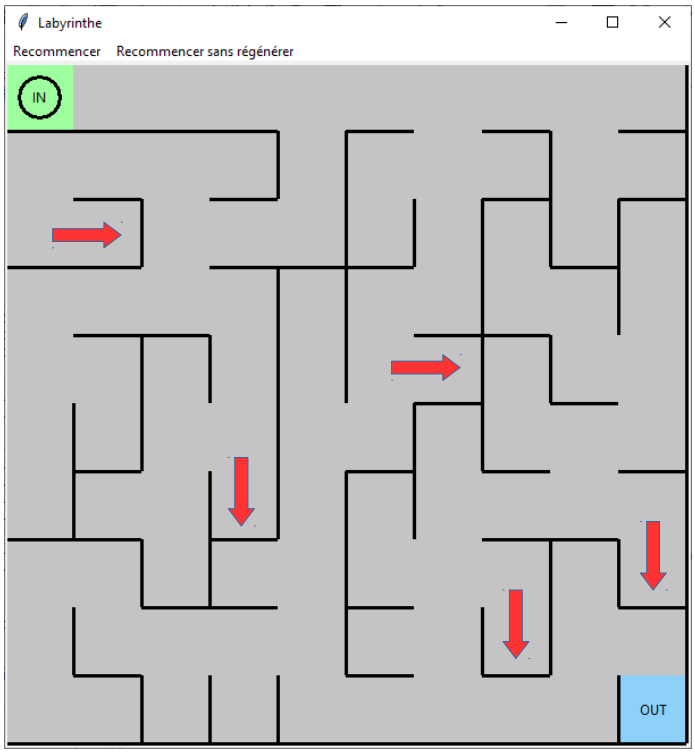
EXPOSE DU PROBLEME

On dispose d'un labyrinthe "parfait". C'est-à-dire sans boucle.
On envisage d'utiliser l'algorithme "glouton" pour sortie du labyrinthe.
On imagine une souris dans le labyrinthe. La souris disposerait d'un indice sur la distance par rapport à la sortie.

DEFAUTS DE L'ALGORITHME GLOUTON

On rappelle que l'algorithme glouton consiste à considérer les solutions optimales locales. Donc, la souris a tendance à préférer les cases les plus proches de la sortie.

On peut imaginer facilement que l'algorithme glouton ne permet pas de sortir des "cul de sac", des voies sans issue. La souris aurait tendance à se coincer dans les culs de sac, à moins de déroger à la règle et de ne plus préférer les cases les plus proches de la sortie.



Les flèches rouges illustrent la souris coincée dans les culs de sac

Un "cul de sac" est une case fermée de tous les côtés sauf celui par où on est arrivé.

SOLUTION

La solution consiste à changer d'algorithme lorsqu'on se rend compte d'être coincé dans un cul de sac.
On imagine que la souris dispose d'un moyen de marquer son passage. Lorsqu'elle est coincée dans un cul de sac, elle ferait demi-tour toute en marquant les cases derrière elle. Les cases du cul de sac seront considérées comme définitivement fermées.

MISE EN ŒUVRE : PROGRAMME EN LANGAGE PYTHON

Programme : labyrinthe_generator.py

Pour pouvoir tester l'algorithme, il faut disposer d'un labyrinthe "parfait" sans boucle.
Pour cela, on dispose d'un programme générateur de labyrinthes "labyrinthe_generator.py".

Le programme "labyrinthe_generator.py" utilise l'algorithme "fusion aléatoire de chemin" pour générer le labyrinthe.
Référence : https://fr.wikipedia.org/wiki/Mod%C3%A9lisation_math%C3%A9matique_de_labyrinthe

Le labyrinthe généré est "parfait" et sans boucle.

Le programme comporte :

- l'objet "Labyrinthe" qui représente un labyrinthe.
  La méthode "Generate()" lance la génération du labyrinthe.
- l'objet "LabCase" qui représente une case du labyrinthe. L'objet "Labyrinthe" dispose d'une liste de "LabCase".
  Les côtés d'une case est modélisé par un tableau d'entiers "side[]". L'index 0 correspond à la case du haut, 1 = case à droite, 2 = case en bas, 3 = case à gauche. Une valeur de 0 correspond à un côté fermé, un 1 à un côté ouvert.
  Chaque case comporte une variable "value" qui servira à marquer les cases pour la génération du labyrinthe par l'algorithme "fusion aléatoire des chemins" (=numéro des chemins). Elle est réexploitée pour la résolution du labyrinthe pour marquer les cases.

Programme : labyrinthe_solver.py

Ce programme implante les algorithmes de résolution du labyrinthe décrits précédemment, c'est-à-dire l'algorithme glouton associé à un autre algorithme de "sortie de cul de sac".

Le programme implémente la classe d'objet "LabSolver" qui dérive de la classe "Labyrinthe".
La méthode "Solve()" lance la résolution du labyrinthe.

Pour la résolution du labyrinthe, on réexploite la variable "value" de "LabCase" pour marquer les cases.

- "value" = 0, la case n'a pas encore été exploitée.
- "value" = 1, la souris est déjà passé par cette case. A la fin de la résolution, ces cases représentent le chemin pour sortir du labyrinthe.
- "value" = -1, la case est dans un "cul de sac", elle est désormais fermée. La souris ne doit plus passer dessus.

Programme : test_solver.py

Il faut lancer le programme "solver.py" pour tester le programme.
Le programme de test est dans la fonction "Test()" dans le programme "labyrinthe_solver.py". Comme c'est une fonction, il faut la lancer depuis le programme principal. Le programme "test_solver.py" fait appel à la fonction "Test()".

TRAVAIL DEMANDE

Activité 1

Dans le programme "labyrinthe_solver.py" qui vous est fourni, il manque la partie qui applique l'algorithme glouton dans la fonction "Solver()" de la classe "LabSolver". C'est après le commentaire :

```
# Déterminer la case d'à côté plus proche de la sortie
```

Complétez le code.

Indications

- On rappelle qu'avec la programmation orientée objet, quand on fait appelle à une variable ou une méthode (= fonction) qui appartient à l'objet, il faut rajouter "self." en préfixe. Exemple :

  La classe LabSolver dispose d'une variable "pos". Si l'on souhaite accéder à cette variable, il faut écrire "self.pos".

- La position courant de la souris est représenté par la variable "pos" qui est en fait un tableau de 2 valeurs. "self.pos[0]" c'est la colonne (abscisse, coordonnée x), et "self.pos[1]" c'est la ligne (ordonnée, coordonnée y).

- Les cases d'entrée et de sortie sont respectivement représentées par les variables "entree" et "sortie".

- Pour calculer la distance d'une case par rapport à la sortie, on dispose de la fonction "DistanceSortie(x,y)". Elle calcule la distance euclidienne de la position donnée (x,y) par rapport à la position de la case de sortie (dans la variable "sortie").

- Les cases du labyrinthe sont représentées par la classe d'objet "LabCase" dans le fichier "Labyrinthe_generator.py". Dans "LabCase", il y a un tableau "side[]" qui représente l'état de chaque côté. Il s'agit d'un tableau d'entier. L'index 0 correspond à la case du haut, 1 = case à droite, 2 = case en bas, 3 = case à gauche. Une valeur de 0 correspond à un côté fermé, un 1 à un côté ouvert.
  Avant le code à compléter, on a constitué un tableau de côté traversables dans la variable "s". Il s'agit en fait d'un tableau d'entiers, mais comportant uniquement les côtés traversables codé selon le principe précédent.

- Pour récupérer l'instance d'objet qui représente la case d'à côté, on dispose de la fonction "GetNextCase(x,y,i)". "x" et "y" sont les coordonnées de la case actuelle. Pour notre problème, ce sera les coordonnées actuelles de la souris, autrement dit, les coordonnées dans la variable "pos". "i" est l'index du côté à traverser. Exemple, si l'on souhaite récupérer la case à droite de la case actuelle :

  ```
  self.GetNextCase(self.pos[0], self.pos[1], 1)
  ```

- A la fin du code à compléter, on doit fourni une variable "cmin" qui représente la case d'à côté qui serait la plus proche de la sortie.

- L'algorithme du code à compléter peut se décrit comme suite :

  On dispose de 2 variables, l'une "mindist" pour représenter la distance minimale trouvée et l'autre "cmin" la case d'à côté correspondante à cette distance minimale. On initie la variable de la distance "mindist" à une grande valeur "math.inf" par exemple, et "cmin" à "None".

  On effectue une boucle sur le tableau "s".

  Pour chaque itération, on récupère la case d'à côté correspondant, et on calcule sa distance par rapport à la sortie.

  Si la distance est inférieure à "mindist", alors on met à jour "mindist" avec cette distance, et on mémorise la case d'à côté correspondante dans la variable "cmin".

  A la sortie de la boucle, on devrait avoir dans la variable "cmin" la case d'à côté la plus proche de la sortie.

Activité 2

On se propose d'essayer d'autres algorithme que l'algorithme glouton afin de constater les effets sur la résolution du labyrinthe.

Pour cela, on peut remplacer le bloc de code qui a été fait précédemment par d'autres codes.

Code 1 : la première case

Au lieu de choisir la case la plus proche de la sortie, on voudrait choisir la première case du tableau "s".
Donc, le code à mettre à la place serait :

```
cmin = self.GetNextCase(self.pos[0], self.pos[1] ,s[0])
```

Faîtes le remplacement, et constatez le résultat de la résolution. A comparer avec l'algorithme glouton.

Code 2 : une case au hasard

Au lieu de la première case du tableau "s", on souhaite choisir une case au hasard dans le tableau "s".
Cela pourrait se faire avec le code suivant :

```
cmin = self.GetNextCase(self.pos[0], self.pos[1] ,s[random.randint(0,len(s)-1)])
```

Faîtes le remplacement, et constatez le résultat de la résolution. A comparer avec l'algorithme glouton.