ALGORITHME GLOUTON ET LABYRINTHE (FICHE PROF)

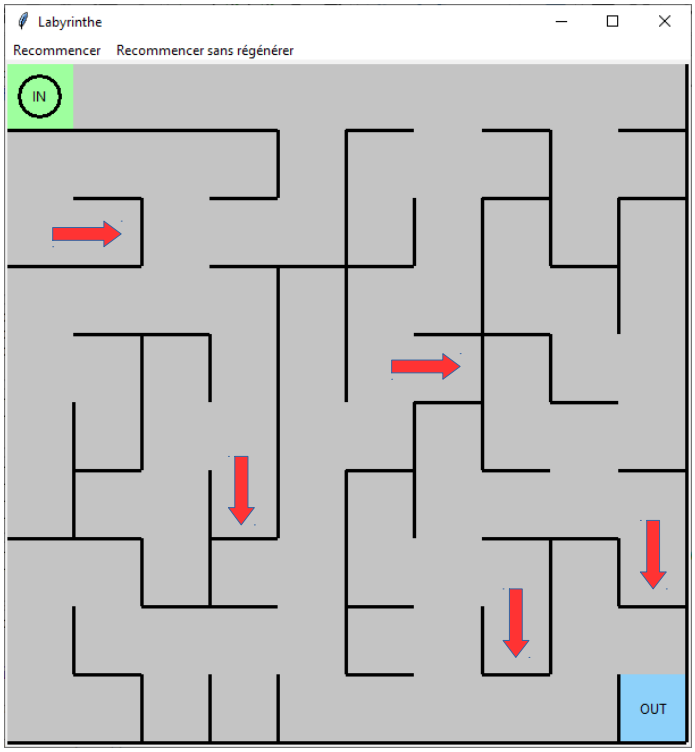
EXPOSE DU PROBLEME

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()".

ASPECT PEDAGOGIQUE

Référentiel

L'activité traite du point "Algorithmique > Algorithmes gloutons" du référentiel de programme de 1ère spécialité NSI.

Contenus	Capacités attendues	Commentaires
Algorithmes gloutons	Résoudre un problème grâce à un algorithme glouton.	Exemples : problèmes du sac à dos ou du rendu de monnaie. Les algorithmes gloutons constituent une méthode algorithmique parmi d'autres qui seront vues en terminale.

Activités

On peut proposer une activité dans laquelle, on donne le programme complet sauf pour la partie qui applique le principe de l'algorithme glouton, c'est-à-dire, la portion de code dans la fonction "Solve()" de "LabSolver" qui recherche la case accessible la plus proche de la sortie.

```
# Déterminer la case d'à côté plus proche de la sortie
mindist = math.inf
cmin = None
for i in s:
    c2 = self.GetNextCase(self.pos[0], self.pos[1] ,i)
    d = self.DistanceSortie(c2.x, c2.y)
    if (d < mindist):
        mindist = d
        cmin = c2
```

Avant ces lignes, on dispose d'un tableau "s" des côtés menant à une case accessible. Il faudra balayer ce tableau, récupérer la case adjacente et vérifier si la distance à la sortie est plus courte.

On peut aussi proposer de ne pas appliquer le principe de la distance la plus court, mais une case au hasard ou la première case du tableau. Et vérifier que dans ces cas-là le chemin est plus long ou la résolution plus difficile.
Pour choisir systématiquement la première case accessible, il faut remplacer le bloc précédent par :

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

Pour choisir aléatoirement une case :

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

On peut faire constater que l'algorithme glouton est quand même plus efficace.

L'étude de l'algorithme de "sortie de cul de sac" peut également être intéressante, même si on sort du sujet "algorithme glouton".
Cet algorithme comporte un défaut qui fait que le parcourt de la souris n'est pas identique d'une résolution à l'autre en ce qui concerne le parcourt des culs de sac. Ceci est dû au choix du chemin à la première case avant la sortie du cul de sac. Dans le programme, le choix est aléatoire. Il faudrait que l'on reprenne l'algorithme glouton dès la 1ère case après la sortie.
On peut proposer cette correction comme activité complémentaire.