projet PacMan

1 Introduction

Dans ce projet, vous allez définir un agent Pacman qui devra trouver son chemin dans un monde labyrinthique, pour atteindre un endroit précis tout en ramassant de la nourriture. Pour cela, vous allez construire des algorithmes de recherche généraux et les appliquer à des scénarios de Pacman.

1.1 Fichiers du projet

Le code pour ce projet se compose de plusieurs fichiers Python, dont certains que vous aurez besoin de lire et de comprendre afin de terminer le travail, et certains que vous pourrez ignorer. Vous pouvez télécharger tout le code et les fichiers associés (y compris la présente description) sous forme d'archive zip.

- Fichiers à éditer :
  - search.py
    Fichier contenant l'ensemble de vos algorithmes de recherche.
  - searchAgents.py
    Fichier contenant l'ensemble des programmes gérant la recherche de l'agent.
- Fichiers que vous voudrez peut-être regarder :
  - pacman.py
    Fichier principal de fonctionnement du jeux Pacman. Ce fichier décrit un état du jeu Pacman que vous utilisez dans ce projet.
  - game.py
    La logique qui fait fonctionner le monde Pacman. Ce fichier décrit plusieurs types de soutien comme AgentState, Agent, Direction, et Grid.
  - util.py
    Structures de donnée utiles pour l'implémentation des algorithmes de recherche.
- Fichier que vous pouvez ignorer (mais que vous pouvez connaître aussi) :
  - graphicsDisplay.py
    Interface graphique pour Pacman
  - graphicsUtils.py
    Fonctions d'aide pour l'interface graphique de Pacman
  - textDisplay.py

ASCII pour Pacman
— ghostAgents.py
Agents de control pour les fantômes
— keyboardAgents.py
Interface clavier pour contrôler Pacman
— layout.py
Code pour la lecture de fichiers et le stockage de leur contenu

1.2 Ce que vous devez rendre

Vous devrez compléter certaines parties de search.py et searchAgents.py.

Chaque groupe de 2 personnes deva soumettre ces deux fichiers (seulement) avec un fichier rapport.pdf. Votre rapport devra reprendre de manière synthétique les problèmes rencontrés et les solutions apportées.

1.3 Evaluation

Votre code sera évalué pour vérifier l'implémentation. Il est interdit de changer les noms des fonctions ou des classes prévues dans le code, sinon, l'évaluation sera considérée comme intégralement fausse. Cependant, l'exactitude de votre mise en oeuvre sera le point principal de l'évaluation.

Si nous constatons des dysfonctionnement dans le groupe, une notation individuelle sera réalisée.

Par ailleurs, si nous constatons que deux rendus sont identiques avec quelques modifications mineures, les deux groupes seront largement sanctionnés (souvenez-vous que si le copier-coller est facile, détecter le copier-coller est encore plus facile).

1.4 Vous n'êtes pas seul

N'hésitez pas à nous solliciter pendant la durée du projet. Nous serons là pour vous aider. Cependant, avant de nous solliciter, chercher les solutions par vous même.

2 Bienvenue chez Pacman

Après avoir téléchargé le code (PacMan.zip), unzippez le et déplacez le dans un répertoire dédié. Vous devriez être capable de lancer votre première partie ! Pour cela essayer la ligne de commande :

```
python pacman.py
```

Pacman vit dans un petit monde bleu composé de couloirs et de nourriture blanche ! Naviguer dans ce monde de manière efficace sera la première étape de Pacman dans la maîtrise de son domaine. Le plus simple agent dans searchAgents.py est le GoWestAgent, qui va toujours vers l'Ouest

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

Mais les choses tournent court pour cet agent s'il devient nécessaire de tourner :

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

Si pacman est bloqué, vous pouvez quitter le jeu en tapant CTRL-C dans votre terminal. Bientôt, votre agent pourra résoudre non seulement tinyMaze, mais également des labyrinthes plus compliqués.Vous noterez que pacman.py peut prendre un certain nombre d'options.Vous pouvez accéder à la liste des options (et des valeurs par défaut) avec la commande :

```
python pacman.py -h
```

En outre, toutes les commandes qui apparaissent dans ce projet sont listées dans le fichier commands.txt, depuis lequel vous pouvez copier-coller facilement (Sous UNIX / Mac OS X, vous pouvez même exécuter toutes ces commandes dans l'ordre avec bash commands.txt).

3 Recherche de la nourriture avec des algorithmes de recherche

Dans searchAgents.py, vous trouverez un SearchAgent entièrement implémenté, qui prévoit un chemin à travers le monde de Pacman, puis exécute ce chemin étape par étape.

Les algorithmes de recherche pour la formulation d'une stratégie de recherche ne sont pas implémentés. c'est votre travail. Tout d'abord, vérifier que le SearchAgent fonctionne correctement en exécutant :

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

La commande ci-dessus indique au SearchAgent d'utiliser tinyMazeSearch comme algorithme de recherche (séquence d'actions pour résoudre tinyMaze ; ne marchera pas dans un autre contexte), qui est implémenté dans search.py. Pacman doit naviguer dans le labyrinthe avec succès.

Remarque importante

Toutes vos fonctions de recherche doivent retourner une liste d'actions qui mèneront l'agent de sa position initiale à l'objectif. Toutes ces actions doivent être des coups acceptables (directions valides, ne se déplaçant pas à travers les murs). Rappelez-vous que les noeuds de recherche ne doivent pas seulement contenir l'état, mais aussi les informations nécessaires pour reconstruire le chemin vers cet état.

Aide

— Tous les algorithmes sont très similaires. DFS, BFS, UCS, A* ne diffèrent que dans la gestion de la frontière. Donc, concentrez-vous sur DFS et le reste devrait être relativement simple.

— Utilisez les structures de pile, file et PriorityQueue qui sont fournies dans util.py !

Question 1 *(3 points) Implémentez l'algorithme depth-first search (DFS) (voir cours 2 sur les algorithmes de recherche, slide 20) dans la fonction depthFirstSearch du fichier search.py. Pour rendre votre algorithme complet, écrivez la version de DFS pour les graphes qui empêche l'expansion d'un noeud déjà visité. Votre code devrait rapidement trouver une solution pour :*

```
python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

La représentation graphique du monde de Pacman va proposer une superposition des états explorés, et l'ordre dans lequel ils ont été explorées (rouge vif signifie exploration plus tôt). Est-ce que l'ordre d'exploration est celui attendu ? Est-ce que Pacman visite tous les angles sur son chemin vers le but ?

Aide

Si vous utilisez une pile comme structure de données, la solution trouvée par votre algorithme DFS pour mediumMaze doit avoir une longueur de 130 Est-ce une solution à moindre coût ? Si non, pensez aux problèmes de la recherche en profondeur.

Question 2 *(2 point) Mettez en œuvre l'algorithme de recherche en largeur (BFS) (voir cours 2 sur les algorithmes de recherche, slide 3) dans la fonction breadthFirstSearch du fichier search.py. Encore une fois, écrivez un algorithme de recherche qui permet d'éviter l'expansion des états déjà visités. Testez votre code de la même manière que vous l'avez fait pour la recherche en profondeur d'abord.*

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

Est-ce que BFS trouve une solution de moindre coût ? Sinon, vérifiez votre implémentation.

Aide *Si Pacman se déplace trop lentement pour vous, essayez l'option*

```
--frameTime=0
```

Note

Si vous avez écrit votre code de recherche de manière générique, votre code devrait fonctionner aussi bien pour le problème de recherche pour le taquin à huit pièces sans modification.

```
python eightpuzzle.py
```

4 Variation de la fonction de coût

Alors que BFS trouve un chemin qui utilise le moins d'actions, nous pourrions trouver des chemins qui sont "meilleurs" dans un autre sens. Considérez mediumDottedMaze et mediumScaryMaze. En changeant la fonction de coût, nous pouvons encourager Pacman à trouver d'autres chemins. Par exemple, nous pouvons augmenter le cout des chemins où les fantômes sont présents où diminue la valeur pour les zones riches en nourriture. L'agent Pacman doit alors modifier son comportement en conséquence

Question 3 *(3 points) Implémentez l'algorithme de recherche à coûts uniformes (voir cours 2 sur les algorithmes de recherche, slide 12) dans la fonction uniformCostSearch du fichier search.py. Nous vous encourageons à regarder util.py pour certaines structures de données qui peuvent être utiles dans votre application. Vous devriez maintenant observer avec succès le comportement de Pacman dans les trois situations suivantes, où les agents ci-dessous sont tous des agents UCS qui ne diffèrent que dans la fonction de coût qu'ils utilisent (les agents et les fonctions de coût sont écrits pour vous) :*

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Note *Vous devriez obtenir les coûts de chemin très bas et très hauts pour StayEastSearchAgent et StayWestSearchAgent, en raison de leurs fonctions de coût exponentiel (voir searchAgents.py pour plus de détails).*

5 A* search

Question 4 *(3 points) Implémentez une recherche A* (voir cours 3 sur les heuristiques, slide 30) dans la fonction aStarSearch du fichier search.py. A* prend une fonction heuristique en paramètre. Heuristics prend deux arguments : un état de l'espace d'états et le problème lui-même. La fonction nullHeuristic du fichier search.py est un exemple trivial.*

Vous pouvez tester votre version de A sur le problème de départ de recherche d'un chemin dans le labyrinthe d'une position donnée, en utilisant l'heuristique dite de Manhattan (qui est implémentée par manhattanHeuristic dans le fichier searchAgents.py).*

```
python pacman.py -l bigMaze -z .5 -p SearchAgent
   -a fn=astar,heuristic=manhattanHeuristic
```

Vous verrez que A trouve la solution optimale plus rapidement que UCS (environ 549 vs 620 noeuds explorés dans notre implémentation, mais vous pouvez trouver d'autres valeurs).*

6 Manger toutes les pastilles...

Maintenant, nous allons résoudre un problème difficile : que Pacman mange toute la nourriture en un minimum d'étapes. Pour cela, nous devons définir le problème de manger toute la nourriture : FoodSearchProblem dans le fichier searchAgents.py (implémenté pour vous). Une solution est définie par le chemin qui passe par toute la nourriture dans le monde de Pacman. Pour le présent projet, les solutions ne prennent pas en compte les fantômes ou le pouvoir de certaines nourritures, elles ne dépendent que de la position des murs dans le labyrinthe et du comportement de Pacman. Si vous avez correctement implémenté votre méthode de recherche A*, avec une heuristique non-nulle (equivalent à UCS), elle doit normalement trouver rapidement une solution à testSearch sans modifier le code (cout total de 7).

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

Note AStarFoodSearchAgent est un raccourci pour

```
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

Vous devriez trouver que UCS commence à ralentir, même pour les tinySearch apparemment simple. Notre mise en œuvre prend 2,5 secondes pour trouver un chemin de longueur 27 après expansion de 5169 nœuds de recherche.

Question 5 *(5 points) Remplir foodHeuristic dans le fichier searchAgents.py avec une heuristique consistente pour FoodSearchProblem. Essayez votre agent sur trickySearch :*

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Notre UCS agent trouve la solution optimale en environ 13 secondes, en explorant 16 000 noeuds. Toute heuristique non-triviale et consistante aura 1 point et vous aurez les points supplémentaires avec la règle suivante :

Moins de tant de noeud :	*Points*
15000	*1*
12000	*2*
9000	*3 (medium)*
7000	*4 (hard)*

Si votre heuristique est inconsistante, vous n'aurez pas de point. Toute heuristique avec des valeurs négatives ou ayant pour valeur 0 pour les noeuds buts perdra des points.

7 Recherche sous-optimale

Parfois, même avec A* et une bonne heuristique, trouver une solution passant par toutes les nourritures est difficile. Dans ces cas, nous serions tout de même heureux de trouver rapidement un chemin assez bon. Dans cette section, vous allez écrire un agent qui mange toujours la plus proche nourriture. ClosestDotSearchAgent est implémenté pour vous dans searchAgents.py, mais il manque une fonction importante qui trouve un chemin vers la plus proche nourriture.

Question 6 *(3 points) Implémentez la fonction findPathToClosestDot dans le fichier searchAgents.py. Notre agent résout le labyrinthe (sous-optimalement) en moins d'une seconde avec un cout pour le chemin de 350.*

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Aide

La façon la plus rapide pour compléter findPathToClosestDot est de remplir AnyFoodSearchProblem à qui il manque le test de but. Ensuite, résoudre ce problème avec une fonction de recherche approprié. La solution doit être très courte !

8 Et les fantômes ?

Il serait normal d'utiliser également les fantômes pour le jeux de Pacman. Dans ce cas, on se retrouve avec plusieurs joueurs qui jouent à leur tour...

Question 7 *(1 point) Quelle(s) stratégie(s) peut on utiliser pour modéliser à la fois les fantômes et Pacman ?*

9 Object Glossary

Voici un glossaire des principaux points dans le code lié aux problèmes de recherche :

- **SearchProblem (search.py)**
  Un SearchProblem est un objet abstrait qui représente un espace d'états, une fonction de successeur, un cout et un état de but du problème. Vous interagirez avec les SearchProblems seulement par les méthodes définies au début de search.py.
- **PositionSearchProblem (searchAgents.py)**
  Un type spécifique de SearchProblem avec qui vous allez travailler. Il correspond à la recherche d'une seule pastille dans un labyrinthe.
- **FoodSearchProblem (searchAgents.py)**
  Un type spécifique de SearchProblem avec qui vous allez travailler. Il correspond à la recherche d'une façon de manger toutes les pastilles dans un labyrinthe.

— **Search Function**
Une fonction de recherche est une fonction qui prend une instance de SearchProblem comme paramètre, applique un algorithme, et retourne une séquence d'actions qui mènent à un but. Des exemples de fonctions de recherche sont depthFirstSearch et breadthFirstSearch, que vous avez à écrire. Vous disposez tinyMaze qui est une fonction de recherche très mauvaise qui ne fonctionne correctement que sur tinyMaze.

— **SearchAgent**
SearchAgent est une classe qui implémente un agent (un objet qui interagit avec le monde) et fait sa planification grâce à une fonction de recherche. Le SearchAgent utilise d'abord la fonction de recherche fournie pour faire un plan des actions à réaliser pour atteindre l'état but, puis exécute les actions une par une.