Travail pratique 1 : IFT-4102/7025, Hiver 2022

Université Laval, Département d'informatique et de génie logiciel

Date de remise: 12 février 2022 à 23h59

1 Recherche dans le monde de Pacman [25 points]

Dans cette partie du travail pratique, votre agent Pacman trouvera des chemins dans son monde de labyrinthe, à la fois pour atteindre un endroit particulier et pour collecter de la nourriture efficacement. Vous allez construire des algorithmes de recherche généraux et les appliquer au monde de Pacman.

Ce projet Pacman a été développé par l'université Berkeley dans le cadre du cours CS188. Les questions et instructions qui suivent sont d'ailleurs essentiellement une traduction de la page web https://inst.eecs.berkeley.edu/~cs188/fa18/project1.html que vous pouvez aussi consulter.

Le code de ce projet se compose de plusieurs fichiers Python (search.zip), certains devront être lus et compris (au moins partiellement), et d'autres peuvent être ignorés.

Fichiers que vous allez modifier et remettre:

search.py	Où résideront tous vos algorithmes de recherche.
searchAgents.py	Où résideront tous vos agents basés sur la recherche.

Fichiers que vous pourriez vouloir regarder:

pacman.py	Le fichier principal qui exécute les jeux Pacman.
game.py	La logique derrière le fonctionnement du monde Pacman.
util.py	Structures de données utiles pour la mise en oeuvre d'algorithmes de recherche.

Les fichiers de support que vous pouvez ignorer:

graphicsDisplay.py	Graphiques pour Pacman
graphicsUtils.py	Support pour les graphiques Pacman
textDisplay.py	Graphiques ASCII pour Pacman
ghostAgents.py	Agents pour contrôler les fantômes
keyboardAgents.py	Interfaces clavier pour contrôler Pacman
layout.py	Code pour lire les fichiers de mise en page et stocker leur contenu
autograder.py	Correcteur automatique
testParser.py	Analyse les fichiers de tests et de solutions pour l'autograder
testClasses.py	Classes de tests générales
`test_cases`	Répertoire contenant les cas de tests pour chaque question
searchTestClasses.py	Les classes de tests spécifiques au présent projet.

Évaluation et remise: Seulement les fichiers search.py et searchAgents.py sont à remettre. Il ne faut pas modifier les autres fichiers car votre code sera exécuter avec les originaux lors de la correction. Un correcteur automatique est fourni, ce qui vous permettra de tester vos implémentations. Veuillez ne pas changer les noms des fonctions ou classes fournies dans le code, sinon le correcteur automatique ne fonctionnera plus.

Nous vérifierons votre code par rapport aux autres soumissions dans la classe pour la redondance logique. Si vous copiez le code de quelqu'un d'autre et le soumettez avec des modifications mineures,

nous le saurons. Ces détecteurs de triche sont assez difficiles à tromper, alors n'essayez pas s'il vous plaît. Nous vous faisons confiance pour soumettre votre propre travail seulement. De plus, puisque ce projet Pacman a été utilisé dans divers cours dans différentes universités au cours des années, beaucoup d'étudiants l'ont fait, et certains d'entre eux ont possiblement rendu publiques leurs solutions. Merci de ne pas plagier ces solutions.

Nous sommes maintenant prêts à commencer! Après avoir téléchargé le code, décompressé, et en changeant dans le répertoire, vous devriez être capable de jouer à un jeu de Pacman en tapant ce qui suit sur la ligne de commande:

```
python pacman.py
```

Pacman vit dans un monde bleu brillant de couloirs sinueux et de délicieuses gâteries rondes. Naviguer efficacement dans ce monde sera la première étape de Pacman dans la maîtrise de son domaine.

L'agent le plus simple dans searchAgents.py s'appelle GoWestAgent, il va toujours à l'Ouest (un agent réflexe trivial). Cet agent peut parfois gagner:

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

Mais, les choses deviennent problématiques pour cet agent lorsqu'il est nécessaire de tourner:

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

Si Pacman est coincé, vous pouvez quitter le jeu en tapant CTRL-c dans votre terminal.

Bientôt, votre agent ne résoudra pas seulement tinyMaze, mais n'importe quel labyrinthe que vous voulez. Notez que pacman.py supporte un certain nombre d'options qui peuvent chacune être exprimées de manière longue (par exemple, --layout) ou de manière courte (par exemple, -l). Vous pouvez voir la liste de toutes les options et leurs valeurs par défaut via:

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

En outre, toutes les commandes qui apparaissent dans ce projet apparaissent également dans commands.txt. Sous UNIX / Mac OS X, vous pouvez même exécuter toutes ces commandes dans l'ordre avec bash commands.txt.

Finalement, lorsque vous avez complété une question, vous pouvez tester votre code avec le correcteur automatique. Par exemple, pour la question 1:

```
python autograder.py -q q1
```

1.1 Exploration en profondeur d'abord [3 points]

Dans searchAgents.py, vous trouverez un SearchAgent entièrement implémenté, qui planifie un chemin dans le monde de Pacman, puis exécute ce chemin étape par étape. Les algorithmes de recherche pour la formulation d'un plan ne sont pas implémentés - c'est votre travail.

Tout d'abord, vérifions 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, implémenté dans search.py. Pacman devrait naviguer dans le labyrinthe avec succès.

Maintenant, il est temps d'écrire des fonctions de recherche génériques à part entière pour aider les plans de Pacman! Rappelez-vous qu'un nœud de recherche doit contenir non seulement un état mais aussi les informations nécessaires pour reconstruire le chemin (plan) qui aboutit à cet état.

Remarque importante: Toutes vos fonctions de recherche doivent renvoyer une liste d'actions qui mèneront l'agent du début à l'objectif. Ces actions doivent toutes être légales (directions valides, pas de déplacement à travers les murs).

Note importante: Assurez-vous d'utiliser les structures de données Stack, Queue et PriorityQueue qui sont fournies dans util.py ! Ces implémentations de structure de données ont des propriétés particulières qui sont requises pour la compatibilité avec l'autograder.

Astuce: Les algorithmes sont très similaires. Donc, concentrez-vous sur profondeur d'abord et le reste devrait être relativement simple. En effet, une implémentation possible ne nécessite qu'une seule méthode de recherche générique qui est configurée avec une stratégie de mise en file d'attente spécifique à l'algorithme. (Votre mise en œuvre n'a pas besoin d'être de ce type pour recevoir un crédit complet).

Implémentez l'algorithme d'exploration en profondeur d'abord (version pour graphes) dans search.py. Votre code devrait trouver rapidement 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
```

Le tableau de Pacman montrera une superposition des états explorés, et l'ordre dans lequel ils ont été explorés (un rouge plus clair signifie une exploration plus tôt). Est-ce que Pacman va réellement à tous les carrés explorés sur son chemin vers le but?

Astuce: Si vous utilisez une Stack comme structure de données, la solution trouvée par votre algorithme profondeur d'abord pour mediumMaze devrait avoir une longueur de 130 (à condition d'insérer les successeurs dans la frange dans l'ordre fourni par getSuccessors, vous pourriez obtenir 246 si vous les insérez dans l'ordre inverse).

1.2 Exploration en largeur d'abord [3 points]

Implémentez l'algorithme largeur d'abord (version pour graphes) dans search.py. Testez votre code de la même manière que pour la recherche en profondeur.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 --frameTime 0
```

Est-ce que l'exploration en largeur d'abord trouve une solution de coût minimal? Sinon, vérifiez votre implémentation.

Remarque: Si vous avez écrit votre code de recherche de manière générique, votre code devrait fonctionner aussi bien pour le problème de recherche du huit-puzzle sans aucun changement.

```
python eightpuzzle.py
```

1.3 Exploration à coût uniforme (UCS) [3 points]

Considérez mediumDottedMaze et mediumScaryMaze. En changeant la fonction de coût, nous pouvons encourager Pacman à trouver des chemins différents. Par exemple, nous pouvons facturer plus pour des étapes dangereuses dans les zones fantômes ou moins pour les étapes dans les zones riches en nourriture, et un agent Pacman rationnel devrait ajuster son comportement en conséquence.

Implémentez l'algorithme de recherche de graphes à coût uniforme dans search.py. Nous vous encourageons à parcourir util.py pour des structures de données qui pourraient vous être utiles dans votre implémentation. Testez votre code:

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

Remarque: Vous devriez obtenir des coûts de chemin très bas pour StayEastSearchAgentet et très élevés pour StayWestSearchAgent, en raison de leurs fonctions de coût exponentielles (voir searchAgents.py pour plus de détails).

1.4 Exploration A* [3 points]

Implémentez une recherche de graphe A* dans la fonction vide aStarSearch dans search.py. A* prend une fonction heuristique comme argument. Les heuristiques prennent deux arguments: un état dans le problème de recherche (l'argument principal), et le problème lui-même (pour l'information de référence). La fonction heuristique nullHeuristic dans search.py est un exemple trivial.

Vous pouvez tester votre implémentation A* sur le problème original consistant à trouver un chemin à travers un labyrinthe vers une position fixe en utilisant l'heuristique de distance de Manhattan (déjà implémentée dans searchAgents.py).

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

Vous devriez voir que A* trouve la solution optimale légèrement plus rapidement que la recherche à coût uniforme (environ 549 contre 620 nœuds de recherche développés dans notre implémentation, mais les égalités en priorité peuvent faire légèrement différer vos nombres). Que se passe-t-il sur openMaze pour les différentes stratégies de recherche?

1.5 Problème des coins: représentation [3 points]

Le véritable pouvoir de A* ne sera apparent qu'avec un problème de recherche plus complexe. Maintenant, il est temps de formuler un tel problème.

Dans les "labyrinthes à coins", il y a quatre points, un dans chaque coin. Notre nouveau problème de recherche est de trouver le chemin le plus court à travers le labyrinthe qui touche les quatre coins (que le labyrinthe ait ou non de la nourriture). Notez que pour certains labyrinthes comme tinyCorners, le chemin le plus court ne va pas toujours à la nourriture la plus proche en premier!

Indice : le chemin le plus court à travers tinyCorners prend 28 étapes.

Remarque: Assurez-vous de compléter la question 2 avant de travailler sur la question 5, car la question 5 s'appuie sur votre réponse à la question 2.

Implémentez le problème de recherche *CornersProblem* dans searchAgents.py. Vous devrez choisir une représentation d'état qui code toutes les informations nécessaires pour détecter si les quatre coins ont été atteints. Maintenant, votre agent de recherche devrait résoudre:

python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

Pour obtenir un crédit complet, vous devez définir une représentation d'état abstraite qui n'encode pas les informations non pertinentes (comme la position des fantômes, où la nourriture supplémentaire est, etc.). En particulier, n'utilisez pas un Pacman GameState comme état de recherche. Votre code sera très, très lent si vous le faites (et aussi faux).

Astuce: Les seules parties de l'état du jeu dont vous avez besoin dans votre implémentation sont la position de départ de Pacman et l'emplacement des quatre coins.

L'implémentation de largeur d'abord étend un peu moins de 2000 nœuds de recherche sur mediumCorners. Cependant, une heuristique (utilisée avec A*) peut réduire la quantité de recherche requise.

1.6 Problème des coins: heuristique [3 points]

Note: Assurez-vous de compléter la question 4 avant de travailler sur la question 6, car la question 6 s'appuie sur votre réponse à la question 4.

Implémentez une heuristique consistante non-triviale pour CornersProblem dans cornersHeuristic.

python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

Remarque: AStarCornersAgent est un raccourci pour

-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic.

Admissibilité et cohérence: Rappelez-vous que les heuristiques ne sont que des fonctions qui prennent des états de recherche et renvoient des nombres qui estiment le coût à un objectif le plus proche. Des heuristiques plus efficaces retourneront des valeurs plus proches des coûts réels. Pour être admissibles, les valeurs des heuristiques doivent être des bornes inférieures du coût du chemin le plus court vers l'objectif le plus proche (et être non négatives). Pour être consistantes, il faut en plus considérer que si une action a coûté c, alors prendre cette action ne peut que provoquer une chute de l'heuristique d'au plus c.

Rappelez-vous que l'admissibilité n'est pas suffisante pour garantir l'optimalité dans la recherche de graphes - vous avez besoin de la notion plus forte de consistance. Cependant, les heuristiques admissibles sont généralement également consistantes, surtout si elles sont dérivées de relaxations du problème. Par conséquent, il est généralement plus facile de commencer par considérer les heuristiques admissibles. Une fois que vous avez une heuristique admissible qui fonctionne bien, vous pouvez vérifier si elle est effectivement consistante. La seule façon de garantir la consistance est avec une preuve. Cependant, l'inconsistance peut souvent être détectée en vérifiant que pour chaque nœud que vous développez, ses nœuds successeurs sont égaux ou supérieurs en valeur de la fonction f. De plus, si UCS et A* retournent des chemins de longueurs différentes, votre heuristique est inconsistante.

Heuristique non-triviale: Il y a deux heuristiques triviales: celle qui retourne zéro partout et l'heuristique qui calcule le vrai coût. La première ne sauverais aucun temps, tandis que la dernière prendrais beaucoup trop de temps. Vous voulez une heuristique qui réduit le temps de calcul total, bien que pour ce travail, le correcteur automatique ne vérifie que le nombre de nœuds (en dehors de l'application d'une limite de temps raisonnable).

Notes: Votre heuristique doit être une heuristique cohérente, non-triviale et non-négative pour recevoir des points. Assurez-vous que votre heuristique renvoie 0 à chaque état de but et ne renvoie jamais de valeurs négatives. Vous serez noté en fonction du nombre de nœuds développés:

Nombre de noeuds développés	Note
plus de 2000	0/3
moins de 2000	1/3
moins de 1600	2/3
moins de 1200	3/3

1.7 Manger tous les points: heuristique [4 points]

Maintenant, nous allons résoudre un problème de recherche difficile: manger tous les points en aussi peu d'étapes que possible. Pour cela, nous aurons besoin d'une nouvelle définition de problème de recherche: FoodSearchProblem dans searchAgents.py (implémenté pour vous). Une solution est définie pour être un chemin qui recueille tous les points dans le monde Pacman. Pour le présent projet, les solutions ne tiennent pas compte des fantômes ou des pastilles de puissance; Les solutions

dépendent uniquement de l'emplacement des murs, de la nourriture habituelle et de Pacman. Si vous avez écrit correctement vos méthodes de recherche générales, A* avec une heuristique nulle (équivalente à une recherche de coût uniforme) devrait rapidement trouver une solution optimale pour testSearch sans changement de code de votre part (coût total de 7).

python pacman.py -l testSearch -p AStarFoodSearchAgent

Remarque: AStarFoodSearchAgent est un raccourci pour
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic.

Vous devriez trouver que l'exploration à coût uniforme commence à ralentir même pour le tinySearch apparemment simple. À titre de référence, notre implémentation prend 2,5 secondes pour trouver un chemin de longueur 27 après l'expansion de 5057 nœuds de recherche.

Remarque: Assurez-vous de compléter la question 4 avant de travailler sur la question 7, car la question 7 s'appuie sur votre réponse à la question 4.

Remplissez foodHeuristic dans searchAgents.py avec une heuristique consistante pour le FoodSearchProblem. Essayez votre agent sur le trickySearch:

python pacman.py -l trickySearch -p AStarFoodSearchAgent

Notre agent UCS trouve la solution optimale en environ 13 secondes, explorant plus de 16 000 nœuds.

Toute heuristique consistante non-négative recevra au moins 1 point. Assurez-vous que votre heuristique renvoie 0 à chaque état de but et ne renvoie jamais de valeur négative. En fonction du nombre de nœuds que votre heuristique développe, vous obtiendrez le nombre de points suivant:

Nombre de noeuds développés	Note
plus de 15000	1/4
moins de 15000	2/4
moins de 12000	3/4
moins de 9000	4/4
moins de 7000	5/4

1.8 Recherche sous-optimale [3 points]

Parfois, même avec A* et une bonne heuristique, trouver le chemin optimal à travers tous les points est difficile. Dans ce cas, nous aimerions trouver un chemin raisonnablement bon, rapidement. Dans cette section, vous allez écrire un agent qui mange toujours avidement le point le plus proche. ClosestDotSearchAgent est implémenté pour vous dans searchAgents.py, mais il manque une fonction importante qui trouve un chemin vers le point le plus proche.

Implémentez la fonction findPathToClosestDot dans searchAgents.py. Notre agent résout ce labyrinthe (sous-optimalement!) en moins d'une seconde avec un coût de chemin de 350:

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

Astuce: Le moyen le plus rapide pour compléter findPathToClosestDot est de terminer le AnyFoodSearchProblem (il manque le test d'objectif). Ensuite, résolvez ce problème avec une fonction de recherche appropriée. La solution devrait être très courte!

Votre ClosestDotSearchAgent ne trouvera pas toujours le chemin le plus court possible dans le labyrinthe. Assurez-vous de comprendre pourquoi et essayez de trouver un petit exemple où toujours aller au point le plus proche n'aboutit pas à trouver le chemin le plus court pour manger tous les points.