TP1 - Stratégie de recherche

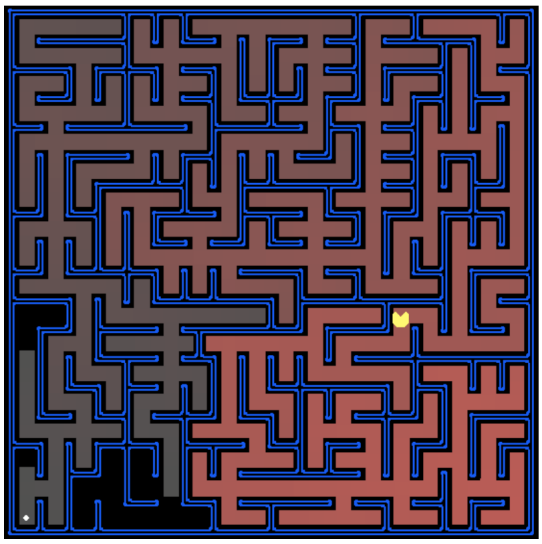
Remise le vendredi 1er octobre sur Moodle (avant minuit) pour tous les groupes.

Consignes

- Le devoir doit être fait par groupe de 2 au maximum. Il est fortement recommandé d'être 2.
- Vous devrez remettre les fichiers `search.py` et `searchAgents.py` à la racine d'un seul dossier compressé (matricule1_matricule2_TP1.zip).
- Indiquez vos noms et matricules en commentaires en haut des deux fichiers de code soumis.
- Il n'y a aucun rapport à remettre pour ce devoir.
- Toutes les consignes générales du cours (interdiction de plagiat, etc.) s'appliquent pour ce devoir.
- Il est permis (et encouragé) de discuter de vos pistes de solution avec les autres groupes. Par contre, il est formellement interdit de reprendre le code d'un autre groupe ou de copier un code déjà existant (StackOverflow ou autre). Tout cas de plagiat sera sanctionné de la note minimale pour le TP.

Enoncé

Oh non! Notre vieil ami Pacman s'est encore une fois retrouvé coincé dans un labyrinthe. Heureusement, il n'y a pas d'ennemis dans celui-ci. Pour gagner une partie, Pacman devra manger tous les succulents ronds jaunes se trouvant dans le labyrinthe. Étant très gourmand, il tient à le faire le plus rapidement possible. Votre objectif sera donc de trouver le chemin le plus court pour tout manger. Vous devrez donc user de vos nouvelles connaissances en stratégie de recherche générale afin de l'aider à tout manger le plus rapidement possible.



Ressources fournies

Vous trouverez tous les fichiers nécessaires dans le dossier TP1.zip. Vous pouvez ignorer la plupart des fichiers de codes, car ils contiennent la logique pour faire fonctionner la plateforme de Pacman.

Les fichiers que vous devrez modifier et soumettre complets sont les suivants :

- `search.py` : implémentation de vos algorithmes de recherche
- `searchAgents.py` : implémentation de vos agents

D'autres fichiers pourraient vous aider pour ce travail, mais ils ne sont ni à modifier ni à remettre :

- `pacman.py` : fichier principal qui permet de lancer le programme.
- `game.py` : implémentation de l'environnement de pacman. Il décrit plusieurs classes telles `AgentState`, `Agent`, `Direction` et `Grid`.
- `util.py` : contient des structures de données intéressantes pour implémenter vos algorithmes de recherche

Un script d'auto-évaluation est également fourni (similaire à celui du tutoriel d'introduction). Pour l'exécuter, il suffit de lancer la commande suivante :

```
python autograder.py
```

Le script montre un pointage pour chaque question séparément, n'hésitez donc pas à le lancer à chaque fois que vous compléterez une question pour vous assurer d'une progression correcte dans le devoir.

Critères d'évaluation

Les questions 1 à 7 seront évaluées de manière automatique avec le script qui vous est fourni. Vous ne devez donc en aucun cas modifier le nom des fonctions. Si le script ne compile pas avec vos fichiers, une note de zéro vous sera accordée. Votre note finale sera accordée selon l'exactitude de votre implémentation et non celle donnée par le script. Vous pourriez également perdre des points si votre code manque de lisibilité.

Les points sont distribués de la manière suivante :

Question	Points
Partie 1	
1	/3
2	/3
3	/3
4	/3
Partie 2	
5	/3
6	/3
Partie 3	
7	/5
Lisibilité du code	/2
Total	/25

Partie 1 : Un seul rond jaune

Pour la première partie du devoir, on utilisera une version simplifiée du problème initial. L'idée est de ne considérer que des labyrinthes qui n'ont qu'un seul rond jaune. Votre premier mandat est de trouver le plus court chemin entre la position initiale de Pacman et la position de son unique rond jaune. Pour résoudre ce problème, il vous est demandé d'implémenter quatre algorithmes de recherche (BFS, DFS, UCS, A*). Chaque implémentation fait office d'un code à rendre.

⚠Vos quatre algorithmes doivent mémoriser les états déjà vus (recherche en graphe et non une recherche en arbre)

Le problème a déjà été modélisé pour vous. Chacune des fonction que vous devrez implémenter prend en paramètre le problème en question, qui est de type `SearchProblem` (classe définie dans le fichier `search.py`). Voici les méthodes de cette classe que vous aurez à utiliser dans votre implémentation :

- `problem.getStartState()` retourne la position (*state*) initiale (*x, y*) de Pacman.
- `problem.isGoalState(state)` retourne un boolean indiquant si une position *(x, y)* correspond à celle du rond jaune. Si c'est le cas, alors l'état correspond à l'objectif à atteindre.
- `problem.getSuccessors(state)` retourne tous les voisins légaux d'un état. Cette fonction s'assure pour vous qu'il n'y a pas de murs ou d'obstacles. Elle ne retourne que les positions réellement disponibles. Elle retournera le tuple *(position, action, coût)* avec la position *(x, y)*, l'action menant à cette position *(North, South, East, West)* et un coût numérique (cette dernière information ne vous sera pas utile pour les questions 1 et 2).

La fonction `tinyMazeSearch(problem)`, du fichier `search.py` présente un exemple trivial du format de retour qui est attendu.

```
65  def tinyMazeSearch(problem):
66      """
67      Returns a sequence of moves that solves tinyMaze.  For any other maze, the
68      sequence of moves will be incorrect, so only use this for tinyMaze.
69      """
70      from game import Directions
71      s = Directions.SOUTH
72      w = Directions.WEST
73      return  [s, s, w, s, w, w, s, w]
74
```

On s'attend donc à ce que vous retourniez une liste ordonnée de mouvements de type `Direction` (fichier `game.py`) permettant de se rendre jusqu'au rond jaune. Pour tester que tout fonctionne tel que prévu, cette fonction triviale fournit la solution au plus petit labyrinthe. Commencez par la tester à l'aide de la commande suivante :

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

En cas de problèmes, n'hésitez pas à communiquer avec votre chargé de laboratoire.

⚠Ne changez la signature d'aucune fonction ni d'aucune classe. N'écrivez votre code code qu'aux endroits demandés.

Question 1 : Recherche en profondeur (DFS - depth first search)

Vous devez implémenter un algorithme de recherche en profondeur. Votre implémentation se fera dans la fonction `depthFirstSearch` du fichier `search.py`. Pour valider votre solution, assurez-vous que votre algorithme résout pratiquement instantanément les labyrinthes suivants :

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

⚠**Vous devez utiliser des structures de données de type `Queue` (FIFO), `PriorityQueue` ou `Stack` (LIFO) fournies dans le fichier `util.py`.**

Question 2 : Recherche en largeur (BFS - breadth first search)

Vous devez implémenter un algorithme de recherche en largeur. Votre implémentation se fera dans la fonction `breadthFirstSearch` du fichier `search.py`. Pour valider votre solution, vérifiez votre solution de la même manière qu'à la question précédente.

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

⚠**Vous devez utiliser des structures de données de type `Queue` (FIFO), `PriorityQueue` ou `Stack` (LIFO) fournies dans le fichier `util.py`.**

Question 3 : Recherche à coût uniforme (UCS - Uniform cost search)

Pour cette méthode, vous devez pour la première fois utiliser l'argument de coût qui vous est retourné par la fonction `problem.getSuccessors(state)`. Vous n'avez donc pas à calculer vous même le coût d'un état à l'autre. Utilisez celui qui vous est fourni par cette fonction. Votre implémentation doit se faire dans la fonction `uniformCostSearch` du fichier `search.py`. Une fois complété, vous serez capable d'observer un Pacman vainqueur dans les scénarios suivants :

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

⚠**Vous devez utiliser des structures de données de type `Queue` (FIFO), `PriorityQueue` ou `Stack` (LIFO) fournies dans le fichier `util.py`.**

Question 4 : Recherche A*

Vous devez ici implémenter une recherche A* dans la fonction `aStarSearch` du fichier `search.py`. Vous remarquerez que cette fonction prend un argument supplémentaire au problème : la fonction heuristique (paramètre `heuristic`). Il s'agit de la fonction qui permet d'estimer le coût d'un état jusqu'à l'état final. Cette fonction prend en paramètres l'état actuel (*state*) et le problème initial (*problem*). À titre d'exemple, observez la fonction `nullHeuristic` décrite en haut de la méthode `aStarSearch`.

Pour cette question, vous ne devez pas écrire une fonction heuristique vous-même. Concentrez vous sur la logique de l'algorithme et utilisez la fonction qui est passée en paramètre. Une fois l'implémentation complétée, testez la avec l'heuristique de Manhattan (déjà implémentée) via la commande suivante :

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

Vous devriez observer que l'algorithme A* explore légèrement moins d'états que l'algorithme UCS pour le labyrinthe *bigMaze* (549 contre 620, le nombre exact peut légèrement différer en cas d'égalité dans les priorités).

⚠**Vous devez utiliser des structures de données de type `Queue` (FIFO), `PriorityQueue` ou `Stack` (LIFO) fournies dans le fichier `util.py`.**

Partie 2 : Le problème des quatre coins

⚠**Les question 2 et 4 doivent absolument avoir été complétées avant de débuter la partie 2.**

La force d'une recherche A* s'observe mieux lorsqu'on a affaire à problèmes de plus grande envergure. Pour ce nouveau problème, nous considérerons des labyrinthes qui ont quatre ronds jaunes, spécifiquement positionnés aux quatre coins de la grille. Le nouvel objectif de Pacman sera ainsi de trouver le chemin le plus court pour atteindre chacun des quatre coins.

Question 5 : Modéliser le problème

Pour cette situation, on vous demande d'abord de modéliser l'environnement du problème et de définir adéquatement les états. Pour le problème précédent, il était suffisant de ne considérer que la position de pacman. Vous devez ici construire une représentation qui encode toutes les informations nécessaires pour détecter que les quatre coins ont bien été atteints. Concrètement, vous devez compléter la classe `CornersProblem` du fichier `searchAgents.py` [1]. Testez ensuite votre implémentation avec les commandes suivantes :

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

⚠**Il n'est aucunement nécessaire de modifier du code de la partie 1 pour que tout soit fonctionnel. Pensez plutôt à adapter le format que vous aurez choisi pour représenter l'état au code déjà écrit. Vous verrez ainsi que vos algorithmes de recherche pourront être réutilisés tels quels.**

⚠**Vous perdrez des points si vous encodez de l'information inutile, car cela pourrait ralentir drastiquement l'exécution de votre code. En particulier, n'utilisez pas la classe`GameState` comme un état de la recherche.**

Question 6 : Concevoir une heuristique

Le problème des quatre coins étant maintenant modélisé, vous pouvez maintenant utiliser les diverses méthodes de recherche pour le résoudre. Cependant, les heuristiques définies pour vous précédemment ne sont forcément pas les plus adaptées à ce nouveau problème. On vous demande ici de concevoir une nouvelle heuristique dans la fonction `cornersHeuristic` du fichier `searchAgents.py`. Votre heuristique doit avoir les caractéristiques suivantes pour obtenir des points (une note de zéro sera accordée à cette question pour tout manquement) :

- **non-triviale** : elle ne doit pas retourner systématiquement une valeur nulle ou constante.

1. Indice : La structure que vous aurez choisie doit permettre d'obtenir la position actuelle de Pacman et la position des coins (ont-ils déjà été mangés ?).

- **admissible** : elle ne doit jamais surestimer le coût pour se rendre à l'état final.
- **consistante** : elle ne doit pas surestimer le coût de transition d'un état à l'autre.

N'hésitez pas à vous référer au cours théorique pour plus de détails sur ces propriétés.. Le script d'auto-évaluation vous aidera à déceler un manquement à l'une de ces dernières. Testez votre implémentation avec la commande suivante :

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

Votre note pour cette question dépendra de qualité de votre heuristique. De manière générale, plus l'estimation se rapproche du coût réel, sans le dépasser, plus elle sera efficace pour économiser du temps de calcul. Ainsi, plus votre estimation permet d'éviter de visiter des états, meilleure sera votre note. Le barème utilisé est le suivant.

Nombre d'états visités	Note
plus de 2000	0/3
au plus 2000	1/3
au plus 1600	2/3
au plus 1200	3/3

Partie 3 : Manger tous les points!

⚠Assurez d'avoir complété la question 4 avant de débuter cette partie.

Dans cette dernière partie, nous augmentons une dernière fois la complexité du problème. Votre mandat est de trouver le plus court chemin pour manger tous les ronds jaunes se trouvant un peu partout dans le labyrinthe. Ce dernier problème a déjà été modélisé pour vous dans la classe `FoodSearchProblem` du fichier `searchAgents.py`. Si vos implémentations de la partie 1 sont correctes, l'algorithme A* avec une heuristique triviale (`nullHeuristic`) devrait permettre de trouver rapidement une solution optimale au labyrinthe de test suivant :

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

Question 7

L'heuristique triviale n'est pas le meilleur choix pour permettre d'améliorer l'efficacité de notre algorithme de recherche. On peut déjà observer que pour le plus petit labyrinthe, le temps d'exécution passe à environ 2 à 3 secondes, avec 5057 états visités, pour trouver une solution à notre nouveau problème.

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

Vous devez donc concevoir une heuristique de meilleure qualité pour ce problème. Écrivez votre implémentation dans la fonction `foodHeuristic` du fichier `searchAgents.py`. Testez votre implémentation à l'aide ce labyrinthe complexe :

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

Vous serez évalués de la même manière qu'à la question 6, en utilisant cette fois-ci le barème suivant :

Nombre d'états visités	Note
plus de 15000	1/5
au plus 15000	2/5
au plus 12000	3/5
au plus 9000	4/5
au plus 7000	5/5

⚠Vous aurez une note de 0 pour cette question si votre fonction ne respecte pas les critères de consistance et de non-trivialité.

Bon travail! N'hésitez pas à partager sur Slack les résultats (nombre de noeuds visités) de vos meilleurs heuristiques!