Artificial Intelligence
AIC_4301C
TP1
2021-2022

1. This project uses python 3.
2. You should execute your code in a linux environment by executing the command on a terminal.

1 Tests

1. Create a folder TP1_AIC_4301B_names-of-the-group-members
2. If you are using your own laptop:
   (a) In this folder download the file **requirements_AIC.txt** from Blackboard.
   (b) Execute pip install -r requirements_AIC.txt
3. Download the project **search_AIC.zip** from Blackboard, unzip it.
4. In the folder search_AIC execute python3 pacman_AIC.py
5. If you have the message if No module named 'tkinter' using your own laptop, execute sudo apt-get install python3-tk
6. Execute python3 pacman_AIC.py –layout testMaze –pacman GoWestAgent

   You should have in the terminal:

   ```
   Pacman emerges victorious! Score: 503
   Average Score: 503.0
   Scores: 503.0
   Win Rate: 1/1 (1.00)
   Record: Win
   ```
7. See the list of all options and their default values via:

   python3 pacman_AIC.py -h
8. The commands that appear in this project are in **commands_AIC.txt**, you can even run all these commands in order with:

   bash commands_AIC.txt

2 Recommendations

1. A search node must contain not only a state but also the information necessary to reconstruct the path which gets to that state.
2. All of your search functions need to return a list of actions that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).
3. Make sure to use the Stack, Queue and PriorityQueue data structures provided in util.py.

3 Python Files

In the folder **search_AIC** you will find the following python files:

1. search.py: Where all of your search algorithms will reside.
2. searchAgents.py: Where all of your search-based agents will reside.
3. pacman_AIC.py The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this project.
4. game.py The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
5. util.py Useful data structures for implementing search algorithms. We encourage you to look through util.py for some data structures that may be useful in your implementations.
6. You can ignore all other .py files.

4 Depth-first search (DFS)

The goal is finding a Fixed Food Dot using a Depth-first search.

1. Implement the depth-first search (DFS) algorithm in the **depthFirstSearch** function in **search.py**. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states. Your search algorithm needs to return a list of actions that reaches the goal.
2. Test your code:

```
python3 pacman_AIC.py -l tinyMaze -p SearchAgent
python3 pacman_AIC.py -l mediumMaze -p SearchAgent
python3 pacman_AIC.py -l bigMaze -z .5 -p SearchAgent
```

5 Breadth-first Search (BFS)

1. Implement the breadth-first search (BFS) algorithm in the **breadthFirstSearch** function in **search.py**. Write a graph search algorithm that avoids expanding any already visited states. Your search algorithm needs to return a list of actions that reaches the goal.
2. Test your code:

```
python3 pacman_AIC.py -l mediumMaze -p SearchAgent -a fn=bfs
python3 pacman_AIC.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

6 Uniform Cost Serach (UCS)

1. Implement the uniform-cost graph search algorithm in the **uniformCostSearch** function in **search.py**.
2. Test your code:

```
python3 pacman_AIC.py -l mediumMaze -p SearchAgent -a fn=ucs
python3 pacman_AIC.py -l mediumDottedMaze -p StayEastSearchAgent
python3 pacman_AIC.py -l mediumScaryMaze -p StayWestSearchAgent
```

7 A^* Serach

1. Implement A* graph search in the empty function **aStarSearch** in **search.py**. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The **nullHeuristic** heuristic function in search.py is a trivial example.

   You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as **manhattanHeuristic** in **searchAgents.py**).

2. Test your code:

   python3 pacman_AIC.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=nullHeuristic

   python3 pacman_AIC.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

8 Finding All the Corners

In corner mazes, there are four dots, one in each corner. Our **new search problem** is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like tinyCorners, the shortest path does not always go to the closest food first.

1. Implement the **CornersProblem** search problem in **searchAgents.py**. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Functions to implement: def __init__(self, startingGameState), getStartState(self), isGoalState(self, state), getSuccessors(self, state)

2. Test your code:

   python3 pacman_AIC.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

   python3 pacman_AIC.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

9 Corners Problem: Heuristic

The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost.

1. Implement a non-trivial, consistent heuristic for the **CornersProblem** in **cornersHeuristic**. If UCS and A* ever return paths of different lengths, your heuristic is inconsistent.

2. Test your code:

   python3 pacman_AIC.py -l mediumCorners -p AStarCornersAgent -z 0.5

   AStarCornersAgent is a shortcut for:

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

10 Eating All The Dots

Eating all the Pacman food in as few steps as possible need a new search problem definition which formalizes the food-clearing problem: **FoodSearchProblem** in **searchAgents.py** (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) If you have written your general search methods correctly, A* with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to **testSearch** with no code change on your part (total cost of 7):

```
python3 pacman_AIC.py -l testSearch -p AStarFoodSearchAgent
AStarFoodSearchAgent is a shortcut for:
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

1. Fill in foodHeuristic in searchAgents.py with a consistent heuristic for the FoodSearchProblem.
2. Test your code:
   ```
   python3 pacman_AIC.py -l trickySearch -p AStarFoodSearchAgent
   ```

11 Suboptimal Search

Write an agent that always greedily eats the closest dot. **ClosestDotSearchAgent** is implemented for you in **searchAgents.py**, but it's missing a key function that finds a path to the closest dot.

1. Implement the function **findPathToClosestDot** in **searchAgents.py**.
2. Test your code:
   ```
   python3 pacman_AIC.py -l bigSearch -p ClosestDotSearchAgent -z .5
   ```