SWE3011 Homework 1: Search*

JinYeong Bak (jy.bak@skku.edu)

Deadline: Oct 9 2022, 23:59†

1 Introduction

Please read the homework description carefully and make sure that your program meets all the requirements stated. The homework is an individual task. You can discuss the problem with your friends but you should not program together. You will get an F on the entire course if your homework includes any plagiarism.

2 Goal

In this homework, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios [1].

This project includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

3 Description

All descriptions in this section are based on the page [1]. You can read the page to understand basic tasks (questions 1, 2, 3, 4) and extra tasks (questions 5, 6). But we add some more requirements for each task. Please read this description carefully.

3.1 Welcome to Pacman

You are able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

The simplest agent in searchAgents.py is called the GoWestAgent, which always goes West (a trivial reflex agent). This agent can occasionally win:

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

But, things get ugly for this agent when turning is required:

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

*Source: `http://ai.berkeley.edu`

†You are encouraged to ask questions about this homework on icampus Q&A board. Please do not write your code on the board.

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.

Soon, your agent will solve not only tinyMaze, but any maze you want.

Note that pacman.py supports a number of options that can each be expressed in a long way (e.g., –layout) or a short way (e.g., -l). You can see the list of all options and their default values via:

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

Also, all of the commands that appear in this project also appear in commands.txt, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with bash commands.txt.

3.2 Question 1: Finding a Fixed Food Dot using Depth First Search

In searchAgents.py, you'll find a fully implemented SearchAgent, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented – that's your job.

First, test that the SearchAgent is working correctly by running:

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

The command above tells the SearchAgent to use tinyMazeSearch as its search algorithm, which is implemented in search.py. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note: 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).

Important note: Make sure to use the Stack, Queue and PriorityQueue data structures provided to you in util.py! These data structure implementations have particular properties which are required for compatibility with the autograder.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the frontier is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need not be of this form to receive full credit).

Hint: The argument 'problem' is an instance of SearchProblem class.

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.

Requirement: Please write the code for this question in depthFirstSearch function. If you want to define new functions, please define them in the depthFirstSearch function. We will read your code in the depthFirstSearch function, not other spaces. If not, you will get a penalty.

Your code should quickly find a solution for:

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

python pacman.py -l mediumMaze -p SearchAgent

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

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Hint: If you use a Stack as your data structure, the solution found by your DFS algorithm for mediumMaze should have a length of 130 (provided you push children onto the frontier in the order provided by expand; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

3.3 Question 2: Breadth First Search

Implement the breadth-first search (BFS) algorithm in the breadthFirstSearch function in search.py. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

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

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

Does BFS find a least cost solution? If not, check your implementation.

Requirement: Please write the code for this question in breadthFirstSearch function. If you want to define new functions, please define them in the breadthFirstSearch function. We will read your code in the breadthFirstSearch function, not other spaces. If not, you will get a penalty.

Hint: If Pacman moves too slowly for you, try the option –frameTime 0.

Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

```
python eightpuzzle.py
```

3.4 Question 3: A* search

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).

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

Requirement: Please write the code for this question in aStarSearch function. We will read your code in the aStarSearch function, not other spaces. If not, you will get a penalty.

You should see that A* finds the optimal solution slightly faster than BFS (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on openMaze for the various search strategies?

3.5 Question 4: Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

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! Hint: the shortest path through tinyCorners takes 28 steps.

Note: Make sure to complete Question 2 before working on Question 4, because Question 4 builds upon your answer for Question 2.

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. Now, your search agent should solve:

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

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

To receive full credit, you need to define an abstract state representation that does not encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman GameState as a search state. Your code will be very, very slow if you do (and also wrong).

Hint 1: The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

Number of nodes expanded	Grade
more than 2000	0/3
at most 2000	1/3
at most 1600	2/3
at most 1200	3/3

Hint 2: When coding up expand, make sure to add each child node to your children list with cost getActionCost and next state getNextState. Note you will also need to code up the getNextState function.

Hint 3:You should store states of the tuple format ((x,y), ____). You will need to decide what information to store in the blank.

Our implementation of breadthFirstSearch expands just under 2000 search nodes on mediumCorners. However, heuristics (used with A* search) can reduce the amount of searching required.

3.6 Question 5: Corners Problem: Heuristic

This task is an extra task. Please solve this question if you want to take additional points.

Note: Make sure to complete Question 3 before working on Question 5, because Question 5 builds upon your answer for Question 3.

Implement a non-trivial, consistent heuristic for the CornersProblem in cornersHeuristic.

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

Note: AStarCornersAgent is a shortcut for

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

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be admissible, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be consistent, it must additionally hold that if an action has cost c, then taking that action can only cause a drop in heuristic of at most c.

Remember that admissibility isn't enough to guarantee correctness in graph search – you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. Consistency can be verified for a heuristic by checking that for each node you expand, its child nodes are equal or lower in in f-value. If this condition is violated for any node, then your heuristic is inconsistent. Moreover, if UCS (A* with the 0 heuristic) and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Grading: Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded:

Remember: If your heuristic is inconsistent, you will receive no credit, so be careful!

3.7 Question 6: Eating All The Dots

This task is an extra task. Please solve this question if you want to take additional points.

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll 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

Number of nodes expanded	Grade
more than 15000	1
at most 15000	2
at most 12000	3
at most 9000	4 (full credit; medium)
at most 7000	5 (optional extra credit; hard)

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).

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

Note: AStarFoodSearchAgent is a shortcut for

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

You should find that UCS starts to slow down even for the seemingly simple tinySearch. As a reference, our implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 search nodes.

Note: Make sure to complete Question 3 before working on Question 6, because Question 6 builds upon your answer for Question 3.

Fill in foodHeuristic in searchAgents.py with a consistent heuristic for the FoodSearchProblem. Try your agent on the trickySearch board:

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

Our UCS agent finds the optimal solution in about 13 seconds, exploring over 16,000 nodes.

Any non-trivial non-negative consistent heuristic will receive 1 point. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll get additional points:

Remember: If your heuristic is inconsistent, you will receive no credit, so be careful! Can you solve mediumSearch in a short time? If so, we're either very, very impressed, or your heuristic is inconsistent.

4 Submission and Evaluation

You need to submit the followings:

- search.py
- searchAgents.py

You MUST compress the above folders with your own source code files into 'HW1_yourid.zip' (e.g.), HW1_2022123456.zip and submit the compressed file to icampus [2].

If you do not follow the above submission policy, you will get penalty.

Your code will be autograded for technical correctness including the original testcases in autograder.py and our own testcases. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation - not the autograder's judgements - will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.

We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only. If you do, we will give F grade.

You are not alone! If you find yourself stuck on something, contact the TA for help. Please use icampus Q&A board. If you are in a struggle, other students are also the same. You can ask common questions, and you can find the solutions by searching. But please follow the communication rules in the first week's pdf file. If not, we might ignore your questions.

References

[1] Dawn Song Stuart Russell. Cs 188. https://inst.eecs.berkeley.edu/~cs188/sp21/, 2022.

[2] SKKU i-Campus. https://icampus.skku.edu/, 2022.