CS440/ECE448 Spring 2020

Assignment 1: Search

Due date: Wednesday Feb 5th, 11:59pm

Credits: Berkeley CS188 Pacman projects
Micky Abir, Chris Benson, Krishna Kothapalli, and JD Lee (Fall 2018)
Updated By: Rahul Kunji & Jason Nie (Spring 2019)
Updated By: Yuqian Zhou (yuqian2) (Spring 2020)

In this assignment, you will build general-purpose search algorithms and apply them to solving puzzles. Specifically, we implement path finding algorithm including BFS and A* for Pacman Game.

The whole project will have three parts. In Part 1, you will be in charge of a "Pacman"-like agent that needs to find a path through maze to eat a dot or "food pellet." In Part 2, try to find a path going through all the four corners of the maze. In Part 3, you will need to find a single path that goes through all the dots in the maze.

Programming language

This MP will be written in Python. If you've never used Python before, you should start getting used to it. A good place to start is the Python Tutorial (also available in hardcopy form). You should install **version 3.6 or 3.7** on your computer, as well as the pygame graphics package.

Your code may import extra modules, but only ones that are part of the standard python library . **Unless otherwise specified in the instructions for a specific MP, the only external library available during our grading process will be pygame. For example: in mp1, numpy is not allowed**

Contents

Part 1: Finding a single dot

Consider the problem of finding the shortest path from a given start state while eating one or more dots or "food pellets." The image at the top of this page illustrates the simple scenario of a single dot, which in this case can be viewed as the unique goal state. The maze layout will be given to you in a simple text format, where '%' stands for walls, 'P' for the starting position, and '.' for the dot(s) (see sample maze file). All step costs are equal to one.

For further usage, you are suggested to directly implement the state representation, transition model, and goal test needed for **solving the problem in the general case of multiple dots**. For the state representation, besides your current position in the maze, is there anything else you need to keep track of? For the goal test, keep in mind that in the case of multiple dots, the Pacman does not necessarily have a unique ending position. Next, implement the following search strategies, as covered in class and/or the textbook:

- Breadth-first search (BFS)
- A* search

To implement A* for finding single dot, you can use the Manhattan distance from the current position to the goal as the heuristic function.

To check for correctness, you can run each of the two search strategies on the following example inputs:

- Tiny maze
- Medium maze
- Big maze

You may expect the path lengths retruned by BFS and A* are enqual, and A* achieves a fewer state exploration. The provided code will also generate a pretty picture of your solution for visualization.

Part 2: Finding all corners

Now we consider a little harder search problem to experience the real power of A*. In this part, we define one type of objectives called corner mazes. In corner mazes, there are only four dots, one in each corner. Our new search problem is to find the shortest path through the maze, starting from P and touching all four corners (the maze actually has food there). Note that for some mazes like tinyCorners, the shortest path does not necessarily go to the closest food first! It means you may not start from the nearest corner and apply part 1 recursively.

As instructed in Part 1, your state representation, goal test, and transition model should already be adapted to deal with multiple dots case. The next challenge is to solve the following inputs using A* search using an **admissible and consistent heuristic** designed by you:

- Tiny corner
- Medium corner
- Big corner

You should be able to handle the tiny search using uninformed BFS. In fact, it is a good idea to try that first for debugging purposes, to make sure your representation works with multiple dots. However, to successfully handle all the inputs, it is crucial to use A* and come up with a good heuristic. For full credit, your heuristic should be **admissible and consistent** and should permit you to find the solution for the medium search 1) with much fewer explored states than uninformed BFS and 2) in a reasonable amount of time. Make sure your algorithm will not exceed 2 seconds for the given examples.

[*Notes]: To be admissible, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal and non-negative (like Manhatten distance in Part 1). 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. Once you brainstormed 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. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in f-value. Moreover, if A* ever returns paths of different lengths, your heuristic is inconsistent.

Part 3: Finding multiple dots

Now consider a more general and harder problem of finding the shortest path through a maze while hitting multiple dots. The Pacman is initially at P, the goal is achieved whenever the Pacman manages to eat all the dots.

As instructed in Part 1, your state representation, goal test, and transition model should already be adapted to deal with this scenario. The next challenge is to solve the following inputs using A* search using an admissible heuristic designed by you:

- Tiny search
- Small search
- Medium search

You can still debug your method with tinySearch or tinyCorner using BFS, or the heuristic defined in part 2. However, to successfully handle all the inputs, it is crucial to use A* and come up with a better heuristic with better efficiency. Once

again, for full credit, your heuristic should be **admissible** and should permit you to find the solution for the medium search 1) with much fewer explored states than uninformed BFS and 2) in a reasonable amount of time. If you have some other clever way to approach the multiple-dot problem, implement that as extra credit.

Hints for Part 3

In the past almost all working solutions to this problem have used a heuristic based on the minimum spanning tree. The minimum spanning tree of a set of points can be computed easily via Kruskal's algorithm or Prim's algorithm. If T is the total length of the edges in the minimum spanning tree, then the shortest path connecting all the points must have length between T and 2T.

Now, suppose you are in the middle of a search. You're at some location (x,y) with a set of S dots still to reach. Your heuristic function h might be the sum of the distance from (x,y) to the nearest dot, plus the MST length for the dots in S. To compute the MST for a set of dots, you'll need the distance between each pair of dots. The Manhattan distances (or real shortest-path lengths precomputed by A* for single dot) will work here. You may also be able to find a better method.

During search, you'll have many states with the same set of objectives S. So, once you compute the MST length for a set of dots S, you'll probably need to use this number again. Make a table of known MST values to avoid re-doing the MST computation.

Extra Credit

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. Write a suboptimal search algorithm that will do a good job on this big maze. Note that applying part 3 code to this maze will be very slow. Your algorithm could either be A* with a non-admissible heuristic, or something different altogether. Note that the extra credit will be capped to 10% of what the assignment is worth.

Specifically, your extra credit function must solve this maze in less than 6 minutes (when we test it on gradescope). Assuming that it finishes in that amount of time, grading will be based only on the length of your returned path.

Provided Code Skeleton

We have provided the code skeleton zip file including all the code and example mazes to get you started, which means you will only have to write the search functions. You should only modify search.py. **Use the provided API functions (e.g. getNeighbors) and do not modify code in files other than search.py.** Otherwise the autograder may be unable to run your code and/or may decide that your outputs are incorrect.

maze.py

- getStart() :- Returns a tuple of the starting position, (row, col)
- getObjectives() :- Returns a list of tuples that correspond to the dot positions, [(row1, col1), (row2, col2)]
- isValidMove(row, col) :- Returns the boolean **True** if the (row, col) position is valid. Returns **False** otherwise.
- getNeighbors(row, col) :- Given a position, returns the list of tuples that correspond to valid neighbor positions. This will return at most 4 neighbors, but may return less.

search.py

There are 4 methods to implement in this file, namely **bfs(maze), astar(maze), astar_corner(maze),** and **astar_multi(maze)**. The method **astar_corner** is for part 2, **astar_multi** is for part 3, and the other methods are for part 1. There is also one optional method **extra(maze)** which you will use if you decide to do the extra credit. Each of these functions takes in a maze instance, and should return the path taken (as a list of tuples). The path should include both the starting state and the ending state. The maze instance provided will already be instantiated, and the above methods will be accessible.

To understand how to run the MP, read the provided **README.md** or run **python3 mp1.py -h** into your terminal. The following command will display a maze and let you create a path manually using the arrow keys.

python3 mp1.py --human maze.txt

The following command will run your astar search method on the maze.

python3 mp1.py --method astar maze.txt

You can also save your output picture as a file in tga format. If your favorite document formatter doesn't handle tga, tools such as gimp can convert it to other formats (e.g. jpg).

Tips & Notes

- Check that you are using python 3. Running our code (especially the display code) under python 2 may cause mysterious errors.
- You can (and should) create additional test mazes to make sure your code is working properly and/or help you debug problems. The final evaluation mazes will be different from the shown examples.
- In your implementation, make sure you get all the bookkeeping right. This includes handling of repeated states (in particular, what happens when you find a better path to a state already on the frontier) and saving the optimal solution path.
- Pay attention to tie-breaking. If you have multiple nodes on the frontier with the same minimum value of the evaluation function, the speed of your search and the quality of the solution may depend on which one you select for expansion.
- Implement all strategies using a similar approach and coding style. You must store the frontier in an explicit queue or priority queue (depending on the search algorithm).
- You will be graded primarily on the correctness of your solution. The evaluation on executation time will include all the process of precomputing and path searching. Your code must run in a generally reasonable amount of time, but you do not need to do significant optimization. For example, we don't care whether your priority queue or repeated state detection uses brute-force search, as long as you end up expanding exactly the correct number of nodes (except for small differences caused by differences among tie-breaking strategies) and find the optimal solution. So, feel free to use "dumb" data structures as long as it makes your life easier and still enables you to find the solutions to all the inputs in a reasonable amount of time.
- After fininsing all the parts, you may find the final implmentation can be general enough for all the parts in the assignment, but we still expect to have fewer execuation time for simpler tasks. (i.e. no unnecessary pre-computing). Note that for part 2, all the solutions should not exceed 2 seconds.

Plagiarism

- It's ok to copy small amounts of utility code from 3rd party sources, as long as the source is acknowledged.
- It's not ok to consult or copy from full implementations of the algorithm in question, e.g. old code from similar MPs (at UIUC or elsewhere).
- We will utilize a code similarity detetion system. Please do not copy codes from your classmates. Thanks!

Deliverables

This MP will be submitted via gradescope.

Please upload only **search.py** to gradescope.

After submission, gradescope will run preliminary tests to determine whether or not your submission appears valid. These preliminary tests are worth 0 points, and are for your information only. Passing all of these preliminary tests does *not* guarantee that your implementations are correct.