Credits: Berkeley CS188 Pacman projects

Foundations of Artificial Intelligence, Spring 2025

Homework 1: Search

Due date: Tuesday March 18, 11:59pm

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 four parts. In Part 1, you will be in charge of a "Pacman"-like agent that needs to find a path through the 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. In Part 4, you will need optimize the algorithm in part 3 and make it goes through this big maze in a reasonable amount of time.

This homework will be written in Python 3. Please use python=3.13.2, pygame=2.6.1, and numpy=2.2.3. Your code may import extra modules, but only ones that are part of the standard python library. Other than those, the only external libraries available during our grading process will be pygame and numpy.

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.

So that your code will also work for the later parts of the assignment, we strongly suggest that your implementation for Part 1 use 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 equal, and but A* explores fewer states. The provided code will also generate a pretty picture of your solution for visualization.

Extra note on BFS: you must explore neighbors **in the exact order** returned from getNeighbors(), do not re-order the output of getNeighbors().

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 many fewer explored states than uninformed BFS and 2) in a reasonable amount of time. Make sure your algorithm will not exceed 1 second for the given examples.

Note: To be admissible, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal and non-negative (like Manhattan 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 have 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 many 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 Part 4.

Part 4: Fast heuristic Search for multiple dots

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. Your function must solve this maze within a reasonable amount of time.

Provided Code Skeleton

We have provided the code skeleton zip file tar 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.

- isValidPath(path) :- Run some sanity checks for given path. Returns str **"Valid"** if the path is valid. Returns all sorts of errors 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 5 methods to implement in this file, namely **bfs(maze), astar(maze), astar_corner(maze),**, **astar_multi(maze)**, and **fast(maze)**. The method **astar_corner** is for part 2, **astar_multi** is for part 3, **fast** is for part 4, and the other methods are for part 1. 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 homework, read the provided **README.md** or run **python3 hw1.py -h** into your terminal. The following command will display a maze and let you create a path manually using the arrow keys.

python3 hw1.py --human [maze file]

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

python3 hw1.py --method astar [maze file]

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).

Submission

This homework will be submitted via NTU COOL.

Please upload only **search.py** to NTU COOL.
All your code must be in this file; the autograder will ignore separate files of utility code.