Projet 1 : PacMan

March 15, 2023

In this project, 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.

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. You can download all the code and supporting files on moodle.

Files you'll edit:

- search.py : Where all of your search algorithms will reside.
- searchAgents.py: Where all of your search-based agents will reside.

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:

> python3 pacman.py -h

You can also find all the commands you need to test your code while solving the different questions in the file commands.txt.

Make sure to work on the different questions in the given order, because some of them build up on previous ones. Furthermore, you will notice that it is crutial to focus on the depth first search implementation, as the following three questions will be resolved in a similar manner, so it's normal to take more time for the first one.

Note that the course staff is always available for any questions you might have. Don't stay stuck for too long : If you cannot resolve a question, ask questions for more specific guidance!

Question 1 (3 pts) : 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.

To test your search agent, you can start by this simple command :

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

The option *tinymaze* picks the maze that the agent will search through, and *fn=tinyMazeSearch* will tell the code which search algorithm (implemented in search.py) to use to naviguate the maze.

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.

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).

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.

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 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?

Question 2 (3 pts) : 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.

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

> python eightpuzzle.py

Question 3 (3 pts) : Varying the Cost Function

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider mediumDottedMaze and mediumScaryMaze.

By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

Implement the uniform-cost graph search algorithm in the uniformCostSearch function in search.py. We encourage you to look through util.py for some data structures that may be useful in your implementation.

You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

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

Question 4 (3 pts) : 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) by executing:

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

You should see that A* finds the optimal solution slightly faster than uniform cost search. What happens on *openMaze* for the various search strategies?

Question 5 (3 pts) : 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!

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).

An instance of the CornersProblem class represents an entire search problem, not a particular state. Particular states are returned by the functions you write, and your functions return a data structure of your choosing (e.g. tuple, set, etc.) that represents a state.

Furthermore, while a program is running, remember that many states simultaneously exist, all on the queue of the search algorithm, and they should be independent of each other. In other words, you should not have only one state for the entire CornersProblem object; your class should be able to generate many different states to provide to the search algorithm.

Question 6 (3 pts) : Corners Problem: Heuristic

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 Copy

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. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher

in in f-value. Moreover, if UCS 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.

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

Question 7 (3 pts) : Eating All The Dots

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 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.

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

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.

Question 8 (3 pts) : Suboptimal Search

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. In this section, you'll 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.

Implement the function findPathToClosestDot in searchAgents.py.

Test it with :

> python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

Your ClosestDotSearchAgent won't always find the shortest possible path through the maze. Make sure you understand why and try to come up with a small example where repeatedly going to the closest dot does not result in finding the shortest path for eating all the dots.

1 Submission

In order to submit your project upload the Python files you edited.