Homework for Search

2024 Fall

Overview

Due November 4, 23:59, China Standard Time

This homework includes written and programming problems related to Lecture 5.

- For **writing component**, please write your answers in a `pdf` file. You can use LaTeX, convert from Word, from notebook apps on tablets, scan your handwritten documents, and etc. Your derivation and proof should be completed in the `pdf` file. Besides, your section titles and subsection titles should be the number of question. i.e., # Q1 ## Q1.1.
- For **programming component**, please submit your modified `search` project folder. Note that your analysis and some requested results should be included in the report `pdf` file.

Please organize your submitted files as follows:

```
report_${student_id}.pdf
search
    |--...
```

Finally, please zip the `report_${student_id}.pdf` file and `search` directory (rather than their parent directory) into `${student_id}.zip`. You should submit your `${student_id}.zip` to Web Learner before **November 4, 23:59, China Standard Time**.

1 Writing Component (40 pts)

1.1 Search with Heuristics (25 pts)

Imagine an $n \times n$ grid where there is a car in each of cells $(1, 1)$ to $(n, 1)$–the bottom row. Every time step, each car can move up, down, left, right, or stay put. If a car does not move, at most one other car can hop over it. At most one car can be in a cell at any given time. Cars move simultaneously (so a car can move into the place of another car if the other car moves out), and two cars are not allowed to move into the same cell. Moving a car in any direction (including hopping over another car) incurs a cost of 1, whereas staying put has no cost. The total cost at any time step is the sum of the cost for each car. The goal is to move all the cars to the top row but in reverse order so that the car starting in cell $(i, 1)$ ends up in cell $(n-i+1, n)$.

(a) What is the size of the state space in big-$\mathcal{O}$?

(b) Suppose there was only one car starting at $(i, 1)$, and the problem was to only move that car to its goal location. Consider the following heuristic for moving car i from cell (x_i, y_i) to its goal at $(n-i+1, n)$: $h_i(x_i, y_i) = |(n-i+1) - x_i| + |n - y_i|$. Is it an admissible heuristic? Would it be admissible when the other $n-1$ cars are also at various positions on the grid?

(c) Which of the following are admissible heuristics for the original problem and why? (Note: h_i is the heuristic in the previous part.)

(i) $\sum_i h_i$ (ii) $\min(h_1, \cdots, h_n)$ (iii) $\max(h_1, \cdots, h_n)$

(iv)$n \min(h_1, \cdots, h_n)$ (v) $n \max(h_1, \cdots, h_n)$

1.2 Adjusted Heuristics (15 pts)

The consistent (or monotone) condition on the heuristic function requires that $h(n_i) \leq h(n_j) + c(n_i, n_j)$ for all node-successor pairs (n_i, n_j), where (n_i, n_j) is the cost on the arc from n_i to n_j. It has been suggested that when the consistent condition is not satisfied, we can adjust h during the search to satisfy the condition. The idea is that whenever a node n_i is expanded, with successor node n_j, we can increase $h(n_j)$ by whatever amount is needed to satisfy the consistent condition. Construct an example to show that even with this scheme, when a node is expanded, we have not necessarily found the least costly path to it.

2 Programming Component: Search in Pacman (60 pts)

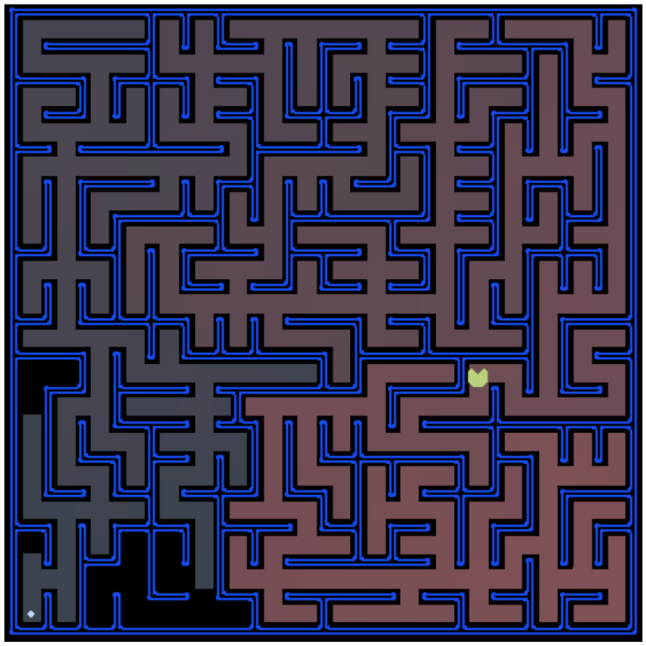

Figure 1: Pacman environments.

2.1 Introduction

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

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

```
python autograder.py
```

You can find all the code and supporting files in `search` folder. 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.

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.

Files you might want to look at:

`pacman.py` The main file that runs Pacman games. This file describes a Pacman GameState type you use in this project.

`game.py` The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.

`util.py` Useful data structures for implementing search algorithms.

Supporting files you can ignore:

`graphicsDisplay.py` Graphics for Pacman

`graphicsUtils.py` Support for Pacman graphics

`textDisplay.py` ASCII graphics for Pacman

`ghostAgents.py` Agents to control ghosts

`keyboardAgents.py` Keyboard interfaces to control Pacman

`layout.py` Code for reading layout files and storing their contents Project autograder

`autograder.py` Parses autograder test and solution files

`testParser.py` General autograding test classes

`testClasses.py` Directory containing the test cases for each question

`test_cases/ searchTestClasses.py` Project 1 specific autograding test classes

Files to Edit: You will fill in portions of search.py and searchAgents.py during the assignment. Please do not change the other files in this distribution.

Evaluation: Your code will be autograded for technical correctness. 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 judgments – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

2.2 Welcome to Pacman

Changing to the directory of `search`, you should be 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
```

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`.

2.3 Question 1 (2 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.

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! The pseudocode for the search algorithms you'll write can be found in the lecture slides. Remember that a search node must contain a state and the information necessary to reconstruct the path (plan) that 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 fringe is managed. So, concentrate on getting DFS right; 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).

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
```

The Pacman board will show an overlay of the states explored and the order in which they were explored (brighter red means earlier exploration).

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q1
```

The screenshot of the output should be included in the report `pdf` file.

2.4 Question 2 (2 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 the 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. 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
```

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q2
```

The screenshot of the output should be included in the report `pdf` file.

2.5 Question 3 (2 pts): Varying the Cost Function

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

We can encourage Pacman to find different paths by changing the cost function. 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
```

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q3
```

The screenshot of output should be included in the report `pdf` file.

2.6 Question 4 (4 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`).

```
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 (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?

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q4
```

The screenshot of the output should be included in the report `pdf` file.

2.7 Question 5 (10 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 finding the shortest path through the maze that touches all four corners (whether the maze 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 5, because Question 5 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: 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.

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.

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q5
```

The screenshot of the output should be included in the report `pdf` file.

2.8 Question 6 (10 pts): Corners Problem: Heuristic

Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.

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 the 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 also hold that if an action has cost c, then taking that action can only cause a drop in the heuristic of at most c.

Remember that admissibility isn't enough to guarantee correctness in graph search – you need a 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 also check whether it is consistent. The only way to guarantee consistency is with 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, your heuristic is inconsistent if UCS and A* ever return paths of different lengths. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS), and the heuristic computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic that 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:

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

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

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q6
```

The screenshot of the output should be included in the report `pdf` file.

2.9 Question 7 (15 points): 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 that formalizes the food-clearing problem: `FoodSearchProblem` in `searchAgents.py` (implemented for you). A solution is defined as a path collecting 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!) 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`. 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 4 before working on Question 7, because Question 7 builds upon your answer for Question 4.

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:

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

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.

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q7
```

The screenshot of the output should be included in the report `pdf` file.

2.10 Question 8 (15 points): Suboptimal Search

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. We'd still like to find a reasonably good path quickly in these cases. 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`. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:

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

Hint: The quickest way to complete `findPathToClosestDot` is to fill in the `AnyFoodSearchProblem`, which is missing its goal test. Then, solve that problem with an appropriate search function. The solution should be very short!

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

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q8
```

The screenshot of the output should be included in the report `pdf` file.

Grading

This assignment counts for a final grade of 10 points. Following is the conversion of your homework score (implied by points after each exercise) and final grade:

$$\text{Final Grade} = \left\lceil \frac{\min(\text{This homework score}, 100)}{100} \times 10 \right\rceil$$

We will actively be checking for plagiarism. If plagiarism is found, your final grade will be **F**.

Acknowledgment

This homework references UC Berkeley and CMU courses.