Homework 1

9/23/2023

50 Possible Points

Attempt 1

In Progress
NEXT UP: Submit Assignment

Add Comment

Unlimited Attempts Allowed
9/23/2023

Details

Homework 1

Submission instructions

- Due date and time: September 23 (Saturday), 11:59 pm ET
  - Remember, no late submissions are accepted, please start early! (Especially with the first midterm just before the due date!)
- Expected time commitment: 4-8 hrs
- Carmen submission: Submit one .zip file named HW1_programming_name_number.zip (e.g., HW1_programming_barker_348.zip), which contains 1) a .py file with the search algorithms you implemented: `search.py`; 2) a `collaboration.txt` which lists with whom you have discussed the homework (see more details below).
- Collaboration: You may discuss with your classmates. However, you need to write your own solutions and submit them separately. In your submission, you need to list with whom you have discussed the homework in a .txt file `collaboration.txt`. Please list each classmate's name.number (e.g., barker.348) as a row in the .txt file. That is, if you discussed with two classmates, your .txt file will have two rows. If you did not discuss with your classmates, just write "no discussion" in `collaboration.txt`. Please consult the syllabus for what is and is not an acceptable collaboration.

Introduction

In this homework, you are to implement four search algorithms --- DFS, BFS, UCS, and A* --- in `search.py` so that a pacman planning agent can complete the search problem. You are to implement the "graph" search rather than the "tree" search version.

Download `hw1_template.zip` from Carmen and unpack it. This code, and the idea for the assignment, comes from UC Berkeley (https://inst.eecs.berkeley.edu//~cs188/pacman/home.html) .

- Open up the Windows Command Line or Mac Terminal or Linux Terminal.
- Change your directory to the folder with the pacman code. You should see a file called `commands.txt` and four folders: `py`, `layouts`, `test_cases` and `images`.
- Run some of these commands (as listed in `commands.txt`) to make sure your setup works. Below are some examples:

```
python3 py/pacman.py
```

```
python3 py/pacman.py --layout tinyMaze --pacman GoWestAgent
```

- Make sure you can execute the above. You will not be able to complete the assignment if your environment does not allow you to run the above commands.

Possible errors and notes

If you're unable to run `pacman.py` or get its graphics to work, or if you're getting an error about "tkinter" or "_tkinter", that's okay. As long as you can run autograder.py, you should be able to complete the assignment fully.

- Note, you can add the '--textGraphics' argument to the command to make the program run in text mode instead of showing graphics. E.g:
  ```
  python3 py/pacman.py --layout tinyMaze --pacman GoWestAgent --textGraphics
  ```

We note that, the provided commands are designed to work with Mac/Linux with Python version 3. If you use Windows (like me!), we recommend that you run the code in the Windows command line (CMD), and make the following changes:

- Please use `\` instead of `/` while specifying the path to the file. (This isn't strictly necessary, Windows will accept both.)
- If it still does not work, you may use `py -3` instead of `python3` in the command you're executing on CMD. An example command that works for us is `py -3 py\autograder.py`.
- We suggest that you run the code on Command Line. You may use editors like PyCharm to write your code.

Implementation (in `search.py`)

- Please use python3 and write your own solutions from scratch. Do not import any packages yourself except for those we have included and specified.
- Please only implement your code at where we indicate.
- Please implement the "graph" search version, not the tree search version of each algorithm. That is, you will create a closed-set, and you will not expand the already expanded states again.
- In defining the close-set, please simply create a "set" (or "list") and insert visited/expanded states into the close-set by yourself. There is a "self.expanded" variable in `searchagent.py`, but we highly suggest that you do NOT use that variable since our grading script may not check it.
- Once you finish your implementation, you can execute the autograder by

```
python3 py/autograder.py
```

- The full grade shown in the autograder is 35, while the full grade of your programming part is 50.
- Please also read **Other information** at the end of this page for some other hints.

Task 1 (12.5 pts): Depth-First Search (DFS)

Open the file `py/search.py` and find the function `depthFirstSearch` (./py/search.py#L70) .

Take the provided template and finish the code (at "YOUR CODE HERE") so that depth-first search works. To do so, please first check the class `SearchProblem` (./py/search.py#L16) . This class outlines the structure of a search problem. It provides functions like `getStartState`, `isGoalState` (i.e., goal test), `getSuccessors`, and `getCostOfActions`. In your implementation, you will be using these functions to get necessary information about the search problem. Please note that, this is a abstract class that we put here to help you understand a search problem. The detail of the class is implemented in some other .py files by us already.

We suggest that you put the successors into the fringe in either the right-to-left or left-to-right fashion, not the others.

You can test it with pacman by running the following command:

```
python3 py/pacman.py -l mediumMaze -p SearchAgent -a fn=dfs
```

Task 2 (12.5 pts): Breadth-First Search (BFS)

Open the file `py/search.py` and and find the function `breadthFirstSearch` (./py/search.py#L90) .

Take the template and finish the BFS alorithm (at "YOUR CODE HERE"). We suggest that you put the successors into the fringe in either the right-to-left or left-to-right fashion, not the others.

You can test it with pacman by running the following command:

```
python3 py/pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

(Note that this should be simple if you've completed Task 1.)

Task 3 (12.5 pts): Uniform Cost Search (UCS)

Open the file `py/search.py` and find the function `uniformCostSearch` (./py/search.py#L96) .

Take the template and finish the code (at "YOUR CODE HERE") so that UCS works.

You can test it with pacman by running the following command:

```
python3 py/pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

(Note that adapting your implementation of DFS or BFS maybe useful for UCS.)

Useful Python code

Task 3 above asks you to implement UCS for Pacman. So, you want to have an open-set that's ordered by the accumulated cost. You may use `heaps` for this purpose. Whenever you `pop` a value from a heap, the lowest value comes out. Use a tuple to keep the value and other data together. For example:

```
from util import heappush, heappop
openset = []
heappush(openset, (5, "foo"))
heappush(openset, (7, "bar"))
heappush(openset, (3, "baz"))
heappush(openset, (9, "quux"))
```

```
best = heappop(openset)
print(best)
```

Alternatively, you can use the `PriorityQueue` data structures provided to you in `py/util.py`!

Task 4 (12.5 pts): A* Search

Open the file `py/search.py` and find the function `aStarSearch` (./py/search.py#L109) .

Finish the implementation of A* search (at "YOUR CODE HERE"). You can use the argument heuristic as a function: `dist = heuristic(state, problem)`. That is, try `h_start = heuristic(problem.getStartState(), problem); print(h_start)`. The class `nullHeuristic` (./py/search.py#L102) outlines the input and output of a heuristic function. We have implemented the heuristic funcstions.

You can test it with pacman by running the following command:

```
python3 py/pacman.py -l mediumMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

Other information

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 judgements -- will be the final judge of your score.

- Discussion: Please be careful not to post spoilers.
- Helpful Reading: Path Finding Algorithms (https://medium.com/omarelgabrys-blog/path-finding-algorithms-f65a8902eb40) , BFS and DFS (https://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/)

Important Tips

Keep these things in mind while working on your solutions!

- 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`, `PriorityQueue`, or the `heaps` (as mentioned in Task 3, Useful Python code) data structures provided to you in `py/util.py`! These data structure implementations have particular properties which are required for compatibility with the autograder. Note that, you can use `PriorityQueue` or `heaps` to implement `Stack` and `Queue`.
- Get familiar with the methods in the `SearchProblem` class in `py/search.py`! You'll need to use these methods as part of your search implementations.
- Remember that lists in Python are passed by reference; if you're seeing that actions show up in a list that shouldn't be there, make sure you're copying your actions to a new list every time!
- The autograder is not the final word! It is very possible to correctly implement these algorithms, but have the autograder consider it wrong because you didn't use the right data structures or methods. Final grades will be assigned by examining your implementation, not just using the autograder output.
  - Conversely, the autograder is not designed to catch every possible little detail. (It wouldn't be practical, nor do we want to.) It is possible to pass the autograder and still not receive full credit due

to a flaw in your code. In other words, the autograder does not excuse you from testing your code yourself.

Files you'll edit and submit:

- `py/search.py`: Where your search algorithms will reside.

Files you'll want to take a look at:

- `py/searchAgents.py`: Where all search-based agents are defined.
- `py/util.py`: Useful data structures you'll need for defining search algorithms.

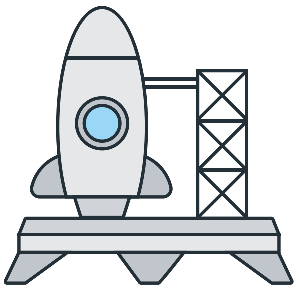
Supporting files you can ignore (unless you're curious):

- `py/pacman.py`: The main file that runs Pacman games. This file describes a `Pacman` `GameState` type, which you use in this project.
- `py/game.py`: The logic behind how the Pacman world works. This file describes several supporting types like `AgentState`, `Agent`, `Direction`, and `Grid`.
- `py/graphicsDisplay.py`: Graphics for Pacman
- `py/graphicsUtils.py`: Support for Pacman graphics
- `py/textDisplay.py`: ASCII graphics for Pacman
- `py/ghostAgents.py`: Agents to control Ghosts
- `py/keyboardAgents.py`: Keyboard interfaces to control Pacman
- `py/layout.py`: Code for reading layout files and storing their contents
- `py/autograder.py`: Project autograder
- `py/testParser.py`: Parses autograder test and solution files
- `py/testClasses.py`: General autograding test classes
- `py/test_cases/`: Directory containing the test cases for each question
- `py/searchTestClasses.py`: Testcases to support autograding

Choose a submission type

Upload

More



Choose a file to upload

File permitted: ZIP

or

Canvas Files