Michael Pedersen

ITCS-3153-001-Spring 2016-Intro Artificial Intelligence

hw2

In this assignment (adapted from UC Berkeley CS188), your Pacman agent will find paths through a 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. For this assignment, you will:

- continue using Python for AI programming,
- work with existing AI code/libraries, and
- implement informed search algorithms.

Environment Setup and Assignment Code

This assignment is a continuation of hw1, so you'll need a Python 2.7 environment and the same code and supporting files.

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, which 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.
autograder.py	Autograder for you to grade your answers on your machine

Assignment

If your Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal. All of the commands that appear in this project also appear in `commands.txt`, for easy copying and pasting. You can see the list of all options and their default values via:

```
python pacman.py -h
```

Q4: 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,heuristi
c=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).

Q5: 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! *Hint*: the shortest path through `tinyCorners` takes 28 steps.

Note: This part relies on a working BFS (Q2 from hw1).

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=Corners
Problem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=Corne
rsProblem
```

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.

Q6: Corners Problem: Heuristic

Note: Make sure to complete Q4 before working on Q6, because Q6 builds upon Q4.

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=cornersHeurist
```

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*. *Most* admissible heuristics are consistent.

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. 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!

Q7: 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`. 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.

Autograder

The command `python autograder.py` grades your solution. If you run it before editing any files, you will see that you fail all of the tests. The autograder will check 8 questions, but we are only considering questions 4-7 for this assignment.

HW1 Redux

If you did not complete hw1 satisfactorily, you'll have an opportunity to get 50% credit for Q1-Q3. Indicate in your README file that you would like us to regrade your hw1. All of the policies for academic integrity still apply. If you looked at a classmate's solution or found a solution online ***after*** the hw1 deadline, you should not request that your solutions be regraded.

Submission

By the due date, submit (via Moodle) a single ZIP file containing the following files:

- Modified versions of **search.py** and **searchAgents.py**
  - Your code should be well-commented and organized.
  - ***Do not*** change the names of any provided functions or classes within the code.
  - The autograder does not determine your final grade. Your code will be evaluated for technical correctness.
- **autograder.txt**
  - This file should contain the output of autograder.py
  - You can generate this file by typing the following at the command-line: `python autograder.py > autograder.txt`
- **readme.txt**
  - Use this plain-text file to explain any design decisions you made for this assignment and any other information that you think may help with grading. Describe any problems you ran into or errors in your submission. Also, be sure to clearly indicate if you would like hw1 regarded for 50% credit.

Your submission should be a single ZIP file, which includes only the 4 files specified above. Do not include any other files or internal folders in your submission. Part of your score for this assignment will be for following directions.

We will be checking your code against other submissions in the class for similarity. Modern cheat detectors are quite hard to fool, so please don't try. You are far better off submitting your own incomplete or non-functional code than taking a chance copying (or even looking at) code from a classmate or the Web. As stated in the course syllabus, violations of academic integrity are dealt with harshly.

Submission status

Submission status	No attempt
Grading status	Not marked
Due date	Monday, 22 February 2016, 11:00 AM
Time remaining	13 hours 44 mins
Last modified	Thursday, 11 February 2016, 9:46 PM
Submission comments	▸ Comments (0)

Add submission

Make changes to your submission