AI Challenge 3: A* and Heuristics

COSC4550/COSC5550
Artificial Intelligence
University of Wyoming

1 Overview

In this Challenge you will implement A* and various heuristics to increase its performance.

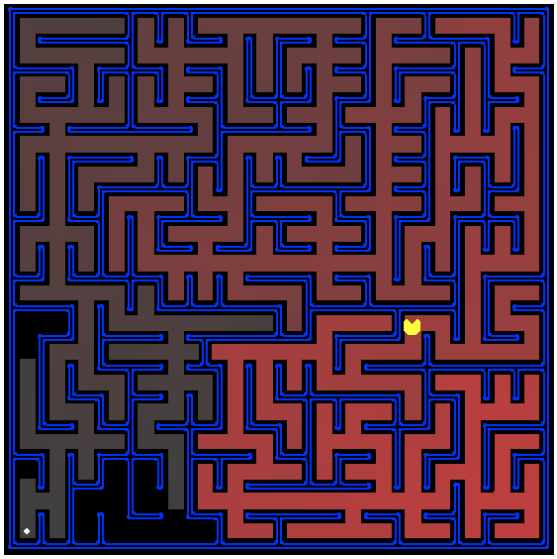

Figure 1: A picture of the Pacman maze.

Acknowledgment: This assignment is based on the one created by Dan Klein and John DeNero given as part of Berkeley's CS188 course. This assignment was also inspired by the modifications made by Peter Stone in his CS343 course of 2012. We thank Dan and John for creating the assignment and granting the permission to use it and we thank Peter for the ideas on how to adapt the assignment for this course.

1.1 Chapters

Chapters are from the book 'Artificial Intelligence, A Modern Approach', third edition, by Stuart Russel and Peter Norvig. The relevant chapter for this challenge is chapter 3, section 5. The most relevant sub-section is 3.5.2 (A* search).

1.2 Program files

The archive for this challenge contains a number of Python files. These files have been tested on, and should work with, Python version 2.7.3 and Python version 2.6.6. More recent versions of Python 2.x probably work as well, but these files do not work with Python 3.x.

The code-base for this challenge is very similar to the code-base of challenge 2 but not identical. Do not assume that what was true for the challenge 2 is also true for this one. The important files for this assignment are:

search.py	This file should be extended with your implementations of A* and its heuristics.
searchAgents.py	This file contains the search agents for this assignment.
pacman.py	The main file that runs Pacman games. This file describes a Pacman GameState type, which you will 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	Use this tool to test the correctness of your algorithms.

After downloading the code, unzipping it and changing to its directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

1.2.1 The autograder

For this assignment we provide you with an autograder, which is a program that will check the correctness of your algorithms in various scenarios. Note that the autograder is not perfect and you might still have some bugs in your code even if you pass all tests. It is also possible, that the autograder will 'fail' a correct solution because of a minor implementation difference. If you feel that

the autograder is mistakenly failing a correct solution, please let us know, and we will try to update the autograder to account for your case. That said, the autograder has been tested by many students before you, so it is quite likely that the autograder is working as intended, and that you will have to debug your solution.

The autograder has various options which might help you to debug your program. Useful options include `-p` which will print the test case before doing the test and `--no-graphics` which will not display the Pacman game for faster grading. To see all options us `-h`.

1.3 Deliverables

For this challenge you should submit your version of *search.py*. Please, to make it easier to distinguish files from different students, rename your file to *[yourname]_ search.py* before submitting.

Important: Make absolutely sure that your implementation will run all questions without any modifications being necessary on our part. It should run on either python 2.7.3 or python 2.6.6 and, when in doubt, you can always test your implementation on hive. You will only receive partial credit for implementations that do not run.

To run your solution on hive, first copy your project to hive (hive.cs.uwyo.edu) using any protocol accepted by hive (such as scp), then ssh onto hive and run your solution. Do not forget to use the `-q` option, as you will not have a display on hive and the program will crash if you try to run it without the `-q` option. The `-q` option is not necessary for the autograder as the autograder does not require any visualizations.

2 Questions

This challenge is a continuation of challenge 2, and as a result most of the hints for challenge 2 also work for the questions on this challenge. Your uniform-cost-search implementation from the previous challenge should make a good starting point for this challenge.

2.1 Question 1 (10 points): 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.

For this part of the assignment, 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 can test your code with the autograder using the following command:

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

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.)

2.1.1 Grading: 10 points

You will get full credit if your algorithm solves the problem using A* with the provided *manhattanHeuristic*. Test the correctness of your solution by running the autograder.

2.2 Question 2 (4 points): Implement a heuristic

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). Your task is to implement a heuristic for the *CornersProblem* in *cornersHeuristic* found in *searchAgents.py*. Note that for some mazes like *tinyCorners*, the shortest path does not always go to the closest food first!

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

You can test your code with the autograder using the following command:

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

Important: Before starting this assignment make sure your A* implementation is entirely correct! It is very easy to make a mistake in A* that will cause it to expand too many nodes. If your A* has one of these mistakes you might not be able to solve *mediumCorners* in any acceptable time. Use print statements and the autograder to detect and fix these issues before attempting this or the next question.

2.2.1 Hints and Observations

- Remember, heuristic functions just return numbers, which, to be admissible, must be equal to or lower than the lower bound for reaching the goal (i.e. it must never overestimate the cost of reaching the goal).
- The shortest path through *tinyCorners* takes 28 steps, while the shortest path in *mediumCorners* takes 106 steps.
- The corners heuristic takes two arguments: `state`, and `problem`. The state will be a `CornerState` and the problem will be a `CornersProblem`, both are defined in `searchProblems.py`. Look at the class definitions to see which functions are available for you to use.

2.2.2 Grading: 4 points

Test your solution on the *mediumCorners* map. If your heuristic is admissible and consistent, you will receive the following score, depending on how many nodes your heuristic expands.

Nodes expanded	**Points COSC 4550**	**Points COSC 5550**
$nodes > 1600$	2	2
$1600 \geq nodes > 1200$	4	3
$1200 \geq nodes > 800$	+0.5 extra credit	4
$800 \geq nodes > 692$	+0.5 extra credit	+1 extra credit
$692 \geq nodes$ (current record)	+1 extra credit	+1 extra credit

Important: You heuristic has to be admissible and consistent in order to receive full credit! Check admissibility and consistency by running the autograder for this question.

2.3 Question 3 (6 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 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. 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
```

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 4902 search nodes.

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
```

You can test your code with the autograder using the following command:

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

Our UCS agent finds the optimal solution in about 13 seconds, exploring over 16,000 nodes.

2.3.1 Hints and Observations

- The food heuristic takes two arguments: `state`, and `problem`. The state will be a tuple consisting of a `position` and a `foodGrid` (see `game.py` for the grid implementation), and the problem will be a `FoodSearchProblem` (defined in `searchProblems.py`). Look at the class definitions to see which functions are available for you to use.

2.3.2 Grading: 6 points

If your heuristic is admissible and consistent, you will receive the following score, depending on how many nodes your heuristic expands.

Nodes expanded	**Points COSC 4550**	**Points COSC 5550**
$nodes > 15000$	3	3
$15000 \geq nodes > 12000$	4	4
$12000 \geq nodes > 9000$	6	5
$9000 \geq nodes > 7000$	+0.5 extra credit	6
$7000 \geq nodes > 1516$	+0.5 extra credit	+1 extra credit
$1516 \geq nodes$ (current record)	+1 extra credit	+1 extra credit

Important: You heuristic has to be admissible and consistent in order to receive full credit! Check admissibility and consistency by running the autograder for this question.

3 FAQ

Q: *How do heuristics work with problems that have more than one goal?*

A: In the Corners problem and the AllFood problem you need to collect more than one piece of food. For these problems the heuristic should give an estimate for how many steps it would take to collect *all* pieces of food. Fore example, given the following maze:

```
+--+
|**|
|.p|
|**|
+--+
```

Here `*` indicates food, `.` indicates an empty space, and `p` indicates Pacman's current position. The most accurate prediction is 5; it takes at least 5 steps for Pacman to collect all the food. As a result, for your heuristic to be admissible, your heuristic is not allowed to return more than 5.

Also remember that your heuristic should be based on the current state of the world, and it should thus take into account when food is eaten. For example, if a few steps later the maze looks like this:

```
+--+
|..|
|p.|
|**|
+--+
```

Your heuristic may not return more than 2, since all food can be collected in 2 steps.

Obviously walls may increase the number of steps necessary to complete the problem. The following maze, for example, needs at least 9 steps to solve:

```
+---+
|*|*|
|..P|
|*|*|
+---+
```

However, this doesn't mean you heuristic has to return exactly 9, it may return any number that is lower than 9. If your heuristic would return 7 (the number of steps required if the walls weren't there) your heuristic would still be admissible.

Q: *I receive a TimeoutFunctionException when running the autograder, what is wrong?*

A: For the heuristic questions your algorithm may not take longer than 5 minutes to complete. If you see this exception then it means that your

code is simply too slow. The most common issues that can make your code too slow are: an inefficient A* star algorithm, an inefficient heuristic, calling your heuristic too many times.

To check for the first problem, run your A* algorithm with the Null heuristic; if that is too slow, try to optimize your A* algorithm. To check if your heuristic is called too many times, add a counter in your heuristic function (put a global variable at the top of your file, and increment it at the top of your heuristic function. See: http://www.python-course.eu/global_vs_local_variables.php for information on using global variables). Your heuristic function should be called once for every state. If you don't run into either of those problems, try to optimize your heuristic function.