AI Challenge 2: Pacman Search

COSC4550/COSC5550
Artificial Intelligence
University of Wyoming

1 Overview

In this Challenge you will implement the search algorithms: depth-first search, breadth-first search and uniform-cost search to help Pacman clear-out all the food in the maze.

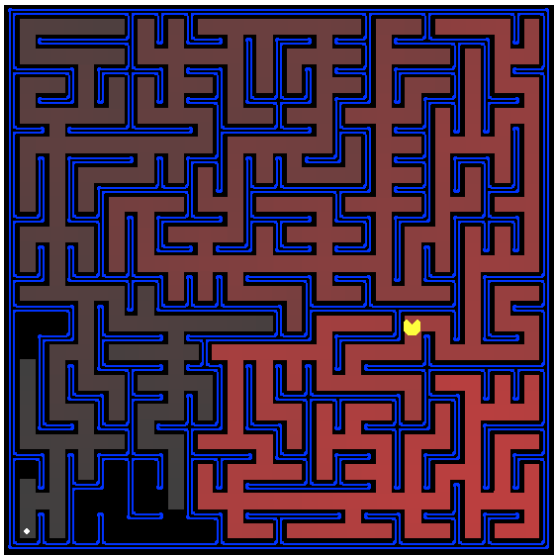

Figure 1: A picture of the Pacman maze.

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 4. The most relevant sub-sections are 3.4.1 (Breadth-First search), 3.4.2 (Uniform-cost search), and 3.4.3 (Depth-first 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 the Cleaning Agents challenge but not identical. Do not assume that what was true for the Cleaning Agents challenge is also true for this one. The important files for this assignment are:

search.py	This file should be extended with your implementations of the various search algorithms, and it should be handed in.
searchAgents.py	This file describes various search agents. Especially useful for question 4.
searchProblems.py	This file describes various search problems. Especially useful for question 4.
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
```

You can test a very simple reflex agent, called the GoWestAgent (which always goes west) on a very simple maze using the following command:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

On this maze the agent actually wins. However, things go awry when turning is required:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

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
```

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.

Before you submit, make sure that your algorithm works on both the provided test problems, as well as on the autograder. Also, keep in mind that, while the autograder has been thoroughly tested by many students, it is still possible that it will reject correct solutions. If you are absolutely sure that your solution is correct, but the autograder does not accept it, please let us know, and we will try to update the assignment (provided we agree that your solution is correct).

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

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! Pseudocode for the search algorithms you'll write can be found in the lecture slides and textbook. 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.

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).

Hint: Make sure to check out the *Stack*, *Queue* and *PriorityQueue* types provided to you in *util.py*!

Hint: Each algorithm is very similar. Algorithms for DFS, BFS and UCS differ only in the details of how the frontier is managed. So, concentrate on getting DFS right and 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). Do note that the supplied stacks and queues do *not* necessarily allow you to create the most generic search method possible, some changes have to be made in order to make this work.

2.1 Question 1 (4 points): Depth-first search

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. In addition, make sure your algorithm can deal with arbitrary states and transitions; the autograder will test your implementation on abstract trees, and your implementation has to be able to handle those. Lastly, while there exist variants of depth-first search that handle the frontier differently, you will need to implement a version that *does not* add nodes to the frontier that are already in the frontier, as described in the pseudo code for GRAPH-SEARCH in figure 3.7 (page 77) of the text book.

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
```

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

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

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?

2.1.1 Hints and Observations

- It is highly recommended that you implement a loop-based depth-first search rather than a recursive depth-first search. The loop-based algorithm will make it easier to keep track of the path, and it can be reused in later questions.
- The number of nodes expanded depends on the number of calls you make to *getSuccessors*. To make your solutions correct you should call this function only when truly necessary.
- If you use a Stack as your data structure, the solution found by your DFS algorithm for *mediumMaze* should have a length of 130 (provided you push successors onto the frontier in the order provided by *getSuccessors*; you might get 244 if you push them in the reverse order).
- If Pacman moves too slowly for you, try the option `--frameTime 0`.

2.1.2 Grading: 4 points

You will get full credit if your algorithm solves the problems in a depth-first manner. Test the correctness of your solution by running the autograder.

2.2 Question 2 (4 points): 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, and make sure your algorithm can deal with arbitrary states and transitions. In addition, implement your algorithm such that it tests whether a state is a goal state before pushing it into the frontier. 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
```

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

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

2.2.1 Grading: 4 points

You will get full credit if your algorithm solves the problems in a breadth-first manner. Test the correctness of your solution by running the autograder.

2.3 Question 3 (4 points): Uniform cost search

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*, and make sure your algorithm can deal with arbitrary states and transitions. Important: in contrast to breadth-first search you should *only* check the goal when you pop it from the frontier, if you check for the goal before pushing it into the frontier you may miss the optimal path. 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
```

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

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

2.3.1 Hints and Observations

- The first agent uses a cost function where moving into any square has a cost of one. The second agent uses a cost function where moving into a square (x, y) costs $(\frac{1}{2})^x$, causing it to stay east. The third agent uses a cost function where stepping onto a square (x, y) has a cost of 2^x, causing it to stay west.
- Costs are associated with actions, not with nodes. This means that when we say that the cost of moving to a square is c, we actually mean that the cost of the actions to reach this node from any of its neighbors is c. For other cost functions it would be perfectly possible that moving to a square from the east is more expensive than moving to that same square from the west.
- You should get very low and very high path costs for the *StayEastSearchAgent* and *StayWestSearchAgent* respectively, due to their exponential cost functions.

2.3.2 Grading: 4 points

You will get full credit if your algorithm solves the problem using Uniform Cost Search. Test the correctness of your solution by running the autograder.

2.4 Question 4 (5 points, COSC5550 only, COSC4550 bonus): Approximate search

(Mandatory for those enrolled in the graduate student version of the class. Bonus points for undergraduates)

Sometimes finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. Implement an *ApproximateSearchAgent* in *search.py* that finds a short path through the *bigSearch* layout. You can use the *ClosestDotSearchAgent* in *searchAgents.py* as a starting point.

To test your solution directly use:

```
python pacman.py -l bigSearch -p ApproximateSearchAgent -z .5 -q
```

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

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

We will time your agent using the no graphics option `-q`, and it must complete in under 30 seconds on our grading machines. If you are unsure if your solution will make the time limit, try your solution on hive first before handing it in.

2.4.1 Grading: 5 points

If your agent solves the maze in under 30 seconds your grade will depend on the total path cost your agent accumulates during its run.

Path cost	**Points COSC 4550**	**Points COSC 5550**
$cost > 340$		3
$340 \geq cost > 325$	+0.5 extra credit	4
$325 \geq cost > 298$	+0.5 extra credit	5
$298 \geq cost$ (current record)	+1 extra credit	+1 extra credit

3 FAQ

Q: *How do I use a data-structure from* ***`util.py`****?*

A: You can create a data-structure from `util.py` like this: `my_stack = util.Stack()`. The same goes for the Queue and the PriorityQueue, you just replace Stack with Queue or PriorityQueue respectively.

Q: *The data-structures provided in* ***`util.py`*** *do not provide the functionalities that I need, should I stick with them and find another way, or should I use a different data-structure?*

A: The data-structures provided in `util.py` are provided 'as is', and if they do not work for you, do not hesitate to use different data-structures instead. That said, it is possible to complete the assignment with the data-structures from `util.py`, so do be careful that you are not making things needlessly complex.

Q: *My code traverses nodes in the correct order, but the autograder does not accept my solution. What is wrong?*

A: Difficult to say without further information, but you can look at the output of the autograder for a hint. For example, if the student solution does not match either of the 'correct' solutions, there is something wrong with the

path that you return. Remember that your function should return a list of actions that brings your agent from the start to the goal without the backtracking that your search algorithm did.

Q: *My solution works correctly on the maze, but the autograder keeps complaining. What is wrong?*

A: There are many possible problems, but one thing to keep in mind is that the autograder does not use the same kind of states or actions. For example, the Pacman maze has actions like `'West'` and `'East'`, but the autograder may have action like `'2:A->D'` and `'0:D->G'`. To avoid issues, all you have to do is make sure you get all your actions and states from the appropriate functions (e.g. `problem.getStartState()`, `problem.getSuccessors(state), ect.`). Do not make any assumptions about what a state or action may be, they may be positions, they may be strings, or they may be complicated objects.

Q: *How can I look at the maze without the screen immediately closing upon completion?*

A: There are a few ways to slow down the simulation to allow you to look at the maze more closely. First, you can use `--frameTime 200` to increase the time it takes to draw a frame. Second, you can make a screenshot of the maze. Third, you can add a sleep command to your code: `import time; time.sleep(60)`. Finally, you can look at the maze files (with the `.lay` extension) directly with a text editor; they are located in the `layouts` folder.

Q: *How do I return the path found by my algorithm?*

A: Because your search algorithm may backtrack, or find multiple different paths to the same location, you can not simply keep track of a single path; you need a separate path for every state. The easiest way to keep track of a path for each state is to use an (implicit) linked list. That is, each state should track from which state it was discovered, such that you can recreate the path that led to this state.

To create 'pointers' in Python you can use lists. For example, to create a short, two element, list that also stores the action at each state you can use the following code:

```
initial_state = [position, None, None]
next_state = [new_position, action, initial_state]
```

To traverse the list until you get back to the initial position you can do something like this:

```
while state:
    position, action, state = state
    print action
```

Q: *How do I replace a high cost path in Uniform Cost Search?*

A: That depends on how you store your current frontier. If you use a list as your frontier, and the items you store look something like this `[position, action, cost, previousState]`, you can manually search and remove other items like this (don't forget to add the new item if you removed a worse item):

```
i = 0
while i < len(frontier):
    position, action, cost, previousState = frontier[i]
    if((position == newPosition) and (cost > newCost)):
        del frontier[i]
    else:
        i+=1
```

If you use a priority queue you can not easily remove items, but you can add the same item twice with different costs. Because the lowest cost item will always be popped first, as long as you immediately check that you did not already visit that state, you do not actually need to remove the other items. The following example will never visit the same position twice, and it will always return the lowest cost path first if the frontier is a properly managed priority queue.

```
position, action, cost, previousState = frontier.pop()
if position in visited:
    continue
visited.append(position)
```