Assignment 1: Search

Version 2.00. Last Updated: 10/02/2021

Publication Date: Tuesday 09/02/2021
Due Dates:

Friday groups:	**Friday 12/03/2021 at 08:59 pm**
Wednesday groups	**Wednesday 10/03/2021** at 08:59 pm
Thursday groups:	**Thursday 11/03/2021** at 08:59 pm

All those colored walls,
Mazes give Pacman the blues,
So teach him to search.

Credit: This assignment is based on https://inst.eecs.berkeley.edu/~cs188/fa20/projects/

Important

Anti-plagiarism, copying & publication of results

As Berkeley's disclaimer says, you are welcome to use the Pac-Man projects and infrastructure for any educational or personal use. However, **it is strictly prohibited to distribute or post solutions to any of the projects**. Furthermore, following the ACADEMIC EVALUATION REGULATIONS of the ESCUELA POLITÉCNICA SUPERIOR at the UNIVERSIDAD AUTÓNOMA DE MADRID, Article 14; "In the case of a **copy**, the subject will be graded in the call where the copy was produced with zero points. As an additional measure, the teacher can initiate an informative file (a disciplinary procedure), according to the UAM Assessment Regulations".

Submission

The assignment must be completed in teams of two, and submitted by uploading it to Moodle.

Each submission should consist of a single zip file named **ia2021_pr1_gXXXX_YY.zip**, where XXXX is the lab group number, and YY is the identifier assigned to each person/pair in your group lab.

It is important to always write the authors's name and the team number assigned to you in your lab group in all the files that you submit.

Submission contents

Your zip file should consist of a **single folder**, named as the zip file but without the extension, containing all your assignment files. This folder should contain:

- The **complete code** including the files you modified. You should **only** change the files required below, and only in the parts marked with "*** YOUR CODE HERE ***". Do not include any additional files.

- A file `autograder_results.txt` with the output of the autograder.

- A **report** containing a summary of the concepts covered in this assignment and a section for each exercise including:

  1. a personal comment on your approach to the problem and the decisions you took in order to tackle it, and why;
  2. a list and an explanation of the framework's functions you have used;
  3. the code you added;
  4. screenshots of the tests carried out to verify its correct operation;
  5. your conclusions about the observed behavior and answers to each of the questions posed in the section

Grading

Grading will be divided between the report (40%) and the code (60%). Both elements will be evaluated in the scale 0-10.

Your tasks

The goal of this assignment is to explore the part of the course dedicated to search. To do this we will use a modified version of the software provided by the University of Berkeley, that you can download from Moodle. The programming language is Python (version 3.6 or higher), although it is not necessary to know it beforehand. The Berkeley portal has a simple tutorial covering Python's basics (https://inst.eecs.berkeley.edu/~cs188/fa20/project0/), that you are encouraged to at least review.

The assignment consists of 6 exercises.

Let's get started...

Introduction

In this project, your Pacman agent will find paths through his 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. This project includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

The Python tutorial linked above has more information about using the autograder.

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. You can download all the code and supporting files as a zip archive

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'll edit	
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.
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.
`autograder.py`	Project autograde.
`testParser.py`	Parses autograder test and solution files.
`testClasses.py`	General autograding test classes.
`test_cases/`	Directory containing the test cases for each question.
`searchTestClasses.py`	Autograding test classes for this assignment.

Files to Edit and Submit: You will fill in portions of `search.py` and `searchAgents.py` during the assignment, to be included in your submission as described above. Please *do not* change the other files in this distribution.

Code 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. We will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours and the discussion forum are there for your support; please use them. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Discussion: Please be careful not to post spoilers.

Welcome to Pacman

After downloading the code (search.zip), unzipping it, and changing to the directory, 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`.

A useful function that may interest you is the frameTime option (in seconds), which tells Pacman how fast it is going. Modifying it can help you to debug or simply see the game better. For example:

```
python pacman.py --frameTime 0.5
```

Question 1 (1 point): 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! Pseudocode for the search algorithms you'll write can be found in the theory lecture slides. Remember that for the assignment 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).

Important note: Make sure to **check** data structures provided to you in `util.py` ! These data structure implementations have particular properties which are required for compatibility with the autograder and will ease your programming.

Important note: When you implement the search algorithms, in the case that the search fails (the algorithm does not find a solution), return False or None.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the nodes are 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).

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
```

```
python pacman.py -l mediumMaze -p SearchAgent
```

```
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

***Remember** to print screenshots of these tests and explain what you have observed in your report.

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration).
Question to answer: Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Hint: 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 fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order).
Question to answer: Is this a least cost solution? If not, think about what depth-first search is doing wrong.

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

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

Question 2 (1 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. Test your code the same way you did for depth-first search.

```
python pacman.py -l tinyMaze -p SearchAgent
```

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

```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

***Remember** to print screenshots of these tests and explain what you have observed in your report.

Question to answer: 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
```

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

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

Question 3 (2 points): Varying the Cost Function

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 (UCS) 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
```

***Remember** to print screenshots of these tests and explain what you have observed in your report.

Note: You should get very low and very high path costs for the `StayEastSearchAgent` and `StayWestSearchAgent` respectively, due to their exponential cost functions (see `searchAgents.py` for details).

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

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

Question 4 (2 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. 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=manhat
```

***Remember** to print screenshots of these tests and explain what you have observed in your report.

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).
Question to answer: What happens on `openMaze` for the various search strategies?

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

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

Question 5 (2 points): 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: 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 (and justify in the report) 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=CornersProble
```

***Remember** to print screenshots of these tests and explain what you have observed in your report.

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 1: 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.

Hint 2: When coding up `getSuccessors`, make sure to add children to your successors list with a cost of 1.

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.

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

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

Question 6 (2 points): 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=cornersHeurist
```

***Remember** to print screenshots of these tests and explain what you have observed in your report.

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

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).

Question to answer: Explain the logic behind your heuristic.

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 1600	0/3
at most 1600	1/3
at most 1200	2/3
at most 700	3/3

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

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

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