## Introduction

In this project, your Pacman agent will find paths through the 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.

As in homework 1, this project includes an autograder for you to grade your answers on your machine. This can be run by running the python module autograder.py in PyCharm or by typing the following in a terminal window:

python autograder.py

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 file: homework2.zip.

**Files you'll edit:**

search.py: where all of your search algorithms will reside

**Files you might want to look at:**

searchAgents.py: where all of your search-based agents will reside.
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 autograder
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:  homework2 specific autograding test classes

**Files to Edit and Submit:** You will fill in portions of search.py during the assignment. You should submit this file with your code and comments. Please *do not* change the other files in this distribution or submit any of the original files other than this file.

**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. If necessary, 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.

**Getting Help:** You are not alone! If you find yourself stuck on something, please ask for help. Office hours  and the discussion forum are there for your support; please use them. I want these projects to be rewarding and instructional, not frustrating and demoralizing. But, I 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 and unzipping it, you should be able to play a game of Pacman by running pacman.py in PyCharm or 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 closing the window.

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

---

## Question 1 (3 points): 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. As you work through the following questions, you might find it useful to refer to the object glossary.

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. 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).

***Important note:*** Make sure to **use** the Stack, Queue and PriorityQueue data structures provided to you in util.py! These data structure implementations have particular properties which are required for compatibility with the autograder.

*Hint:* Each algorithm is very similar. Algorithms for DFS, BFS and UCS differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. You may find the Node data structure (in util.py) useful in your implementation.

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
```

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?

*Hint:* If you use a Stack as your data structure, the solution found by your DFS algorithm for mediumMaze should have a cost 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). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

---

### Question 2 (3 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 mediumMaze -p SearchAgent -a fn=bfs

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

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
```

---

### Question 3 (3 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 in the uniformCostSearch function in search.py. 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
```

*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).

### Submission

Upload **search.py** to get credit on your project.