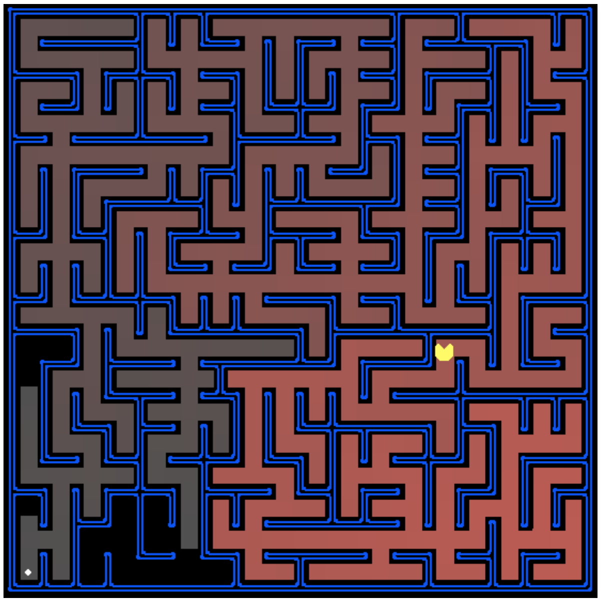
CPSC 481 Artificial Intelligence
Project 1 – State Space Search for Pacman

Mode: Team of up to four persons
Attempt: Multiple times before the due date
Due Date: As shown

Welcome to Pacman



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 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 (**written in Python 3.6**).

Files you'll edit:

search.py	Where all of your search algorithms will reside.

Files you might want to look at:
pacman.py The main file that runs Pacman games. This file describes a Pacman Game State type, which you use in this project.
game.py The logic behind how the Pacman world works. This file describes several supporting types like Agent State, Agent, Direction, and Grid.
util.py Useful data structures for implementing search algorithms.
searchAgents.py A variety of different types of Pacman agents.
test_cases/ Directory containing the test cases for each question

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
searchTestClasses.py Project 1 specific autograding test classes

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

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. 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, office hours, section, and the discussion forum are there for your support; please use them. If you can't make our office hours, let us know and we will schedule more. 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.

Getting Started:

Step 1: After downloading the code (**search.zip**), unzipping it.
Step 2: While holding down Windows key press R (or click Start) and then type cmd. Press Enter.
Step 3: Move into the folder containing autograder.py and pacman.py.

To access the Unix command prompt in Mac OS X, open the Terminal application.
It is located by default inside the Utilities folder, which in turn is inside the Applications folder.

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.

Question 1: 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 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 A* differ only in the details of how the fringe 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.

Implement the depth-first search (DFS) algorithm in the depthFirstSearch function in search.py.

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 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). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

Question 2: 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
```

Plz. Ignore questions 3-8 for this project.

Grading for this project:
Q1: 4.5 pts
Q2: 4.5 pts
There is a project 1 rubric attached below project 1 description on Canvas.
Your program needs to pass autograder tests.
You can find the test cases located in "test_cases" folder in search.zip
Autograder will not only test Pacman but also some search graphs in the "test_cases" folder.

Deliverables:
Include team members in a readme file.
Submit search.py and readme to the submission link.
One team only submit ONE copy. Pick a team member who will be responsible for submitting the project.
Other members do not need to submit. Canvas may mark your project as "Missing", but you do not need to worry about it.
I'll contact you if I cannot find your name in any readme files.