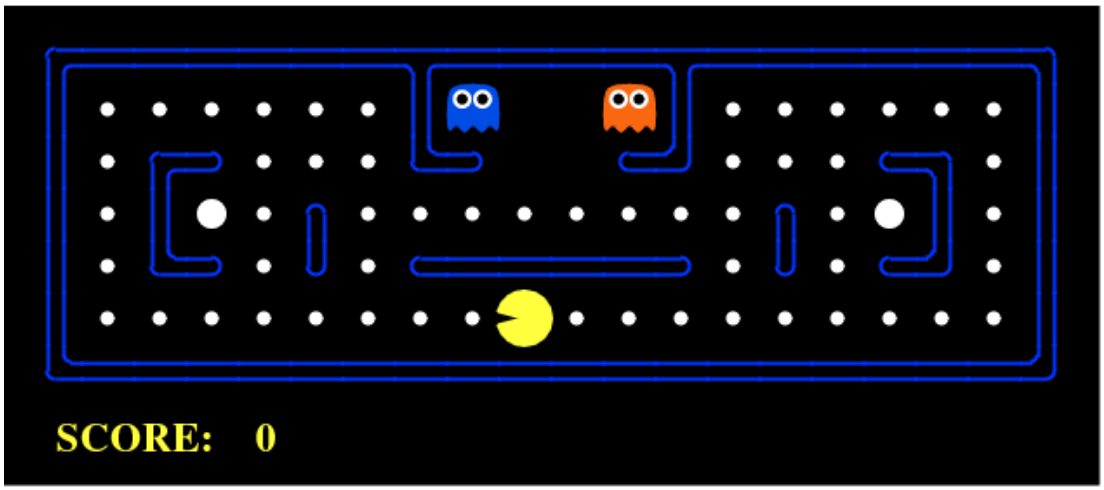
CMP4501 Introduction to Artificial Intelligence and Expert Systems
Project Assignment Section 1

For your term project, we will be following the pac-man game given in http://ai.berkeley.edu/project_overview.html.



Your first assignment covers the tutorial and the search sections. This project uses Python 2.7 (and you can't run it with 3.x without modifications).

If you don't have Python installation in your computers, you can use Anaconda environments to ease the installation procedure.

https://www.anaconda.com/

You can follow the instructions from
https://docs.anaconda.com/anaconda/install/
for installation.

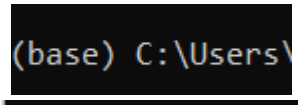
Once installed, you can open the Anaconda prompt (Command window with anaconda support) through the shortcut. Generally, the default environment (workspace) comes with Python 3.x installation. You can create an environment with Python 2.7 using the below codes:

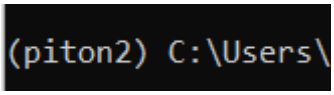
conda create –name piton2 python=2.7

Once the environment is created, you can activate it using

activate piton2

You can check which environment you are currently on using the info on the parenthesis,





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.

For a quick introduction regarding python and Unix commands, you can follow the information http://ai.berkeley.edu/tutorial.html

Tutorial

You will fill in portions of addition.py, buyLotsOfFruit.py, and shopSmart.py in tutorial.zip during the assignment. You should submit these files with your code and comments. 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 auto-graded 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 auto-grader. However, the correctness of your implementation -- not the auto-grader'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.

To get you familiarized with the auto-grader, we will ask you to code, test, and submit solutions for three questions.

Unzip tutorial.zip file and examine its contents.

This contains a number of files you'll edit or run:

- addition.py: source file for question 1
- buyLotsOfFruit.py: source file for question 2
- shop.py: source file for question 3
- shopSmart.py: source file for question 3
- autograder.py: autograding script (see below)

and others you can ignore:

- test_cases: directory contains the test cases for each question
- grading.py: autograder code
- testClasses.py: autograder code
- tutorialTestClasses.py: test classes for this particular project
- projectParams.py: project parameters

The command ``python autograder.py'' grades your solution to all three problems. If we run it before editing any files we get a page or two of output:

```
Finished at 23:39:51
```

```
Provisional grades
==================
Question q1: 0/1
Question q2: 0/1
Question q3: 0/1
-----------------
Total: 0/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Question 1: Addition

Open addition.py and look at the definition of add:

```
def add(a, b):
  "Return the sum of a and b"
  "*** YOUR CODE HERE ***"
  return 0
```

The tests called this with a and b set to different values, but the code always returned zero. Modify this definition to read:

```
def add(a, b):
  "Return the sum of a and b"
  print "Passed a=%s and b=%s, returning a+b=%s" % (a,b,a+b)
  return a+b
```

Now rerun the autograder:

Question 2: buyLotsOfFruit function

Add a buyLotsOfFruit(orderList) function to buyLotsOfFruit.py which takes a list of (fruit,pound) tuples and returns the cost of your list. If there is some fruit in the list which doesn't appear in fruitPrices it should print an error message and return None. Please do not change the fruitPrices variable.

Run python autograder.py until question 2 passes all tests and you get full marks. Each test will confirm that buyLotsOfFruit(orderList) returns the correct answer given various possible inputs. For example, test_cases/q2/food_price1.test tests whether:

Cost of [('apples', 2.0), ('pears', 3.0), ('limes', 4.0)] is 12.25

Question 3: shopSmart function

Fill in the function shopSmart(orders,shops) in shopSmart.py, which takes an orderList (like the kind passed in to FruitShop.getPriceOfOrder) and a list of FruitShop and returns the FruitShop where your order costs the least amount in total. Don't change the file name or variable names, please. Note that we will provide the shop.py implementation as a "support" file, so you don't need to submit yours.

Run python autograder.py until question 3 passes all tests and you get full marks. Each test will confirm that shopSmart(orders,shops) returns the correct answer given various possible inputs. For example, with the following variable definitions:

```
orders1 = [('apples',1.0), ('oranges',3.0)]
orders2 = [('apples',3.0)]
dir1 = {'apples': 2.0, 'oranges':1.0}
shop1 =  shop.FruitShop('shop1',dir1)
dir2 = {'apples': 1.0, 'oranges': 5.0}
shop2 = shop.FruitShop('shop2',dir2)
shops = [shop1, shop2]
```

test_cases/q3/select_shop1.test tests whether:

```
shopSmart.shopSmart(orders1, shops) == shop1
```

and test_cases/q3/select_shop2.test tests whether:

```
shopSmart.shopSmart(orders2, shops) == shop2
```

Search

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.

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 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 auto-grader
- testParser.py Parses auto-grader test and solution files
- testClasses.pyGeneral auto-grading test classes
- test_cases/ Directory containing the test cases for each question
- searchTestClasses.py Project 1 specific auto-grading test classes

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

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.

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, UCS, 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. (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
```

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
```

Question 3: 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. 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
```

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).

Question 4: 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=manhattanHeuristic

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). What happens on openMaze for the various search strategies?