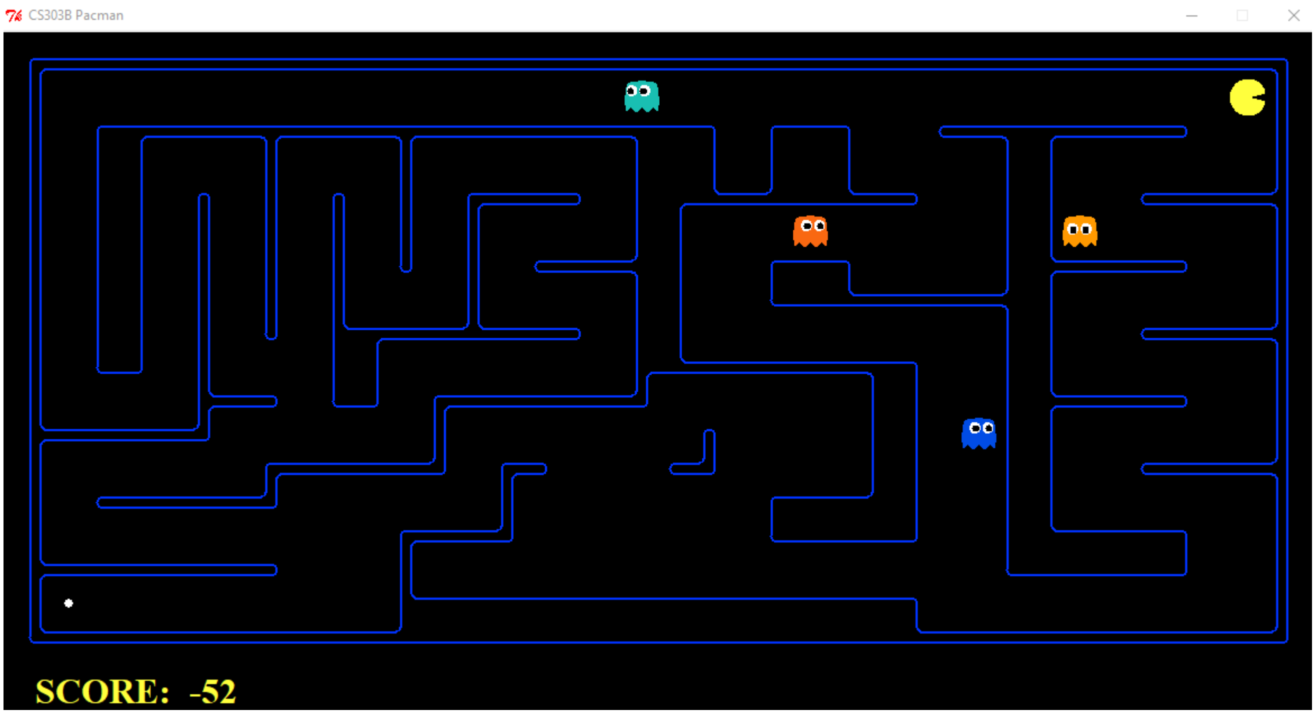
Assignment 1

Welcome to the first assignment!



Introduction

We will develop knowledge of the AI techniques for search that we have covered in the lectures:

1. Depth First Search (question 1)
2. Breadth First Search (question 2)
3. Cost Function and Uniform Cost Search (question 3)
4. Heuristics and A* Search (question 4)

In addition, we develop knowledge about the key concepts of a rational agent and goal directed behaviour (minimise the cost function and goals for path finding).

Hint: DFS, BFS and A* Search all have the same structure. If you implement DFS then it is not difficult to implement the other algorithms by changing the policy to choose how to explore the tree (what nodes to explore on the fringe, see lecture 2 and 3).

Files used for the assignment

First download the assignment files from blackboard (assignment_1.zip)

Files you will edit in the assignment (you will submit these 2 files only):

These are the only 2 files you should edit in the assignment:

- **`search.py`**	Where your search algorithms will be put.
- **`searchAgents.py`**	Where your agents will be put.

Files for the game (you should look at these to understand but DO NOT edit them):

`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.

Files to support the program and game (you don't need to look at these files):

`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`	Project 1 specific autograding test classes

Important: the assignment will be graded partially using an automated script so please don't change any files **except search.py** and **searchAgents.py** which you submit.

Pacman Game

Unzip the directory containing the assignment and run pacman to test if it is working.

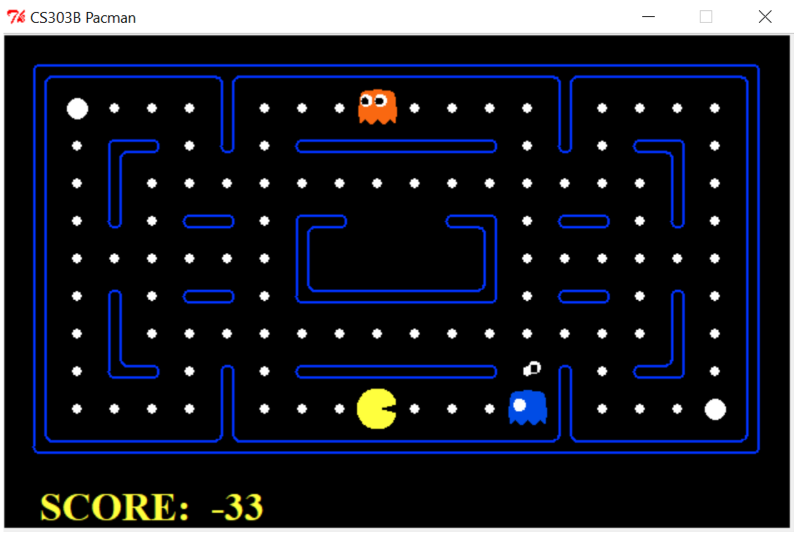
Navigate to the pacman directory in the command line and type:

- `python2 pacman.py`

Please note that the assignment used Python 2.7. If you have not installed Python 2.7 please review Lecture 2 and if still having problems seek help from the course staff.

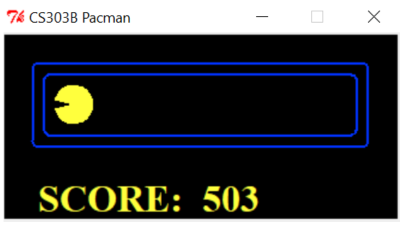
Now play the game a little bit to understand it:

- You can move the pacman agent North, East, South, West by the up, left, down, and right arrows.
- You receive 10 points for each pellet eaten,
- You loose 1 point for each second,
- You win if eat all pellets,
- If you eat a power pellet (large one) you make the ghosts scared for 10 seconds – can't eat you.



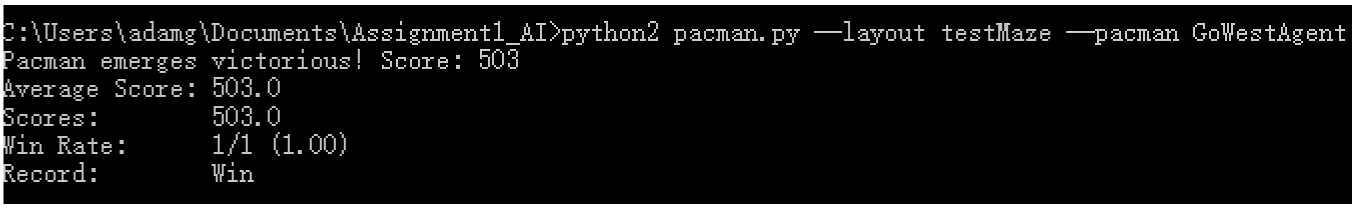
Now test a simple agent (a reflex agent which always goes west):

-
  ```
  python2 pacman.py --layout testMaze --pacman
  GoWestAgent
  ```



This agent doesn't do well if the maze is slightly more complex:

-
  ```
  python2 pacman.py --layout tinyMaze --pacman
  GoWestAgent
  ```



```
C:\Users\adamg\Documents\Assignment1_AI>python2 pacman.py —layout testMaze —pacman GoWestAgent
Pacman emerges victorious! Score: 503
Average Score: 503.0
Scores:        503.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Note: you can exit the game by pressing ctr-c and you can see a list of options by typing `python2 “pacman.py -h”`

Soon your agent will be able to solve any maze, much bigger than tinymaze!

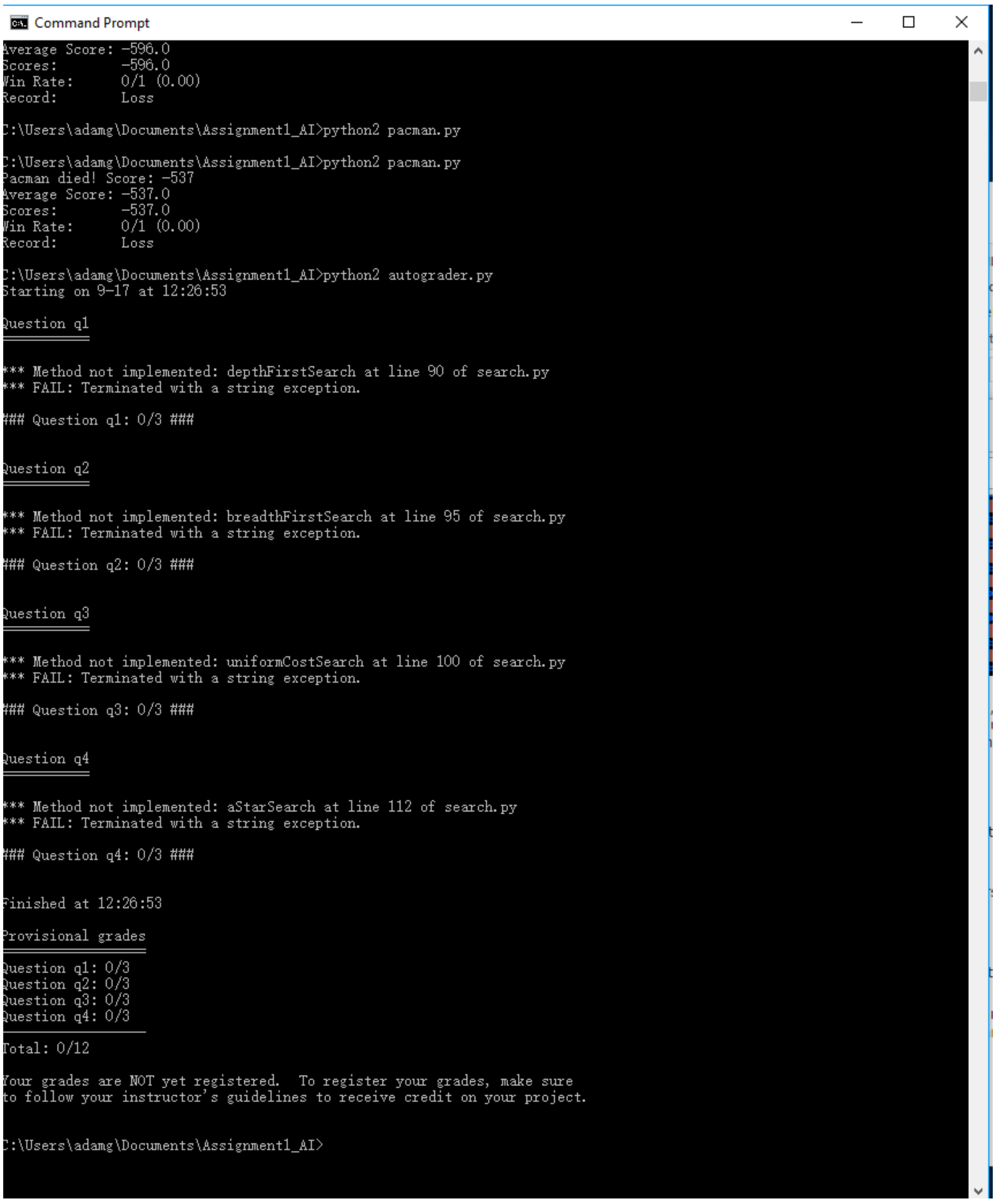
Autograder

You can use the autograder provided to test your work against some test cases that have already been implemented:

Type (at a command line):

- `python2 autograder.py`

You should see some output similar to the following (as you implement your solution will hopefully see your program is working):



```
Average Score: -596.0
Scores:        -596.0
Win Rate:      0/1 (0.00)
Record:        Loss

C:\Users\adamg\Documents\Assignment1_AI>python2 pacman.py

C:\Users\adamg\Documents\Assignment1_AI>python2 pacman.py
Pacman died! Score: -537
Average Score: -537.0
Scores:        -537.0
Win Rate:      0/1 (0.00)
Record:        Loss

C:\Users\adamg\Documents\Assignment1_AI>python2 autograder.py
Starting on 9-17 at 12:26:53

Question q1
===========

*** Method not implemented: depthFirstSearch at line 90 of search.py
*** FAIL: Terminated with a string exception.

### Question q1: 0/3 ###


Question q2
===========

*** Method not implemented: breadthFirstSearch at line 95 of search.py
*** FAIL: Terminated with a string exception.

### Question q2: 0/3 ###


Question q3
===========

*** Method not implemented: uniformCostSearch at line 100 of search.py
*** FAIL: Terminated with a string exception.

### Question q3: 0/3 ###


Question q4
===========

*** Method not implemented: aStarSearch at line 112 of search.py
*** FAIL: Terminated with a string exception.

### Question q4: 0/3 ###


Finished at 12:26:53

Provisional grades
==================
Question q1: 0/3
Question q2: 0/3
Question q3: 0/3
Question q4: 0/3
------------------
Total: 0/12

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


C:\Users\adamg\Documents\Assignment1_AI>
```

All questions are equally weighted in this assignment (3 points each for a total of 12).

Please develop your programs in small parts and test it by the autograder or just running it (eg python2 pacman.py –layout tinyMaze –pacman YourAgent) often to avoid numerous errors at the same time which will be difficult to solve…

Tree Search Algorithms

All of the questions in this assignment require you to implement tree search algorithms.

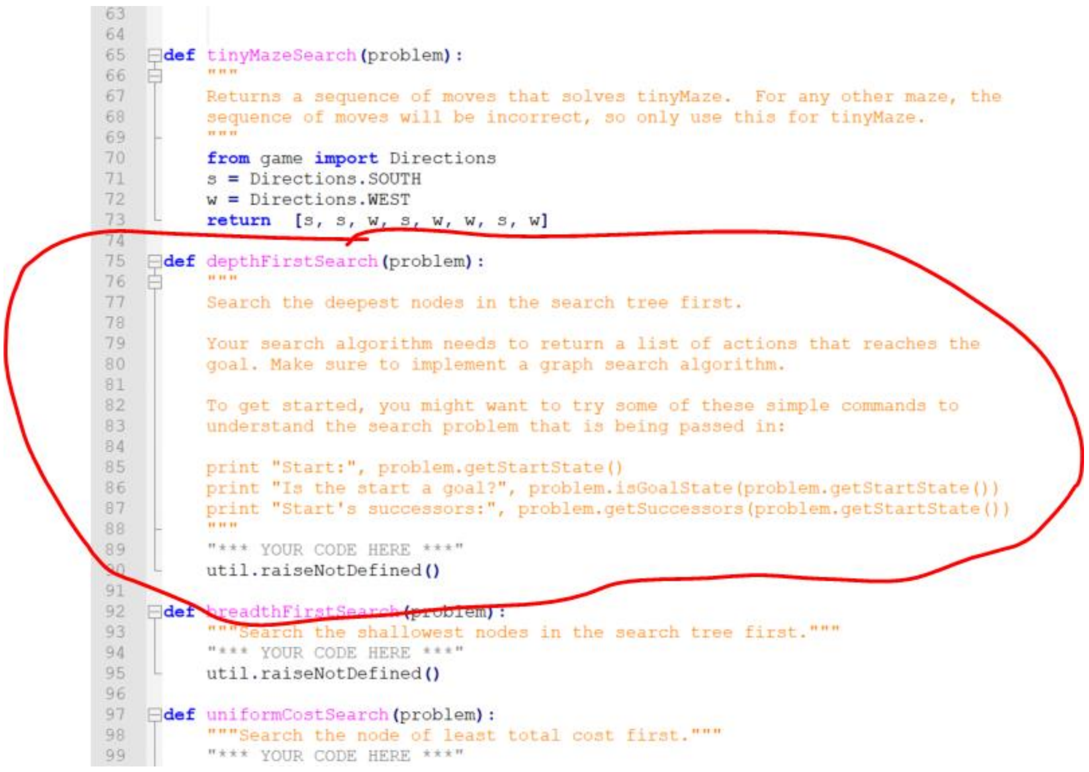
These algorithms follow a similar template:

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

Each algorithm is different by having a different strategy for leaf node expansion. Once you have done the first question (implement DFS) it should not be hard to complete the other questions.

Question 1: Finding a Fixed Food Dot with Depth First Search

This question requires you to implement DFS in the file search.py:



```python

def tinyMazeSearch(problem):
    """
    Returns a sequence of moves that solves tinyMaze.  For any other maze, the
    sequence of moves will be incorrect, so only use this for tinyMaze.
    """
    from game import Directions
    s = Directions.SOUTH
    w = Directions.WEST
    return  [s, s, w, s, w, w, s, w]

def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print "Start:", problem.getStartState()
    print "Is the start a goal?", problem.isGoalState(problem.getStartState())
    print "Start's successors:", problem.getSuccessors(problem.getStartState())
    """
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()

def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()

def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
```

1. The file searchAgents.py includes an implemented SearchAgent that plans a path through the world and then executes the path step-by-step.
2. The search algorithms which this agent uses are not implemented: your job in this assignment to write them.
3. The algorithms are implemented in the file search.py (see above picture showing where to put code for depth first search).
4. First test the SearchAgent is working by running (you do not need to understand

in detail the code for this search agent and how it maps from the agent to the search function just notice the names are important):

   a. Python2 pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch

5. This command tells the search agent to use tinyMazeSearch as its search algorithm which is implemented in search.py. As you can see this implementation just provides a sequence of moves – open the file and look at the function definition, see: "def tinyMazeSearch(problem):". We will implement more general algorithms able to solve any maze.
6. 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).
7. Now implement Depth First Search to return a list of actions using the data structures provided in util.py (Stack, Queue and PriorityQueue – which one should you use for DFS???).
8. Your implementation should avoid expanding any already visited states!

IMPORTANT: 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.

Testing: your code should quickly find a solution for

```
Python2 pacman.py -l tinyMaze -p SearchAgent

Python2 pacman.py -l mediumMaze -p SearchAgent

Python2 pacman.py -l bigMaze -z .5 -p SearchAgent
```

Hint: when you use medium maze if you correctly use stack as your data structure to store nodes to explore next the solution should have a length of 130 if you push successors onto the fringe in the order provided by getSucessors.

Also notice in the visualisation the board has an overlay of states explored – bright red squares were explored earlier. Does the pacman go to all these squares on the way to the goal?

Question 2: Breadth First Search

1. Implement the breadth-first search (BFS) algorithm in the breadthFirstSearch function in search.py.
2. Write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

```
Python2 pacman.py -l mediumMaze -p SearchAgent -a
fn=bfs

Python2 pacman.py -l bigMaze -p SearchAgent -a fn=bfs
-z .5

(-z is a zoom – see python2 pacman.py -h)
```

Does BFS find a least cost solution (fewest moves in this case until we do the next question)? 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.

- Python2 eightpuzzle.py

Question 3: Varying the Cost Function

1. 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.
2. By changing the cost function, we can make Pacman find find different paths. For example, we can charge more for dangerous steps in areas with lots of ghosts or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.
3. Implement the uniform-cost graph search algorithm in the uniformCostSearch function in search.py. Please **look through util.py and select data structures (ie which of queue, stack, priority queue etc should you use?)** that may be useful in your implementation.
4. You should now observe successful behavior in all three of the following layouts (win the game), 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

1. 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.
2. 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). Check what happens on openMaze for the various search strategies.