PACMAN BFS SEARCH SOLUTION

Note - this is a lab - **not** a project so it's not required, but it should be done (to some extent) and discussed. There is some **ambiguity** around the instructions for the lab as it pertains to the score of 20. There are 8 questions plus extra credit which amount to 22 points, but that is not a "score", and it's a **lot** of work. I've queried the Udacity team for clarification, let us see what they come back with.

Overall Strategy:

This "Labs" difficulty comes not in understanding the DFS, BFS or A* search (code abounds on the Internet), but in the software engineering process of integrating that into an already standing infrastructure (PACMAN), understanding the data structures in play, the python to interrogate indiscriminate class instances, and figure out how to modify the DFS, BFS or A* search algorithms to handle complex nodes instead of the nice, clean, academic examples given in the videos.

Udacity has a habit of making work unnecessarily hard to keep up its reputation as a difficult MOO - but it doesn't add in the corresponding (and compelling) educational components to go along with it - it seems to be their stock and trace and we'll have to deal with it as a team moving forward.

In support - Some R&D:

Problem:
The problem description is horribly Spartan - with next to no information on how to determine what data structures are in play and if you aren't at a certain level in python acumen, you'd not be able to interrogate a random class instance, which is required to complete the integration process.

Research:

How to Implement Breadth-First Search in Python

https://pythoninwonderland.wordpress.com/2017/03/18/how-to-implement-breadth-first-search-in-python/

- This is a great source to obtain the code for BFS in python and experiment with it particularly as it pertains to goal searching. It's a great model to craft the BFS algorithm from.

Get all object attributes in Python?

https://stackoverflow.com/questions/6886493/get-all-object-attributes-in-python

How to get a complete list of object's methods and attributes?

https://stackoverflow.com/questions/191010/how-to-get-a-complete-list-of-objects-methods-and-attributes

- Since the code give you little to work with and a simple class instance to work with, you'll have to interrogate it to determine how to work with it. These links provide the necessary clues.

Running PACMAN and what to expect:

To run the BFS on various maze sizes, execute on the command line:

- **python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs**
- **python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs**
- **python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5**

The below are ways to interrogate the API, and the "problem" instance for information on what it contains and how to use it:

```
print "Start:", problem.getStartState()
print "Is the start a goal?", problem.isGoalState(problem.getStartState())
print "Start's successors:", problem.getSuccessors(problem.getStartState())
print "TYPE: ", type(problem)
print "DIR: ", dir(problem)
# print "VARS: ", vars(problem)
print "DICT: ", problem.__dict__
```

- **Start**: (5, 5)
- **Is the start a goal**? False
- **Start's successors**: **[((5, 4), 'South', 1), ((4, 5), 'West', 1)]**
- **TYPE**: <type 'instance'>
- **DIR**: ['__doc__', '__init__', '__module__', '_expanded', '_visited', '_visitedlist', 'costFn', 'getCostOfActions', 'getStartState', 'getSuccessors', 'goal', 'isGoalState', 'startState', 'walls']
- **DICT**: {'**costFn**': <function <lambda> at 0x0000000002A9DF98>, '**goal**': (1, 1), '**_visitedlist**': [(5, 5)], '**_expanded**': 1, '**_visited**': {(5, 5): True}, '**startState**': (5, 5), '**walls**': <game.Grid instance at 0x0000000002CAAC88>}

For instance, in the function definition of search.py:

```
"*** YOUR CODE HERE ***"
# print "Start:", problem.getStartState()
# print "Is the start a goal?", problem.isGoalState(problem.getStartState())
# print "Start's successors:", problem.getSuccessors(problem.getStartState())
# print "Type Starts Sccessors:", type(problem.getSuccessors(problem.getStartState()))
# print "TYPE: ", type(problem)
# print "DIR: ", dir(problem)
# print "VARS: ", vars(problem)
# print "DICT: ", problem.__dict__
# print "GOAL: ", problem.goal
# print "_visitedlist: ", problem._visitedlist
# print "_expanded: ", problem._expanded
# print "_visited: ", problem._visited
# print "startState: ", problem.startState
# print "TYPE-startState: ", type(problem.startState)
# print "DOC: ", problem.__doc__
# print "INIT: ", problem.__init__
# print "MODULE: ", problem.__module__
# print "COSTFN: ", problem.costFn
# print "Next successors:", problem.getSuccessors((5,4))
# print "\n****************"
# print "*** Start BFS **"
# print "****************\n"
```

Issue: - dealing with complex nodes

The PACMAN problem presents a node in the form of:

- **((5, 4), 'South', 1)**

Which is a "touple" of length 3, containing

- Position
- Direction
- Cost

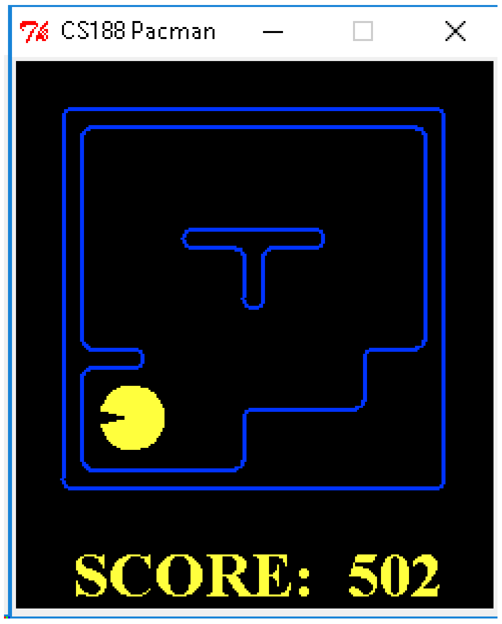
A Strategy: - (there are others)

- Construct a traditional BFS search and at all key interactions, keep an account of both the required graph attributes (path, node, frontier, explored, etc...) but also it's doppelganger of the full node instead of just the appropriate parts of it used to execute a graph based BFS.

SOLUTION CODE: - search.py

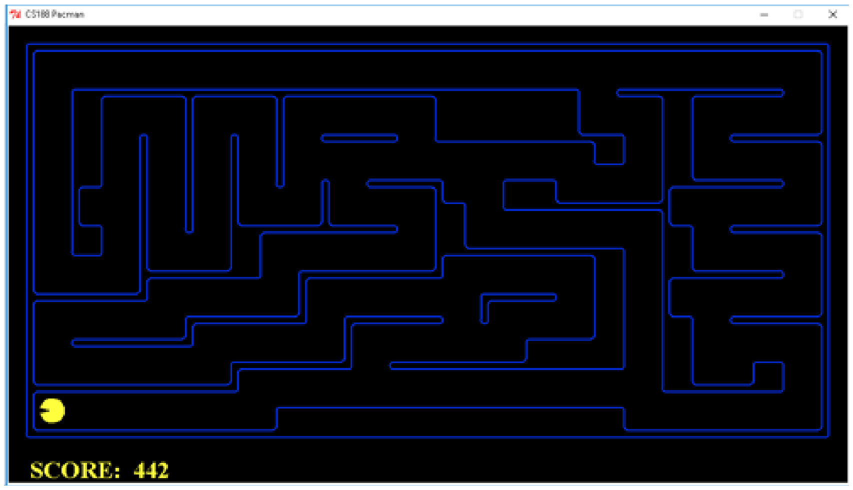
```
def breadthFirstSearch(problem):
  """
  Search the shallowest nodes in the search tree first.
  [2nd Edition: p 73, 3rd Edition: p 82]

  Your search algorithm needs to return a list of actions that reaches
  the goal.  Make sure to implement a graph search algorithm
  [2nd Edition: Fig. 3.18, 3rd Edition: Fig 3.7].

  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 ***"
  # print "Start:", problem.getStartState()
  # print "Is the start a goal?", problem.isGoalState(problem.getStartState())
  # print "Start's successors:", problem.getSuccessors(problem.getStartState())
  # print "Type Starts Sccessors:", type(problem.getSuccessors(problem.getStartState()))
  # print "TYPE: ", type(problem)
  # print "DIR: ", dir(problem)
  # print "VARS: ", vars(problem)
  # print "DICT: ", problem.__dict__
  # print "GOAL: ", problem.goal
  # print "_visitedlist: ", problem._visitedlist
  # print "_expanded: ", problem._expanded
  # print "_visited: ", problem._visited
  # print "startState: ", problem.startState
  # print "TYPE-startState: ", type(problem.startState)
  # print "DOC: ", problem.__doc__
  # print "INIT: ", problem.__init__
  # print "MODULE: ", problem.__module__
  # print "COSTFN: ", problem.costFn
  # print "Next successors:", problem.getSuccessors((5,4))
  # print "\n****************"
  # print "*** Start BFS **"
  # print "****************\n"

  ss_node = [problem.startState, 'NA', 0]          # Experiment - full node w/start state
  frontier = [[get_position(ss_node)]]             # populate the Frontier w/the start state position
  frontier_full_node = [[ss_node]]                 # populate full node frontier w/full node of start position
  explored = []                                     # initialize explored state
  path = []                                         # initialize Path
  path_full_node = []                               # intiialize full node path
  node = []                                         # initialize node
  node_full_node = []                               # initialize full node
  new_path_full_node = []                           # initialize new_path_full_node

  # keeps looping until all possible paths have been checked
  while frontier:
      # Check for empty frontier - no solution possible.
      if not frontier:
          raise Exception('No Solution Possible')
      # pop the first path from the frontier & full node frontier
      path = frontier.pop(0)
      path_full_node = frontier_full_node.pop(0)
      # get the last node from the path & full node path
      node = path[-1]
      node_full_node = path_full_node[-1]
      # Explore if node not yet explored
      if node not in explored:
          neighbours = problem.getSuccessors(node)
          # go through all neighbour nodes, construct a new path and push it into the frontier
          # (and full node frontier),along with their full node doppelgangers
          for neighbor in neighbours:
              new_path = list(path)
              new_path_full_node = list(path_full_node)
              new_path.append(neighbor[0])
              new_path_full_node.append(neighbor)
              frontier.append(new_path)
              frontier_full_node.append(new_path_full_node)
              # if you find the goal - return the path (and full node path)
              if neighbor[0] == problem.goal:
                  # print "\n** PATH ** \n", new_path, extract_sequence_of_moves(new_path_full_node)
                  # return the extracted sequence of moves to reach the goal per the utility function.
                  return extract_sequence_of_moves(new_path_full_node)
              # mark node as explored if it's not been explored before.
              if node not in explored:
                explored.append(node)
  # if we haven't returned prior, return now.
  return extract_sequence_of_moves(new_path_full_node)



def extract_sequence_of_moves(path):
    sequence_of_moves = []
    new_path = path[1:]
    for node in new_path:
        sequence_of_moves.append(node[1])
    return sequence_of_moves

def get_position(node):
    return node[0]

def get_direction(node):
    return node[1]
```

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



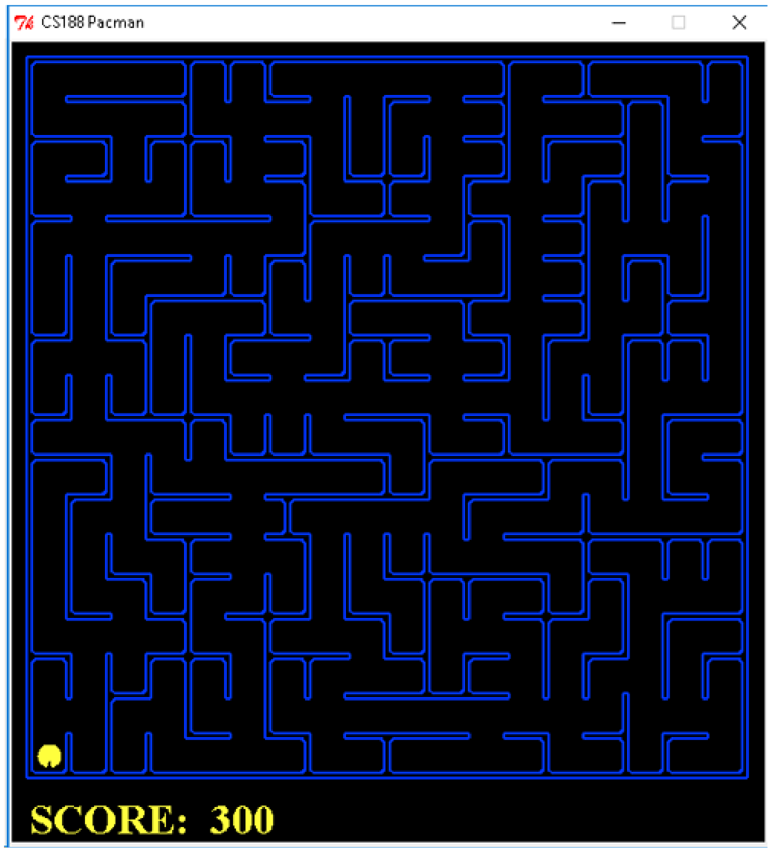
```
C:\Users\Matthew\workspaces\udacity\nano\artificial_intelligence\LAB_Search_PacMan\search>python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502
Win Rate:      1/1 (1.00)
Record:        Win
```

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



```
C:\Users\Matthew\workspaces\udacity\nano\artificial_intelligence\LAB_Search_PacMan\search>python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 267
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442
Win Rate:      1/1 (1.00)
Record:        Win
```

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



```
C:\Users\Matthew\workspaces\udacity\nano\artificial_intelligence\LAB_Search_PacMan\search>python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 617
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300
Win Rate:      1/1 (1.00)
Record:        Win
```