Answer the questions in the spaces provided on the question sheets. If you run out of room for an answer, continue on the back of the page.

Name: ____________________

Python Concepts

1. We will now review some basic list operations.[1]

   (a) Create a list called `lst` containing the following elements: $(3, 5), (2, 2), (1, 3), (0, 0)$.

   (b) Append the item (1,1) to the end of `lst`.

   (c) Return the first element (index 0) of the list into the variable `x`:

   (d) Remove *and* return the last item the list and store it in variable `y`.

   (e) At this point, what are the elements of `lst`?

   (f) Write Python code to determine if the element $(1, 3)$ is in the list `lst`.?

   (g) Write Python code to determine whether or not the element $(1, 5)$ is not in the list `lst`.

[1] `http://docs.python.org/tutorial/datastructures.html`

2. The three primary data structures in Python are dictionaries, tuples, and lists. We have seen how these primary structures can be used to implement more complex structures, such as the `numpy` array. We will now look at the `deque` data structure, found in the `collections` module.[2]

   (a) Import `deque` so that a double-ended queue can be created using `d = deque()`

   (b) Create a deque called `d` initialized using the list `[(0,1), (1,0)]` (incidentally, these points also constitute basis vectors such that $a \begin{bmatrix} 0 \\ 1 \end{bmatrix} + b \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$).

   (c) Append the element $(-1, -1)$ to `d`.

   (d) Append the item (-1,-1) to the beginning of `d`.

   (e) Remove *and* return the last item from the deque and store it in variable `x`.

   (f) Remove *and* return the first item from the deque and store it in variable `y`.

   (g) What are the advantages of using a `deque` over a general purpose Python `list`? What are the disadvantages?

[2] `http://docs.python.org/library/collections.html#collections.deque`

Stacks and Queues

3. Now that we have a `deque`, let us implement stacks and queues. For this problem, assume that the "front" or the "top" of the data structure is at index 0 of the `deque`.
   (a) Given a non-empty `deque` called `s`, implement the push, pop, and empty operations in terms of the `dequeue` API.

   (b) Given a non-empty `deque` called `q`, implement the enqueue, dequeue, and empty operations in terms of the `dequeue` API.

   (c) Given a stack in which we push the following sequentially: $1, 2, 3, 4, 5$, pop the elements and list the order in which they are popped (that is, using `popleft`). A stack is a LIFO data structure.

   (d) Given a queue in which we enqueue the following sequentially: $1, 2, 3, 4, 5$, dequeue the elements and list the order in which they are popped (that is, using `popleft`). A queue is a FIFO data structure.

Static Paths

4. Rotate is a useful operation for game agents that have static cyclical paths, for example, a patrol route that a guard can follow blindly. These agents are simple to implement, but can also be easily fooled. Still, they are inexpensive to implement.
   (a) Create a `deque` called `path` containing the tiles: $(0, 0), (0, 1), (1, 1), (0, 0)$. Assume that the guard executes the path from the first element to the last element.

(b) Draw the static route using an x-y plot.

(c) Write Python code to obtain the first tile.

(d) Write Python code to obtain the last tile. **Hint:** Use array slicing tricks.

(e) Write Python code to rotate the path 1 to the left. What is the new `path`?

Tile Graphs

5. In this class, we will use tiles as the basic building blocks of level design. Tiles help simplify the mathematical operations because every tile is assumed to be the same size. For this exercise, we will use `'.'` for a valid tile and `'#'` for an invalid tile (such as a wall).

   (a) In Python, create a two-dimensional 2x2 tile representation of a level (called `level1`) containing no obstacles.

   (b) In Python, create a two-dimensional 5x5 tile representation of a level (called `level2`) that contains obstacles at $(1,2),(2,3)$, and $(1,4)$.

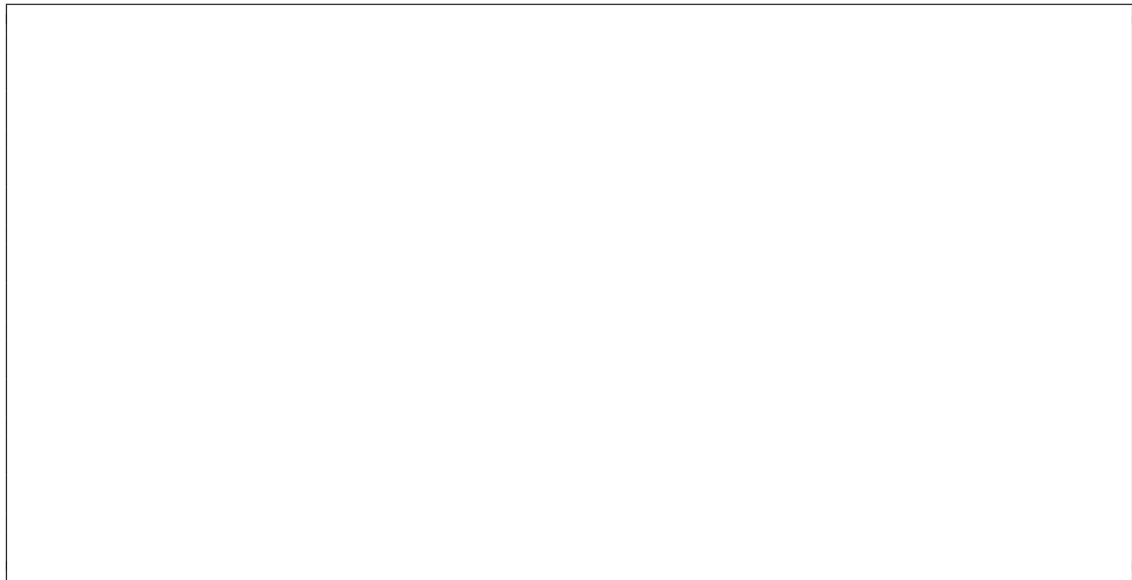
(c) If a tile is actually 100 (`TILE_WIDTH`) by 100 (`TILE_HEIGHT`) pixels, write Python code to determine the (x, y) of the tile itself if the avatar has the property `a.position`.

(d) Write Python code to access the tile at $(2, 3)$ for `level`. This problem is trickier than it appears.

(e) Write Python code to determine how many rows the level has; store this result in `rows`.

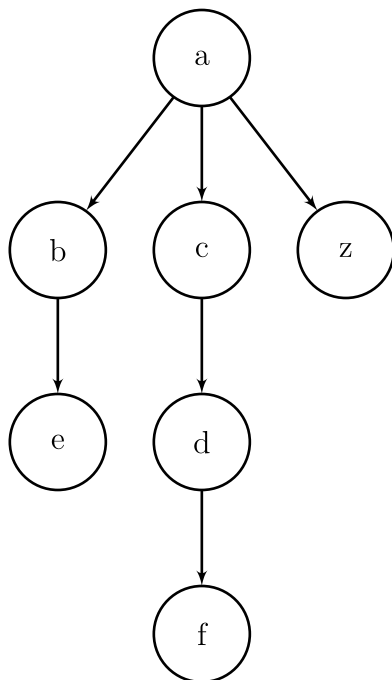
(f) Write Python code to determine how many columns the level has; store this result in `cols`. Assume that a level must have at least 1 tile, and therefore at least one row (that is, levels cannot be completely empty).

(g) Write a function `get_edges(level, location)` that returns a list of position tuples for all the valid moves from `location`. Assume that the agent cannot move diagonally or past the boundaries of the level.

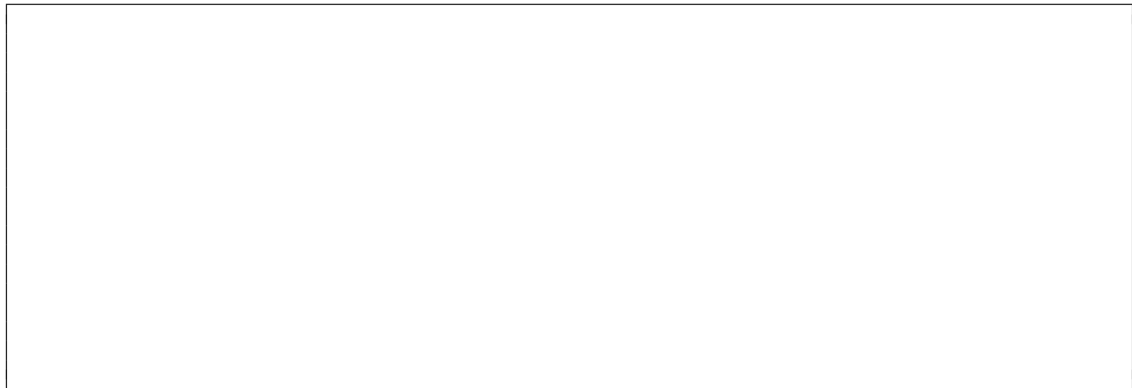


Breadth-first Search

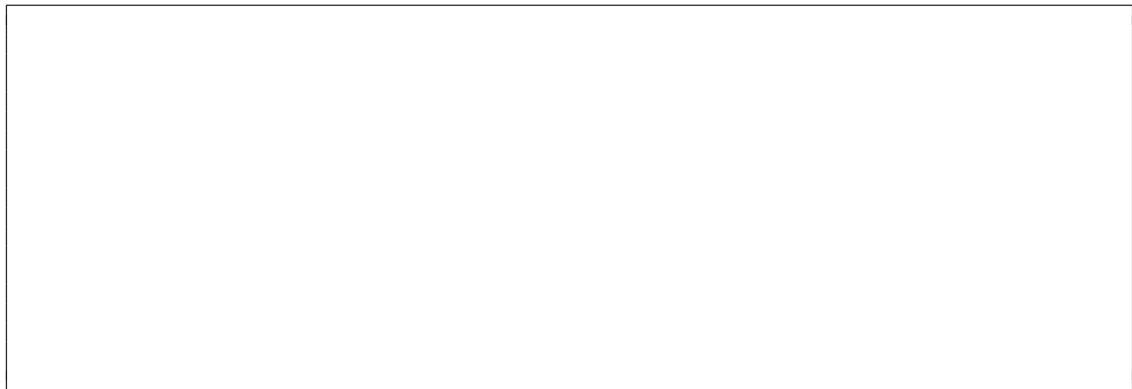
6. In this section we will hand-compute the algorithm for breadth-first search. This is an uninformed (unit-cost) search algorithm because it ignores the edge costs. With this understanding, we will then be able to implement the algorithm in Python. Given the following directed acyclic graph (also known as a DAG), demonstate the BFS algorithm. Assume that edges are returned in a list in alphabetical order.



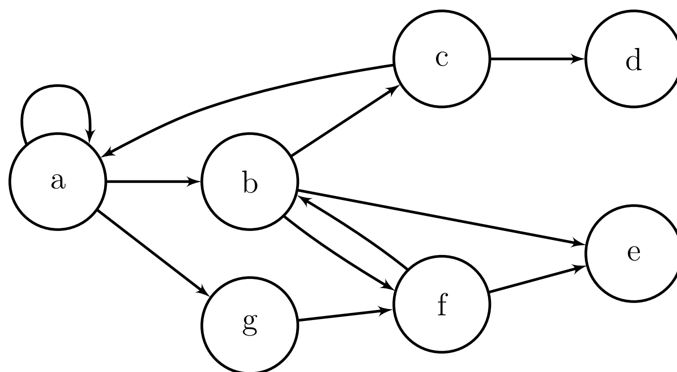
   (a) With a start state of a and a goal state of e, show the executation trace of the BFS algorithm using the explored list and the frontier list.



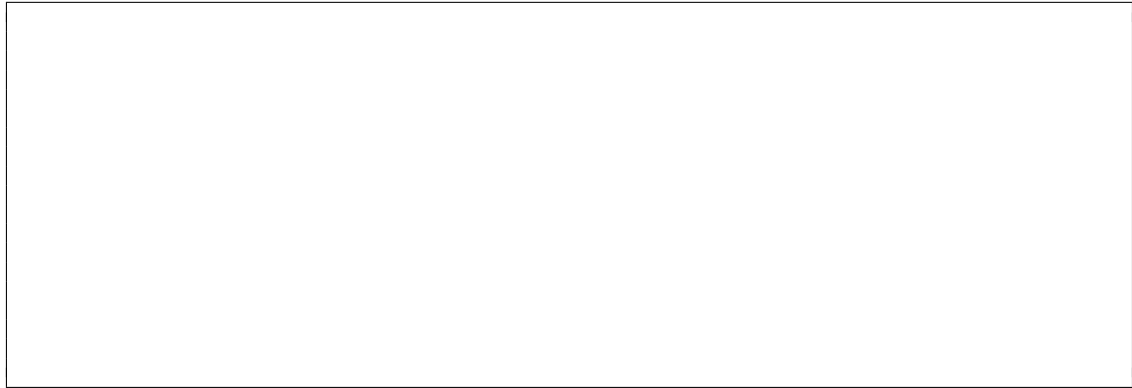
(b) With a start state of a and a goal state of f, show the executation trace of the BFS algorithm using the explored list and the frontier list.



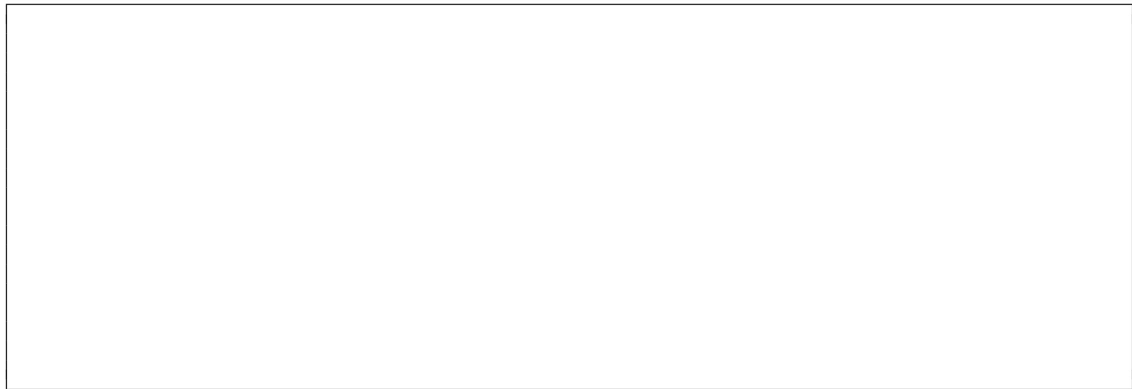
7. Given the following graph:



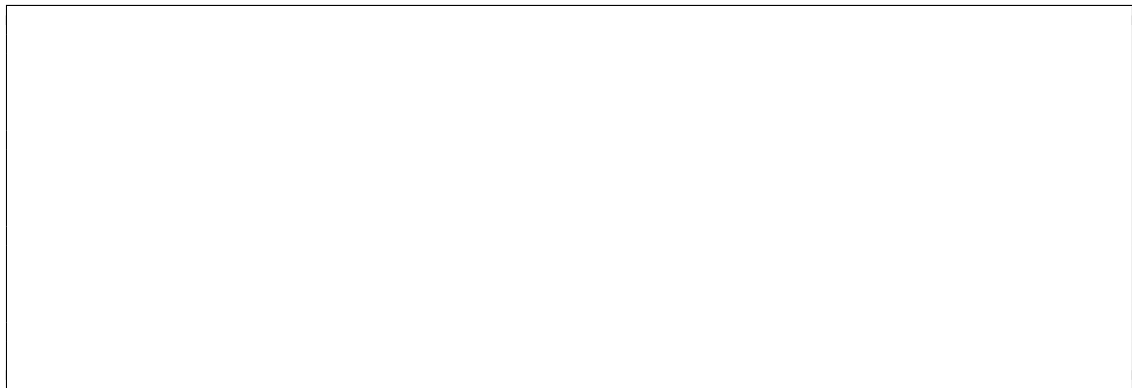
(a) With a start state of a and a goal state of f, show the executation trace of the BFS algorithm using the explored list and the frontier list.



(b) With a start state of c and a goal state of e, show the executation trace of the BFS algorithm using the explored list and the frontier list.



(c) Generate the graph representation of the a 2x2 tile-based level.



Breadth-first Search in Python

8. Write a function `bfs(level, start, goal)` that implements the breath-first search algorithm as discussed in class. To test this function, you may want to use something like:

```
level1 = [['.','.','.','.','.'],
```

```
      ['.','.','.','.','.'],
      ['.','.','.','.','.']
    ]
```

and

```
print bfs(level1, (0,0), (1,1))
```

after which you can introduce additional obstacles.