CMPS 6100 Lab 07

In this lab, you implement and use graph algorithms.

Some prompts will require you to edit `main.py` and others will require answers will go in `answers.md`.

Refer back to the README.md for instruction on git, how to test your code, and how to submit properly to get all the points you've earned.

Binary Search Tree (6 pts)

Binary Search Tree code is given to you in `bst.py`.

1. Implement `traverse_preorder` which takes in a list and adds each element of the tree to the list in the order that they are visited. Ensure that `test_traverse_preorder_1` and `test_traverse_preorder_2` pass.
2. Implement `traverse_inorder` which takes in a list and adds each element of the tree to the list in the order that they are visited. Ensure that `test_traverse_inorder` and `test_traverse_inorder_random` pass.
3. Implement `traverse_postorder` which takes in a list and adds each element of the tree to the list in the order that they are visited. Ensure that `test_traverse_postorder_1` and `test_traverse_postorder_2` pass.

Unit Tests

As for last lab, all unit tests for this lab's implementations are provided in `unit_tests.py`.

Unit Tests are provided for the entire BST class, not just the part you are implementing.

Queue (4 pts)

In the next session, you will implement a version of breadth first search. As you may recall BFS uses a Queue to determine the order in which to traverse the Graph. In our notes, we used python's built in `deque`. In this lab, you will implement your own Queue.

You will again use your `LinkedList` as the foundation of your `Queue`. Copy your linked list implementations from Lab-06 over into lab-07's `linked_list.py`.

4. Implement `push`, `poll`, and `peek` in `my_queue.py`, and ensure that `test_queue` passes.

Unit Tests

Since you are again using your `LinkedList`, the tests for it have been provided for you in `linked_list_tests.py`. None of the points for this lab come from these tests. They are provided for your convenience only.

`test_queue` is in `unit_tests.py`.

Breadth First Seach and Depth First Search (7 pts)

5. In the Graph Exploration notebook, an implementation for Breadth First Search was given. Breadth First Search is a graph traversal algorithm in which all vertices 1 step away from the source are visited, then all vertices 2 steps away are visited and so on. We say that a vertex a that is adjacent to s is one step away and thus is a distance of one away from s. It is possible to modify Breadth to keep track of vertex distances.

Implement `bfs_distances` which takes in a `graph` and a `source` vertex and returns a dictionary whose keys are vertices and values are the distance of that vertex from the source vertex. The source vertex s is distance 0 from itself.

Ensure that the tests pass (2 pts ea.):

- `test_bfs_distances_simple`
- `test_bfs_distances_triangle`
- `test_bfs_distances_full`

6. Suppose that you are running Depth First Search (DFS) on a tree. Which of the tree traversals would visit the elements of the tree in a valid order for DFS, that is, in the same order that DFS could? (1 pt)

   Add your answer to `answers.md`.

Connected Components (8 pts)

7. We can use either of our graph search algorithms to discover the connected components of a graph.

   Implement `connected_components` which takes in a graph and returns a list of sets where each set contains the vertices in one of the connected components of the graph.

   Ensure that the tests pass (2 pts ea.):

   - `test_connected_components_one`
   - `test_connected_components_two`
   - `test_connected_components_three`

8. Implement `num_connected_components` which takes in a graph and returns the number of connected components in that graph. Ensure that `test_num_connected_components` passes (2 pts).