**Homework 11**

**ECE 309 Fall 2019**

**Due: November 18, 2019**

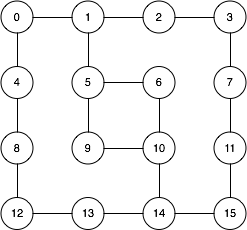
Upload an electronic copy of your answers to Moodle under HW11.

*This is a shared google document. This means (1) it may change to clarify content, and (2) other people can view your comments on this file. If you have questions, you are encouraged to comment directly on this document, but* ***do not add your answers here****. Make a copy into your private Google Drive and then edit the document.*

*DO NOT ADD ANSWERS TO THE SHARED DOC! THAT’S CONSIDERED CHEATING!*

# 1. Graph Traversal

For the following graph, show the order of the following traversals. If there are multiple adjacent edges to choose from, choose the one with the lowest number first.



1. [15 points] A breadth-first search starting at 10.
2. [15 points] A depth-first search starting at 10.
3. [15 points] A depth-first search starting at 15.

# 2. Finding a Path

(a) [20 points] Show pseudo-code for an algorithm that finds a path between two nodes. The algorithm must take a start node and an end node, and the algorithm returns a list showing the sequence of adjacent nodes to pass through from start to end -- in other words, return the path from start to end.

Hint: use a depth-first search, that begins at the start node and stops when the end node is reached.

(b) [10 points] Given an example of how your pseudo-code works on a graph. [Note, it should illustrate the changes you make to the search algorithm.]

# 3. ZyLabs

* [25 points] ZyLab 19.14. Implement a recursive depth-first search using the definition for a Graph node below.
  1. [15 points] for passing the tests on ZyLabs.
  2. [10 points] for implementing a proper recursive function.

|  |
| --- |
| // DO NOT MODIFY GRAPH NODE DEFINITION  class Graph {  // This class represents a single node    public:  std::string name; // name of the node  list<Graph\*> adjList; // adjacency list for the node  using iterator = list<Graph\*>;  // constructor requires a name, so that we can print out the  // nodes by name.  Graph(std::string n, list<Graph\*> adj={}):name(n),adjList(adj) {}  // Add an edge to adjacency list; add to both nodes  void addEdge(Graph \*g)  {  adjList.push\_back(g);  g->adjList.push\_back(this);  }  };  // DO NOT MODIFY FUNCTION ARGUMENTS OR RETURN TYPE  // dfs\_search:  // Graph \*g : a pointer to the node where you should begin DFS  // list<Graph\*> &dfs : list of visited nodes; append to this list  // as you traverse. Use push\_back() member function.  // unordered\_set<Graph\*> &visited:  // this is basically a hash table provided by the std library.  // Use insert(g) member function to add to it. Use find()  // member function to search if a node has been visited.  //  // You must implement this as a recursive function.  void dfs\_search(Graph \*g, std::list<Graph\*> &dfs, std::unordered\_set<Graph\*> &visited)  {  // IMPLEMENT AS RECURSIVE FUNCTION  } |