Name: _____

Section: _____

Github URL: _____

Name of ALL collaborators: _____

URLs/ISBNs for ALL consulted websites/textbooks: _____

(cont:) _____

CS 435 – Project 2: Graphs

Due Dates:

Parts 1-3 due 11:59pm, March 30th.
Parts 4-7 due 11:59pm, April 6th.
Part 8 due 11:59pm, April 13th.
Part 9 due 11:59pm, April 20th.

For this project, you will be responsible for uploading all code in a Github repository. Please print this and turn in all written answers here. **Do not turn in written code here! Code must be submitted via a link to a Github repository!**

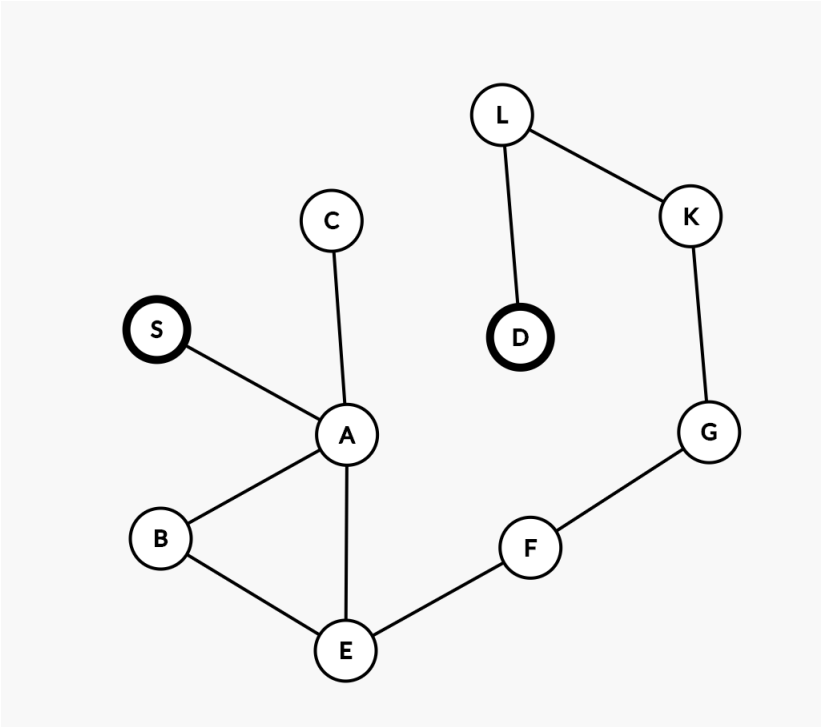
You will be required to give 3 peers a high-quality code review for full credit. Please refer to the slides from the code review conversation to determine what types of high quality comments to leave.

Note: For this project, you may use whatever programming language you want. However, we will implement this project in Java, Python, and C++, so using one of those languages would make it significantly easier to get help from Sresht and the TAs.

Part:	1	2	3	4	5	6	7	8	9	Total
Points:	5	7	30	15	20	18	0	20	15	115
Score:										

1. **Gonna Take My Horse To The Old Town Node (5 points)**

   (a) (1 point) In *Fig. 1*, what would be a possible order in which Breadth-First Search (BFS) would visit nodes when trying to find the Destination node (D) from Source node (S)? Note that there are multiple correct answers.

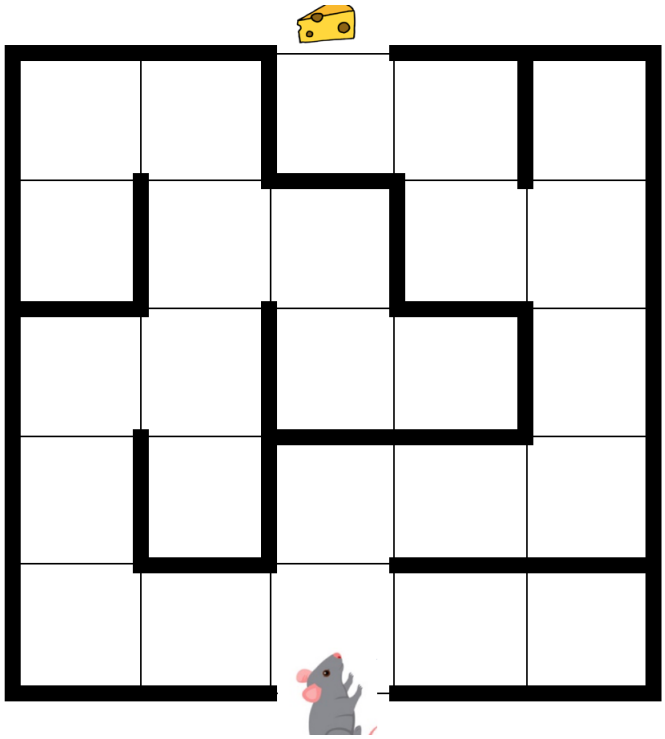


(b) (2 points) In code, how would we represent the **edges** between the graph on the previous page? Please write out either an adjacency list or matrix to represent this graph's edges.

(c) (2 points) Draw a graph in which Depth-First Search (DFS) would likely visit fewer nodes than Breadth-First Search (BFS). Label your starting node S and your destination node D.

2. **Boulevard of Broken Cheese (7 points)**

You are Jerry the precocious mouse. As part of a scientific experiment, you find yourself at the start of a maze (see below). As an expert in graph algorithms, you've heard from your professor that everything can be converted into a graph. So you're now interested in converting this maze into a graph!



Grid lines are provided for your convenience.

(a) (1 point) If you were to convert this maze into a graph, how many nodes would you have?

(b) (1 point) What would your edges represent in the graph?

(c) (2 points) What are properties of your graph? Please refer to the properties discussed in the "Intro to Graphs" lecture.

(d) (3 points) Draw a graph representation of this maze based on your reasoning above.

3. **Traverse This Town (30 points)**
(You must submit code for all parts in this question!) The scientists are incredibly amazed at your ability to solve the small maze they gave you! So they are now asking you to create and solve your own random mazes, and they want to see if you can do it efficiently.

   (a) (5 points) *(You must submit code for this question!)* Write a class `Graph` that supports the following methods:

      i. `void addNode(final String nodeVal)` - This adds a new node to the graph.

      ii. `void addUndirectedEdge(final Node first, final Node second)` - This adds an undirected edge between `first` and `second` (and vice versa).

      iii. `void removeUndirectedEdge(final Node first, final Node second)` - This removes an undirected edge between `first` and `second` (and vice versa).

      iv. `HashSet<Node> getAllNodes()` - This returns a set of all `Node`s in the graph.

   (b) (2 points) *(You must submit code for this question!)* In your `Main` class, create a non-recursive method called `Graph createRandomUnweightedGraphIter(int n)` that creates n random nodes with randomly assigned unweighted, bidirectional edges. You should use some of the methods you implemented in part (a). Make sure you're either implementing an adjacency list or an adjacency matrix to keep track of your edges!

   (c) (2 points) *(You must submit code for this question!)* In your `Main` class, create a non-recursive method called `Graph createLinkedList(int n)` that creates a `Graph` with n nodes where each node only has an edge to the next node created. For example, if you create nodes 1, 2, and 3, Node 1 only has an edge to Node 2, and Node 2 only has an edge to Node 3.

   (d) (3 points) *(You must submit code for this question!)* In a class called `GraphSearch`, implement `ArrayList<Node> DFSRec(final Node start, final Node end)`, which recursively returns an `ArrayList` of the `Node`s in the `Graph` in a valid Depth-First Search order. The first node in the array should be `start` and the last should be `end`. If no valid DFS path goes from `start` to `end`, return `null`. **EDIT: This should NOT use `Graph` as an input! The only exception to this if you need your Graph instance to call some API like `getNeighborNodes(final Node node)`. Other than this, you should not be using a `Graph` parameter for anything else**

   (e) (5 points) *(You must submit code for this question!)* In your `GraphSearch` class, implement `ArrayList<Node> DFSIter(final Node start, final Node end)`, which iteratively returns an `ArrayList` of the `Node`s in the `Graph` in a valid Depth-First Search order. The first node in the array should be `start` and the last should be `end`. If no valid DFS path goes from `start` to `end`, return `null`. **EDIT: This**

should NOT use `Graph` as an input! The only exception to this if you need your Graph instance to call some API like `getNeighborNodes(final Node node)`. Other than this, you should not be using a `Graph` parameter for anything else.

(f) (3 points) *(You must submit code for this question!)* In your `GraphSearch` class, implement `ArrayList<Node> BFTRec(final Graph graph)`, which recursively returns an `ArrayList` of the `Node`s in the `Graph` in a valid Breadth-First Traversal order.

(g) (5 points) *(You must submit code for this question!)* In your `GraphSearch` class, implement `ArrayList<Node> BFTIter(final Graph graph)`, which iteratively returns an `ArrayList` of **all** of the `Node`s in the `Graph` in a valid Breadth-First **Traversal**.

(h) (3 points) *(You may submit a screenshot for this question, but you're not required to.)* Using the methods above in `GraphSearch`, in your `Main` class, implement `ArrayList<Node> BFTRecLinkedList(final Graph graph)`. This should run a BFT recursively on a `LinkedList`. Your `LinkedList` should have 10,000 nodes.

i. Did you run into any issues with this? If so, try running it on a `LinkedList` of size 100 and see if it works. If it does, what might cause your function to work on size 100 but not size 10,000? Use asymptotic complexity to justify your answer. If you didn't run into issues, please explain why someone might run into issues with this. Use asymptotic complexity to justify your answer.

(i) (2 points) Using the methods above in `GraphSearch`, in your `Main` class, implement `ArrayList<Node> BFTIterLinkedList(final Graph graph)`. This should run a BFT **iteratively** on a `LinkedList`. Your `LinkedList` should have 10,000 nodes.

i. Why should this code not result in issues, but the recursive function might?

4. **Thank U, Vertext (15 points)**

Wow, you've really outdone yourself this time! You're so good at solving basic mazes that the scientists want you to try a new challenge. This time, there are multiple changes:

- **You can only move forward or right**
- **There is now cheese on every single square, not just at the end**
- **There could be multiple, unconnected parts to the maze, each with its own entrance (and possibly their own exits). In other words, the maze is no longer a connected graph**

As a genius in graph algorithms (and quite a greedy little mouse), you have decided that you're going to handle this by making a DAG, solving it using a topological sort, and gobbling up all the cheese along the way!

(a) (1 point) Describe in words the properties of a DAG. Describe in words what will change about the edges and nodes in your new graph as compared to your previous graph.

(b) (3 points) *(You must submit code for this question!)* Write a class `DirectedGraph` that supports the following methods. You may use similar code as `Graph` above (or better yet, use an Interface to group these classes together).

EDIT: Note: You do not need to worry (yet) about a node only having two neighbors. If you do that here, that's fine, but that's not necessary yet. A node can have as many neighbors as it wants, as long as they're all directed (for example, a node could have a 85 degree forward neighbor, a 70 degree forward neighbor, and a 50 degree forward neighbor or something).

i. `void addNode(final String nodeVal)` - This adds a new node to the graph.

ii. `void addDirectedEdge(final Node first, final Node second)` - This adds a directed edge between `first` and `second` (but not vice versa).

iii. `void removeDirectedEdge(final Node first, final Node second)` - This removes an directed edge between `first` and `second` (but not vice versa).

iv. `HashSet<Node> getAllNodes()` - This returns a set of all `Node`s in the graph.

(c) (3 points) *(You must submit code for this question!)* In your `Main` class, create a non-recursive method called `DirectedGraph createRandomDAGIter(final int n)` that creates n random nodes with randomly assigned unweighted, directed edges. You should use some of the methods you implemented in part (a) of this question.

Make sure you're either implementing an adjacency list or an adjacency matrix to keep track of your edges, and keeping track of directionality!

(d) (4 points) *(You must submit code for this question!)* In a class called `TopSort`, implement `ArrayList<Node> Kahns(final DirectedGraph graph)`. This should do a valid toplogical sort of the graph using Kahn's algorithm. **EDIT: Call this method on** `createRandomDAGIter(1000).`

(e) (4 points) *(You must submit code for this question!)* In a class called `TopSort`, implement `ArrayList<Node> mDFS(final DirectedGraph graph)`. This should do a valid toplogical sort of the graph using the mDFS algorithm. **EDIT: Call this method on** `createRandomDAGIter(1000).`

5. **Uno, Do', Tre', Cuatro, I Node You Want Me (20 points)**

   As another challenge, the scientists have decided to put treadmills between different cells in the maze, which is now connected again (with only one start and one end). So to move from one cell to another is not the same amount of effort, nor is it a symmetrical relationship! As a lazy mouse, you decide that you want to get to the end of the maze with as little effort as possible. Luckily, you've attended CS435 lectures, so you decide to implement Dijkstra's Algorithm to help you do exactly that!

   (a) (1 point) What properties of the graph make it possible for you to use Dijkstra's on this graph?

   (b) (3 points) *(You must submit code for this question!)* Write a class `WeightedGraph` that supports the following methods. You may use similar code as `Graph` or `DirectedGraph` above (or better yet, use an Interface to group these classes together).

      i. `void addNode(final String nodeVal)` - This adds a new node to the graph.

      ii. `void addWeightedEdge(final Node first, final Node second, final int edgeWeight)` - This adds a directed, weighted edge between `first` and `second` (but not vice versa).

      iii. `void removeDirectedEdge(final Node first, final Node second)` - This removes an directed, weighted edge between `first` and `second` (but not vice versa).

      iv. `HashSet<Node> getAllNodes()` - This returns a set of all `Node`s in the graph.

   (c) (4 points) *(You must submit code for this question!)* In your `Main` class, create a non-recursive method called `WeightedGraph createRandomCompleteWeightedGraph(final int n)`. This should make a complete weighted graph, which means that each node has a randomly weighted positive integer edge to every other edge in the graph. Make sure you're either implementing an adjacency list or an adjacency matrix to keep track of your weighted edges, and keeping track of directionality! **EDIT:** You should have a total of $n \cdot (n-1)$ directed edges, each with a random edge weight to each other node.

   (d) (2 points) *(You must submit code for this question!)* In your `Main` class, create a non-recursive method called `WeightedGraph createLinkedList(final int n)`. This should make a weighted graph with n nodes, each having a single edge to the next node of uniform weight (perhaps weight 1). This can look very similar to the method you implemented in part 3c. Make sure you're either implementing an adjacency list or an adjacency matrix to keep track of your edges, and keeping track of directionality!

   (e) (10 points) *(You must submit code for this question!)* **EDIT: Changed this to add it to your Main class.** In your `Main` class, implement `HashMap<Node,`

`Integer> dijkstras(final Node start)`. This should return a dictionary mapping each Node node in the graph to the minimum value from start to get to node.

EDIT: This should NOT use `Graph` as an input! The only exception to this if you need your Graph instance to call some API like `getNeighborNodes(final Node node)`. Other than this, you should not be using a `Graph` parameter for anything else.

6. **When You Wish Upon A* (23 points)**

EDIT: Completely changed this problem. Please do this version, NOT the old version!!!

You now have one final maze that you want to solve using Graph algorithms. This time, your movement is constrained specifically to left, right, down, and up. In other words, **a node may have at maximum four neighbors.** Luckily, you know that you can use A* on this maze to ensure that you are finding the optimal solution in the fastest time possible!

(a) (5 points) *You must submit code for this question!* Write a class `GridGraph` that supports the following functions:

i. `void addGridNode(final int x, final int y, final String nodeVal)` - This adds a new GridNode to the graph with coordinates x and y. A GridNode keeps track of its value as well as its (x,y) coordinates.

ii. `void addUndirectedEdge(final GridNode first, final GridNode second)` - This adds an undirected, unweighted edge between `first` and `second` (and vice versa) **if first and second are neighbors**. If they are not neighbors, do nothing.

iii. `void removeUndirectedEdge(final GridNode first, final GridNode second)` - This removes an undirected edge between `first` and `second` (and vice versa).

iv. `HashSet<GridNode> getAllNodes()` - This returns a set of all `GridNode`s in the graph.

(b) (5 points) *(You must submit code for this question!)* In your `Main` class, create a non-recursive method called `GridGraph createRandomGridGraph(int n)` that creates n^2 random nodes with randomly assigned unweighted, bidirectional edges. These nodes should be (0,0), (1,0), (2,0),..., (n,0),(0,1),(1,1),...(n,1),...,(n,n). Nodes should have a 50% chance of being connected to their neighbors, and a 0% chance of being connected to non-neighbors. For example, the Node at (1,1) should have a 50% chance of being connected to the node (0,1), 50% to (1,0), 50% to (2,1), and 50% to (1,2), but a 0% chance of being connected to (2,2).

(c) (3 points) You want to find the optimal path from the start node (0,0) to the end node (n,n) using the A* algorithm. What is an admissible and consistent heuristic that you can use to help you solve the maze using A*? Justify why it's both admissible and consistent.

(d) (10 points) *(You must submit code for this question!)* In `Main`, call createRandomGridGraph(100) and store `sourceNode` as the node at (0,0) and `destNode` as the node at (100,100). implement `ArrayList<Node> astar(final sourceNode,`

`final destNode`). Ensure that you are using the heuristic you established in part (e). This should return an ordered list, from sourceNode and ending at destNode.

7. **Edgextra Credit (2 points extra credit)**

   (a) *(You must submit code for this question to receive extra credit!)* **EDIT: Generalized the requirements for this.** Determine how many nodes are finalized in your implement of Dijkstra's and A* in parts 5 and 6. Is there a noticeable difference in those numbers? Note that because these are technically different random graphs, it's not doing a 1:1 comparison, but it may still help you visualize what the differences in the algorithm may look like.

8. **Code Review Resubmission (20 points + 5 points extra credit)**

   (a) (20 points) **EDIT: further clarification** After submitting your code, you must review at least three students' code by April 13^{th} and add comments on how to improve it. You don't need to review all parts of their code, but you must provide at least 10 comments leaving actionable technical feedback to get full credit on this part. If you are not the first reviewer on their project, please try to review a different piece of their code than the reviewers before you (but if the first reviewer reviewed all of their code, then feel free to review the same block of code). *Leaving even a single rude, mean, or counterproductive comment will earn you 0 points on this part, and possibly the entire project.* Comments such as "Everything looks good" are not eligible for any points.

   (b) (5 points extra credit) Review at least eight students' code total for extra credit.

9. **Code Review Resubmission (15 points)**

   (a) After submitting your code, at least three students will be individually reading and reviewing your code. Comments will be added to your code. At some point before April 20^{th}, it is your responsibility to amend your code and push up a new commit to the same repository. If you sufficiently address all points made in code review, you will receive full credit on this part of the project.