Name: David Kotaev

Section: 2

Github URL: https://github.com/flexadecimal/cs435-project2

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!**

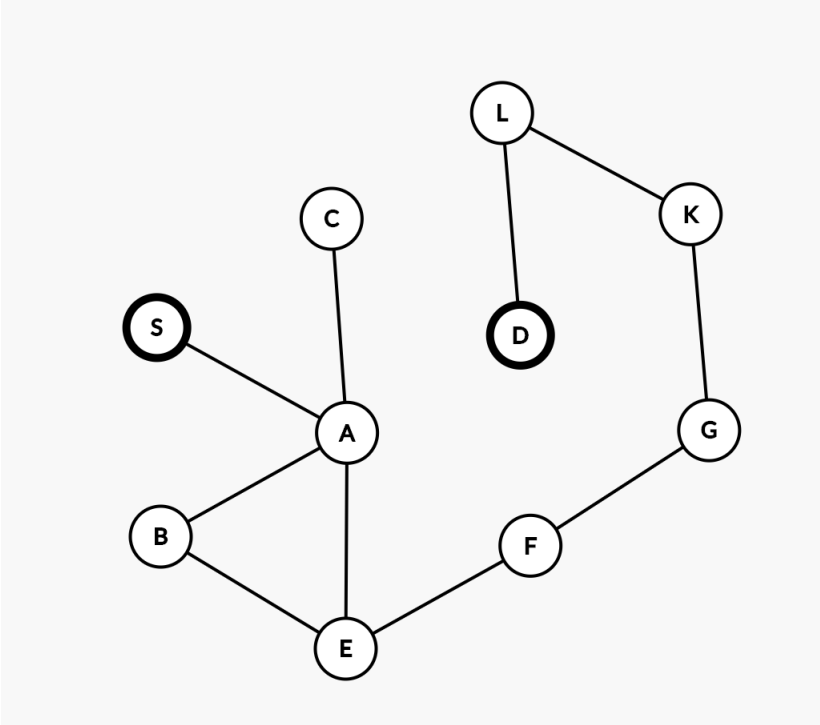
You will be required to give 3 peers a high-quality code review for full credit. Please refer to the slides from the code review conversation to determine what types of high quality comments to leave.

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.

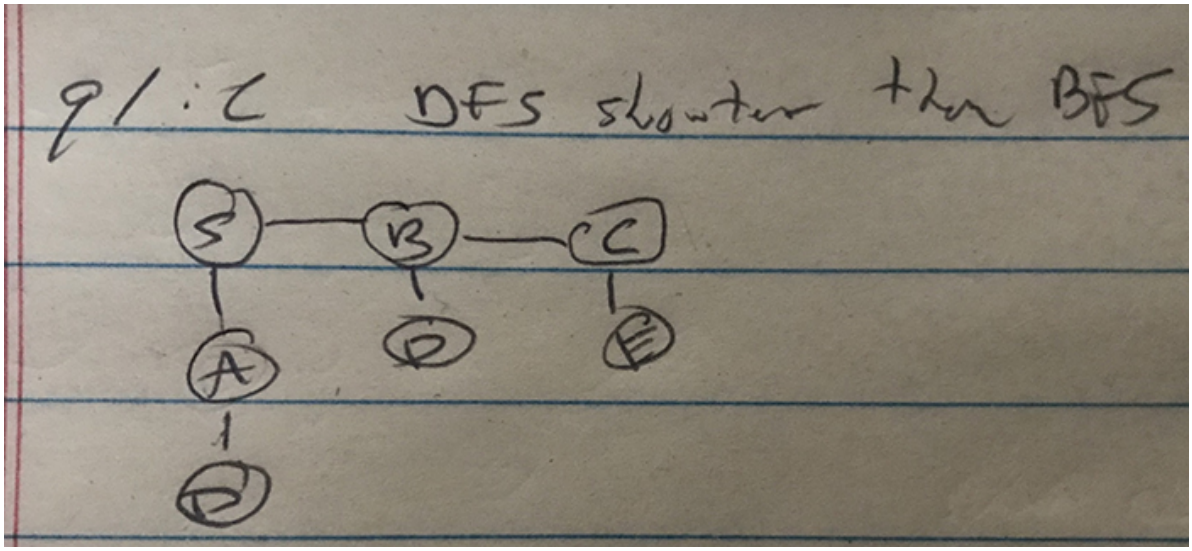


S, A, C, B, E, F, G, K, L, D

(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.

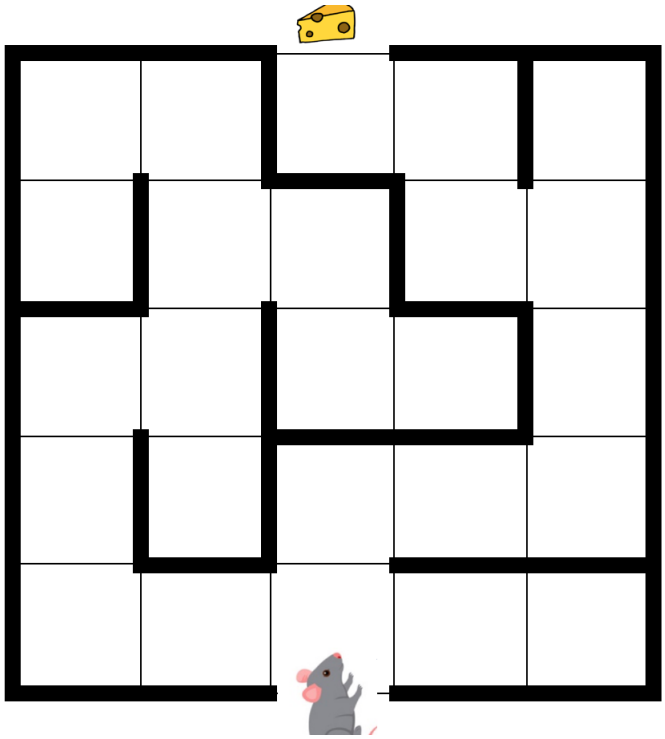
(c) (2 points) Draw a graph in which Depth-First Search (DFS) would likely visit fewer nodes than Breadth-First Search (BFS). Label your starting node S and your destination node D.

S: [A]
C: [A]
A: [S, C, B, E]
B: [A, E]
E: [A, F]
F: [E, G]
G: [F, K]
K: [G, L]
L: [K, D]
D: [L]



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!



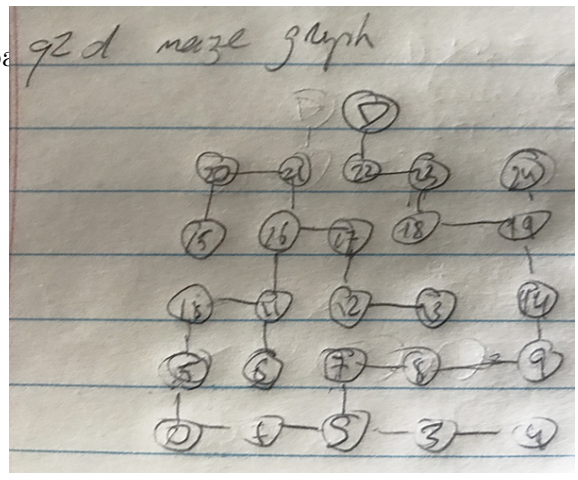
Grid lines are provided for your convenience.

(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 ba



a. As a graph, the nodes would be places in the maze to go, including cheese destination: 26 nodes
b. The edges represent valid movements you can make: up, down, left, right.
c. The graph is undirected - you can move back and forth between as you would if backtracking. The graph is connected, because you have to go through the maze to get to any other location. It is also unweighted and acyclic - there are no 'loops' in the maze.

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`.

   (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`.

   (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).

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.

(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.

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)` - 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!

   (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!)* In a class called `TreadmillMazeSolver`, implement `ArrayList<Node> dijkstras(final WeightedGraph graph)`.

6. **When You Wish Upon A* (18 points)**

   You now want to compare whether you can go even faster by using a greedy path-finding algorithm!

   (a) (3 points) What is an admissable and consistent heuristic that you can use to help you solve the maze using A*? Justify why it's both admissable and consistent.

   (b) (10 points) *(You must submit code for this question!)* In `TreadmillMazeSolver`, implement `ArrayList<Node> astar(final WeightedGraph graph)`. Ensure that you are using the heuristic you established in part (a).

   (c) (5 points) *(You must submit code for this question!)* In your `Main` class, run `dijkstras(createRandomCompleteWeightedGraph(10,000)` and `astar(createRandomCompleteWeightedGraph(10,000)`. Do they both return the same path?

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

   (a) *(You must submit code for this question to receive extra credit!)* In your `Main` class, run `dijkstras(createRandomCompleteWeightedGraph(10,000)` and `astar(createRandomCompleteWeightedGraph(10,000)`. Add time packages in `Java` to determine how fast each runs (in milliseconds/picoseconds). Is there a noticeable difference in the time it takes to run each?

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

   (a) (20 points) 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 must provide at least 10 comments leaving actionable technical feedback to get full credit on this part. Leaving even a single rude, mean, or counterproductive comment will earn you 0 points on this part. 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.