COMP250: Project 4: Coding BFS + DFS

**DUE DATE: End of day (11:59pm), Sunday December 18th. No late submissions allowed – this is past the last day of class, and I intend on having grading finished up ASAP.**

# Summary

In this project, you’ll be coding BFS as well as *both* the iterative and recursive approaches to DFS. Even though we may not have finished all the examples at the time of releasing this, the slides are fully available for you to view, so I expect people to start the project ASAP.

You have been given:

* **Node.java**: Contains a complete class definition for Nodes for this project. I gave this instead of leaving it blank because the project wouldn’t compile without it working.
* **Main.java:** Contains methods for **bfs/dfsIterative/dfsRecursive**. They contain a **return false;** which allows the code to compile in the meantime. **Remove these before starting steps 1-3.**It also contains a **main** method which contains code that calls the search methods and prints the results. Alongside that, it includes a **buildGraph()** method which builds a test graph I made. The project 4 download includes **Graph Image.png** which is a lazily drawn image of the graph. See **Testing your Code** on page 3 for more.

# Step 1: Breadth-First Search (30 points)

Breadth-First Search uses a **Queue** to determine which Node to visit next. We’ll use the built-in **LinkedList** class for this. If you create it like **Queue<Node> queue = new LinkedList<>();** you’ll have a working Queue. The **add()**/**remove()** methods are what you should use to modify it.

**Main.java** has an empty **bfs()** method provided. Note that its return type is **boolean**. You should begin by removing my placeholder **return false**, creating the queue, and adding **start** to it. Then, make a **while** loop that runs while the queue isn’t empty, so it ends when there’s no nodes left.

Inside the loop, adapt the steps from the slides into code, but including a step that checks if the Node’s data matches **targetData**:

1. Remove (and store a copy of) the next node from the queue.
2. If the node’s data matches **targetData**, print “found target” and return **true**.
3. Else, visit that node using its **visit()** method so that we don’t visit it again later.
4. Next (still inside the else), use a foreach loop on the Node’s array of **neighbors.** For each Node, if it has both not been visited *and* is not already in the queue, you should add it to the queue.

The last thing to do is **return false** after the while loop is over, which signals that the search did not find the target data in the graph. And that’s it for BFS!

# Step 2: Depth-First Search (Iterative) (20 points)

Depth-First Search uses a **Stack** to determine which Node to visit next. Like BFS, we’ll use a built-in class. **Stack<Node> stack = new Stack<>();** will set one up.

This is significantly less guided than step 1, because DFS and BFS *only* differ in which data structure (Queue or Stack) they’re using. In other words, once you verify that BFS is working, you should be able to copy your code from BFS, change all the Queue references to Stack references (including **add**/**remove** being changed to **push**/**pop**), and it should work!

# Step 3: Depth-First Search (Recursive) (50 points)

Recursive DFS takes more thinking. It doesn’t need a Stack to determine where to go next, because the recursion itself keeps track of that. When a method finishes, it returns to whichever method called it, which takes care of remembering the node order.

First, check if the current node’s data matches the target, and if so, print “target found” and **return true**. If it *doesn’t* match, enter an **else** statement. Inside, set up a boolean **found** initialized as **false**. We’re assuming the target hasn’t been found unless it’s overridden with **true** later. After setting that up, you can visit the node.

Then, use a foreach loop for the node’s neighbors. For each unvisited neighbor, set the value of **found** to be the result of calling **dfsRecursive()** on that neighbor. This means that if this neighbor ends up being the target, **found** will be set to **true**.

After, still in the foreach loop, do a second condition that returns **true** if **found** is **true**. This allows us to return a true result from potentially multiple layers deep of recursion. Lastly, after the loop (still in the **else**), **return false** to signal that the target was not found.

***See page 3 for how to test the code + grading/submission information.***

# Checking if Searches Work Properly

Do 2 tests with your code: one where the target exists in the graph, and one where it doesn’t. This ensure that the pathing is correct in both situations and the searches behave correctly.

To do this, modify **targetData** in **main()**. By default, it will be set to “C”, which is a node that exists in the graph, so running the searches should be successful. To check for a node that *doesn’t* exist in the graph, change **targetData** to something other than A-I. I used “Z” for this.

**Cross-check with the expected paths below prior to submitting.**

**Searching for C**BFS: **E A B G C**  
DFS (Iterative): **E G B I H F D C**   
DFS (Recursive): **E A B C**

**Searching for Z**BFS: **E A B G C F I D H**  
DFS (Iterative): **E G B I H F D C A**  
DFS (Recursive): **E A B C F D H I G**

# Submission & Grading

You should submit all relevant files in a .zip file (or similar) to Project 4’s submission page on Blackboard. Please name the .zip file using your first name and first initial of last name (for example, “SeanS\_Project4”). Grading is as shown in the previous headings. There is also a grading breakdown text file attached on Blackboard.