Thomas Mathew
CS 261
Assignment 7: Answers

1. How is the graph stored in the provided code? Is it represented as an adjacency matrix or list?

The graphs are stored as lists: the values are stored in a simple array and show, for each value, its connections to other values. An adjacency matrix representation would include a 2D array and show visually show the edges and vertices of the graph.

2. Which of the 3 graphs are connected? How can you tell?

Graph #1 is **not** connected. One example to prove it: 7 is only connected to 6. 6 is connected to 7, as well as 3 and 4. However 3 and 4 are connected to 6 only. So it is impossible to move from 7 to any value other than 6, 3, and 4.
Graph # 2 is actually a "Complete" graph, where every vertex is connected to all other vertices by a unique edge for each pair of vertices. So this graph is obviously connected.
Graph # 3 is connected as well. The key vertex here is 8. 8 connects to 6, 0, and 3. 0 connects to 7, which connects to 9. 3 starts a path to 2, 1, 5, and 4 consecutively. This accounts for all the vertices, showing that there is a path from any starting point to any destination.

3. Imagine that we ran each depth-first and breadth-first searches in the other direction (from destination to source). Would the output change at all? Would the output change if the graphs were directed graphs?

If we ran searches in the other direction, the output should not change, because the output is only concerned whether a path exists, not what the path is. The program would still find (or not find) the path between the two vertices, it would just process elements in a different order. It might take more or less time depending on the graph.

However if the graphs were directed graphs, and all or some of the edges only allowed traversal in one direction, the results definitely could be different. Consider a simple example: the source is connected to exactly two other vertices by two edges, and both of these edges are one-directional away from the source. If would then be impossible to reach the source from any other point in the graph.

4. What are some pros and cons of DFS vs BFS? When would you use one over the other?

One key pro of BFS is that it will always find the shortest path to the destination, since it behaves like a wave flowing through the maze. So if we needed to find the shortest route through a graph, BFS would be the clear choice. DFS is often better in practice when it comes to memory usage and space. This is because BFS is looking at (potentially) many paths all at once. So if memory usage was a concern, DFS would be a good choice.

In terms of how the given graph affects this question, if the destination/solution is relatively close to the source/root, then BFS is probably better, since it won't "miss" the solution and it can't "flow" relatively quickly to the solution, whereas the DFS might go down the wrong paths and waste a lot of time looking for a solution that was close to the source all along. However, if there are multiple destinations/solutions and they are relatively "deep" in the graph, then DFS may work better since it is

less likely to “miss” and have to backtrack to another path, whereas BFS may waste a lot of time and memory flowing over many different paths and take a long time to find the deep solution.

5. What is the Big O execution time to determine if a vertex is reachable from another vertex?

For both DFS and BFS, the execution time is O(V+E), with V and E representing vertices and edges respectively. This is because as we are searching, we are adding the vertices and edges that are “Reachable” as we iterate along a path of neighboring vertices.