Assignment 7: Graph traversals

This assignment is shorter than the previous assignments and consists of two parts: **DFS and BFS graph traversal** implementations and a **written part**.

Prerequisites

You are provided with a complete circular linked-list deque implementation in `deque.c`. You should be familiar with the Deque data structure and how to use it before starting this assignment. You should also be familiar with the DFS and BFS graph traversal algorithms.

DFS and BFS implementation

Your job is to implement iterative versions of DFS and BFS. These functions take two vertices and return 1 if there is a path between the vertices (a sequence of edges connecting them) and 0 otherwise. As a reference, you are provided with a complete recursive DFS implementation, `dfsRecursive`, in `graph.c`. The `Deque` implemented in `deque.c` can be used as a stack or a queue for your DFS and BFS implementations. This gives you a chance to apply some of your data structure knowledge from earlier in the course. Implement the functions in `graph.c` with the `// FIXME: Implement` comments.

There are two tests in `graphTests.c` that compare the results of your functions against the results of `dfsRecursive` for every pair of vertices on a series of random graphs. Each test prints the graph it is about to test your functions on, so you can more easily use these tests to help you debug. If your implementations pass these tests, then they are likely (but maybe not guaranteed) correct. You can build the tests with `make` or `make tests` and run them with `./tests`. Feel free to tweak or add your own tests for debugging.

> Note: You can ignore the deque tests in `dequeTests.c`. Those are just verification for the skeleton code.

Written questions

The program in `main.c` loads 3 predetermined graphs from files and prints the results of your DFS and BFS functions on each pair of vertices. For each pair, `path` or `no path` is printed for DFS and BFS separately, indicating if there is a path between vertices according to that function. You can build the program with `make` or `make prog` and run it with `./prog`.

After you finish your DFS and BFS implementations, run this program and then answer the following questions.

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

2. Which of the 3 graphs are connected? How can you tell?
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?
4. What are some pros and cons of DFS vs BFS? When would you use one over the other?
5. What is the Big O execution time to determine if a vertex is reachable from another vertex?

Tips

- Pay careful attention to the struct definitions. In particular the `Graph` struct contains an array of `Vertex` structs (all of the verices in the graph), while the `Vertex` struct contains an array of pointer to the `Vertex` structs that are neighbors of the vertex. These pointers point to the same vertices stored in the `Graph` struct's array.
- The edges in this graph are undirected, meaning that an edge from vertex1 to vertex2 and an edge from vertex2 to vertex1 are the same edge.
- A graph is "connected" if there is at least one path between every pair of vertices in the graph.
- If a `Vertex* vertex` had at least three neighbors, to access the third neighbor we would write `vertex->neighbors[2]`.
- You can check for memory leaks on flip by typing `make memcheckTests` or `make memcheckProg` on the command line for `tests` or `prog` respectively.

Grading

- Compiles – 10
- DFS implementation – 20
- BFS implementation – 20
- Written answers – 25 (5 each)

Submission

Submit to TEACH and Canvas the following files. Do not zip either submission.

- `graph.c`
- `answers.txt` or `answers.pdf` containing your answers to the written questions.