Project 6b: Graph Searching

CS271: Data Structures

April 2024

1 Learning Goals

In the first part of this project, we built a graph class(es). Now we use these graph classes to add algorithms to implement both Breadth First Search and also Depth First Search. We will then use both searching algorithms to perform simple tasks.

2 Logistics

This project is the second of a multi-part project. This part of the project uses the class infrastructure from the first part of the project. We will continue to build on this graph class further in Project 7 for other graph algorithms like Spanning Trees, Shortest Path, and various coloring algorithms.

This first part of the project had a "B Level" which requires the construction of one class, a `Graph` class. Optionally, teams may complete an "A Level" which is a more sophisticated graph class structure involving both sparse and dense graphs as derived classes from a base graph class. See previous Part A assignment for details. If you did not implement the A level in Parta, you can still add it in Part B and get credit for it. The main files I am giving you assume A level. You can change the comment sections easily to convert the main files to the B level. You will have to adapt the makefile on your own for the B level as the makefile assumes A level completion.

3 BFS

See the file `main_bfs.cpp`. This file uses our graph class to call BFS. Then it uses the BFS algorithm results to answer some questions about the graph.

The BFS algorithm uses a queue. I suggest using the STL library for the queue class there.

For the BFS, we will be assuming undirected graphs. Your makefile will still compile a directed and undirected version, but we will only run the undirected version for BFS analysis.

We will use a data file called `big_graph1.txt` which contains 100 vertices and 294 edges. We will read this file via stdin using the same file format from Part A of this assignment. The main file does this for you already.

3.1 Run BFS

Your first task is to implement the BFS algorithm as it is presented in the book. You will need to create a table to store the information used by the BFS algorithm as it traverses vertices. Since the table is dependent on the number of vertices, the table will need to be dynamically allocated either as part of the constructors or in the start of the BFS algorithm. Be sure you add appropriate code to deallocate the table in the destructor.

Implement a print table function for BFS so we can see the table information correctly completed. The main file calls the print table for graphs with 10 or fewer vertices.

3.2 Most Distant Vertices

After BFS runs, main calls a graph function `printMostDistant(source)` which should print a list of all the vertices which are the furthest from our source vertex that we just ran BFS on. You will use the information already in the table to find the max distance, and then print a list of all vertices which have this same max distance from the source vertex.

3.3 Connected

Next, main calls a graph function `isConnected()` which returns true if the graph is connected. In an undirected graph, the BFS traversal from *any* source will be sufficient to determined connectedness or not. This is not the case for directed graphs, but we will not consider those here.

3.4 Print BFS Path

Next main calls `printBFSPath(source,destination)` which will print a path from the source node (that we just ran BFS on) to a specified destination node. This algorithm was an exercise in our practice problem set. If no such path exists (in which case the isConnected() above would have returned false), then print ”No such path”.

3.5 Logistics

This concludes the work we will do with our BFS algorithm. I have provided my output files when running these algorithms on the big data file.

I have provided a `main_bfs.cpp` file. This runs the above code. The makefile will compile `main_bfs_undirected` and `main_bfs_directed`. We will only use the undirected version of this, but your directed version should work just fine without any changes to the code.

```
> make
> ./main_bfs_undirected <big_graph1.txt
```

4 Depth First Search

In this next section, we will run the DFS algorithm. This is run on the whole graph instead of just a single source node. While the DFS algorithm works just fine with undirected graphs and the makefile will compile a version for undirected graphs, we will instead use the directed version, treating our graphs as directed graphs.

I suggest implementing the DFS using the book code instead of the stack-based version (exercise in Practice) because the stack based version is hard to get the clock times correct. Your dfs algorithm will need to fill in information in a table: color, discover time (d), finish time (f), and predecessor.

Implement a print table function for DFS so we can see the table information correctly completed. The main file calls the print table for graphs with 10 or fewer vertices.

4.1 Parenthesization

The main file then calls a `printParenthesization()` function (for graphs with 10 or fewer vertices). Implement this function to use the DFS table information to print out the parenthesization of the search. You can see my results in the sample output file.

4.2 Edge Classification

The third part of the DFS section is to classify edges in our graph. See Proof 12 for the three categories and some tips on how to categorize edges into: forward/tree edges, back edges, or cross edges. See the sample output for how this is to work.

4.3 Topological Sort

The fourth and final part of the DFS search is to implement the topological sorting algorithm as it is specified in Section 20.4 in the book. Again, I provide my own sample output file.

5 Methods

For this Part B of the project I created the following algorithms.

```
// BFS based algorithms
void     BFS                     ( int source );
void     printBFSTable           ( int source );
void     printBFSPath            ( int s, int d );
void     printMostDistant        ( int s );
bool     isConnected             ( void );

// DFS based algorithms
void     DFS                     ( void );
void     DFS_Visit               ( int v, int &clock );
void     printDFSTable           ( void );
void     printTopologicalSort    ( void );
void     printDFSParenthesization( void );
void     classifyDFSEdges        ( void );
void     indexSort               ( int a[] );
```

As a division of labor, a group with two people can divide it between BFS and DFS. For a group with three people, I suggest one person implement the basic BFS and DFS algorithms (with DFS-Visit). Person 2 can implement the follow up BFS derived algorithms while Person 3 can implement the follow up DFS derived algorithms.

I put all of these algorithms in my base Graph class (for the A level implementation). Note the indexSort algorithm is my own internal (protected) method which sorts the vertex indices for the topological sorting.

6 Submission

Students should submit a tar/zip file or compressed file with the following:

- `main_bfs.cpp` and `main_dfs.cpp`: I have already given you these two files. Your project should work with them. Include them in your submission package. If you are completing the B Level, then you should modify these two main files so they work with your project – most of these modifications have obvious comments in the files.
- A makefile which compiles the project. It should compile four targets
  - A `main_bfs_undirected.cpp` executable which runs BFS on a graph that is undirected.
  - A `main_bfs_directed.cpp` executable which runs BFS on a graph that is directed.
  - A `main_dfs_undirected.cpp` executable which runs DFS on a graph that is undirected.
  - A `main_dfs_directed.cpp` executable which runs DFS on a graph that is directed.

  I have included my makefile as an example. You should be able to just keep this same makefile and submit it with your project. If you complete the B level, you will have to modify the makefile so it works with your project. It should be pretty easy to delete the things that go with the A level implementation.
- Graph.h
- Graph.cpp
- Optionally: DenseGraph.h DenseGraph.cpp SparseGraph.h SparseGraph.cpp

You should absolutely take your tar/zip file, copy it into an empty directory, and test that it compiles and executes correctly.

7 Auxiliary Files

I have provided to you:

- `big_graph1.txt`: To be used with the BFS algorithm.
- `BFS_out.txt`: My sample output when I run the following

```
> make
> ./main_bfs_undirected <big_graph1.txt
```

- `graph1.txt`: To be used with the DFS algorithm.
- `graph2.txt`: To be used with the DFS algorithm.
- `graph3.txt`: To be used with the DFS algorithm.
- `graph4.txt`: To be used with the DFS algorithm.
- `BFS_out4.txt`: My sample output when I run the following

```
> make
> ./main_dfs_directed <graph4.txt
```