Homework 2

Matthew Wilkinson

mmw071@uark.edu

100434077

1. Describe the purpose of the homework and what you did / what you implemented.
2. Describe your approach and explain how you solved the problem (the key ideas for implementing each required function).
3. Discuss any bugs or problems you encountered and how you fixed them.
4. Include screenshots or images of your results.

Purpose

The purpose of the homework is for students to implement fundamental graph search algorithms like: Breadth-First Search (BFS), Depth-First Search (DFS), and Recursive Depth-First Search (rDFS). The goal was to correctly compute the path from the start node to the destination node. Additionally, the algorithms were tested using the small graph from homework 1 (the campus map) that will find the path from JBHT to HAPG. I implemented bfs(), dfs(), and rdfs() functions that correctly searched through the graph example to find the path from JBHT to HAPG.

Approach

For all functions (bfs, dfs, and rdfs), I used lecture notes to guide me to my final implementation. For BFS, I used the pseudocode in lecture 7 page 26 to point me in the right direction. I noticed that the first section of code missing was setting the start as visited and adding the start to the queue. The next section of missing code was setting the next node from the queue as visited so we don't revisit the node/value. I also decided to implement the IF statement that would check if the value was destination since we if was, we'd end there since we know we're done. The last section of missing code was just checking if the neighbor was visited, and if not, add to the queue and setTrace to keep track of where we are.

For DFS, I reference page 23 on lecture 7. I noticed that the first section of missing code was pushing the start value to the stack. The next section missing code was missing checking if the node was visited or not and the IF statement to see if the value was destination. The last section of missing code mainly check if the neighbor was visited, if not, it was pushed onto the stack and setTrace.

For rDFS, I reference page 24 on lecture 7. The first part of missing code was marking the start value of the function as visited and checking if the start value was equal to destination or not. The last section checked if the neighbor was visited, if not, it would setTrace and recursively call the rDFS function with value *v* (neighbor).

Bugs/Problems

One problem I had was with the main.cpp file. It had an error saying 'numeric_limits' is not a member of std. This occurred because the file used *std::numeric_limits<std::streamsize>::max()* in main.cpp and the library wasn't included in the file. So, I fixed this by including the library limits. I inserted the line '#include <limits>' on line 13 of main.cpp which seemed to resolve the issue. I know I am not supposed to edit main.cpp but I couldn't figure out any other simple way to resolve the problem.

I still had problems with OpenCV working with my machine. I think I just need to restart, I guess. However, my makefile still worked perfectly and the programs executed like I expected from the homework 2 pdf. Besides that, I really didn't have issues with this assignment, the lecture notes helped a lot!

RESULTS SECTION ON NEXT PAGE

Results

Terminal of Mingw64 of Final Results:

```
mww61@Matthew MINGW64 /c/Users/mww61/gitRepos/algor/algorithmsclass/hw2
$ mingw32-make bfs
g++ -I./include/ src/linked_list.cpp src/graph.cpp src/queue.cpp src/stack.cpp src/bfs.cpp src/dfs.c
pp src/rdfs.cpp src/main.cpp  -o bin/main
./bin/main bfs
Perform unit test on your bfs implementation
Path from 0 to 5 by bfs: 0 1 3 4 5


Path from JBHT to HAPG: JBHT -> HILL -> WJWH -> HAPG

You have to use OpenCV to visualize your map road

clear

mww61@Matthew MINGW64 /c/Users/mww61/gitRepos/algor/algorithmsclass/hw2
$ mingw32-make dfs
g++ -I./include/ src/linked_list.cpp src/graph.cpp src/queue.cpp src/stack.cpp src/bfs.cpp src/dfs.cpp
 src/rdfs.cpp src/main.cpp  -o bin/main
./bin/main dfs
Perform unit test on your dfs implementation
Path from 0 to 5 by dfs: 0 1 2 4 5


Path from JBHT to HAPG: JBHT -> HILL -> WJWH -> HAPG

You have to use OpenCV to visualize your map road

clear

mww61@Matthew MINGW64 /c/Users/mww61/gitRepos/algor/algorithmsclass/hw2
$ mingw32-make rdfs
g++ -I./include/ src/linked_list.cpp src/graph.cpp src/queue.cpp src/stack.cpp src/bfs.cpp src/dfs.cpp
 src/rdfs.cpp src/main.cpp  -o bin/main
./bin/main rdfs
Perform unit test on your rdfs implementation
Path from 0 to 5 by rdfs: 0 1 3 4 5


Path from JBHT to HAPG: JBHT -> HILL -> WJWH -> HAPG

You have to use OpenCV to visualize your map road

clear
```

Operation Counting Results

Algorithm 1:

Since the line in question (line 6) has 3 operators (+, *, and +), each time that line is ran, the count will increase by 3. Since the inside FOR loop will run 100 times (from j = 0 to j = 99), we can do 3 x 100 = 300. The outside FOR loop will run an additional 10 times (from i = 0 to i = 9), we will multiply the previous number 300 by 10, 300 x 10 = 3000.

Therefore, the total count of operators is **3000**.

Algorithm 2:

Since we're looking at number of assignments, the only lines we increase the count are lines 6, 7, and 8. Therefore, when those 3 lines are ran, we will increase the number of assignments count

by 3. With this being a bubble sort algorithm, the best-case scenario (minimum number of assignments) would be if the data/array was already sorted, which means no assignments would ever occur. However, the worst-case scenario would be the data/array being sorted in reverse (like 3,2,1 instead of 1,2,3). This means that a swap would occur during every FOR loop iteration. With the inside FOR loop running n-1 times, and the outside FOR loop running n times. The maximum number of swaps would be n x (n-1). We will need to multiply the number of swaps times 3 to account for the assignments count incrementing by 3 every time.

Therefore, the maximum number of assignments would be **3n(n-1)**

AND the minimum number of assignments would be **0**.

Algorithm 3:

I hand walked through the code to figure out the number of comparisons in the algorithm. The only comparisons that counted towards increasing the number of comparisons counter were lines 9, 11, and 15. My work for algorithm 3 is show in the screenshot below.

The number of comparisons this algorithm has is **11**.

Screenshot of Walking Through the Code by Hand: