Analysis of Algorithms II

BLG 336E

Homework 1

Seyyid Osman Sevgili
504221565

2023 - Spring

Contents

1 Explanation of Code and Solution

My code consists of five main steps:

- Retrieving the file path as an argument, reading it, and creating Kid objects.
- Creating a Graph object and calculating the adjacency matrix based on the power and squared distances.
- Running the BFS algorithm.
- Running the DFS algorithm.
- Writing the results to the relevant txt files.

To elaborate, first, I retrieved the command line arguments using char* argv[] in the main code. Then, I read the input file using the <fstream> library, obtaining the necessary inputs from the command line arguments.

Second, I created the Graph object, which initializes the num_vertices and adj_m variables in its constructor. I calculated the adjacency matrix using a for loop and if statements, looping over each kid and checking their connection to all other kids to create a matrix with 1s and 0s representing whether a kid can pass a ball to another kid.

Third, I implemented the BFS algorithm using the adjacency matrix. I used the queue data structure to traverse through the neighbors. I dequeued the first element in the queue and traversed each neighbor of that element, pushing every unvisited neighbor onto the queue. I then found the shortest path between the source and target nodes.

Fourth, I developed the DFS algorithm using a recursive approach. I used two flags, in_progress and all_done, to check if the current node was in progress or had already been processed. I searched for a cycle by checking the depth and avoiding depth 2 cycles.

Finally, I wrote the results to the txt files in each algorithm.

1.1 Pseudo Code and Time Complexities

1. **Reading the input file:** $O(\text{num\_kids})$
2. **Filling the Kids vector:** $O(\text{num\_kids})$
3. **Creating a Graph object:** $O(\text{num\_kids}^2)$ (due to the adjacency matrix representation)

4. **Filling the adjacency matrix:** $O(\text{num\_kids}^2)$ (nested loop comparing each pair of kids)
5. **Writing the graph to an output file:** $O(\text{num\_kids}^2)$ (writing the adjacency matrix)
6. **Performing BFS on the graph:** $O(\text{num\_kids}^2)$ (since we are using an adjacency matrix)
7. **Writing the BFS results to an output file:** $O(\text{num\_kids})$
8. **Performing DFS on the graph:** $O(\text{num\_kids}^2)$ (since we are using an adjacency matrix)
9. **Writing the DFS results to an output file:** $O(\text{num\_kids}^)$

2 Why should you maintain a list of discovered nodes? How does this affects the outcome of the algorithms?

There are several reasons for maintaining a list of discovered nodes in graph traversal algorithms. Two of the main reasons are preventing infinite loops and increasing efficiency. If we do not maintain a list of visited nodes, we might end up in an infinite loop due to the presence of multiple paths leading back to the same node. Furthermore, without a list of visited nodes, the algorithm may revisit nodes that have already been processed, leading to redundant computation and wasting time. By maintaining a list of discovered nodes, we can ensure that the traversal algorithm avoids infinite loops and runs more efficiently by preventing unnecessary revisits to previously explored nodes.

3 How does increasing the number of the kids affects the memory complexity of the algorithms?

Memory requirements are directly affected by the number of kids, as all algorithms' space complexities depend on the number of vertices in the graph. Therefore, space complexity increases quadratically with the increasing number of vertices, which in this case is proportional to the number of kids.

3.1 Space Complexities

1. **Kids vector:** $O(\text{num\_kids})$
2. **Graph object (Adjacency matrix):** $O(\text{num\_kids}^2)$

3. **BFS related data structures:**
   (a) *Visited array:* O(num_kids)
   (b) *Parent array:* O(num_kids)
   (c) *Wait list (Queue):* O(num_kids)
4. **DFS related data structures:**
   (a) *In-progress vector:* O(num_kids)
   (b) *All-done vector:* O(num_kids)
   (c) *DFS cycle path (Stack):* O(num_kids)

3.2 Run Time Table and Graph

Case	Child Number	Runtime (microseconds)
1	7	1218
2	10	1365
3	20	1378
6	49	2147
7	191	11423
9	376	36752
11	821	106593
13	1260	202773

Table 1: Runtime results for different input cases

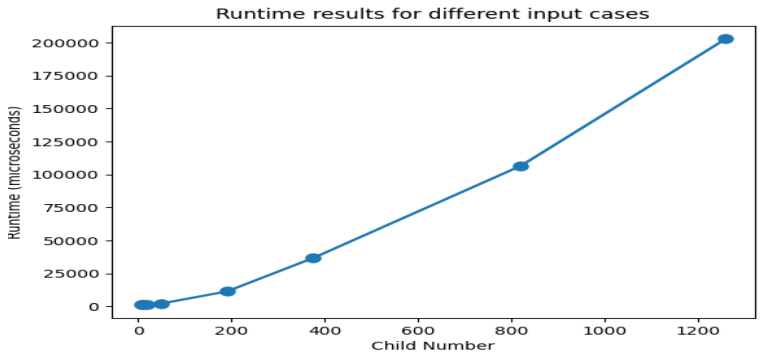

Figure 1: Runtime results for different input cases

As we can see from the above graph and table run time increases quadratically with increasing child number.

4 Test Case Results

Finally, i run the code with shared public test cases on the docker. I got following results.

```
vscode ➜ /workspaces/Project 1 (main) $ calico public_test_cases.t
init ..................................... PASSED
build .................................... PASSED
case_run1 ................................ PASSED
case_graph1 .............................. 0.7 / 0.7
case_bfs1 ................................ 2.2 / 2.2
case_dfs1 ................................ 2.1 / 2.1
case_run2 ................................ PASSED
case_graph2 .............................. 0.7 / 0.7
case_bfs2 ................................ 2.2 / 2.2
case_dfs2 ................................ 2.1 / 2.1
case_run3 ................................ PASSED
case_graph3 .............................. 0.7 / 0.7
case_bfs3 ................................ 2.2 / 2.2
case_dfs3 ................................ 2.1 / 2.1
case_run6 ................................ PASSED
case_graph6 .............................. 0.7 / 0.7
case_bfs6 ................................ 2.2 / 2.2
case_dfs6 ................................ 2.1 / 2.1
case_run7 ................................ PASSED
case_graph7 .............................. 0.7 / 0.7
case_bfs7 ................................ 2.2 / 2.2
case_dfs7 ................................ 2.1 / 2.1
case_run9 ................................ PASSED
case_graph9 .............................. 0.7 / 0.7
case_bfs9 ................................ 2.2 / 2.2
case_dfs9 ................................ 2.1 / 2.1
case_run11 ............................... PASSED
case_graph11 ............................. 0.7 / 0.7
case_bfs11 ............................... 2.2 / 2.2
case_dfs11 ............................... 2.1 / 2.1
case_run13 ............................... PASSED
case_graph13 ............................. 0.7 / 0.7
case_bfs13 ............................... 2.2 / 2.2
case_dfs13 ............................... 2.1 / 2.1
Grade: 40.00000000000001 / 40.00000000000001
```

Figure 2: Test Case Results

This shows that i wrote algorithm correctly.