CCDSALG S13 Group 7
November 25, 2023
Clavano, Angelica Therese I. (Jack)
Homssi, Yazan M.
Young, Henzley Emmanuel S.
Jimenez, Yves Alvin Andrei A.

Major Course Output 2: Social Graph Dataset

I. Introduction

A brief introduction to the project and an outline of the contents of the report.

This project aims to process data taken from a Facebook snapshot containing networks of select American colleges and universities in 2005. Originally presented in (.mat) MATLAB format, these (.txt) text files (as paths) are inputted into our program. The program then converts the graph data into a data structure. The user is given three options they could use on the given dataset: (1) Display the friend lists of a user given their ID, (2) Display connections between user IDs, and (3) exit the program. This paper breaks down the functionality of the code implemented, analyzes the algorithms present, and provides a concise summary of our findings.

Outline

II. Data Structures Used

A clear description of the data structure used in representing the social graph data, including the rationale behind the use of such data structure/s. Show supporting code if necessary.

The data structure used in the project involves an Adjacency Matrix. This is encapsulated as a pointer within **struct Friendship_Graph** (Figure 1) and creates a 2D array, or an n by n, square-shaped matrix that shows if a user has connections or not by filling the matrix with 1s and 0s, where 1 represents a connection and 0 does not. The graph itself is dynamically allocated in memory by **struct Friendship_Graph* initializeGraph** (Figure 2) and it's filled by **void InsertEdge** (Figure 3). The rationale behind using this data structure is that it's easier to see and check for connections or if user1 has user2 added. This structure is fitting for depicting relationships in a social network where connections exhibit symmetry; if A is considered a friend of B, then conversely, B is also acknowledged as a friend of A. This makes sense for the source

of the data set, where if someone adds someone as a friend, they would have each other “added.”

```c
/*
 * Structure that represents a graph data type.
 */
struct Friendship_Graph {
    int nAccounts;
    int** adjacencyMatrix; // adjacency matrix is just like a square matrix
};
```

Figure 1. Implementation of Friendship_Graph().

Notice that adjacencyMatrix has data type int**. This is a pointer to a pointer indicating that it holds a 2-Dimensional array.

```c
/*
 * Initializes a graph with the specified number of accounts.
 * @param numAccounts The number of accounts in the graph
 * @return A pointer to the initialized graph
 */

struct Friendship_Graph* initializeGraph(int numAccounts) {
    struct Friendship_Graph* graph = (struct Friendship_Graph*)malloc(sizeof(struct Friendship_Graph));
    graph->nAccounts = numAccounts;

    // Allocate memory for the adjacency matrix
    graph->adjacencyMatrix = (int**)malloc(numAccounts * sizeof(int*));
    for (int i = 0; i < numAccounts; i++) {
        graph->adjacencyMatrix[i] = (int*)calloc(numAccounts, sizeof(int));
    }

    return graph;
}
```

Figure 2. Implementation of InitializazeGraph().

In the for-loop, the program goes through each row and allocates memory for an array of integers which will become the columns of the 2D Array and initializes each element to have 0. This is called within the FileReader function.

```
/*
 * Adds an edge between two nodes in the graph, representing a friendship.
 * @param graph The friendship graph
 * @param node1 The ID of the first person
 * @param node2 The ID of the second person
 */
void InsertEdge(struct Friendship_Graph* graph, int node1, int node2)
{
    graph->adjacencyMatrix[node1][node2] = 1;
    graph->adjacencyMatrix[node2][node1] = 1; // friendship is bi=directional (goes both ways)
}
```

Figure 3. Implementation of InsertEdge().

This function is called within FileReader() which serves two functions, to handle if the file exists or not, and to help create the matrix by calling InitializeGraph and InsertEdge depicted above. InsertEdge is called after FileReader reads the lines after the first one, which has the number of accounts and number of relationships. Every line after this depicts user (or node) 1 and which user (as user or node 2) they have “added” on the social media platform. Since the friendship is bidirectional, we have to add “1” on the inverse row manually as well.

Another data structure used is the Stack data structure, which is used in the Depth-First Search within CheckConnections(). (Figure 5). This search method makes use of the “Stack” Data Structure, unlike Breadth-First Search– where BFS makes use of “Queue” Data Structure for finding the shortest path. “Stack” is a linear type of data structure that follows the LIFO (Last-In-First-Out) principle and allows insertion(push) and deletion(pop) operations from one end of the stack data structure, that is top. Implementation of the stack can be done by an array and a linked list.

III. Algorithms Used

A clear description of the algorithms used to display the friend list as well as to display the connection between two people in the network.

When it comes to graph traversals, we know that there are two methods in doing so:

1. Depth-First Search or DFS algorithm
2. Breadth-First Search or BFS algorithm

The algorithm used for our project is called the “Depth-First Search” algorithm, it is a recursive algorithm that uses the backtracking principle to traverse or search all the vertices of a tree data structure or a graph. The algorithm starts at the initial/root node, and explores as far as possible along each branch before backtracking. When a dead-end occurs in our iteration, the Depth-First Search algorithm uses the “Stack” Data Structure and performs two stages, first visited vertices are pushed into the stack, and second if there are no vertices then visited vertices are popped, the outcome of a Depth-First Search traversal of a graph is a spanning tree (basically a graph devoid of loops). The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

```
void ShowFriends(struct Friendship_Graph* graph, int personId)
{
    if (personId < 0 || personId >= graph->nAccounts) {
        printf("Error: Person with ID %d not found in the dataset.\n", personId);
    } else {
        printf("Friend list for person with ID %d: ", personId);
        int totalFriends = 0;
        for (int i = 0; i < graph->nAccounts; ++i) {
            if (graph->adjacencyMatrix[personId][i] == 1) {
                printf("%d ", i);
                totalFriends++;
            }
        }
        printf("\nTotal number of friends: %d\n", totalFriends);
    }
}
```

Figure 4. Implementation of ShowFriends().

The ‘showFriends’ function (See above) iterates through a single row in the adjacency matrix corresponding to the personID. For each entry in the row with value 1, it prints the corresponding column index, which represents a friend.

```c
int CheckConnections(struct Friendship_Graph* graph, int person1, int person2) {
    if (person1 < 0 || person1 >= graph->nAccounts || person2 < 0 || person2 >= graph->nAccounts) {
        printf("Error: One or both persons not found in the dataset.\n");
        return 0;
    }

    // check for a connection using depth-first search (DFS search)
    int* visited = (int*)calloc(graph->nAccounts, sizeof(int));
    int stack[graph->nAccounts];
    int top = -1; // initialize the top of the stack
    int current = person1;
    visited[current] = 1;

    printf("There is a connection from %d to %d!\n", person1, person2);
    printf("%d is friends with", person1);

    while (1) {
        int foundConnection = 0;

        for (int i = 0; i < graph->nAccounts; ++i) {
            if (graph->adjacencyMatrix[current][i] == 1 && visited[i] == 0) {
                stack[++top] = i;
                visited[i] = 1;

                printf(" %d", i);

                if (i == person2) {
                    foundConnection = 1;
                    break;
                }
            }
        }

        if (foundConnection) {
            printf("\n");
            break;
        }

        if (top == -1) {
            printf("\n");
            printf("%d is not friends with %d.\n", person1, person2);
            break;
        }

        current = stack[top--];
        printf(" ->");
    }
```

Figure 5. Implementation of CheckConnections().

The ‘CheckConnections’ function (See above) in our program makes use of the DFS algorithm that traverses the friendship graph, and checks if there is a connection between two given people.

IV. Algorithm Analysis

An algorithmic analysis of the algorithms in terms of time complexity.

The time complexity of the DFS algorithm is O(V+E), where V is the number of vertices and E is the number of edges in the graph.

V. Code Testing

A quick test of the code to demonstrate that everything is working properly.

For the purposes of testing, we will use the text file, Rice31.txt, that was provided alongside the specifications of the assignment.

1. Display Friends List:
    a. For the purposes of this test, we will use ID: 1 and 4086.

b. According to Rice31.txt ID: 1 has 119 friends on their friend's list.
c. The program's output supports this:
d.

```
Enter the file path: Rice31.txt

MENU:
1. Display Friend List
2. Display Connections
3. Exit
Enter your choice (1-3): 1
Enter the ID number of a person: 1
Friend list for person with ID 1: 22 45 75 284 287 364 423 483 603 659 671 683 691 752 789 799 870 930 957 997 1015 1023
 1026 1053 1064 1157 1257 1270 1280 1316 1337 1368 1380 1452 1495 1546 1613 1621 1638 1644 1678 1810 1822 1834 1926 1931
 1996 2027 2047 2055 2057 2080 2139 2176 2191 2242 2282 2357 2367 2369 2374 2377 2413 2431 2448 2459 2462 2464 2468 2511
 2516 2556 2584 2663 2668 2672 2706 2741 2748 2815 2821 2840 2851 2886 2887 2915 2919 2933 2960 2986 3024 3126 3135 3136
 3188 3319 3343 3395 3410 3434 3442 3450 3513 3550 3557 3593 3625 3722 3833 3844 3866 3878 3926 3950 3966 3996 3998 4056
 4080
Total number of friends: 119
```

e. According to Rice31.txt, ID: 4086 has a total of 86 friends.
f. The program's output also supports this:
g.

```
MENU:
1. Display Friend List
2. Display Connections
3. Exit
Enter your choice (1-3): 1
Enter the ID number of a person: 4086
Friend list for person with ID 4086: 15 115 128 136 205 230 233 253 300 337 365 377 383 419 421 478 497 525 588 625 651
690 745 758 774 849 929 958 1082 1206 1279 1297 1348 1439 1452 1468 1472 1480 1544 1576 1593 1788 1794 1823 1857 1872 19
18 1921 1988 1993 2044 2054 2085 2135 2153 2324 2368 2391 2392 2418 2421 2502 2578 2581 2626 2950 3132 3252 3271 3318 33
97 3459 3663 3719 3747 3856 3862 3904 3932 3981 4008 4045 4072
Total number of friends: 83
```

2. Display Connections:
   a. For the purposes of this test, we will be using IDs: 1&2 and IDs: 1&9999:
   b. According to Rice31.txt, IDs:1 and 2 have a connection
   c. The program's output support's this:
   d.

```
Enter the file path: Rice31.txt

MENU:
1. Display Friend List
2. Display Connections
3. Exit
Enter your choice (1-3): 2
Enter two ID numbers: 1
2
There is a connection from 1 to 2!
1 is friends with 22 -> 129 -> 6 -> 8 -> 41 -> 12 -> 75 -> 29 -> 0 -> 23 -> 136 -> 3 -> 63 -> 25 -> 147 -> 259 -> 54 ->
73 -> 167 -> 86 -> 87 -> 32 -> 55 -> 292 -> 792 -> 38 -> 84 -> 69 -> 46 -> 24 -> 9 -> 119 -> 36 -> 94 -> 138 -> 111 -> 1
17 -> 5 -> 45 -> 19 -> 20 -> 121 -> 10 -> 194 -> 106 -> 62 -> 114 -> 76 -> 118 -> 385 -> 202 -> 788 -> 13 -> 11 -> 250 -
> 85 -> 364 -> 15 -> 26 -> 175 -> 66 -> 268 -> 403 -> 2
Connection exists between 1 and 2.
```

   e. According to Rice31.txt, IDs 1 and 9999 have no connection since ID 9999 does not exist.
   f. The program's output also supports this:

g.

```
Enter the file path: Rice31.txt

MENU:
1. Display Friend List
2. Display Connections
3. Exit
Enter your choice (1-3): 2
Enter two ID numbers: 1
9999
Error: One or both persons not found in the dataset.
Connection does not exist.
```

VI. Summary

Summary of findings and the group's learnings, insights, and realizations with respect to data structures and graphs after accomplishing this project.

Perhaps the most difficult aspect of the project was actually starting to conceptualize what kind of approach we were going to utilize in representing the graphs of the given specification. Particularly, selecting the most appropriate kind of data structure, visualizing how it will function a priori, and most importantly, actually implementing it to a C program. Furthermore, the most interesting aspects of this project were the following insights:

1) Alternative traversals
   - Although DFS was utilized in the program, an option to utilize BFS was also viable for implementing the traversal within the graph. However, because stacks were more extensively discussed during the duration of CCDSALG, the group opted to use DFS in order to use a data structure that was more familiar to the group due to the fact that BFS contrarily uses queues.

2) Memory allocation
   - Similar to the group's first MCO, we opted to use C instead of Java because most group members are unfamiliar with Java's syntax. Be as it may that C is easier to use for the group, it also brings to light the memory allocation constrictions of C; since Java makes dynamic memory allocation and pointer handling more efficient, the implementation of the code in C was significantly more annoying because of the constant need to free up space–which also emphasizes the importance of memory and space complexity which will be tackled in the course succeeding CCDSALG.

VII. Members and Contributions

Member	**Contributions**
Clavano, Angelica Therese I. (Jack)	1. Introduction 2. Data Structures Used 3. Added PrintGraph for checking in

	code
Homssi, Yazan M.	1. Data Structures Used 2. Algorithms Used
Young, Henzley Emmanuel S.	1. Code Testing 2. Test Files
Jimenez, Yves Alvin Andrei A.	1. Code 2. Summary

VIII. References (APA 7th)

Depth First Search (DFS) algorithm. (n.d.).

https://www.programiz.com/dsa/graph-dfs

DFS Algorithm - javatpoint. (n.d.) https://www.javatpoint.com/depth-first-search-algorithm

GeeksforGeeks. (2023a, June 9). *Depth first search or DFS for a graph*.

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

GeeksforGeeks. (2023b, August 8). *Difference between BFS and DFS*.

https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/

S, R. A. (2023a, April 10). *Implementing stacks in data structures*. Simplilearn.com.

https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures

S, R. A. (2023b, October 12). *Learn Depth-First Search(DFS) algorithm from scratch*.

Simplilearn.com.

https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm

Graph Adjacency Matrix (With code examples in C++, Java and Python). (n.d.). Programiz;

Programiz. Retrieved November 27, 2023, from

https://www.programiz.com/dsa/graph-adjacency-matrix