

CMSC 123: Data Structures

1st Semester AY 2019-2020

Prepared by: CE Poserio & KBP Pelaez

[INLAB] Exercise 08: Graph Traversal Algorithms

Graph Traversals

Graph traversal (or *graph search*) is the process of visiting (or traversing/updating) each vertex in a graph. This is a generalization of tree traversals.

In a graph traversal algorithm, a *search tree* is superimposed in the graph and a list of vertices are kept to determined which were already visited and should be explored further. Graph traversal algorithms mainly differ on how they *choose the next vertex* to explore.

Depth-First Search

Depth-first search (DFS) is a generalization of the *post order traversal* in trees, *i.e.* DFS visits the child vertices before visiting the siblings (in terms of trees). Since there is no hierarchical ordering in graphs, this means that in DFS, the search begins from an arbitrary vertex, then a path is explored as deep as possible by recursively “*DFS-ing*” each of its neighbor.

The DFS algorithm for any graph, $G = (V, E)$, can be summarized as follows:

1. create boolean array **visited** of size $|V|$, initialized as **FALSE** values; this will be used to check and avoid cycles
2. use the following recursive routine:

```
DFS(u):  
    visited[u] = TRUE  
    display vertex u  
  
    for each vertex v adjacent to u:  
        if !visited[v]:  
            DFS(v)
```

3. initial call: **DFS(start_vertex)**

Aside from displaying the vertices of a graph, DFS can also be used to solve other graph problems such as checking if a graph is *connected* or not, counting the number of *connected components* in a graph, and generating *subgraphs*.

Breadth-First Search

Breadth-first search (BFS), on the other hand, visits the neighbors of the current graph before exploring *deeper* into the graph. BFS explores the *breadth* of the search tree first and guarantees that the *nearest* vertices are visited before exploring farther vertices. In a tree, BFS visits the nodes per level, *i.e.*, all nodes in level N must be visited first before exploring any node in level $N + 1$.

The BFS algorithm for any graph, $G = (V, E)$, can be summarized as follows:

1. create a boolean array **visited** of size $|V|$, initialized as **FALSE** values;
2. create an empty list **frontier** which will keep track of the vertices that must be visited next;
3. use the following recursive routine:

```

BFS(u):
    display vertex u

    for each vertex v adjacent to u:
        if !visited[v]:
            visited[v] = TRUE
            frontier.append(v)

    if frontier is not empty:
        BFS(frontier.removeFirst())

```

4. initial call: `BFS(start_vertex)`

BFS is usually used to generate the *shortest path* (i.e., path with the smallest number of steps) from a source vertex to a destination vertex, to detect *cycles* in a graph, and in networking, to broadcast messages to the other nodes in the network.

For both algorithms above, it is assumed that the routines have access to the graph G and the `visited` array (and `frontier` list for BFS).

Tasks

Create a program that displays a valid BFS or DFS¹ traversal of a given unweighted graph with $2 \leq N \leq 25$ vertices and $N - 1 \leq M \leq \frac{n(n-1)}{2}$ edges. The graph is given as described in the next section.

Input

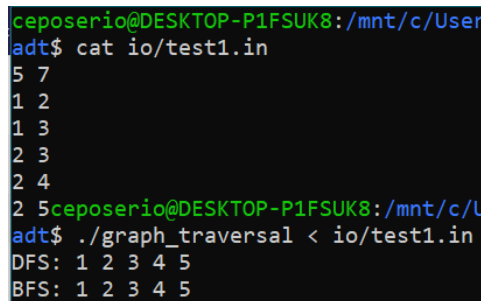
The input consists of several lines. The first line of input consists of two integers N and M . This is followed by M lines; each line consists of two integers u and v where $1 \leq u, v \leq N$.

Output

Output must consist of one line: the DFS traversal or the BFS traversal.

Sample Run

Here is a sample run:



```

ceposerio@DESKTOP-P1FSUK8:/mnt/c/User
adt$ cat io/test1.in
5 7
1 2
1 3
2 3
2 4
2 5
ceposerio@DESKTOP-P1FSUK8:/mnt/c/U
adt$ ./graph_traversal < io/test1.in
DFS: 1 2 3 4 5
BFS: 1 2 3 4 5

```

Figure 1: Sample Run

Several files are given inside the `io` folder. These files are sample input and output test cases. each of the `*.in` files contains an input test case and the corresponding `*.out` file contain the expected output for such input. For example, if `test1.in` is used, `test1.out` is the expected result.

Here are the steps to guide you on how to use these files:

¹The instructor will decide which of the two traversals will be used in the exercise.

1. Compile your program: `gcc -o graph_traversal main.c`. (edit the command, if needed)
2. Use *input redirection* to run the program with desired input test case: `./graph_traversal < test.in`. Take note that the input would still come from `stdin` and not the file; use `scanf` as is.
3. Compare your programs output with `test.out`.

Submission

Submit a `.zip` file named **SurnameEx08.zip** to our Google Classroom. Inside this archive file, there must be a file named `main.c` which contains the `main()` function and other source files needed to run your program.

Questions

If you have any questions, approach your lab instructor.

References

Cormen, Thomas H., Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT press.

Weiss, Mark Allen. *Data Structures and Algorithms*. Benjamin/Cummings.