Assignment Report: Breadth-First Search

CSE-0408 Summer 2021

Md.Taskir Rahman Tasin
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
taskir.rahman72@gmail.com

***Abstract*—Solving problem and learn about breadth first search algorithm using c++ language.Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.**

***Index Terms*—Nodes,Edges,source,graph,queue,visited, IDE codeblocks.**

I. Introduction

The breadth-first algorithm is a particular graph-search algorithm that can be applied to solve a variety of problems such as finding all the vertices reachable from a given vertex, finding if an undirected graph is connected, finding (in an unweighted graph) the shortest path from a given vertex to all other vertices, determining if a graph is bipartite, bounding the diameter of an undirected graph, partitioning graphs, and as a subroutine for finding the maximum flow in a flow network (using Ford-Fulkerson's algorithm). As with the other graph searches, BFS can be applied to both directed and undirected graphs.

II. Literature Review

Breadth-First Search is an important kernel used by many graph-processing applications. In many of these emerging applications of BFS, such as analyzing social networks, the input graphs are low-diameter and scale-free. We propose a hybrid approach that is advantageous for low-diameter graphs, which combines a conventional top-down algorithm along with a novel bottom-up algorithm. The bottom-up algorithm can dramatically reduce the number of edges examined, which in turn accelerates the search as a whole. On a multi-socket server, our hybrid approach demonstrates speedups of 3.3 – 7.8 on a range of standard synthetic graphs and speedups of 2.4 – 4.6 on graphs from real social networks when compared to a strong baseline. We also typically double the performance of prior leading shared memory (multicore and GPU) implementations.

III. Proposed Methodology

A standard BFS implementation puts each vertex of the graph into one of two categories:

1.Visited
2.Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1.Start by putting any one of the graph's vertices at the back of a queue.
2.Take the front item of the queue and add it to the visited list.
3.Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4.Keep repeating steps 2 and 3 until the queue is empty.
The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node.

A. *Inputs:*

Firstly input the number of edges and number of nodes use in the graph.Then inputs are a graph(directed or undirected) G =(V, E) and a source vertex s, where s is an element of V. The adjacency list representation of a graph is used in this analysis.

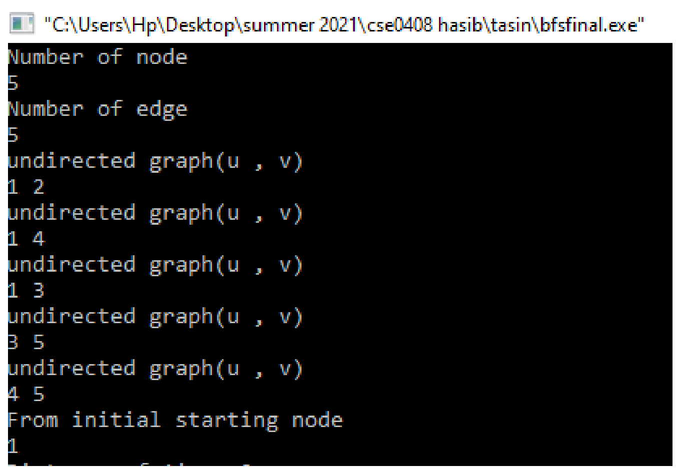

Fig. 1. input format

B. *Outputs:*

The outputs are a predecessor graph, which represents the paths travelled in the BFS traversal, and a collection of distances, which represent the distance of each of the vertices from the source vertex.

```
"C:\Users\Hp\Desktop\summer 2021\cse0408 hasib\tasin\bfsfinal.exe"
Number of node
5
Number of edge
5
undirected graph(u , v)
1 2
undirected graph(u , v)
1 4
undirected graph(u , v)
1 3
undirected graph(u , v)
3 5
undirected graph(u , v)
4 5
From initial starting node
1
Distance of 1is : 0
Distance of 2is : 1
Distance of 3is : 1
Distance of 4is : 1
Distance of 5is : 2

Process returned 0 (0x0)   execution time : 99.196 s
Press any key to continue.
```

Fig. 2. output format

```cpp
#include<bits/stdc++.h>
using namespace std;

#define MX 110

vector  < int > graph[110];
bool vis[MX];
int dist[MX];

void bfs(int source)
{
    queue < int > Q;
    vis[source]=1;
    dist[source]=0;
    Q.push(source);

    while(!Q.empty())
    {
        int node =Q.front();
        Q.pop();

        for (int i=0; i < graph[node].size(); i++)
        {
            int next = graph [node][i];
            if(vis[next]==0)
            {
                vis[next]=1;
                dist[next]=dist[node]+1;
                Q.push(next);

            }
        }

    }


}
int main()
{
    int nodes,edges;
    cout << "Number of node " << endl;
    cin >> nodes;
    cout << "Number of edge " << endl;
    cin >> edges ;
    for (int i=1; i<=edges; i++)
    {
        int u,v;
        cout << "undirected graph(u , v) " << endl;
        cin>> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);

    }
    int source ;
    cout << "From initial starting node " << endl;
    cin >> source ;
    bfs(source);
    for (int i=1; i<=nodes ; i++)
    {
        cout << "Distance of " << i << "is : " <<dist[i] << endl;

    }
    return 0;
}

```

Fig. 3. code

ACKNOWLEDGMENT

I would like to thank my honourable**Khan Md. Hasib Sir** for his time, generosity and critical insights into this project.

REFERENCES

[1] S. Beamer, K. Asanovic and D. Patterson, ”Direction-optimizing Breadth-First Search,” SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1-10, doi: 10.1109/SC.2012.50.

[2]