**TECHNICAL UNIVERSITY OF MOLDOVA**

REPORT

Eratosthenes Sieve

**DISCIPLINE:** Caclulability and Complexity

**Student :** Ionasco Gheorghe

**Group :** FAF-193

Chișinău 2021

## Breadth-First Search, Depth-First Search

The Time complexity of both BFS and DFS will be O(V + E), where V is the number of vertices, and E is the number of Edges. This again depends on the data strucure that we user to represent the graph. If it is an adjacency matrix, it will be O(V^2) . If we use an adjacency list, it will be O(V+E). Now , to explain how, lets us consider the difference between a sparsely connected graph and a densely connected graph.

Dense graph is a graph in which the number of edges is close to the maximal number of edges. Sparse graph is a graph in which the number of edges is close to the minimal number of edges. Sparse graph can be disconnected.

Now, what this means is that, if the graph has so many edges and is still a small graph with only few vertices(for example, a graph with 100 vertices but with 4950 edges since it is totally connected( n\*(n-1)/2 ), then the number of iterations or recursive calls is dominated by the number of edges, because 4950 > 100. Now as n -> very large , this difference matters. This kind of graph is a dense graph. Time complexity will be O(E).

So, what does it mean by O(V + E) ? It means , whichever term is bigger will dominate the time complexity. That is why the time complexity of BFS is O(V+E).

**DFS(analysis):**

* Setting/getting a vertex/edge label takes O(1) time
* Each vertex is labeled twice
  + once as UNEXPLORED
  + once as VISITED
* Each edge is labeled twice
  + once as UNEXPLORED
  + once as DISCOVERY or BACK
* Method incidentEdges is called once for each vertex
* DFS runs in O(n + m) time provided the graph is represented by the adjacency list structure
* Recall that Σv deg(v) = 2m

**BFS(analysis):**

* Setting/getting a vertex/edge label takes O(1) time
* Each vertex is labeled twice
  + once as UNEXPLORED
  + once as VISITED
* Each edge is labeled twice
  + once as UNEXPLORED
  + once as DISCOVERY or CROSS
* Each vertex is inserted once into a sequence Li
* Method incidentEdges is called once for each vertex
* BFS runs in O(n + m) time provided the graph is represented by the adjacency list structure
* Recall that Σv deg(v) = 2m

**BFS**

#include <time.h>

#include<iostream>

#include <list>

using namespace std;

// This class represents a directed graph using

// adjacency list representation

class Graph

{

int V; // No. of vertices

// Pointer to an array containing adjacency

// lists

list<int> \*adj;

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// prints BFS traversal from a given source s

void BFS(int s);

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::BFS(int s)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

for(int i = 0; i < V; i++)

visited[i] = false;

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

// 'i' will be used to get all adjacent

// vertices of a vertex

list<int>::iterator i;

while(!queue.empty())

{

// Dequeue a vertex from queue and print it

s = queue.front();

cout << s << " ";

queue.pop\_front();

// Get all adjacent vertices of the dequeued

// vertex s. If a adjacent has not been visited,

// then mark it visited and enqueue it

for (i = adj[s].begin(); i != adj[s].end(); ++i)

{

if (!visited[\*i])

{

visited[\*i] = true;

queue.push\_back(\*i);

}

}

}

}

// Driver program to test methods of graph class

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Breadth First Traversal "

<< "(starting from vertex 2) \n";

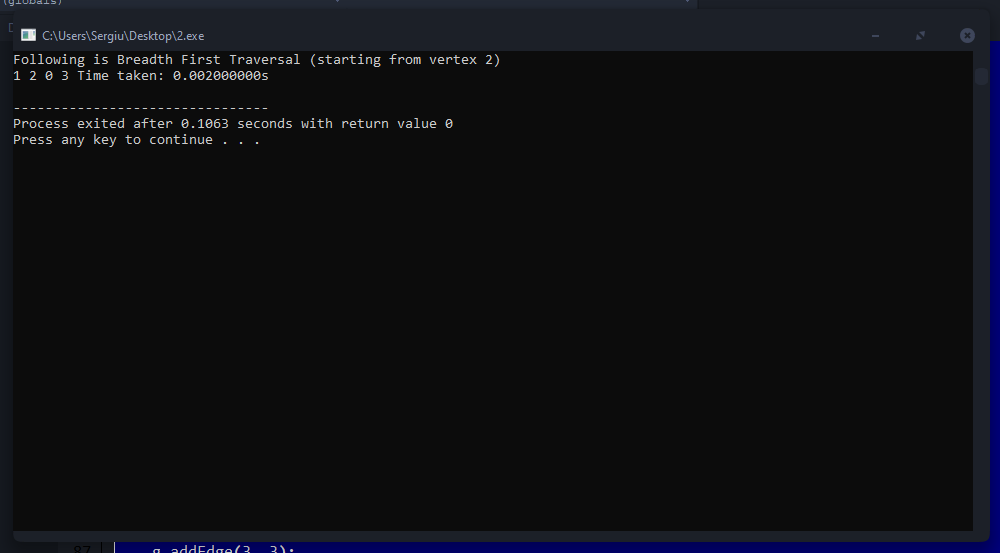
clock\_t tStart = clock();

g.BFS(1);

printf("Time taken: %.9fs\n", (double)(clock() - tStart)/CLOCKS\_PER\_SEC);

return 0;

}



**DFS**

#include <time.h>

#include <bits/stdc++.h>

using namespace std;

// Graph class represents a directed graph

// using adjacency list representation

class Graph

{

public:

map<int, bool> visited;

map<int, list<int> > adj;

// function to add an edge to graph

void addEdge(int v, int w);

// DFS traversal of the vertices

// reachable from v

void DFS(int v);

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::DFS(int v)

{

// Mark the current node as visited and

// print it

visited[v] = true;

cout << v << " ";

// Recur for all the vertices adjacent

// to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFS(\*i);

}

// Driver code

int main()

{

// Create a graph given in the above diagram

Graph g;

g.addEdge(0, 1);

g.addEdge(0, 9);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(9, 3);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Depth First Traversal"

" (starting from vertex 2) \n";

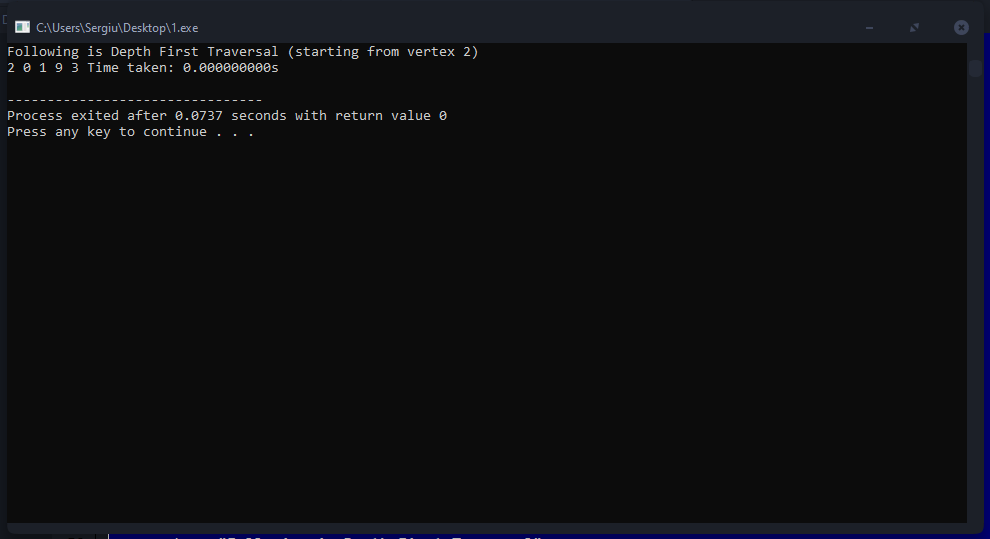
clock\_t tStart = clock();

g.DFS(2);

printf("Time taken: %.9fs\n", (double)(clock() - tStart)/CLOCKS\_PER\_SEC);

return 0;

}



**Conclusion**

BFS uses Queue to find the shortest path. DFS uses Stack to find the shortest path. ... Time Complexity of BFS = O(V+E) where V is vertices and E is edges. Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

So, the conclusion is that both algorithms have same time comlpexity in average but there are some diferences that depends on memory use of algorithms.