**NATIONAL INSTITUTE OF TECHNOLOGY**

**RAIPUR, CHATTISGARH**



***Term Report on Representation of Graphs and Implementation of Depth First Search Traversal***

Submitted To– Submitted By -

Dr. Govind Prasad Gupta sir Name - Shruti Rawal

Branch – Information Technology

Semester - Fifth

Roll Number – 18118075

ACKNOWLEDGEMENT

I would like to use this opportunity to thank all those who provided me their bit of much-needed support during the entire process of preparation of this project.

First of all, I would like to extend my gratitude for my teacher Dr. G.P. Gupta sir whose continuous guidance even during these tough times of Covid19 has helped me to complete the required study on the topic. His unwavering determination towards stimulating suggestions for improvements gave me the required motivation for completing this report.

Next, I would like to thank my parents who have helped me with all their means to assemble all the necessary resources for the completion of this project.

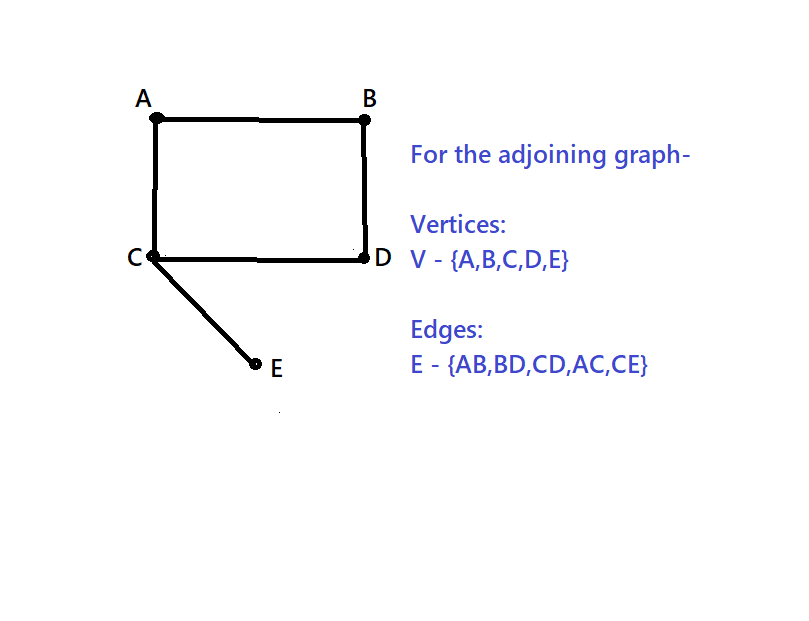
Last but not the least, I would like to thank my batchmates for being so supportive and creating a healthy workspace where all of us could learn and grow. They helped me in assembling the missing pieces of the picture, thereby making it more beautiful, complete and elegant.

* Shruti Rawal

Graphs

Introduction:

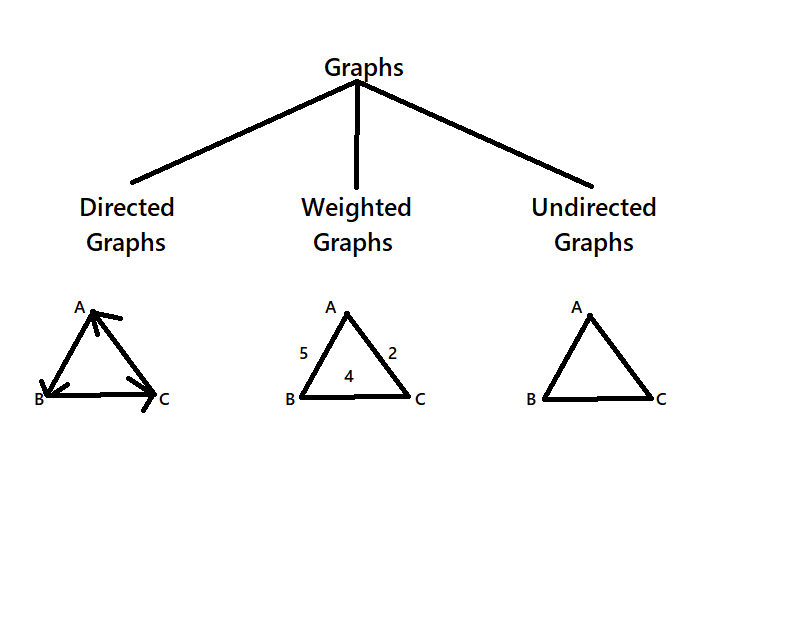
Graphs are an abstract data-type used to represent the pairwise relationship between objects of similar nature. A graph consists of a finite number of vertices (also referred to as nodes) connected by a specified set of pairs of vertices (called edges). Edges can be uni-directional or bi-directional in nature.



Graphs are mathematical structures, hence it’s node structure can be modified to fit to user’s need.

Types of Graphs:

Graphs are broadly classified into the following three types –



1. Directed Graphs – Directed graphs have edges which point in a certain direction i.e. its edges are uni-directional in nature.

2. Undirected Graphs – Undirected graphs have edges which do not specify direction of movement i.e. its edges are bi-directional in nature.

3. Weighted Graphs – Weighted graphs have edges that have a weight assigned to them which is incurred while traversing the graph.

Trees are a special case of graphs in which there are N vertices connected by N-1 edges such that no cycles or loops are present in the structure.

Representation of Graphs:

Representing graphs is a challenging task for all beginners. On a major note, graphs can be represented via following three structures in practice –

1. Adjacency List

2. Adjacency Matrix

3. Incidence Matrix

These representations were a result of continuous efforts for improvement in performance by programmers all over the globe. Let us first have a brief look on each of these representations –

1. Adjacency List – A linked list is used for storing vertices as key entries. Every vertex has another of set of vertices corresponding to itself that consists of all the vertices that are adjacent to the key vertex. This representation saves a lot of space but is a bit difficult to follow and visualize.

2. Adjacency Matrix – A two-dimensional matrix is used for storing the source and destination vertices. The rows represent the source vertex and the column represent destination ones. This representation is very intuitive and easy to understand but it turns out to be very slow while performing operations on graphs.

3. Incidence Matrix – A two-dimensional boolean matrix is used for indicating whether the vertex is incident to the edge. The rows represent the vertices and columns represent the edges of the graph.

In all these representations, the use of adjacency list is always overpowered over the use of adjacency matrix because, in case of adjacency matrix, most of the space gets wasted as many of the entries of the matrix is zero. In such a case, adjacency list representation performs better and saves a lot of space for the programmer to proceed with.

Adjacency Matrix:

In this representation, a two-dimensional matrix is used for storing the source and destination vertices. The rows represent the source vertex and the column represent destination ones. This representation is very intuitive and easy to understand but it turns out to be very slow while performing operations on graphs.

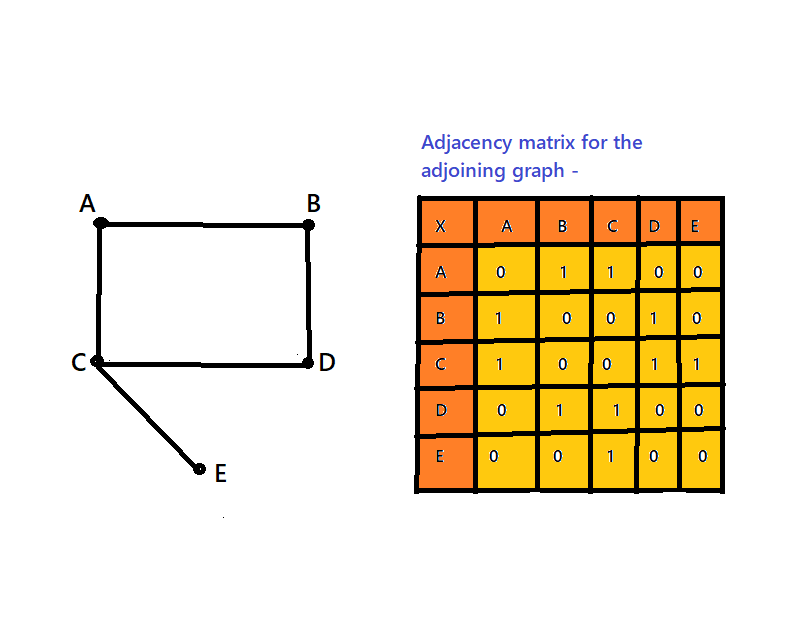
This can be used for representing all types of graphs.

Consider a two-dimensional matrix adj[][] that is used for representing graphs, then-

i) For an undirected/directed graph – adj[i][j] = 1, means there is an edge from vertex i to vertex j.

ii) For a weighted graph – adj[i][j] = c, means there is an edge between vertex i and vertex j having an associated weight c corresponding to it.

Consider the following example –



As the given graph was an undirected graph, therefore the associated matrix is symmetrical in nature.

Advantages –

i) Easier to implement and understand

Disadvantages –

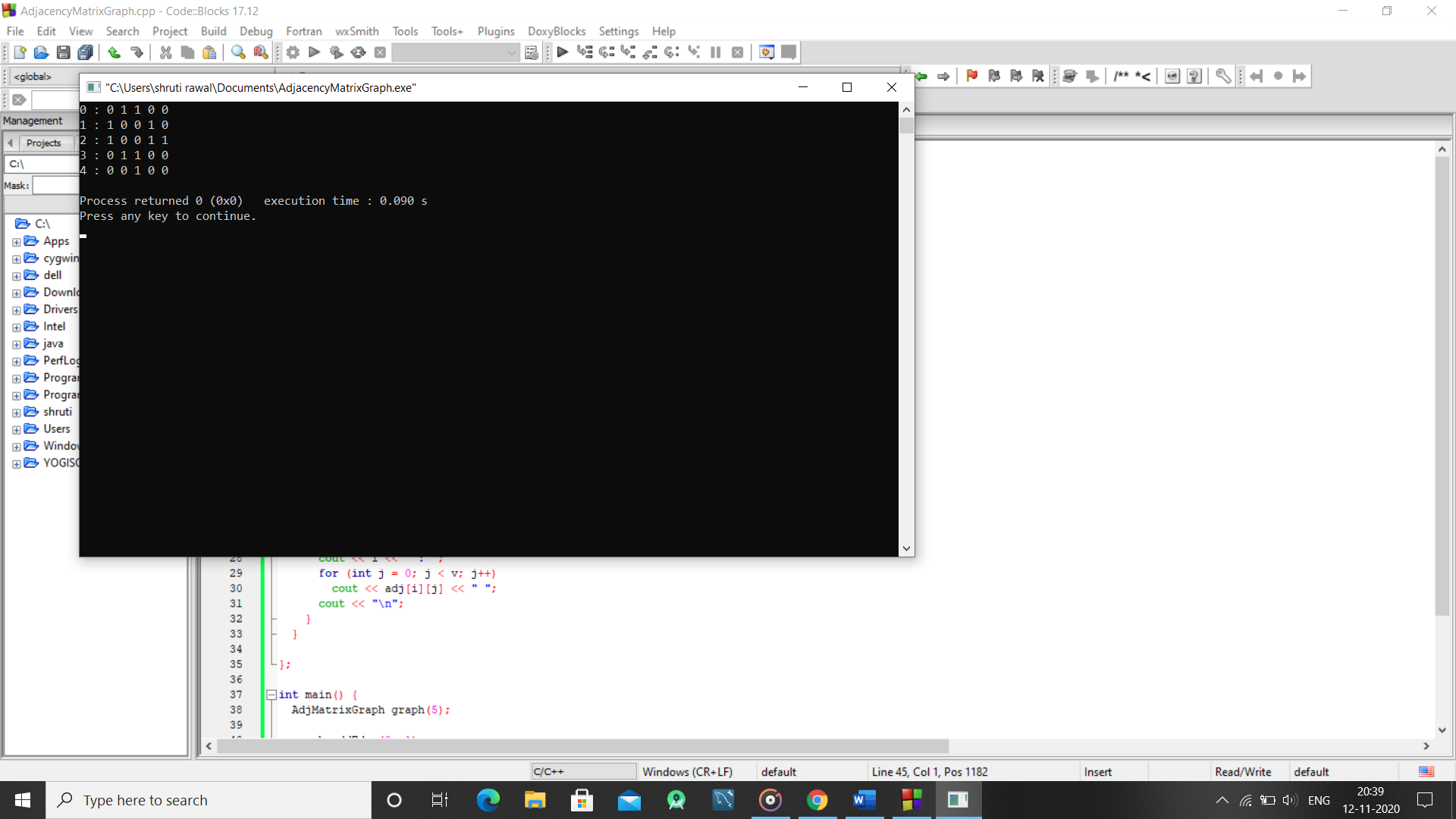
i) Takes up more space than adjacency list representation.

ii) All major graph operations are slow in terms of time complexities when implemented using adjacency matrix representation.

Implementation –

<https://github.com/ShrutiRawal/DAATermPaper/blob/main/Source%20Code/AdjacencyMatrixGraph.cpp>

Working Screenshot –



The source code shown above implements the graph taken into consideration as example. The code takes note of the unidirectional property of the graph. However, to remove this property, programmers are requested to remove line 23 of the given source code.

Adjacency List:

In this representation, a linked list is used for storing vertices as key entries. Every vertex has another of set of vertices corresponding to itself that consists of all the vertices that are adjacent to the key vertex. This representation saves a lot of space but is a bit difficult to follow and visualize.

For a weighted graph, the cost of travelling from the key vertex to the adjacent vertex can also be stored in the adjacency list in the form of pairs.

This can be used for storing all sorts of graphs with improved performance over adjacency matrix in terms of graph operations.

Advantages-

i) Saves more space than adjacency matrix for smaller number of edges.

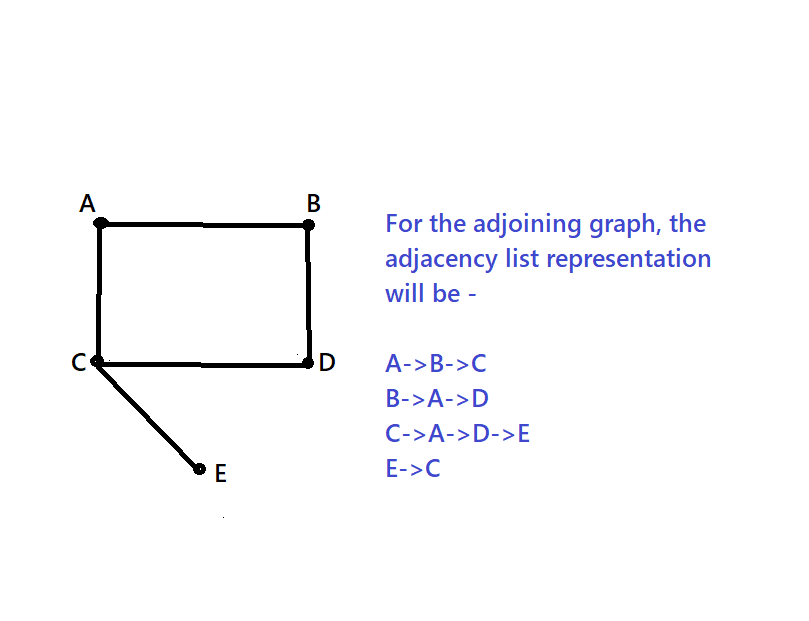
ii) Due to less space consumption, operations like adding and removing a vertex is very easy.

Disadvantages –

i) In case the graph is large enough and has a lot of edges, implementation becomes more cumbersome and inefficient in terms of working with operations on graphs.

ii) Certain queries like whether a path exists between any pair of vertices is not very efficient as it has to be searched in the entire list just to be sure.

Consider the following example –

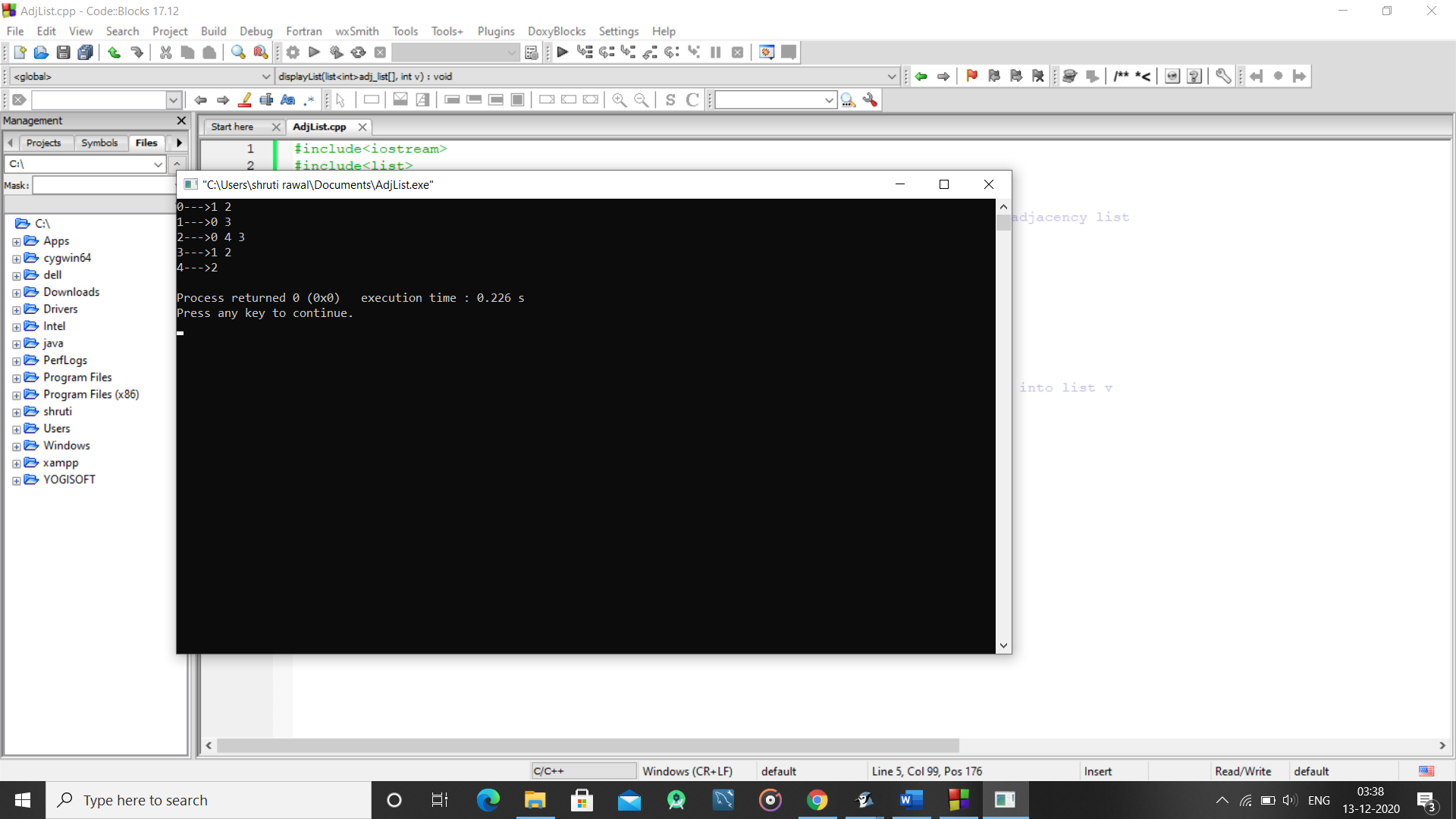


This graph is an undirected graph, but if the graph was a weighted one, then instead from just vertices, the elements of the linked list would have been a custom pair having first element as vertex name and second element as the cost associated with travelling to that vertex from the key vertex.

Implementation –

<https://github.com/ShrutiRawal/DAATermPaper/blob/main/Source%20Code/AdjList.cpp>

Working Screenshot –



Operations on graphs and their costs:

Following is a list of major operations and their costs for different representations of graphs –

|  |  |  |  |
| --- | --- | --- | --- |
| S. No | Operation | Adjacency List | Adjacency Matrix |
| 1 | Store graph | O(V+E) | O(V\*V) |
| 2 | Add a vertex to the graph | O(1) | O(V\*V) |
| 3 | Remove a vertex from the graph | O(E) | O(V\*V) |
| 4 | Add an edge to the graph | O(1) | O(1) |
| 5 | Remove an edge from the graph | O(V) | O(1) |
| 6 | Does edge exist between two vertices? | O(V) | O(1) |

Here, V denotes the number of vertices in the graph and E denotes the number of edges in the graph. The time complexities shown is the asymptotic worst-case time complexity for the operation specified.

Applications of graphs:

Graphs find a variety of applications in the technological world today –

In operating system, resource allocation graphs employ the usage of graphs. Social networking sites like Facebook use graphs for their specialized algorithms of friend suggestion. Google Maps also employ the usage of graphs in creating and storing navigation results.

Graph Traversal:

The goal of graph traversal is to visit all the nodes of the graph in some fashion such that all of the nodes is visited only once. There are two major graph traversal algorithms –

i) Breadth First Search (BFS) – In this algorithm, we start from a root node and then traverse all of its adjacent nodes and proceed in this manner towards the end. As the name suggests, using this approach the graph is traversed via it’s breadth first.

Time Complexity – O(V+E) where V denotes the number of vertices and E denotes the number of edges.

ii) Depth First Search (DFS) – In this algorithm, we start from a root node and then traverse the graph in a manner that covers the graph depth-first. We use a stack for this algorithm. After visiting a root node, its adjacent vertices are pushed into a stack and the next vertex to be traversed is popped out from the stack.

Time Complexity – O(V+E) where V denotes the number of vertices and E denotes the number of edges.

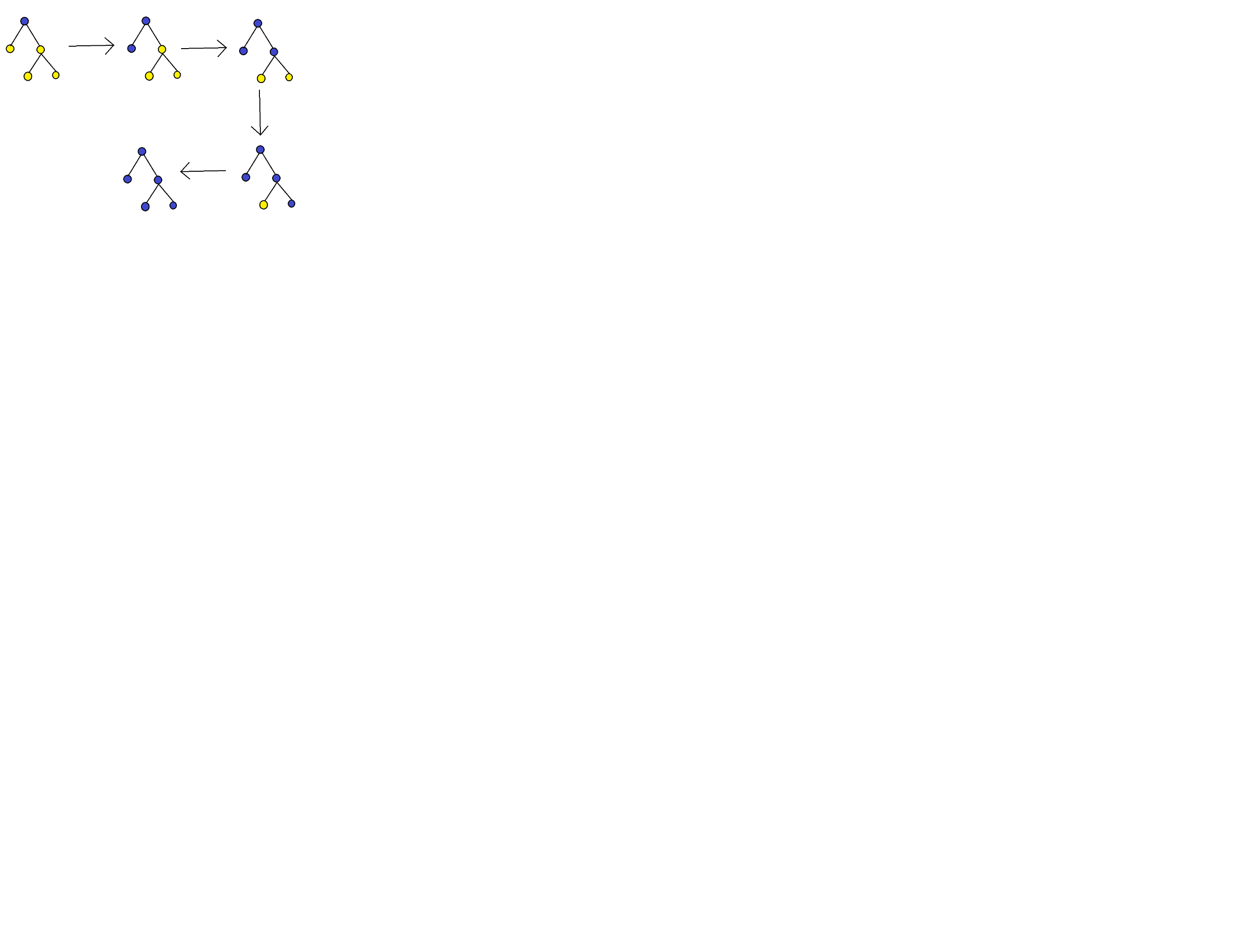
Depth First Search (DFS):

In this algorithm, we start from a root node and then traverse the graph in a manner that covers the graph depth-first. We use a stack for this algorithm. After visiting a root node, its adjacent vertices are pushed into a stack and the next vertex to be traversed is popped out from the stack.

Video Animation link –

<https://drive.google.com/file/d/17qEW2xaa5yWOIy_nQqtntgYM7rnxGerE/view?usp=sharing>

This algorithm uses the idea of backtracking. The recursive approach is designed such that all the nodes are visited only once, thereby removing the possibility of the program getting stuck in an infinite loop. Consider the following example –



The nodes which are marked as blue have been visited by the algorithm and the ones marked yellow are yet to be visited.

**Applications of DFS –**

Following are the major applications of Depth First Search Traversal algorithm –

i) Detecting loops in a graph

ii) Topological Sorting algorithm for graph

iii) Finding connected components of a graph

iv) To find whether a given graph is a bipartite graph

v) To find the path between two given nodes in the graph

For implementing depth-first search traversal technique, there are two paradigms

1. Recursive

2. Iterative

**Implementation-**

Pseudocode -

DFS (G, s) //G is the given graph and s is the starting node

mark s as visited

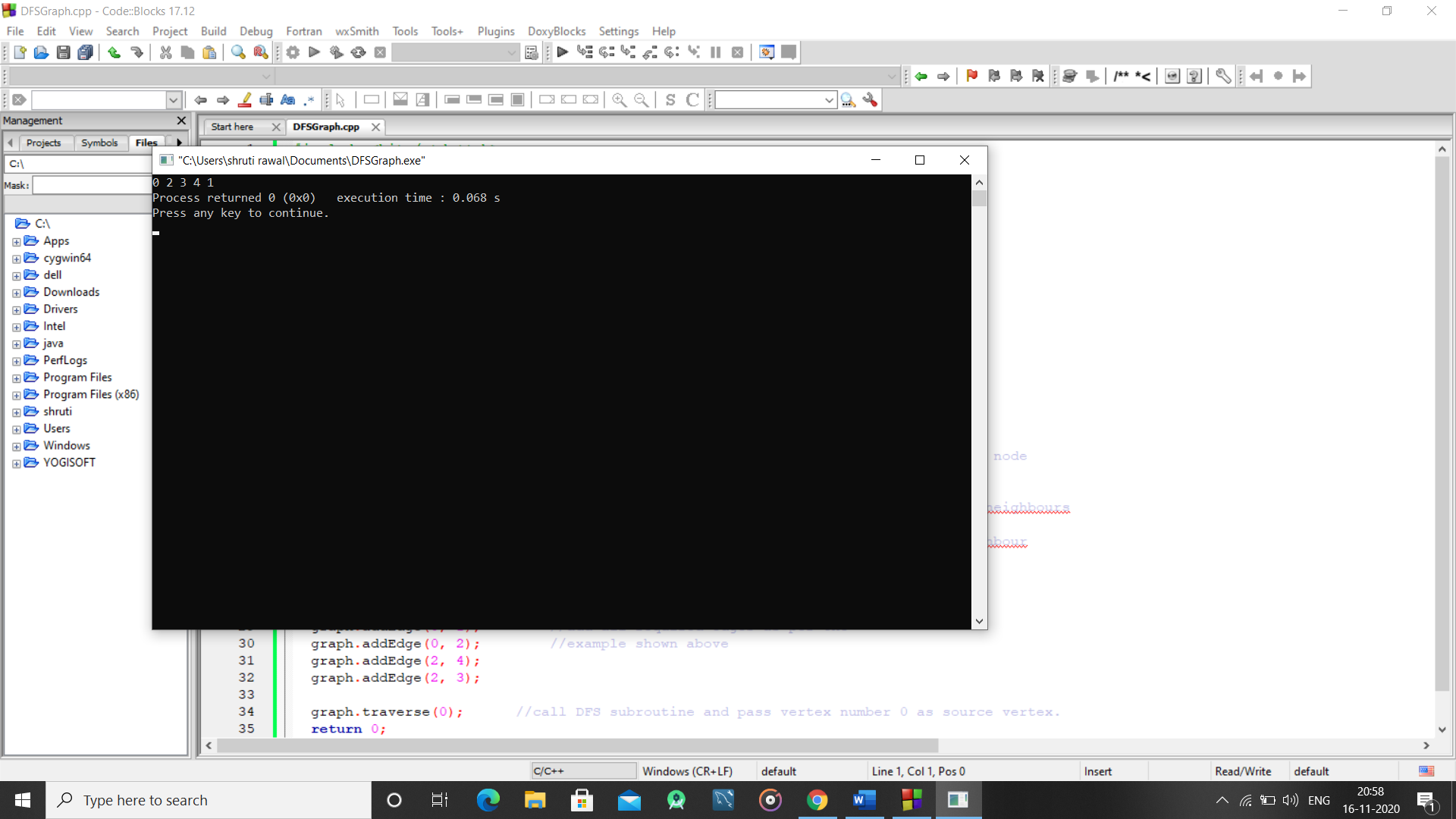
for all adjacent nodes of start node s in graph G

if neighbour is not visited

DFS (G, neighbour of s)

Source Code -

Source code link - <https://github.com/ShrutiRawal/DAATermPaper/blob/main/Source%20Code/DFSGraph.cpp>

Working Screenshot – 

Time Complexity – O(V+E) where v denotes the number of vertices in the graph and E denotes the number of edges in the graph.