DEMO OF 5 GRAPH THEORY TOPICS IN AN OFFLINE OPENGL APP

Ashish Jacob Sam (M21CS003)

IIT Jodhpur

ABSTRACT

This project will be provide visual demonstration of 5 graph theory algorithms. A user can use to create a custom graph (with less than 20 vertices, but no limits in edges), move the vertices manually to arrange the graph, choose to keep the graph directed/undirected and weighted/unweighted and edit the edges in the graph. After creating the graph, the user can then see the visualized workings of the following algorithms on their custom graph: 1) Depth First Search, 2) Breadth First Search, 3) Dijkstra Algorithm, 4) Floyd-Warshall's All pairs shortest path algorithm, and 5) Kruskal Minimum Spanning Tree algorithm. For each algorithm, there will be a step-by-step visualization provided, indicated by colouring of the vertices or the edges. An exception is in case of Floyd-Warshall, where instead of the graph, the matrix would be shown being edited for every step.

Index Terms— Demo;Visualizer;Graph Theory;Depth-First Search;Breadth-First Search;Dijkstra; Floyd-Warshall; Kruskal Minimum Spanning Tree;

1. INTRODUCTION

Graph theory deals with the graph structure which is defined as a set of vertices(or nodes) and edges(links between two nodes). 1 is a simple example of graph. This graph contains 5 vertices namely $\{A, B, C, D, E\}$ and 6 edges, namely $\{(AB), (BC), (CD), (DE), (AE), (EC)\}$

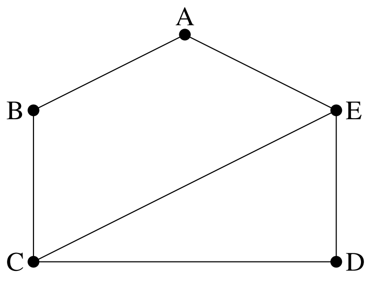


Fig. 1. Undirected, unweighted graph G_1

1 is an example of a undirected and unweighted graph. An undirected graph has edges as an unordered pair of vertices, meaning that the edge (A, B) is equivalent to edge (B, A). A directed graph has edges as ordered pair of vertices. It would be represented as shown in 2

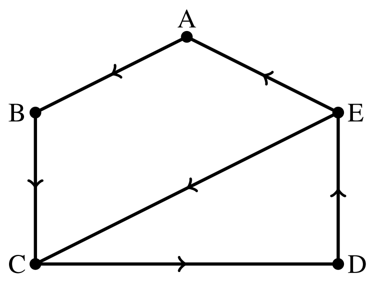


Fig. 2. Directed, unweighted graph G_2

Furthermore, graphs can be weighted. Both the graphs 1 and 2 were unweighted. 3 is an example of an undirected and weighted graph, while 4 is an example of an directed, weighted graph.

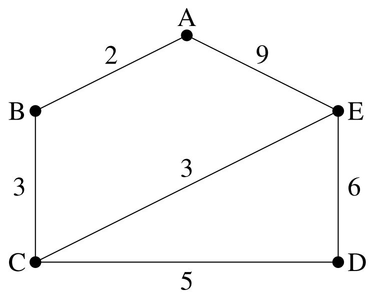


Fig. 3. Undirected, weighted graph G_3

Notice that for all these graphs, The vertex set are the same. The edge set is ordered in the case of directed graph, and unordered in case of undirected graph. Furthermore, the edge set is a tuple of pair of vertices in unweighted graph, and in a weighted graph the edge set is a tuple of the pair of vertices along with the weights associated with it.

Graphs are simple yet powerful structure to provide an abstraction to many real-life problems, like constructing social network, biological network, road/transportation network, block-chain, knowledge representation, etc.

For such abstractions, there are many common problems that we would wish to solve. This document details

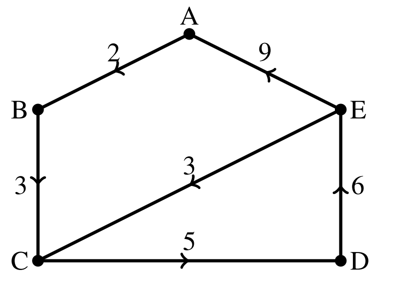


Fig. 4. Directed, weighted graph G_4

the project of a visualizer using OpenGL to draw user defined custom graphs and visualize the working of some basic algorithms within the application.

The five algorithms are

1. Depth First Search (DFS)
2. Breadth First Search (BFS)
3. Dijkstra's Algorithm of Single source Shortest path search
4. Floyd-Warshall Algorithm, or All-pairs-shortest path search
5. Kruskal's Algorithm for Minimum Spanning Tree

Details of these algorithms are described in the later sections

2. DEFINITIONS

Here we first mathematically define a graph. A graph is defined as $G = (V, E)$, where V is the set of vertices and E is a set of Edges.

For describing all the edges of a graph, we use the adjacency matrix representation. The adjacency matrix A has elements $a_{ij} = 0$ if there does not exist any edge between the vertices (i, j). Otherwise the weight of the edge is stored in the element a_{ij}.

For unweighted graphs, the default weight can be taken as any real non-negative number for the purpose of representation.

Using the adjacency matrix representation, the graphs G_1, G_2, G_3, G_4 can be represented as follows

$$A_{G_1} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

$$A_{G_2} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$A_{G_3} = \begin{bmatrix} 0 & 2 & 0 & 0 & 9 \\ 2 & 0 & 3 & 0 & 0 \\ 0 & 3 & 0 & 5 & 3 \\ 0 & 0 & 5 & 0 & 6 \\ 9 & 0 & 3 & 6 & 0 \end{bmatrix}$$

$$A_{G_4} = \begin{bmatrix} 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 6 \\ 9 & 0 & 3 & 0 & 0 \end{bmatrix}$$

Notice that for unweighted graph, the default weight associated with edges is 1, and for undirected graphs, the adjacency matrix is symmetric.

Also note that while every undirected graph can be represented as a directed graph(by defining edges such that $\forall (a, b) \in E \implies (b, a) \in E$), the converse is not true. Similarly every unweighted graph can be represented as a weighted graph (by assigning a default edge weight to all edges), but the converse is not true.

3. ALGORITHMS ON GRAPHS

In this section, the implementation is provided for all the algorithms in terms of pseudocode. The implementations of these algorithms are referred from the textbook written by Heineman et al. [1]

3.1. Single source path search

The first 3 algorithms can be classified as single source search algorithms. These algorithm start at a specified source, search for a particular destination, and return the distance(s) or path found.

3.1.1. Depth First Search

A very basic algorithm for exploring a graph is the Depth First Search (DFS) algorithm. This algorithm explores every neighbour of a node and moves ahead (in no particular order) and terminates when either no more node can be explored, or when the destination node is found. The algorithm is given in 1

Notice that a single LIFO stack is used to process all the vertices. The algorithm finds a node, discovers all its neighbours, select a node, and continues this search until no more nodes can be found. While this means DFS can be used for a general search (that is, to merely check for the existence of a

Algorithm 1 DFS algorithm

```
procedure DFS(G, src, dest)
Require: $G$ graph, $src$ source, $dest$ Destination
    for each $v \in V(G)$ do
        $pred[v] \leftarrow -1$
        $visited[v] \leftarrow 0$
    end for
    stack.push(src)
    while $stack$ is not empty do
        $u = stack.Pop()$
        if $u$ is $dest$ then
            break;
        else
            for each $(u, x) \in G$ do
                if $visited[x]$ is 0 then
                    $pred[x] \leftarrow u$
                    $visited[x] \leftarrow 1$
                    stack.push(x)
                end if
            end for
        end if
    end while
return $(pred, cost)$
end procedure
```

Algorithm 2 BFS algorithm

```
procedure BFS(G, src, dest)
Require: $G$ graph, $src$ source, $dest$ Destination
    for each $v \in V(G)$ do
        $pred[v] \leftarrow -1$
        $visited[v] \leftarrow 0$
    end for
    queue.push(src)
    while $queue$ is not empty do
        $u = queue.Pop()$
        if $u$ is $dest$ then
            break;
        else
            for each $(u, x) \in G$ do
                if $visited[x]$ is 0 then
                    $pred[x] \leftarrow u$
                    $visited[x] \leftarrow 1$
                    queue.push(x)
                end if
            end for
        end if
    end while
return $(pred, cost)$
end procedure
```

path to the destination node), the path it traverses can be much longer than the actual shortest path between the vertices. This approach can be improved by maintaining an order in which all vertices are explored.

3.1.2. Breadth First Search

Breadth First Search (BFS) is very similar to DFS, but with a subtle difference in the order of the nodes explored. This is done by using a FIFO queue instead of a LIFO stack. With queue, the nodes are inserted in order of the hops required. This means that all nodes at a distance of k hops away are explored first, before moving on to all nodes at a distance of $(k + 1)$ This algorithm gets a shorter paths from a single source in a graph compared to the DFS, but not necessarily the shortest path.

For an unweighted graph (that is when all edges have the same weight), or a weighted graph containing no cycle, BFS will select the shortest path. For any general graph, it returns the shortest path with fewest hops. The BFS algorithm is given in 2

3.1.3. Dijkstra algorithm

Dijkstra algorithm is a greedy algorithm that further improves on the BFS algorithm by using a priority queue instead of a simple LIFO queue. Each node is assigned the distance covered as a priority. The node with the lowest distance is given the highest priority. This way the destination node is guaranteed to be reached by the shortest path, if a path exists. The algorithm is given in 3

3.2. Floyd-Warshall all pair shortest path search

Simply put, the Floyd-Warshall algorithm finds the shortest path from all vertices to all vertices. It uses the concept of dynamic programming to find and keep track of the shortest distance between all pairs of vertices. It maintains a $n \times n$ matrix to store all distances between edges. It executes multiple passes to compute the shortest path between all pair of vertices. The algorithm is given in 4

3.3. Kruskal Minimum spanning tree

The Kruskal algorithm finds the minimum spanning tree in a given undirected graph. As such it is not defined for a directed graph. It is a greedy algorithm, which sorts all the edges in increasing order of their weights, and selects $|V| - 1$ edges such that no cycles are formed. The algorithm for it is given in 5

4. IMPLEMENTATION

The implementation for this project is done using .NET 6 with C#. For rendering in OpenGL, the MonoGame framework is used. MonoGame is a popular framework for creating games,

Algorithm 3 Dijkstra algorithm

```
procedure DIJKSTRA(G, src, dest)
Require: $G$ graph, $src$ source, $dest$ Destination
    $PQueue \leftarrow \phi$
    for each $v \in V(G)$ do
        $pred[v] \leftarrow -1$
        $PQueue.push(v, \infty)$
    end for
    $queue.SetPriority(src, 0)$
    while $queue$ is not empty do
        $u = queue.GetMin()$
        if $u$ is $dest$ then
            break;
        else
            for each $(u, x) \in G$ do
                $EdgeCost \leftarrow w(u, x)$
                $PrevCost \leftarrow PQueue.Priority(u)$
                $newlen \leftarrow PrevCost + EdgeCost$
                if $newlen < PQueue.Priority(x)$ then
                    $PQueue.SetPriority(x, newlen)$
                    $pred[x] = u$
                end if
            end for
        end if
    end while
    return $(pred, cost)$
end procedure
```

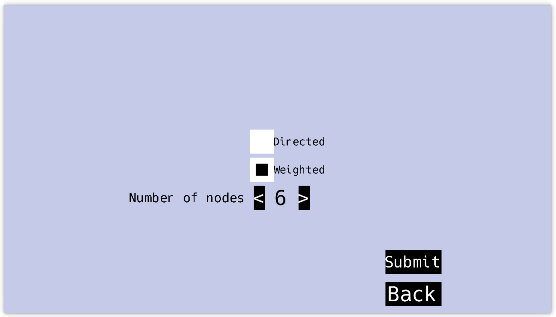
Algorithm 4 Floyd-Warshall algorithm

```
procedure FLOYDWARSHALL(G)
Require: $G$ graph
    for each $u, v \in V(G)$ do
        if $(u, v) \in E(G)$ then
            $dist[u][v] \leftarrow w(u, v)$
        else if $u = v$ then
            $dist[u][v] \leftarrow 0$
        else
            $dist[u][v] \leftarrow \infty$
        end if
    end for
    for each u,v,w in V(G) do
        if u,v,w is not distinct then
            continue;
        end if
        $d_{uv} \leftarrow dist[u][v]$
        $d_{vw} \leftarrow dist[v][w]$
        $d_{uw} \leftarrow dist[u][w]$
        if $d_{uv} + d_{vw} \leq d_{uw}$ then
            $dist[u][w] \leftarrow d_{uv} + d_{vw}$
        end if
    end for
    return dist
end procedure
```

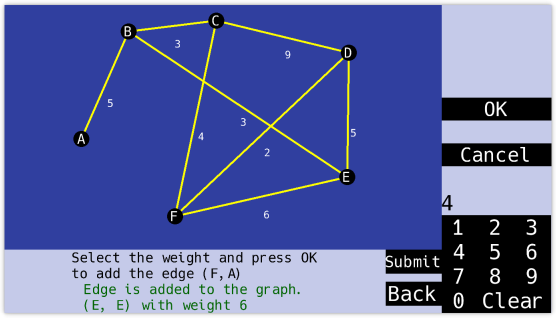
Algorithm 5 Kruskal MST algorithm

```
procedure KRUSKAL(G)
Require: $G$ graph
    for each $e = (u, v) \in E(G)$ do
        $list.add(e, w(e))$
    end for
    $list.sort()$
    $mst \leftarrow$ tree()
    for each edge $e \in$ list do
        if $e$ does not creates cycle in mst then
            mst.add(e)
        end if
    end for
return mst
end procedure
```

and supports a large number of platforms including Windows, Linux, Mac, Android, etc. The project is a Windows/Linux application, which accepts a graph, allows user to drag the nodes to arrange the graph, add/remove the edges, and then select an algorithm to visualize. The user is first asked to enter the graph properties. Here the user should detail if the graph is Directed or not, Weighted or not, and how many Vertices it has.

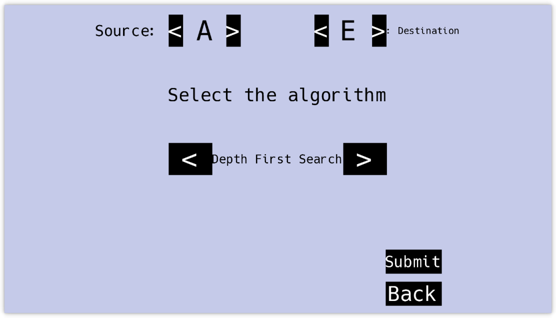


Next, the user must arrange the nodes of the graph as they like it. They can drag/drop the vertices as shown in the screenshot. The user can choose the options they like and click the 'submit' button. Next the user should enter the edges between the graph. The edge is specified by pressing the node (as a button) between which the edge is to be created.

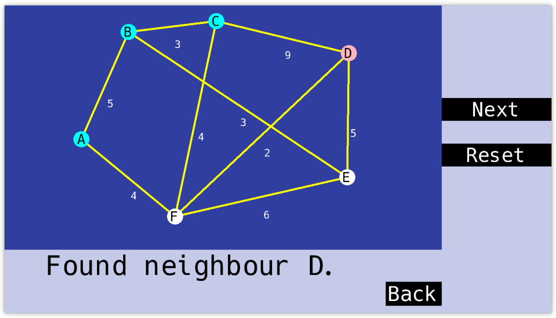


Select the weight and press OK
to add the edge (F, A)
Edge is added to the graph.
(E, E) with weight 6

If the graph is undirected, the pair of edges are inserted with default weight of 1. Otherwise for directed graphs, the source edge and destination edges must be specified in the correct order. For removing an existing edge in a unweighted graph, the edge should be specified again and the removal is done. Otherwise if the graph is weighted, putting the edge weight as 0 is understood by the program for removal of the existing edge. After selecting the edges, the user can go ahead by selecting the 'submit' button.

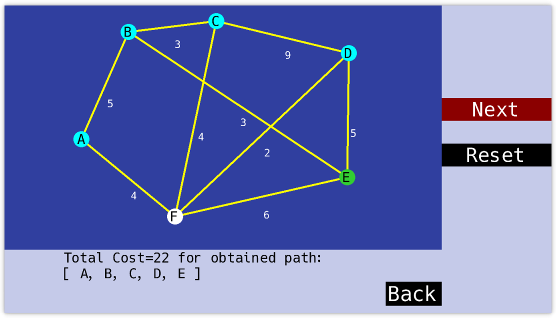


Next, the user should select the algorithm they wish to visualize, and select additional parameters if applicable. As shown in the image, Depth First Search algorithm is being used with the source node A and the destination node as E. It should be noted that for directed graph the Kruskal Algorithm cannot be viewed. As stated in the previous sections, the Minimum spanning tree is defined for only undirected graphs. After pressing 'submit' the visualization for the algorithms can be viewed.



Found neighbour D.

The algorithms are implemented in a sequence of step. For each step, the graph is appropriately coloured and a log text is shown reporting what occurs in each step.



```
Total Cost=22 for obtained path:
[ A, B, C, D, E ]
```

Here, the Depth First Search is complete, and the path and distance of the destination vertex is displayed. For all single-source search algorithms, if the destination is not reached, the search goes on until all vertices are explored. In the case of Dijkstra algorithm for searching a non-reachable vertex, the distance vector for all the reachable vertices are displayed at the end.

When the sequence ends, the 'Next' button is disabled. The user can Reset to view the algorithm from the starting steps again, or the user can go back. For all back pages, all the details of the previous steps are preserved, meaning the user does not need to construct a new graph again before selecting a different algorithm unless they want to.

This project is available on GitHub [2] and can be executed on any desktop platform. It is recommended to download the software from the 'release' page for the appropriate platform and execute it. These details are also mentioned in the Readme file within the repository.

5. SUMMARY

Hence a simple OpenGL application is prepared in this project that allows a user to draw their own custom graphs, select an algorithm, specify source and destination (when applicable) and see a step-by-step working of the algorithm.

6. REFERENCES

[1] George T. Heineman, Gary Pollice, and Stanley Selkow, *Graph Algorithms*, vol. 1, chapter 6, pp. 133–168, O'Reilly, 2 edition, 2016.

[2] Ashish Jacob Sam, "Graph Algorithms Visualizer," `https://github.com/AzuxirenLeadGuy/GTT-Project`, 4 2022.