Implementation of Prim's and Kruskal's MST Algorithm using Graphics

Tarun Kumar
UE143103
Computer Science Engineering
University Institute of Engineering and Technology
Panjab University
Chandigarh, 160012
Email: tarunk.1996@gmail.com

***Abstract*—This practical illustrate about the implementation of the two popular algorithms (i.e. Prim's Algorithm and Kruskal's Algorithm) used for finding the minimum-cost spanning tree of a graph. Graphics libraries used for the implementation of program are explained below. Prim's Algorithm and Kruskal's Algorithm for finding minimum-cost spanning tree were designed and implemented using javascript only.**

I. Introduction

A **graph** is a pictorial representation of a set of vertices and edges. The vertices of a graph are represented by points or a filled circle, and edges are the links connecting the two vertices at a time, represented by a line joining the two vertices. A graph is mathematically represented by

$$G = (V, E)$$

where V is the set of vertices present in graph G and E is the set of edges present in the graph G.

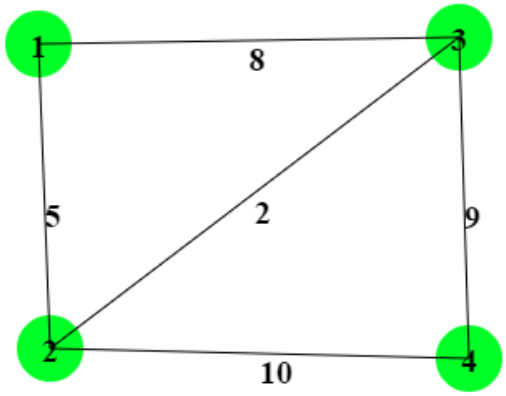

Fig. 1. This graph is generated from the program usedin this project

The above figure 1 has a total of 4 vertices and 5 edges in the graph. The number written over the edge is called the weight of the edge (means the cost to visit from one vertex to another vertex using that path or edge).

Minimum-cost Spanning Tree : A minimum-cost spanning tree (MST) of an edge-weighted graph is a spanning tree whose total weight (the sum of the weights of all the edges) is minimum or least compared to other spanning trees of the graph. There are many algorithms for finding the minimum-cost spanning tree of a graph. But, in this practical only two algorithms are broadly focused and implemented. Following are the two algorithms used for finding the minimum-cost spanning tree of a graph:

A. Prim's Algorithm

It is a greedy approach for finding the Minimum-cost spanning tree of a graph. The spanning tree in this method grows edge by edge. The next edge to be selected is based on some criteria for optimal results. Firstly, a least cost edge is selected from the graph. Afterwards, the edge is selected having the least cost and is connected to the previously visited vertices of the tree. Also, the edge to be selected must have a single vertex attached to it which previously has not being visited by the tree. This selection of next edge is continued $n-2$ number of times because first edge is already selected (the edge with the minimum cost). Here n means the total number of vertices present in the graph and a minimum-cost spanning tree always have $n-1$ edges in it if all the vertices are connected to each other through edges. The growth of the minimum-cost spanning tree is always like extending a single tree. The time complexity of the Prim's Algorithm is $O(n^2)$. Here, n represents the total number of vertices present in the graph.

B. Kruskal's Algorithm

Another approach used for finding the Minimum-cost spanning tree of a graph is Kruskal's Algorithm. For the implementation of Kruskal's method, the cost list is sorted in increasing order. The minimum cost edge is selected at first. After the selection of the first edge the next minimum-cost edge is selected and some conditions are checked. In the conditions, it is checked that after the selection of new edge whether a cycle is formed or not. If the cycle is formed the

edge is dropped from the list and the next edge is selected having the minimum-cost edge present in the cost list. Again the same conditions are checked until no cycle is formation occurs. Once, a new edge is being selected such that no cycle is formed, the edge is deleted from the cost list and edge is highlighted. This process continues till $n-1$ edges are not being highlighted. The formation of minimum-cost spanning tree in kruskal's algorithm is like forest type structure which keeps on merging as new edges are selected and at the end a tree is formed which is minimum-cost spanning tree. The algorithm used in this piratical for implementation of kruskal's algorithm has a time complexity of $O(n^2)$. Here, n represents the total number of vertices in the graph.

II. Aim

To write a program which finds the minimum-cost spanning tree of a graph constructed by user in a graphical user interface environment. For finding minimum-cost spanning tree use both Prim's and Kruskal's Algorithm, compare and contrast both the algorithm. Also, give a detailed description about the graphics libraries and functions used in order to develop the program.

III. Approach

Prim's and Kruskal's both the algorithm are used for finding the minimum-cost spanning tree are implemented using JavaScript and some inbuilt functions present in it are used for creation of vertices and edges in the graph. In the next section a brief description of the functions used are provided. As the program is coded in JavaScript it requires a browser to run perfectly on a system. Whenever a user left clicks on the space provided in the canvas a new node or vertex is created there with some node number associated with it staring with 1 and so on. The created node has some radius and filled with some color in it. The creation of nodes occurs till a right click is not encountered. After the right click, user needs to select two nodes where a edge needs to be inserted and provides the edge weight. Once, all the edges are inserted user can select either from Prim's or Kruskal's Algorthim to find the minimum-cost spanning tree of the graph.

IV. Experiment

To create a graphical user interface in the browser, canvas element is used in HTML. The HTML ¡canvas¿ element is used to draw graphics, via scripting (usually JavaScript). The ¡canvas¿ element is only a container for graphics. You must use a script to actually draw the graphics. Canvas has several methods for drawing paths, boxes, circles, text, and adding images.

```
e.g. < canvas id = "canvas1"
width = "800" height = "700" >
</canvas>
```

This is how to declare a canvas element in a HTML file. Using CSS, it can be styled and given any background color as required.

To read the coordinates where the user has clicked anywhere inside the canvas. It can be achieved using following code:

```
var e = window.event;
var posX = e.clientX;
var posY = e.clientY;
```

Here, $event$ is passed via function argument. $clientX$ and $clientY$ will return the x and y coordinates where the event has occurred. Following are examples of some functions used in the JavaScript to draw different figures in the canvas.

Draw a Line

```
var c=document.getElementById("canvas1");
var ctx = c.getContext("2d");
ctx.moveTo(0,0);
ctx.lineTo(200,100);
ctx.stroke();
```

This code will create a line from coordinates $(0, 0)$ to $(200, 100)$ in the canvas. At the top left corner of the canvas file is given $(0, 0)$ coordinates. On moving downwards y coordinate increases keeping x constant while on moving towards right x increases keeping y constant. This is used when a edge is drawn in the graph by selecting two vertices one after the another. Some other functionalities that could be used while drawing a line for example : coloring a line, setting its with. It can be achieved using following code.

```
ctx.strokeStyle="#d0ff08";
ctx.lineWidth=4;
```

This must be included before closing the stroke. In this program, this is used when a edge is needed to be highlighted such that it is selected for minimum-cost spanning tree.

Draw a Circle

```
var can = document.getElementById('canvas1');
var context = can.getContext('2d');
context.fillStyle = "#00ff27";
context.beginPath();
var radius = 20;
context.arc(posX, posY, radius,0, Math.PI*2);
context.closePath();
context.fill();
```

Above code is used to draw circles in the canvas of radius 20 and center positioning at coordinates $(posX, posY)$. Also, the circle is filled with some color to make it visible or glow in the canvas. This is used to create a new node or Vertex when user left clicks anywhere in the canvas. A vertex number is also associated with the vertex and needs to display it while creation of the vertex. The coordinates of the vertices are stored in an array containing the node number, and it's x,y coordinates.

Draw a Text

```
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.font = "30px Arial";
ctx.fillText("Hello World",10,50);
```

This is used to write text inside the canvas element using JavaScript. In fillText function a string and the x,y coordinates are passed to the function where the string will be displayed.

The above 3 built-in functions are generally used in the program to create a graphical user interface in the browser window where file is opened. Apart from this basic functionalities many user defined functions are also defined for proper functioning of the program. Some of them are listed below.

To chcek no two nodes are together

```
for(i=1;i<n;i++)
{
if((edges[i][0]+100)>posX &&
(edges[i][0]-100)<posX)
{
 if((edges[i][1]-100)<posY &&
 (edges[i][1]+100)>posY)
  {
  alert("Two vertices can't be together!!");
  return ;
  }
 }
}
```

In the above code, it will check whether the current event performed for creating a node is not near to the previously created nodes. So it will check through all the previously created nodes and verifies that new node is at a suitable distance from the previous node. If the new node is near about any previous node a alert window is show displaying "Two vertices can't be together!". Otherwise, new node will be formed and the coordinates of the newly generated node is inserted in the list edges storing all the coordinates of nodes generated so far. This function has a time complexity of $O(n)$ as all the vertex are checked one after the another.

To select the vertex while creating edges

```
var e = window.event;
var posX = e.clientX;
var posY = e.clientY;
while(flag&&i<edges.length)
{
 if((edges[i][0]+25)>posX &&
  (edges[i][0]-25)<posX)
 if((edges[i][1]-25)<posY &&
  (edges[i][1]+25)>posY)
 {
  posX = edges[i][0];
  posY = edges[i][1];
  flag = 0;
 }
 i++;
}
```

This code is used while creating edges. As a user can never always selects the center of the nodes everytime while creating an edge. To resolve the problem, this code will automatically assigns the center of the node to the x,y coordinates even when user click anywhere near about that node. This function has a time complexity of $O(n)$ as all the nodes are checked using brute-force.

V. Algorithms and Implementation

A. Prim's Function

```
function prims()
{
 var i,j;
 mincost=0;
 var near=[];
 var min=1000000;
 var k,l;
 for(i=1;i<n;i++)
 {
  for(j=1;j<n;j++)
    if(ew[i][j]!= undefined
    && min>parseInt(ew[i][j]))
    {
     min=parseInt(ew[i][j]);
     k=i;
     l=j;
     }
 }
 mincost+=min;
 t[1][0]=k;
 t[1][1]=l;
 for(i=1;i<n;i++)
 {
  for(j=1;j<n;j++)
  if(ew[i][j]!=undefined)
   ew[j][i]=ew[i][j];
 }
 for(i=1;i<n;i++)
 {
  for(j=1;j<n;j++)
   if(ew[i][j]==undefined)
    ew[i][j]=1000000;
 }
 for(i=1;i<n;i++)
 {
  if(parseInt(ew[i][l])<parseInt(ew[i][k]))
   near[i]=l;
  else
   near[i]=k;
  near[k]=near[l]=0;
 }
 for(i=2;i<n-1;i++)
 {
  min=1000000;
  var k=0;
  for(j=1;j<n;j++)
  {
  if(near[j]!=0)
  {
   if(min>parseInt(ew[j][near[j]]))
   {
     k=j;
     min=parseInt(ew[j][near[j]]);
   }
  }
 }
 mincost+=min;
 t[i][0]=k;
 t[i][1]=near[k];
 near[k]=0;
 for(z=1;z<n;z++)
 {
  if(near[z]!=0&&
  parseInt(ew[z][near[z]])>
```

```
  parseInt(ew[z][k]))
     near[z]=k;
  }
 }
 color_edge();
 alert("Minimum Cost Generated from
  Prim's Algorithm : "+ mincost);
}
```

B. Kruskal's Function

```
function kru()
{
 var w=[];
 for(i=0;i<10;i++)
  w[i]=[];
 var i,j,m,k=1,z;
 mincost=0;
 for(i=1;i<n;i++)
 {
  for(j=1;j<n;j++)
   if(ew[i][j]==undefined)
      ew[i][j]=1000000;
 }
 for(i=1;i<n;i++)
 {
  for(j=1;j<n;j++)
  {
    if(ew[i][j]<1000000)
    {
     w[k][0]=ew[i][j];
     w[k][1]=i;
     w[k][2]=j;
     ew[j][i]=1000000;
     k++;
    }
  }
 }
 for(i=1;i<k;i++)
 {
  for(j=1;j<k-i;j++)
  {
   if(parseInt(w[j][0])>parseInt(w[j+1][0]))
   {
    m=w[j][0];
    w[j][0]=w[j+1][0];
    w[j+1][0]=m;
    m=w[j][1];
    w[j][1]=w[j+1][1];
    w[j+1][1]=m;
    m=w[j][2];
    w[j][2]=w[j+1][2];
    w[j+1][2]=m;
   }
  }
 }
 m=n;
 for(i=1;i<m-1;i++)
 {
  n1=w[i][1];
  n2=w[i][2];
  u=find(n1);
  v=find(n2);
  if(uni(u,v))
  {
    ce();
    mincost +=parseInt(w[i][0]);
  }
   else
    m++;
 }
 alert("Minimum Cost Generated from
  Kruskal's Algorithm : "+mincost);
}
function uni(i,j)
{
 if(i!=j)
 {
   parent[j]=i;
   return 1;
 }
 return 0;
}
function find(i)
{
 while(parent[i])
  i=parent[i];
 return i;
}
```

VI. Sample Outputs

Following are the some sample outputs that are generated :

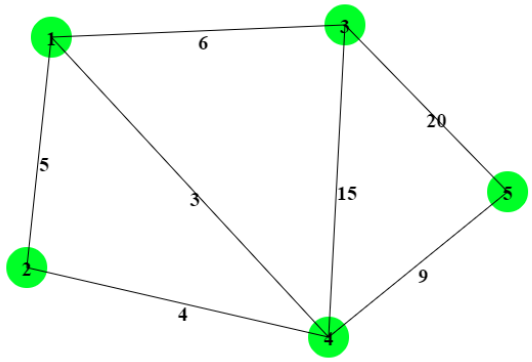

Fig. 2. A sample problem

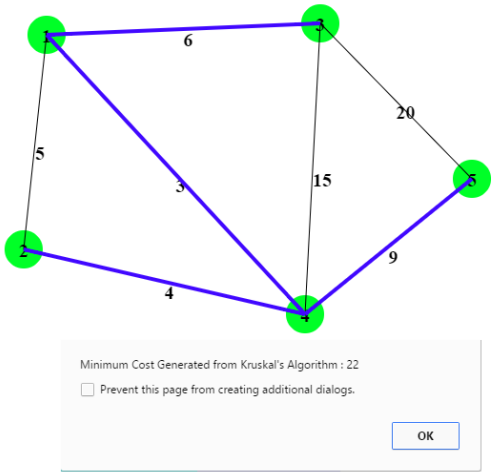

Fig. 3. Minimum-cost spanning tree generated from Kruskal's Algorithm

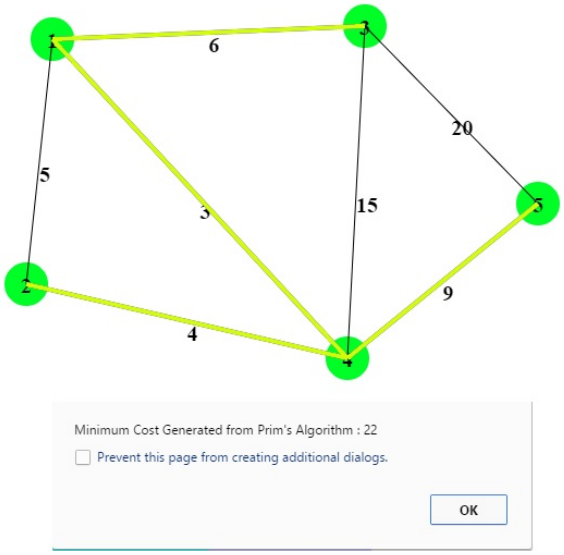

Fig. 4. Minimum-cost spanning tree generated from Kruskal's Algorithm

VII. Conclusion

- The time complexity of both the algorithms used in the program has $O(n^2)$.
- In case of Prim's Algorithm, the minimum-cost spanning tree is generated by selecting a least cost edge and extending it until MST is not formed. I looks like a single tree is extending its branches.
- While in case of the Kruskal's Algorithm, a forest like structures are created but as more or more edges are selected the forest keeps on joining and at the end only a single tree is left which is minimum-cost spanning tree.

VIII. Attachments

The Program code used for performing the above experiment is uploaded here : http://pastebin.com/YA0UXknW