

# Animations of Algorithms

## Team members -

Miryala Harsha Vardhan - 15114045

Sanju Prabhath Reddy - 15114042

Bandla Rakesh - 15114017

Dinesh Reddy - 15114026

## Instructions:

Install opengl

Cd to the directory

Run the following commands for required algorithm.

→ Compiling Line for dfs.cpp :: "g++ -std=c++0x dfs.cpp ./src/graph-graphics.cpp -l. -lglut -IGL -IGLU -o dfs"

→ Running Line for dfs.cpp :: ./dfs

→ Compiling Line for bfs.cpp :: "g++ -std=c++0x bfs.cpp ./src/graph-graphics.cpp -l. -lglut -IGL -IGLU -o bfs"

→ Running Line for bfs.cpp :: ./bfs

→ Compiling Line for krushkal.cpp :: "g++ -std=c++0x krushkal.cpp ./src/graph-graphics.cpp -l. -lglut -IGL -IGLU -o krushkal"

→ Running Line for krushkal.cpp :: ./krushkal

→ Compiling Line for dijkstra.cpp :: "g++ -std=c++0x dijkstra.cpp ./src/graph-graphics.cpp -l. -lglut -IGL -IGLU -o dijkstra"

→ Running Line for dijkstra.cpp :: ./dijkstra

## Code and Working:

Input: The inputs being taken in all the algorithms are

- Number of nodes in the graph
- Number of edges in the graph
- Each of the edges (i.e input is “0 1” if there is a edge from vertex 0 to vertex 1 in the graph)

Output: The animations of the workings of the algorithms.

We will be visualizing the specific algorithm working on the given example.

We will be taking nodes as input from mouse, i.e a white circle denoting the node will be formed as you “**left click**” your mouse on the yellow screen. After all the nodes are drawn, edges taken in as input will be formed between the appropriate nodes as black lines (used DDA for this line drawing algorithm).

After Right Clicking the mouse the algorithm execution starts.

Then according to the algorithm nodes will be traversed in the BFS or DFS. In the case of Krushkal the minumum spanning tree will be highlighted with white nodes. Whereas in the case of Dijkstra, we obtain the shortest paths to all the nodes from a given node.

## DFS Screenshots :

```
graph TD
    0((0)) --- 1((1))
    0((0)) --- 2((2))
    1((1)) --- 3((3))
    1((1)) --- 4((4))
    2((2)) --- 5((5))
    2((2)) --- 6((6))
```

```
37 {
38     if(color[i] == 0)
39
40     Depth First Search
41
42     0
43
44     1
45
46     2
47
48     3
49
50     4
51
52     5
53
54     6
55
56
57
58 }
59 }
60
61 void
62 {
63
64
65
66
67
68
69
70
71
72     usleep(600000);
73 }
74
75 if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
76 {
77     cout<<"\nRunning DFS\n";
78     for(int i=0;i<n;i++)
79     {
80         if(color[i]==0)
81         {
82             usleep(600000);
83         }
84     }
85 }
```

```
./dfs
1 3
1 4
2 5
2 6
3 21 652
102 535
473 550
71 467
258 398
394 413
573 398

Running DFS
greying node 0
exploring neighbors of 0
greying node 2
exploring neighbors of 2
greying node 6
exploring neighbors of 6
blacking, completely visited 6
greying node 5
exploring neighbors of 5
blacking, completely visited 5
```

```
graph TD
    0((0)) --- 1((1))
    0((0)) --- 2((2))
    1((1)) --- 3((3))
    1((1)) --- 4((4))
    2((2)) --- 5((5))
    2((2)) --- 6((6))
```

```
37 {
38     if(color[i] == 0)
39
40     Depth First Search
41
42     0
43
44     1
45
46     2
47
48     3
49
50     4
51
52     5
53
54     6
55
56
57
58 }
59 }
60
61 void
62 {
63
64
65
66
67
68
69
70
71
72     usleep(600000);
73 }
74
75 if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
76 {
77     cout<<"\nRunning DFS\n";
78     for(int i=0;i<n;i++)
79     {
80         if(color[i]==0)
81         {
82             usleep(600000);
83         }
84     }
85 }
```

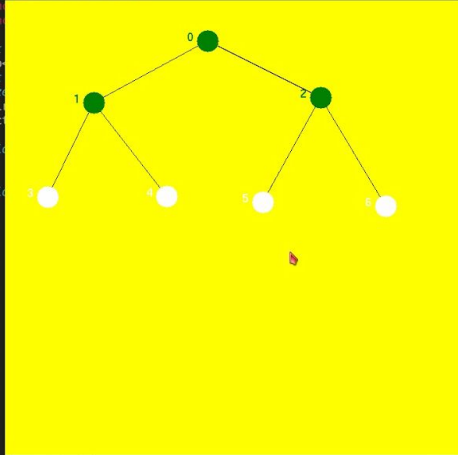
```
./dfs
greying node 0
exploring neighbors of 0
greying node 2
exploring neighbors of 2
greying node 6
exploring neighbors of 6
blacking, completely visited 6
greying node 5
exploring neighbors of 5
blacking, completely visited 5
greying node 1
exploring neighbors of 1
greying node 4
exploring neighbors of 4
blacking, completely visited 4
greying node 3
exploring neighbors of 3
blacking, completely visited 3
blacking, completely visited 1
blacking, completely visited 0

0 2 6 5 1 4 3
```

## BFS Screenshots:

Breadth First Search

```
1 #include <bits/stdc++.h>
2 #include <ctime>
3 #include <iostream>
4 #include <vector>
5 #include <queue>
6
7 int n, m;
8 map<int, vector<int>> adj;
9 int color[100000];
10 int size[100000];
11 pair<int, int> p[100000];
12 vector<int> v[100000];
13
14 void bfs(int src) {
15     queue<int> q;
16     q.push(src);
17     color[src] = 0;
18     while (!q.empty()) {
19         int u = q.front();
20         q.pop();
21         cout<<"blackening, completely visited "<<src<<"\n";
22         color[src] = 2;
23         display();
24         usleep(1200000);
25         if (final.size() == n && src == 0) {
26             break;
27         }
28     }
29 }
```



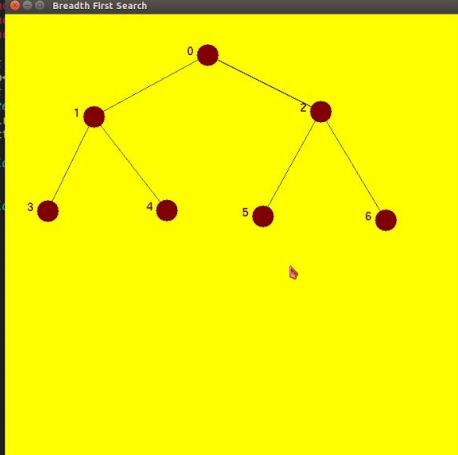
node numbers

```
7
Enter the number of edges in the graph
6
Enter the edges of the graph (u -> v) pairs
0 1
0 2
1 3
1 4
2 5
2 6
312 637
337 542
486 558
66 397
249 398
397 389
586 383
```

Running BFS  
greying node 0  
exploring neighbors of 0  
greying node 1  
greying node 2

Breadth First Search

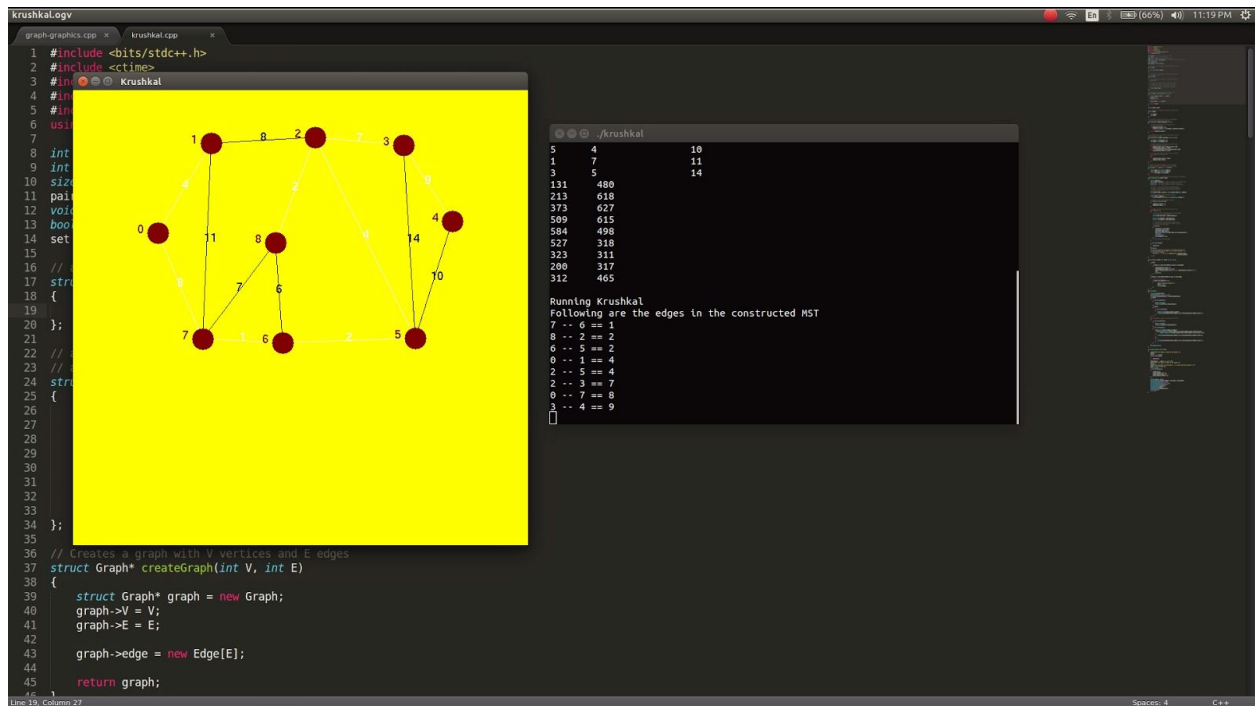
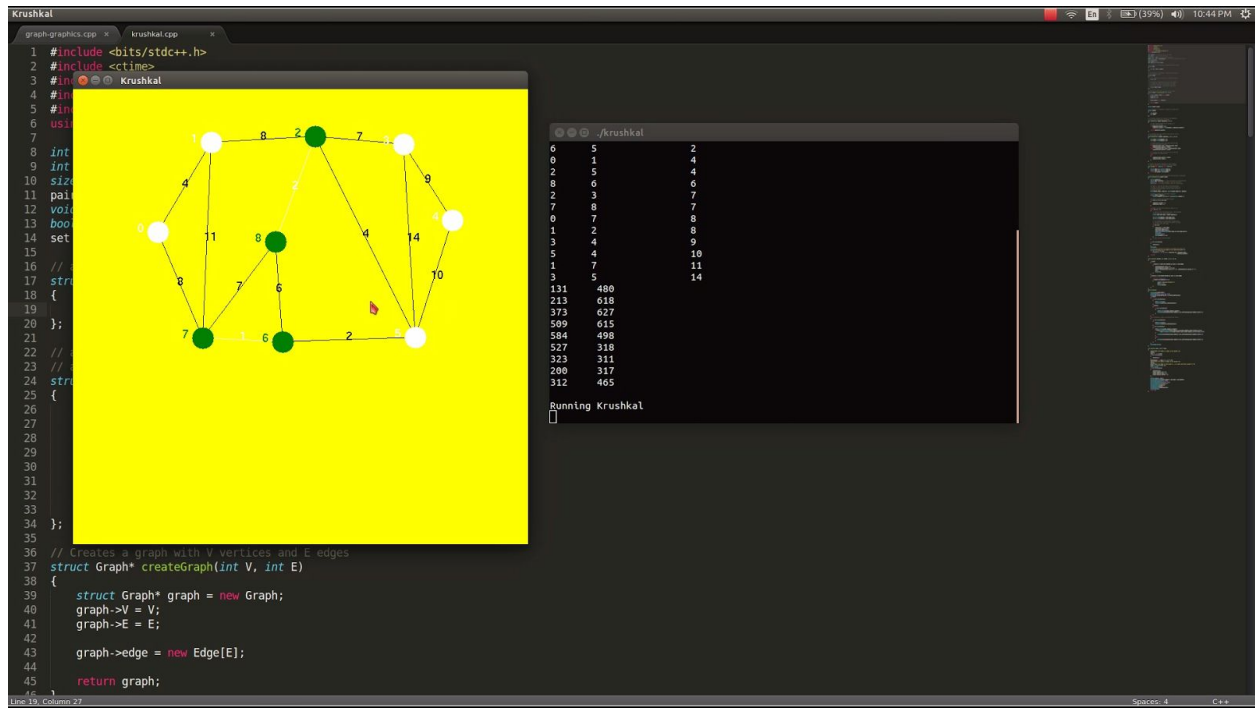
```
1 #include <bits/stdc++.h>
2 #include <ctime>
3 #include <iostream>
4 #include <vector>
5 #include <queue>
6
7 int n, m;
8 map<int, vector<int>> adj;
9 int color[100000];
10 int size[100000];
11 pair<int, int> p[100000];
12 vector<int> v[100000];
13
14 void bfs(int src) {
15     queue<int> q;
16     q.push(src);
17     color[src] = 0;
18     while (!q.empty()) {
19         int u = q.front();
20         q.pop();
21         cout<<"blackening, completely visited "<<src<<"\n";
22         color[src] = 2;
23         display();
24         usleep(1200000);
25         if (final.size() == n && src == 0) {
26             break;
27         }
28     }
29 }
```



node numbers

```
greying node 0
exploring neighbors of 0
greying node 1
greying node 2
exploring neighbors of 1
greying node 3
greying node 4
exploring neighbors of 3
exploring neighbors of 4
blackening, completely visited 3
blackening, completely visited 4
blackening, completely visited 1
exploring neighbors of 2
greying node 5
greying node 6
exploring neighbors of 5
blackening, completely visited 5
exploring neighbors of 6
blackening, completely visited 6
blackening, completely visited 2
blackening, completely visited 0
0 1 2 3 4 5 6
```

## Kruskal Screenshots:



## Dijkstra Screenshots:

```

Dijkstra
→ cd Desktop/OpenGL-Animations/d
cd: no such file or directory: Desktop/OpenGL-Animations/d
→ cd Desktop/OpenGL-Animations/dijkstra
→ dijkstra git:(master) X is
dijkstra dijkstra.cpp testinput.txt
→ dijkstra git:(master) X g++ -std=c++0x dijkstra.cpp ../src/graph-graphics.cpp
-I. -lglut -lGL -lGLU -o dijkstra
→ dijkstra git:(master) X ./dijkstra
Enter the number of nodes in the graph vertex 1 is included / in shortest
9 distance from src to 1 is finalized
Enter the number of edges in the graph
14
Enter the edges of the graph (u --> v) pairs and their weight w
0 1 4
0 7 8
0 7 8
1 2 8
1 7 11
2 3 7
2 8 2
2 5 4
3 4 9
3 5 14
4 5 10
5 6 2
6 7 1
6 8 6
7 8 7
123 396
178 527
393 554 // find shortest path for all vertices
394 558 for (int count = 0; count < n-1; count++)
753 451
691 306
481 279
292 279 // mark the picked vertex as processed
430 406
95 sptSet[u] = true;
color[u] = 1;
Running Dijkstra display();
97 usleep(1200000);
98 // Update dist value of the adjacent vertices of the
99 // picked vertex
100 for (int v = 0; v < n; v++)
101 { if(sptSet[v] && graph[u][v] && dist[u] + graph[u][v]
102 {
103 parent[v] = u;
104 dist[v] = dist[u] + graph[u][v];
105 }
106 }
107 done = true;
108 // print the constructed distance array
109 printSolution(n, parent);
110 display();
111 }
112 void mouseClickButton --(src, x, y, int u)
113 {

```

Dijkstra-Shortest Path Algorithm  
White - Initial color of the node  
Green - The node is being processed  
Red - All of its neighbours are reached or the final processing is done

```

outlogv
→ cd Desktop/OpenGL-Animations/d
cd: no such file or directory: Desktop/OpenGL-Animations/d
→ cd Desktop/OpenGL-Animations/dijkstra
→ dijkstra git:(master) X is
dijkstra dijkstra.cpp testinput.txt
→ dijkstra git:(master) X g++ -std=c++0x dijkstra.cpp ../src/graph-graphics.cpp
-I. -lglut -lGL -lGLU -o dijkstra
→ dijkstra git:(master) X ./dijkstra
Enter the number of nodes in the graph vertex 1 is included / in shortest
9 distance from src to 1 is finalized
Enter the number of edges in the graph
14
Enter the edges of the graph (u --> v) pairs and their weight w
0 1 4
0 7 8
0 7 8
1 2 8
1 7 11
2 3 7
2 8 2
2 5 4
3 4 9
3 5 14
4 5 10
5 6 2
6 7 1
6 8 6
7 8 7
123 396
178 527
393 554 // find shortest path for all vertices
394 558 for (int count = 0; count < n-1; count++)
753 451
691 306
481 279
292 279 // mark the picked vertex as processed
430 406
95 sptSet[u] = true;
color[u] = 1;
Running Dijkstra display();
97 usleep(1200000);
98 // Update dist value of the adjacent vertices of the
99 // picked vertex
100 for (int v = 0; v < n; v++)
101 { if(sptSet[v] && graph[u][v] && dist[u] + graph[u][v]
102 {
103 parent[v] = u;
104 dist[v] = dist[u] + graph[u][v];
105 }
106 }
107 done = true;
108 // print the constructed distance array
109 printSolution(n, parent);
110 display();
111 }
112 void mouseClickButton --(src, x, y, int u)
113 {

```

Dijkstra-Shortest Path Algorithm  
White - Initial color of the node  
Green - The node is being processed  
Red - All of its neighbours are reached or the final processing is done

## Algorithms:

### 1) Breadth First Search:

What we do in a BFS is a simple step-by-step process, which is -

1. Start from a vertex S. Let this vertex be at, what is called.... "Level 0".
2. Find all the other vertices that are immediately accessible from this starting vertex S, i.e., they are only a single edge away.
3. Mark these vertices to be at "Level 1".
4. There will be a challenge that you might be coming back to the same vertex due to a loop or a ring in the graph. If this happens your BFS will take  $\infty$  time. So, you will go only to those edges who do not have a Level set to some value.
5. We use a queue to implement this we push all the current unvisited neighbours of the current vertex to the queue
6. Mark which is the parent vertex of the current vertex you are at, i.e., the vertex from which you accessed the current vertex. Do this for all the vertices at Level 1.
7. Now, find all those vertices that are a single edge away from all the vertices which are at "Level 1". These new set of vertices will be at "Level 2".
8. Repeat this process until you run out of graph.

### 2) Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

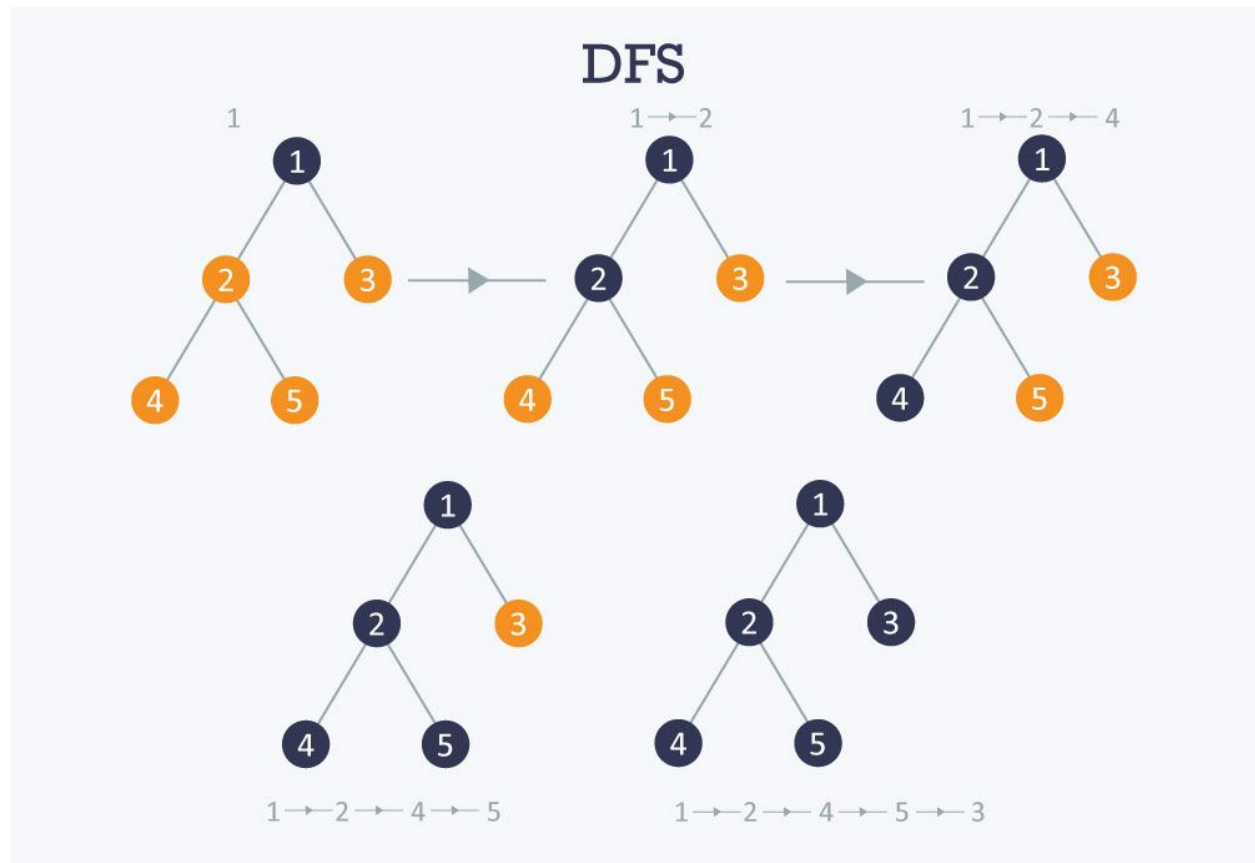
Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack.

Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.





### 3)Krushkal:

Used to find the minimum spanning tree of a graph.

Below are the steps for finding MST using Krushkal's algorithm.

1. Sort all the edges in non-decreasing order of their weight
2. Pick the smallest edge. Check if it forms a cycle with spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step 2 until there are  $(V-1)$  edges in the spanning tree

#### After Sorting:

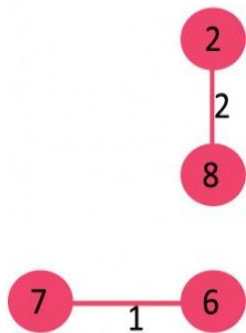
Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from sorted list of edges

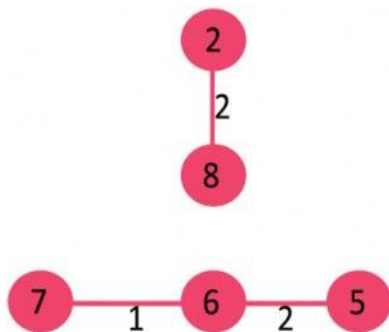
1. Pick edge 7-6: No cycle is formed, include it.



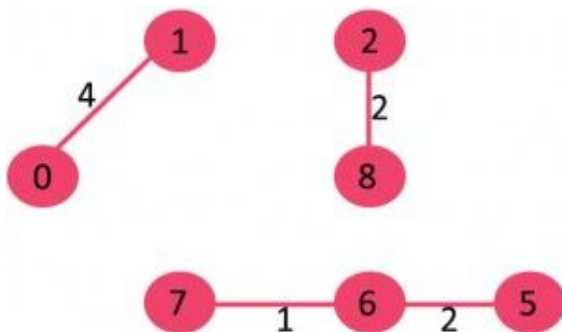
2. Pick edge 8-2: No cycle is formed, include it.



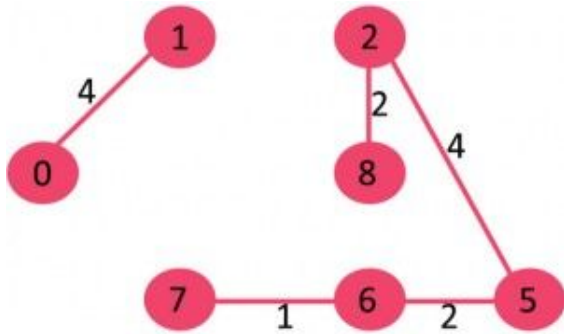
3. Pick edge 6-5: No cycle is formed, include it.



4. Pick edge 0-1: No cycle is formed, include it.

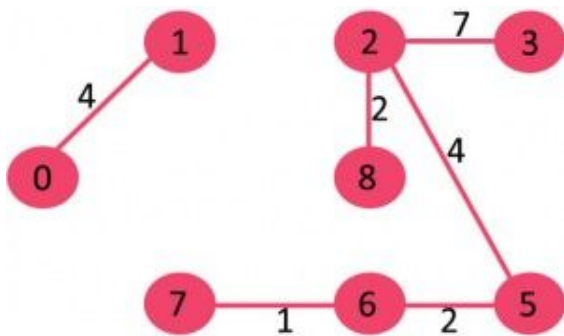


5. Pick edge 2-5: No cycle is formed, include it.



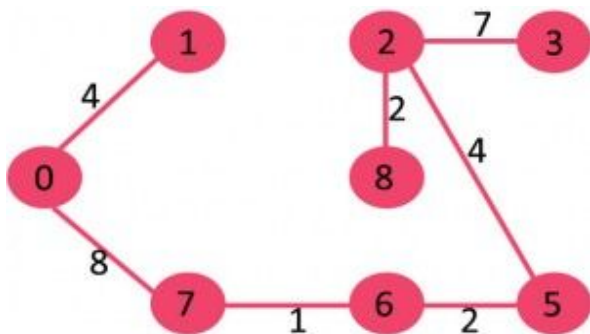
6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.



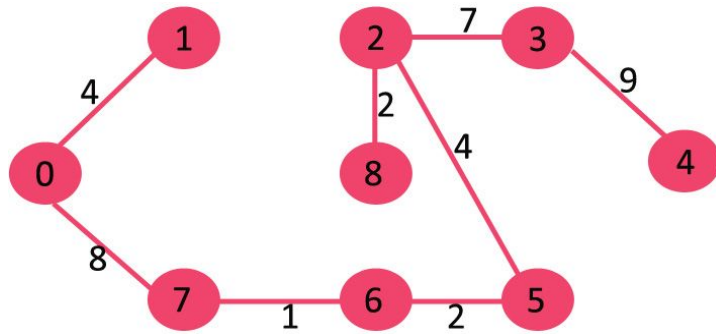
8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals  $(V - 1)$ , the algorithm stops here.

#### 4) Dijkstra's Algorithm

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

- 1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

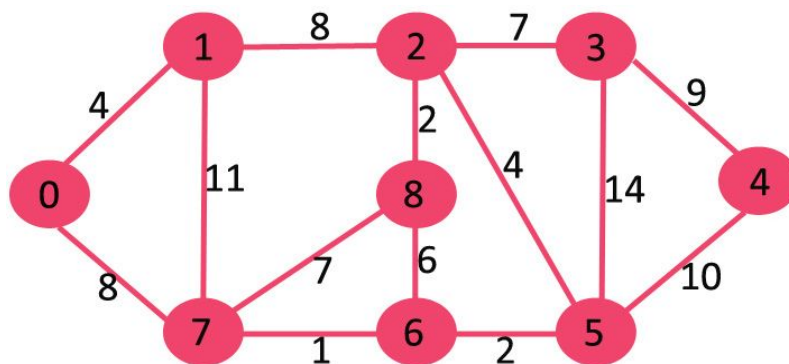
3) While sptSet doesn't include all vertices

....a) Pick a vertex  $u$  which is not there in sptSet and has minimum distance value.

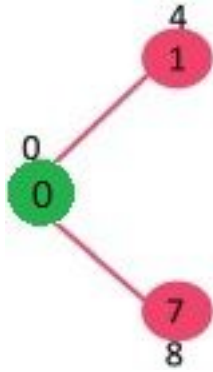
....b) Include  $u$  to sptSet.

....c) Update distance value of all adjacent vertices of  $u$ . To update the distance values, iterate through all adjacent vertices. For every adjacent vertex  $v$ , if sum of distance value of  $u$  (from source) and weight of edge  $u-v$ , is less than the distance value of  $v$ , then update the distance value of  $v$ .

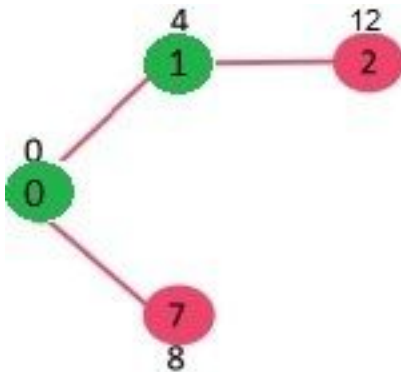
Let us understand with the following example:



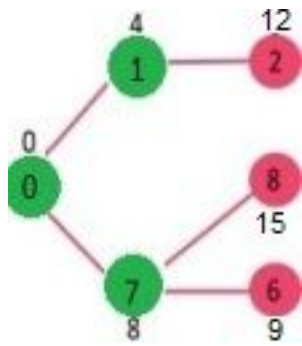
The set sptSet is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.



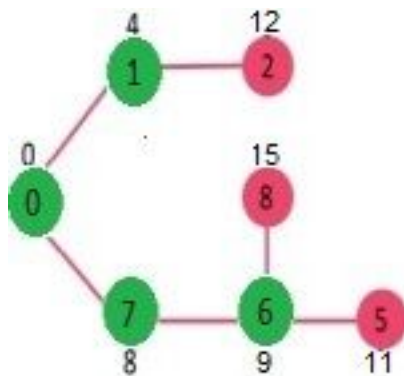
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSet). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until sptSet doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

