ISSN No. 0976-5697

Volume 5, No. 7, September-October 2014

International Journal of Advanced Research in Computer Science

CASE STUDY AND REPORT

Available Online at www.ijarcs.info

Sorting of Numbers in a Graph-Theoretic Approach

Sinchan Sengupta
Computer Science Dept. ,
Scottish Church College,
Kolkata, India

Ritwika Law
Computer Science Dept. ,
Scottish Church College,
Kolkata, India

Abstract: Till now we have not come across any algorithm that uses graphs to sort any set of numbers (only topological sorts like-Breadth First Search and Depth First Search exist). Our aim in this paper is to sort a set of numbers in either ascending or descending order, using undirected weighted graphs and is based on already existing Kruskal's algorithm. This paper highlights this new application of the algorithm and the fact that it can be extended to sorting numbers as well. The given set of numbers to be sorted are assigned to the vertices of the graph and the edges incident to that ordered pair of vertices are assigned some integer value(weights) based on which the sorting is done. The complexity of the algorithm in this paper is in accordance with the complexities of already existing sorting algorithms.

Keywords: Graph completeness, 'good' graphs, sorting, Kruskal's Algorithm, complexity.

I. INTRODUCTION

Our aim in this paper is to implement a sorting algorithm using graphs.
According to [1] linear graph (or simply a graph) G = (V, E) consists of a set of objects V={v1,v2,…} called vertices and another set E={e1,e2,…}, whose elements are called edges such that each edge is identified by an unordered pair of vertices.
A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a pair of unique edges. It is impossible to add an edge to a complete graph because every possible edge has been drawn. To implement this algorithm we will be requiring a complete graph always.
A graph structure can be extended by assigning a weight to each edge of the graph. Graphs with weights, or weighted graphs, are used to represent structures in which pair wise connections have some numerical values.
To represent the graphs used for this algorithm we will use an adjacency matrix. In [1] the author noted that the adjacency matrix of a graph with n vertices and no parallel edges is an n by n symmetric binary matrix X=[xij] defined over the ring of integers such that:
Xij=1, if there is an edge between i'th and j'th vertex
Xij=0, if there is no edge between them
Here the adjacency matrix will be taken as an input from the user.
According to [1] a very important concept used widely in the field of graph theory, is a minimum spanning tree, which is a tree T, having the minimum sum of edge weights, and is a subgraph of the original graph G, having all the vertices of G. The most widely known algorithm that finds the minimum spanning tree is the Kruskal's Algorithm. According to [2] Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).
According to [3] the general procedure followed for obtaining a minimum spanning tree by Kruskal's algorithm are as under:

a) A forest is constructed with each node in a separate tree.

b) The edges are placed in a priority queue.

c) Until we have added (n-1) edges where 'n' is the number of vertices in the graph,
   (i) Extract the cheapest edge from the queue,
   (ii) If it forms a cycle then reject it.
   (iii) Else add it to the forest. Adding it to the forest will join two trees together.

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.
This paper consists of two sections. The first section presents how to assign integer values or weights to all the edges of the graph. The second section discusses how the sorting of numbers is carried out using graphs.

II. THEORY

SECTION A

The main objective in this section is to design such a data structure that can be easily accessed and manipulated to generate the sequence of sorted numbers.

For this purpose, we have used a concept, very close to that of 'good' graphs. For e.g., to represent the relationship among the numbers : 1,5,7 ; we take the help of an undirected, complete graph, such that each edge represent the relation among the numbers at two end-point vertices, and the weight of that edge being the absolute difference of the magnitude of it's vertex values at the two end-points.

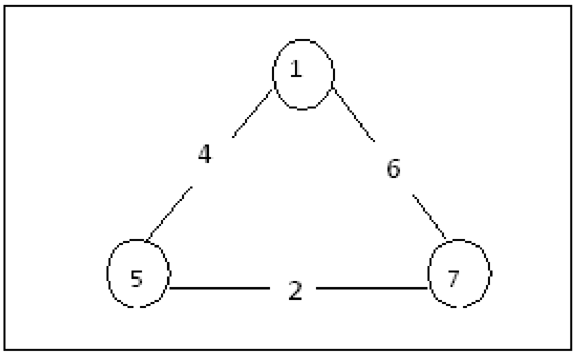

Fig – 01: Relationship graph for numbers: 1, 5, 7

Mathematically, for this kind of graph: G= (V, E), we have 'E' as the set of edges, where each e Є E, is an edge such that V1 and V2 (Є V) are incident on it, and it's weight be defined as:

W (e) = | | V1 | - | V2 | | …………………. (1)

Now, let us analyze all possible cases that one can encounter while sorting numbers based on comparison (i.e., between two adjacent vertices connected by an edge):

a) Comparison between two positive numbers or two negative numbers.

b) Comparison between one positive and one negative numbers.

For the first case, the weight of the edge is given by (1); but for the second case, this same equation has to be modified to calculate the weight as:

W (e) = | V1 - V2 | …………………. (2)

, where V1 is positive and V2 is negative.

Now, we are concerned with sorting 'n' numbers (n>0), where the numbers may be in a random order, mixture of positive as well as negative numbers. To deal with all such random order of numbers (distinct permutation of or arrangement of 'n' elements), we need to construct a complete graph, where all vertices represent the numbers to be sorted, and every vertex is connected to every other vertex (or number) in that graph. This is done because the sorted order includes any combination of 'n' numbers, where any number can be present before or after any other number. This complete graph gives us the opportunity to have all options and combinations open to include the edges and vertices and traverse them in any order, since every path exists in such a graph. For e.g.,

Let us take the following set of numbers: [2, -1, 3, 8]

The vertices of the graph in this case are the numbers given above: 2,-1, 3, 8; and the weight of the edges of the graph are given by (1) and (2). Therefore, the graph can be constructed as:

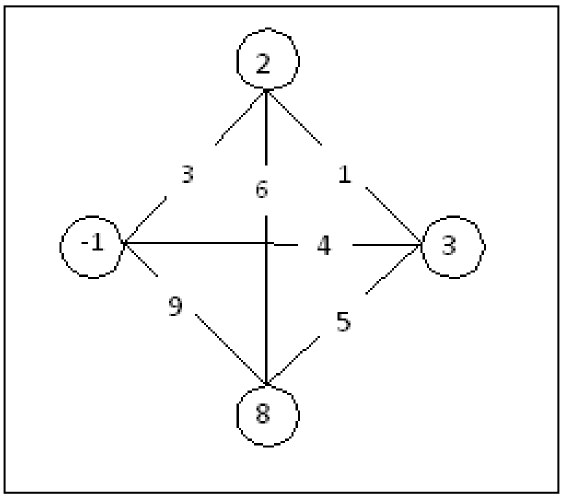

Fig-02 : Graphical representation of a random set of 4 elements.

SECTION B

Section A specifies how to represent a graph that will be used in this algorithm to sort a set of given numbers. So now we can assume that the graph has already been created which is represented in the computer memory by an adjacency matrix. Here, we specify how the given n elements, which are assigned to the vertices, are being sorted.

ALGORITHM

[**Input**: 'n' numbers to be sorted in a 1D array.
Data Structures: 2D array to represent the adjacency matrix.
Output : 1D array to display the numbers.]

```
Step 1: Start
Step 2: Accept 'n' numbers from the user and store them in an array.
Step 3: Find the smallest element of A. Store its value in 'Vs' and its position in 'p'. 'Vs' is
the source vertex.
Step 4: Set i: =i+1
Step 5: Set j: =j+1
Step 6: a) If A[i] and A[j] are both positive and both negative then,
                Xij=|| A[i] |-| A[j] ||
        b) If either of A[i] or A[j] is positive and the other is negative then,
                Xij=|a-b|
           Where 'a' is the positive vertex and 'b' is the negative vertex.
        c) Set Xij: =0 when i=j
    Step 7: If j<=n then go to step 5
    Step 8: If i<=n then go to step 4
```

Step 9: Select an edge (i,j) from the connected graph represented by the adjacency matrix such that:

a) Xij is minimum
b) d(Vs)<2 and Vt ≠ Vs, where Vt ∈ A and Vt ≠Vs
c) component(i) ≠ component(j)

Step 10: a) Set Xij: =M
b) Set Xij: =M
where $M \rightarrow +\infty$

Step 11: Repeat from Step 9 until (n-1) edges are selected.
Step 12: Select the edge (i,j) such that Xij=M from the pth row.
Step 13: Set k: = 0
Step 14: B[k]: =A[i]
Step 15: k: =k+1
Step 16: Select an edge (u,v) from the jth row such that Xuv=M and v≠i
Step 17: a) Set i: = u
b) Set j: = v
Step 18: Set B[k+1]: = A[i]
Step 19: Repeat from Step 12 until (n-1) edges are selected.
Step 20: End

[Here, B contains the sorted order of the set of 'n' elements. The above algorithm is designed to sort the numbers in ascending order. To sort the numbers in descending order, the same algorithm is used, only making a change in Step 2, where we have to store the largest element as the initial vertex.]

Example 1 : Sorting the numbers : 4,6,-3,-4 and 5 in ascending order. (Sorting of distinct elements)

Procedure : Here, we can consider five vertices as:

Tab 1 - Vertex arrangement for e.g. 1

V_1	V_2	V_3	V_4	V_5
4	6	-3	-4	5

The complete graph corresponding to the set of numbers is:

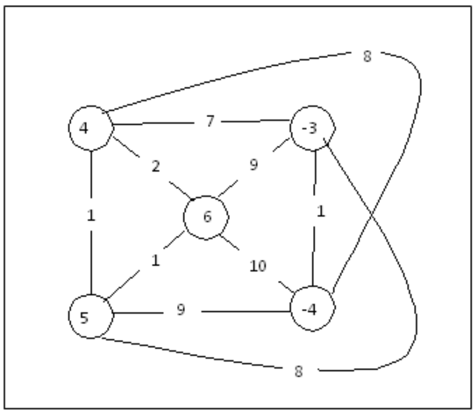

Fig 03 - Graphical representation of the relationship among the set of numbers

The adjacency matrix of the above graph is:

Tab 2 - Adjacency matrix

	V_1	V_2	V_3	V_4	V_5
V_1	0	2	7	8	1
V_2	2	0	9	10	1
V_3	7	9	0	1	8
V_4	8	10	1	0	9
V_5	1	1	8	9	0

Now, we apply the proposed algorithm on this graph.

Step 1: The initial vertex chosen as V_4 since, it is the maximum element in the entire set of numbers. A most minimum edge (V_1,V_5) is selected satisfying the conditions mentioned in step 4 of the algorithm.

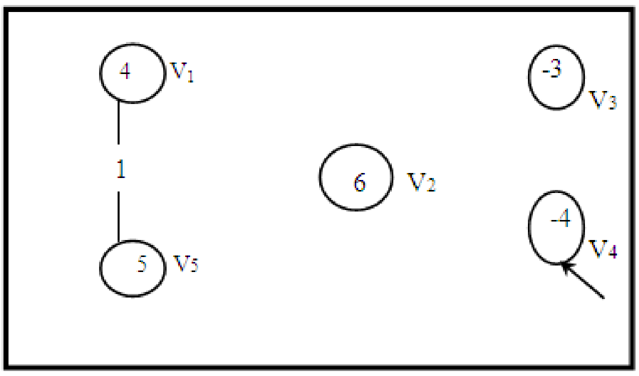

Fig 04 - Selection of first edge

Step 2: Edge (V_2, V_5) is selected as the next minimum edge weight is also 1.

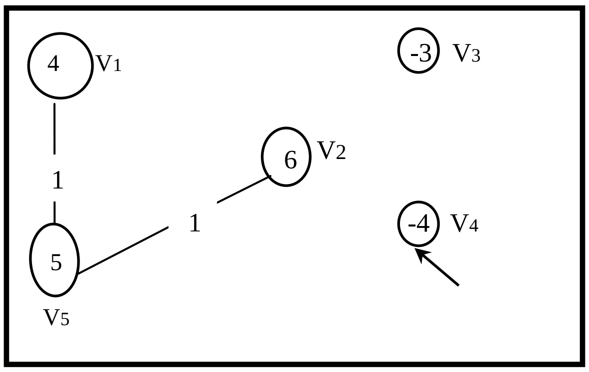

Fig 05- Selection of second edge

Step 3 : Edge (V_3. V_4) is selected as the next minimum edge weight is also 1.

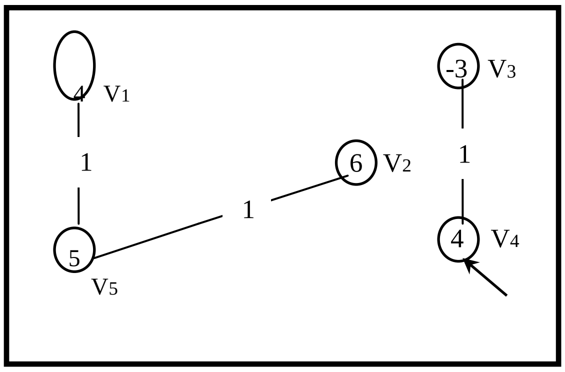

Fig 06- Selection of third edge

Step 4 : Edge (V_1. V_3) is selected (edge value '2' is not selected as it forms a closed circuit) of weight '7'.

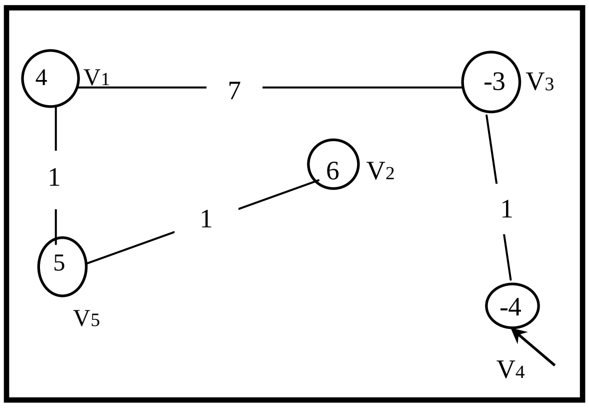

Fig 07- Selection of forth edge

Step 5: So, to get the sorted order, we traverse the edges from the initial vertex, i.e. V4 and following the edge path from it, we get the sorted order as :

Tab 3- Sorted order of the numbers

V_4	V_3	V_1	V_5	V_2
-4	-3	4	5	6

So, the ascending order sorting of the numbers yield the sequence: -4, -3, 4, 5, and 6 , which is obtained by following the edge sequence starting from the initial vertex as depicted in Fig – 07.

Example 2 :

Sorting the numbers: – 6, 7, 1, 5, -6, -2 in descending order. (With repetition cases involved)

Hence, we can consider the six vertices as:

Tab 4- Vertex arrangement

V_1	V_2	V_3	V_4	V_5	V_6
-6	7	1	5	-6	-2

The complete graph corresponding to the set of numbers is:

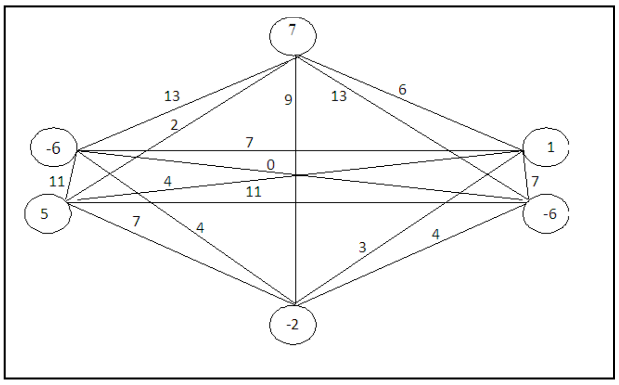

Fig 08 - Complete graph of Tab - 4

The adjacency matrix of the above graph is :

Tab 5- Adjacency matrix

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	13	7	11	0	4
V_2	13	0	6	2	13	9
V_3	7	6	0	4	7	3
V_4	11	2	4	0	11	7
V_5	0	13	7	11	0	4
V_6	4	9	3	7	4	0

Now, we apply the proposed algorithm on this graph.

Step 1: The initial vertex chosen as V_2 since it is the maximum element in the entire set of numbers. A most minimum edge (-6,-6) is selected satisfying the conditions mentioned in step 4 of the algorithm.

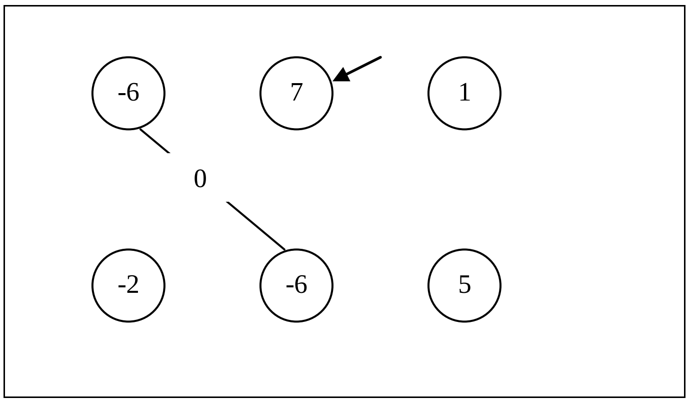

Fig 09 – Selection of first edge

Step -2 Edge (7, 5) is selected as the next minimum edge weight is '2'.

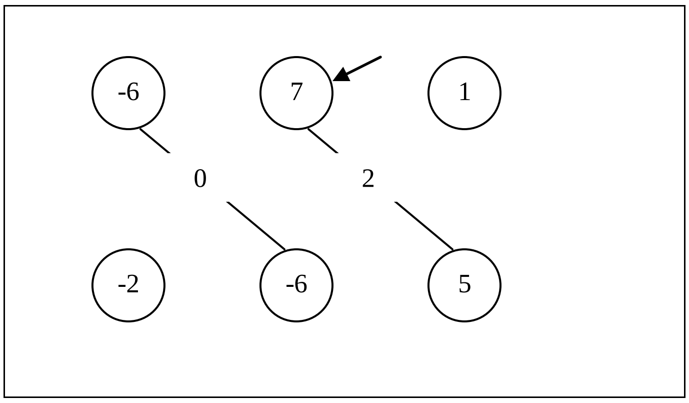

Fig 10 -Selection of second edge

Step -3 Edge (1,-2) is selected as the next minimum edge weight is '3'.

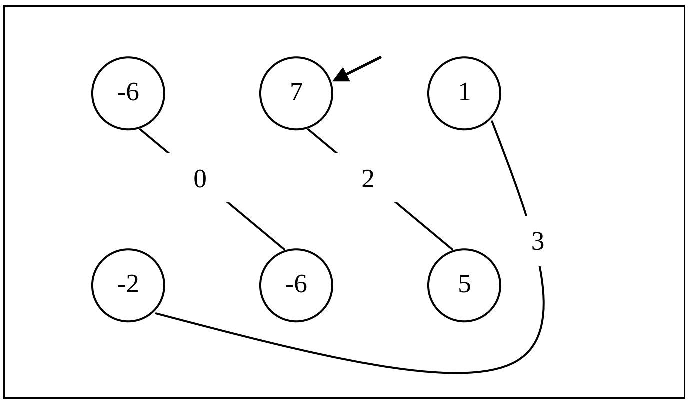

Fig 11 – Selection of third edge

Step-4 Edge (5, 1) is selected as the next minimum edge weight is '4'.

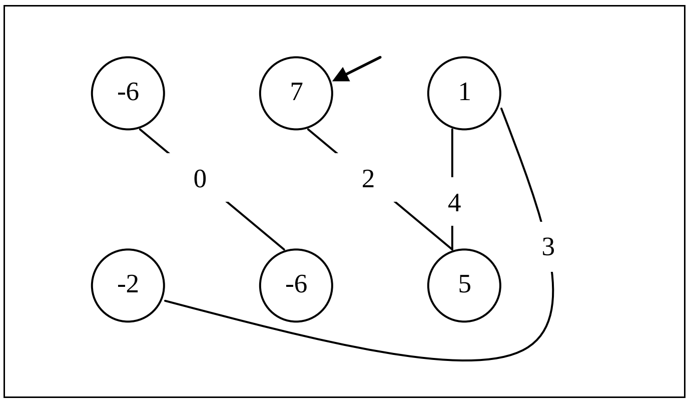

Fig 12- Selection of forth edge

Step 5: Edge (-6, -2) is selected as the next minimum edge weight is also '4'.

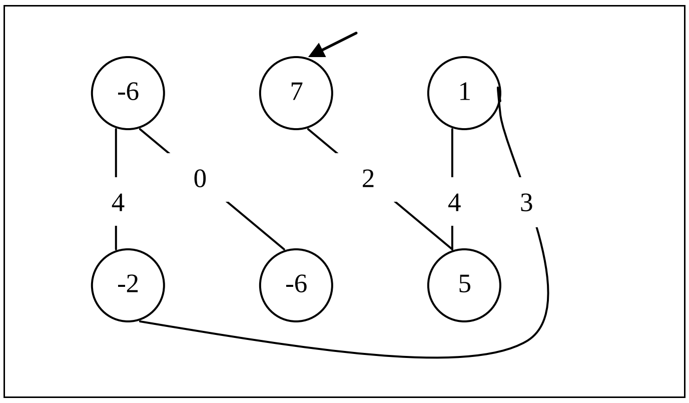

Fig 13 – Selection of fifth edge

Step 6: So the sorted order is,

Tab 6 - Sorted order of the numbers

V_2	V_4	V_3	V_6	V_1	V_5
7	5	1	-2	-6	-6

So, the descending order sorting of the numbers yield the sequence: 7, 5, 1,-2,-6, and -6, which is obtained by following the edge sequence starting from the initial vertex as depicted in Fig – 13.

Since the selection of edges in this algorithm is in accordance with Kruskal's algorithm, with just a few additional restrictions imposed, we can surely say that the proposed method of sorting numbers is an extended application of Kruskal's algorithm.

III. COMPLEXITY

According to [4], initially, ‘E’ is the set of all edges in the graph G. First, we determine an edge with minimum cost and then delete this edge from the graph as that edge can no longer be selected as a prospective edge for the next step. To select the next minimum edge cost from the remaining set, the edges can be either maintained as a minheap, and then we can select the next edge in O (log |E|) time or we have to group the edges/arrange the edges in such a way that one can easily determine whether the vertices V and W are already connected by the earlier selection of edges.

Since, our algorithm is based on the operations performed by the well known Kruskal’s algorithm, analyzing the same will provide a very good insight to the run-time complexity of the proposed algorithm.

According to [2], Kruskal's algorithm is known to run in O (|*E*| log |*E*|) time, or equivalently *O* (|*E*| log |V|), time, all with simple data structures. These running times are equivalent because:

- *E* is at most V^2 and $V^2 = 2 \log V$ is O (log V).
- Each isolated vertex is a separate component of the minimum spanning forest. If we ignore isolated vertices we obtain $V \le E+1$, so log V is O (log *E*).

In the algorithm proposed in Section B, it is required that the minimum element from the array be calculated at the beginning of the method. To do that, a maximum of ‘n’ times traversal through the data structure (1D array of set of vertices) is required. This generates a O (log n) complexity, where ‘n’ is the number of elements (vertices). The application of Kruskal’s algorithm incurs a complexity of *O* (|*E*| log |V|), which is already shown. For the graphical construction considered for sorting numbers explained in Section A, clearly , the cardinality of vertex set is ‘n’, whereas, for the edge set, the maximum value is n(n-1)/2 . ($\approx n^2$).

Hence, the time complexity can be expressed as :

$$O(n) + O(|E| \log |E|)$$

$$= O(n) + O(n^2 \log n^2)$$

$$= O(n) + O(2n^2 \log n)$$

$$\approx O(n^2 \log n)$$

More generally, this algorithm runs in O(|E| log |V|) time.

The space complexity of the above algorithm is $O(n^2)$ since an additional data structure is used to store the adjacency matrix of the graph, which is a 2D array.

III. ACKNOWLEDGMENT

We would like to express our utmost gratitude to prof. Mr. Arun Kumar Chakrabarti, Gurudas College, Kolkata, for his untiring will to make this maiden endeavour of ours successful. Also, our sincere thanks go out to all the professors of Dept. of Computer Science, Scottish Church College, for their encouragement and help in this matter. At the end of the day, our parents are the pillars of strength, without whom any effort is futile.

V. REFERENCES

[1] Narsingh Deo, Graph Theory- With Applications to Engineering and Computer Science, Eastern Economy Edition, 1974, Pub: Asoke K. Ghosh, pp: 1-2, 55-56, 157-158.
[2] http://en.wikipedia.org/wiki/Kruskal's_algorithm#Description
[3] R. B. Patel, Expert Data Structures with C, third edition, Khanna Book Publishing Co. (P) Ltd., pp: 470-471.
[4] http://en.wikipedia.org/wiki/Kruskal's_algorithm#Complexity