**SELECTION SORT**

Selection sort is a simple sorting algorithm. This sorting algorithm is a in-place comparison based algorithm in which the list is divided into two parts, sorted part at left end and unsorted part at right end. Initially sorted part is empty and unsorted part is entire list.

Smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes part of sorted array. This process continues moving unsorted array boundary by one element to the right.

How selection sort works?

Let us take this example,

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the the beginning in the sorted manner.

Selection Sort

The same process is applied on the rest of the items in the array. We shall see an pictorial depiction of entire sorting process −



Algorithm:

Step 1: Set MIN to location 0

Step 2; Search the minimum element in the list

Step 3: Swap with value at location MIN

Step 4: Increment MIN to point to next element

Step 5: Repeat until list is sorted

Pseudo code:

procedure selection sort

list : array of items

n : size of list

for i = 1 to n - 1

/\* set current element as minimum\*/

min = i

/\* check the element to be minimum \*/

for j = i+1 to n

if list[j] < list[min] then

min = j;

end if

end for

/\* swap the minimum element with the current element\*/

if indexMin != i then

swap list[min] and list[i]

end if

**IMPLEMENTATION PROGRAM:**

#include <stdio.h>

#include <stdbool.h>

#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {

int i;

for(i = 0;i <count-1;i++) {

printf("=");

}

printf("=\n");

}

void display() {

int i;

printf("[");

// navigate through all items

for(i = 0;i<MAX;i++) {

printf("%d ", intArray[i]);

}

printf("]\n");

}

void selectionSort() {

int indexMin,i,j;

// loop through all numbers

for(i = 0; i < MAX-1; i++) {

// set current element as minimum

indexMin = i;

// check the element to be minimum

for(j = i+1;j<MAX;j++) {

if(intArray[j] < intArray[indexMin]) {

indexMin = j;

}

}

if(indexMin != i) {

printf("Items swapped: [ %d, %d ]\n" , intArray[i], intArray[indexMin]);

// swap the numbers

int temp = intArray[indexMin];

intArray[indexMin] = intArray[i];

intArray[i] = temp;

}

printf("Iteration %d#:",(i+1));

display();

}

}

main() {

printf("Input Array: ");

display();

printline(50);

selectionSort();

printf("Output Array: ");

display();

printline(50);

}

OUTPUT:

Input Array: [4 6 3 2 1 9 7 ]

Items swapped: [ 4, 1 ]

Iteration 1#:[1 6 3 2 4 9 7 ]

Items swapped: [6, 2]

Iteration 2#:[1 2 3 6 4 9 7 ]

Iteration 3#:[1 2 3 6 4 9 7 ]

Items swapped: [6, 4 ]

Iteration 4#:[1 2 3 4 6 9 7 ]

Iteration 5#:[1 2 3 4 6 9 7 ]

Items swapped: [ 9, 7 ]

Iteration 6#:[1 2 3 4 6 7 9 ]

Output Array: [1 2 3 4 6 7 9 ]

**ANALYSIS:**

* Selection sort Algorithm is used to sort array accordingly but it is little bit different then Insertion sort.
* Actually it first Specify the place with next largest or smallest element and Compare with all other element i.e. In-Position Comparing sort.
* The Idea of Selection sort is very simple repeatedly finding the smallest or greatest element in the array and place it to its final position to sort accordingly in a sequence.
* Let’s see  the Algorithm of the Selection Sort is as follows:

Line                                                      Time                                  Cost

For j <--- 1 to length[Array]                                        n+1                                    C1

* + Do k<--- j+1                                                      n                                       C2

i<--- j                                                     n                                       C3

* + While(k<=length[Array])                            n**∑**j=1  **tj**C4  
                         Do if (Array[i]>Array[k])      n**∑**j=1  **tj -1                           C5**                           Then i<--- k                                     n**∑**j=1  **tj -1                       C6**  
    k++                                  n**∑**j=1  **tj -1**C7               Exchange Array[i] <-->                  n                                       C8

Now we calculate the time complexity of the Selection sort algorithm in worst case as well as best  case as follows

From selection sort algorithm the equation is form is  
               T(n) = C1\*(n+1) + C2 \*n+ C3\*n + C4\* n**∑**j=1  **tj +**C5\*n**∑**j=1  **tj-1**+ C6\*n**∑**j=1  **tj-1** + C7\*n**∑**j=1  **tj-1**+ C8\*n  
  
               **For Worst Case**:  
               Taking totally reverse array example i.e  
                        [17][13][11][9][5][2][1]   
               first of all solving the time problem making Summations  n**∑**j=1  **tj**and n**∑**j=1  **tj-1**in to term  of n in equation.  
  
                n**∑**j=1 **tj =**t1 + t2 + t3 + t4 + t5 +.....................................+ tn  
                                = 1 +   2 +  3 +  4 + 5  +.....................................+ n  
                                = n(n+1)/2                                                                ..............................(1)  
  
                n**∑**j=1 **tj-1 =**t1-1 + t2-1 + t3-1 + t4-1 + t5-1+.............................+tn-1  
                                   = n**∑**j=1 tj - n**∑**j=1 1  
                                   = n(n+1)/2 - (n-1)                                                    ...............................(2)  
  
                Now putting (1) and (2) in the the above equation we get:  
                
                T(n) = C1\*(n+1) + C2\*n + C3\*n + C4\*(n(n+1)/2) + C5\*(n(n+1)/2-(n-1)) + C6(n(n+1)/2-(n-1)) + C7(n(n+1)/2-(n-1)  
                           + C8\*n)  
                T(n) = C1\*n + C1 +C2\*n +C3 \*n + C4\*(n\*n/2) +C4\*(n/2) +C5(n\*n/2) - C5\*(n/2) - C5 +C6(n\*n/2) - C6\*(n/2) - C6  
                           +C7\*(n\*n/2) -C7(n/2) -C7 + C8\*n  
                T(n) = n\*n \*(C4/2+C5/2+C6/2+C7/2) + n(C1+C2+C3+C4/2-C5/2-C6/2-C7/2 + C8) + 1\*(C1 - C5 - C6 - C7)  
                T(n) = O(n\*n)  or  
                T(n) = O(n^2)                           
                                    
             **For Best Case:**  
Actually Selection Sort  most probably take same time in best case that is sorted array as it take in worst Case.  
            So therefore the time complexity of selection sort in best case is O(n^2) i.e it not efficient than insertion sort.

**LAYING TV CABLES IN AN AREA USING MINIMUM SPANNING TREE**

**TEAM MEMBERS:**

D.STEFFY (140071601072)

R.TASNIM TABASUM (140071600180)

**PROBLEM IDENTIFICATION:**

This scenario is to lay cables in an area and the condition is to bury cables only in certain paths. Tv cable laying is an vast network. The cables are laid for the customers to watch television. The cable tv laying is an vast network it should be planned systematically and implemented in cost effective way. Everyone watches television to know what’s happening around them. People from rich to poor people all wants to gain knowledge so they have to watch television. The cables could be laid only in certain paths and it is represented through graph. There is an graph through which those points are connected and these graph has edges with weights. The major problem in this scenario is the laying of cables become more expensive because we lay it longer and deeper. To reduce this expensive problem and to telecast the channels in cost efficient way minimum spanning tree algorithm is used.

**ALGORITHM USED:**

The algorithm used for laying cables in an area with cost efficient way is minimum spanning tree. A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any undirected graph a minimum spanning forest, which is a union of the minimum spanning trees for its connected components. There are two algorithms used to find the minimum spanning tree of a connected graph G are Prim’s algorithm and Kruskal’s algorithms. Both of these algorithms are greedy algorithms. Prim’s algorithm is a Prim’s algorithm that finds a minimum spanning tree for a weighted undirected 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. Kruskal's algorithm is a [minimum-spanning-tree algorithm](https://en.wikipedia.org/wiki/Minimum_spanning_tree#Algorithms) which finds an edge of the least possible weight that connects any two trees in the forest. It is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) in [graph theory](https://en.wikipedia.org/wiki/Graph_theory) as it finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree) for a [connected](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)) [weighted graph](https://en.wikipedia.org/wiki/Glossary_of_graph_theory#Weighted_graphs_and_networks) adding increasing cost arcs at each step. This means it finds a subset of the [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a tree that includes every [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)), 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.

**Kruskal’s Pseudo code:**

KRUSKAL(G):

1 A = ∅

2 **foreach** v ∈ G.V:

3 MAKE-SET(v)

4 **foreach** (u, v) in G.E ordered by weight(u, v), increasing:

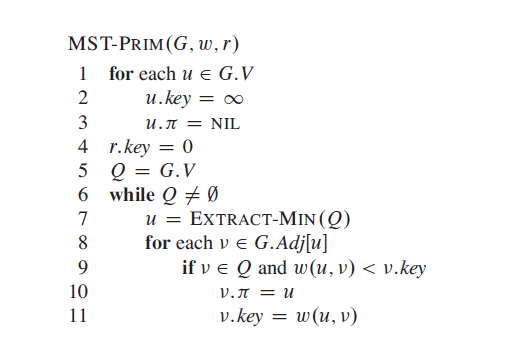
5 **if** FIND-SET(u) ≠ FIND-SET(v):

6 A = A ∪ {(u, v)}

7 UNION(u, v)

8 return

**Prim’s pseudo code:**



**PROBLEM**

**PROBLEM DEFINITION:**

A minimum spanning tree connects all nodes in a given graph. A minimum spanning tree must be connected and undirected graph. It can have weighted edges. Multiple minimum spanning trees can exist within a given graph.

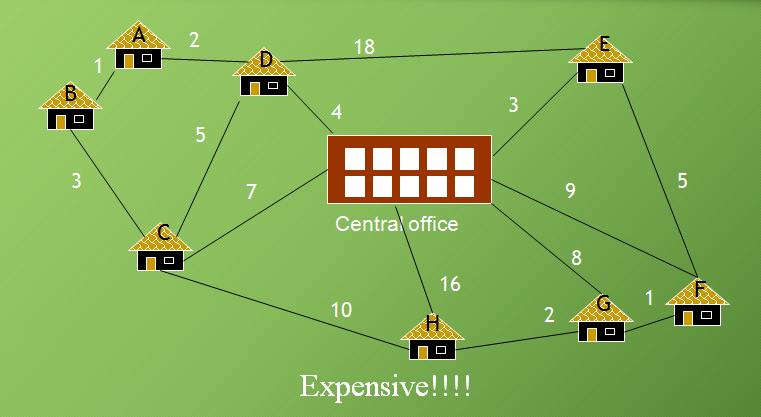
**PROBLEM DESCRIPTION:**

A cable TV company was planning to lay cables in an area and the condition is to bury the cable only in certain paths. There is a graph to represent which points are connected through those paths.

To bury cables in those paths they are very expensive, longer and deeper to lay. The graph of those paths is represented by edges with larger weights. A minimum spanning tree is used for implementing a problem with minimum cost.

**BACKGROUND OF THE PROBLEM:**

This TV cables laying scenario has 8 homes in an area. The central office of an tv channel works as a source vertex and sends the signal to the area. Every home receives signal from the central office and the time complexity is more. The cable connection if buried in such a way it is expensive. To avoid this expensive problem the alternative way is applied by using minimum spanning tree algorithm. Now through minimum spanning tree the cables are laid in cost efficient way. There are several minimum spanning trees of the same weights having same number of edges. If all the edge weights of a give graph are the same, then every spanning tree of that graph is minimum. If there are n vertices in the graph, then each tree has n-1 edges.



**ALGORITHM DESIGN**

**PROCEDURE:**

There are two algorithms to solve minimum spanning tree

* Prim’s Algorithm
* Kruskal’s Algorithm

**PRIMS’S ALGORITHM:**

**STEP 1:** Choose any element r and set S={r} and MST=NULL. Take r as a root of the spanning tree.

**STEP 2:** Find a lightest edge such that one end point is in S and the other is in V\S. Add this edge to A and it’s other point to S.

**STEP 3:** If V/S=NULL, then stop and output the minimum spanning tree.

**KRUSKAL’S ALGORITHM:**

**STEP 1:** Sort all edges in non-decreasing order of their weight**.**

**STEP 2:** Pick the smallest edge.

**STEP 3:** Check if it forms a cycle with Spanning Tree formed so far.

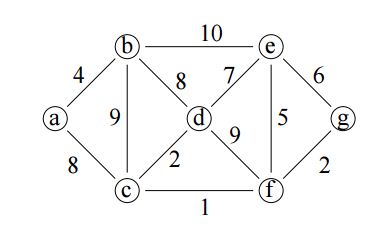
**STEP 4:** If cycle is not formed, include this edge else discard it.

**STEP 5:** Repeat Union-Find algorithm until there are V-1 edges in the spanning tree.

**EXAMPLE:**

**PROBLEM:1**

**Solve using Prim’s algorithm.**

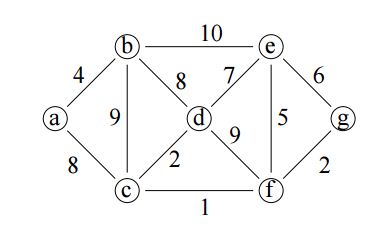
****

**SOLUTION:**

**STEP 1: select source vertex, It can be any node among the vertices.**

**Here, considering source vertex is a.**

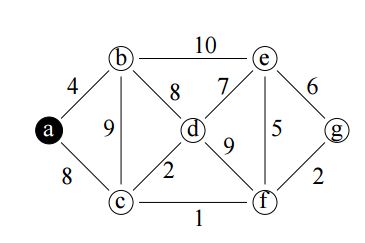
|  |  |  |  |
| --- | --- | --- | --- |
| **VERTEX** | **KNOWN** | ***dv*** | ***pv*** |
| **a** | **0** | **∞** | **0** |
| **b** | **0** | **∞** | **0** |
| **c** | **0** | **∞** | **0** |
| **d** | **0** | **∞** | **0** |
| **e** | **0** | **∞** | **0** |
| **f** | **0** | **∞** | **0** |
| **g** | **0** | **∞** | **0** |

****

**STEP 2:**

**Visit the source vertex a.**

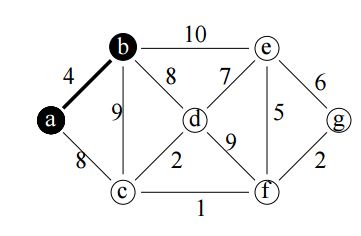
|  |  |  |  |
| --- | --- | --- | --- |
| **VERTEX** | **KNOWN** | ***dv*** | ***pv*** |
| **a** | **1** | **0** | **0** |
| **b** | **0** | **∞** | **0** |
| **c** | **0** | **∞** | **0** |
| **d** | **0** | **∞** | **0** |
| **e** | **0** | **∞** | **0** |
| **f** | **0** | **∞** | **0** |
| **g** | **0** | **∞** | **0** |

****

**STEP 3:**

**Select a node with minimum distance among the adjacent nodes of source vertex.**

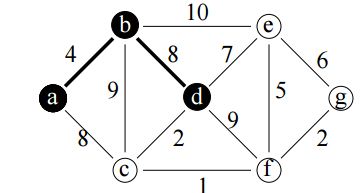
|  |  |  |  |
| --- | --- | --- | --- |
| **VERTEX** | **KNOWN** | ***dv*** | ***pv*** |
| **a** | **1** | **0** | **0** |
| **b** | **0** | **4** | **a** |
| **c** | **0** | **∞** | **0** |
| **d** | **0** | **∞** | **0** |
| **e** | **0** | **∞** | **0** |
| **f** | **0** | **∞** | **0** |
| **g** | **0** | **∞** | **0** |

****

**STEP 4:**

**Select a node with minimum distance among the adjacent nodes of recently visited vertex (i.e.) b**

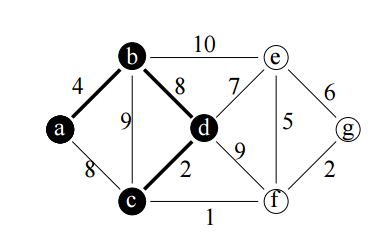
|  |  |  |  |
| --- | --- | --- | --- |
| **VERTEX** | **KNOWN** | ***dv*** | ***pv*** |
| **a** | **1** | **0** | **0** |
| **b** | **1** | **4** | **a** |
| **c** | **0** | **∞** | **0** |
| **d** | **0** | **8** | **b** |
| **e** | **0** | **∞** | **0** |
| **f** | **0** | **∞** | **0** |
| **g** | **1** | **∞** | **0** |

****

**STEP 5:**

**Select a node with minimum distance among the adjacent nodes of recently visited vertex (i.e.) d**

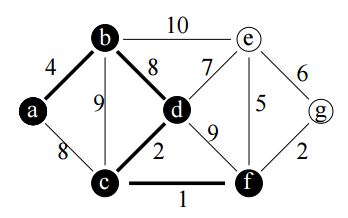
|  |  |  |  |
| --- | --- | --- | --- |
| **VERTEX** | **KNOWN** | ***dv*** | ***pv*** |
| **a** | **1** | **0** | **0** |
| **b** | **1** | **4** | **a** |
| **c** | **0** | **2** | **d** |
| **d** | **1** | **8** | **b** |
| **e** | **0** | **∞** | **0** |
| **f** | **0** | **∞** | **0** |
| **g** | **0** | **∞** | **0** |

****

**STEP 6:**

**Select a node with minimum distance among the adjacent nodes of recently visited vertex (i.e.) c**

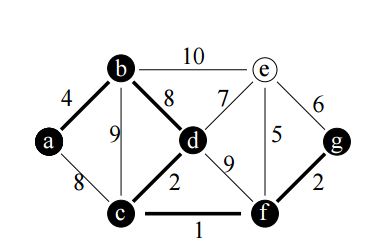
|  |  |  |  |
| --- | --- | --- | --- |
| **VERTEX** | **KNOWN** | ***dv*** | ***pv*** |
| **a** | **1** | **0** | **0** |
| **b** | **1** | **4** | **a** |
| **c** | **1** | **2** | **0** |
| **d** | **1** | **8** | **b** |
| **e** | **0** | **∞** | **0** |
| **f** | **0** | **1** | **c** |
| **g** | **0** | **∞** | **0** |

****

**STEP 7:**

**Select a node with minimum distance among the adjacent nodes of recently visited vertex (i.e.) f**

|  |  |  |  |
| --- | --- | --- | --- |
| **VERTEX** | **KNOWN** | ***dv*** | ***pv*** |
| **a** | **1** | **0** | **0** |
| **b** | **1** | **4** | **a** |
| **c** | **1** | **2** | **0** |
| **d** | **1** | **8** | **b** |
| **e** | **0** | **∞** | **0** |
| **f** | **1** | **1** | **c** |
| **g** | **0** | **2** | **f** |

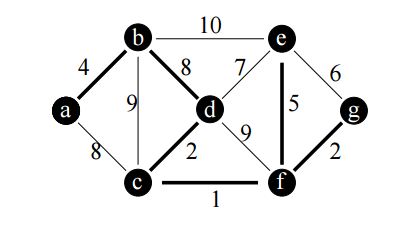
****

**STEP 8:**

**Select a node with minimum distance among the adjacent nodes of recently visited vertex (i.e.) g but from g cost is more than the cost from f. So, an edge (f,e) is taken in to MST.**

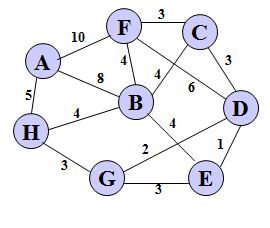
**Finally, e is also visited .It doesnot connect with vertex d because it forms a circle.**

|  |  |  |  |
| --- | --- | --- | --- |
| **VERTEX** | **KNOWN** | ***dv*** | ***pv*** |
| **a** | **1** | **0** | **0** |
| **b** | **1** | **4** | **a** |
| **c** | **1** | **2** | **0** |
| **d** | **1** | **8** | **b** |
| **e** | **1** | **5** | **f** |
| **f** | **1** | **1** | **c** |
| **g** | **1** | **2** | **f** |

****

**MINIMUM COST: 22**

**PROBLEM: 2 Solve using Kruskal’s algorithm.**

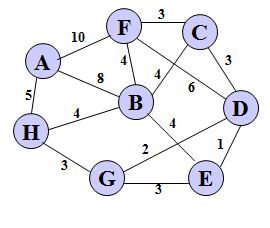
****

**SOLUTION:**

**STEP 1: Sort all edges by increasing order of their weight.**

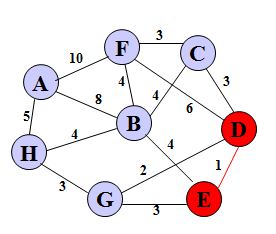
**{{A},{B},{C},{D},{E},{F},{G},{H}}**

|  |  |
| --- | --- |
| ***edge*** | ***dv*** |
| (D,E) | 1 |
| (D,G) | 2 |
| (E,G) | 3 |
| (C,D) | 3 |
| (G,H) | 3 |
| (C,F) | 3 |
| (B,C) | 4 |
| (B,E) | 4 |
| (B,F) | 4 |
| (B,H) | 4 |
| (A,H) | 5 |
| (D,F) | 6 |
| (A,B) | 8 |
| (A,F) | 10 |

****

**STEP 2:**

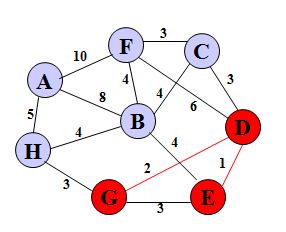
**Select an edge with minimum weight from the table. (i.e.) first row**

****

**{{D,E},{{A},{B},{C},F},{G},{H}}**

**STEP 3:**

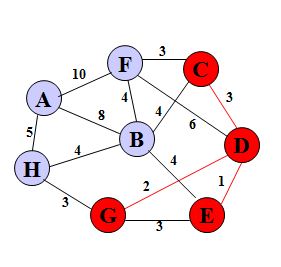
**Select a next edge with minimum weight from the table.**

****

**{{D,E,G },{A},{B},{C},{F},{H}}**

**STEP 4:**

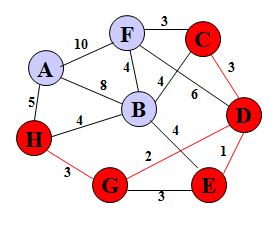
**Select a next edge with minimum weight from the table.**

****

**{{D,E,G,C },{A},{B},{F},{H}}**

**STEP 5:**

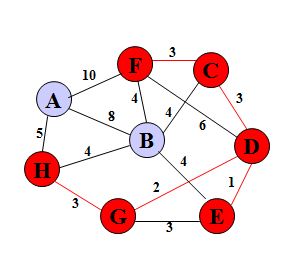
**Select a next edge with minimum weight from the table.**

****

**{{D,E,G,C,H },{A},{B},{F}}**

**STEP 6:**

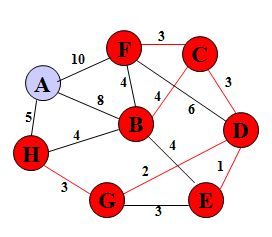
**Select a next edge with minimum weight from the table.**

****

**{{D,E,G,C,H,F },{A},{B}}**

**STEP 7:**

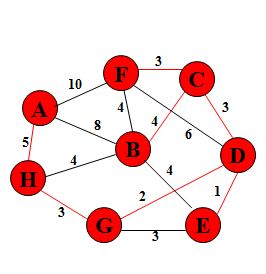
**Select a next edge with minimum weight from the table.**

****

**{{D,E,G,C,H,F ,B},{A}}**

**STEP 8:**

**Select a next edge with minimum weight from the table.**

****

**{D,E,G,C,H,F ,B,A}**

**Minimum cost :21**

**ALGORITHM ANALYSIS**

**PRIM’S ALGORITHM USING ADJACENCY MATRIX REPRESENTATION.**

An **Adjacency Matrix** representation of a graph constructs a V x V matrix (where V is the number of vertices). The value of cell (a, b) is the weight of the edge linking vertices a and b, or zero if there is no edge.

**Prim's Algorithm** is an algorithm that takes a graph and a starting node, and finds a minimum spanning tree on the graph - that is, it finds a subset of the edges so that the result is a tree that contains all the nodes and the combined edge weights are minimized. It may be summarized as follows:

1. Place the starting node in the tree.
2. Repeat until all nodes are in the tree:
   1. Find all edges that join nodes in the tree to nodes not in the tree.
   2. Of those edges, choose one with the minimum weight.
   3. Add that edge and the connected node to the tree.

**ANALYSIS**

We can now start to analyze the algorithm like so:

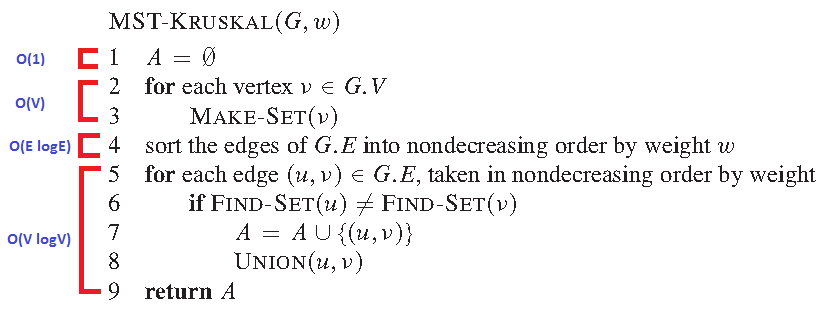
1. At every iteration of the loop, we add one node to the tree. Since there are V nodes, it follows that there are O (V) iterations of this loop.
2. Within each iteration of the loop, we need to find and test edges in the tree. If there are E edges, the naive searching implementation uses O (E) to find the edge with minimum weight.
3. So in combination, we should expect the complexity to be O (VE), which may be O (V^3) in the worst case.

Here the *distance* vector represents the smallest weighted edge joining each node to the tree, and is used as follows:

1. Initialize with the edge weights to the starting node A with complexity O (V).
2. To find the next node to add, simply find the minimum element of *distance* (and remove it from the list). This is O (V).
3. After adding a new node, there are O (V) new edges connecting the tree to the remaining nodes; for each of these determine if the new edge has less weight than the existing distance. If so, update the *distance* vector. Again, O (V).

Using these three steps reduces the searching time from O (E) to O (V), and adds an extra O(V) step to update the *distance* vector at each iteration. Since each iteration is now O (V), the overall complexity is O (V^2).

**KRUSKAL’S ALGORITHM:**



**Calculation:**

T (n) = O (1) + O(V) + O(E log E) + O(V log V)

= O (E log E) + O(V log V)

as |E| >= |V| - 1

T (n) = E log E + E log E

= E log E

*E* is the number of edges in the graph and *V* is the number of vertices, Kruskal's algorithm can be shown to run in [*O*](https://en.wikipedia.org/wiki/Big-O_notation)(*E* [log](https://en.wikipedia.org/wiki/Binary_logarithm) *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 log *V*2 log ⁡ V 2 = 2 log ⁡ V {\displaystyle \log V^{2}=2\log V} {\displaystyle \;} is *O*(log *V*).
* Each isolated vertex is a separate component of the minimum spanning forest. If we ignore isolated vertices we obtain *V* ≤ 2*E*, so log *V* is *O* (log *E*).

We can achieve this bound as follows: first sort the edges by weight using a [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort) in *O*(*E* log *E*) time; this allows the step "remove an edge with minimum weight from *S*" to operate in constant time. Next, we use a [disjoint-set data structure](https://en.wikipedia.org/wiki/Disjoint-set_data_structure) (Union Find) to keep track of which vertices are in which components. We need to perform O(*V*) operations, as in each iteration we connect a vertex to the spanning tree, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform O(*V*) operations in *O*(*V* log *V*) time. Thus the total time is *O*(*E* log *E*) = *O*(*E* log *V*).

**COMPARISON BETWEEN PRIM’S AND KRUSKAL’S ALGORITHM:**

In Prim's, you always keep a connected component, starting with a single vertex. You look at all edges from the current component to other vertices and find the smallest among them. You then add the neighboring vertex to the component, increasing its size by 1. In N-1 steps, every vertex would be merged to the current one if we have a connected graph.  
In Kruskal's, you do not keep one connected component but a forest. At each stage, you look at the globally smallest edge that does not create a cycle in the current forest. Such an edge has to necessarily merge two trees in the current forest into one. Since you start with N single-vertex trees, in N-1 steps, they would all have merged into one if the graph was connected. Both Prim's algorithm and Kruskal's algorithm are greedy algorithms for finding the Minimum Spanning Tree.

* Prim’s algorithm initializes with a node, whereas Kruskal’s algorithm initiates with an edge.
* Prim’s algorithms span from one node to another while Kruskal’s algorithm select the edges in a way that the position of the edge is not based on the last step.
* In prim’s algorithm, graph must be a connected graph while the Kruskal’s can function on disconnected graphs too.
* Prim’s algorithm has a time complexity of O(V2), and Kruskal’s time complexity is O(logV).
* In prim’s algorithm, Graph has to be connected.
* In Prim's algorithm, the next edge in the MST shall be the cheapest edge in the current vertex. In Kruskal's algorithm, we will choose the cheapest edge.
* Prim's algorithm is found to run faster in dense graphs with more number of edges than vertices, whereas Kruskal's algorithm is found to run faster in sparse graphs.

**IMPLEMENTATION**

**TURBO C++ EDITOR:**

**INTRODUCTION:**

Turbo C++ is a discontinued C++ compiler and integrated development environment and computer language originally from Borland. Most recently it was distributed by Embarcadero Technologies, which acquired all of Borland's compiler tools with the purchase of its Code Gear division in 2008. The original Turbo C++ product line was put on hold after 1994 and was revived in 2006 as an introductory-level IDE, essentially a stripped-down version of their flagship C++Builder.

**RELEASE**:

Turbo C++ 2006 was released on September 5, 2006 and was available in 'Explorer' and 'Professional' editions. The Explorer edition was free to download and distribute while the Professional edition was a commercial product. In October 2009 Embarcadero Technologies discontinued support of its 2006 C++ editions. As such, the Explorer edition is no longer available for download and the Professional edition is no longer available for purchase from Embarcadero Technologies. Turbo C++ is succeeded by C++Builder. The first release of Turbo C++ was made available during the MS-DOS era on personal computers.

**VERSION 1.0:**

Version 1.0, running on MS-DOS, was released in May 1990. An OS/2 version was produced as well. Version 1.01 was released on February 28, 1991, running on MS-DOS. The latter was able to generate both COM and EXE programs and was shipped with Borland's Turbo Assembler compiler for Intel x86 processors. The initial version of the Turbo C++ compiler was based on a front end developed by TauMetric (TauMetric was later acquired by Sun Microsystems and their front end was incorporated in Sun C++ 4.0, which shipped in 1994).This compiler supported the AT&T 2.0 release of C++.

**VERSION 3.0:**

Turbo C++ 3.0 was released in 1991 (shipping on November 20), and came in amidst expectations of the coming release of Turbo C++ for Microsoft Windows. Initially released as an MS-DOS compiler, 3.0 supported C++ templates, Borland's inline assembler, and generation of MS-DOS mode executables for 8086 real mode and 286 protected modes (as well as the Intel 80186.) 3.0 implemented AT&T C++ 2.1, the most recent at the time. The separate Turbo Assembler product was no longer included, but the inline-assembler could stand in as a reduced functionality version. Soon after the release of Windows 3.0, Borland updated Turbo C++ to support Windows application development.

**VERSION 3.1:**

The Turbo C++ 3.0 for Windows product was quickly followed by Turbo C++ 3.1 (and then Turbo C++ 4.5). It's possible that the jump from version 1.x to version 3.x was in part an attempt to link Turbo C++ release numbers with Microsoft Windows versions; however, it seems more likely that this jump was simply to synchronize Turbo C and Turbo C++, since Turbo C 2.0 (1989) and Turbo C++ 1.0 (1990) had come out roughly at the same time, and the next generation 3.0 was a merger of both the C and C++ compiler. Starting with version 3.0, Borland segmented their C++ compiler into two distinct product-lines: "Turbo C++" and "Borland C++". Turbo C++ was marketed toward the hobbyist and entry-level compiler market, while Borland C++ targeted the professional application development market. Borland C++ included additional tools, compiler code-optimization, and documentation to address the needs of commercial developers. Turbo C++ 3.0 could be upgraded with separate add-ons, such as Turbo Assembler and Turbo Vision 1.0.

**VERSION 4.0:**

Version 4.0 was released in November 1993 and was notable for its robust support of templates. In particular, Borland C++ 4 was instrumental in the development of the Standard Template Library, expression templates, and the first advanced applications of template meta programming. With the success of the Pascal-evolved product Delphi, Borland ceased work on their Borland C++ suite and concentrated on C++Builder for Windows. C++Builder shared Delphi's front-end application framework, but retained the Borland C++ back-end compiler. Active development on Borland C++/Turbo C++ was suspended until 2006

**Legacy software**

Turbo C++ v1.01 and Turbo C v2.01 can be downloaded, free of charge, from Borland's Antique Software website.

Turbo C 3.0 (DOS) was included in the Turbo C Suite 1.0, which is no longer sold by Borland.

**INTRODUCTION TO C**

C is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system. C is the most widely used computer language. It keeps fluctuating at number one scale of popularity along with Java programming language, which is also equally popular and most widely used among modern software programmers programming is a powerful general-purpose language. It is fast, portable and available in all platforms’ belongs to the structured, procedural paradigms of languages. It is proven, flexible and powerful and may be used for a variety of different applications. Although high-level, C and assembly language share many of the same attributes’ is a general-purpose, imperative computer programming language, supporting structured programming, lexical variable scope and recursion, while a static type system prevents many unintended operations. By design, C provides constructs that map efficiently to typical machine instructions, and therefore it has found lasting use in applications that had formerly been coded in assembly language, including operating systems, as well as various application software for computers ranging from supercomputers to embedded systems.

**DEVELOPED BY**

C was originally developed by Dennis Ritchie between 1969 and 1973 at Bell Labs, and used to re-implement the Unix operating system. It has since become one of the most widely used programming languages of all time,with C compilers from various vendors available for the majority of existing computer architectures and operating systems. C has been standardized by the American National Standards Institute (ANSI) since 1989 (see ANSI C) and subsequently by the International Organization for Standardization (ISO).C is an imperative procedural language. It was designed to be compiled using a relatively straightforward compiler, to provide low-level access to memory, to provide language constructs that map efficiently to machine instructions, and to require minimal run-time support. Therefore, C was useful for many applications that had formerly been coded in assembly language, for example in system programming.

**SYSTEM PROGRAMMING**

C is widely used for "system programming", including implementing operating systems and embedded system applications, because C code, when written for portability, can be used for most purposes, yet when needed, system-specific code can be used to access specific hardware addresses and to perform type punning to match externally imposed interface requirements, with a low run-time demand on system resources’ can also be used for website programming using CGI as a "gateway" for information between the Web application, the server, and the browser is often chosen over interpreted languages because of its speed, stability, and near-universal availability.

**AVAILABILITY & EFFIICIENCY**

One consequence of C's wide availability and efficiency is that compilers, libraries and interpreters of other programming languages are often implemented in C. The primary implementations of Python, Perl 5 and PHP, for example, are all written in C.

**INTERMEDIATE LANGUAGE**

C is sometimes used as an intermediate language by implementations of other languages. This approach may be used for portability or convenience; by using C as an intermediate language, additional machine-specific code generators are not necessary. C has some features, such as line-number preprocessor directives and optional superfluous commas at the end of initializer lists, that support compilation of generated code. C has also been widely used to implement end-user applications

**PROGRAM TO FIND MINIMUM SPANNING TREE USING PRIM’S ALGORITHM:**

#include<stdio.h>

int min\_cost=0;

void main()

{

int n,i,j,min,cost[20][20],a,u,b,v,source,visited[20],EDGE=1;

clrscr();

printf("Enter the no. of nodes:");

scanf("%d",&n);

printf("Enter the costmatrix:\n");

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{

scanf("%d",&cost[i][j]);

}

}

for(i=1;i<=n;i++)

visited[i]=0;

printf("Enter the root node:");

scanf("%d",&source);

visited[source]=1;

printf("\nMinimum cost spanning treeis\n");

while(EDGE<n)

{

min=999;

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{

if(cost[i][j]<min)

if(visited[i]==0)

continue;

else

{

min=cost[i][j];

a=u=i;

b=v=j;

}

}

}

if(visited[u]==0||visited[v]==0)

{

printf("\nEdge %d\t(%d->%d)=%d\n",EDGE++,a,b,min);

min\_cost=min\_cost+min;

visited[b]=1;

}

cost[a][b]=cost[b][a]=999;

}

printf("\nMinimumcost=%d\n",min\_cost);

getch();

}

**PROGRAM TO FIND MINIMUM SPANNING TREE USING KRUSKAL’S ALGORITHM:**

#include<stdio.h>

int noofedges=1,min\_cost=0;

void main()

{

int n,i,j,min,a,u,b,v,cost[20][20],parent[20];

clrscr();

printf("Enter the no. of vertices:");

scanf("%d",&n);

printf("\nEnter the cost matrix:\n");

for(i=1;i<=n;i++)

for(j=1;j<=n;j++)

scanf("%d",&cost[i][j]);

for(i=1;i<=n;i++)

parent[i]=0;

printf("\nThe edges of spanning tree are\n"); while(noofedges<n)

{

min=999;

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{if(cost[i][j]<min)

{min=cost[i][j];

a=u=i;

b=v=j;

}}}

while(parent[u])

u=parent[u];

while(parent[v])

v=parent[v];

if(u!=v)

{

printf("Edge %d\t(%d->%d)=%d\n",noofedges++,a,b,min);

min\_cost=min\_cost+min;

parent[v]=u;

}

cost[a][b]=cost[a][b]=999;

}

printf("\nMinimum cost=%d\n",min\_cost);

getch();

}

**JUSTIFICATION:**

The main objective of this project is to lay a cable in an area with minimum cost. Cable laying company wants to lay a cable in a particular area. But, there is a condition for them to lay a cable in particular paths. So, cable laying company mapped their paths with the graph. While analyzing those graph, they found that the cost to lay a cable in that particular area is very expensive. In order to reduce the cost spending on laying cables in an area, there is a need to find a path which will reduce the cost for implementation. This problem is related to minimum spanning tree concept .

To find minimum spanning tree, two algorithms are used. Prim’s and Kruskal’s algorithm are used. Since this scenario related to spanning tree concept, same algorithms are used here to find minimum path which will reduce cost to lay a cable. Graph which represents the paths in an area is just similar to dense graphs. Among these two algorithms, prim’s algorithm is suitable for this scenario because, prim’s algorithm works well in dense graphs rather than sparse graphs.

**CONCLUSION**

The main problem for the cable company is to lay a cable in an area .There is a condition for them to lay a cable in a certain paths. These cables are laid for the customers to watch television .In this scenario, the major problem is the laying of cables become more expensive. To reduce this problem, minimum spanning tree concept is implemented. The algorithm is used to find minimum spanning trees are prim’s algorithm and kruskal’s algorithm. Here in this scenario, prims algorithm is used because prims algorithm works well in dense graphs. Prim’s algorithm is an algorithm that takes a graph and a starting node and finds a minimum spanning tree on the graph that will have all vertices and the edges which has minimum cost.