Vision of the Department

To be recognized for keeping innovation, research and excellence abreast of learning in the field of computer science & engineering to cater the global society.

Mission of the Department

M1: To provide an exceptional learning environment with academic excellence in the field of computer science and engineering.

M2: To facilitate the students for research and innovation in the field of software, hardware and computer applications and nurturing to cater the global society.

M3: To establish professional relationships with industrial and research organisations to enable the students to be updated of the recent technological advancements.

M4: To groom the learners for being the software professionals catering the needs of modern society with ethics, moral values and full of patriotism.

Program Educational Objectives (PEO's)

PEO1: The graduate will have the knowledge and skills of major domains of computer science and engineering in providing solution to real world problems most efficiently.

PEO2: The graduate will be able to create and use the modern tools and procedures followed in the software industry in the relevant domain.

PEO3: The graduate will be following the ethical practices of the software industry and contributing to the society as a responsible citizen.

PEO4: The graduate will have the innovative mindset of learning and implementing the latest developments and research outcomes in the computer hardware and software to keep pace with the fast changing socio economic world.

LAKSHMI NARAIN COLLEGE OF TECHNOLOGY & SCIENCE, BHOPAL
CS-402
ANALYSIS & DESIGN OF ALGORITHM

LAB OUTCOMES

Student will be able to

CO1: Design new algorithms. Prove them correct, and analyze their asymptotic and Absolute runtime and memory demand.

CO2:. Find an algorithm to solve the problem(create) and prove that the algorithm solve the problem correctly.

CO3: Apply backtracking techniques for solving eight-queen problem.

CO4: Implement branch and bound methods to solve travelling salesman problem.

CO5: Solve knapsack problem using dynamic programming algorithm.

LIST OF EXPERIMENTS

1. Write a program for Iterative and Recursive Binary Search.
2. Write a program for Merge Sort.
3. Write a program for Quick Sort.
4. Write a program for Strassen's Matrix Multiplication
5. Write a program for optimal merge patterns.
6. Write a program for Huffman coding.
7. Write a program for minimum spanning trees using Kruskal's algorithm.
8. Write a program for minimum spanning trees using Prim's algorithm.
9. Write a program for single sources shortest path algorithm
10. Write a program for traveling salesman problem.

Experiment No. 1

THEORY:

Binary Search

Solve a problem, either directly because solving that instance is easy (typically, because the instance is small) or by dividing it into two or more smaller instances. Each of these smaller instances is recursively solved, and the solutions are combined to produce a solution for the original instance. The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished. If the target value is less than the middle element's value, then the search continues on the lower half of the array; or if the target value is greater than the middle element's value, then the search continues on the upper half of the array. This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (and its associated element position is returned), or until the entire array has been searched (and "not found" is returned).

Characteristic

Every iteration eliminates half of the remaining possibilities. This makes binary searches very efficient - even for large collections. Our implementation relies on recursion, though it is equally as common to see an iterative approach. Binary search requires a sorted collection. This means the collection must either be sorted before searching, or inserts/updates must be smart. Also, binary searching can only be applied to a collection that allows random access (indexing).

In Real World

Binary searching is frequently used to its performance characteristics over large collections. The only time binary searching doesn't make sense is when the collection is being frequently updated, since resorting
will be required.

Recursive Binary Search Algorithm

A straightforward implementation of binary search is recursive. The initial call uses the indices of the entire array to be searched. The procedure then calculates an index midway between the two indices, determines which of the two sub arrays to search, and then does a recursive call to search that sub array. Each of the calls is tail recursive, so a compiler need not make a new stack frame for each call. The variables min and max are the lowest and highest inclusive indices that are searched.

```
int binary_search(int A[], int key, int min, int max)

{ // test if array is empty
```

```
if (max < min)
// set is empty, so return value showing not found
return KEY_NOT_FOUND;
else
{ // calculate midpoint to cut set in half
int mid = midpoint(min, max);
// three-way comparison
if (A[mid] > key)
// key is in lower subset
return binary_search(A, key, min, mid - 1);
else if (A[mid] < key)
// key is in upper subset
return binary_search(A, key, mid + 1, max);
else
// key has been found
return mid;
} }
```

It is invoked with initial min and max values of 0 and n-1 for a zero based array of length n.

Iterative Binary Search Algorithm

The binary search algorithm can also be expressed iteratively with two index limits that progressively narrow the search range.

```
int binary_search(int A[], int key, int min, int max)
{
// continue searching while [min,max] is not empty
while (min <= max)
{
// calculate the midpoint for roughly equal partition
int mid = midpoint(min, max);
if (A[mid] == key)
// key found at index mid
return mid;
// determine which subarray to search
else if (A[mid] < key)
// change min index to search upper subarray
min = mid + 1;

else
// change max index to search lower subarray
max = mid - 1;
}
// key was not found
return KEY_NOT_FOUND;}
```

Experiment No. 2

THEORY

Merge Sort:A merge sort is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Merge sort is a divide and conquer algorithm.

Merge sort works as follows:

1. Divide the unsorted list into two sub lists of about half the size
2. Sort each of the two sub lists
3. Merge the two sorted sub lists back into one sorted list.

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, 1, n).
Merge sort is based on the divide-and-conquer paradigm. Since we are dealing with subproblems, we state each subproblem as sorting a subarray A[p .. r]. Initially, p = 1 and r = n, but these values change as
we recurse through subproblems.
To sort A[p .. r]:

1. Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two subarrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That
is, q is the halfway point of A[p .. r].

2. Conquer Step

Conquer by recursively sorting the two subarrays A[p .. q] and A[q + 1 .. r].

3. Combine Step

Combine the elements back in A[p .. r] by merging the two sorted subarrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Algorithm:

MERGE-SORT (A, p, r)

IF p < r // Check for base case

THEN q = FLOOR[(p + r)/2] // Divide step
MERGE (A, p, q) // Conquer step.
MERGE (A, q + 1, r) // Conquer step.
MERGE (A, p, q, r) // Conquer step.

ANALYSIS

The straightforward version of function *merge* requires at most $2n$ steps (n steps for copying the sequence
to the intermediate array *b,* and at most n steps for copying it back to array *a*). The time complexity of *mergesort* is therefore
$T(n) \leq 2n + 2\ T(n/2)$ and
$T(1) = 0$
The solution of this recursion yields
$T(n) \leq 2n \log(n)\ \varepsilon\ O(n \log(n))$

Worst case performance	O(n log n)
Best case performance	O(n log n) typical, O(n) natural variant
Average case performance	O(n log n)
Worst case space complexity	O(n) total, O(n) auxiliary
Data structure	Array

Experiment No. 3

THEORY

Quick Sort: Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare that, on average, makes (bigO notation) comparisons to sort *n* items. However, in the worst case, it makes comparisons. Typically, quicksort is significantly faster in practice than other algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices which minimize the possibility of requiring quadratic time. Quicksort is a comparison sorting algorithm.Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

Pick an element, called a pivot, from the list.

Reorder the list so that all elements which are less than pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the

pivot is in its final position. This is called the partition operation.

Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Pseudocode For partition(a, left, right, pivotIndex)

pivotValue := a[pivotIndex]

swap(a[pivotIndex], a[right]) // Move pivot to end

storeIndex := left

for i from left to right-1

if a[i] ≤ pivotValue

swap(a[storeIndex], a[i])

storeIndex := storeIndex + 1

swap(a[right], a[storeIndex]) // Move pivot to its final place

return storeIndex

Pseudocode For quicksort(a, left, right)

if right > left

select a pivot value a[pivotIndex]

pivotNewIndex := partition(a, left, right, pivotIndex)

quicksort(a, left, pivotNewIndex-1)

quicksort(a, pivotNewIndex+1, right)

ANALY SIS

The partition routine examines every item in the array at most once, so complexity is clearly **O(*n*)**. Usually, the partition routine will divide the problem into two roughly equal sized partitions. We know that we can divide *n* items in half $\log_2 n$ times.

This makes quicksort a **O(*n*log*n*)** algorithm - equivalent to heapsort.

Experiment No. 4

THEORY

Strassen's Matrix Multiplication

The Strassen's method of matrix multiplication is a typical divide and conquer algorithm. In which how
a large matrices used for multiplication with the help of small sizes matrices.
Here A, and B are the given matrices and C is the result matrix C=AxB

We partition A,B,C into equally sized block matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

For such we divides matrix into smaller size matrix and calculate following 7 equations.
Now comes the important part. we define new matrices

$$\begin{aligned}
\mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}$$

Now using above 7 equations calculate the resultant matrix elements as follows:

$$
\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}
$$

The standard matrix multiplications takes

$n^3 = n^{\log_2 8}$

multiplications of the elements in the field *F*. We ignore the additions needed because, depending on *F*,they can be much faster than the multiplications in computer implementations, especially if the sizes of the matrix entries exceed the word size of the machine.

With the Strassen algorithm we can reduce the number of multiplications to

$n^{\log_2 7} \approx n^{2.807}$.

The reduction in the number of multiplications reduced numeric stability.

Experiment No. 5

THEORY

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time. If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as 2-way merge patterns. As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged. To merge a p-record file and a q-record file requires possibly p + q record moves, the obvious choice being, merge the two smallest files together at each step. Two-way merge patterns can be represented by binary merge trees. Let us consider a set of n sorted files {f1, f2, f3, …,fn}. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

ALGORITHM:

At every stage we merge the files of the least length.

Steps:

```
Create a min heap of the set of elements.
While(heap has more than one element)
{
Delete a minimum element from the heap, and store it in min1;
Delete a minimum element from the heap, and store it in min2;
Create a node with the fields (info, left_link, right_ink);
Let info.node = min1 + min2;
Let left_link.node = min1;
Let right_link_node = min2;
Insert node with valued info into the heap;
}
struct treenode {
struct treenode *lchild, *rchild;
int weight;
};
typedef struct treenode Type;

Type *Tree(int n)
// list is a global list of n single node
// binary trees as described above.
{
for (int i=1; i<n; i++) {
Type *pt = new Type;
// Get a new tree node.
pt -> lchild = Least(list); // Merge two trees with
pt -> rchild = Least(list); // smallest lengths.
pt -> weight = (pt->lchild)->weight
+ (pt->rchild)->weight;
```

```
Insert(list, *pt);
}
return (Least(list)); // Tree left in l is the merge tree.
}
```

PROGRAM

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int i,k,a[10],c[10],n,l;
cout<<"Enter the no. of elements\t";
cin>>n;
cout<<"\nEnter the sorted elments for optimal merge pattern";
for(i=0;i<n;i++)
{
cout<<"\t";
cin>>a[i];
}
i=0;k=0;
c[k]=a[i]+a[i+1];
i=2;
while(i<n)
{
k++;
if((c[k-1]+a[i])<=(a[i]+a[i+1]))
{
c[k]=c[k-1]+a[i];
}
else
{
c[k]=a[i]+a[i+1];
i=i+2;
while(i<n)
{ k++;
if((c[k-1]+a[i])<=(c[k-2]+a[i]))
{
c[k]=c[k-1]+a[i];
}
else
{
c[k]=c[k-2]+a[i];
}i++;
}
}i++;
```

```
}
k++;
c[k]=c[k-1]+c[k-2];
cout<<"\n\nThe optimal sum are as follows......\n\n";
for(k=0;k<n-1;k++)
{
cout<<c[k]<<"\t";
}
l=0;
for(k=0;k<n-1;k++)
{
l=l+c[k];
}
cout<<"\n\n The external path length is ......"<<l;
getch();
}
```

Experiment No. 6

THEORY:

Huffman coding is lossless data compression algorithm. In this algorithm a variable-length code is assigned to input different characters. The code length is related with how frequently characters are used. Most frequent characters have smallest codes, and longer codes for least frequent characters.

There are mainly two parts. First one to create Huffman tree, and another one to traverse the tree to find codes.

For an example, consider some strings "YYYZXXYYX", the frequency of character Y is larger than X and the character Z has least frequency. So the length of code for Y is smaller than X, and code for X will be smaller than Z.

- Complexity for assigning code for each character according to their frequency is O(n log n)

Input − A string with different characters, say "ACCEBFFFFAAXXBLKE"**Output** − Code for different characters:

Data: K, Frequency: 1, Code: 0000

Data: L, Frequency: 1, Code: 0001

Data: E, Frequency: 2, Code: 001

Data: F, Frequency: 4, Code: 01

Data: B, Frequency: 2, Code: 100

Data: C, Frequency: 2, Code: 101

Data: X, Frequency: 2, Code: 110

Data: A, Frequency: 3, Code: 111

ALGORITHM

huffmanCoding(string)

Input − A string with different characters.

Output − The codes for each individual characters.

```
Begin
  define a node with character, frequency, left and right child of the node for Huffman tree.
  create a list 'freq' to store frequency of each character, initially all are 0
  for each character c in the string do
    increase the frequency for character ch in freq list.
  done
```

```
  for all type of character ch do
     if the frequency of ch is non zero then add ch and its frequency as a node of priority queue Q.
  done
  while Q is not empty do
   remove item from Q and assign it to left child of node
   remove item from Q and assign to the right child of node
   traverse the node to find the assigned code
  done
End
```

traverseNode(n: node, code)

Input − The node n of Huffman tree, and code assigned from previous call

Output − Code assigned with each character

```
if left child of node n ≠ φ then
  traverseNode(leftChild(n), code+'0') //traverse through the left child
  traverseNode(rightChild(n), code+'1') //traverse through the right child
else
  display the character and data of current node.
```

PROGRAM:

```
#include<iostream>
#include<queue>
#include<string>
using namespace std;
struct node{
  int freq;
  char data;
  const node *child0, *child1;
  node(char d, int f = -1){ //assign values in the node
     data = d;
     freq = f;
     child0 = NULL;
     child1 = NULL;
```

```
    }
    node(const node *c0, const node *c1){
      data = 0;
      freq = c0->freq + c1->freq;
      child0=c0;
      child1=c1;
    }
    bool operator<( const node &a ) const { //< operator performs to find priority in queue
      return freq >a.freq;
    }
    void traverse(string code = "")const{
      if(child0!=NULL){
        child0->traverse(code+'0'); //add 0 with the code as left child
        child1->traverse(code+'1'); //add 1 with the code as right child
      }else{
        cout << "Data: " << data<< ", Frequency: "<<freq << ", Code: " << code<<endl;
      }
    }
};
void huffmanCoding(string str){
    priority_queue<node> qu;
    int frequency[256];
    for(int i = 0; i<256; i++)
      frequency[i] = 0; //clear all frequency
    for(int i = 0; i<str.size(); i++){
      frequency[int(str[i])]++; //increase frequency
    }
    for(int i = 0; i<256; i++){
      if(frequency[i]){
        qu.push(node(i, frequency[i]));
      }
```

```
   }
   while(qu.size() >1){
      node *c0 = new node(qu.top()); //get left child and remove from queue
      qu.pop();
      node *c1 = new node(qu.top()); //get right child and remove from queue
      qu.pop();
      qu.push(node(c0, c1)); //add freq of two child and add again in the queue
   }
   cout << "The Huffman Code: "<<endl;
   qu.top().traverse(); //traverse the tree to get code
}
main(){
   string str = "ACCEBFFFFAAXXBLKE"; //arbitray string to get frequency
   huffmanCoding(str);
}
```

Output

```
The Huffman Code:
Data: K, Frequency: 1, Code: 0000
Data: L, Frequency: 1, Code: 0001
Data: E, Frequency: 2, Code: 001
Data: F, Frequency: 4, Code: 01
Data: B, Frequency: 2, Code: 100
Data: C, Frequency: 2, Code: 101
Data: X, Frequency: 2, Code: 110
Data: A, Frequency: 3, Code: 111
```

Experiment No. 7

THEORY

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

How does Kruskal's algorithm work?

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -First, sort all the edges from low weight to high.

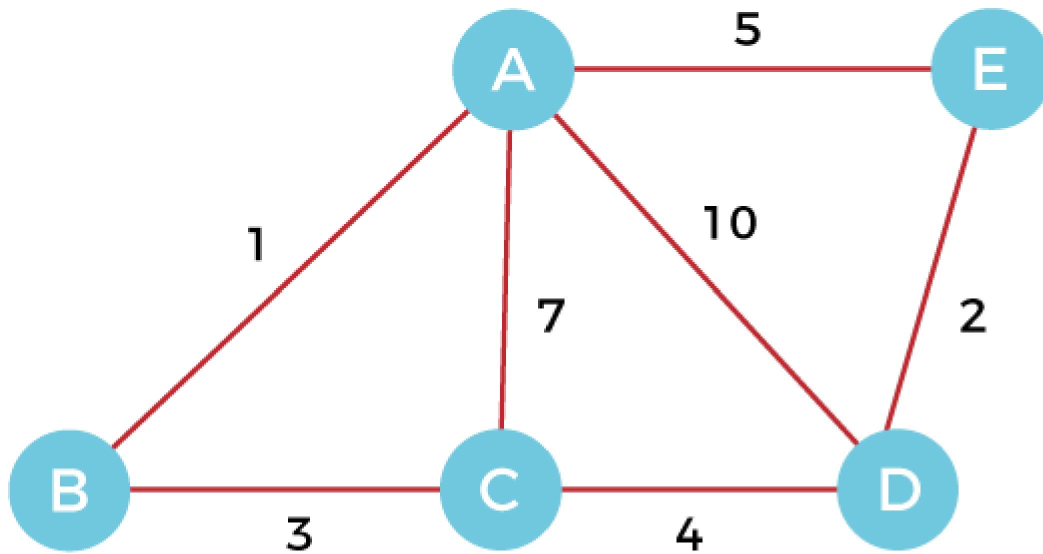
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

The applications of Kruskal's algorithm are -

- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

Example of Kruskal's algorithm

- Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.
- Suppose a weighted graph is -



The weight of the edges of the above graph is given in the below table -

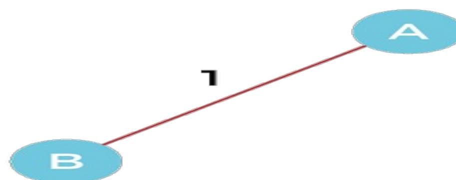
Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Now, sort the edges given above in the ascending order of their weights.

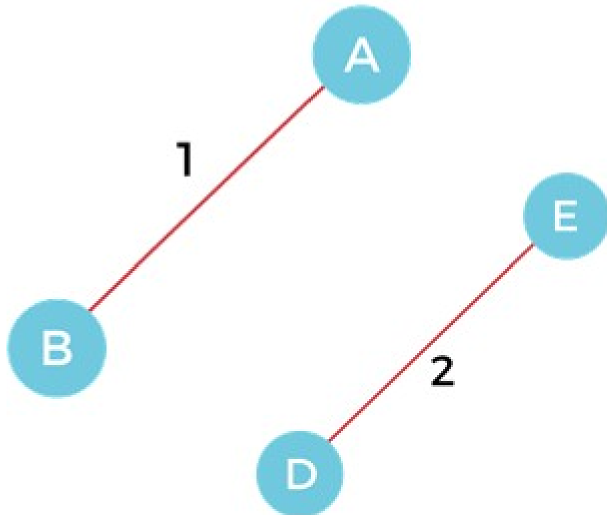
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree.

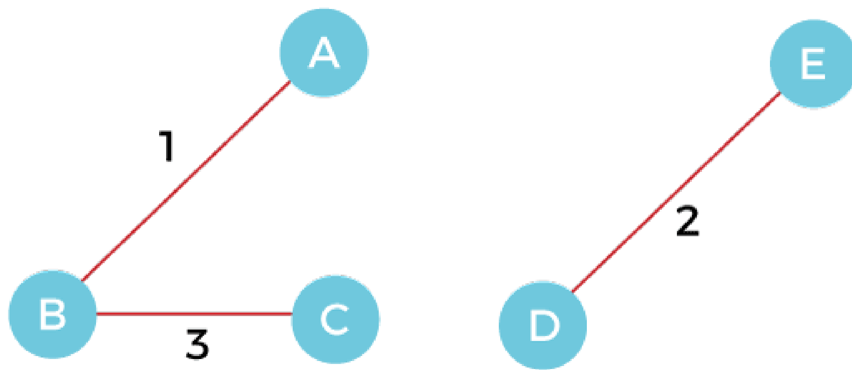
Step 1 - First, add the edge **AB** with weight **1** to the MST.



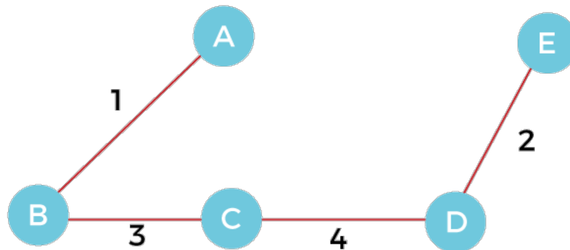
Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

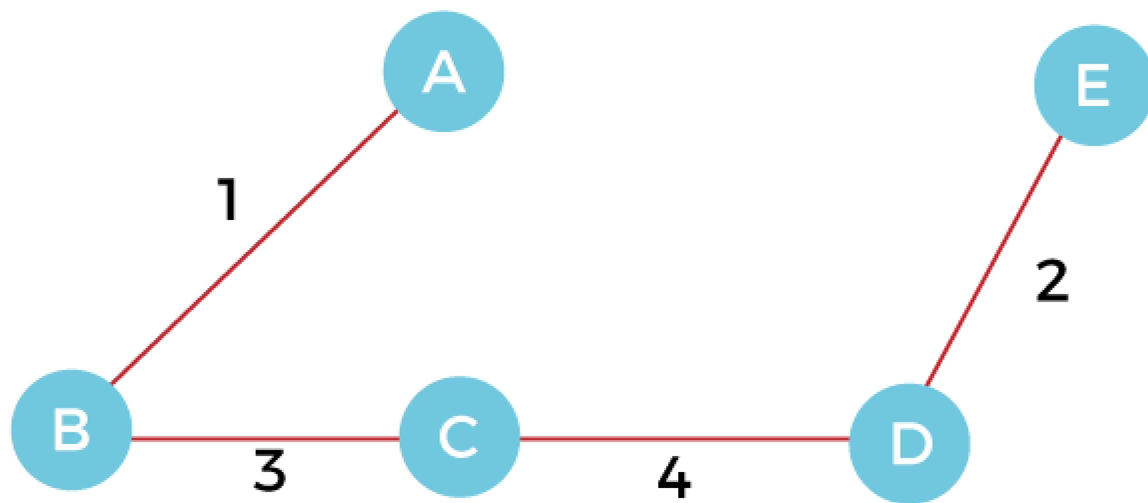


Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

Algorithm

Step 1: Create a forest F in such a way that every vertex of the graph is a separate treeStep 2: Create a set E that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while E is NOT EMPTY and F is not spanning

Step 4: Remove an edge from E with minimum weight

Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F

(**for** combining two trees into one tree).

ELSE

Discard the edge

Step 6: END

Experiment No. 8

THEORY

Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

Steps for finding MST using Prim's Algorithm:

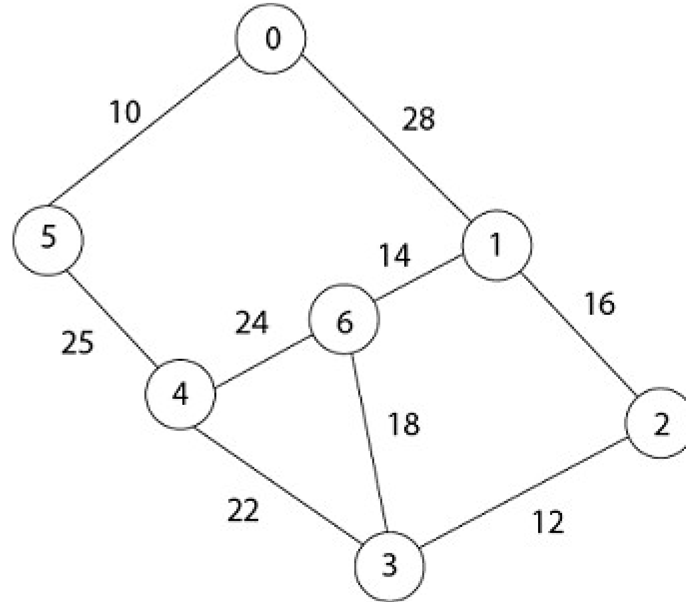
1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.
   a. Pick vertex u which is not is MST set and has minimum key value. Include 'u'to MST set.
   b. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

MST-PRIM (G, w, r)

1. for each u ∈ V [G]
2. do key [u] ← ∞
3. π [u] ← NIL
4. key [r] ← 0
5. Q ← V [G]
6. While Q ? ∅
7. do u ← EXTRACT - MIN (Q)
8. for each v ∈ Adj [u]
9. do if v ∈ Q and w (u, v) < key [v]

10. then π [v] ← u
11. key [v] ← w (u, v)

Example: Generate minimum cost spanning tree for the following graph using Prim's .



Solution: In Prim's algorithm, first we initialize the priority Queue Q. to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT - MIN (Q) procure, now u = r and Adj [u] = {5, 1}.

Removing u from set Q and adds it to set V - Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

Vertex	0	1	2	3	4	5	6
Key Value	0	∞	∞	∞	∞	∞	∞
Parent	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Taking 0 as starting vertex

Root = 0

Adj [0] = 5, 1

Parent, π [5] = 0 and π [1] = 0

Key [5] = ∞ and key [1] = ∞

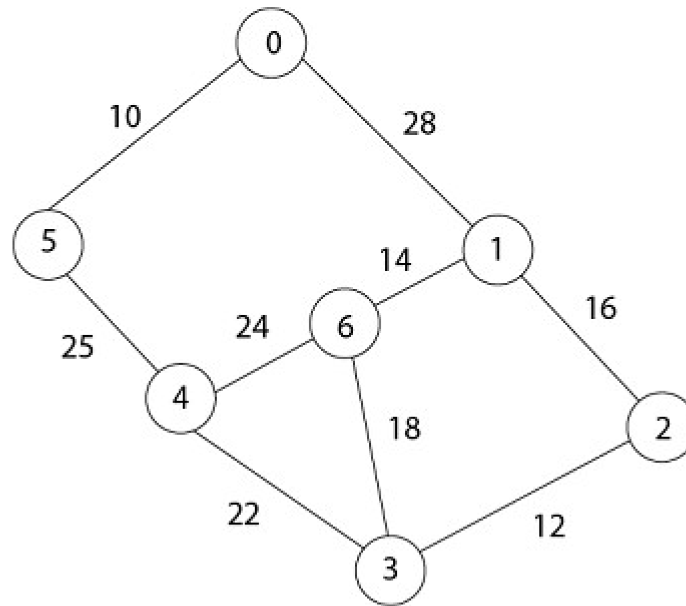
w [0, 5) = 10 and w (0,1) = 28

w (u, v) < key [5] , w (u, v) < key [1]

Key [5] = 10 and key [1] = 28

So update key value of 5 and 1 is:

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	∞	10	∞
Parent	NIL	0	NIL	NIL	NIL	0	NIL



Now by EXTRACT_MIN (Q) Removes 5 because key [5] = 10 which is minimum so u = 5.

Adj [5] = {0, 4} and 0 is already in heap

Taking 4, key [4] = ∞   π [4] = 5

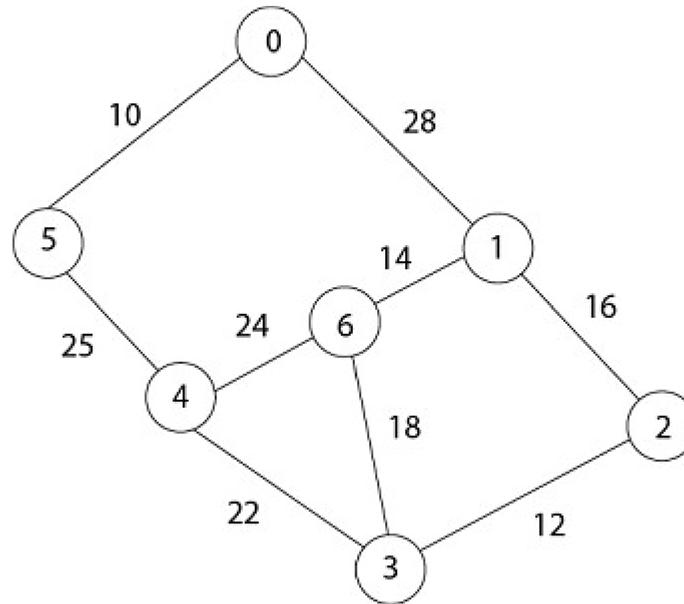
(u, v) < key [v] then key [4] = 25

w (5,4) = 25

w (5,4) < key [4]

date key value and parent of 4.

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	25	10	∞
Parent	NIL	0	NIL	NIL	5	0	NIL



Now remove 4 because key [4] = 25 which is minimum, so u =4

Adj [4] = {6, 3}

Key [3] = ∞ key [6] = ∞

w (4,3) = 22 w (4,6) = 24

w (u, v) < key [v] w (u, v) < key [v]

w (4,3) < key [3] w (4,6) < key [6]

Update key value of key [3] as 22 and key [6] as 24.

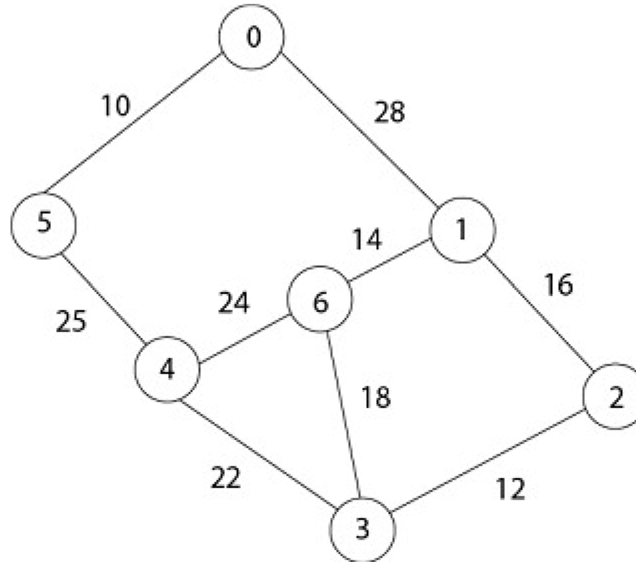
And the parent of 3, 6 as 4.

π[3]= 4 π[6]= 4

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	22	25	10	24
Parent	NIL	0	NIL	4	5	0	4

u = EXTRACT_MIN (3, 6) [key [3] < key [6]]

u = 3 i.e. 22 < 24



Adj [3] = {4, 6, 2}

4 is already in heap

4 ≠ Q key [6] = 24 now becomes key [6] = 18

Key [2] = ∞ key [6] = 24

w (3, 2) = 12 w (3, 6) = 18

w (3, 2) < key [2] w (3, 6) < key [6]

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

π [2] = 3 π[6]=3

Now by EXTRACT_MIN (Q) Removes 2, because key [2] = 12 is minimum.

Vertex	0	1	2	3	4	5	6
Key Value	0	28	12	22	25	10	18
Parent	NIL	0	3	4	5	0	3

u = EXTRACT_MIN (2, 6)

u = 2 [key [2] < key [6]]

12 < 18

Now the root is 2

Adj [2] = {3, 1}

3 is already in a heap

Taking 1, key [1] = 28

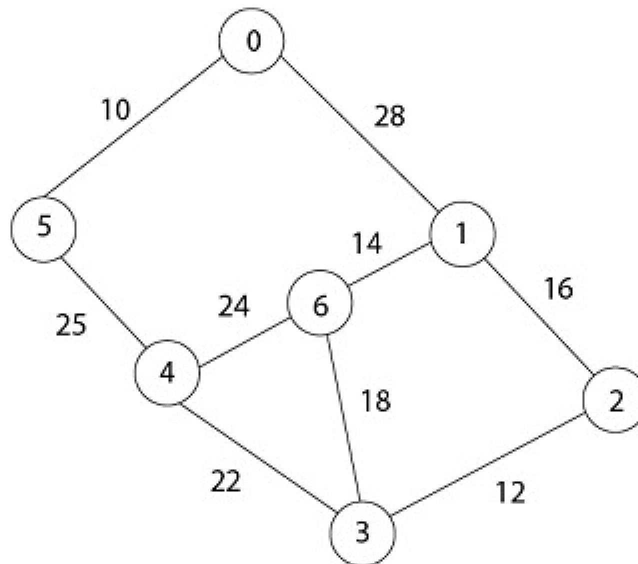
w (2,1) = 16

w (2,1) < key [1]

So update key value of key [1] as 16 and its parent as 2.

π[1]= 2

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	18
Parent	NIL	2	3	4	5	0	3



Now by EXTRACT_MIN (Q) Removes 1 because key [1] = 16 is minimum.

Adj [1] = {0, 6, 2}

0 and 2 are already in heap.

Taking 6, key [6] = 18

w [1, 6] = 14

w [1, 6] < key [6]

Update key value of 6 as 14 and its parent as 1.

Π [6] = 1

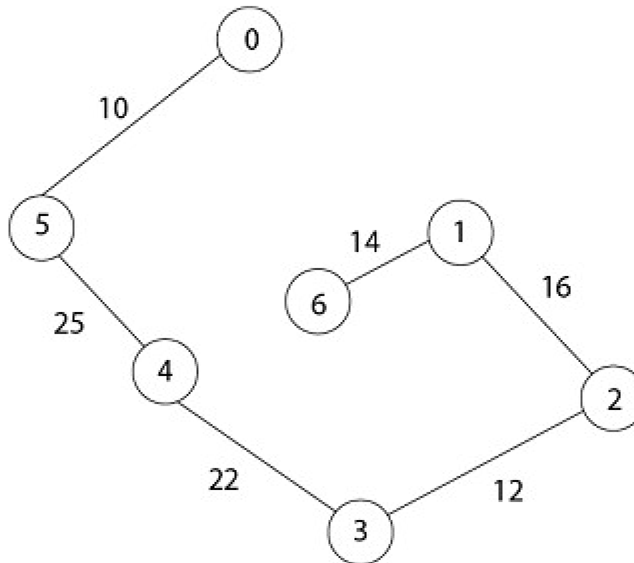
Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	14
Parent	NIL	2	3	4	5	0	1

Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

0 → 5 → 4 → 3 → 2 → 1 → 6

[Because Π [5] = 0, Π [4] = 5, Π [3] = 4, Π [2] = 3, Π [1] =2, Π [6] =1]

Thus the final spanning Tree is



Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99

Experiment No. 9

THEORY

Single sources shortest path algorithm

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex,s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

- **DIJKSTRA (G, w, s)**

1. INITIALIZE SINGLE-SOURCE (G, s)
2. S ← { } // S will ultimately contains vertices of final shortest-path weights from s
3. Initialize priority queue Q i.e., Q ← V[G]
4. while priority queue Q is not empty do
5. u ← EXTRACT_MIN(Q) // Pull out new vertex
6. S ← S È {u} // Perform relaxation for each vertex v adjacent to u for each
vertex v in Adj[u] do
7. Relax (u, v, w)

- ## ANALYSIS

Like Prim's algorithm, Dijkstra's algorithm runs in O(|E|lg|V|) time.

Step by Step operation of Dijkstra algorithm.

Step1. Given initial graph G=(V, E). All nodes nodes have infinite cost except the source node, s, which has 0 cost.

Step 2. First we choose the node, which is closest to the source node, s. We initialize d[s] to 0. Add it to S. Relax all nodes adjacent to source, s. Update predecessor (see red arrow in diagram below) for all nodes updated.

Step 3. Choose the closest node, x. Relax all nodes adjacent to node x. Update predecessors for nodes u, v and y (again notice red arrows in diagram below).

Step 4. Now, node y is the closest node, so add it to S. Relax node v and adjust its predecessor (red arrows remember!).

Step 5. Now we have node u that is closest. Choose this node and adjust its neighbor node v.

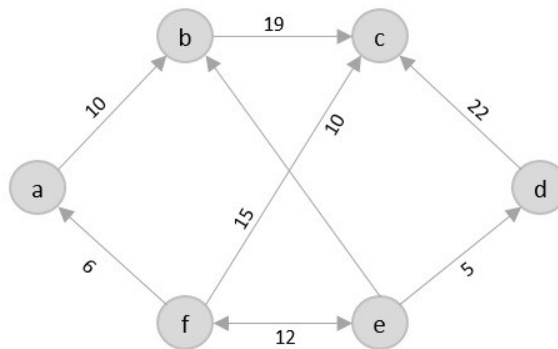
Step 6. Finally, add node v. The predecessor list now defines the shortest path from each node to the source node, s.

Experiment No. 10

THEORY

Traveling salesman problem

The travelling salesman problem is a graph computational problem where the salesman needs to visit all cities (represented using nodes in a graph) in a list just once and the distances (represented using edges in the graph) between all these cities are known. The solution that is needed to be found for this problem is the shortest possible route in which the salesman visits all the cities and returns to the origin city.



There are various approaches to find the solution to the travelling salesman problem: naïve approach, greedy approach, dynamic programming approach, etc. In this tutorial we will be learning about solving travelling salesman problem using greedy approach

Travelling Salesperson Algorithm

As the definition for greedy approach states, we need to find the best optimal solution locally to figure out the global optimal solution. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges. The shortest path of graph G starting from one vertex returning to the same vertex is obtained as the output.

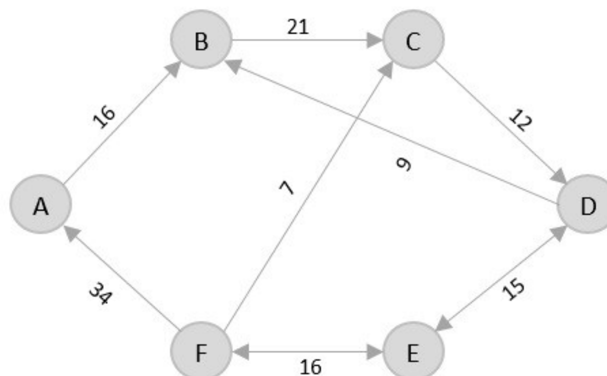
Algorithm

- Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G') which will record the path the salesman is going to take from one node to another.

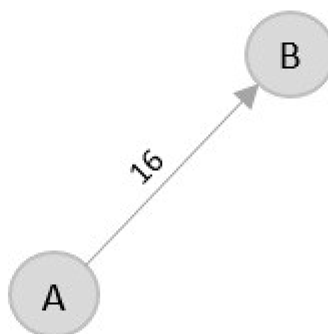
- The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
- The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
- Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
- Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.
- However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

Examples

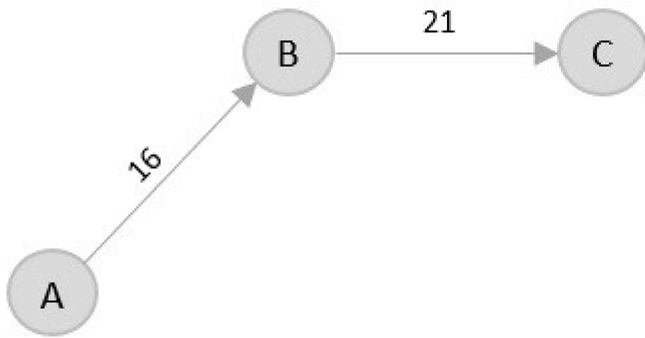
Consider the following graph with six cities and the distances between them −



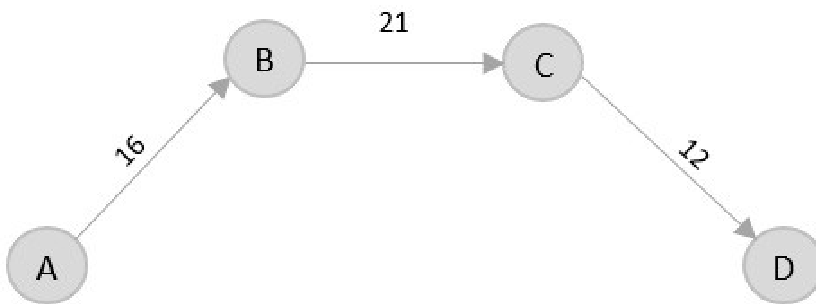
From the given graph, since the origin is already mentioned, the solution must always start from that node. Among the edges leading from A, A → B has the shortest distance.



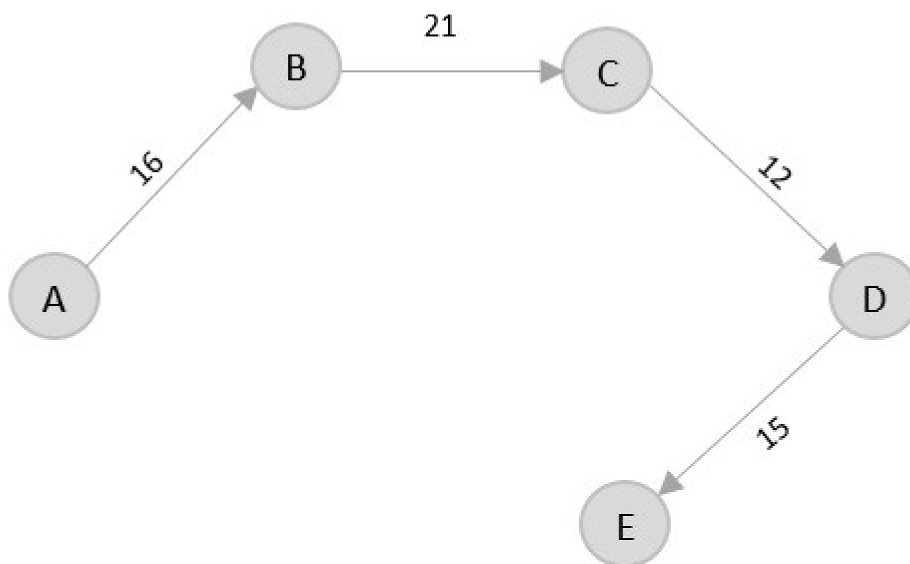
Then, B → C has the shortest and only edge between, therefore it is included in the output graph.



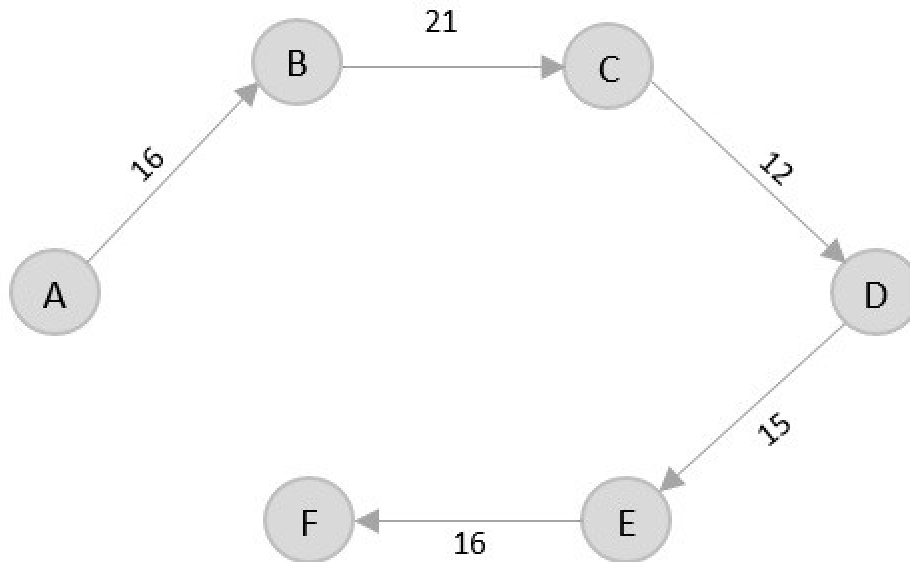
There's only one edge between C → D, therefore it is added to the output graph.



There's two outward edges from D. Even though, D → B has lower distance than D → E, B is already visited once and it would form a cycle if added to the output graph. Therefore, D → E is added into the output graph.

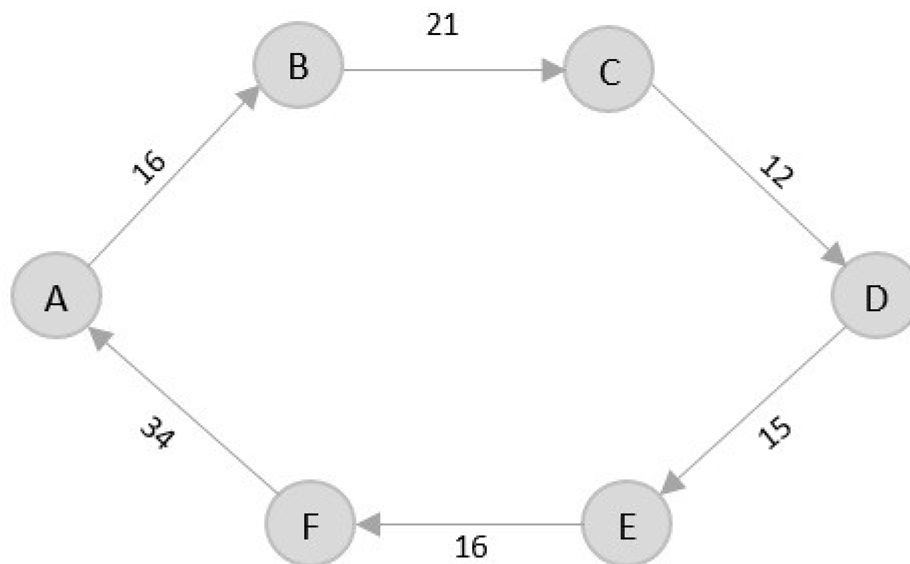


There's only one edge from e, that is E → F. ThE



Therefore, it is added into the output graph

Again, even though F → C has lower distance than F → A, F → A is added into the output graph in order to avoid the cycle that would form and C is already visited once.



The shortest path that originates and ends at A is A → B → C → D → E → F → A

The cost of the path is: 16 + 21 + 12 + 15 + 16 + 34 = 114.

Even though, the cost of path could be decreased if it originates from other nodes but the question is not raised with respect to that.