SPACE & TIME TRADEOFFS

Consider the problem of computing values of a function at many points in its domain. If it is time that is premium, we can precompute the function's values & store them in a table. So, the basic idea here is to pre-process the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. We call this approach INPUT ENHANCEMENT and discuss the following algorithms based on it:

(1) Sorting by counting
(2) Input enhancement in String Matching

DYNAMIC PROGRAMMING

There is one more algorithm design technique related to the space-for-time trade off idea: dynamic programming. This strategy is based on recording solutions to overlapping subproblems of a given problem in a table from which a solution to the problem in questions is then obtained.

(1) Warshall's
(2) Floyd's
(3) Knapsack Problem & Memory functions.

(4) N

GREEDY TECHNIQUE

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step – and this is the central point of this technique – the choice made must be

* feasible, i.e. it has to satisfy the problem's constraints.
* locally optimal. i.e. it has to be the best local choice among all feasible choices available on that step.
* irrevocable, i.e. once made, it cannot be changed on subsequent steps of the algorithm.

(1) Prim's
(2) Kruskal's
(3) Dijkstra's

DIJKSTRA'S ALGORITHM

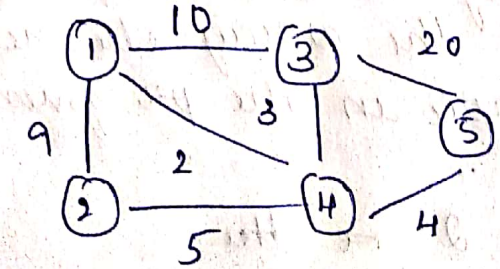
// Here, We consider the Single Source shortest paths problems which asks for a family of paths, each leading from the source to a different vertex in the graph. //

/ In other words for a given vertex called the source in a weighted connected graph, find the shortest path to all its other vertices /

(start from here) → Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distances from a given source. First it finds the shortest path from the source to the vertex nearest to it, then to a second nearest and so on.
In general, before its i^{th} iteration commences, the algorithm has already identified the shortest path to the $i-1$ other vertices nearest to the source

MATHEMATICAL SOLUTION

consider the following graph:



Lost

The adjacency matrix of the graph is

	1	2	3	4	5
1	0	9	10	20	∞
2	9	0	∞	5	∞
3	10	∞	0	3	20
4	2	5	3	0	4
5	∞	∞	20	4	0

Consider an array d which stores the lenght of the shortest path from the source to the vertex

Now from the considered graph find the shortest path to all the vertices considering the source as 1 and fill it up in the 1st column

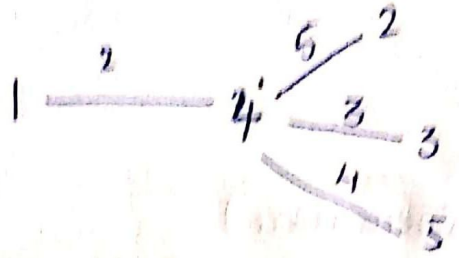
d[1]	0	0	0	0
d[2]	9	7	7	7
d[3]	10	5	5	5
d[4]	2	2	2	2
d[5]	∞	6	6	6

Source

Now from the array d find the shortest path from the and from that path try finding the shortest path to all vertices if present update the array else retain the previous value. for example in the pre given graph the shortest path is 2

so the path from 1 to 4 is 2 ~~is~~ ~~the~~

Step 1: Now find the path from 4 to other vertices



Now find the paths to all the vertices and compare them with the values in d array and update them.

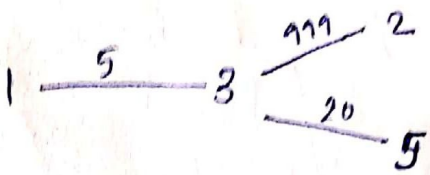
1 → 4 = 2

1 → 3 = 2+3 = 5

1 → 2 = 2+5 = 7

1 → 5 = 2+4 = 6

Step 2: Now find the shortest path in the array and do the same thing leaving 1 to 4.
Now the shortest path is from 1 → 4 → 3.

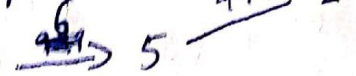


1 → 3 = 5

1 → 2 = 999+5 > 7 So retain the old value i.e 7

1 → 5 = 20+5 > 6 So retain the old value i.e 6.

Step 3: Consi perform the same step with the next shortest path



1 → 2 : 999 + 6 > 7 So retain the old value : i e 7

ALGORITHM :

```
Algorithm dijkstras (n, a, source)
// n is no. of nodes
// a is cost adjacency matrix
// source is the starting node (source node)
Begin
   for i ← to n do
Begin
      d[i] ← a[source][i]
       s[i] ← 0
End
   s[source] ← 1
   for i ← 1 to n do
   Begin
     min ← 999
     for i ← 1 to n do
       If (s[j] = = 0)
         If (d[j] < min)
          Begin
             min ← d[j]
               u ← j
            End
          s[u] ← 1
     for v ← 1 to m do
     If (s[v] = 0 & & d[v] > d[u] + a[u][v])
        d[v] = d[u] + a[u][v]
   End
    // Print distance from source /
   End
```

Module - 4

Greedy Technique

A problem is solved through sequence of subproblems, each sub-problem is solved by Greedy Technique.

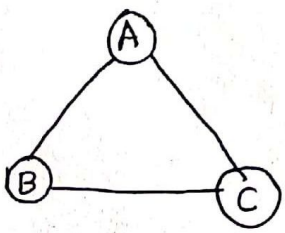
Definition:

Constructing a solution through sequence of steps, expanding partially constructed solution so far, until a complete solution for the problem is reached.

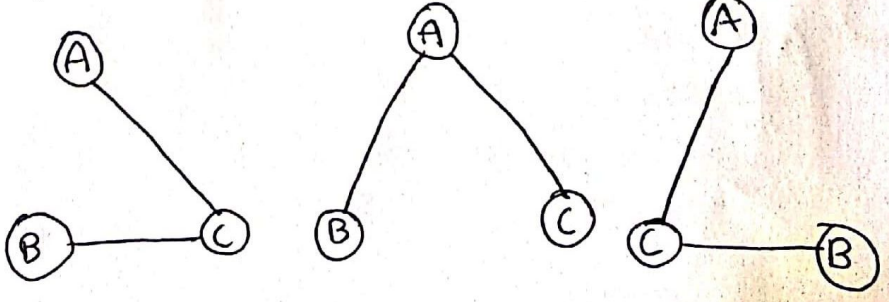
This technique is a certain paradigm that follows the problem solving of making the locally optimal choice at each stage with the intent of finding a optimal solution.

Spanning tree: Spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

Hence we can conclude that every connected and undirected Graph G has atleast one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all vertices.



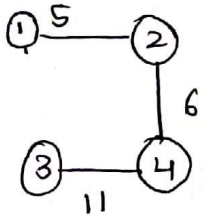
Graph G1



These all are Spanning trees

General properties of Spanning tree:

- A connected Graph G can have more than one Spanning tree.
- All possible Spanning trees of graph G, have the same number of edges and vertices.
- This does not have any loops.
- It has $(n-1)$ edges where n is the number of vertices.
- A complete graph can have maximum n^{n-2} number of Spanning trees.
- Spanning tree can be generated from weighted graph.



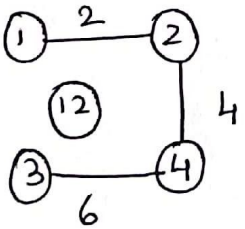
← weighted graph with cost for each edge.

Minimum Spanning Tree (MST):

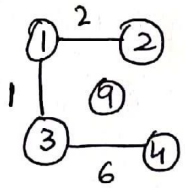
In a weighted graph, a minimum Spanning tree is a Spanning tree that has a minimum weight than all other Spanning trees of the same graph.

Ex:

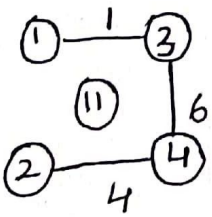
1.



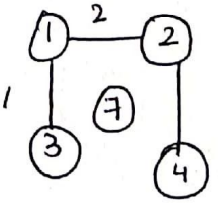
2.



3.



4.



← Optimal solution minimum Spanning tree (MST)

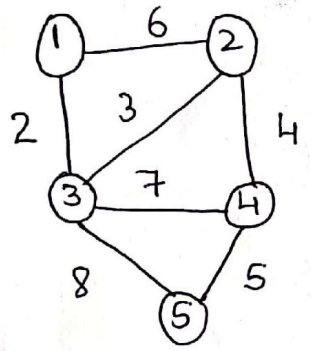
One of the spanning tree based algorithm is Prim's algorithm.

Prim's Algorithm: It is a greedy method used to get a minimum cost spanning tree which builds this tree edge by edge.

There are (n-1) steps with each forming an edge using greedy criteria a "choose an edge that results in a minimum increase in the sum of cost of the edges included so far."

Example : Take the following graph and construct the minimum spanning tree

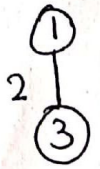
Take the source node as (1)



Step 1:

Visited (V)	Not visited (NV)
{1}	{2, 3, 4, 5}
= min	{(1,2), (1,3), (1,4), (1,5)}
= min	{6, 2, ∞, ∞} (2: cost, minimum)

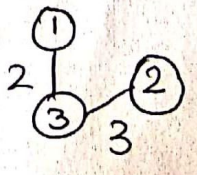
Hence 2 (1 → 3)



Step 2:

Visited	Not-visited
{1, 3}	{2, 4, 5}
= min	{(1,2), (1,4), (1,5), (3,2), (3,4), (3,5)}
= min	{6, ∞, ∞, 3, 7, 8} (3: minimal cost)

Hence 3rd edge is (3) ② (3 → 2)

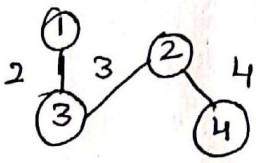


Step 3:

Visited	Not visited
{1,2,3}	{4,5}
= min	{(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)}
= min	{∞, ∞, 4, ∞, 7, 8}

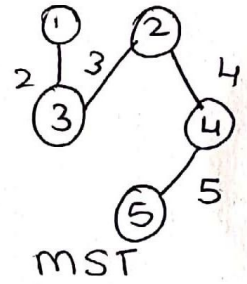
Hence 4th edge is (4)

= 4 (2 → 4)



Step 4:

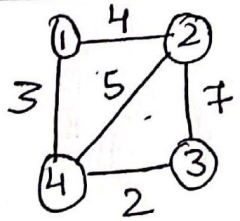
Visited	Not visited
{1,2,3,4}	{5}
= min	{(1,5),(2,5),(3,5),(4,5)}
= min	{∞, ∞ 8, 5}
= 5 (4 → 5)	



MST

14 — minimal cost

Example (2):



Step1: Source = 02 (2)

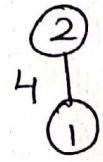
Visited	Not-visited
{2}	{1,3,4}
= min	{(2,1),(2,3)(2,4)}
= min =	{4, 7, 5}

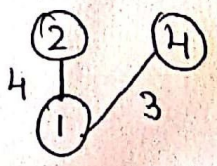
4 (2 → 1)

Step 2:

Visited	Not-visited
{2,1}	{3,4}
= min	{(2,3),(2,4)(1,3),(1,4)}
= min	{7, 5, ∞, 3}

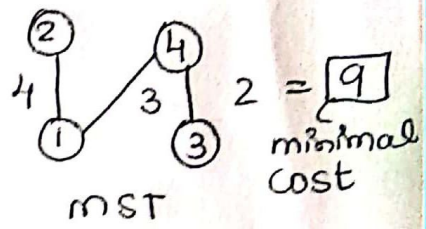
3 (1 → 4)





Step 3:

Visited	Not-visited
{1, 2, 4}	{3}
= min	{(1,3), (2,3), (4,3)}
= min	{∞, 7, 2}
2 (4 → 3)	



Algorithm: Prims(n, a, source)

```
// n is no of nodes
// s is the source ; // a is cost adjacency matrix
begin
    for i ← 1 to n do
    begin
        d[i] ← a[source][i]
        s[i] ← 0
    endfor
    s[source] ← 1
    for(i ← 1 to n-1 do
    begin
        minm ← 999
        for j ← 1 to n do
            if(s[j] = 0)
                if(d[j] < minm)
                begin
                    u ← j
                    minm ← d[j]
                end
            s[u] ← 1
        Sum ← Sum + d[u]
    for V ← 1 to n do
        if(s[v] = 0 && d[v] > a[u][v])
            then d[v] = a[u][v]

    end for
end for
return sum
end
```

KRUSHKAL'S ALGORITHM

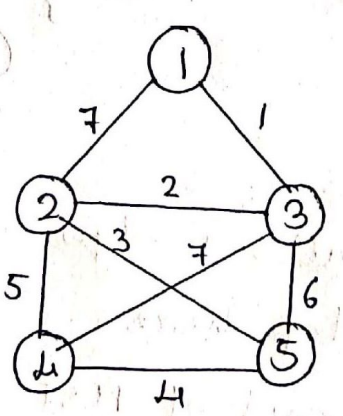
Krushkal's algorithm is a greedy method of finding the cost of minimum spanning tree.

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

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

Let's understand it with the below examples.

Example 1:



1) Remove all 0's & 999's and list the edges.

(1,2) = 7	(2,4) = 5	(3,4) = 7	(4,5) = 4
(1,3) = 1	(2,5) = 3	(3,5) = 6	(5,2) = 3
(2,1) = 7	(3,1) = 1	(4,2) = 5	(5,3) = 6
(2,3) = 2	(3,2) = 2	(4,3) = 7	(5,4) = 4

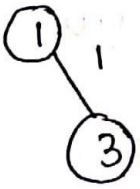
2) Based on cost arrange edges in ascending order

(1,3) = 1	(2,5) = 3	(2,4) = 5	(1,2) = 7
(3,1) = 1	(5,2) = 3	(4,2) = 5	(2,1) = 7
(2,3) = 2	(4,5) = 4	(3,5) = 6	(3,4) = 7
(3,2) = 2	(5,4) = 4	(5,3) = 6	(4,3) = 7

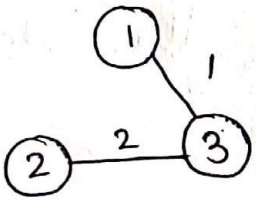
3) Construct MST by taking the edges from 2nd step.

* if any edge forms cycle reject it.

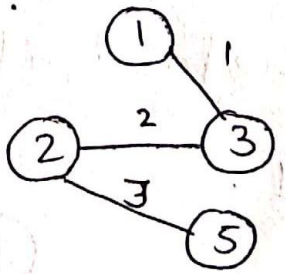
(i) 1-3



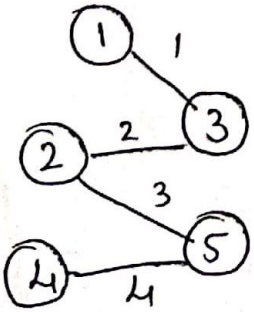
(ii) 3-2



(iii) 2-5



(iv) 4-5

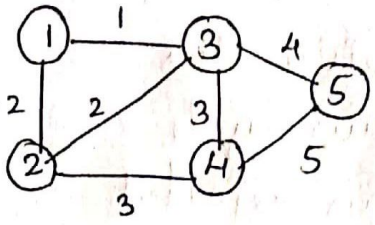


Since all n-1 edges are formed, we stop here.

Thus cost of MST is

1 + 2 + 3 + 4 = 10

Example 2 :



1) Remove all 0's and 999's and list the edges.

(1,2) = 2	(2,4) = 3	(3,5) = 4	(5,3) = 4
(1,3) = 1	(3,1) = 1	(4,2) = 3	(5,4) = 5
(2,1) = 2	(3,2) = 2	(4,3) = 3	
(2,3) = 2	(3,4) = 3	(4,5) = 5	

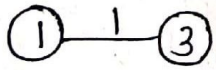
2) Based on cost arrange edges in ascending order.

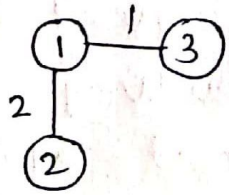
(1,3) = 1	(2,4) = 3	(4,5) = 5
(3,1) = 1	(4,2) = 3	(5,4) = 5
(1,2) = 2	(3,4) = 3	
(2,1) = 2	(4,3) = 3	
(2,3) = 2	(3,5) = 4	
(3,2) = 2	(5,3) = 4	

3) Construct MST by taking the edges from 2nd step.

* if any edge forms cycle reject it.

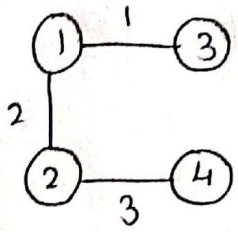
(i) 1, 3 (ii) 1, 2 (iii) 2, 3





Since it forms cycle reject that.

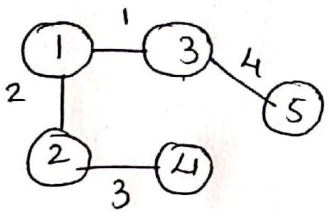
(iv) 2, 4



(v) 3, 4

Since this forms cycle it is rejected.

(vi) 3, 5



Thus cost of MST is 10

Algorithm: struct edge { int begv, endv, cost; };

```
Algorithm Kruskal (a, n)
// n is the no of nodes and a is the cost adjacency matrix.
// o/p: MST
begin
  struct edge ed[100], temp
  count = 1
  For i ← 1 to n do
  [begin
     parent[i] ← -1
  end
  For i ← 1 to n do
  begin
     for i ← 1 to n do
     begin
       if (a[i][j] ≠ 0 AND a[i][j] ≠ 999)
  [then    ed[count].begv = i
           ed[count].endv = j
           ed[count].cost = a[i][j]
           count ++
```

```
    end for
end for
    count = count - 1
    for i ← 1 to count do
    begin
        for i ← 1 to count - i do
        begin
            if (ed[j+1].cost < ed[j].cost)
            then
                temp = ed[j+1]
                ed[j+1] = ed[j]
                ed[j] = temp.
        end for
end for
    for i ← 1 to count do
    begin
        pbegv = getparent(parent, ed[i].begv)
        pendv = getparent(parent, ed[i].endv)
        if (pbegv != pendv)
        then
            sum = sum + ed[i].cost
            noe = noe + 1
            if (noe == n - 1)
            then
                Print "cost of MST"
                return n
        parent[pendv] = pbegv
```

```
end For
end Fuction Kruskal
  get_parent (parent , u)
begin
    while ( parent[v] ≠ -1 ) do
    v = parent[v]
    return v
end
```