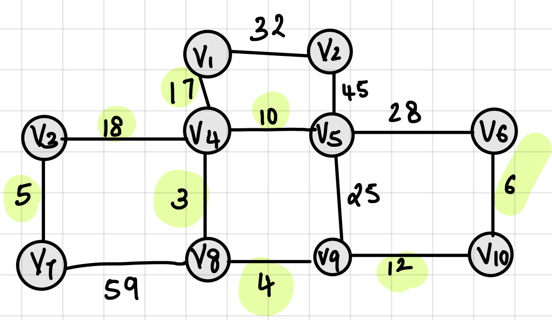
Q2) PRIMS ALGORITHM



- Prim's Algorithm

Let mstSet = {} be a set of all vertices

Let us start with V_1

VERTEX	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}
KEY	0	∞	∞	∞	∞	∞	∞	∞	∞	∞

We pick a vertex v which is not in mstSet & has minimum key value
consider $v = V_1$

mst Set = $\{V_1\}$

Update key values of vertices adjacent to V_1 :- V_2 & V_4

VERTEX	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}
KEY	0	32	∞	17	∞	∞	∞	∞	∞	∞

We must choose another vertex now : V_4
Adjacent to V_4, we must consider - V_3, V_5, V_8

mstSet = $\{V_1, V_4\}$

VERTEX	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}
KEY	0	32	18	17	10	∞	∞	3	∞	∞

Consider vertex V_8
Adjacent to V_8 , we must consider - V_7 & V_9

mst Set = $\{V_1, V_4, V_8\}$

VERTEX	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}
KEY	0	32	18	17	10	∞	59	3	4	∞

Consider V9
Adjacent to V9 ; V10 , V5

mstSet = { V1, V4, V8, V9 }

VERTEX	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
KEY	0	32	18	17	10	∞	59	3	4	12

We don't update V5, as the value while connecting via V9 is higher than the current value

Consider V5
Adjacent to V5 - V6, V9, V4

mstSet = { V1, V4, V8, V9, V5 }

VERTEX	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
KEY	0	32	18	17	10	28	59	3	4	12

Consider V10
Adjacent to V10 - V9, V6

mstSet = { V1, V4, V8, V9, V5, V10 }

VERTEX	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
KEY	0	32	18	17	10	6	59	3	4	12

Consider V6
Adjacent to V6 - V5 & V10

mstSet = { V1, V4, V8, V9, V5, V10, V6 }

VERTEX	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
KEY	0	32	18	17	10	6	59	3	4	12

Consider V3
Adjacent to V3 - V4 & V7

mstSet = { V1, V4, V8, V9, V5, V10, V6, V3 }

VERTEX	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
KEY	0	32	18	17	10	6	5	3	4	12

Consider V7
Adjacent to V7 - V3 & V8

mstSet = { V1, V4, V8, V9, V5, V10, V6, V3, V7 }

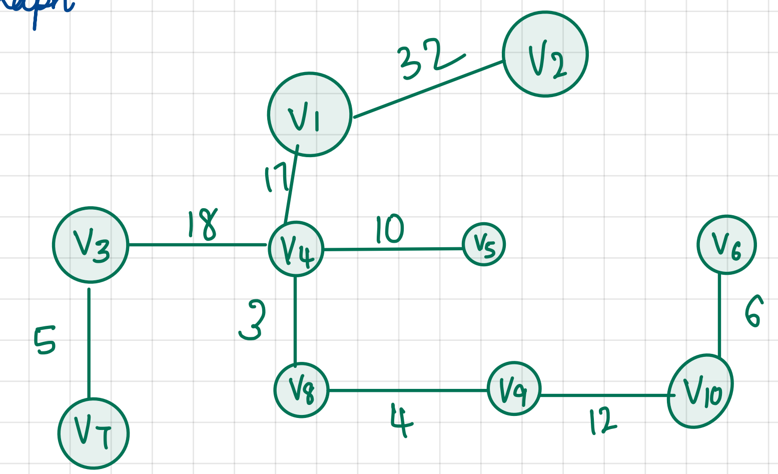
VERTEX	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
KEY	0	32	18	17	10	6	5	3	4	12

Consider V_2

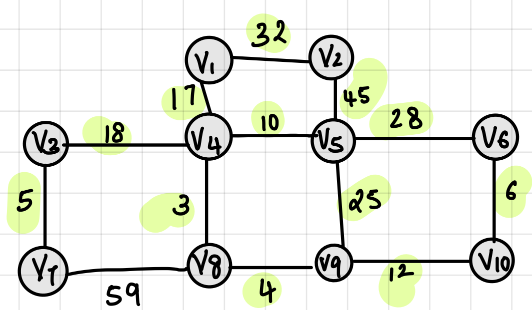
mstSet = $\{V_1, V_4, V_8, V_9, V_5, V_{10}, V_6, V_3, V_7, V_2\}$

VERTEX	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}
KEY	0	32	18	17	10	6	5	3	4	12

Final graph



Q7) KRUSKAL'S ALGORITHM

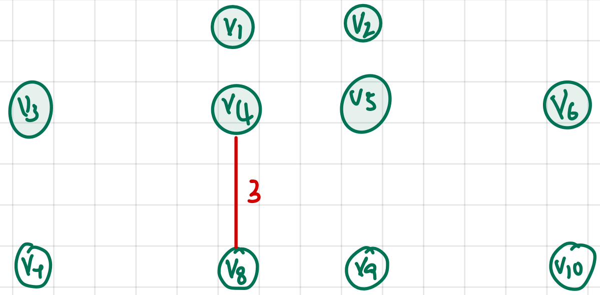


From the graph it may be observed that there are:
1. No parallel edges
2. No loops

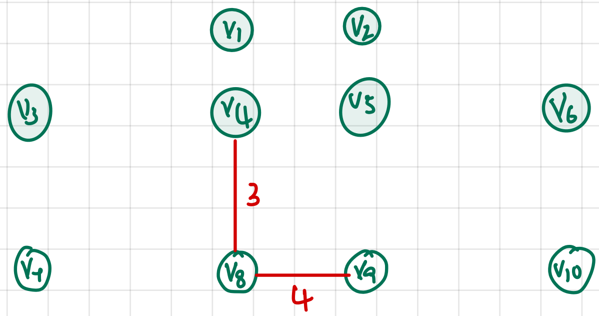
Let us initialise the edge table

Edge	V_4V_8	V_8V_9	V_3V_7	V_6V_{10}	V_4V_5	V_9V_{10}	V_4V_1	V_4V_3	V_5V_9	V_5V_6	V_1V_2	V_2V_5	V_7V_8
Weight	3	4	5	6	10	12	17	18	25	28	32	45	59

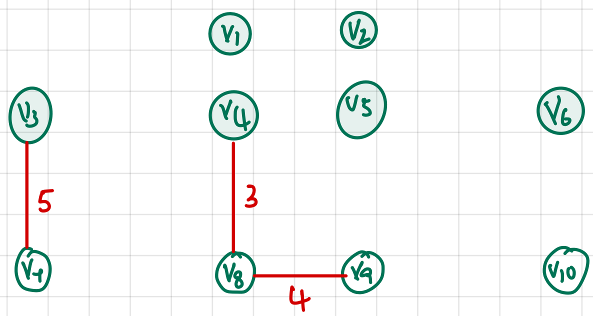
We start with the edge with least weight - V_4V_8



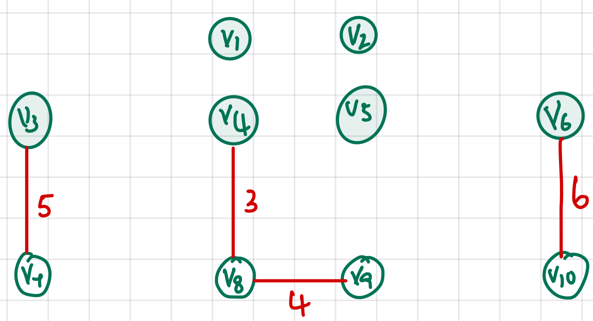
Next we choose $V_8 V_9$



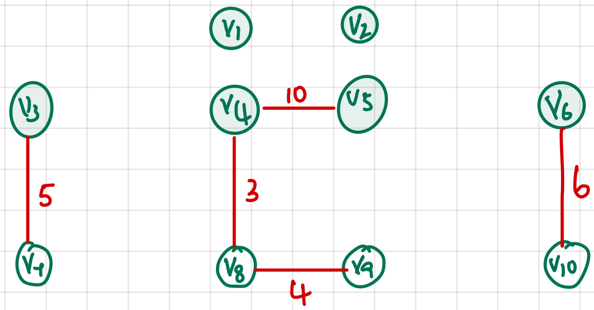
Next we choose $V_3 V_7$



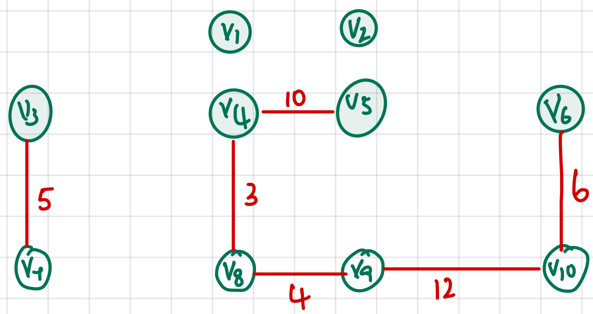
Next we choose $V_6 V_{10}$



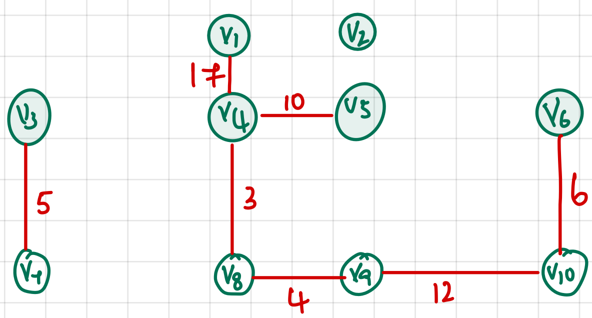
Next we choose V_4V_5



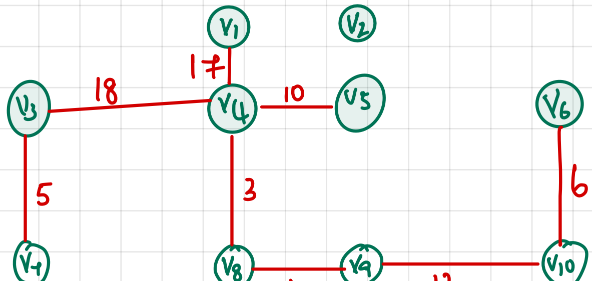
Next we choose V_9V_{10}



Next we choose V_1V_4



Next we choose V_4V_3



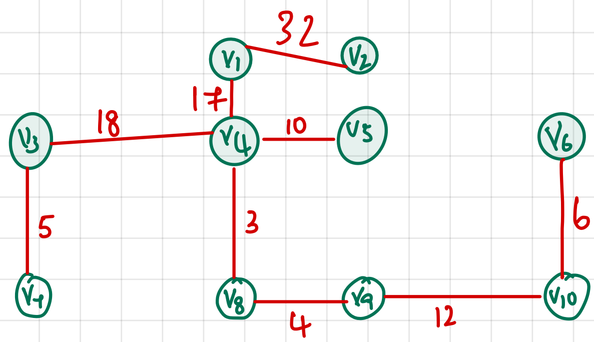
WE CANNOT CONSIDER EDGE V_5V_9 AS IT WOULD CREATE A CYCLE

$$\{V_4 - V_5 - V_9 - V_8 - V_4\}$$

WE CANNOT CONSIDER EDGE V_5V_6 AS IT WOULD CREATE A CYLCE

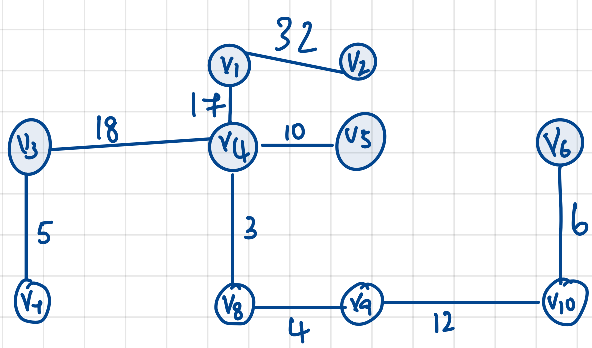
$$\{V_4 - V_5 - V_6 - V_{10} - V_9 - V_8 - V_4\}$$

Next we choose V_1V_2



As we have (n-1) vertices we can stop the algorithm

FINAL GRAPH



Q18) USE INDUCTION TO PROVE DJIKSTRA'S ALGORITHM

G - input graph
S - source vertex
$l(u,v)$ - length of an edge from v to v
V - set of vertices in the graph
$\delta(v)$ - shortest path from s to v
R - set of all nodes whose final shortest path weights from the source node should be determined
$d(x)$ - computed distance

WHAT ARE WE TRYING TO PROVE?
↳ For every vertex v → computed distance $d(v)$ = shortest distance $\delta(v)$

LEMMA : For every vertex x in R
$d(x) = \delta(x)$

Base Case : $|R| = 1$
$|R| = 1$ only when $R = \{s\}$ - i.e. the only node in the graph is the source node
When the only node being considered is the source node - the distance from s to s = 0

$$\left.\begin{array}{l} d(s) = 0 \\ \delta(s) = 0 \end{array}\right\} \therefore \text{The base case holds}$$

Inductive Hypothesis -
↳ We consider that the algorithm has computed a number of vertices that are stored in R
↳ We want to add a new vertex u to R
↳ By the inductive hypothesis, we can assume that for all vertices in R that are not u, the lemma holds true

$$\boxed{x \in R \Rightarrow d(x) = \delta(x) \text{ where } x \neq u}$$

To Prove That : $d(u) = \delta(u)$

CONTRADICTION -
Let there be a shorter path from s to u - called A

Length of A is lesser than computed distance

$$\boxed{l(A) < d(u)}$$

We know that the path A starts from the source vertex and traverses through a number of nodes and then reaches vertex u

A may be a path like this

$$A : s \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow u$$

In this example path $v_1, v_2, v_3 \in R$
This implies that they have already been processed by the algorithm and have correctly calculated distances.

Vertex u has not yet been processed and hence does not belong to set R

Thus A follows a path that at some point leaves set R to reach vertex u

Let the first edge that leaves set R be called $\boxed{cd}$

Here c is present in set R & d is not

We know that if a vertex belongs to set $R \rightarrow d(x) = \delta(x)$

$\therefore$ The distance from the source to vertex c is correctly stored as

$$\boxed{d(c) = \delta(c)}$$

Because d is adjacent to c [connected by edge cd] so d was processed when c was updated

That is

$$d(d) \leq d(c) + l(cd)$$

Djikstra's algorithm attempts to minimise the distance of vertices directly connected to the vertex being processed

The algorithm picked vertex u as it has the smallest known distance

$$\therefore \; d(u) \leq d(d)$$

① $d(c) \leq l(A_c)$: The inductive hypothesis - we know that $d(c) \leq l(A_c)$ where A_c is the subpath from s to c

② $d(d) \leq d(c) + l(cd)$

③ $d(u) \leq d(d)$

Combining in reverse: $d(u) \leq d(d) \leq d(c) + l(cd) \leq l(A)$

This proves that $d(u) \leq l(A)$ which goes against our initial contradiction

$\therefore$ The algorithm is correct

Q35) D.P. ALGORITHM FOR 0-1 KNAPSACK PROBLEM

ALGORITHM: Memoization

Input - values (list) - value of each item
weights (list) - weight of each item
capacity (int) - how much weight can the knapsack hold?
n (int) - number of items

```
Algorithm knapsack_Memoization (values[], weights[], capacity, n):
  create a 2D array memo[n+1][capacity + 1].
   for i ( 0 : n) do:
       memo[i][0] = 0
   for j ( 0 : w) do :
       memo [0][j] = 0

  def knapsack( capacity, n):
     if capacity = = 0 OR n = = 0:
         Return 0
     if memo[n][capacity] != 0:
         Return memo [n] [capacity]
     if weights[n-1] > capacity:
         result = knapsack (capacity , n-1)
     else:
         item = values[n-1] + knapsack (capacity - weights [n-1], n-1)
         e_item = knapsack( capacity, n-1)
         result = max( item, e_item)
     memo [n][ capacity = result
     return result
```

ALGORITHM : TABULATION

Input - values (list) - value of each item
weights (list) - weight of each item
capacity (int) - how much weight can the knapsack hold?
n (int) - number of items

```
Algorithm knapsackTabulation (values[ ], weights [ ], capacity, n):

    Initialise tab[n+1][capacity + 1] = 0

    for i( 1 : n) :
        for w( 1: capacity):
            if weights [i-1] <= w:
                item = values [i-1] + tab[i-1][w - weights[i-1]]
                e-item = tab [i-1][w]

                tab [i][w] = max (item, e-item)

            else :
                tab [i][w] = tab[i-1] [w]

        Return tab [n] [capacity]
```