Graph Algorithms – 3

CS 202 – Fundamental Structures of Computer Science II

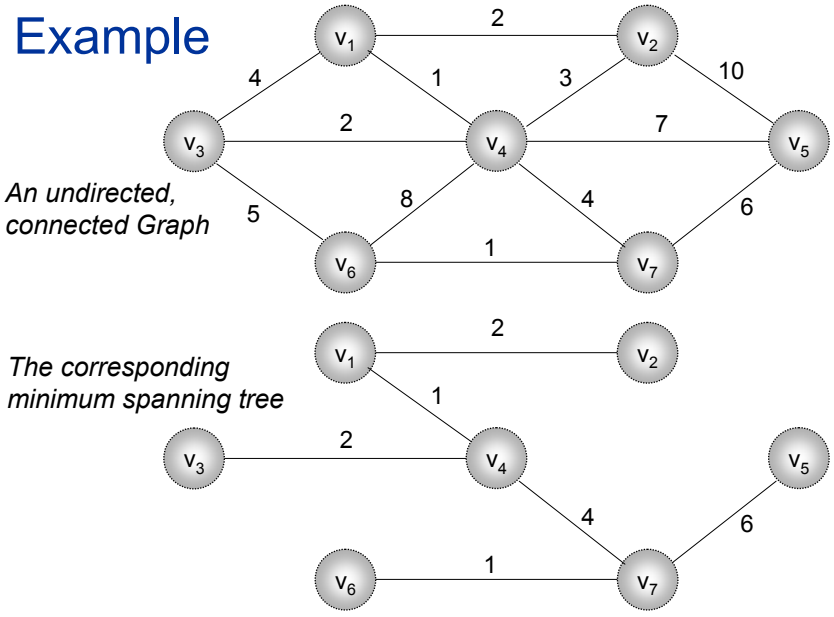
Bilkent University

Computer Engineering Department

Minimum Spanning Tree

- Problem: Finding a minimum spanning tree in an *undirected* and *connected* graph.
- What is minimum spanning tree?
  - A *tree*
    - that covers (spans) all the vertices of a connected graph
    - that has the minimum total cost of edges in the tree.
- A minimum spanning tree exists for a graph if and only if the graph is *connected*.
- The same problem makes sense for directed graphs also, we the solution is more difficult.

Example



An undirected, connected Graph

The corresponding minimum spanning tree

Minimum Spanning Tree (MST)

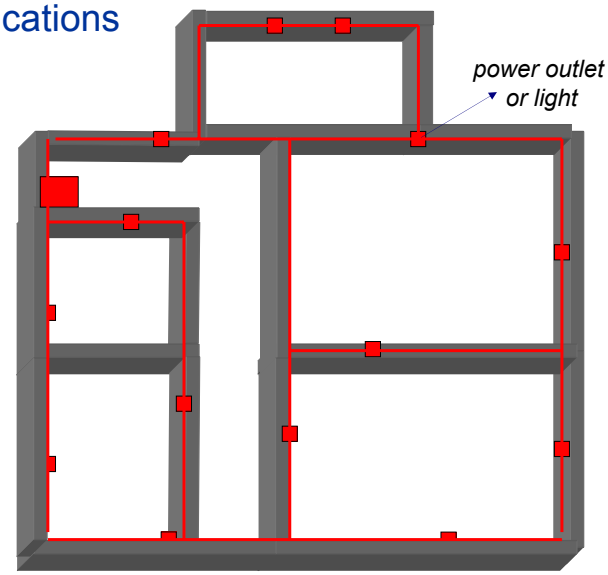
- If the number of vertices of a connected undirected graph is |V|, then its minimum spanning tree will have
  - |V| vertices
  - |V| - 1 edges.
- An MST does not contain any cycles, since it is a tree.
- If we add an extra edge to an MST, then it will have a cycle.

Minimum Spanning Tree (MST)

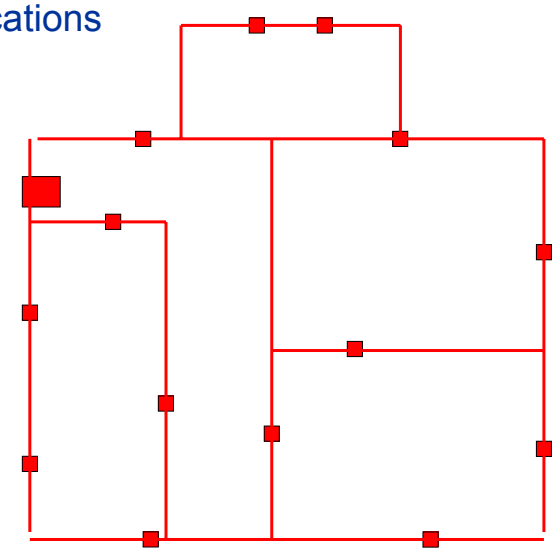
- Greedy approach for finding an MST for a graph works!
- Given a graph G, we start with an initial one vertex MST.
- At each stage we add one more vertex and one more edge (that connects this vertex to the previous MST), so that the edge has minimum possible value.

MST Applications

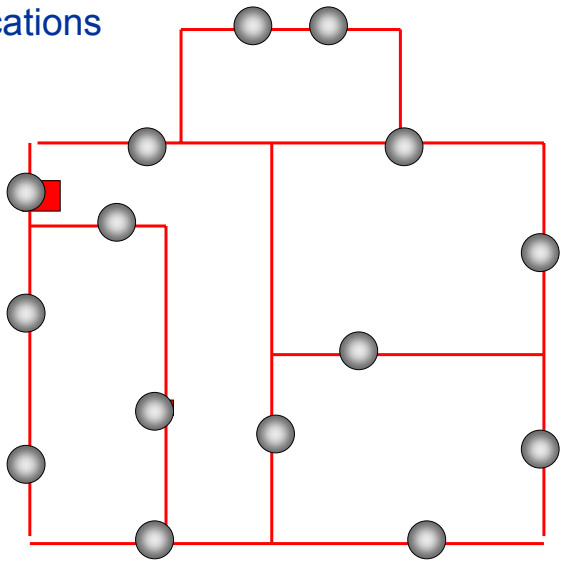
Electrical wiring of a house using minimum amount of wires (cables)



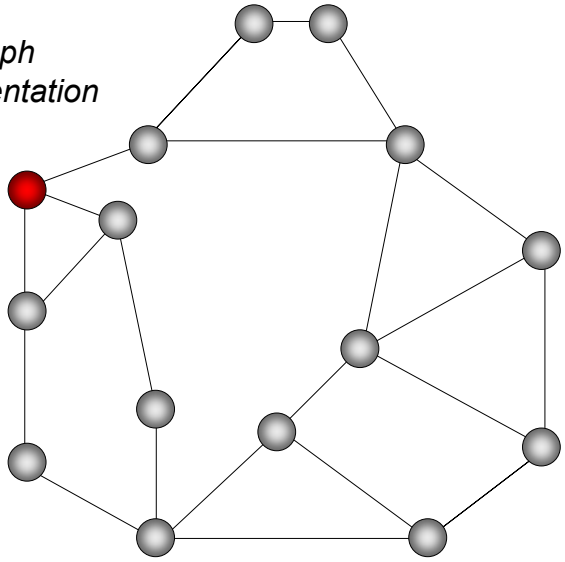
MST Applications



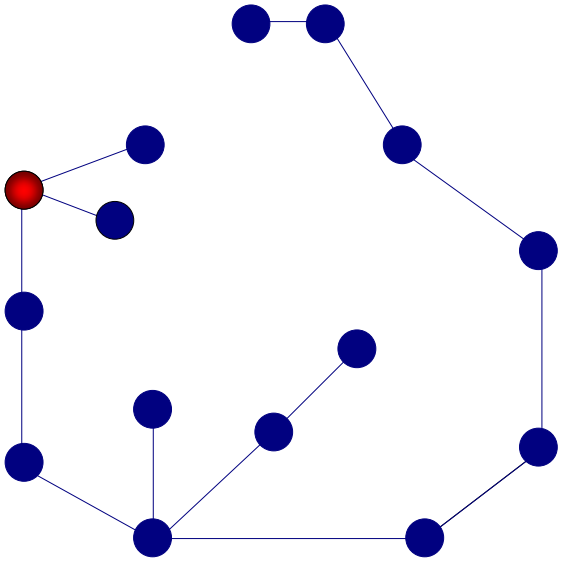
MST Applications



Graph Representation



Minimum Spanning Tree for electrical eiring



MST Algorithms

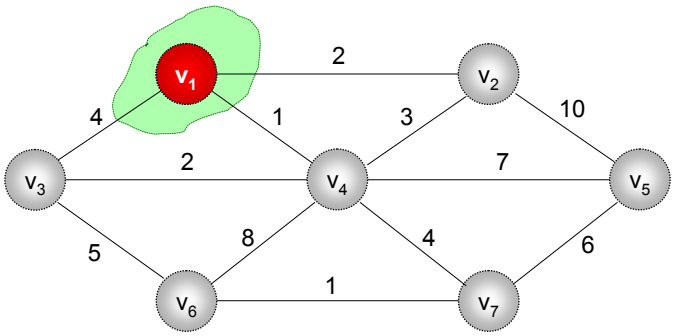
- We will see two algorithms
  - Prim's Algorithm
  - Kruskal's Algorithm

Prim's Algorithm

- MST is grown in successive stages.
- At each stage:
  - A new vertex is added to the tree by choosing the edge (u,v) such that the *cost* of (u,v) is the smallest among all edges where u is in the tree and v is not.

Prim's Algorithm

Start with vertex v_1. It is the initial current tree which we will grow to an MST

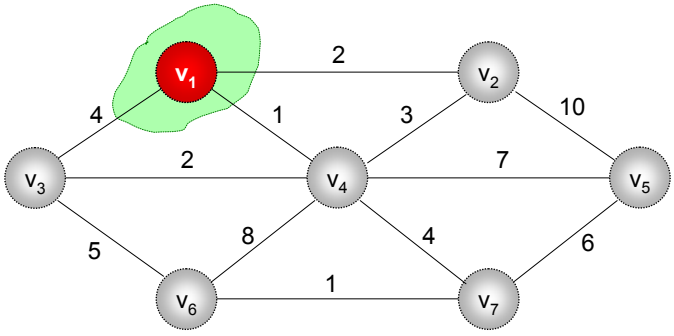


A connected, undirected graph G is given above.

Prim's Algorithm

Step 1

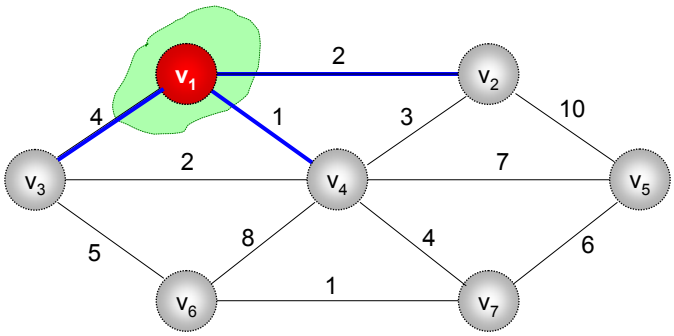
Select an edge from graph:
that is not in the current tree,
that has the minimum cost,
and that can be connected to the current tree.



Prim's Algorithm

Step 1

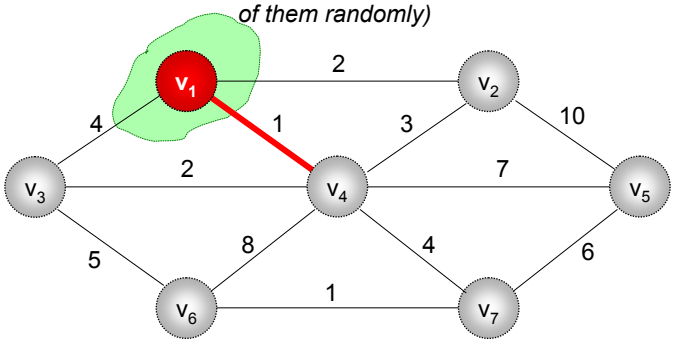
The edges that can be connected are:
(v_1,v_2): cost 2
(v_1,v_4): cost 1
(v_1,v_3): cost 2



Prim's Algorithm

Step 1

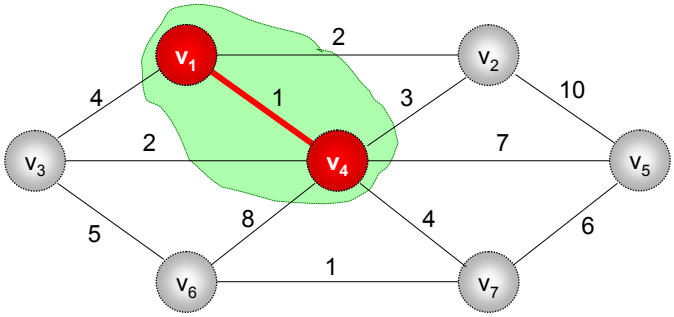
The edge that has the minimum cost is:
(v_1,v_4): cost 1
(there could be more than one. In that case we could choose of of them randomly)



Prim's Algorithm

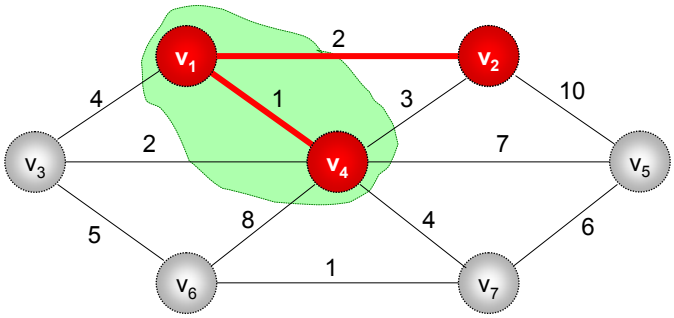
Step 2

We include the vertex v_4, that is connected to the selected edge, to the current tree. In this way we grow the tree.



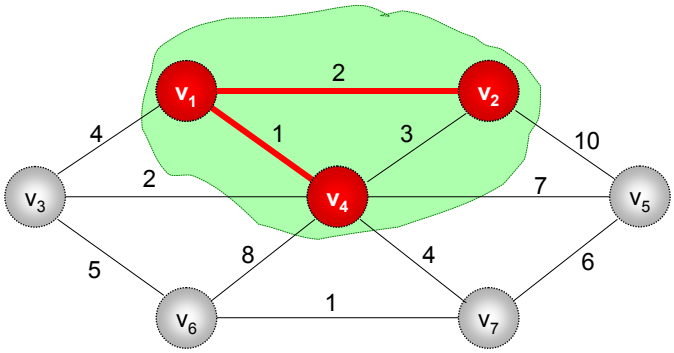
Prim's Algorithm

Repeat previous steps: 1, 2
You can add either edge (v_1, v_2) or (v_1, v_3). Do a random tie-break.
Lets add edge (v_1, v_2)



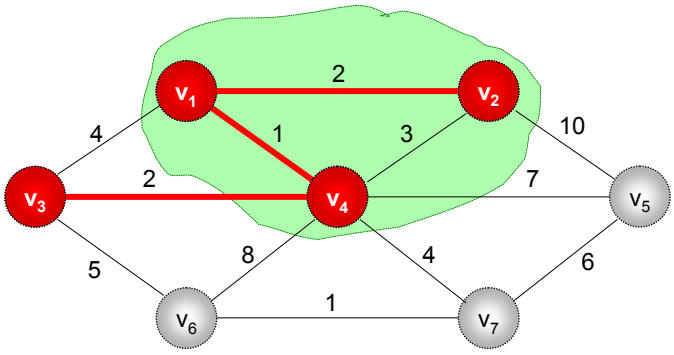
Prim's Algorithm

Current tree grows!



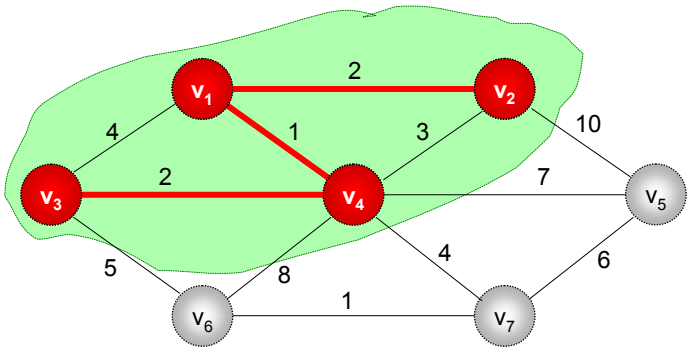
Prim's Algorithm

Repeat steps: 1, and 2
Add either edge (v_4, v_3)



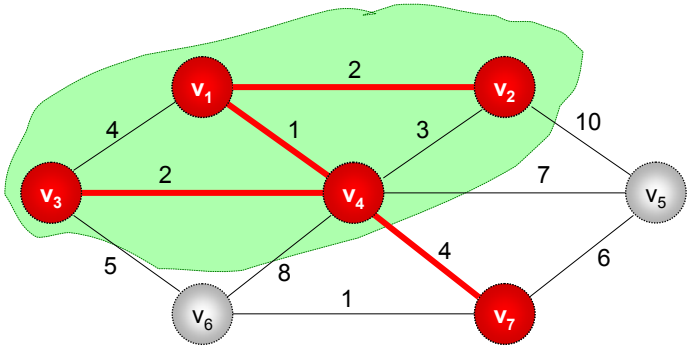
Prim's Algorithm

Grow the tree!



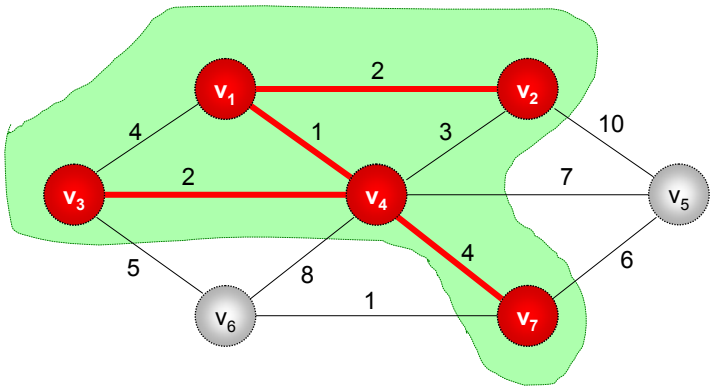
Prim's Algorithm

Add edge (v_4, v_7)



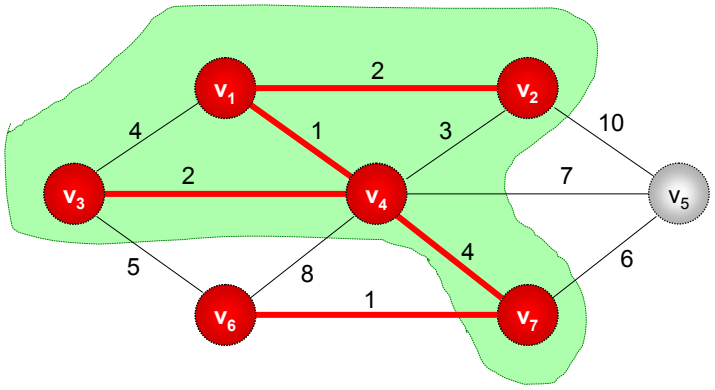
Prim's Algorithm

Grow the tree!



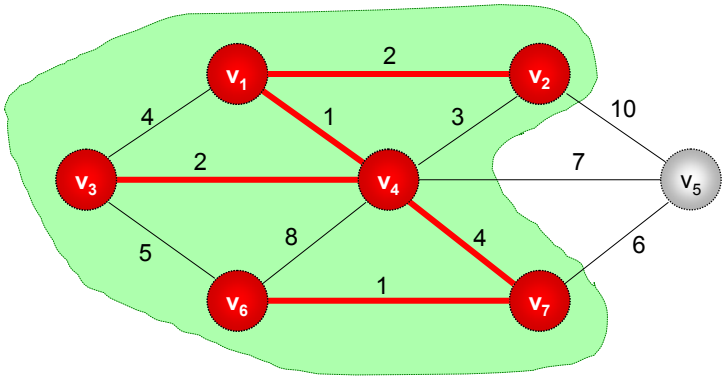
Prim's Algorithm

Add edge (v_7, v_6)



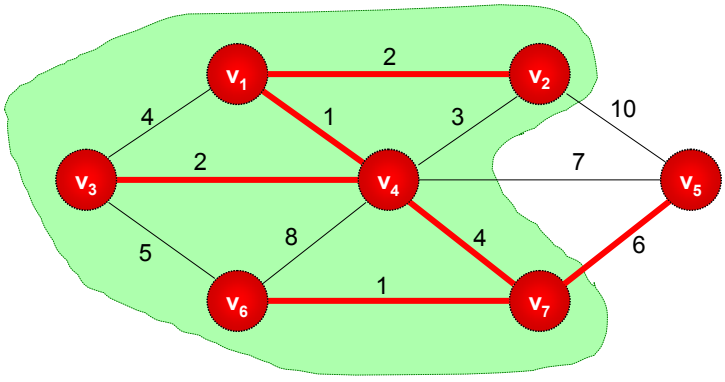
Prim's Algorithm

Grow the tree!



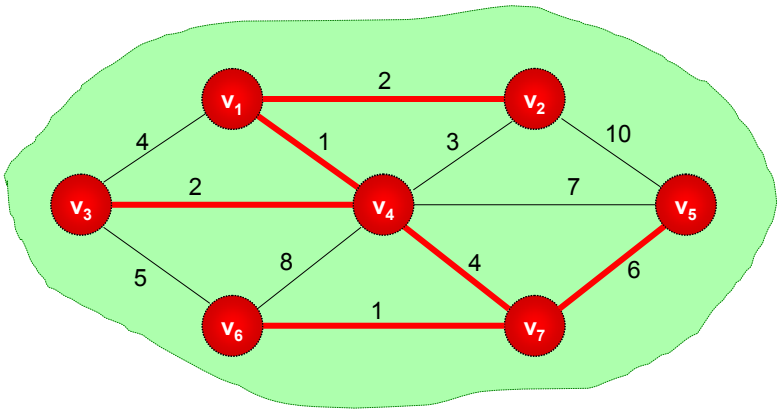
Prim's Algorithm

Add edge (v_7, v_5)



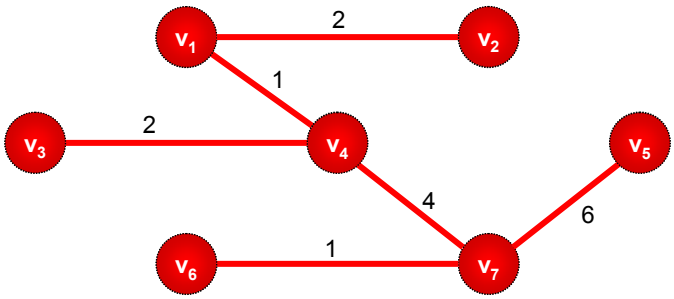
Prim's Algorithm

Grow the tree!



Prim's Algorithm

Finished!
The resulting MST is shown below!



Algorithm Implementation

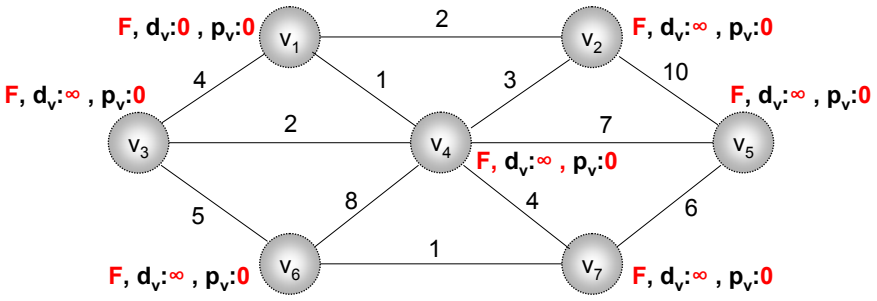
- Very similar to Dijkstra's shortest path algorithm.
  - Both are greedy type of algorithms
- For each vertex v, we keep the following information:
  - Known/unknown
    - Whether we have included the vertex in current tree or not.
  - Distance to previous node (d_v):
    - the cost of the edge that is connecting v to a *known* vertex that is part of current tree.
  - Previous vertex (p_v)
    - The last *known* vertex, that causes a change in the value of d_v

Initial configuration of table used in Prim's Algorithm implementation

Vertex	known	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

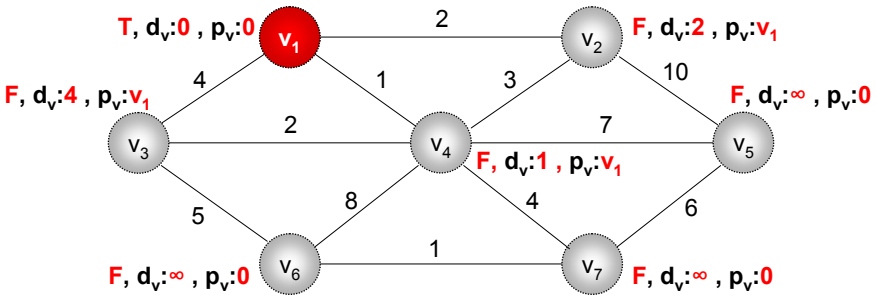
Initial Configuration

Vertex	known	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0



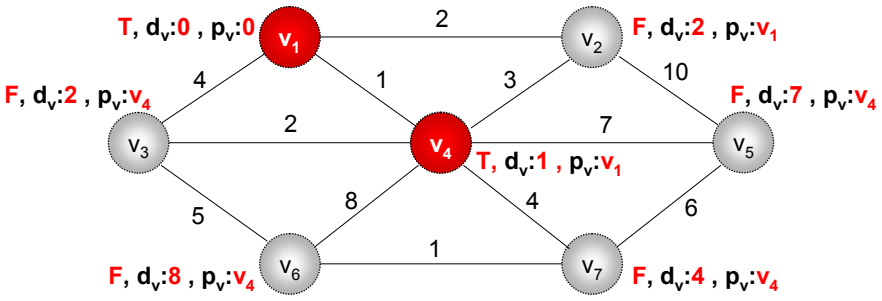
After v_1 is declared known

Vertex	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	4	v_1
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0



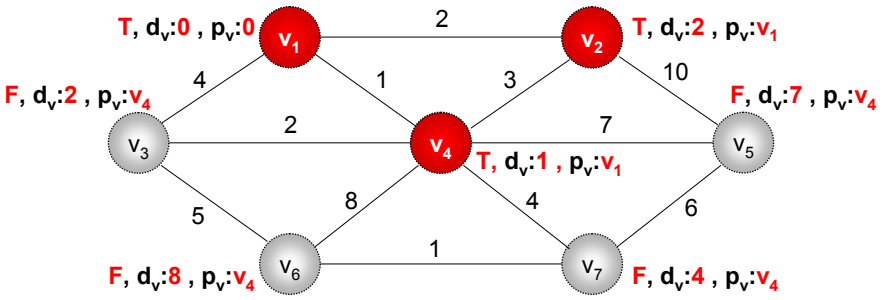
After v_4 is declared known

Vertex	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	8	v_4
v_7	F	4	v_4



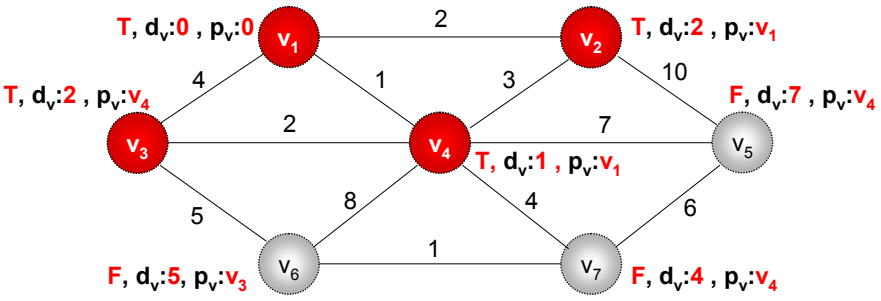
After v_2 is declared known

Vertex	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	8	v_4
v_7	F	4	v_4



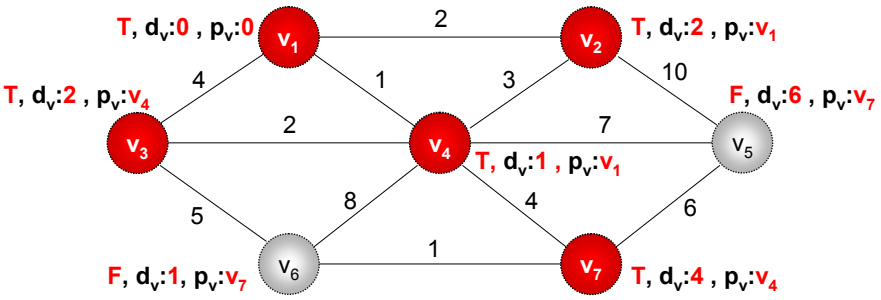
After v_3 is declared known

Vertex	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	5	v_3
v_7	F	4	v_4



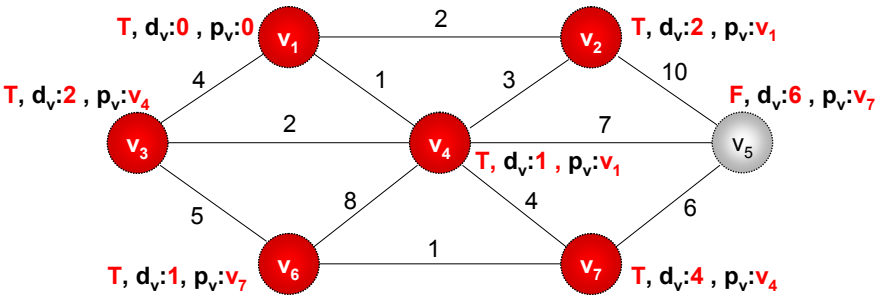
After v_7 is declared known

Vertex	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	6	v_7
v_6	F	1	v_7
v_7	T	4	v_4



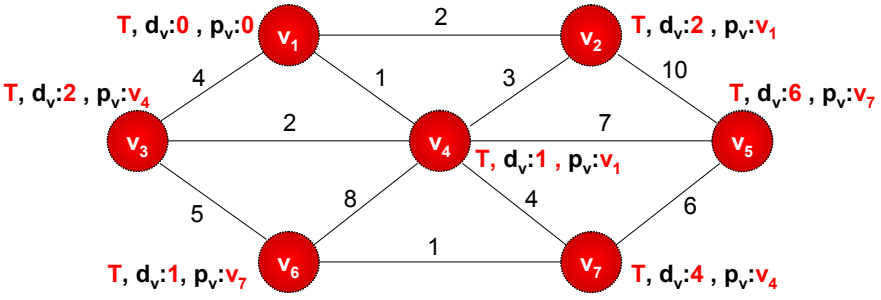
After v_6 is declared known

Vertex	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4



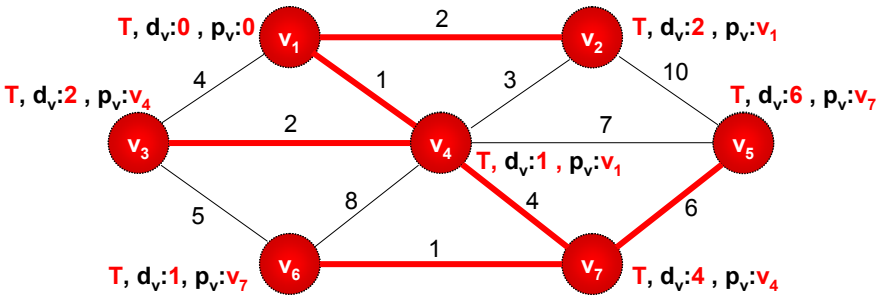
After v_5 is declared known

Vertex	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	T	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4



From the Table, read the edges of MST

Vertex	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	T	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4




```
void Graph::find_prim_mst( vector<Vertex> &s /* initial vertex */)
{
        Vertex v, w;
        s.dist = 0;
        s.known = T;
        for ( ; ; )
        {
                v = an unknown vertex whose distance value is minimum.
                if (v == NOT_A_VERTEX)
                        break; // we are finished
                v.known = TRUE;

                for each w adjacent to v
                {
                        if (w.known == FALSE) {
                                if (cost_v_w < w.dist) {
                                        w.dist  = cost_v_w;
                                        w.path = v;
                                }
                        }
                }
        }
}
```

***cost_v_w** is the cost of edge from vertex v to w.*

```
class Vertex
{
        boolean  known; // T or F
        int      dist;  // dv
        Vertex   path;  //pv
}
```

```
void Graph::print_prim_mst()
{
        Vertex v, w;

        for (each vertex v in G)
        {
                w = v.path;
                print edge (v,w);
}
```

The output is the set of edges in the spanning tree.

Running Time

- We execute the *outer* **for** *loop* at most |V| times (for each vertex).
  - In each iteration we try to find the unknown node that have the minimum distance: O(|V|)
- We execute the *inner* **for** *loop* O(|E|) times.
- Therefore the running time of the algorithm in the given form is:
  - **$O(|E| + |V|^2)$**

Running Time

- The given bound is good if the graph is dense:
  - In a dense graph $|E| = \Theta(|V|^2)$
  - In that case, the running time $O(|V|^2)$ which is $O(|E|)$
  - Therefore the algorithm is very efficient in this case.

Running Time

- The algorithm in the given form is not good if the graph is dense.
  - It is inefficient.
- The running time can be improved if we use priority queue (binary heap).

Running Time

- If we use priority queue of vertices
  - The vertex with minimum distance is kept at the root of the heap.
- The outer loop is executed O(|V|) times.
  - The search inside the outer for loop for a vertex that has the minimum distance takes O(log|V|) time.
- The inner loop is executed O(|E|) times.
  - The distances can be updates O(|E|) times.
  - The distance value of a vertex can be updated using *decreaseKey*() operation of binary heaps, which works in O(|V|) time. (|V| is the size of the binary heap).

Running Time

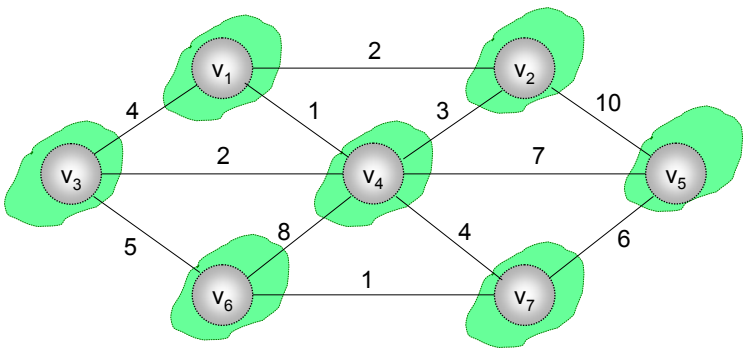
- Therefore the running time of MST algorithm using priority queues is:
  - O(|V| x log(|V|)+ |E| x log(|V|)) = O(|E| x log(|V|))
- If |E| is O(|V|) then
  - Running time is **O(|V| x log(|V|))**
  - This is for sparse graphs.
- Compare this with the running time of the original algorithm (the one that does not use priority queues) for sparse graphs, which is $O(|V|^2)$

Kruskal's Algorithm

- Select edges in the order of smallest weights and accept an edge if it does not cause a cycle.
- Kruskal's algoritm maintains a forest of trees.
  - Initially each vertex is a tree with single node
    - There are |V| trees.
  - Then, adding an accepted edge merges two trees in the forest
- When algorithm terminates, there is a single tree with |V| vertices and it is a minimum spanning tree.

.

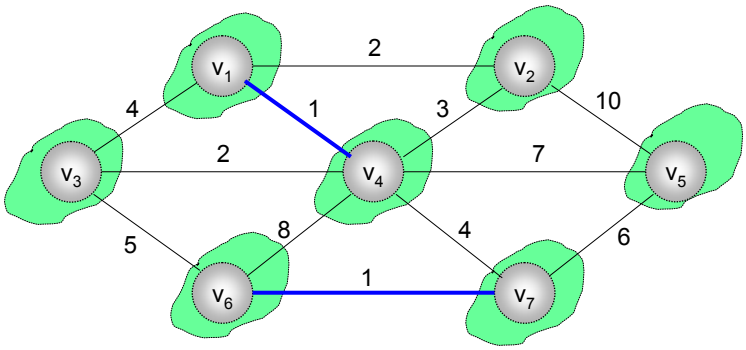
Initial Forest



Initial Forest

Step 1

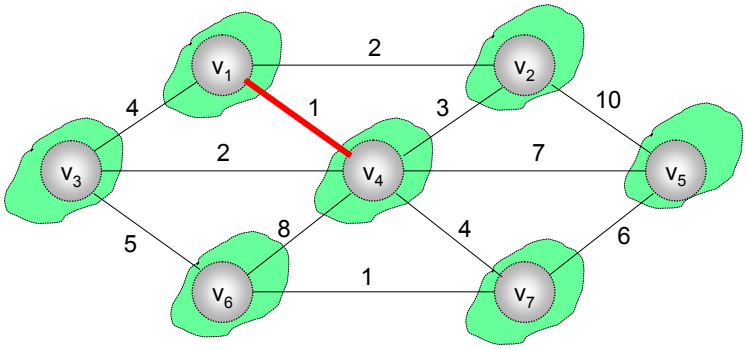
Candidate edges are shown (edges that have low cost and edges that connect two trees)



Initial Forest

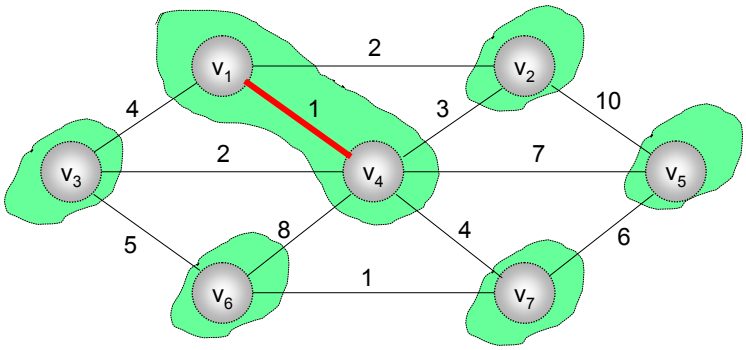
Step 2

Accept one of the candidate edges: (v_1, v_4) (we can do random accept here).

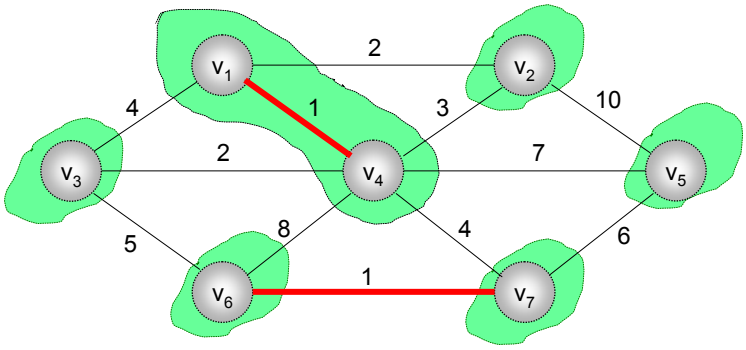


Initial Forest

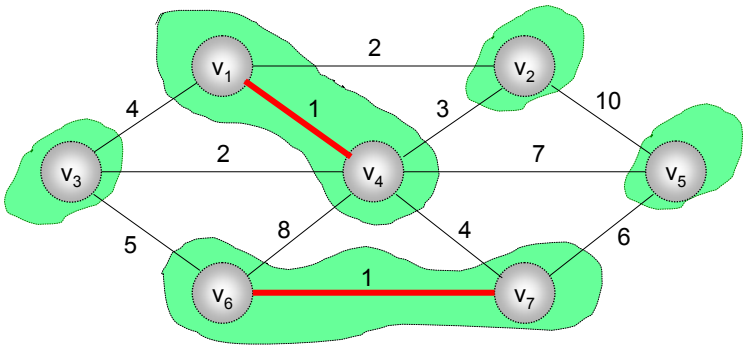
Step 3 *Merge the two trees connected by that edge. Obtain a new tree in this way.*



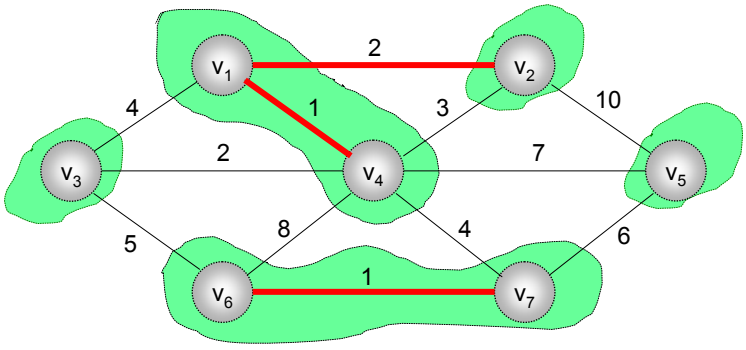
Repeat previous steps!
Edge (v_6-v_7) is accepted.



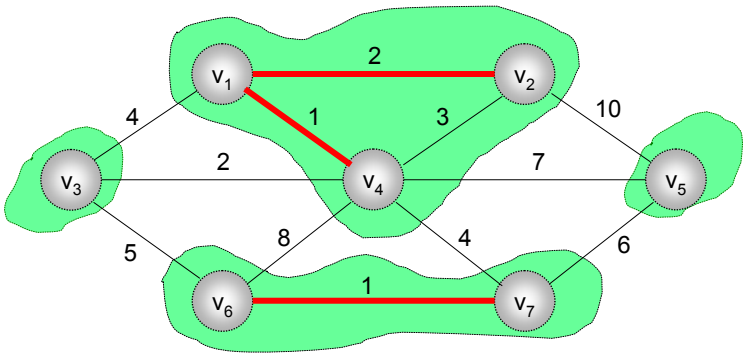
Merge the two trees connected by that edge!



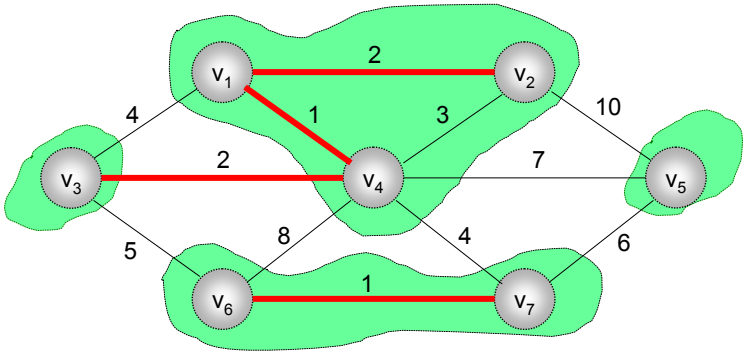
Accept edge (v_1, v_2)



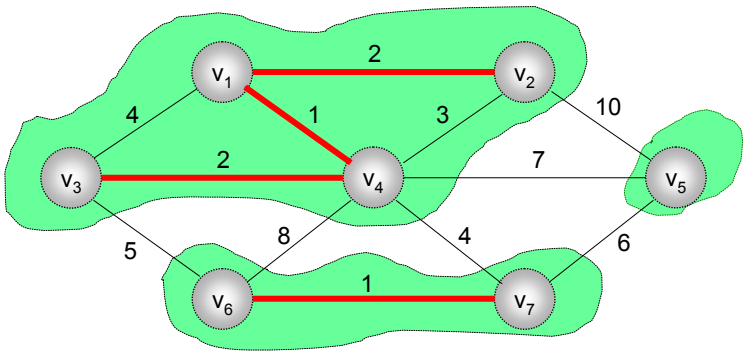
Merge the two trees connected by that edge!



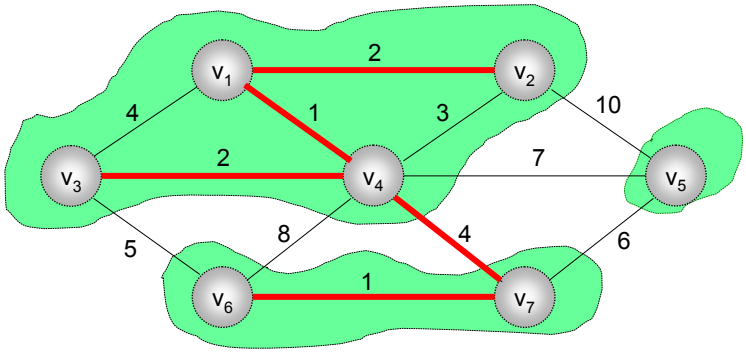
Accept edge (v_3, v_4)



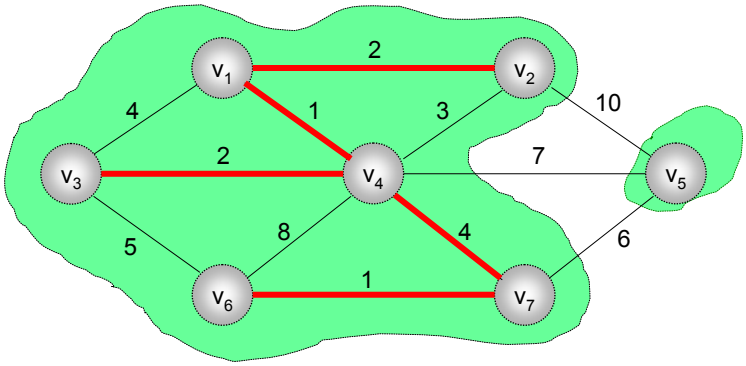
Merge the two trees connected by that edge!



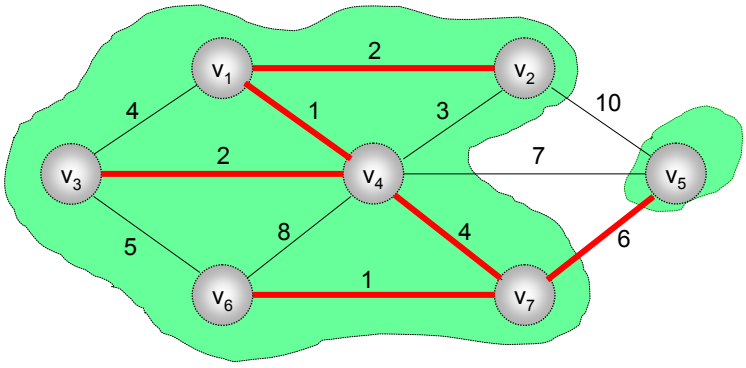
Accept edge (v_4, v_7)



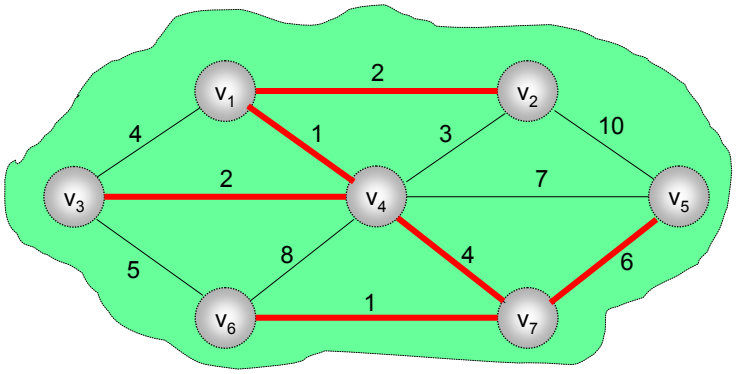
Merge the two trees connected by that edge!



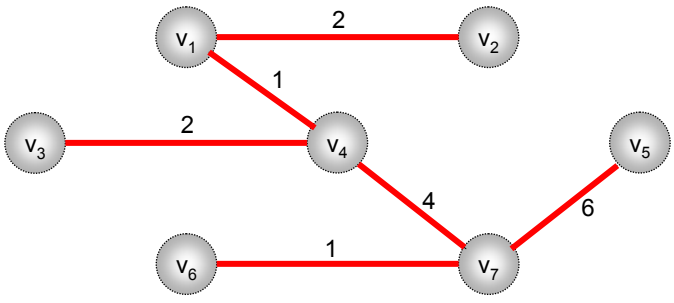
Accept edge (v_7, v_5)



Merge the two trees connected by that edge!



Finished!
The resulting MST is shown below!



```
void Graph::kruskal()
{
        int edgesAccepted;  DisjSet s(NUM_VERTICES);
        PriorityQueue h(NUM_EDGES);
        Vertex u, v;    SetType uset, vset;    Edge e;

        h = readGraphIntoHeapArray();
        h.buildHeap();
        edgesAccepted = 0;

        while (edgesAccepted < NUM_VERTICES – 1)
        {
                h.deleteMin(e); // Edge e = (u,v)
                uset = s.find(u);
                vset = s.find(v);
                if (uset != vset)
                {
                        // Accept the edge
                        edgesAccepted++
                        s.unionSets (uset,m vset);
                }
        }
}
```