UMBC

AN HONORS UNIVERSITY IN MARYLAND

Computational Methods in IS Research
Fall 2017

Graph Algorithms
Minimum Spanning Tree

Nirmalya Roy

Department of Information Systems

University of Maryland Baltimore County

Minimum Spanning Tree Problem

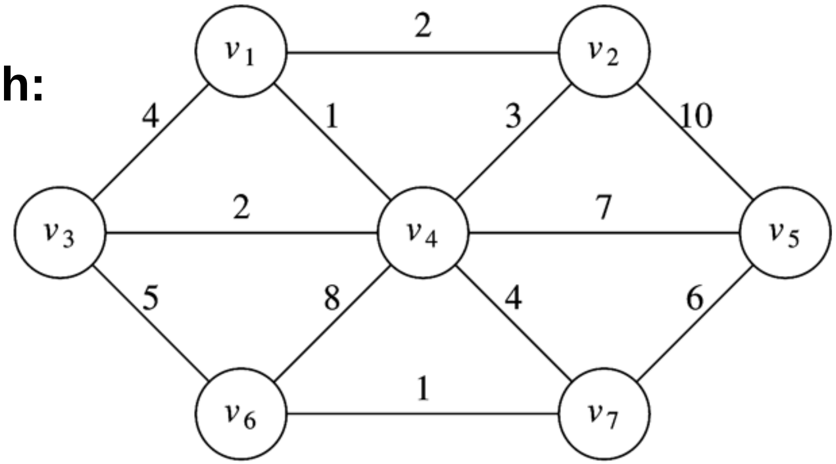
- Find a minimum-cost set of edges that *connect all vertices* of a graph at *lowest total cost*
- Applications
  - Connecting "nodes" with a minimum of "wire"
    - Networking
    - Circuit design
  - Collecting nearby nodes
    - Clustering, taxonomy construction
  - Approximating graphs
    - Most graph algorithms are faster on trees

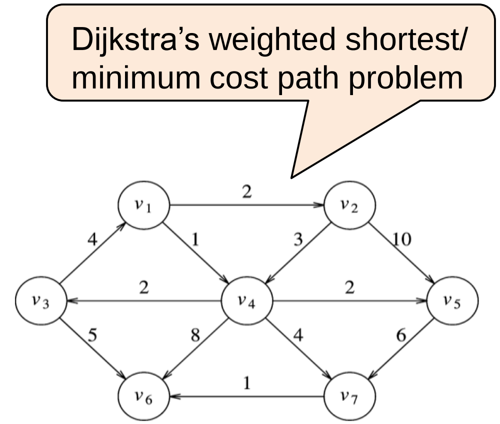
Minimum Spanning Tree

- A tree is an acyclic, undirected, connected graph
- A spanning tree of a graph is a tree containing all vertices from the graph
- A minimum spanning tree is a spanning tree, where the sum of the weights on the tree's edges are minimal

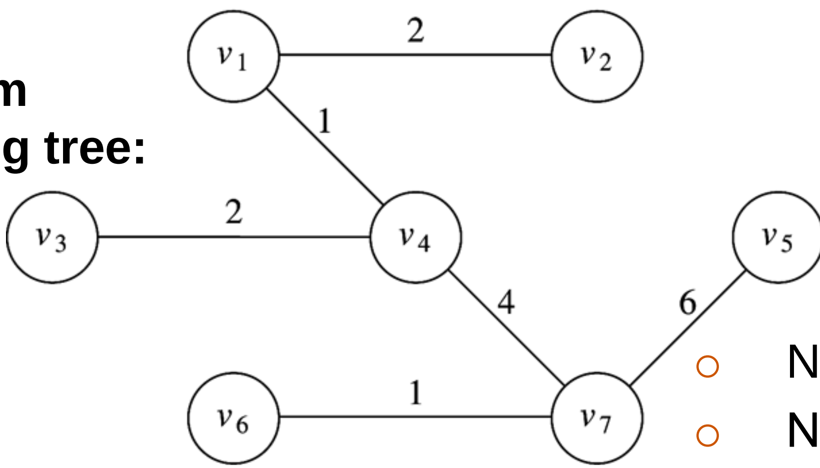
Minimum Spanning Tree (cont'd)

Graph:





Minimum spanning tree:



- Number of edges in MST is |V| - 1
- No fixed start vertex like Dijkstra's
- Undirected graph
- Minimizing summation of total edge cost instead of finding distinct shortest path

Minimum Spanning Tree (cont'd)

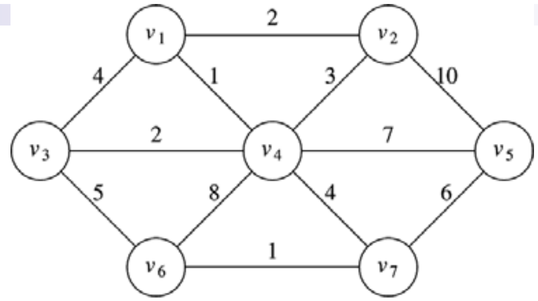
- Problem
  - Given an undirected, weighted graph $G = (V, E)$ with weights $w(u, v)$ for each edge $(u, v) \in E$
  - Find an acyclic, connected graph $G' = (V', E')$, $E' \subseteq E$, that minimizes $\sum_{(u, v) \in E'} w(u, v)$
  - G' is a minimum spanning tree of G
    - There can be more than one minimum spanning tree of a graph G
  - Two algorithms
    - Prim's algorithm
    - Kruskal's algorithm
    - Differ in how a minimum edge is selected

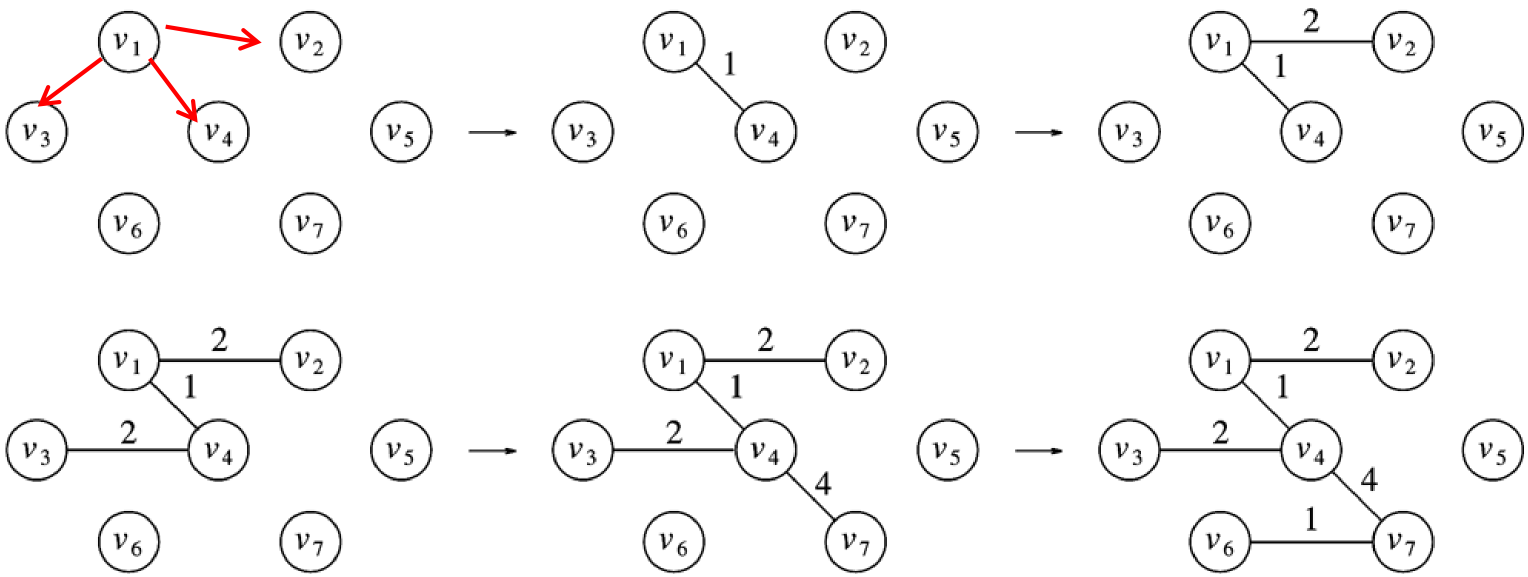
Minimum Spanning Tree

- Solution #1 (Prim's Algorithm (1957))
  - Start with an empty tree T
  - T = {x}, where x is an arbitrary node in the input graph
  - While T is not a spanning tree
    - Find the **lowest-weight edge** that connects a vertex in T to a vertex not in T
    - Add this edge to T
  - T will be a minimum spanning tree

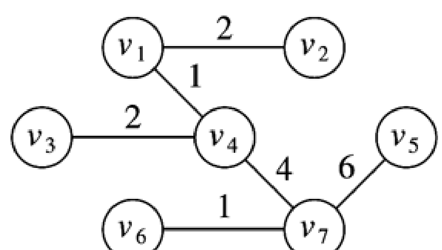
Prim's Algorithm: Example

Input graph:





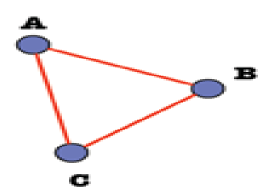
Minimum spanning tree:

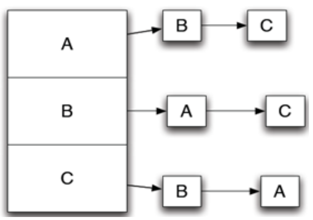


Prim’s Algorithm

- Similar to Dijkstra’s shortest-path algorithm
- Except
  - `v.known` = v in T
  - `v.dist` = weight of lowest-weight edge connecting v to a `known` vertex in T
  - `v.path` = last neighboring vertex changing (lowering) v’s dist value (same as before)
  - Undirected graph, so two entries for every edge in the adjacency list

```
struct Vertex
{
    List      adj;
    bool      known;
    DistType  dist;
    Vertex    path;
        // Other data
};
```



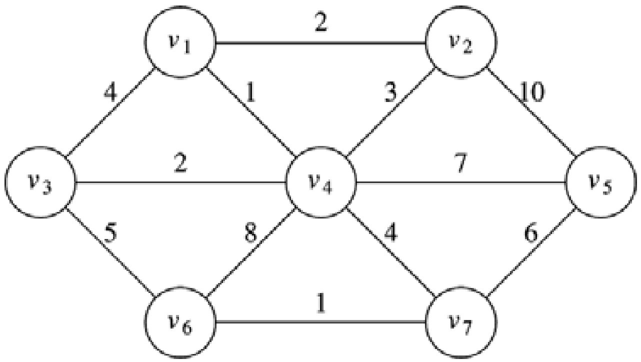


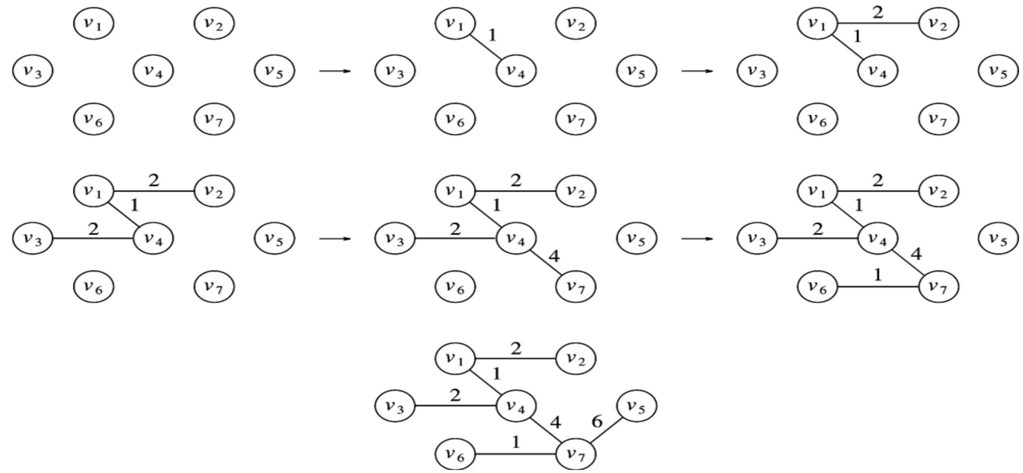
Prim’s Algorithm (cont’d)

```
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
    }
}
```

Running time same as Dijkstra: O(|E| log |V|) using binary heaps

Prim's Algorithm: Example



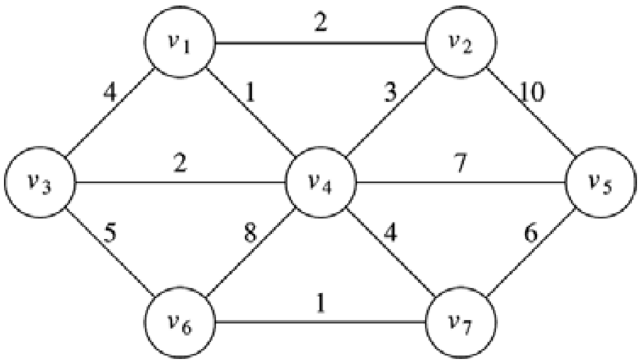


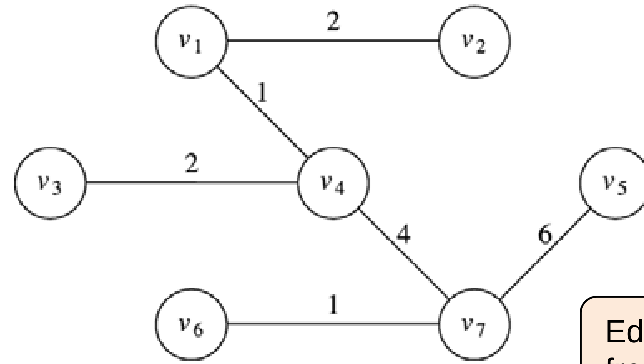
v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v	*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

v	*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

Prim's Algorithm: Example (cont'd)





Edges can be read from this final table

v	*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

v	*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

v	*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

Prim's Algorithm: Analysis

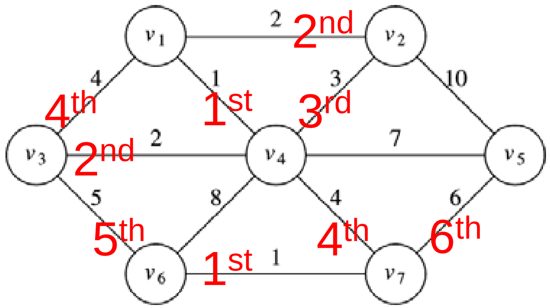
- Running time = $O(|V|^2)$ without heap
  - Optimal for dense graph
- $O(|E| \log |V|)$ using binary heaps
  - Good for sparse graph

Minimum Spanning Tree

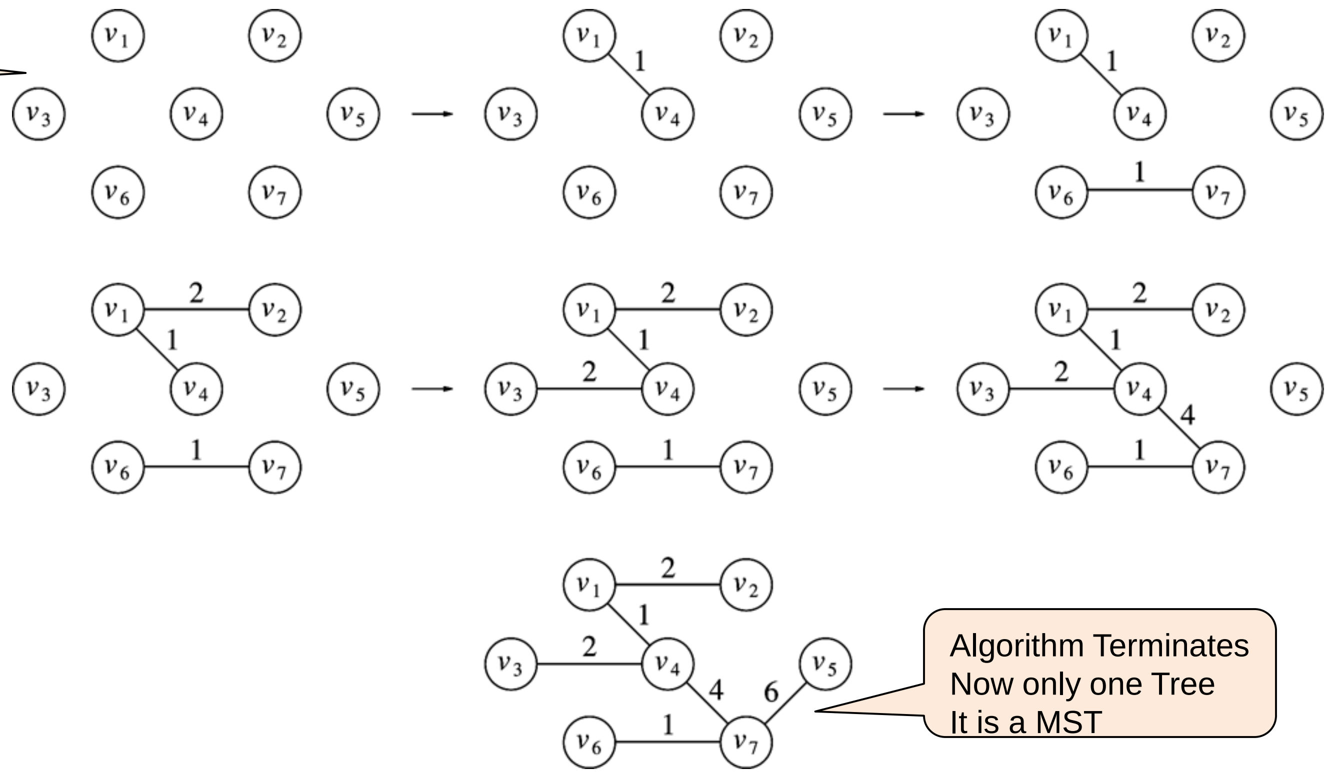
- Solution #2 (Kruskal's algorithm (1956))
  - Start with T = V (with no edges)
  - For each edge in increasing order by weight
    - If adding edge to T does not create a cycle
    - Then add edge to T
  - T will be a minimum spanning tree

Kruskal’s Algorithm: Example

Input graph:



Collection of Trees
|V| single-node Trees



Edge	Weight	Action
(v_1, v_4)	1	Accepted
(v_6, v_7)	1	Accepted
(v_1, v_2)	2	Accepted
(v_3, v_4)	2	Accepted
(v_2, v_4)	3	Rejected
(v_1, v_3)	4	Rejected
(v_4, v_7)	4	Accepted
(v_3, v_6)	5	Rejected
(v_5, v_7)	6	Accepted

Algorithm Terminates
Now only one Tree
It is a MST

Kruskal's Algorithm

```
void Graph::kruskal( )
{
    int edgesAccepted = 0;
    DisjSet ds( NUM_VERTICES );
    PriorityQueue<Edge> pq( getEdges( ) );
    Edge e;
    Vertex u, v;

    while( edgesAccepted < NUM_VERTICES - 1 )
    {
        pq.deleteMin( e );      // Edge e = (u. v)
        SetType uset = ds.find( u );
        SetType vset = ds.find( v );
        if( uset != vset )
        {
            // Accept the edge
            edgesAccepted++;
            ds.unionSets( uset, vset );
        }
    }
}
```

Uses Disjoint Set and Priority Queue data structures

The edges can be sorted, but building a heap in linear time is a better option

deleteMins give the edge to be tested in order

deleteMin: O(|V| log |V|)

find: O(|E| log |V|)

Kruskal's Algorithm: Analysis

- Worst-case: $O(|E| \log |E|)$
- Since $|E| = O(|V|^2)$, worst-case also $O(|E| \log |V|)$
  - Running time dominated by heap operations
- Typically terminates before considering all edges, so faster in practice

Summary

- Finding set of edges that minimally connect all vertices in a graph
- Fast algorithm with many important applications
- Utilizes advanced data structures to achieve fast performance