Minimum Spanning Tree

Bruce Nan

Email: xiaomingnan@gmail.com

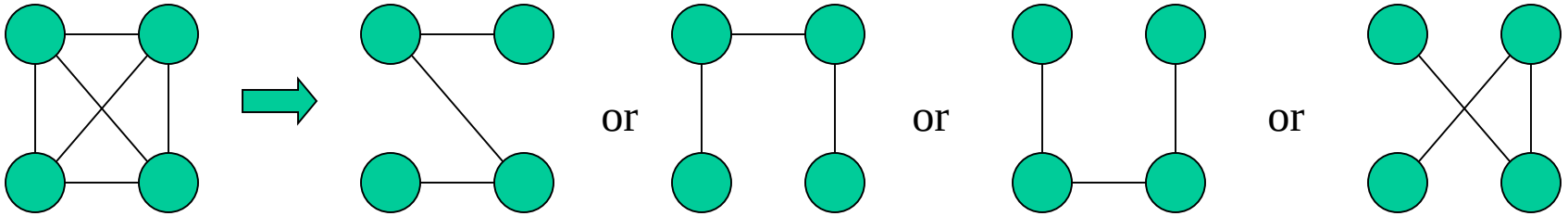
Spanning Trees

A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree.

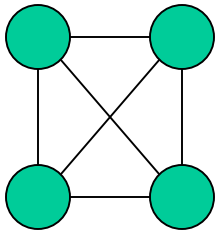
A graph may have many spanning trees.

Graph A

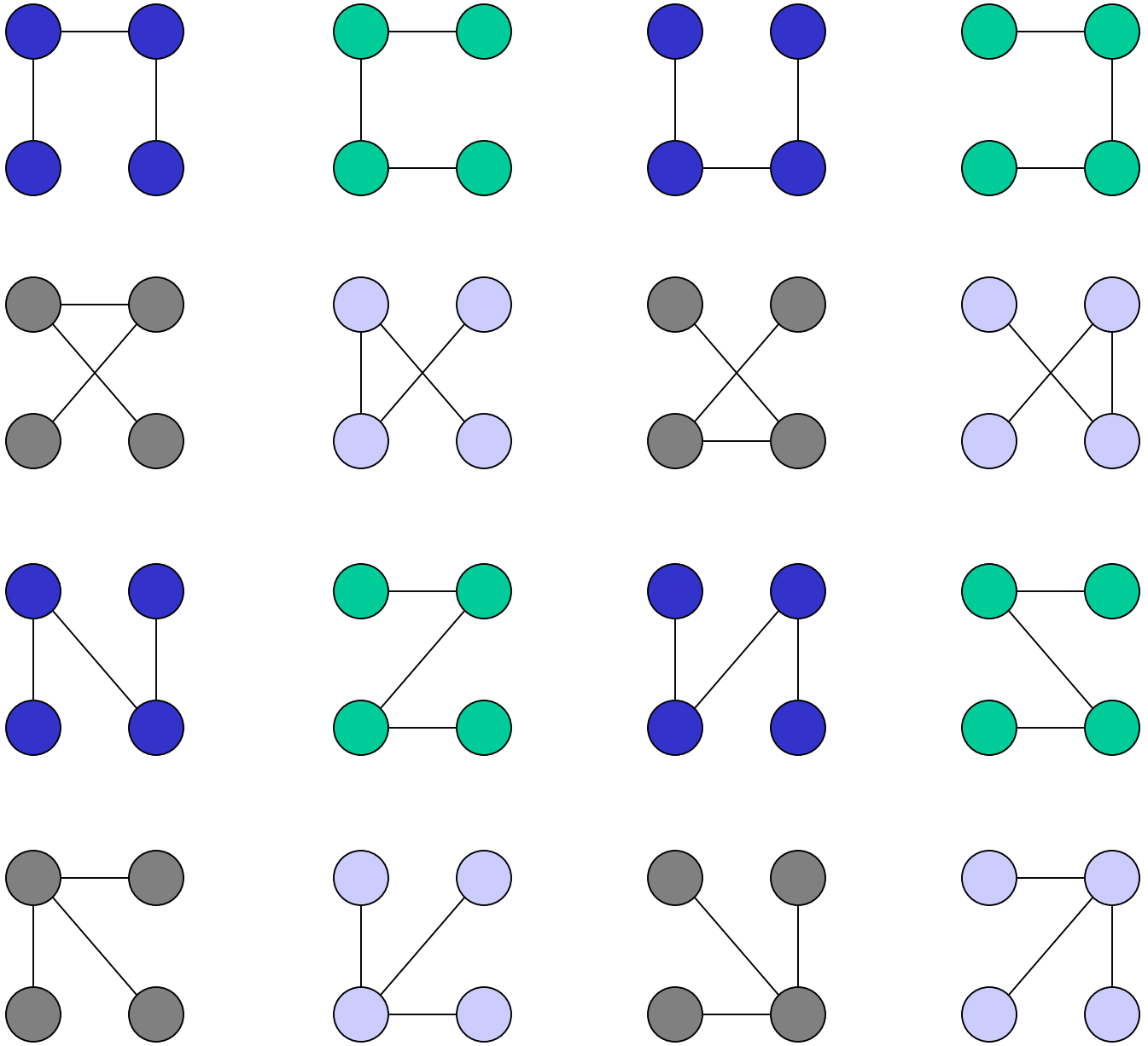
Some Spanning Trees from Graph A



Complete Graph



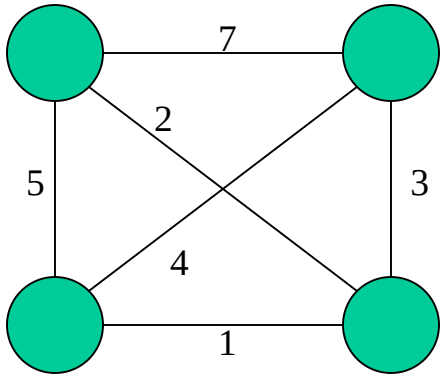
All 16 of its Spanning Trees



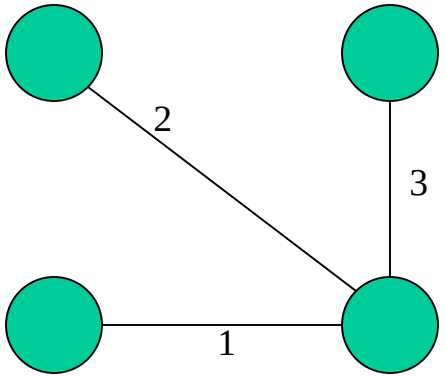
Minimum Spanning Trees

The Minimum Spanning Tree for a given graph is the Spanning Tree of minimum cost for that graph.

Complete Graph

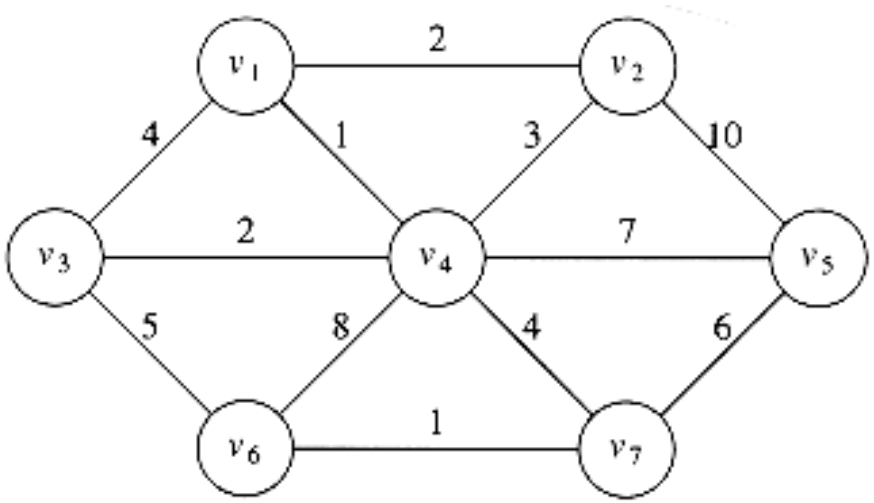


Minimum Spanning Tree



Minimum Spanning Tree

- A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at lowest total cost.
- A minimum spanning tree exists if and only if G is connected.



Notice

- Notice that the number of edges in the minimum spanning tree is |V| - 1.
- The minimum spanning tree is a tree because it is acyclic, it is spanning because it covers every edge, and it is minimum for the obvious reason.
- If we need to wire houses with a minimum of cable, then a minimum spanning tree problem needs to be solved.
- There are two basic algorithms to solve this problem; both are greedy.

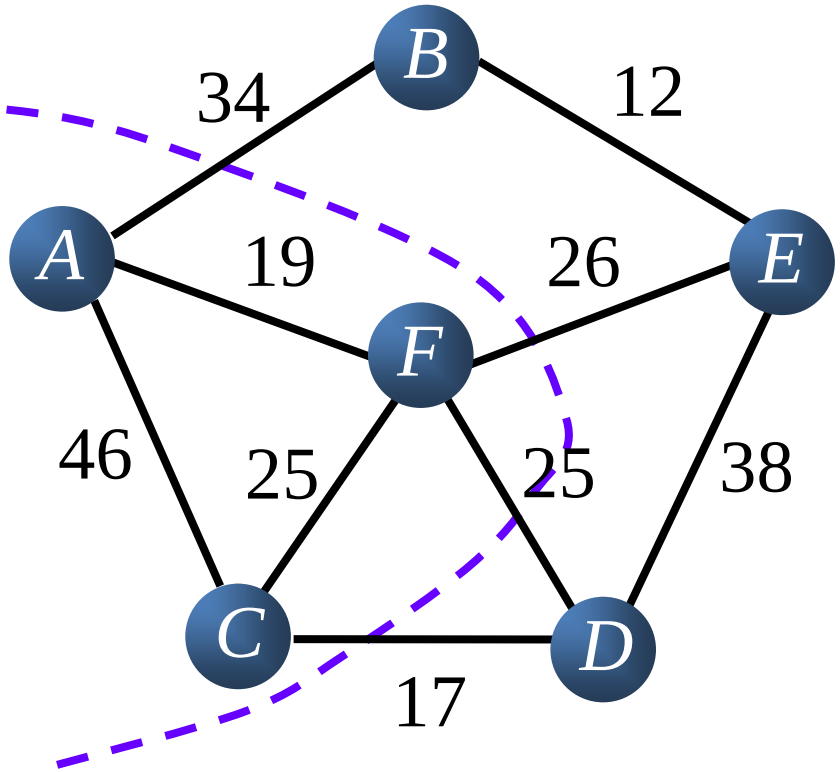
Prim's Algorithm

- One way to compute a minimum spanning tree is to grow the tree in successive stages. In each stage, one node is picked as the root, and we add an edge, and thus an associated vertex, to the tree.
- At any point in the algorithm, we can see that we have a set of vertices that have already been included in the tree; the rest of the vertices have not.
- The algorithm then finds, at each stage, a new vertex to add 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

- Input: A non-empty connected weighted graph with vertices *V* and edges *E* (the weights can be negative).
- Initialize: *V*new = {*x*}, where *x* is an arbitrary node (starting point) from *V*, *E*new = {}
- Repeat until *V*new = *V*:
  - Choose an edge (*u*, *v*) with minimal weight such that *u* is in *V*new and *v* is not (if there are multiple edges with the same weight, any of them may be picked)
  - Add *v* to *V*new, and (*u*, *v*) to *E*new
- Output: *V*new and *E*new describe a minimal spanning tree

Prim Algorithm

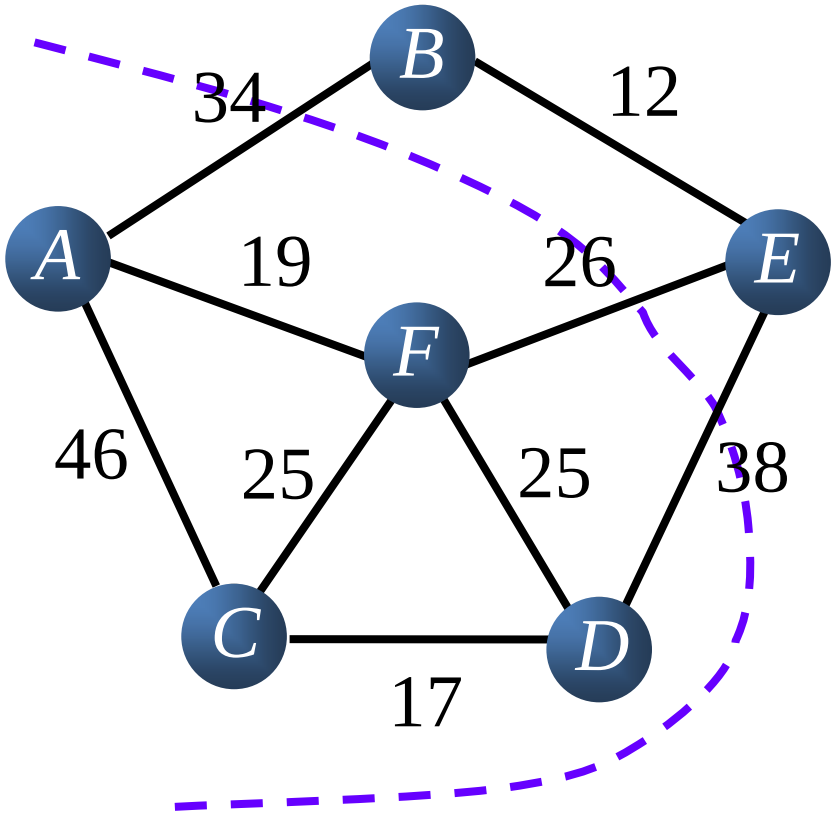


U={A, F, C}

V-U={B, D, E}

cost={(*A*, *B*)34, (*C*, *D*)17, *(F,D) 25*, (*F*, *E*)26}

Prim Algorithm

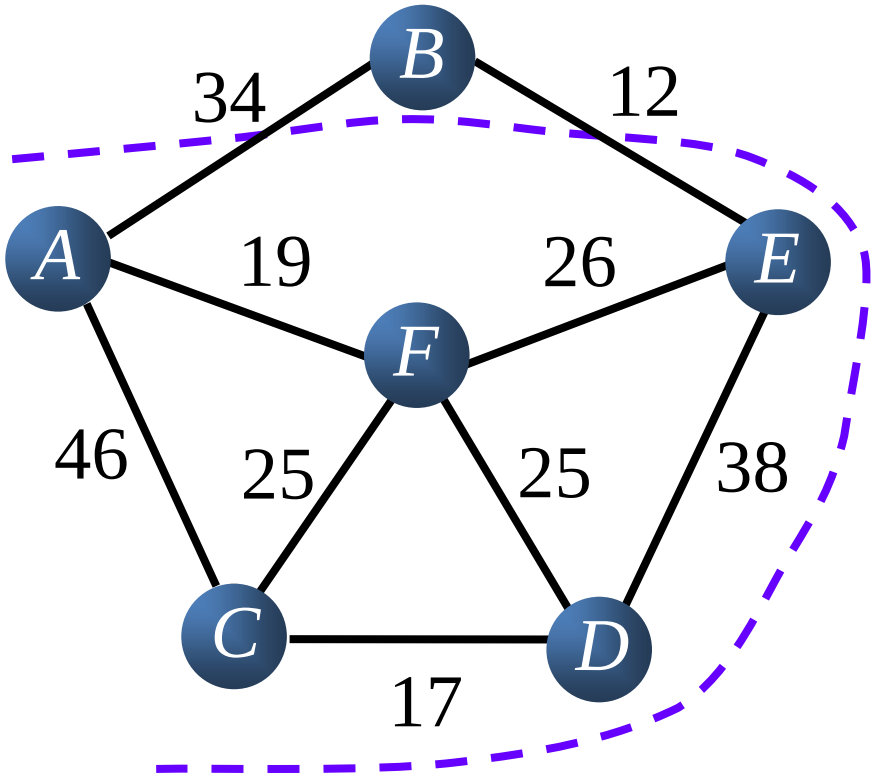


U={A, F, C, D}

V-U={B, E}

cost={(*A*, *B*)34, (*F*, *E*)26}

Prim Algorithm

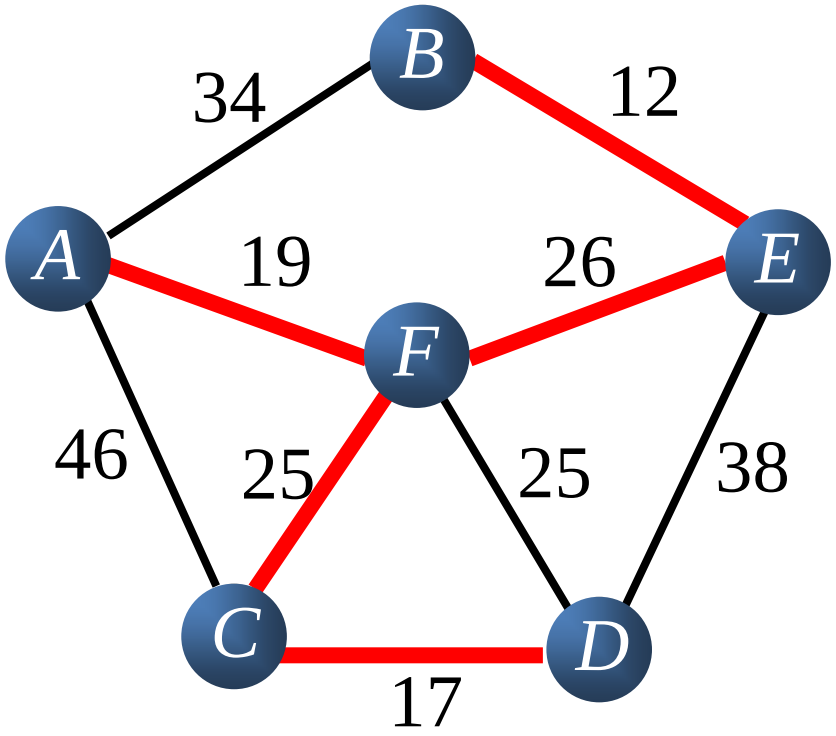


U={A, F, C, D, E}

V-U={B}

cost={(*E*, *B*)12}

Prim Algorithm



U={A, F, C, D, E, B}

V-U={ }

How to Implement Prim's Algorithm?

Data Structure

- Data Structure
  - array lowcost[n]: represent the minimum cost of all vertices of V-U set and all vertices of U set
  - array adjvex[n]: represent the adjacent vertex in V-U set with minimum cost edge connecting with vertex in U set

$$\begin{cases} \text{lowcost[i]=w} & \text{The cost between vertex } v_i \text{ and } v_k \text{ is w}, \ v_i \in V-U \\ \text{adjvex[i]=k} & \text{and } v_k \in U \end{cases}$$

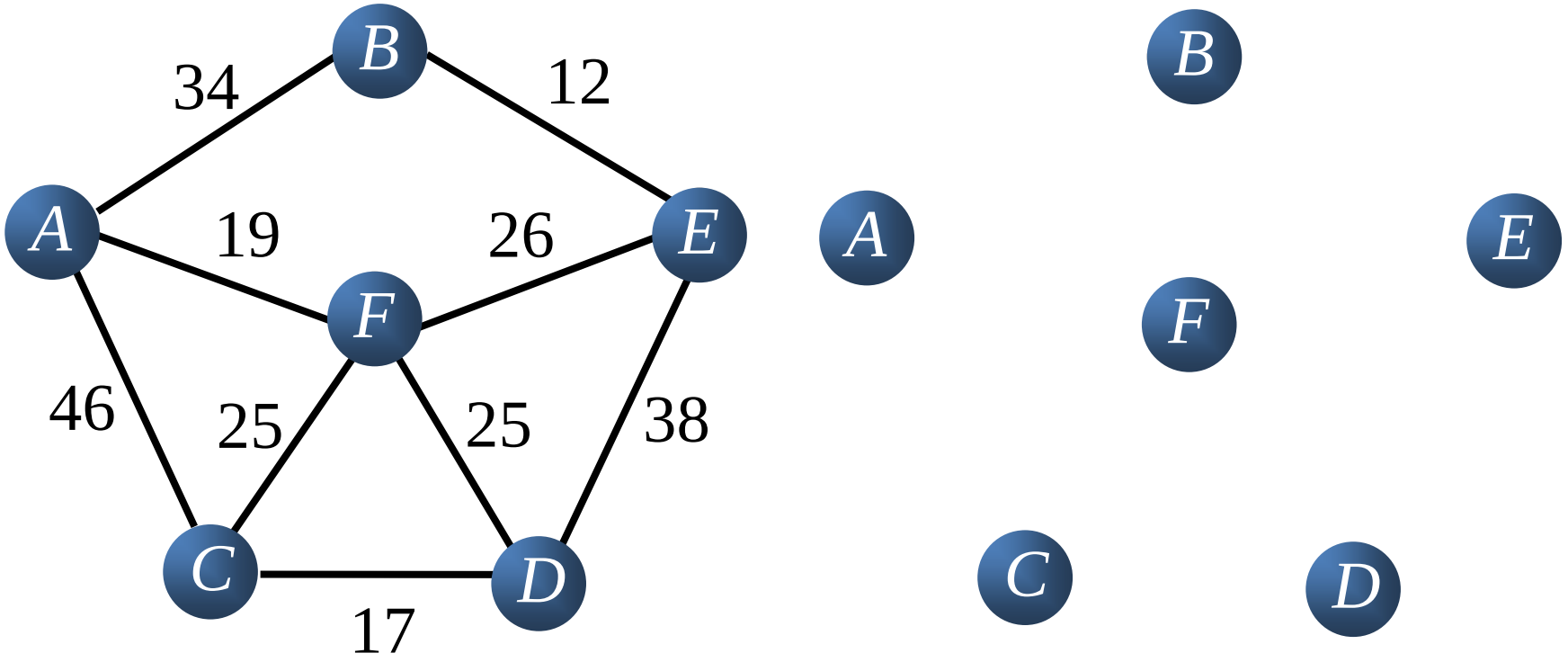
Pseudo Code of Prim's Algorithm

1. Initial two arrays lowcost[] and adjvex[]
2. Output u_0 and add u_0 into U set
3. Repeat until V-U is empty
   1. select minimum cost edge from lowcost and add corresponding adjvex vertex k
   2. output vertex k
   3. add k into U set
   4. update lowcost[] and adjvex[] arrays

Kruskal's Algorithm

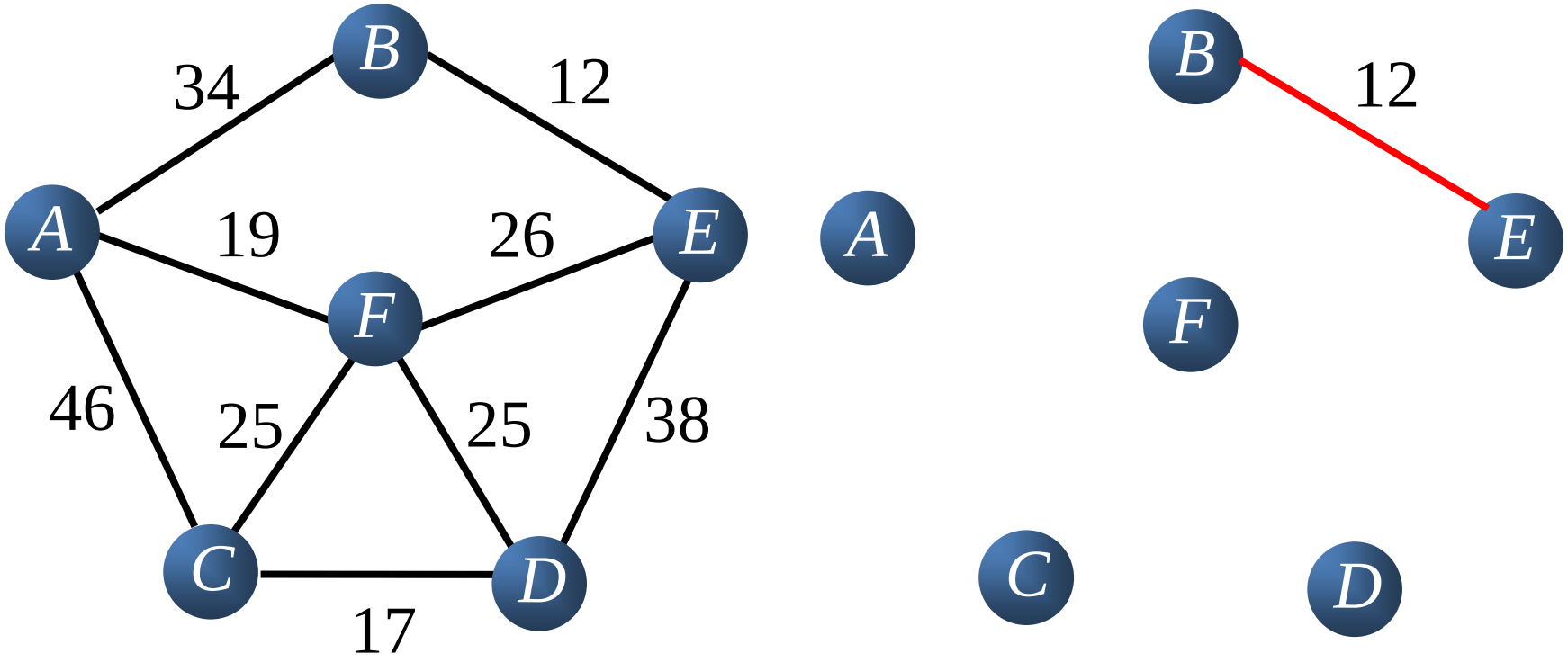
- A second greedy strategy is continually to select the edges in order of smallest weight and accept an edge if it does not cause a cycle.
- The algorithm terminates when enough edges are accepted. It turns out to be simple to decide whether edge (u,v) should be accepted or rejected.

Kruskal Example



Component＝{A}, {B}, {C}, {D}, {E}, {F}

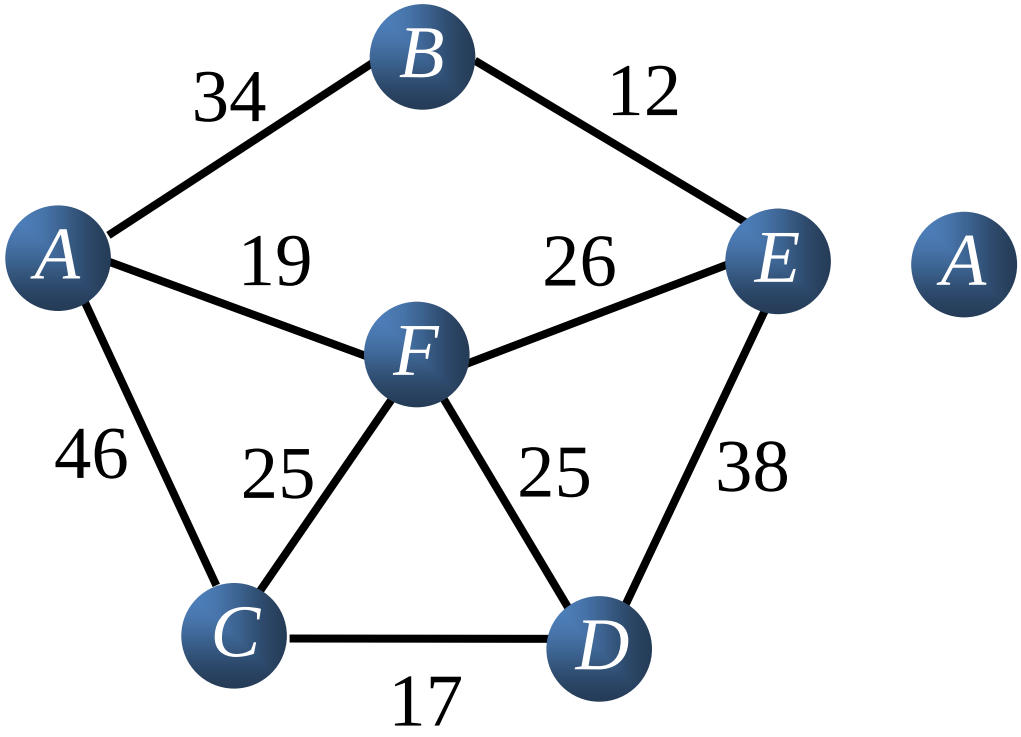
Kruskal Example

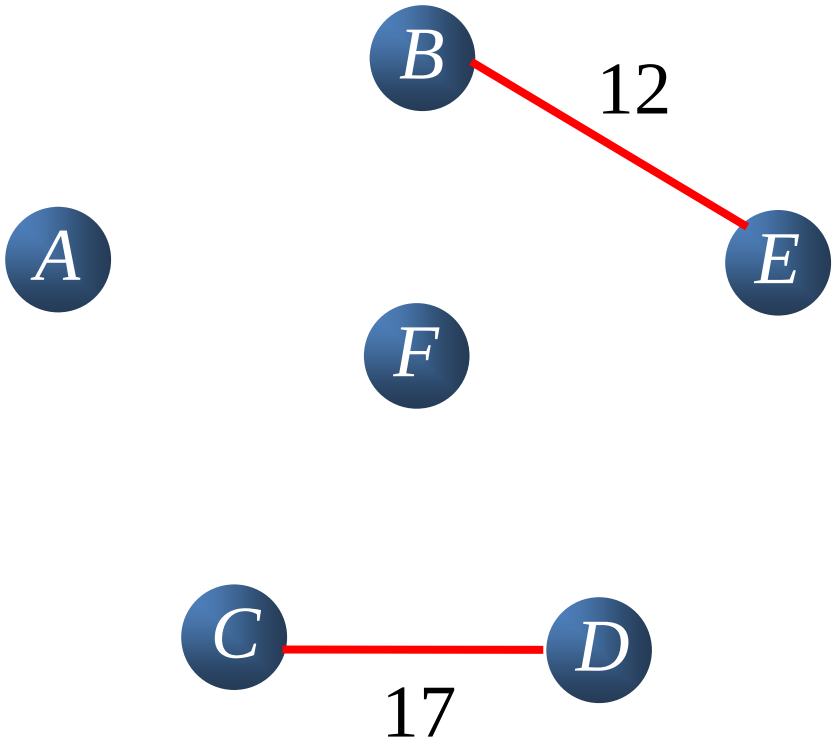


Component＝{A}, {B}, {C}, {D}, {E}, {F}

Component＝{A}, {B, E}, {C}, {D}, {F}

Kruskal Example

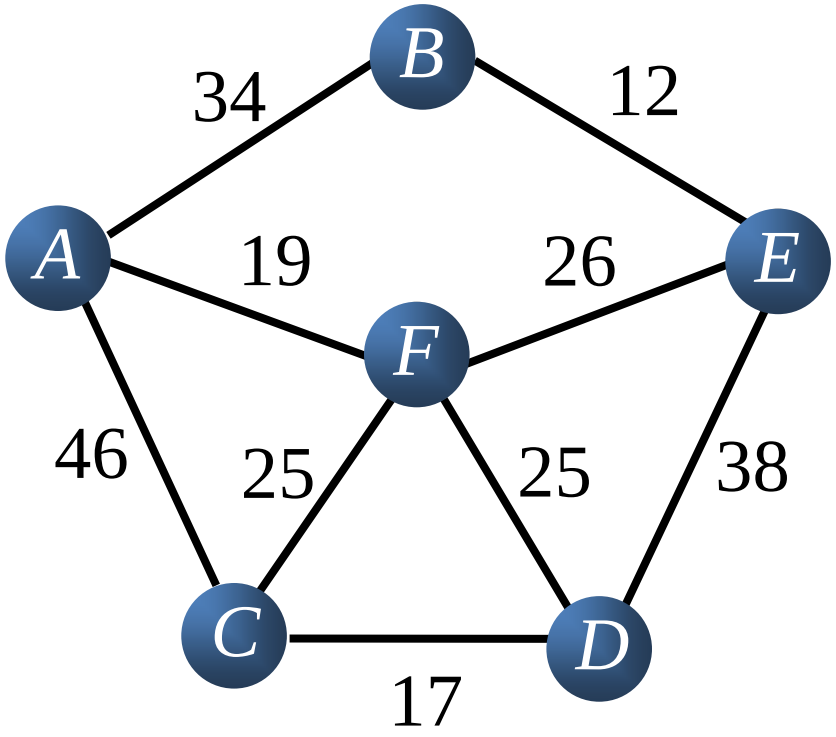


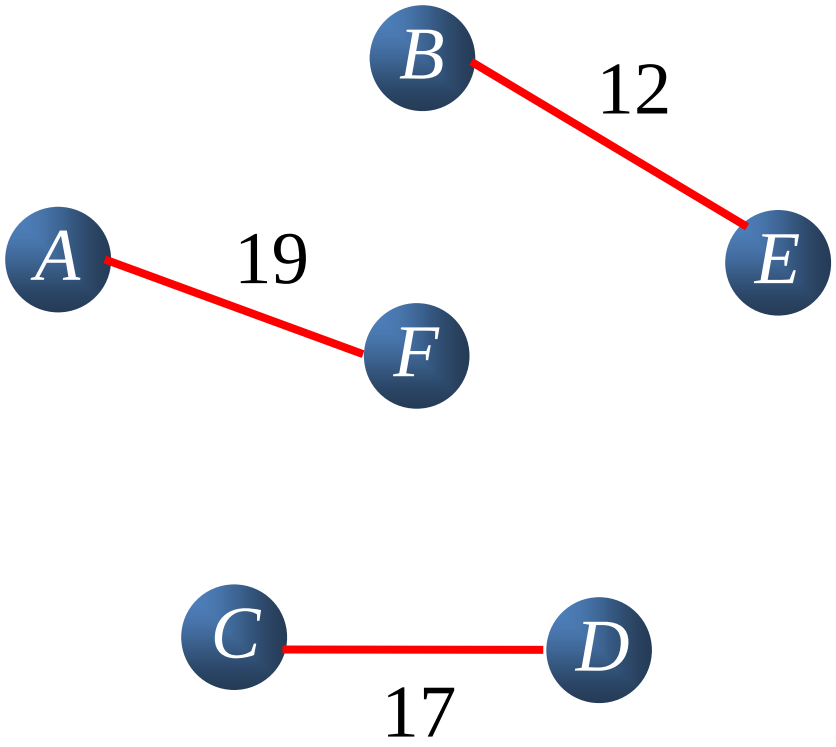


Component＝{A}, { F}, {B, E}, {C}, {D}

Component＝{A}, { F}, {B, E}, {C, D}

Kruskal Example

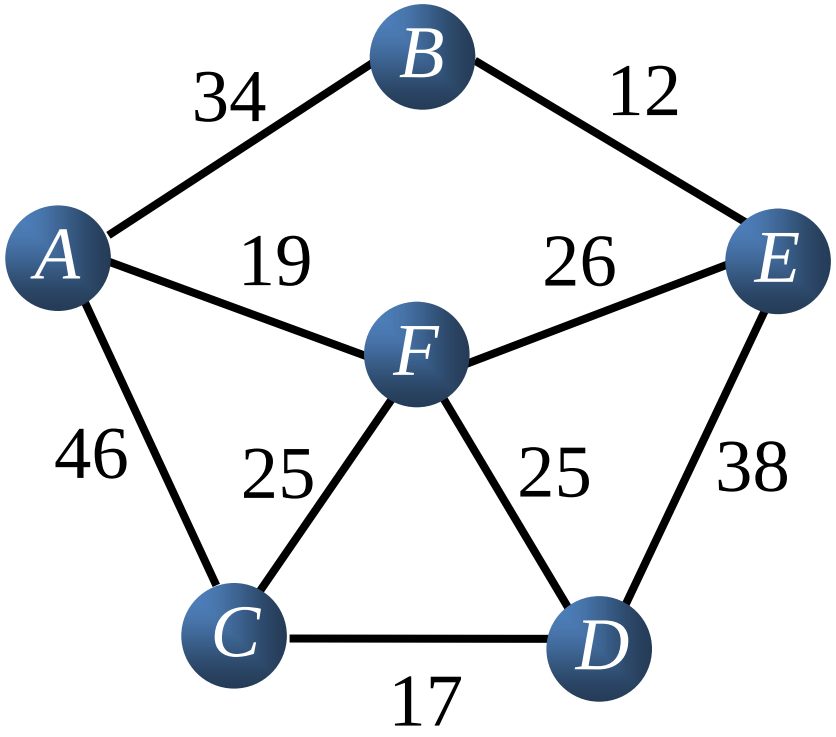


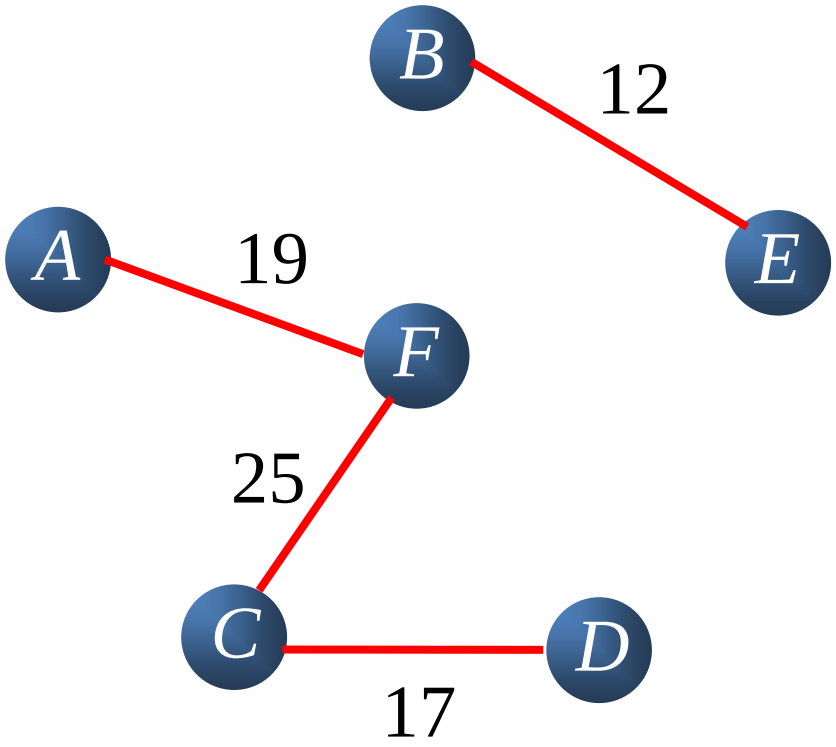


Component＝{A}, {B, E}, {C, D}, {F}

Component＝{A, F}, {B, E}, {C, D}

Kruskal Example

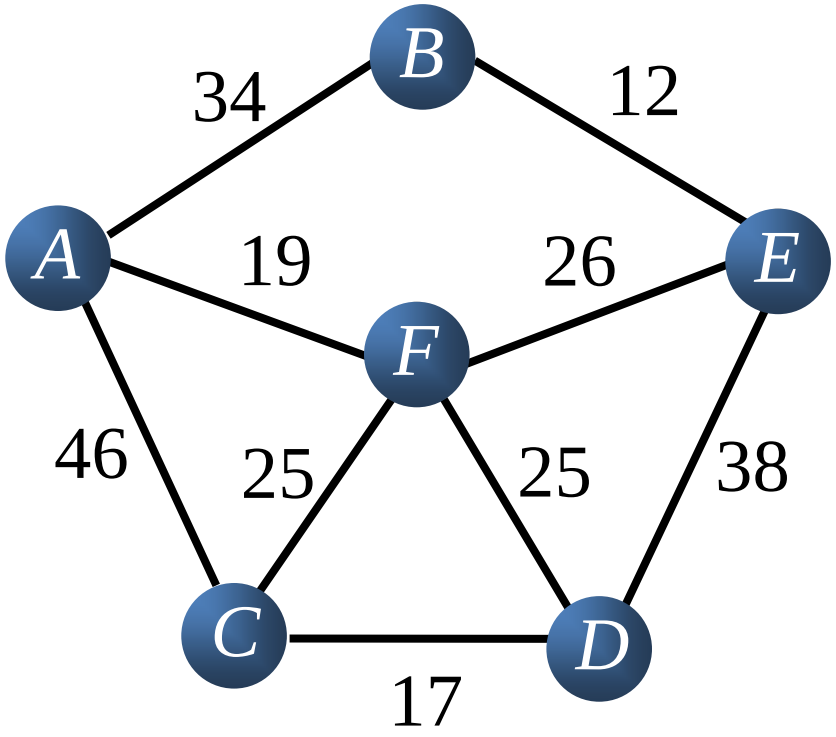


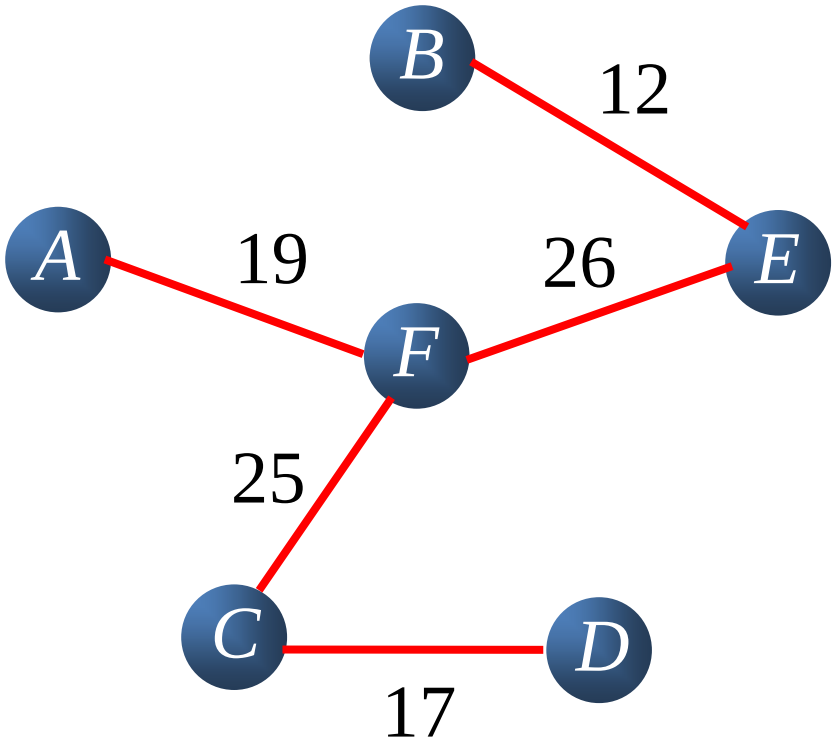


Component＝{A, F}, {B, E}, {C, D}

Component＝{A, F, C, D}, {B, E}

Kruskal Example





Component = {A, F, C, D}, {B, E}

Component = {A, F, C, D, B, E}

How to Implement Kruskal's Algorithm?

How to verify whether there is a circle when adding an edge?

Pseudo Code of Kruskal's Algorithm

```
initialize( S );
read_graph_into_heap_array( G, H );
build_heap( H );
edges_accepted = 0;
while( edges_accepted < NUM_VERTEX-1 )
{
    e = delete_min( H ); /* e = (u, v) */
    u_set = find( u, S );
    v_set = find( v, S );
    if( u_set != v_set )
    {
        edges_accepted++;
        set_union( S, u_set, v_set );
    }
}
```

Thank You