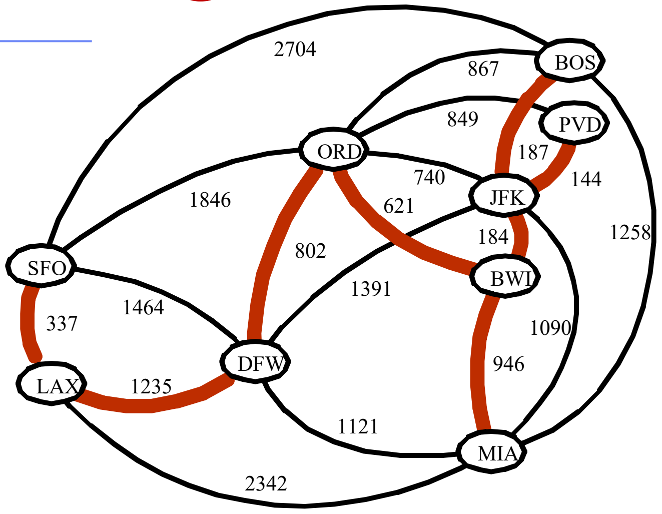
Lecture 17: Minimum Spanning Trees

Infinite Correlation



Wholeness Statement

A minimum spanning tree is a spanning tree subgraph with minimum total edge weight. Efficient greedy algorithms have been developed to compute MST both with and without special data structures. Pure creative intelligence is the source of all creative algorithms. Regular practice of TM improves our ability to make use of our own innate creative potential.

Outline and Reading

- Minimum Spanning Trees
  - Definitions
  - Cycle Property
- The Prim-Jarnik Algorithm (1957, 1930)
- Kruskal's Algorithm (1956)

Minimum Spanning Tree

Spanning subgraph

- Subgraph of a graph G containing all the vertices of G

Spanning tree

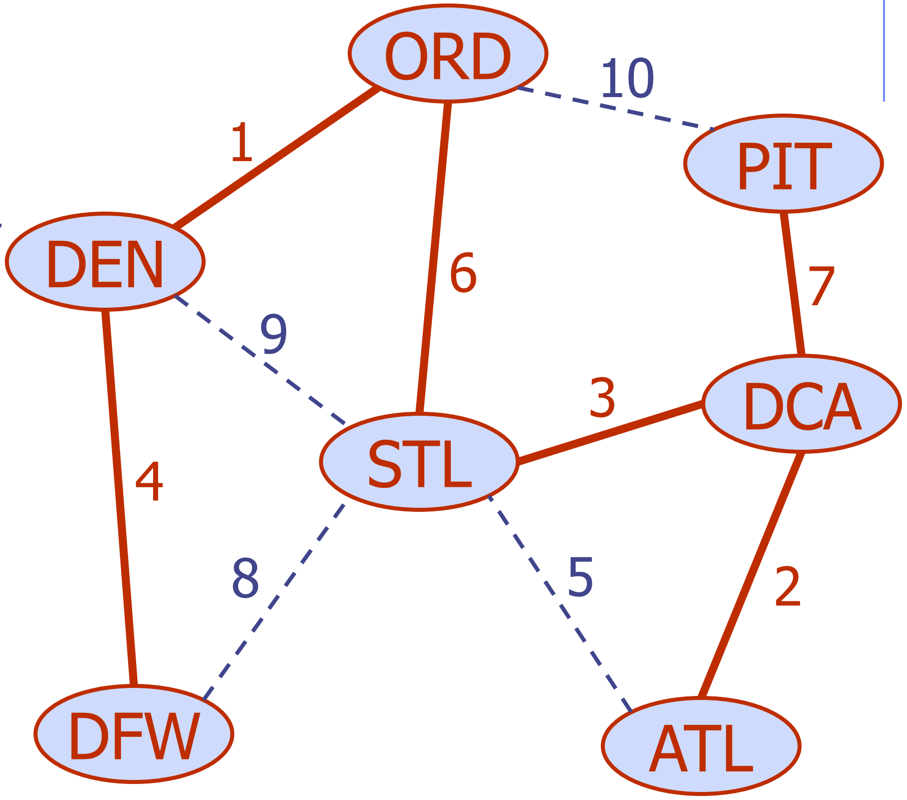
- Spanning subgraph that is itself a tree

Minimum spanning tree (MST)

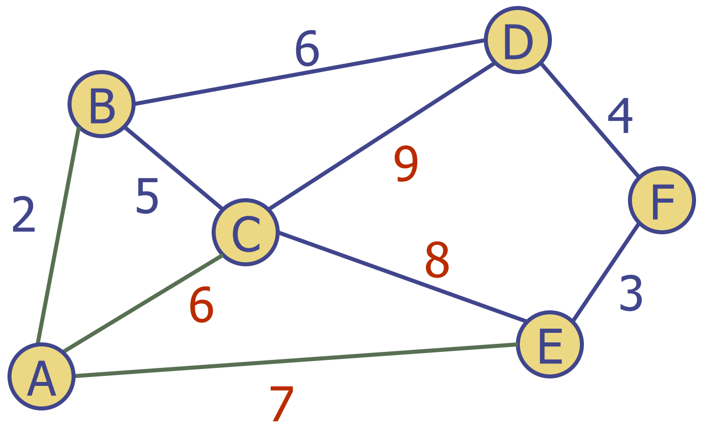
- Spanning tree of a weighted graph with minimum total edge weight

Applications(by providing the cheapest way to achieve connectivity)

- Communications networks
- Transportation networks

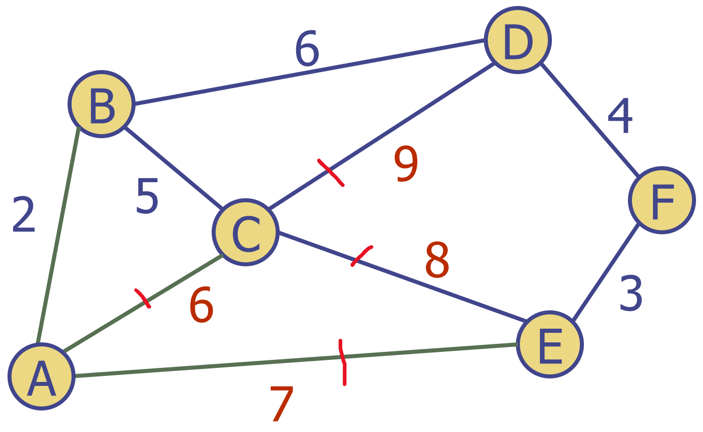


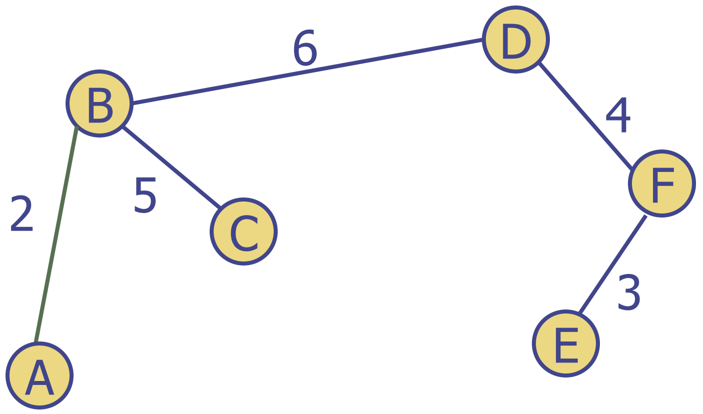
Cycle Property



If the weight of an edge ***e*** of a cycle C is larger than the weights of other edges of C, then this edge cannot belong to a MST.

Cycle Property





If the weight of an edge ***e*** of a cycle C is larger than the weights of other edges of C, then this edge cannot belong to a MST.

Cycle Property

- **Cycle Property**: For any cycle C in a graph, if the weight of an edge ***e*** of C is larger than the weights of other edges of C, then this edge cannot belong to a MST.

- **Proof**: Assume the contrary, i.e. that e belongs to an MST ***T1***; then deleting ***e*** will break ***T1*** into two subtrees with the two ends of ***e*** in different subtrees. The remainder of C reconnects the subtrees, hence there is an edge ***f*** of C with ends in different subtrees, i.e., it reconnects the subtrees into a tree ***T2*** with weight less than that of ***T1***, because the weight of ***f*** is less than the weight of ***e***; thus ***T1*** cannot be a MST.

Generic MST Algorithm

```
Algorithm GenericMST(G)
    T ← an Empty tree
    while T does not form a spanning tree do
        (u, v) ← a safe edge of G
        T ← (u, v) U T
    return T
```

A *safe edge* is one that when added to T forms a subgraph of a MST

Main Point

1. A minimum spanning tree algorithm gradually grows a (sub-solution) tree by adding a “safe edge” that connects a vertex in the tree to a vertex not yet in the tree.

Science Of Consciousness: The nature of life is to grow and progress to the state of enlightenment, fulfillment.

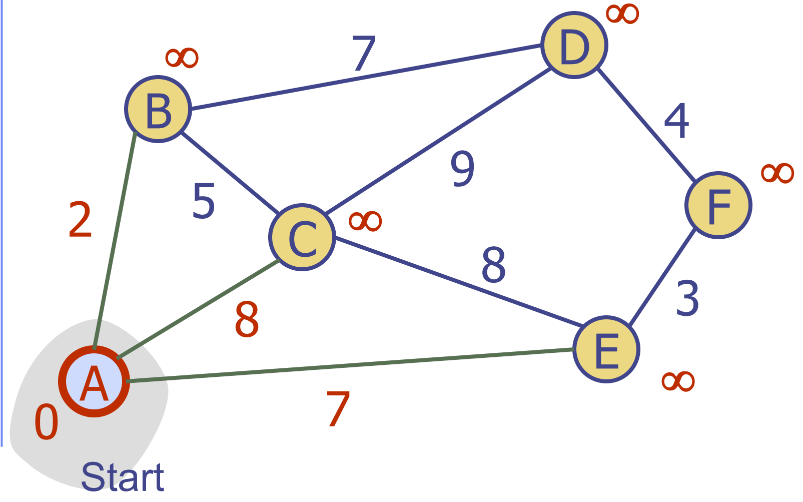
Prim(1957)-Jarnik(1930) Algorithm

AKA Dijkstra-Prim Algorithm

Prim-Jarnik's Algorithm

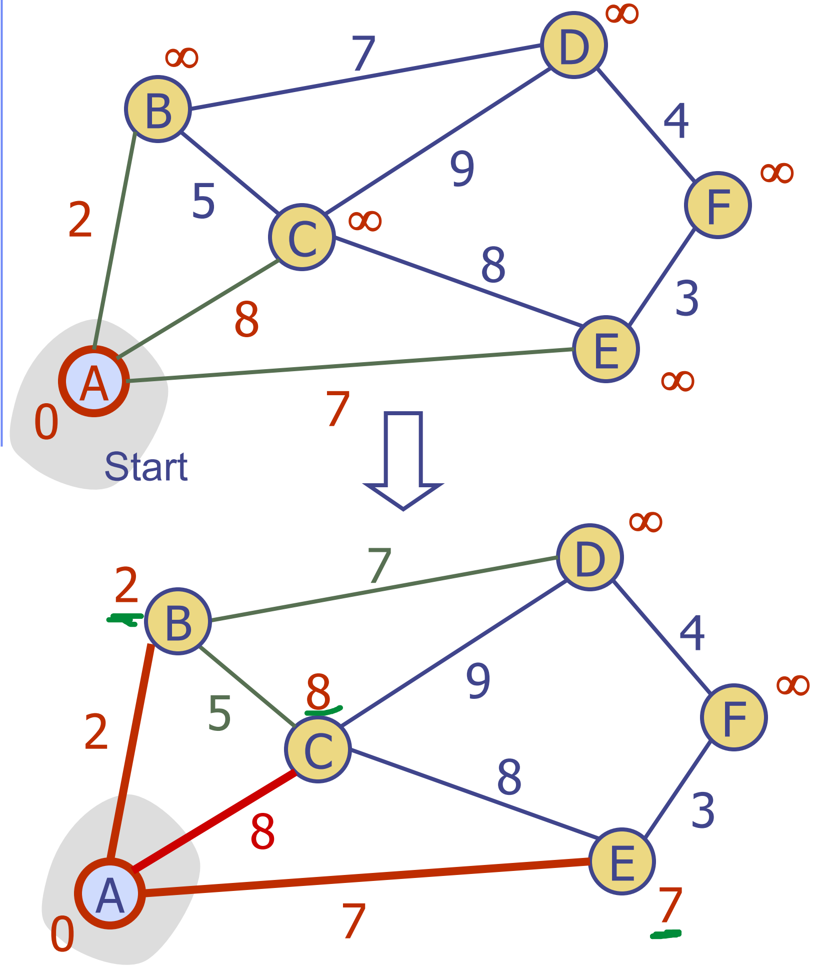
- Similar to Dijkstra's shortest path algorithm (for a connected graph)
- We pick an arbitrary vertex s and we grow the MST as a tree of vertices, starting from s
- We store with each vertex v a parent edge $parent(v)$ with the smallest weight of any edge connecting v to a vertex in the tree
- At each step:
  - We add to the tree, the vertex u outside the tree with the parent edge with the smallest/minimum weight
  - We update the parent edges of the vertices adjacent to u

Example



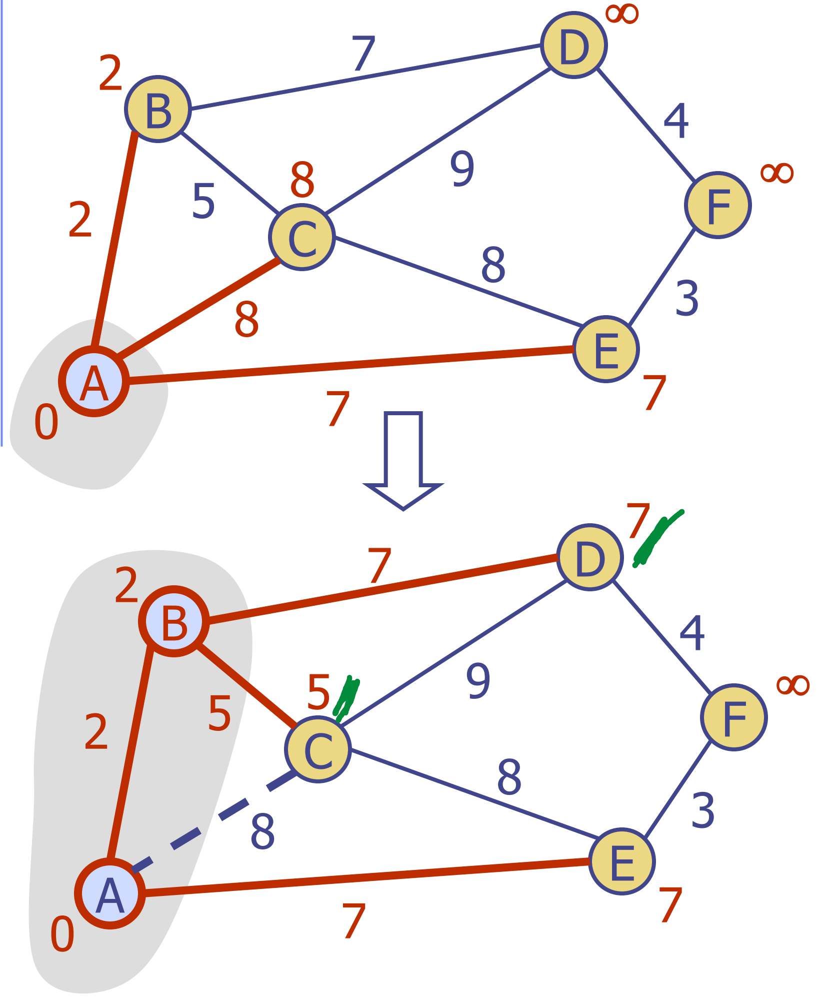
Set all other distance as ∞
and start distance as 0

Example



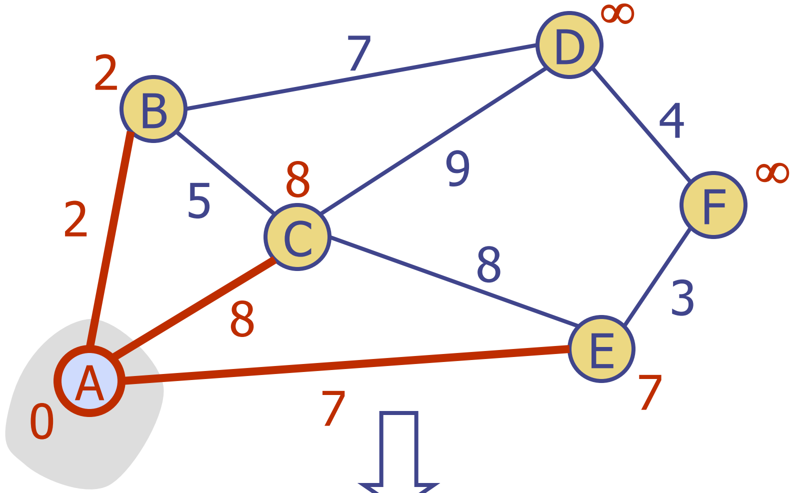
A is in the Tree and update the distance of its adjacent with its weight.

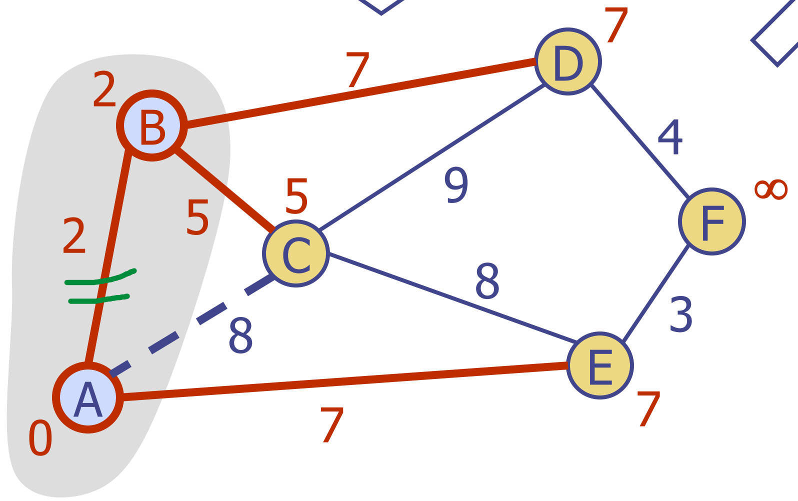
Example

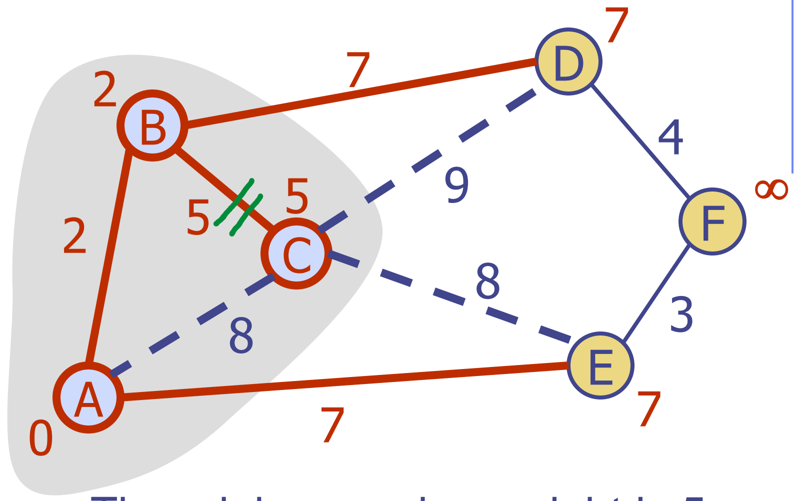


The minimum edge weight is 2. Update the parent edge of B(A). Now B is in the Tree. Continue the process. Update the distance of B's adjacent

Example





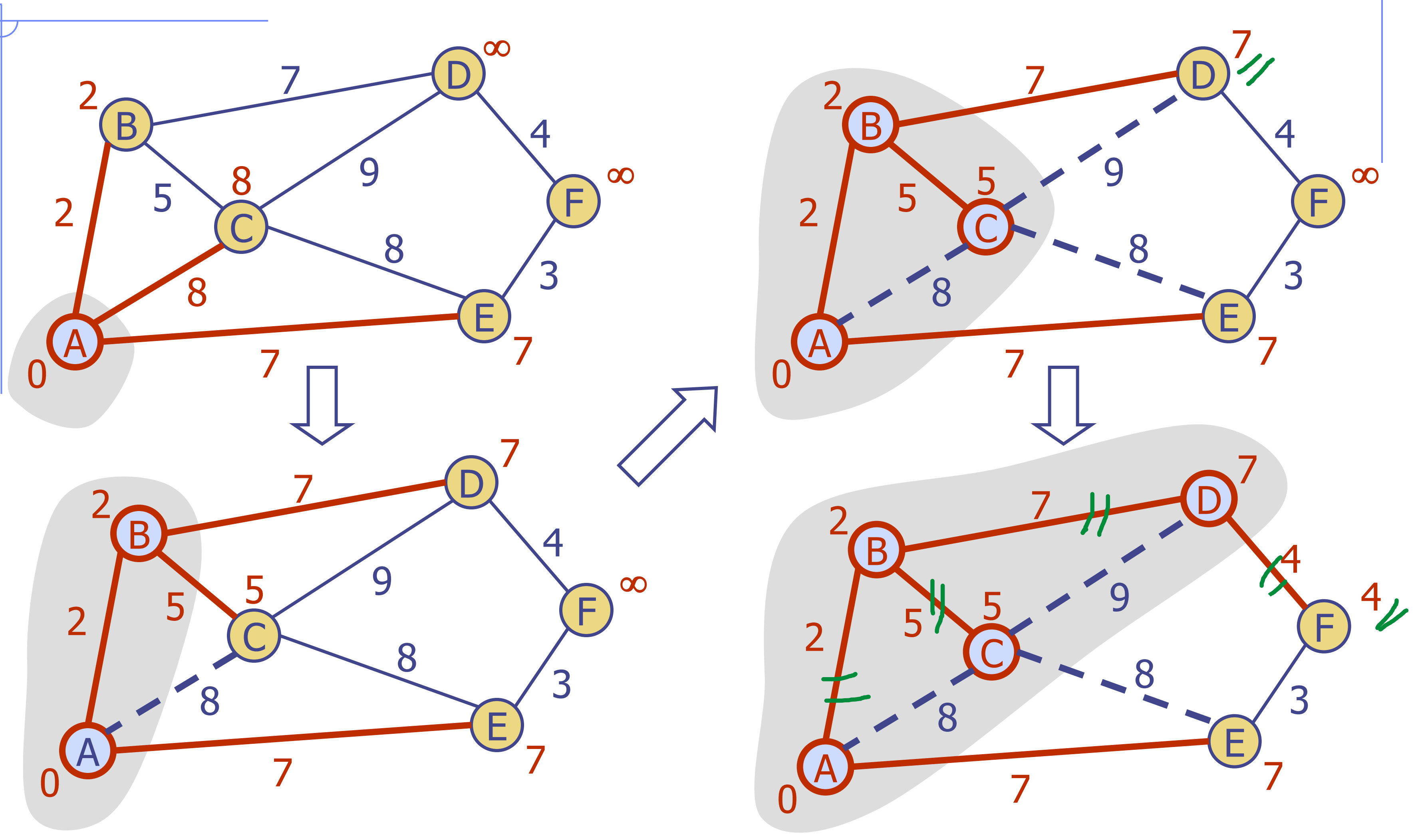


The minimum edge weight is 5. Update the parent edge C(B). Now C is in the Tree. Continue the process.

- - - - Can't pick CA, creates a cycle. The algorithm does not allow selected the edges. Always pick one from the Tree and other outside tree with minimum distance.

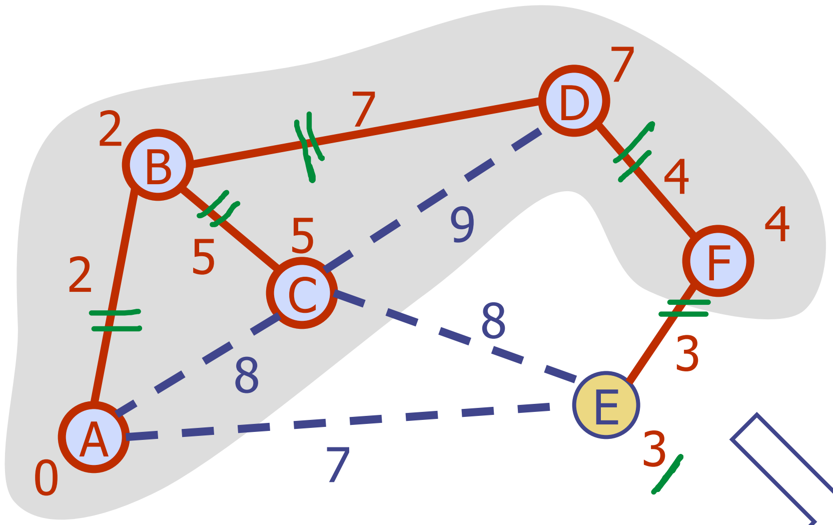
Example

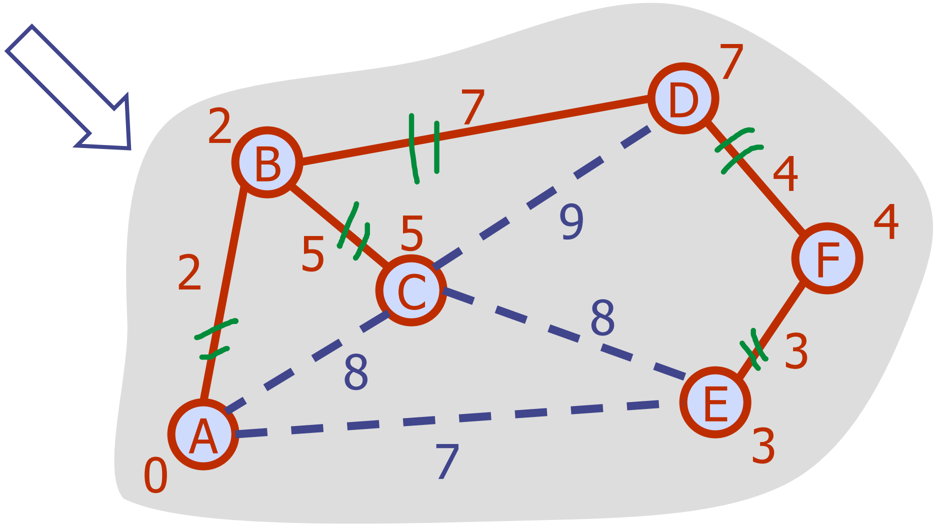
BD and AE has the same weight and minimum distance(can choose anyone), CD and CE are higher distance. D is picked and added in the T and update its parent edge(B). Update distance of D's adjacent.



Example (contd.)

Minimum distance DF is picked. Now F is in the T. Update F's adjacent distance E and the parent edge(D).





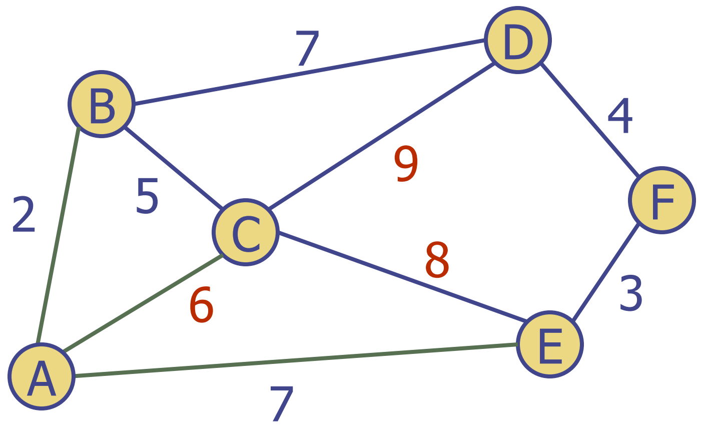
Minimum distance EF is picked. Now E is in the T. Update the parent edge(E). Now n-1 edges are in the tree.

Prim-Jarnik's Algorithm (cont.)

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - *insert*(*k*, *e*) returns a locator
  - *replaceKey*(*l*, *k*) changes the key of an item
- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Locator in priority queue
- Correction in red inspired by Bereket Chalew (May 2014)

```
Algorithm PrimJarnikMST(G)
    PQ ← new priority queue
    for all v ∈ G.vertices() do
        setParent(v, ∅)
        d ← ∞
        p ← PQ.insertItem(d, v)
        setLocator(v, p)
    s ← G.aVertex ()
    p ← getLocator (s)
    PQ.replaceKey(p, 0)
    while ! PQ.isEmpty() do
        u ← PQ.removeMin()
        setLocator(u, ∅)    {u is now in MST}
        for all e ∈ G.incidentEdges(u) do
            z ← G.opposite(u, e)
            r ← weight(e)
            p ← getLocator (z) {p=(weight, vertex)}
            if p ≠ ∅    {z not yet in MST}
                ∧ r < p.key() then
                setParent(z, e)
                PQ.replaceKey(p, r)
```

Cycle Property



By the Cycle Property, AC, CD, and CE cannot be in a MST.
What about AE and BD?

Let's run the algorithm to verify this.

Analysis

- Graph operations
  - Method incidentEdges is called once for each vertex
  - Recall that $\Sigma_v \deg(v) = 2m$
- Label operations
  - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
- The running time is $O(m \log n)$ since the graph is connected

Main Point

2. A defining feature of the Minimum Spanning Tree (and shortest path) greedy algorithms is that once a vertex becomes in-tree (or “inside the cloud”), the resulting subtree is optimal and nothing can change this state.

   Science of Consciousness*:* A defining feature of enlightenment is that once this state is reached, one’s consciousness is optimal and nothing can change this state.

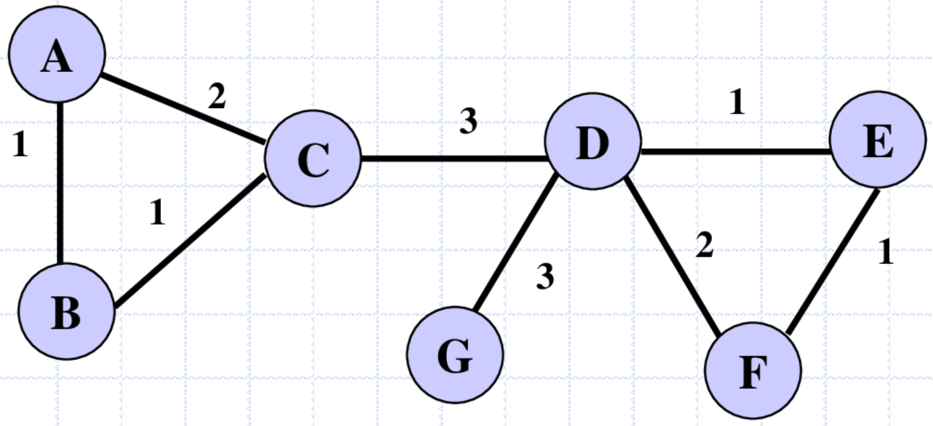
Kruskal's Algorithm

- First step is to sort all edges by weight.
- Second step involves creation of *clusters*
  - Every vertex is initially placed in a trivial *cluster* -- the starting cluster for a vertex v, denoted C(v), is simply {v}. A cluster consists of endpoints of a local minimum spanning tree.
  - When the next edge (u,v) is considered, C(u) and C(v) are compared -- if different, (u,v) is included as an edge in the final output tree, and C(u) and C(v) are merged.

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG

T = {...}

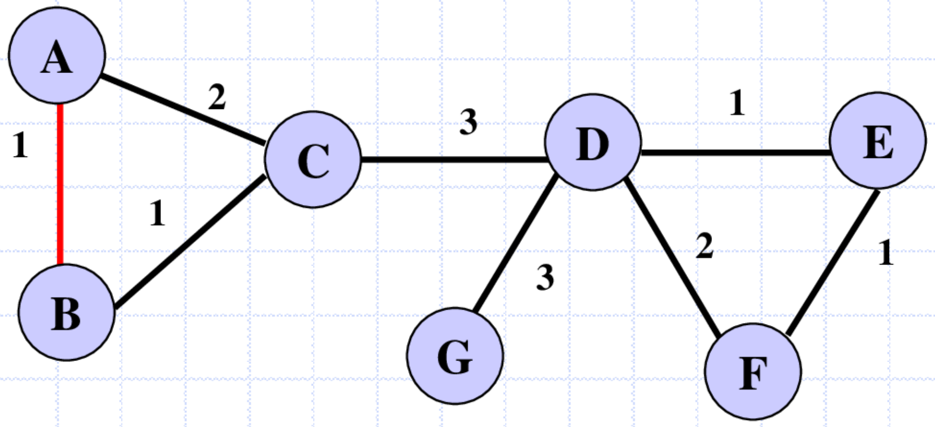


Cluster	*Evolving Values*
C(A)	{A}
C(B)	{B}
C(C)	{C}
C(D)	{D}
C(E)	{E}
C(F)	{F}
C(G)	{G}

Step 1:
Sort the edges and initialize the clusters

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, …}

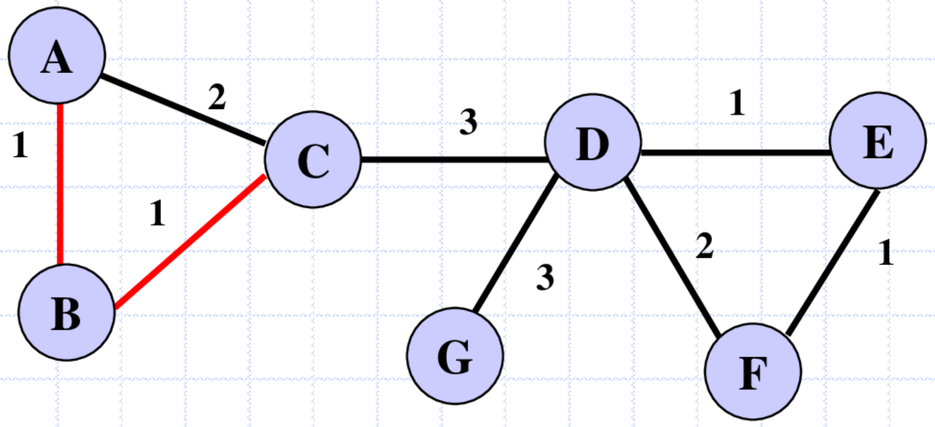


Cluster	*Evolving Values*
C(A)	{A, B}
C(B)	{A, B}
C(C)	{C}
C(D)	{D}
C(E)	{E}
C(F)	{F}
C(G)	{G}

Step 2:
C(A) ≠ C(B)
add AB to T, merge C(A) and C(B)

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, BC, …}

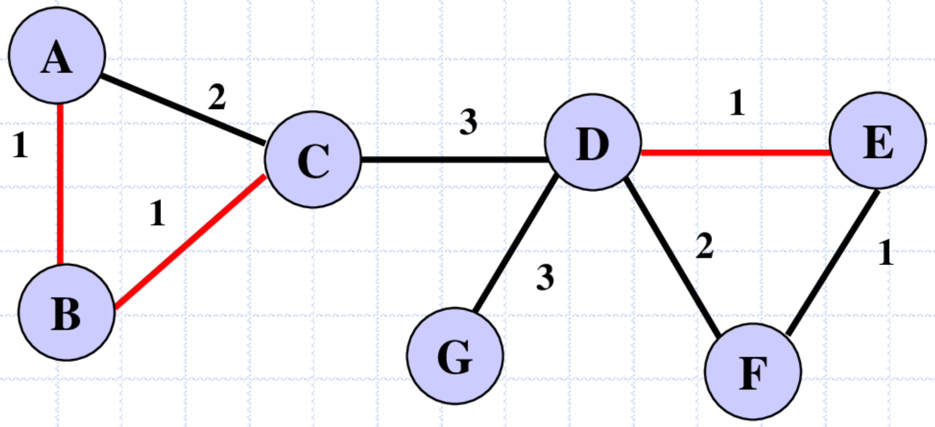


Cluster	*Evolving Values*
C(A)	{A, B, C}
C(B)	{A, B, C}
C(C)	{A, B, C}
C(D)	{D}
C(E)	{E}
C(F)	{F}
C(G)	{G}

Step 3:
C(B) ≠ C(C)
add BC to T, merge C(B) and C(C)

Worked Example

Sorted edges: <u>AB</u>, <u>BC</u>, <u>DE</u>, EF, AC, DF, CD, DG
T = {AB, BC, DE, …}

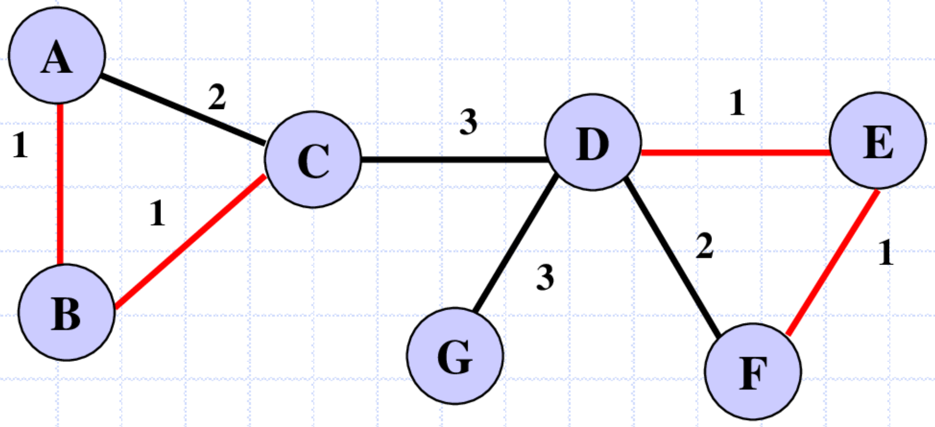


Cluster	*Evolving Values*
C(A)	{A, B, C}
C(B)	{A, B, C}
C(C)	{A, B, C}
C(D)	{D, E}
C(E)	{D, E}
C(F)	{F}
C(G)	{G}

Step 4:
C(D) ≠ C(E)
add DE to T, merge C(D) and C(E)

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, BC, DE, EF, …}

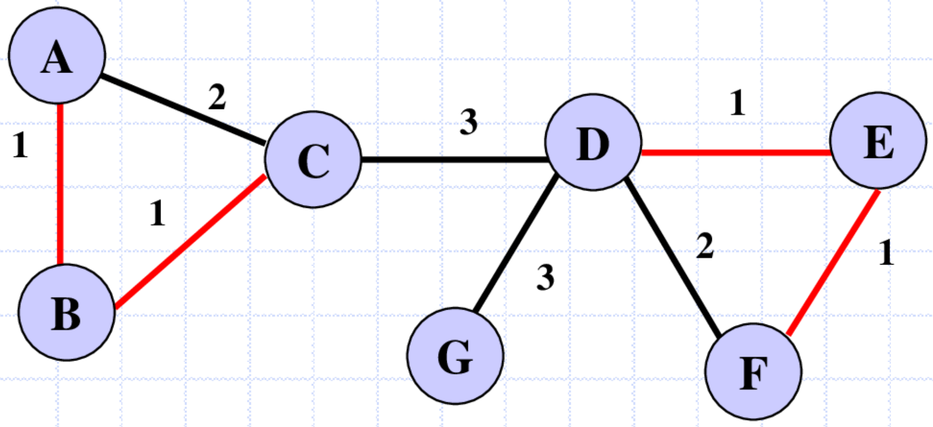


Cluster	*Evolving Values*
C(A)	{A, B, C}
C(B)	{A, B, C}
C(C)	{A, B, C}
C(D)	{D, E, F}
C(E)	{D, E, F}
C(F)	{D, E, F}
C(G)	{G}

Step 5:
C(E) ≠ C(F)
add EF to T, merge C(E) and C(F)

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, BC, DE, EF, ...}

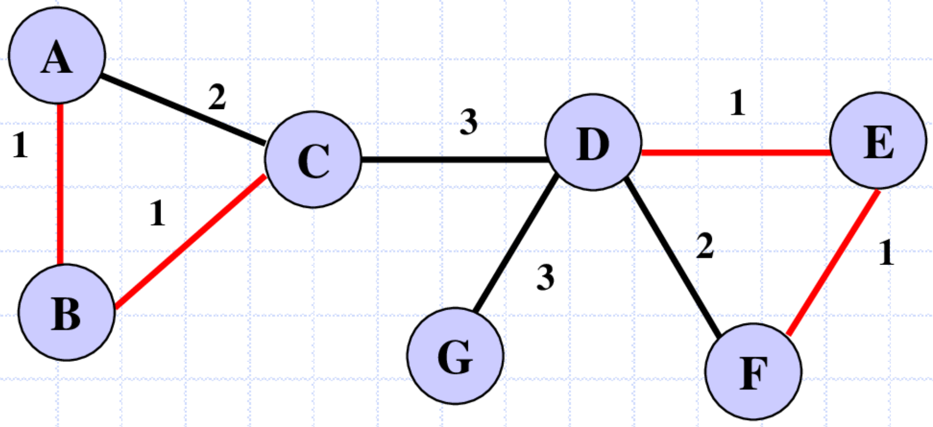


Cluster	*Evolving Values*
C(A)	{A, B, C}
C(B)	{A, B, C}
C(C)	{A, B, C}
C(D)	{D, E, F}
C(E)	{D, E, F}
C(F)	{D, E, F}
C(G)	{G}

Step 6:
C(A) = C(C) , discard AC

Worked Example

Sorted edges: <u>AB</u>, <u>BC</u>, <u>DE</u>, <u>EF</u>, <u>AC</u>, <u>DF</u>, CD, DG
T = {AB, BC, DE, EF, ...}

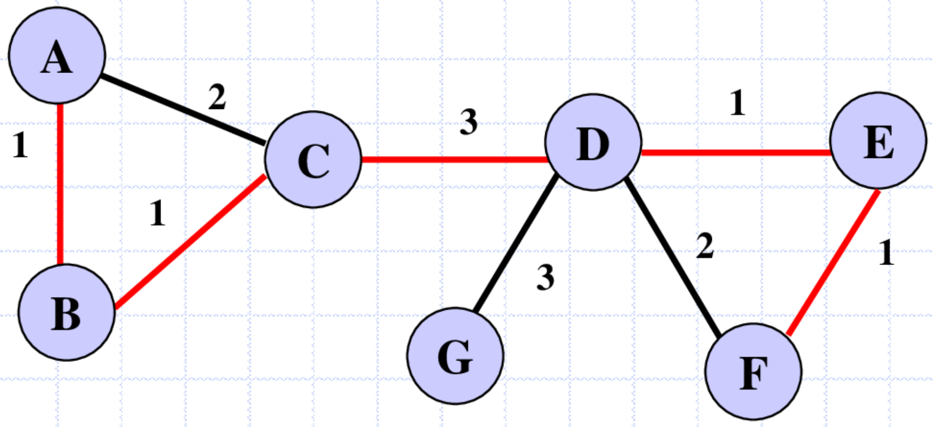


Cluster	*Evolving Values*
C(A)	{A, B, C}
C(B)	{A, B, C}
C(C)	{A, B, C}
C(D)	{D, E, F}
C(E)	{D, E, F}
C(F)	{D, E, F}
C(G)	{G}

Step 7:
C(D) = C(F) , discard DF

Worked Example

Sorted edges: <u>AB</u>, <u>BC</u>, <u>DE</u>, <u>EF</u>, <u>AC</u>, <u>DF</u>, <u>CD</u>, DG
T = {AB, BC, DE, EF, CD, …}

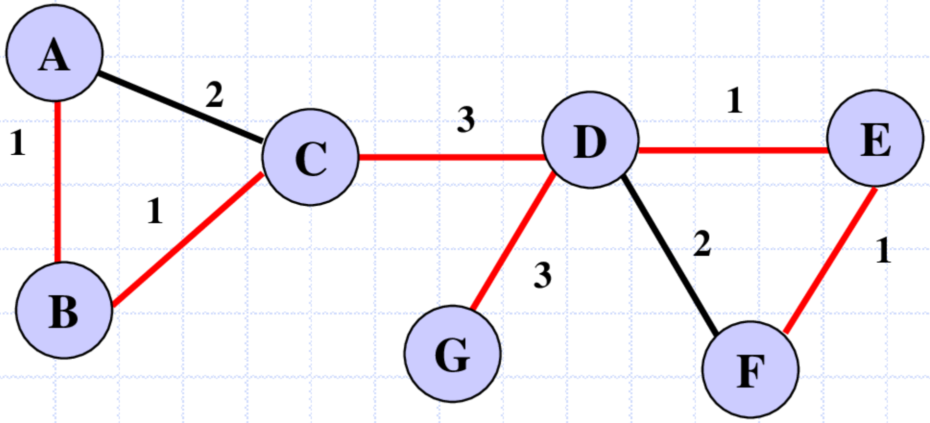


Cluster	*Evolving Values*
C(A)	{A, B, C, D, E, F}
C(B)	{A, B, C, D, E, F}
C(C)	{A, B, C, D, E, F}
C(D)	{A, B, C, D, E, F}
C(E)	{A, B, C, D, E, F}
C(F)	{A, B, C, D, E, F}
C(G)	{G}

Step 7:
C(C) ≠ C(D)
add CD to T, merge C(C) and C(D)

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, BC, DE, EF, CD, DG, …}

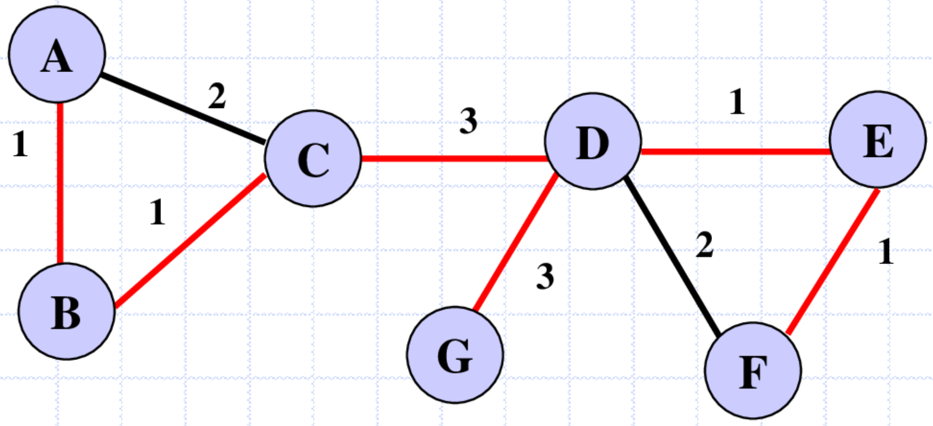


Cluster	*Evolving Values*
C(A)	{A, B, C, D, E, F, G}
C(B)	{A, B, C, D, E, F, G}
C(C)	{A, B, C, D, E, F, G}
C(D)	{A, B, C, D, E, F, G}
C(E)	{A, B, C, D, E, F, G}
C(F)	{A, B, C, D, E, F, G}
C(G)	{A, B, C, D, E, F, G}

Step 8:
C(D) ≠ C(G)
add DG to T, merge C(D) and C(G)

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, BC, DE, EF, CD, DG}



Cluster	*Evolving Values*
C(A)	{A, B, C, D, E, F, G}
C(B)	{A, B, C, D, E, F, G}
C(C)	{A, B, C, D, E, F, G}
C(D)	{A, B, C, D, E, F, G}
C(E)	{A, B, C, D, E, F, G}
C(F)	{A, B, C, D, E, F, G}
C(G)	{A, B, C, D, E, F, G}

Now we have n-1 = 6 edges in T, the algorithm stops.

Kruskal's Algorithm (1956) (High Level)

- A priority queue stores the edges outside the cloud
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - We are left with one cloud that encompasses the MST
  - A tree *T* which is our MST

```
Algorithm KruskalMST(G)
    for each vertex $v$ in $G$ do
        define a $Cloud(v) \leftarrow \{v\}$
    $Q \leftarrow$ new heap-based priority queue.
    for all  $e \in G.edges()$
        $Q.insert(weight(e), e)$
    $T \leftarrow \varnothing$
    while $T$ has fewer than $n$-1 edges  do
        $e \leftarrow Q.removeMin()$
        $(u, v) \leftarrow G.endVertices(e)$
        if $Cloud(v) \neq Cloud(u)$ then
            Add edge $e$ to $T$
            Merge $Cloud(v)$ and $Cloud(u)$
    return $T$
```