Graphs: Minimum Spanning Trees (Kruskal's & Prim's algorithm)

Bijendra Nath Jain

bnjain@iiitd.ac.in

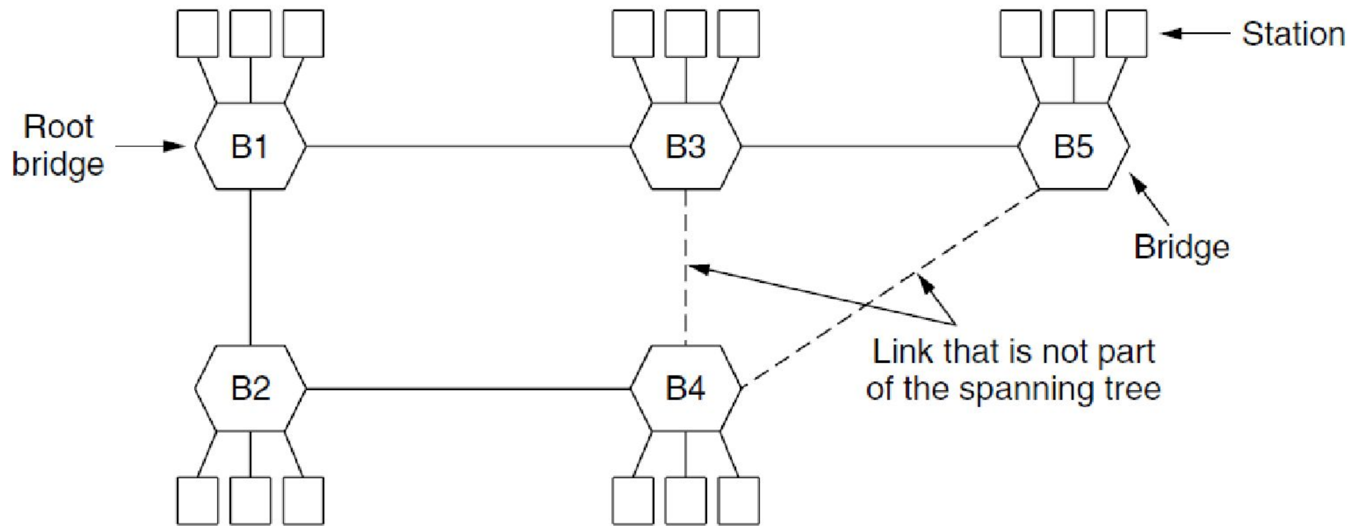
Some of the slides are from https://courses.cs.washington.edu/courses/cse373/22sp/, or from those prepared by Si Dong, M.T. Goodrich and R. Tamassia (reference??)

Outline

- Graphs:
  - Undirected graphs
  - Directed graphs
  - (Directed) acyclic graphs (or DAGs)
  - Sparse graphs
  - Weighted graphs
- Graph applications
- Representation of graphs:
  - Adjacency matrix
  - Linked lists
- Algorithms:
  - Traversal algorithms:
    - BFS
    - DFS
  - Topological sort
  - Minimum spanning trees
  - Dijkstra's Shortest path
    - One-to-one
    - One-to-many
    - Many-to-many

Spanning tree

- Spanning Tree applied to network of routers or bridges



Spanning tree

- Spanning Tree problem:

  Consider:

  a connected, undirected graph G = (V, E)

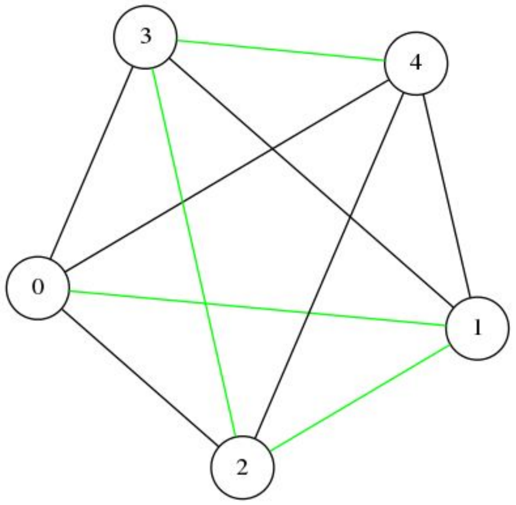
  Objective:

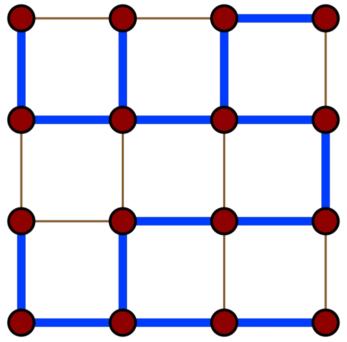
  compute a spanning tree G1 = (V, T), where

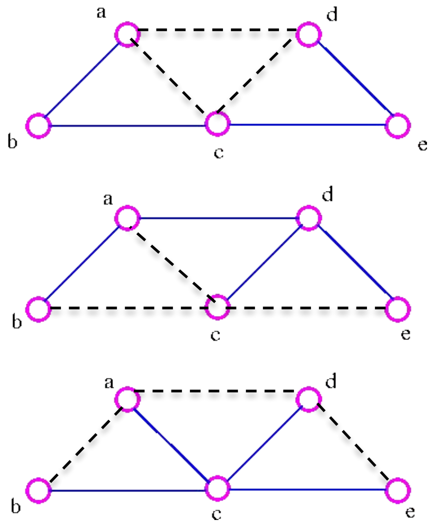
  1. G1 is a sub-graph of G, i.e. T is subset of E, while all vertices in V are in G1
  2. G1 is connected -- all vertices in V are reachable from every other vertex in V but using edges in T only,
  3. there are no cycles in G1

Spanning tree

- Spanning Tree: examples:



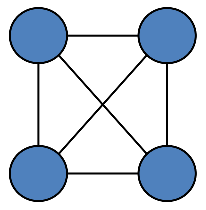




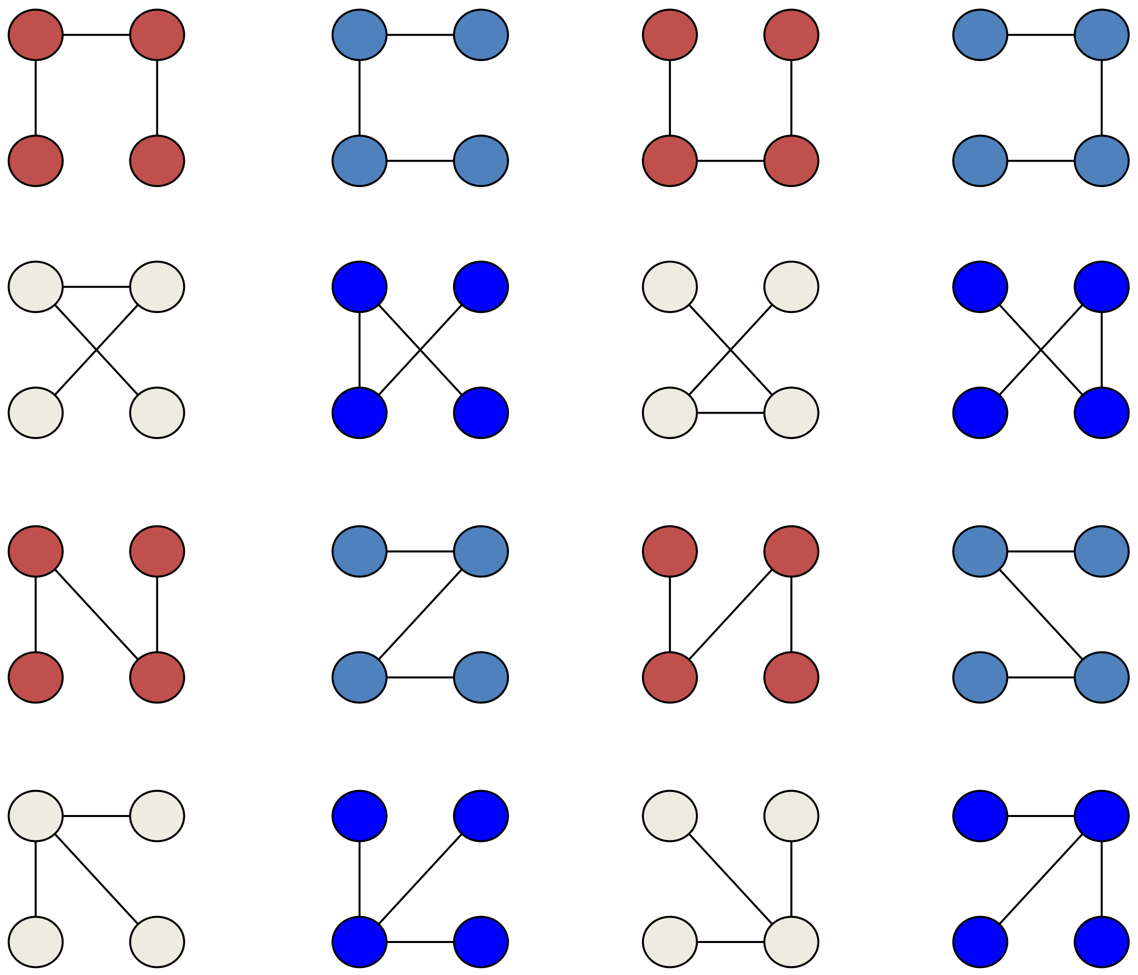
Spanning tree

- Spanning Tree: examples:

Complete Graph

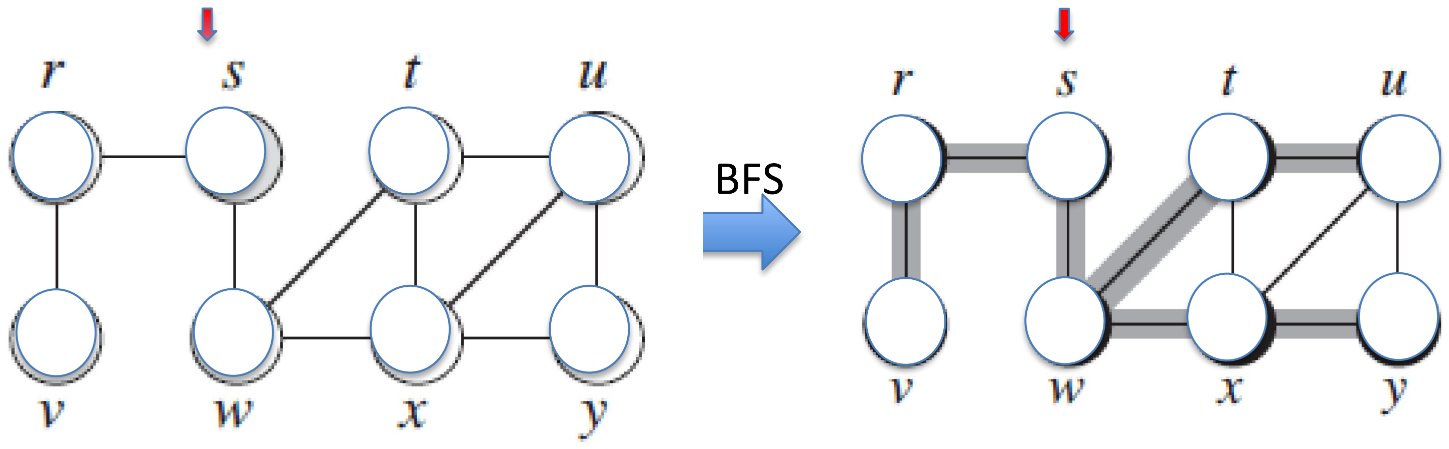


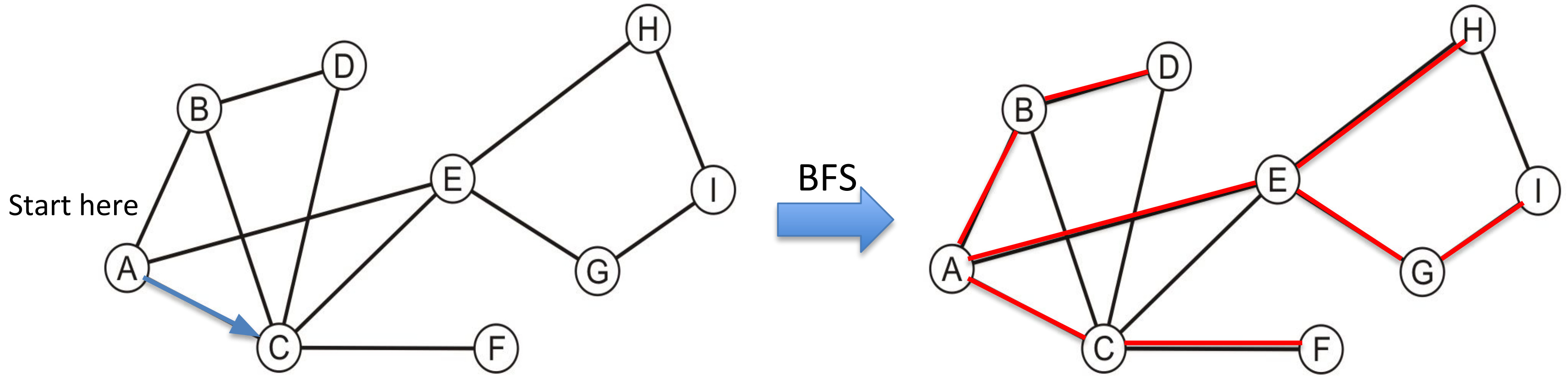
All possible spanning trees



Spanning tree

- Spanning Tree: Use BFS to compute a spanning tree:



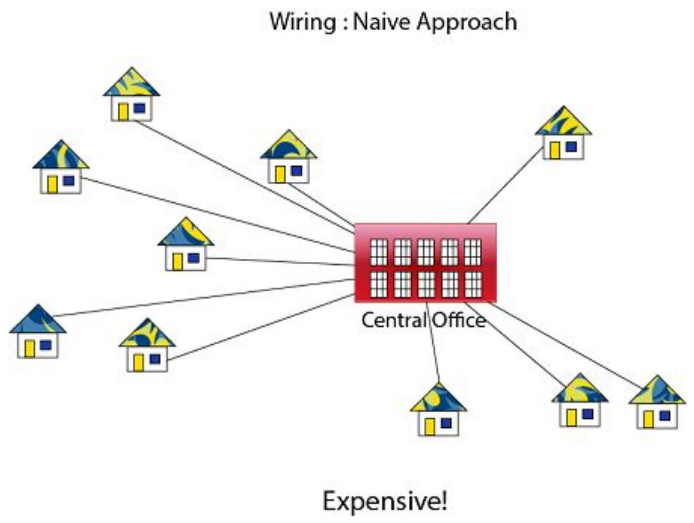


Minimum spanning tree

- Minimum spanning trees == minimum-weight spanning trees
- Applications:
  - routing wires on printed circuit boards
  - Planning sewer pipe layout
  - Road network planning
  - metro train network
  - telephone lines to a set of houses

Minimum spanning tree

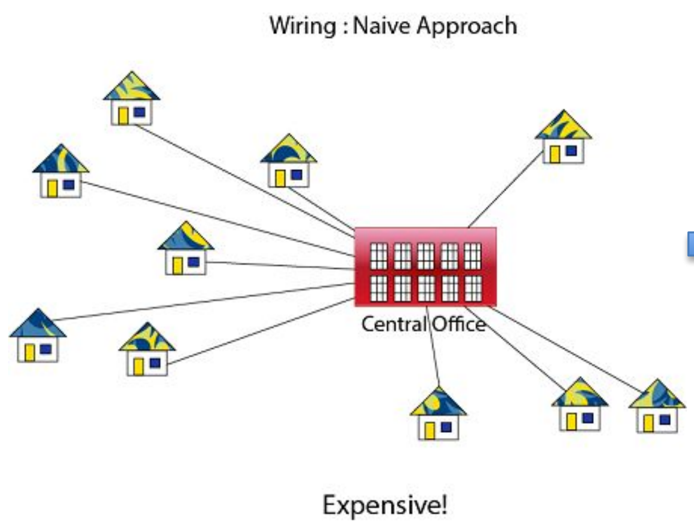
- MST, applied to cabling landline phones to homes

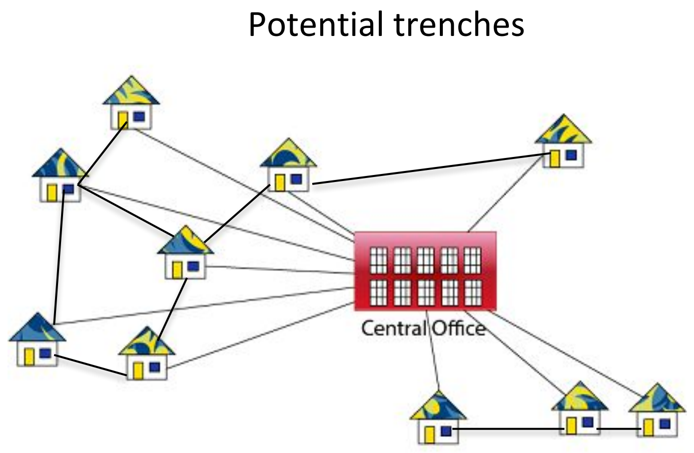


Courtesy www.javatpoint.com

Minimum spanning tree

- MST, applied to cabling landline phones to homes

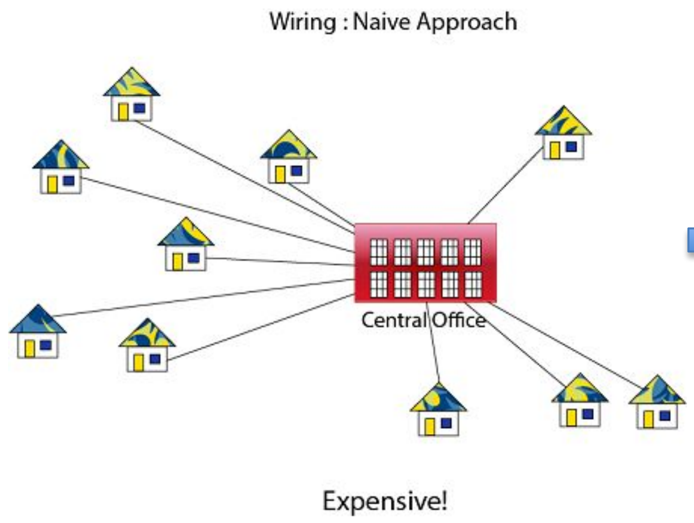


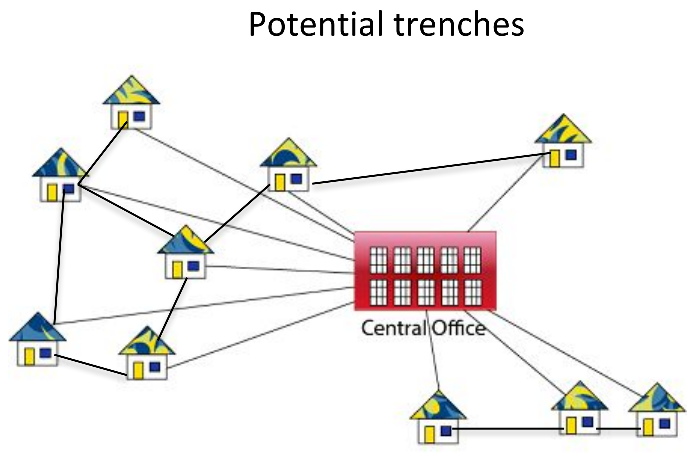


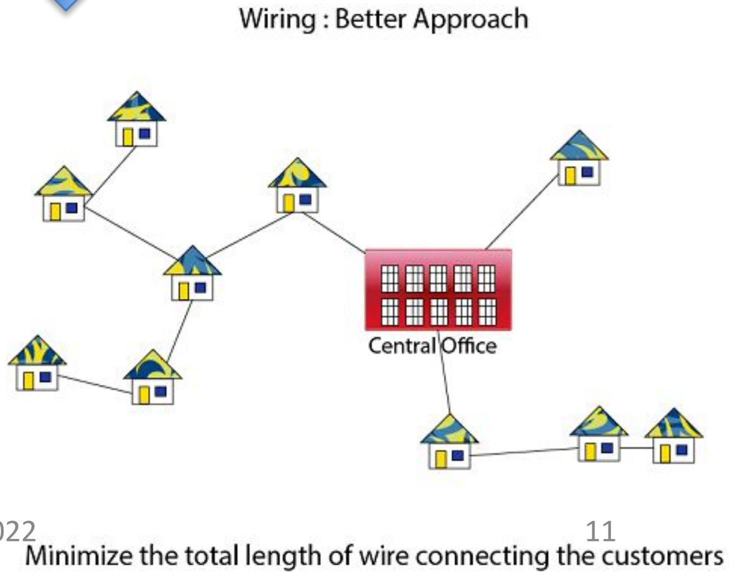
Courtesy www.javatpoint.com

Minimum spanning tree

- MST, applied to cabling landline phones to homes







Courtesy www.javatpoint.com

Minimum spanning tree

- Minimum-weight Spanning Tree problem:

  Consider:

  a connected, undirected graph G = (V, E), with weights w(u, v) associated with each edge, (u, v) in E

  Objective:

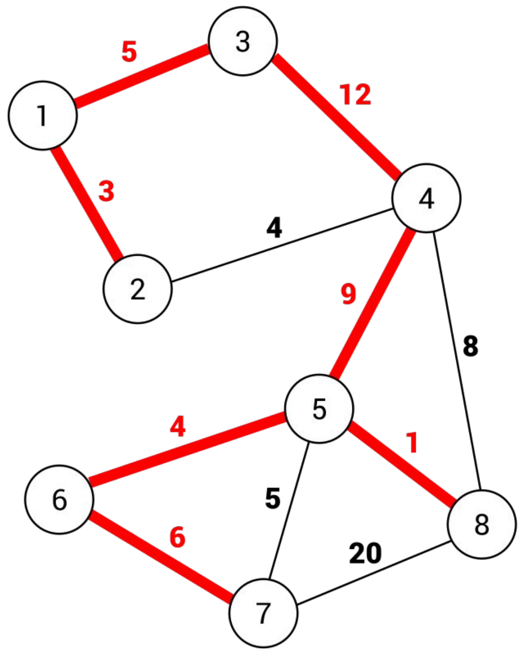
  compute a spanning tree G1 = (V, T), with minimize sum of weight of edges in T, viz.

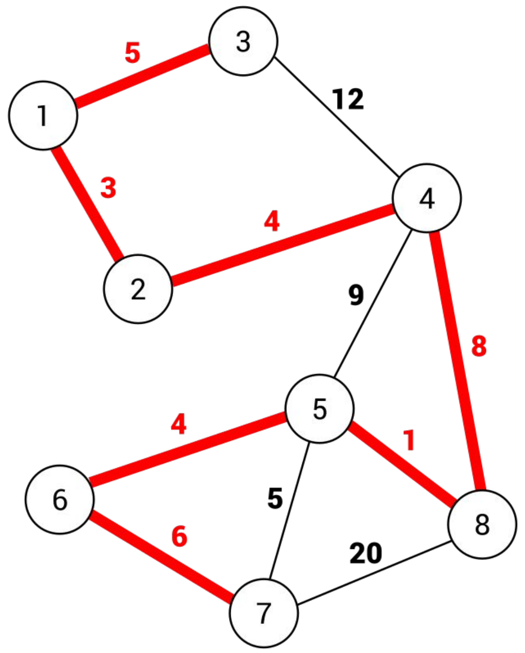
$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

(Note: G1 = (V, T) is a spanning tree of G, and $T \subseteq E$)

Minimum spanning tree

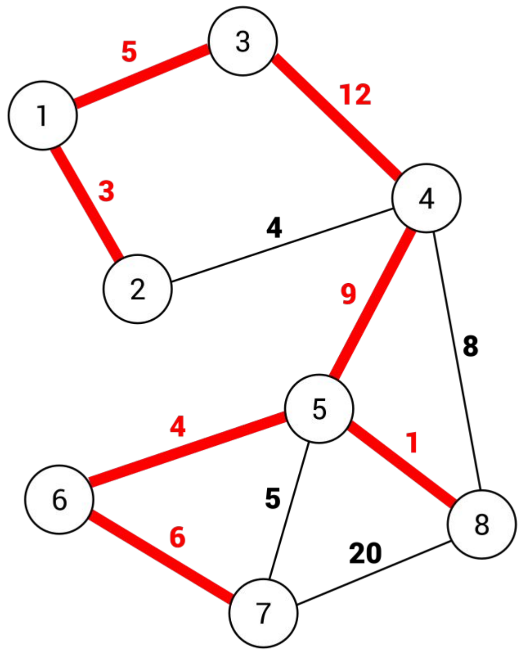
- Minimum spanning Tree problem:

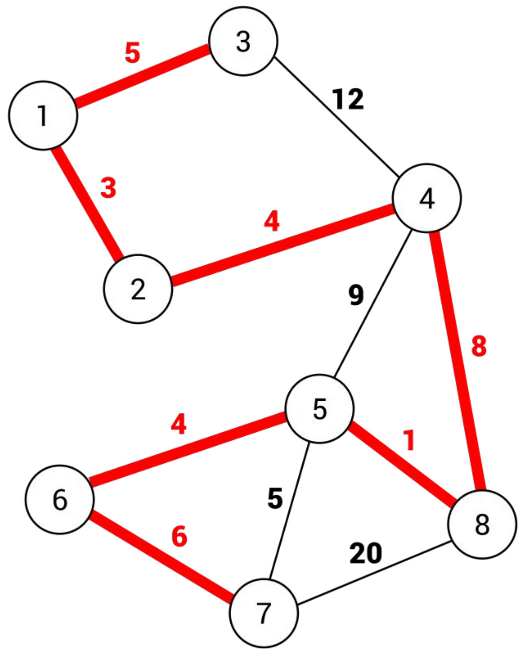




Minimum spanning tree

- Minimum spanning Tree problem:

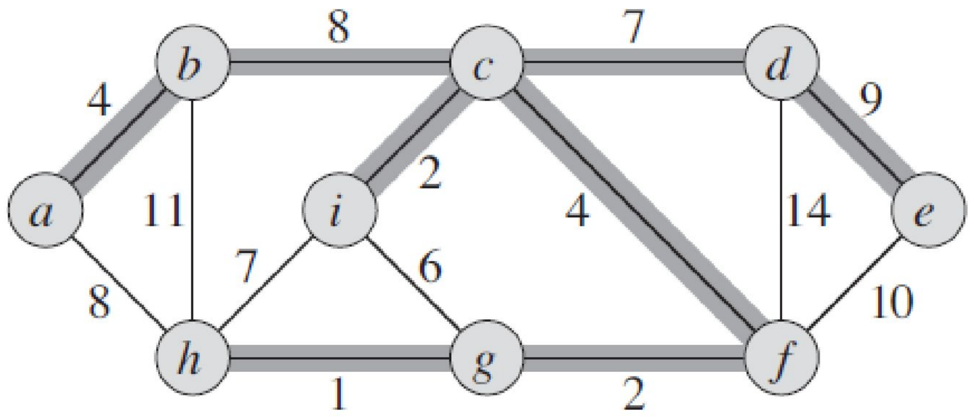




- What is the w(T) in each case?
- Can we still do better? Possibly.

Minimum spanning tree

- Minimum spanning Tree problem:



- Minimum weight = 37
- Not a unique minimum spanning tree – replace edge (b, c) with (a, h)

Minimum spanning tree

- Two algorithms:
  - Kruskal’s algorithm
  - Prim’s algorithm
- Time complexity is O(E logV)
  - May be improved – but that is for later

Kruskal’s minimum spanning tree

Kruskal’s algorithm:

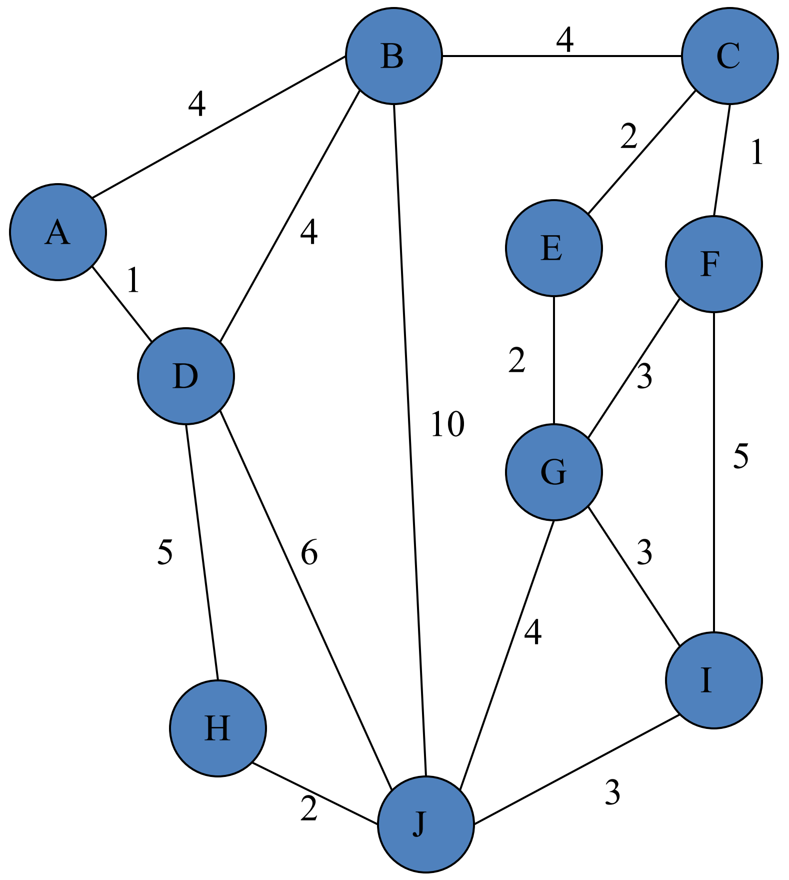
- Start with all vertices but no edges in the spanning tree
- Repeatedly add the cheapest edge that does not create a cycle

Prim’s algorithm:

- Start with any one vertex in the spanning tree
- Repeatedly add the cheapest edge, and the NEW node it leads to
  - the new vertex is not in the spanning tree

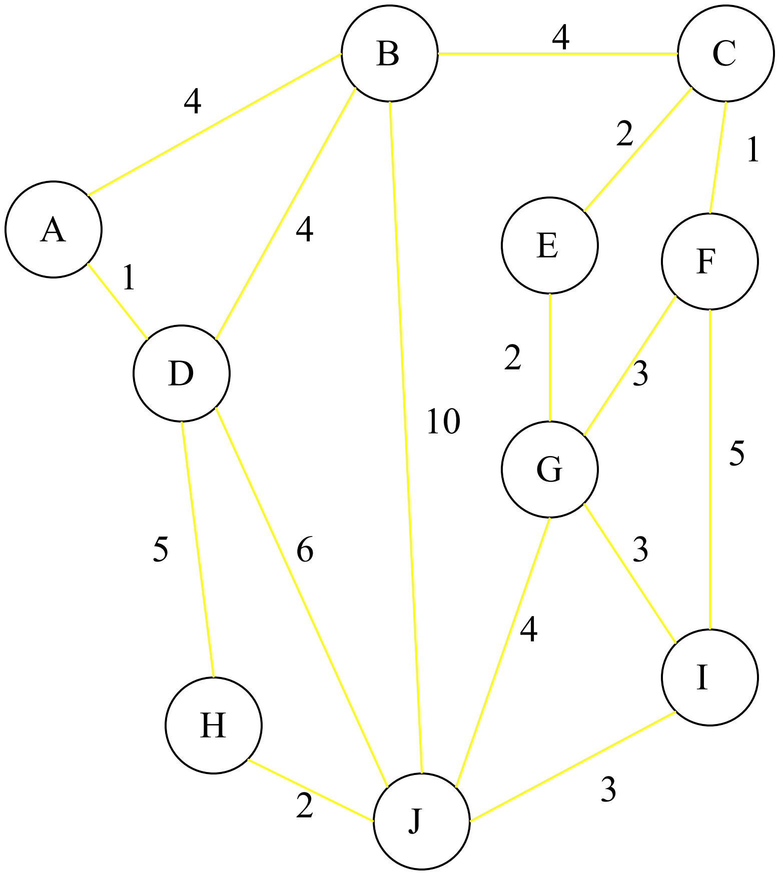
Kruskal’s minimum spanning tree

Graph, G

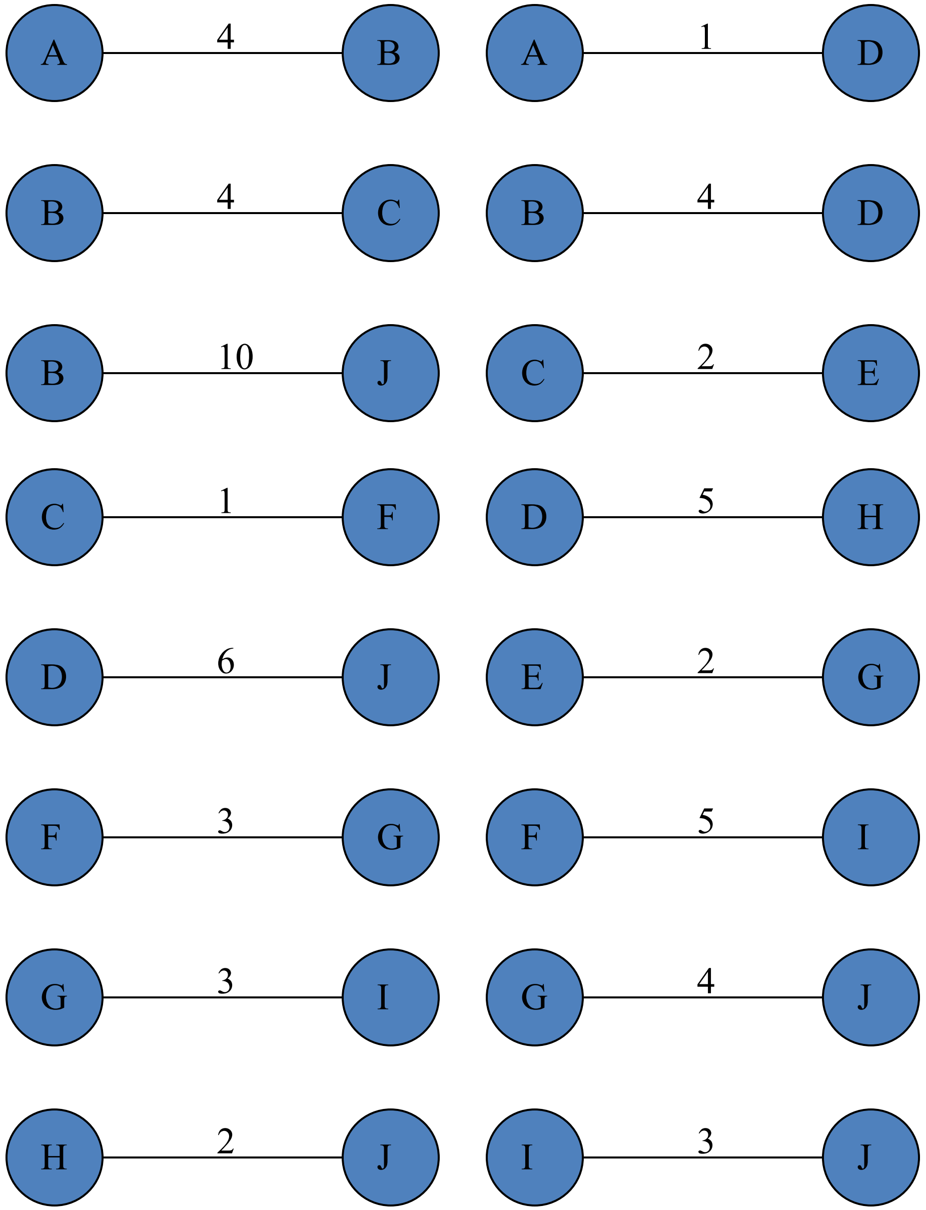


Kruskal's minimum spanning tree

Current state of G1 = (V, T),
Initially T = []

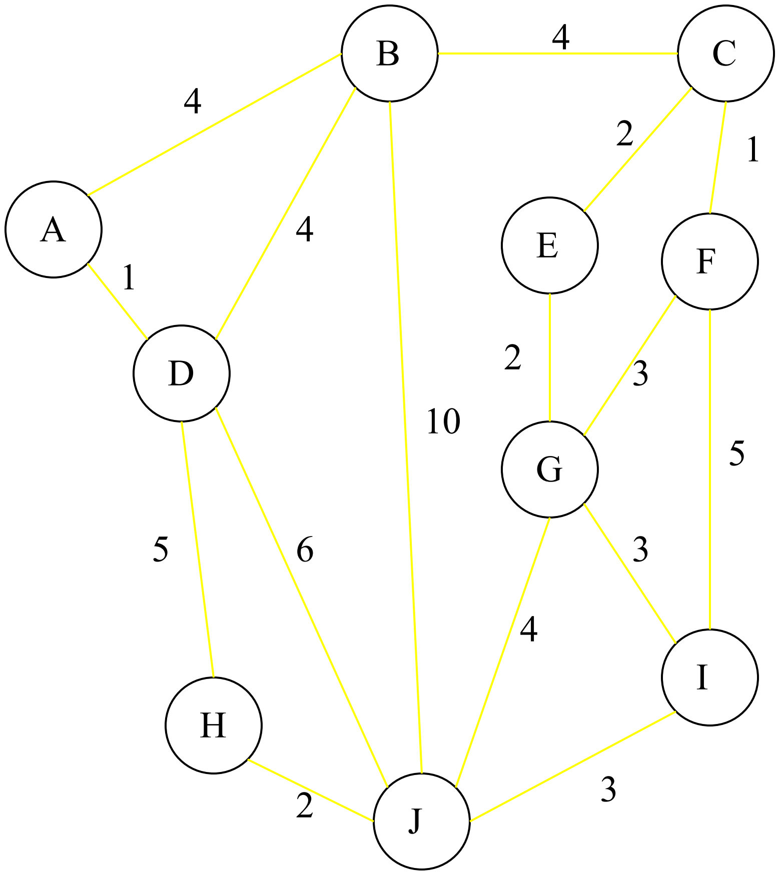


Unsorted list of edges (by weight)

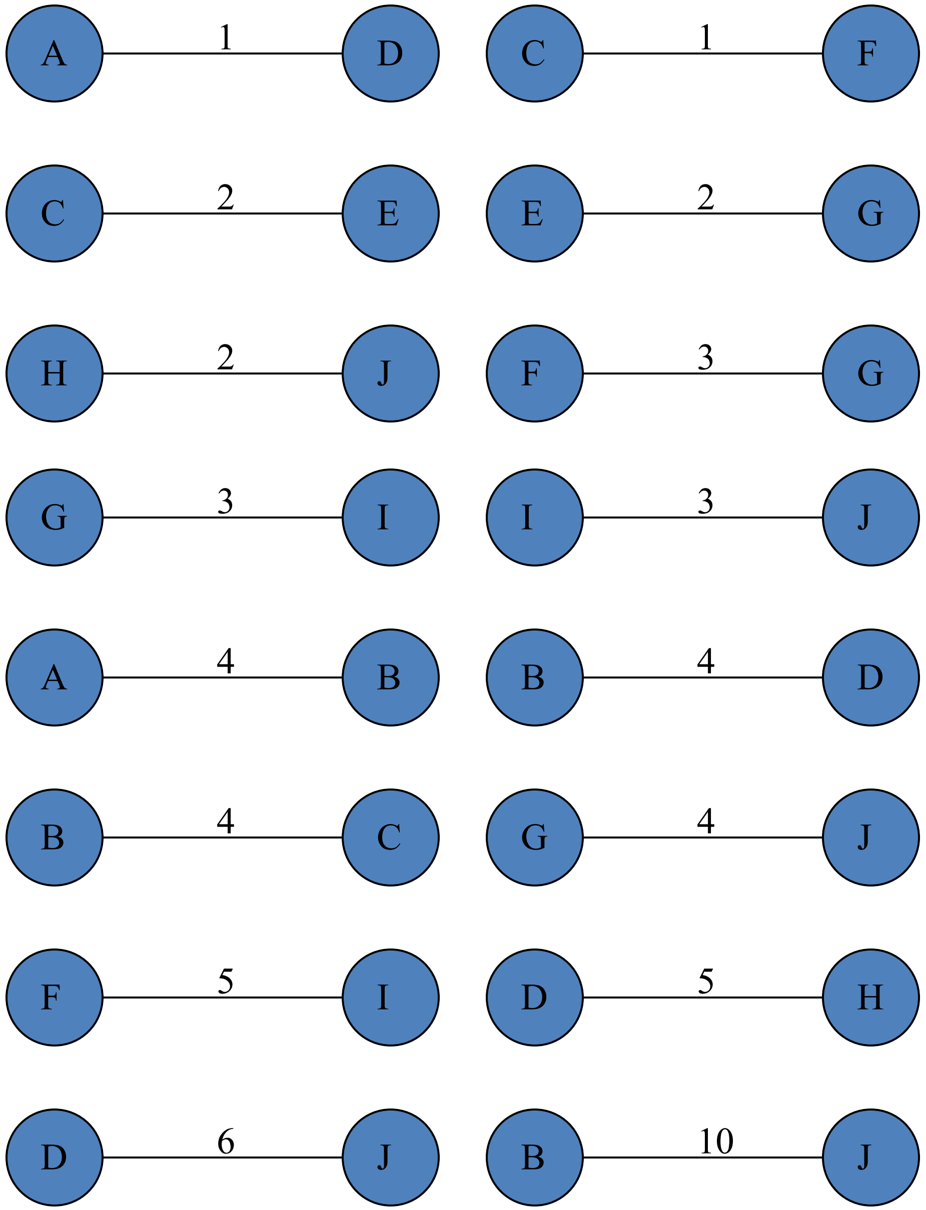


Kruskal's minimum spanning tree

Current state of G1 = (V, T),
Initially T = []

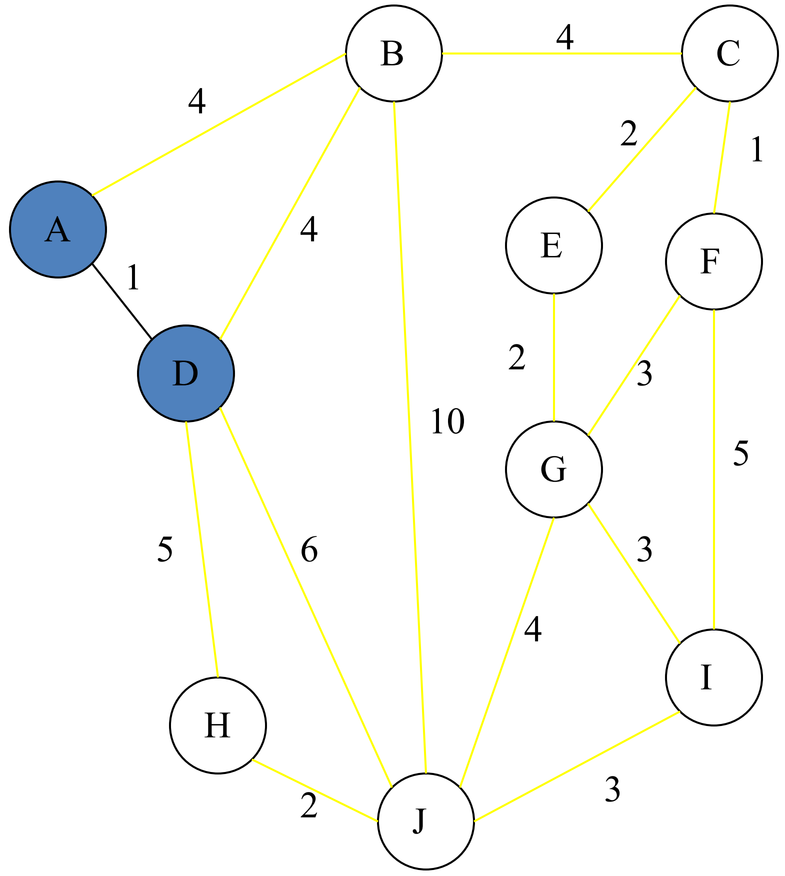


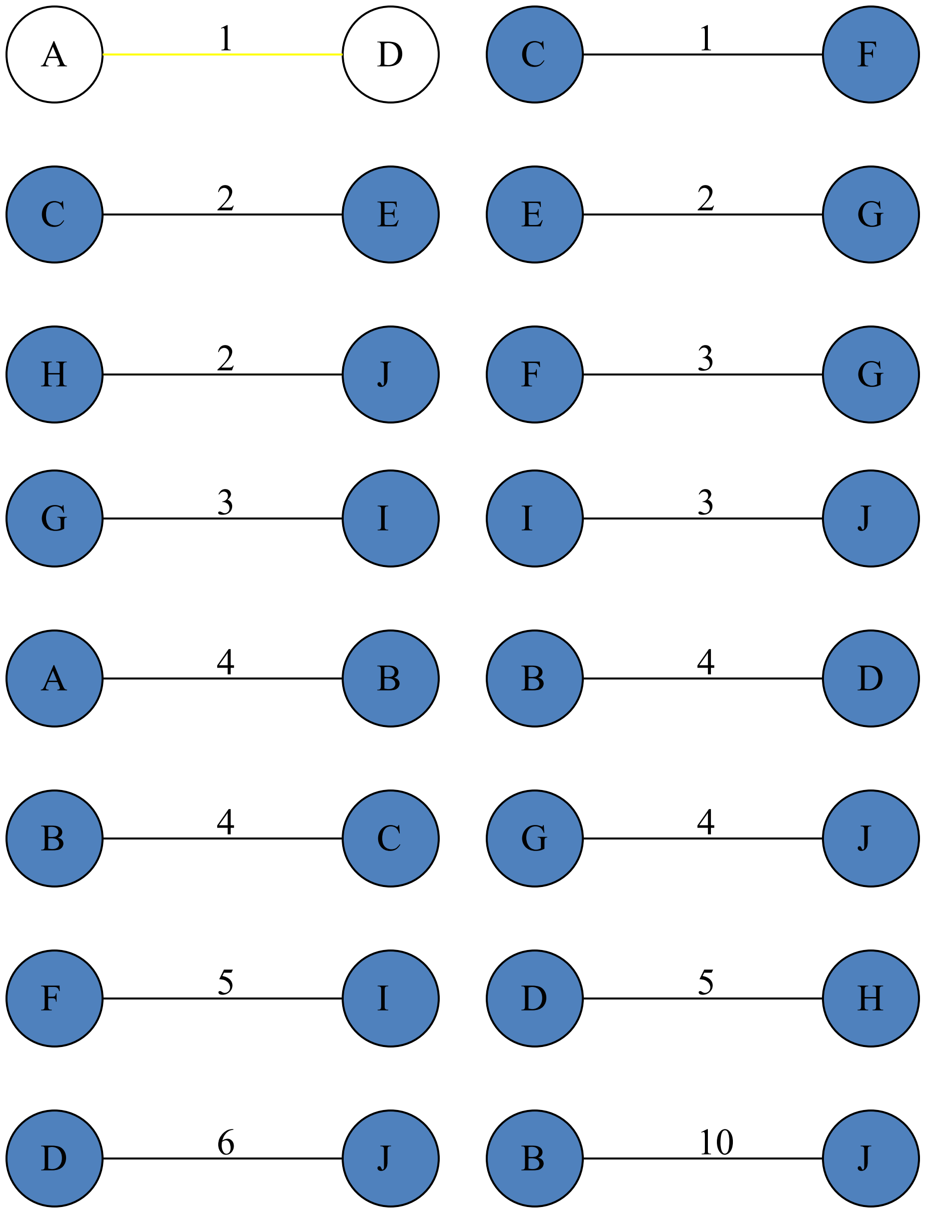
Sorted list of edges (by weight)



Kruskal's minimum spanning tree

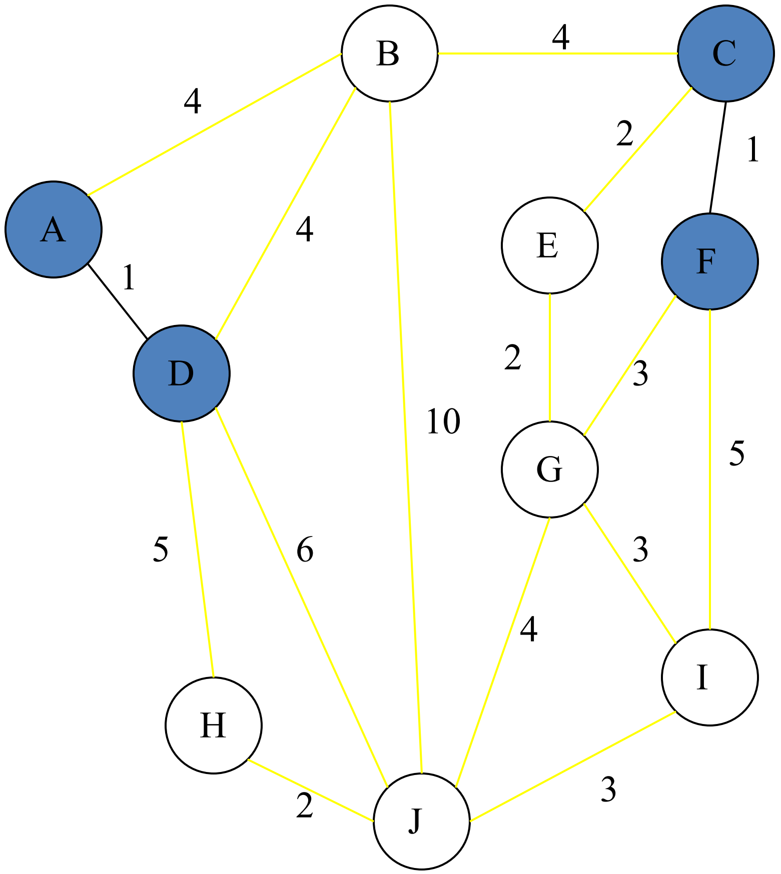
Current state of G1 = (V, T),
Add (a, d) to T

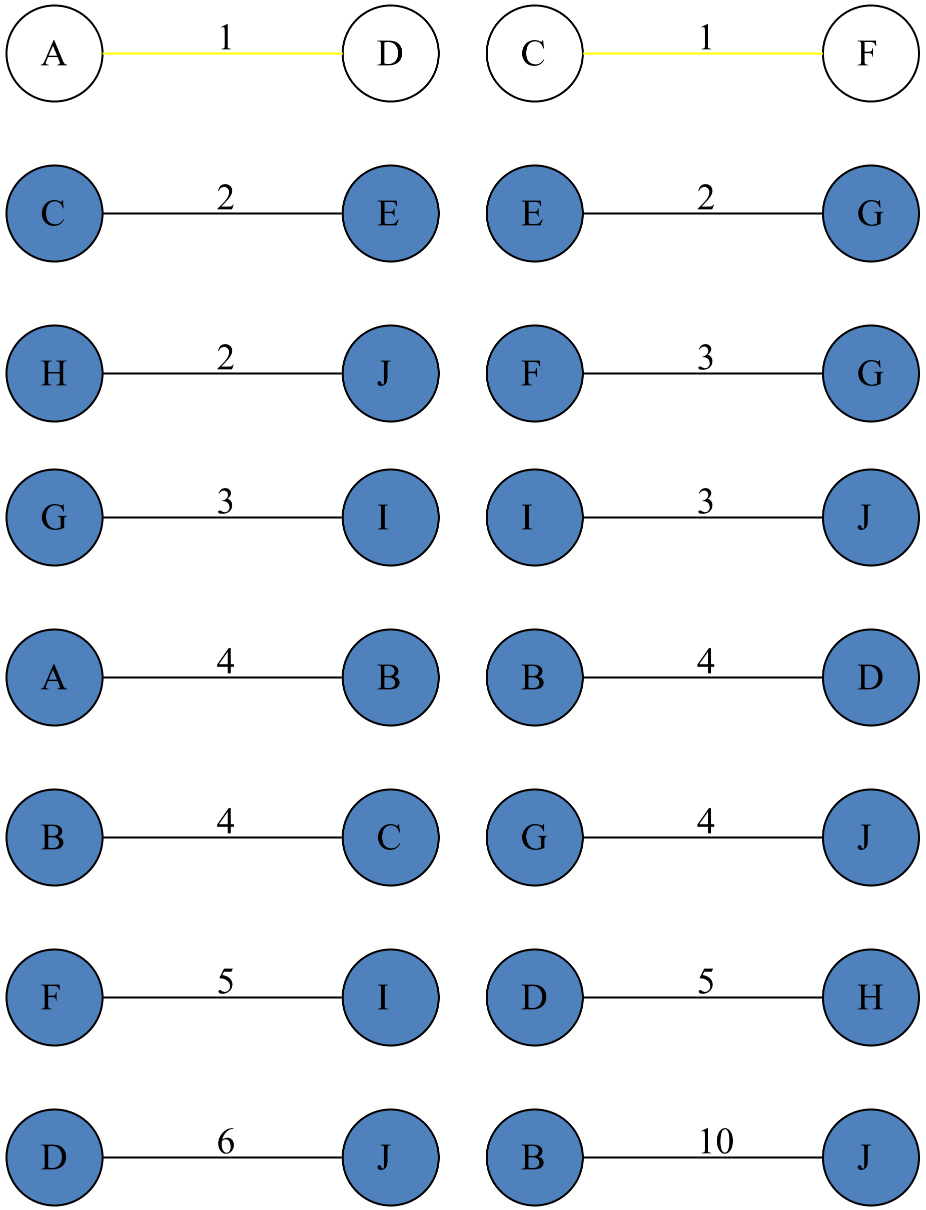




Kruskal's minimum spanning tree

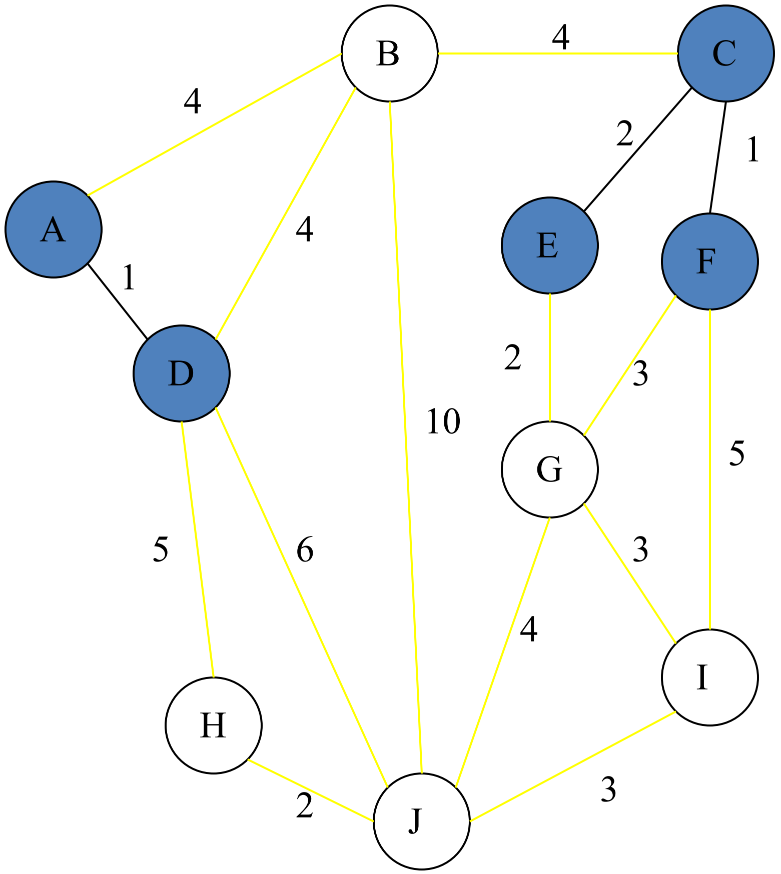
Current state of G1 = (V, T),
Add (c, f) to T

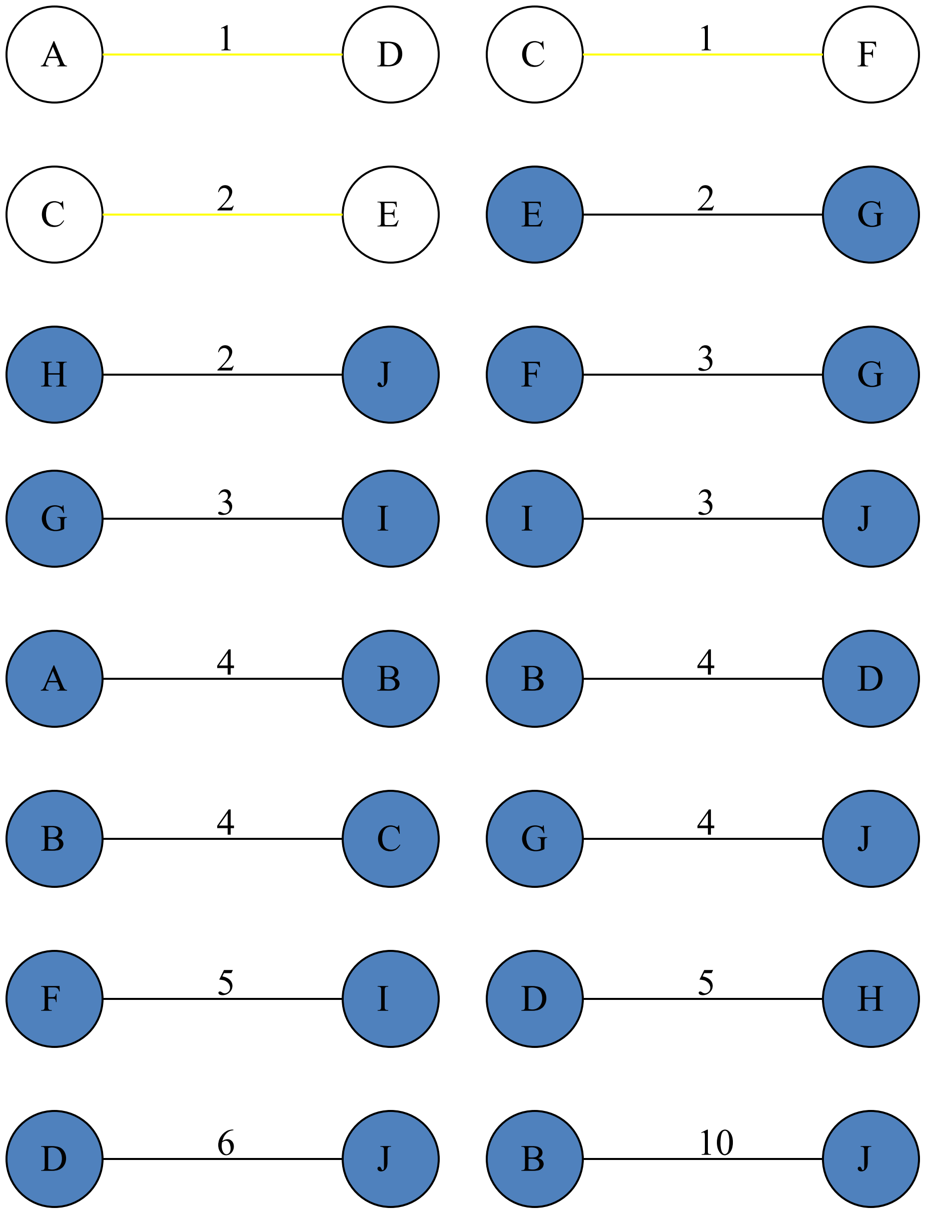




Kruskal's minimum spanning tree

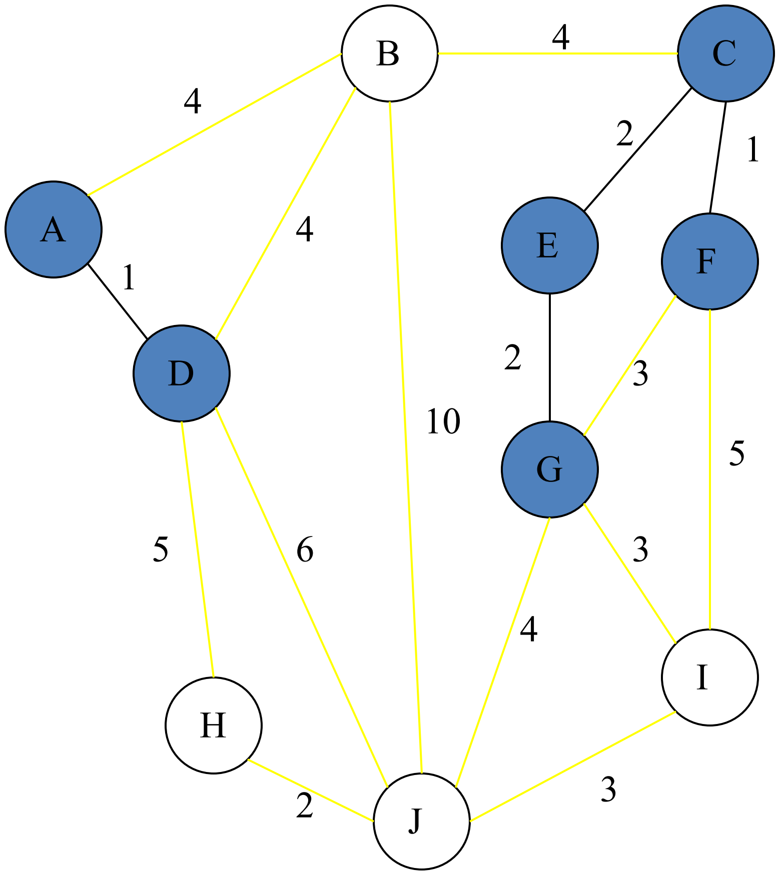
Current state of G1 = (V, T),
Add (c, e) to T

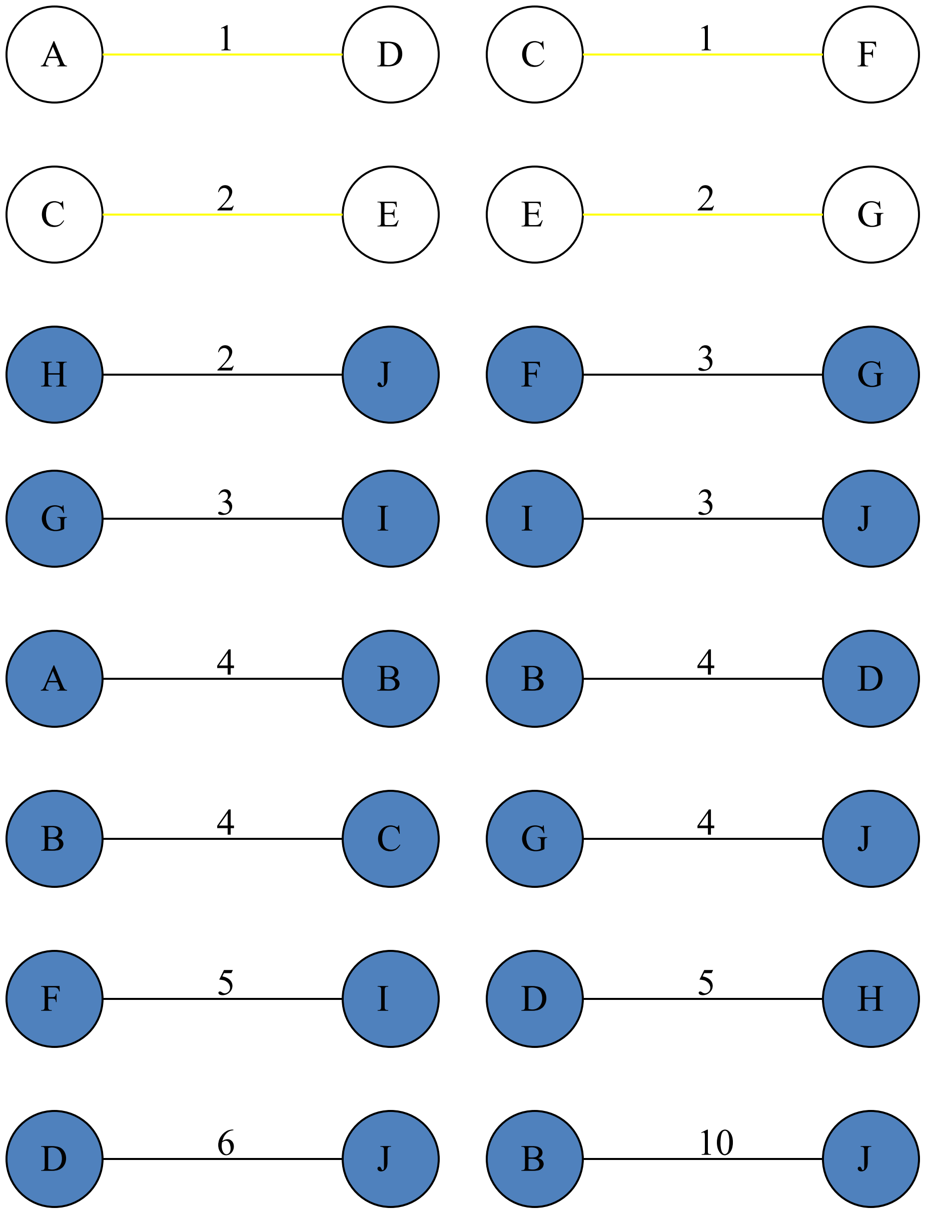




Kruskal’s minimum spanning tree

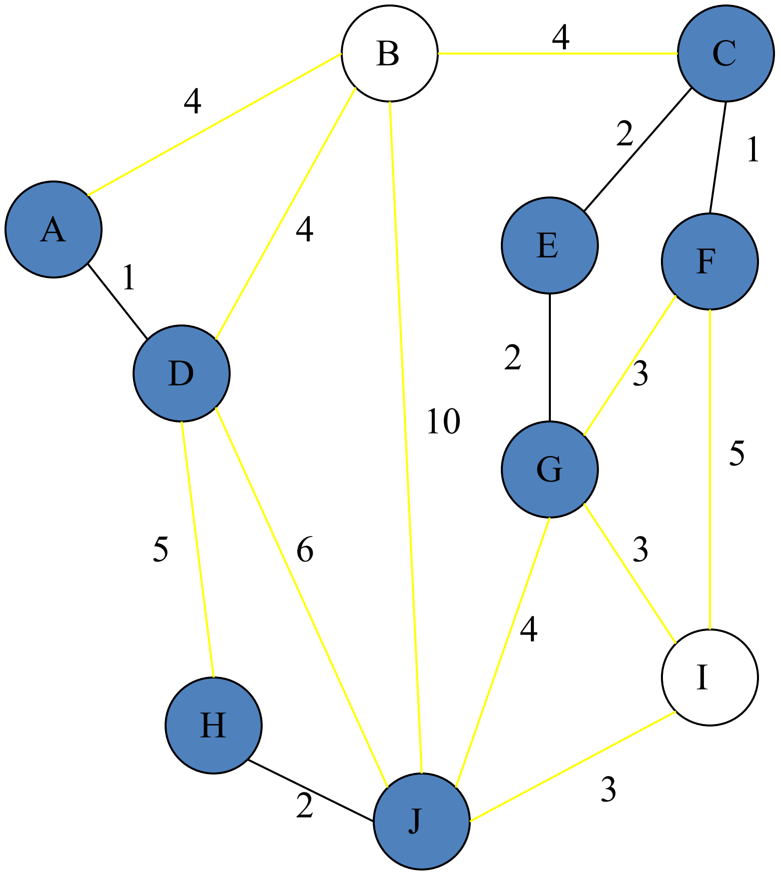
Current state of G1 = (V, T),
Add (e, g) to T

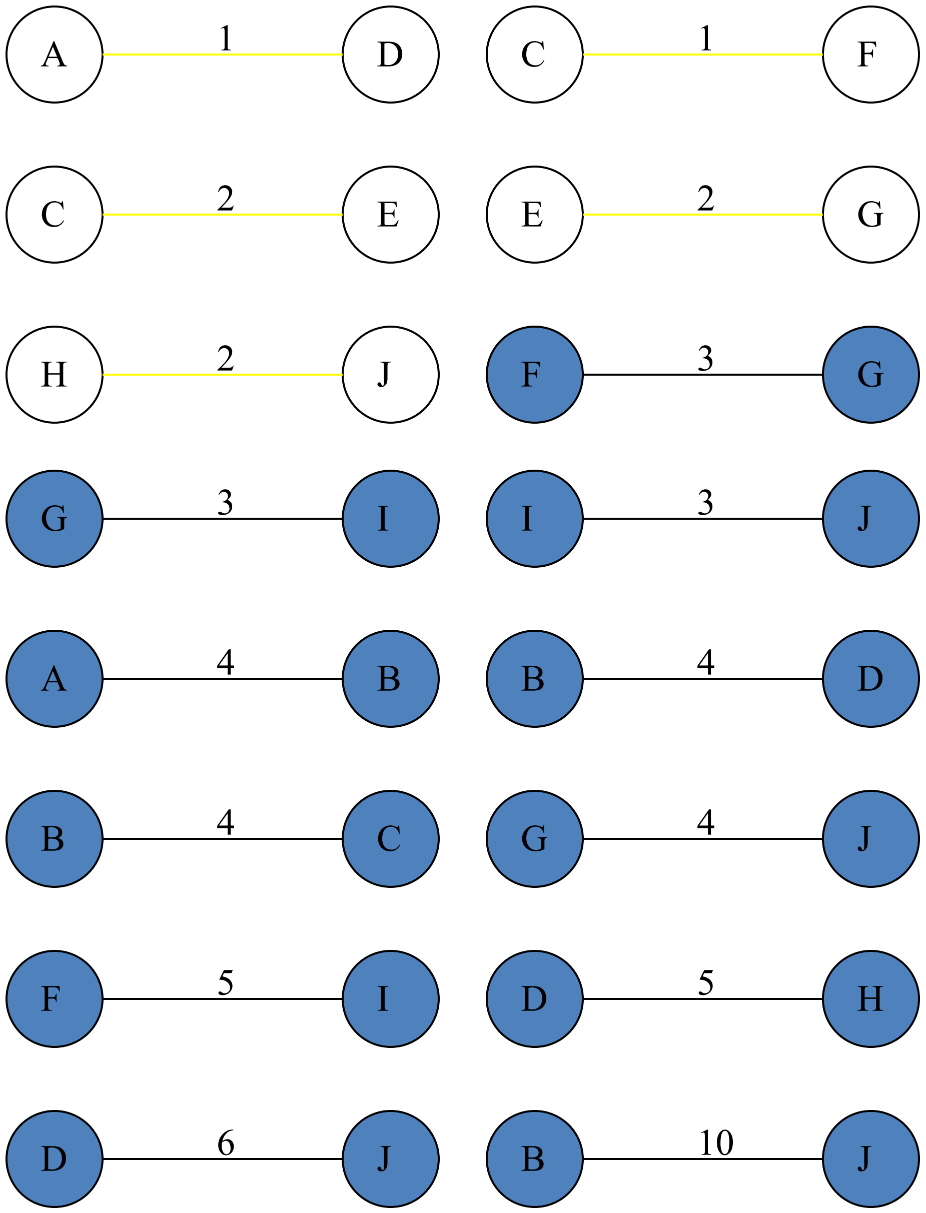




Kruskal's minimum spanning tree

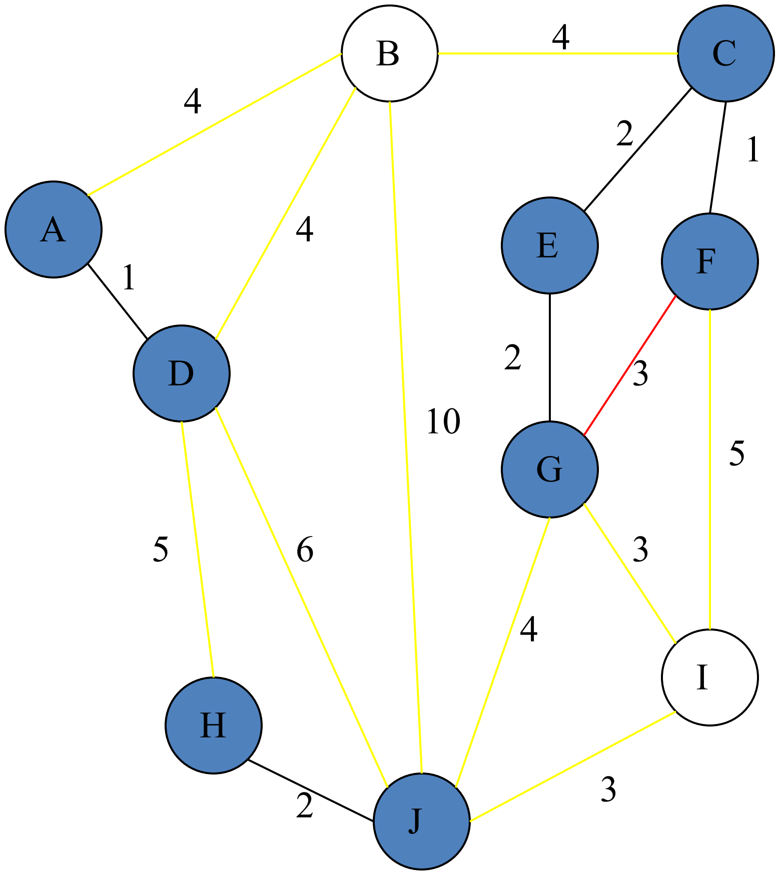
Current state of G1 = (V, T), add (h, j) to T

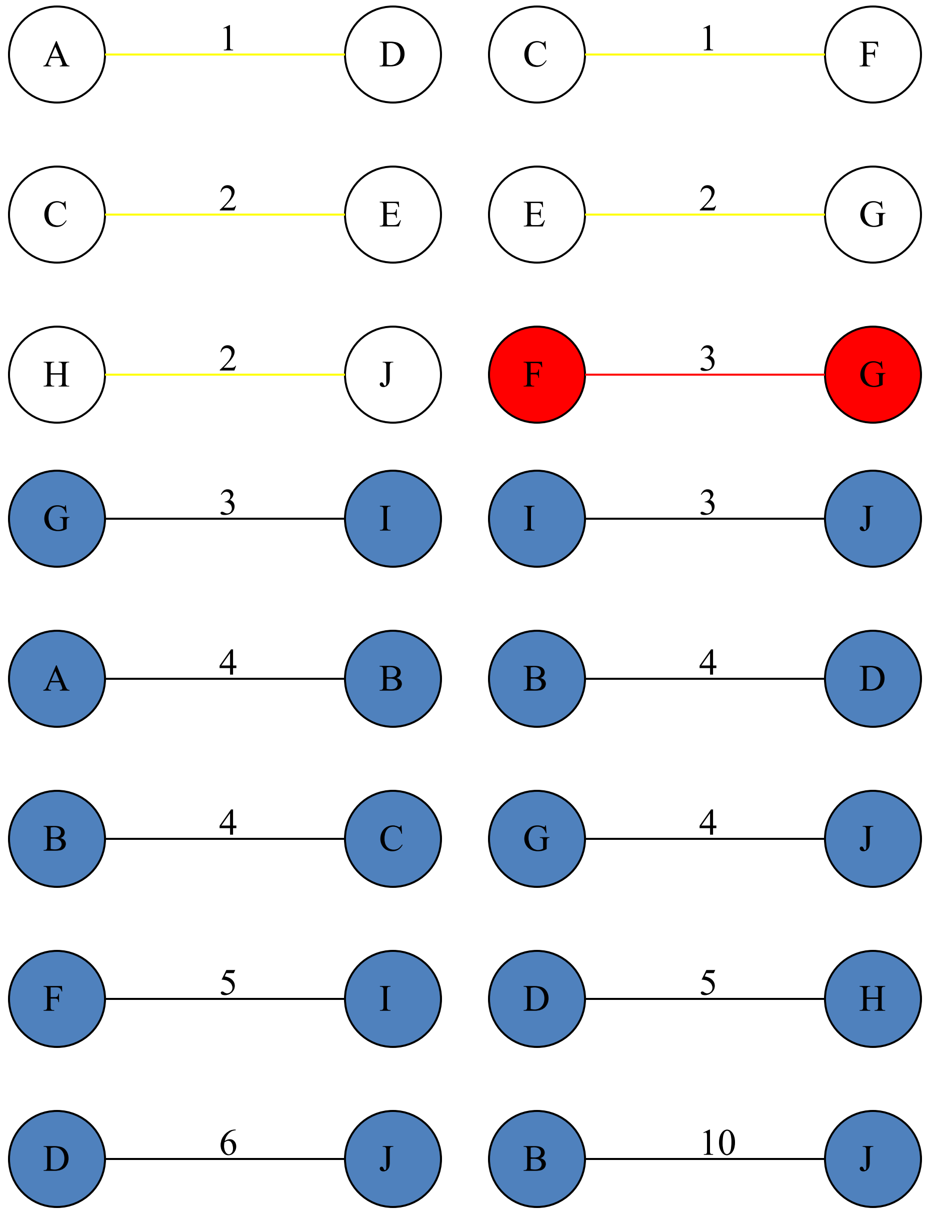




Kruskal’s minimum spanning tree

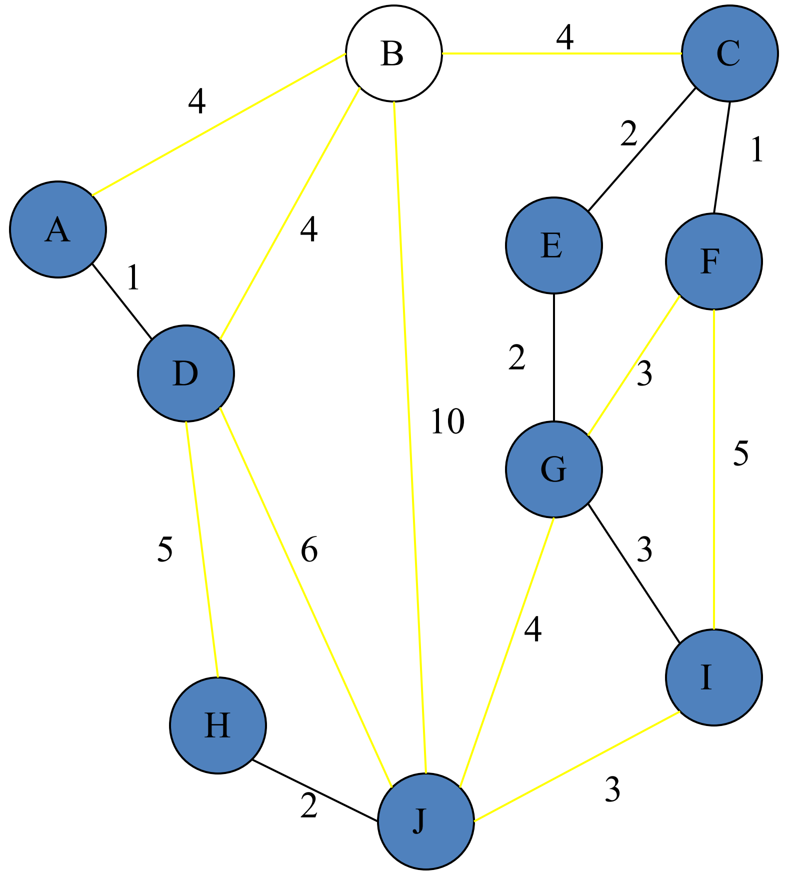
Current state of G1 = (V, T),
(f, g) forms a cycle No change

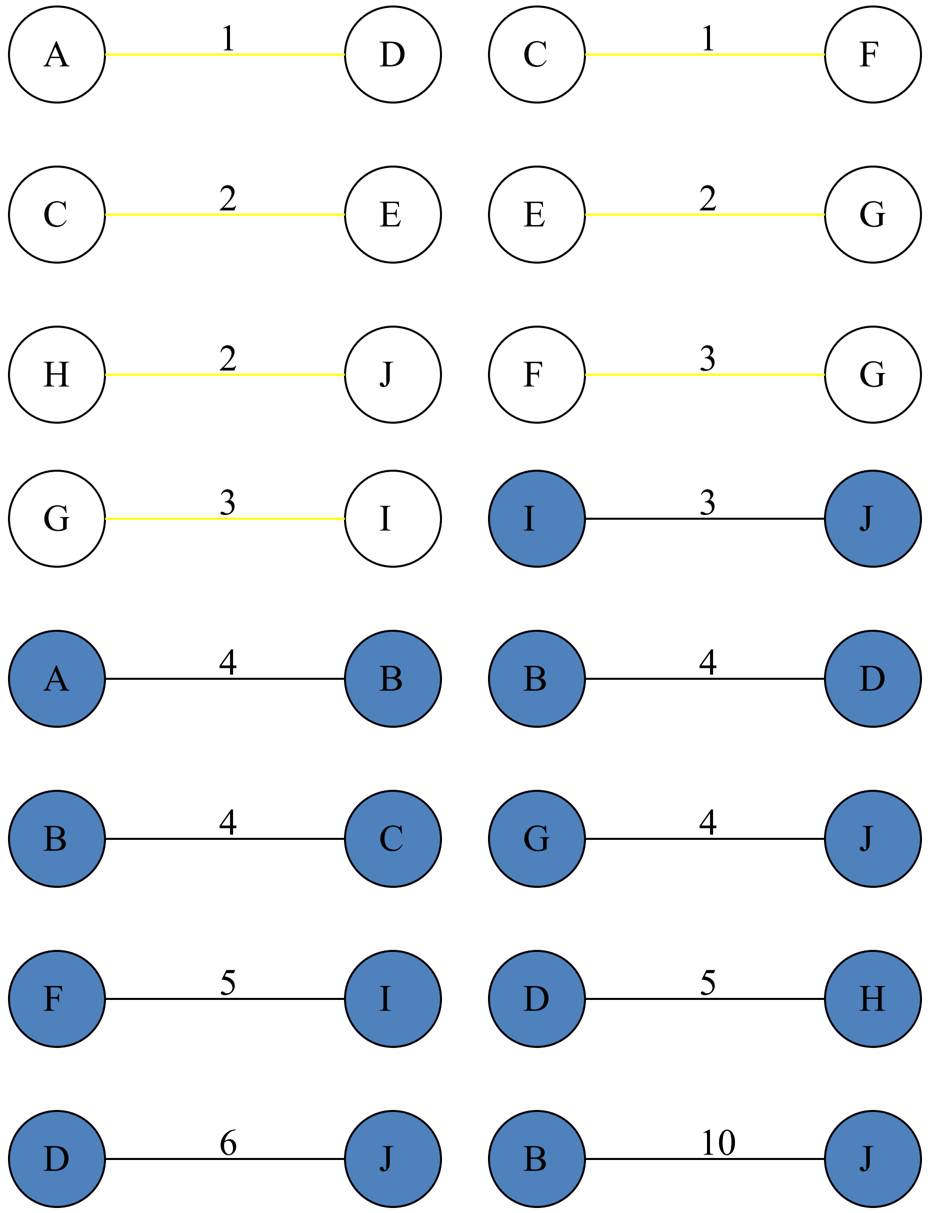




Kruskal’s minimum spanning tree

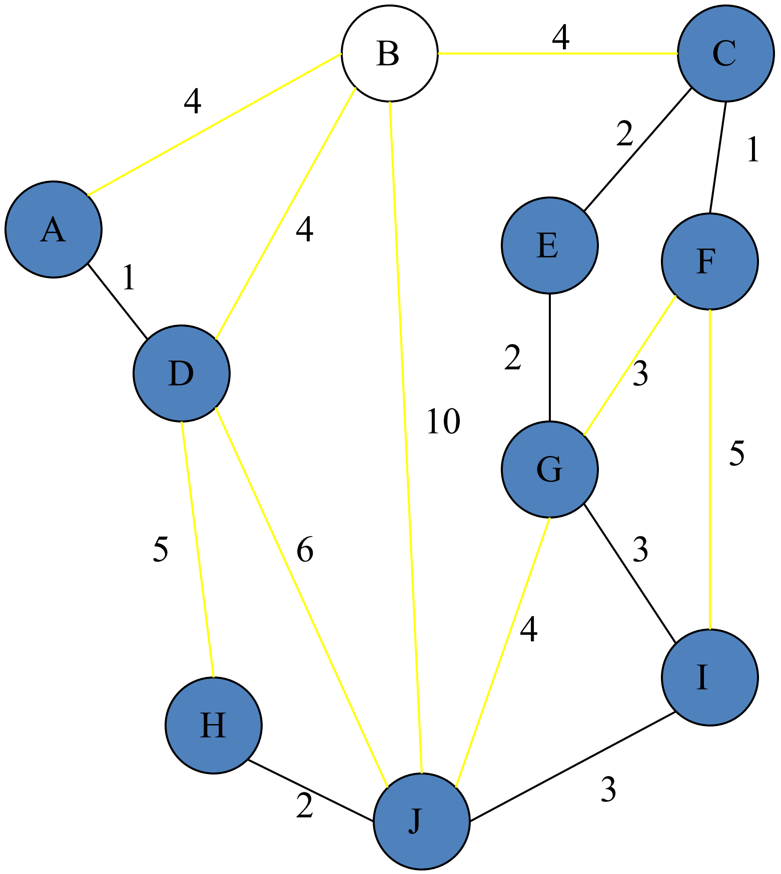
Current state of G1 = (V, T), add (g, i) to T

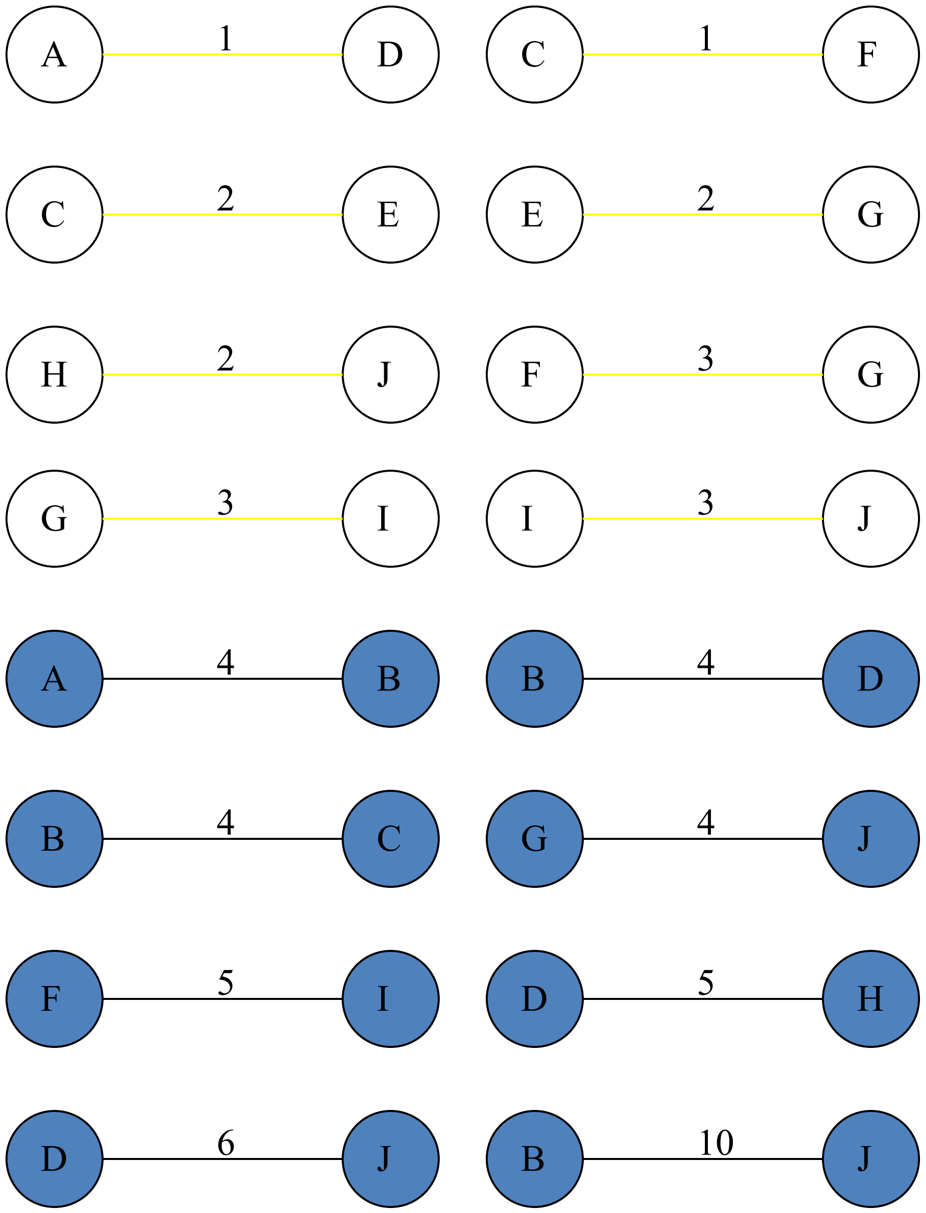




Kruskal's minimum spanning tree

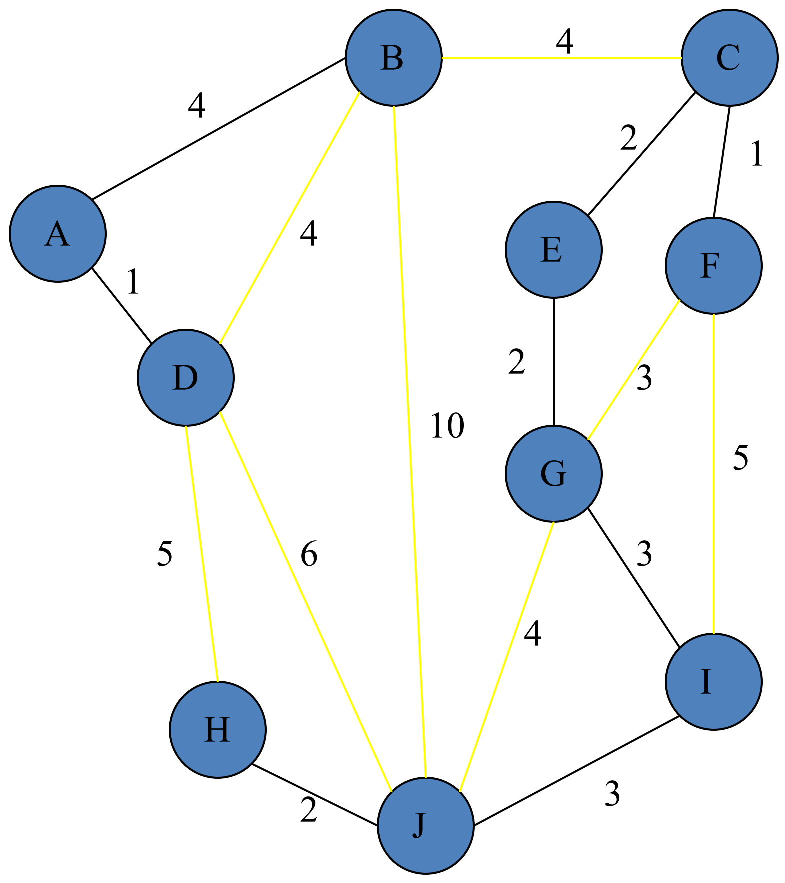
Current state of G1 = (V, T),
add (j, i) to T

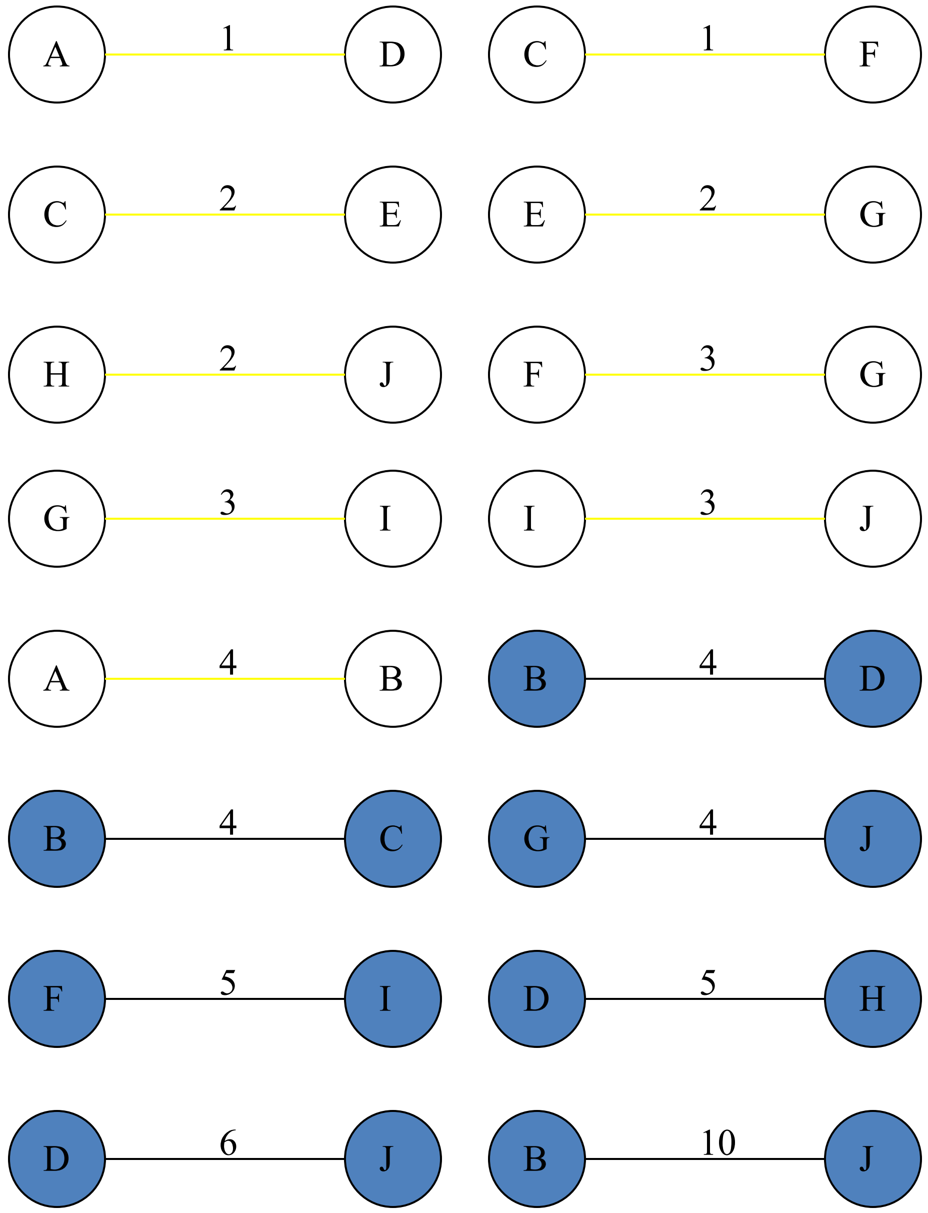




Kruskal’s minimum spanning tree

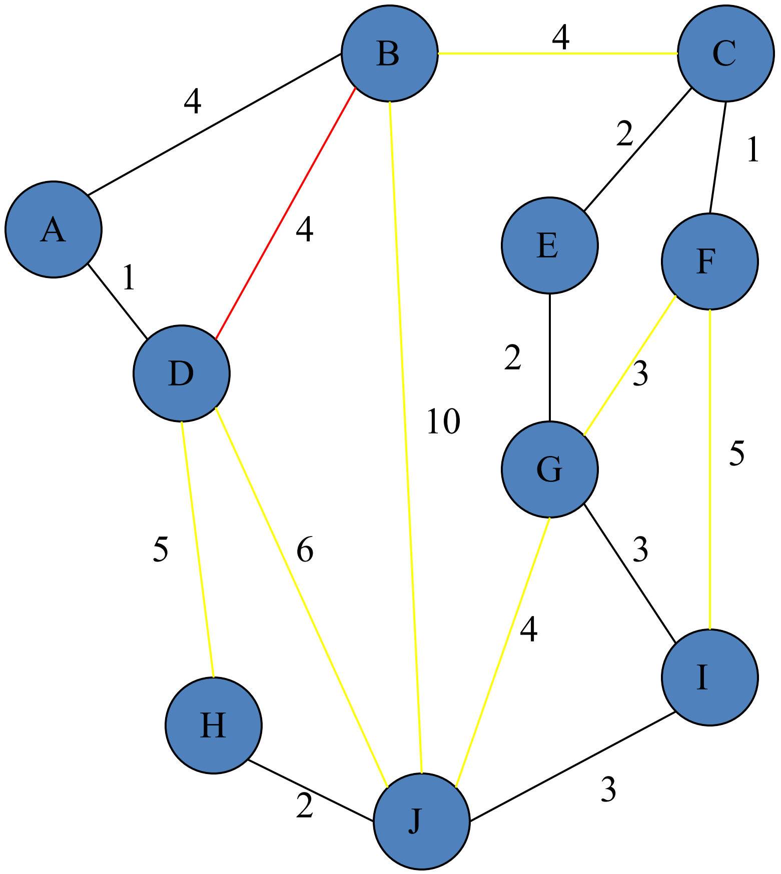
Current state of G1 = (V, T),
add (a, b) to T

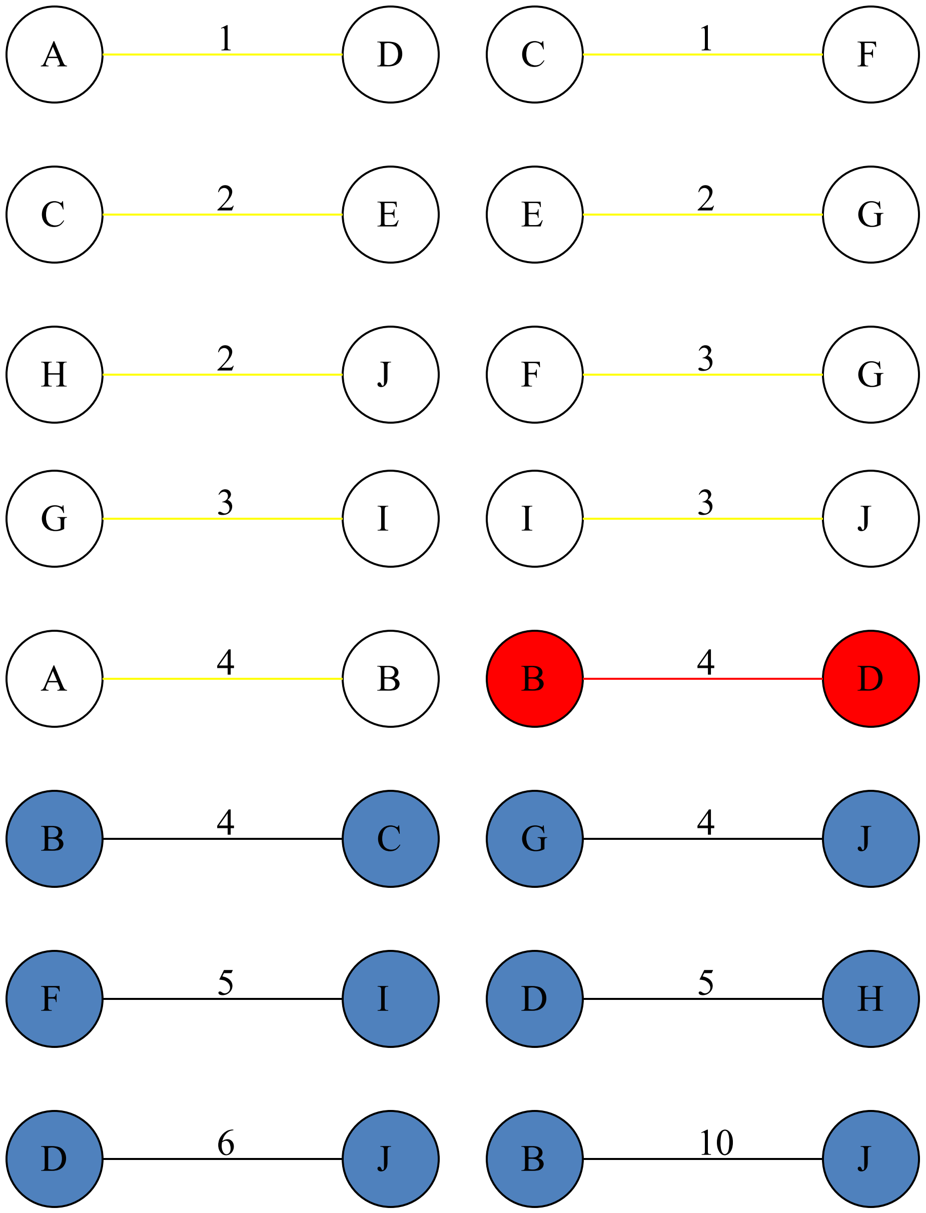




Kruskal’s minimum spanning tree

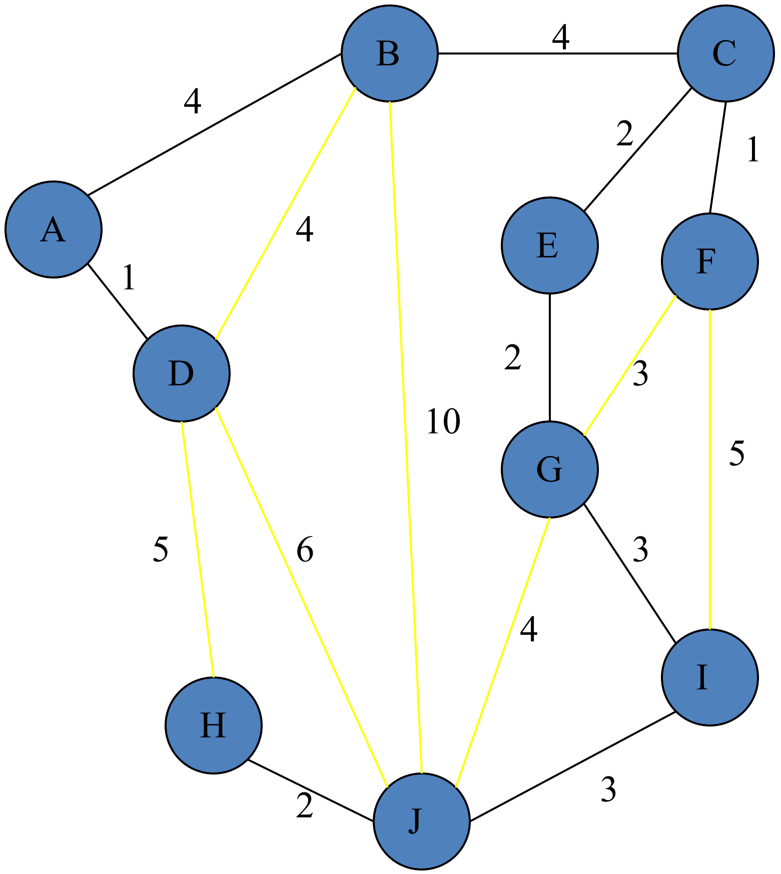
Current state of G1 = (V, T),
(b, d) forms a cycle no change

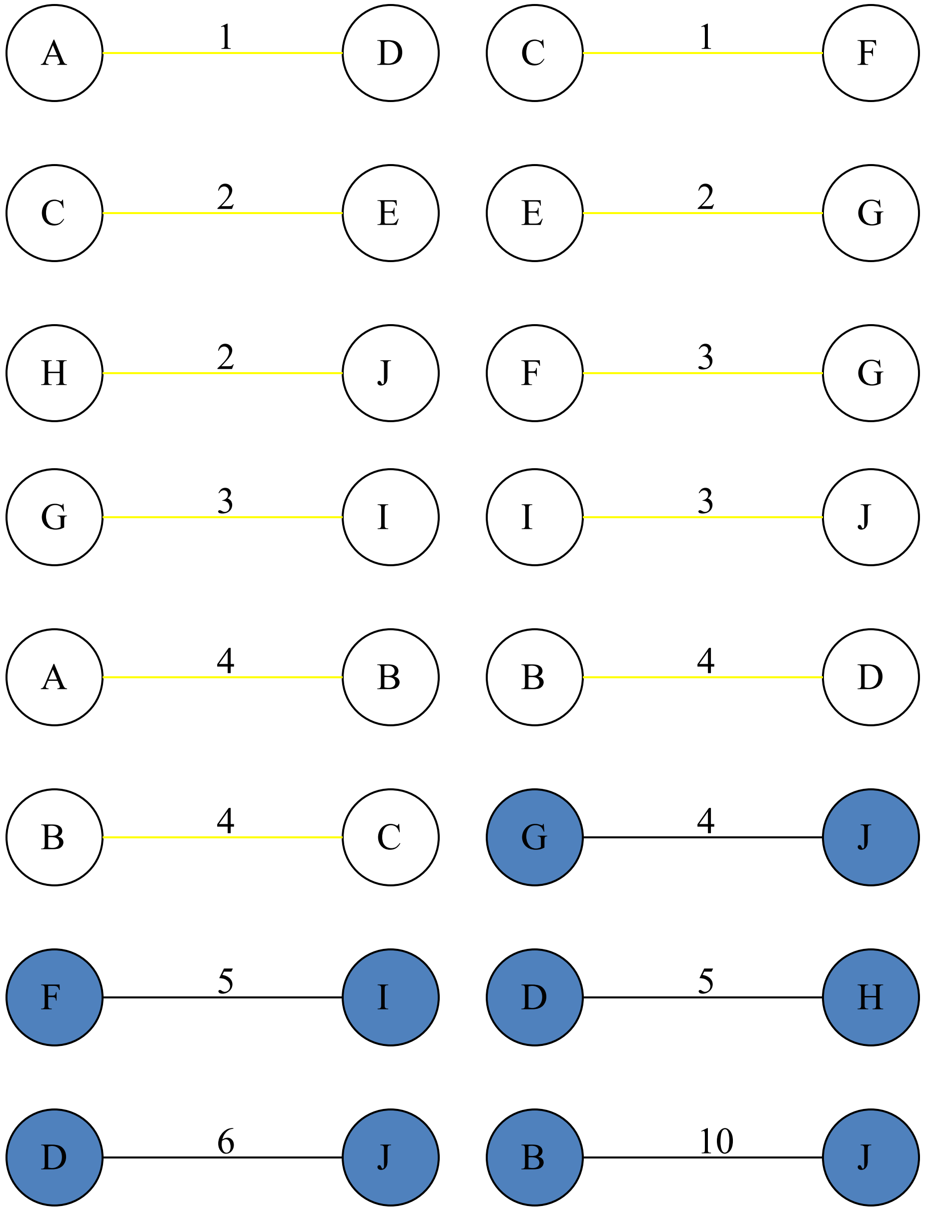




Kruskal's minimum spanning tree

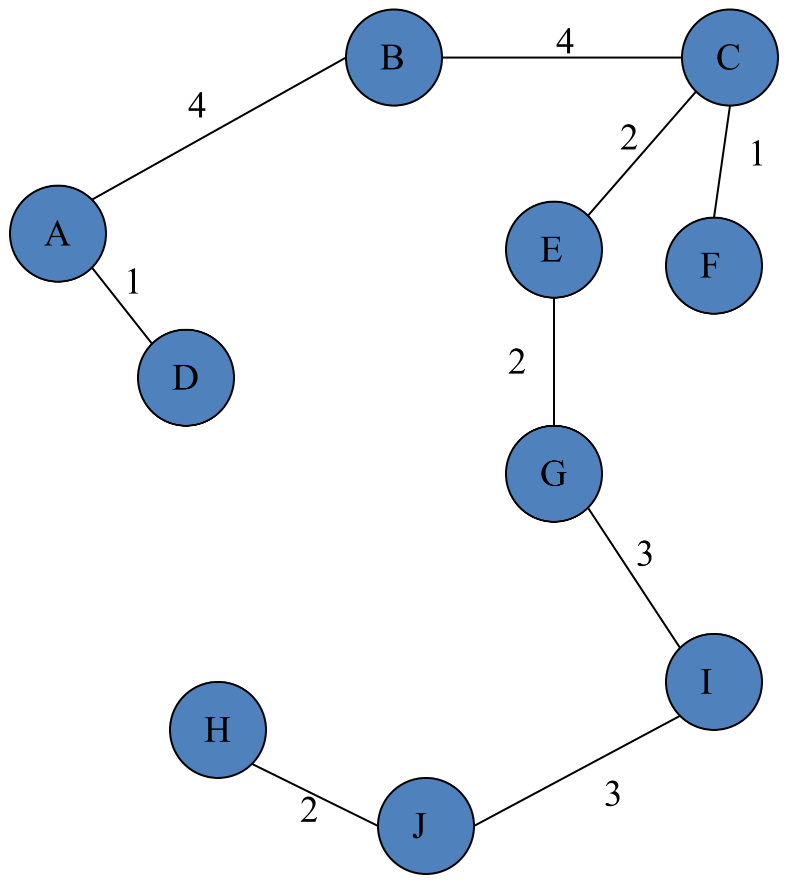
Current state of G1 = (V, T), add (b, c) to T



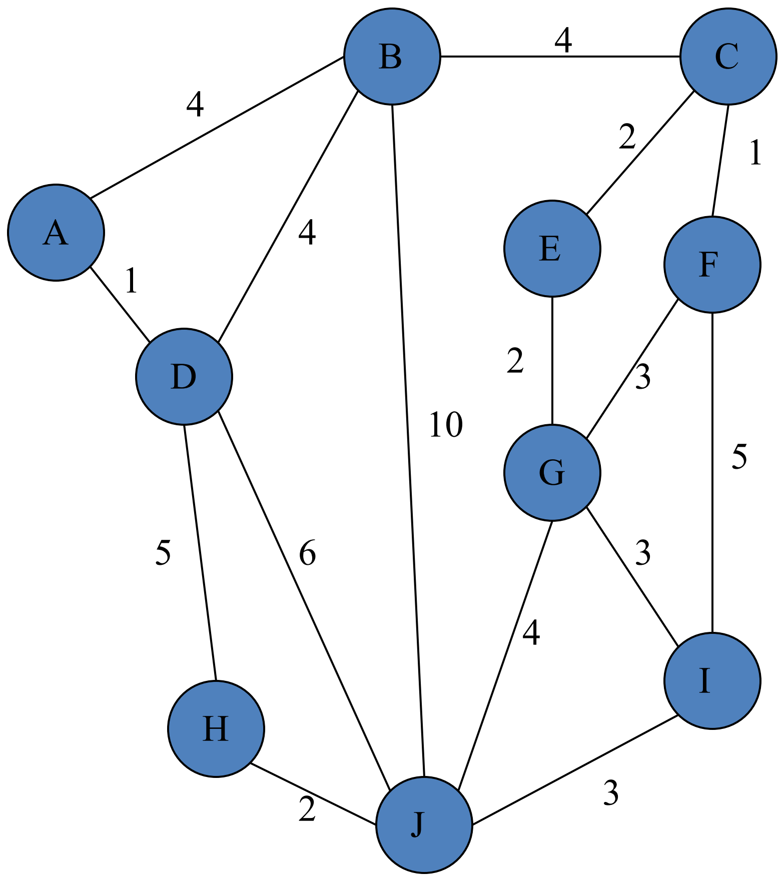


Kruskal’s minimum spanning tree

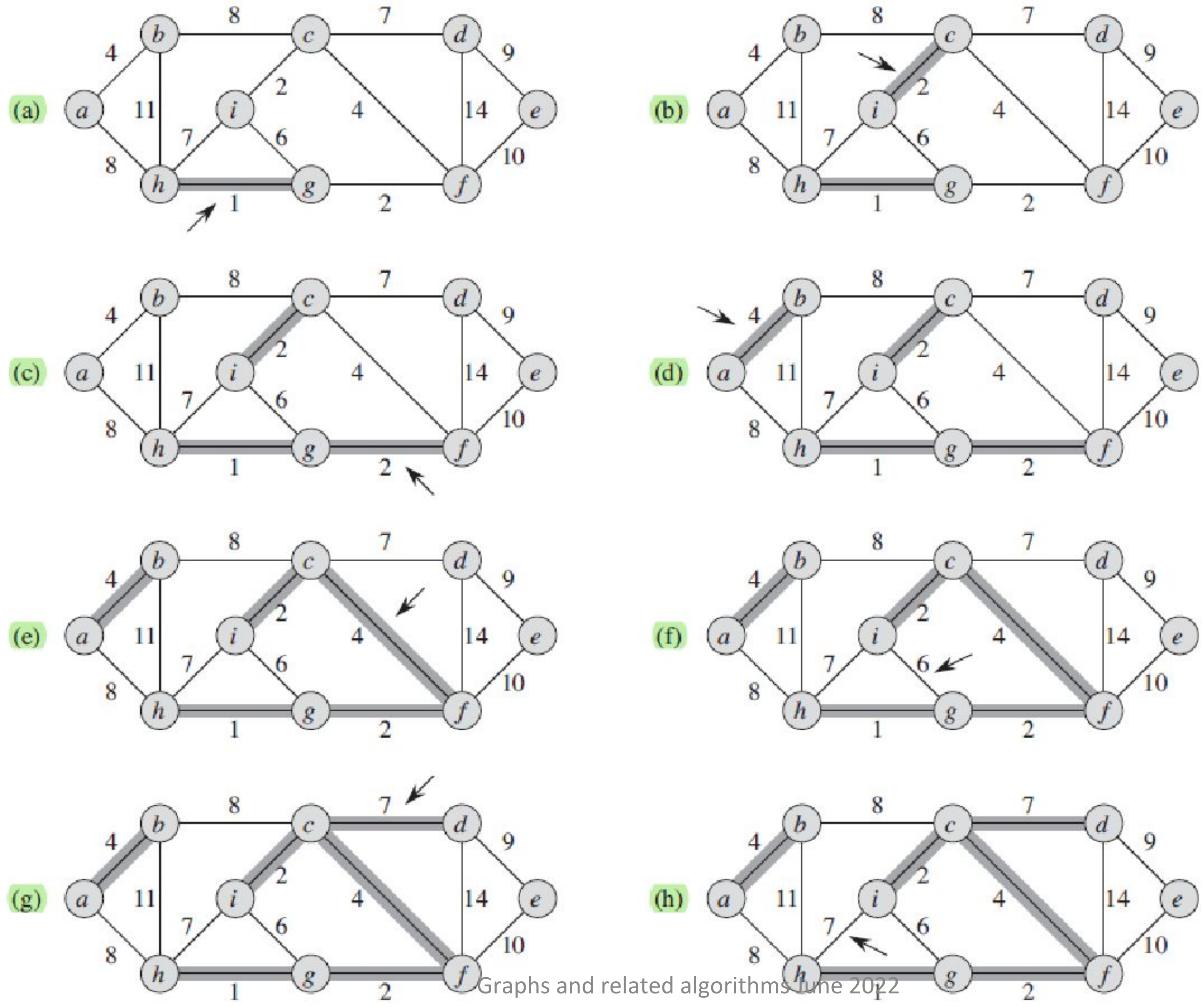
Current state of G1 = (V, T),
We have a minimum spanning tree



Complete graph

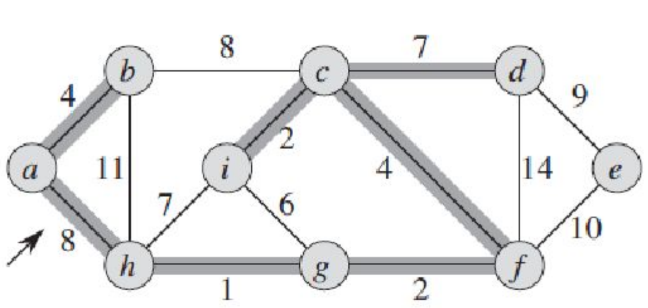


Kruskal’s minimum spanning tree

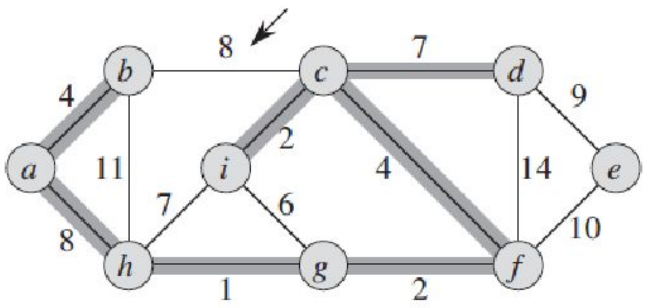


Kruskal’s minimum spanning tree

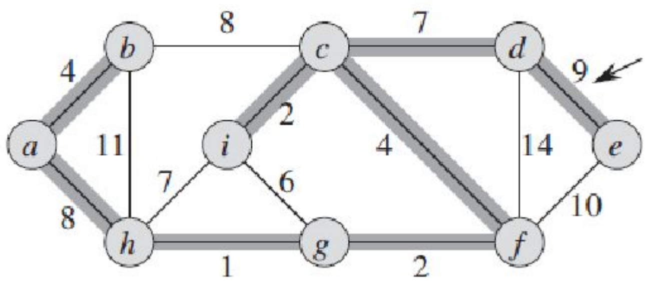
(i)



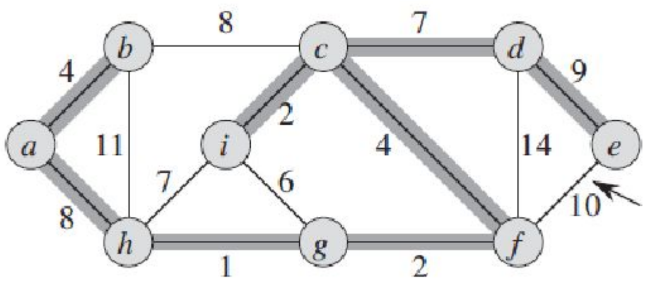
(j)



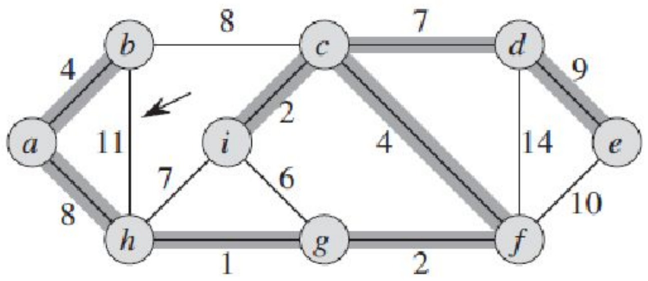
(k)



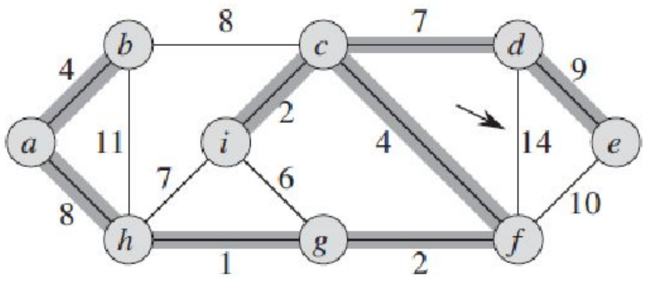
(l)



(m)



(n)



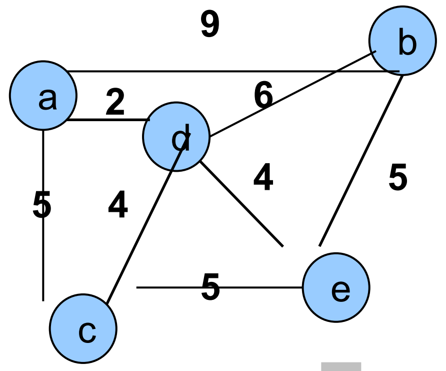
Kruskal's minimum spanning tree

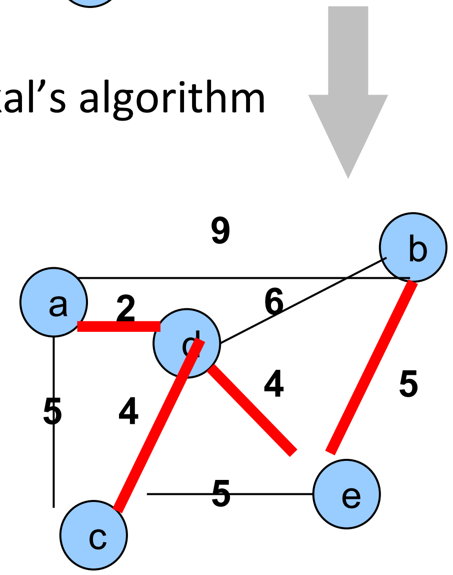
Kruskal's algorithm on G = (V, E), with weights of edges in array W = [w(e)]

```
function MST-Kruskal(G, W)
      T = Φ
      for each vertex v ε V
       Make-Set(v)                  //created |V| sets each with one vertex
                                    //each set is identified by a specific member of the set
      sort edges in E into non-decreasing order by weight w(e)
                                    //instead, partially sort the edges using a (min) binary heap
      for each edge (u, v) in E     //in non-decreasing order of weight w(e)
                                    //Or stop after one has added |V|-1 edges
       if Find-Set(u) ≠ Find-set(v)
              T = T U {(u, v)}      //add edge (u, v) to T
              Union(u, v)           //merge two sets that contain vertices u and v
       delete edge e                //delete edge e from sorted list or from min heap
return T
```

Kruskal’s minimum spanning tree

```
function MST-Kruskal(G, W)
    T = Φ
    for each vertex v ε V
        Make-Set(v)
    sort edges in E into non-decreasing order by weight w(e)
    for each edge (u, v) in E
        if Find-Set(u) ≠ Find-set(v)
            T = T U {(u, v)}
            Union(u, v)
        delete edge e
return T
```





State of computation

Initial:	set of sets = [{a}, {b}, {c}, {d}, {e}]	T = []
++ edge (a, d):	set of sets = [{a, d}, {b}, {c}, {e}]	T = [(a,d)]
++ edge (d, c):	set of sets = [{a, d, c}, {b}, {e}]	T = [(a,d), (d, c)]
++ edge (d, e):	set of sets = [{a, d, c, e}, {b}]	T = [(a,d), (d, c), (d, e)]
++ edge (e, b):	set of sets = [{a, d, c, e, b}]	T = [(a,d), (d, c), (d, e), (e, b)]

Sets, and their representation

Study how to create sets, and manage them:

Operations on sets:

- Make-Set(v)
- Find-Set(u)
- Union(u, v)

Sets, and their representation

Consider the universe of symbols, U = {1, 2, …, N} or U = {red, green, blue, … }

And now consider one or more sets, S1, S2, etc. the Union of which is the universe U
For example S1 = {1, 7, 8, 9}, S2 = {2, 5, 10}, S3 = {3, 4, 6}
Note S1, S2, S3 are disjoint, and together they cover the entire universe, viz. U = S1 U S2 U S3

Equivalently, the universe U is portioned into multiple sets, S1, S2, etc.

Question how do we represent them, and carry out operations efficiently

- Make-Set(v)
- Find-Set(u) ≠ Find-set(v)
- Union(u, v)

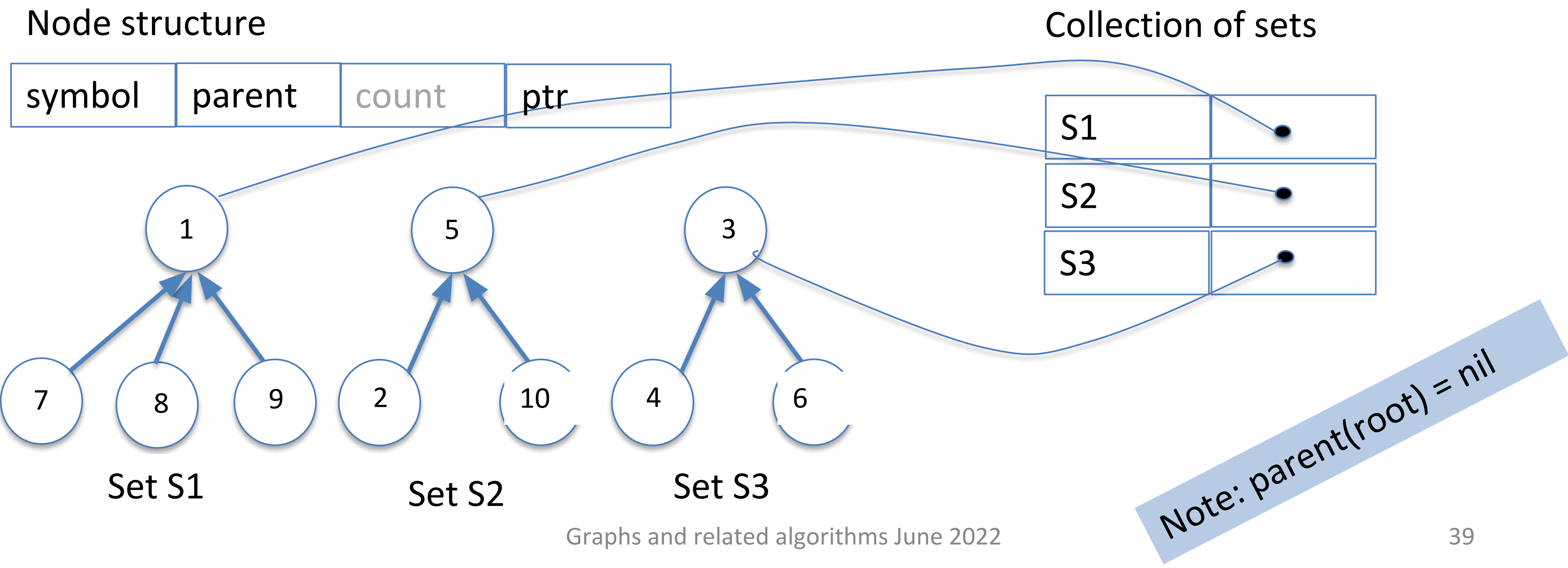
Sets, and their representation

For example, U = {1, 2, …, 10} and disjoints sets S1 = {1, 7, 8, 9}, S2 = {2, 5, 10}, S3 = {3, 4, 6}
Note S1, S2, S3 are disjoint, and together they cover the entire universe, viz. U = S1 U S2 U S3

Here is one way to represent the disjoint sets that makes it efficient to carry out operations:

- Make-Set(v)
- Find-Set(u) ≠ Find-set(v)
- Union(u, v)

That nodes point to their parents will have significance to "Union" and "Find

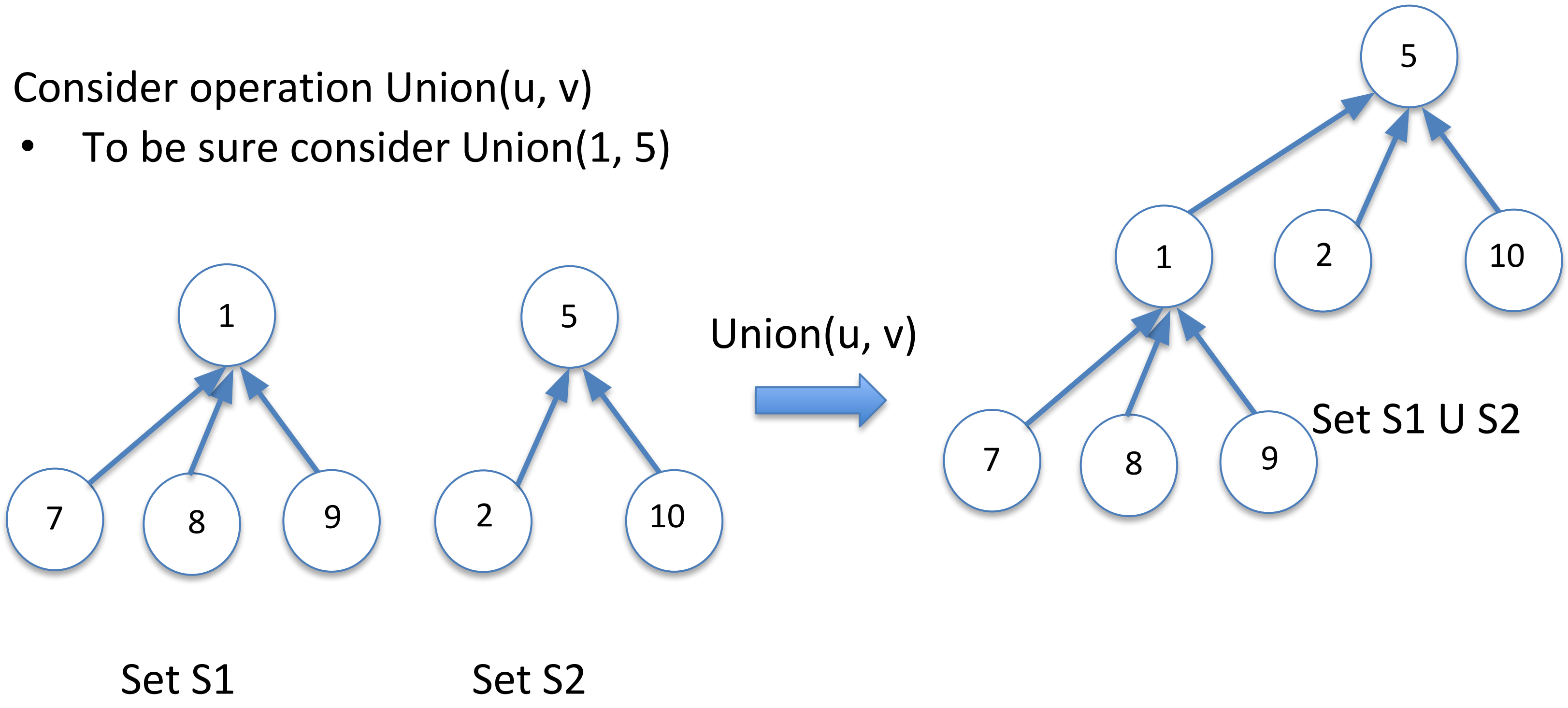


Sets, and their representation

For example, U = {1, 2, …, 10} and disjoints sets S1 = {1, 7, 8, 9}, S2 = {2, 5, 10}, S3 = {3, 4, 6}
Note S1, S2, S3 are disjoint, and together they cover the entire universe, viz. U = S1 U S2 U S3

Consider operation Union(u, v)

- To be sure consider Union(1, 5)



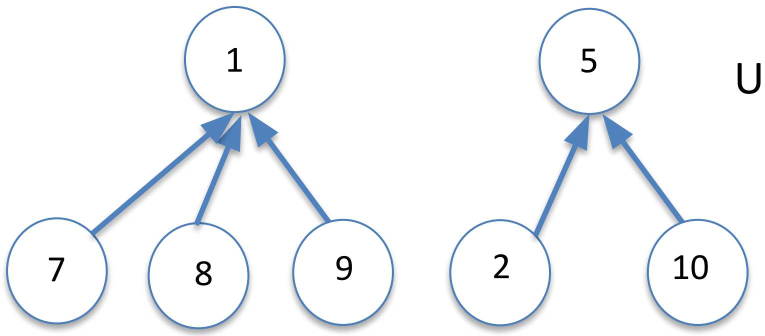
```
function Union (u, v)
        parent(u) = v
return(v)
```

Sets, and their representation

For example, U = {1, 2, …, 10} and disjoints sets S1 = {1, 7, 8, 9}, S2 = {2, 5, 10}, S3 = {3, 4, 6}
Note S1, S2, S3 are disjoint, and together they cover the entire universe, viz. U = S1 U S2 U S3

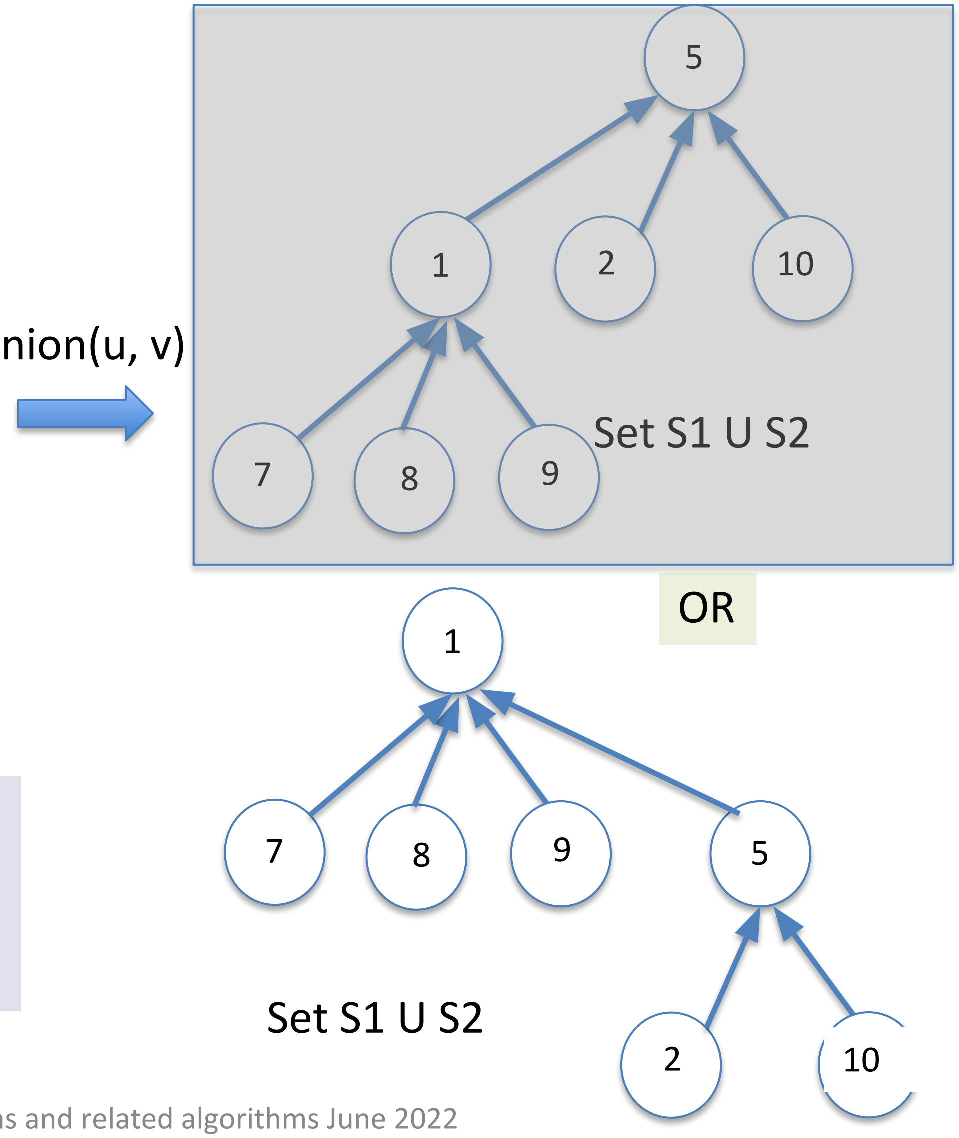
Consider operation Union(u, v)

- To be sure consider Union(1, 5)



Set S1   Set S2

Union(u, v)



Set S1 U S2

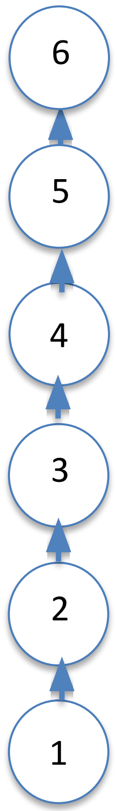
```
function Union (u, v)
        ~~parent(u) = v~~ parent(v) = u
return(v)
```

Sets, and their representation

In the worst case the height of tree will be O(n), where n is the number of symbols

For example, U = {1, 2, …, 6}, S1={1}, S2={2}, S3={3}, S4={4}, S5={5}, S6={6}, and consider

Union(1, 2)
Union(2, 3)
Union(3, 4)
Union(4, 5)
Union(5, 6)



While Union(u,v) is efficient, or O(1), a Find(u) operation will be complex, or O(n) in the worst case, where n = |U|

⍰ Form the union so as to minimize the height of resulting tree

```
function Union (u, v)
        parent(u) = v
return(v)
```

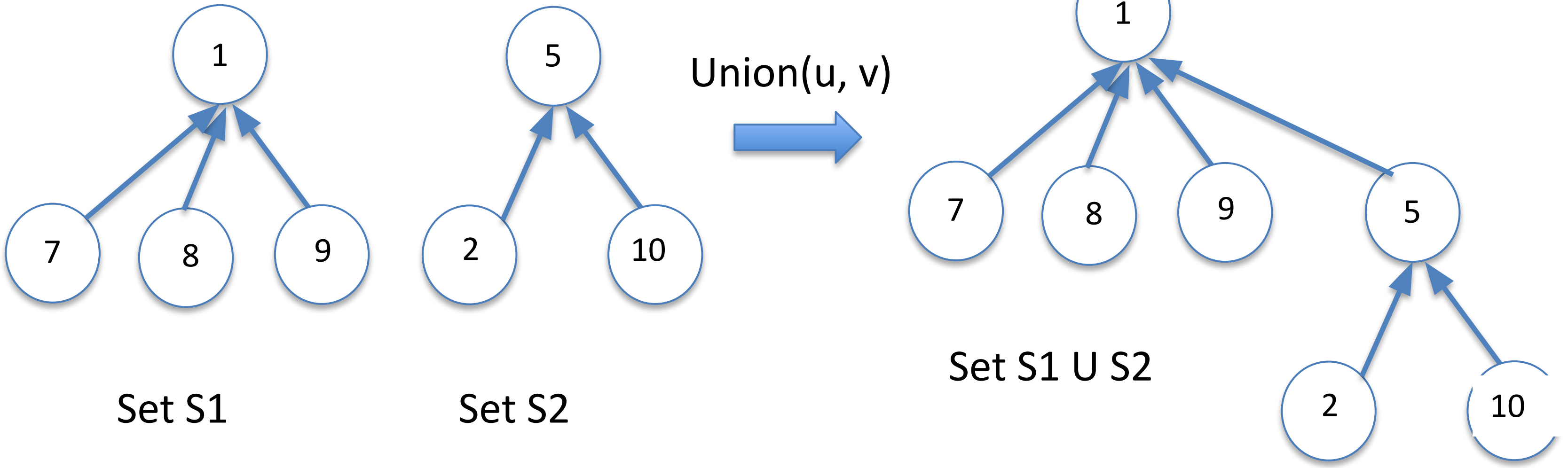
Sets, and their representation

Another approach where we maintain count of symbols in set (or sub-tree):

Node structure

symbol	parent	count

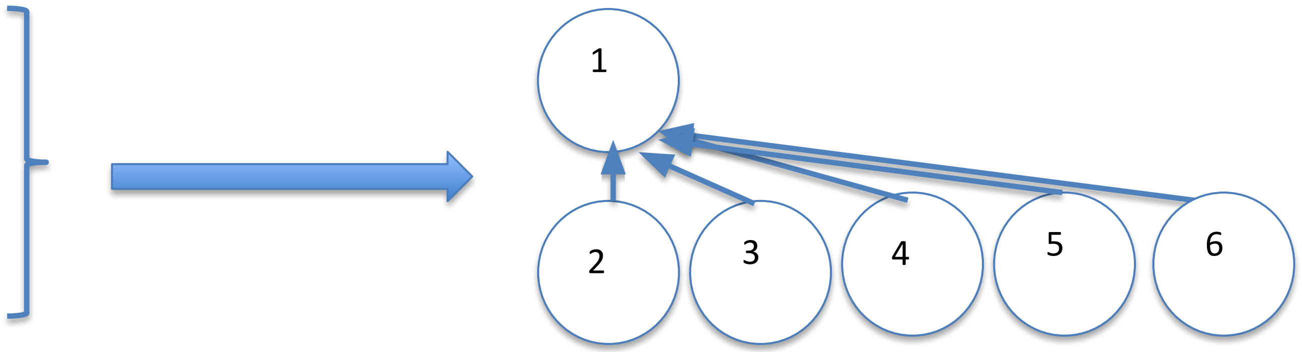
```
function Union (u, v)
    if count(u) < count(v)
    then parent(u) = v
         count(v) = count(v) + count(u)
    else parent(v) = u
         count(u) = count(u) + count(v)
return(v)
```



Sets, and their representation

For example, U = {1, 2, …, 6}, S1={1}, S2={2}, S3={3}, S4={4}, S5={5}, S6={6}, and consider

Union(1, 2)
Union(2, 3)
Union(3, 4)
Union(4, 5)
Union(5, 6)



```
function Union (u, v)
        if count(u) < count(v)
        then parent(u) = v
                count(v) = count(v) + count(u)
        else parent(v) = u
                count(u) = count(u) + count(v)
return(v)
```

Sets, and their representation

Another approach where we maintain count of symbols in set (or sub-tree):

Node structure

symbol	parent	count

```
function Union (u, v)
    if count(u) < count(v)
    then parent(u) = v
        count(v) = count(v) + count(u)
    else parent(v) = u
        count(u) = count(u) + count(v)
return(v)
```

- Every node in resulting tree has level $\leq$ floor($\log_2$ n) + 1
- Find(u) runs in time O($\log_2$ n)

```
function find(u)
    temp = u
    while parent(temp) ≠ nil do
        temp = parent(temp)
return(temp)
```

Sets, and their representation

Time complexity

Union operation: O(1)

```
function Union (u, v)
    if count(u) < count(v)
    then parent(u) = v
        count(v) = count(v) + count(u)
    else parent(v) = u
        count(u) = count(u) + count(v)
return(v)
```

Find operation: $O(\log_2 N)$

```
function find(u)
    temp = u
    while parent(temp) ≠ nil do
        temp = parent(temp)
return(temp)
```

Kruskal’s minimum spanning tree

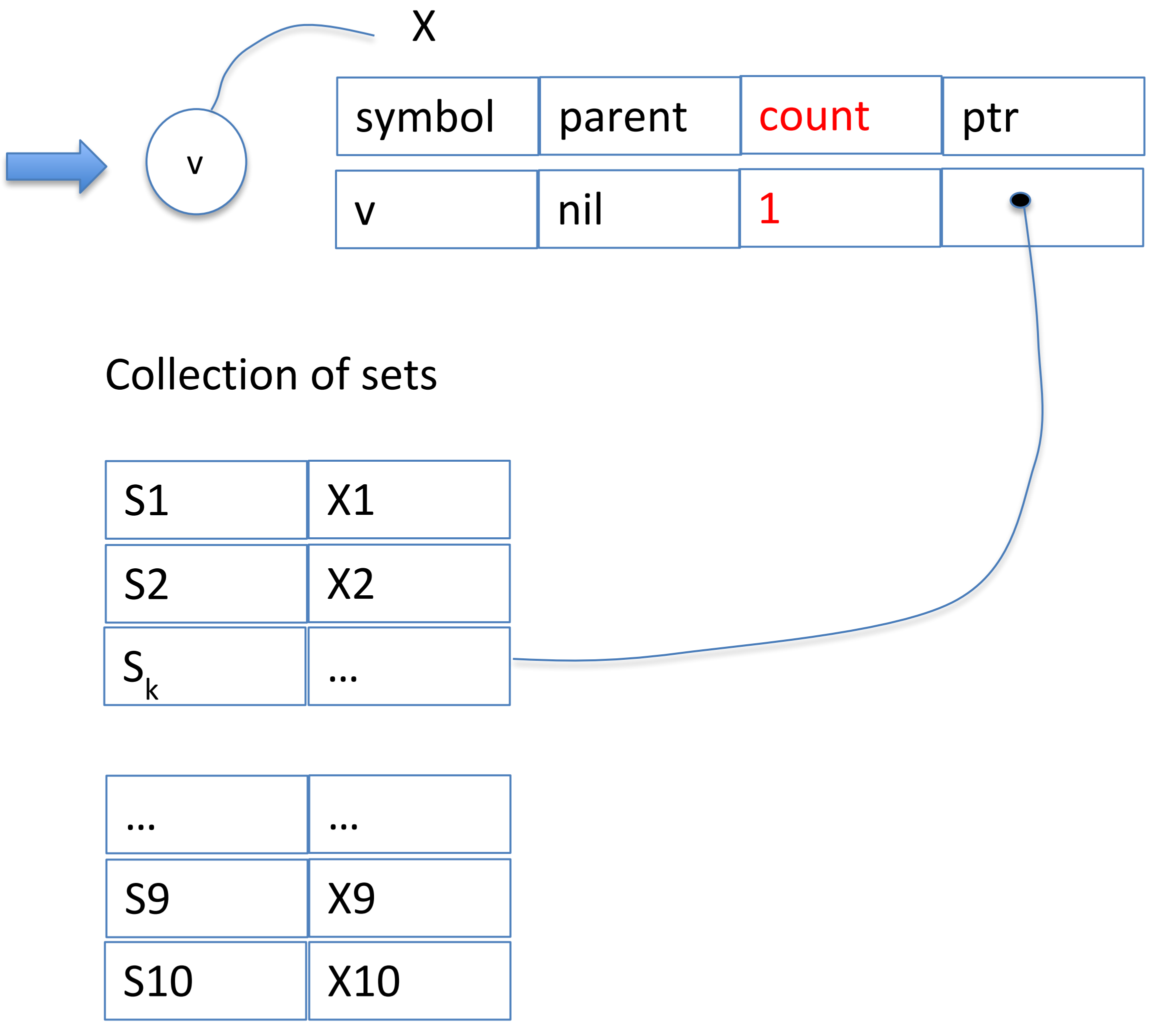
Kruskal’s algorithm on G = (V, E), with weights of edges in array W = [w(e)]

```
function MST-Kruskal(G, W)
      T = Φ
      for each vertex v ε V
       Make-Set(v)                    //created |V| sets each with one vertex
                                      //each set is identified by a specific member of the set
      sort edges in E into non-decreasing order by weight w(e)
                                      //instead, partially sort the edges using a (min) binary heap
      for each edge (u, v) in E             //in non-decreasing order of weight w(e)
                                      //Or stop after one has added |V|-1 edges
       if Find-Set(u) ≠ Find-set(v)
             T = T U {(u, v)}         //add edge (u, v) to T
             Union(u, v)              //merge two sets that contain vertices u and v
       delete edge e                  //delete edge e from sorted list or from min heap
return T
```

Kruskal's minimum spanning tree

Make-Set(v) ⍰

```
function Make-Set(v)
    X = getnode()
    X.symbol = v
    X.parent = nil
    X.count = 1
    X.ptr = k
return X
```



S1	X1
S2	X2
S_k	…
…	…
S9	X9
S10	X10

Kruskal’s minimum spanning tree

Time complexity of Kruskal’s algorithm on G = (V, E), with weights of edges in array W = [w(e)]

Let n = |V|, m = |E|

```
function MST-Kruskal(G, W)
    T = Φ                                                  O(1)
    for each vertex v ε V                                  O(n)
     Make-Set(v)          //cremated |V| sets each with one vertex
                          //each set is identified by a specific member of the set
    sort edges in E into non-decreasing order by weight w(e)       O(m log m) or O(log m)
                          //instead, partially sort the edges using a (min) binary heap
    for each edge (u, v) in E       //in non-decreasing order of weight w(e)       O(m log m) = O(m log n)
                          //Or stop after one has added |V|-1 edges
     if Find-Set(u) ≠ Find-set(v)
         T = T U {(u, v)}   //add edge (u, v) to T
         Union(u, v)        //merge two sets that contain vertices u and v
     delete edge e        //delete edge e from sorted list or from min heap
    return T
```

 Time complexity of Kruskal’s algorithm: O(|E| log |E|) = O(|E| log |V|)

Prim’s minimum spanning tree

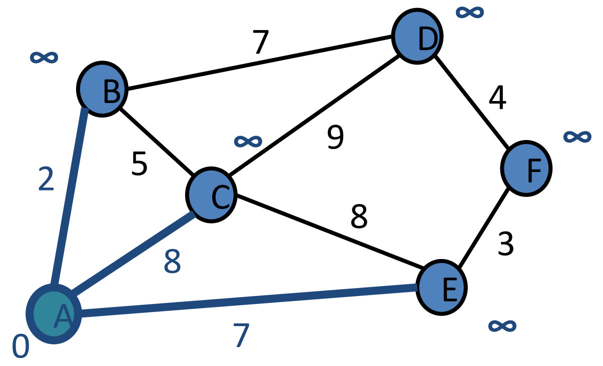
Kruskal’s algorithm:

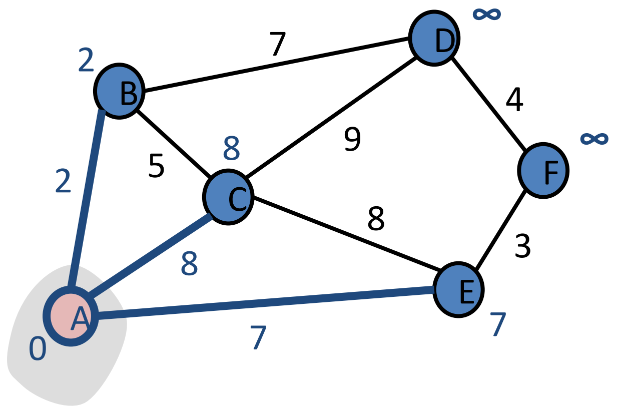
- Start with all vertices but no edges in the spanning tree
- Repeatedly add the cheapest edge that does not create a cycle

Prim’s algorithm:

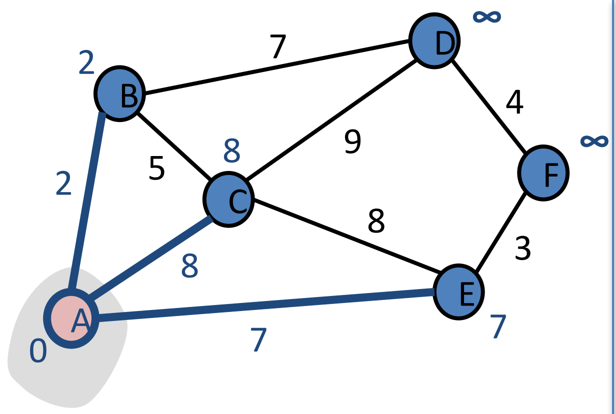
- Start with any one vertex in the spanning tree
- Repeatedly add the cheapest edge, and the NEW node it leads to
  - the new vertex is not in the spanning tree

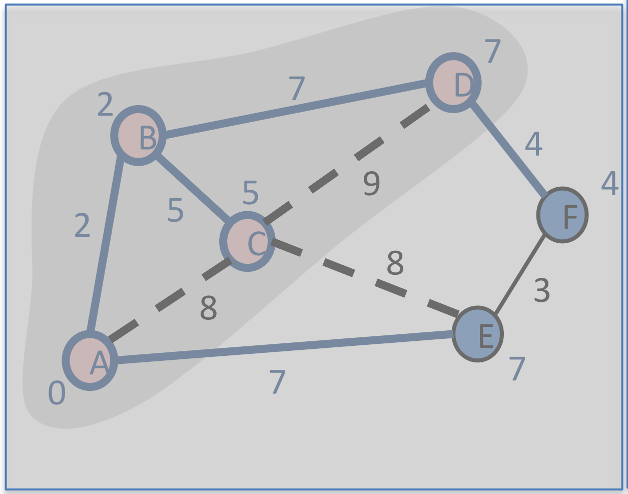
Prim's minimum spanning tree

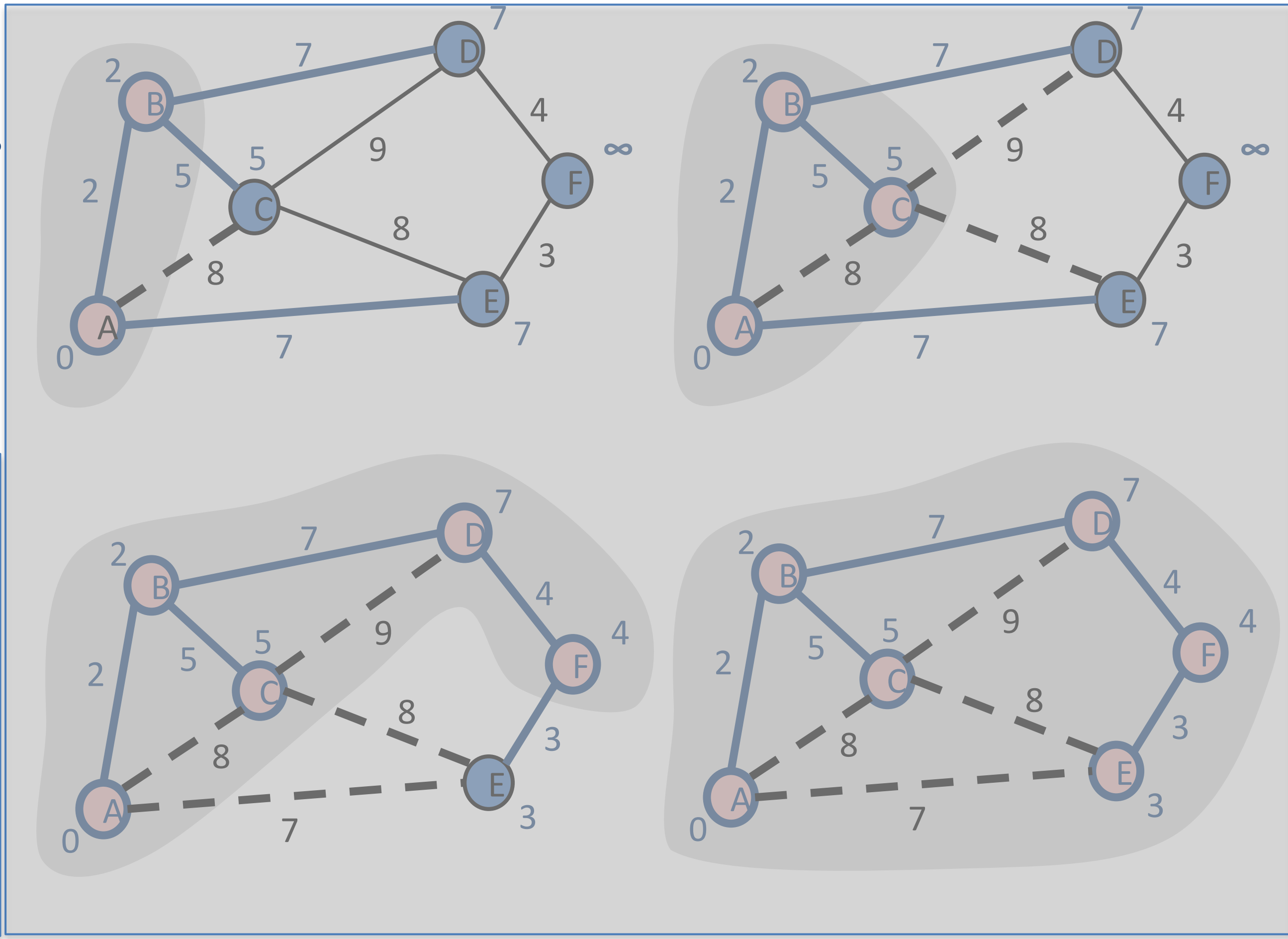




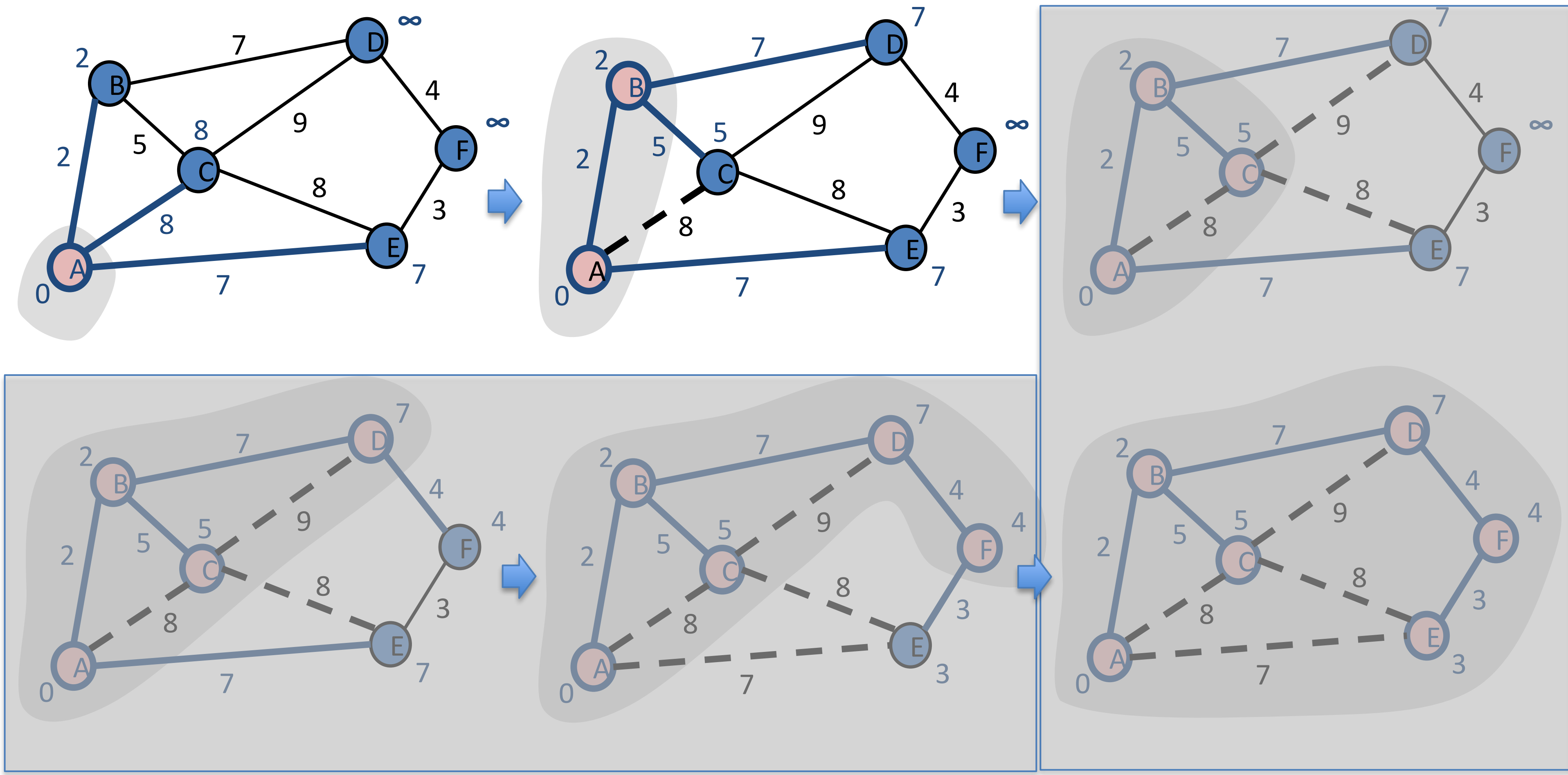
Prim’s minimum spanning tree



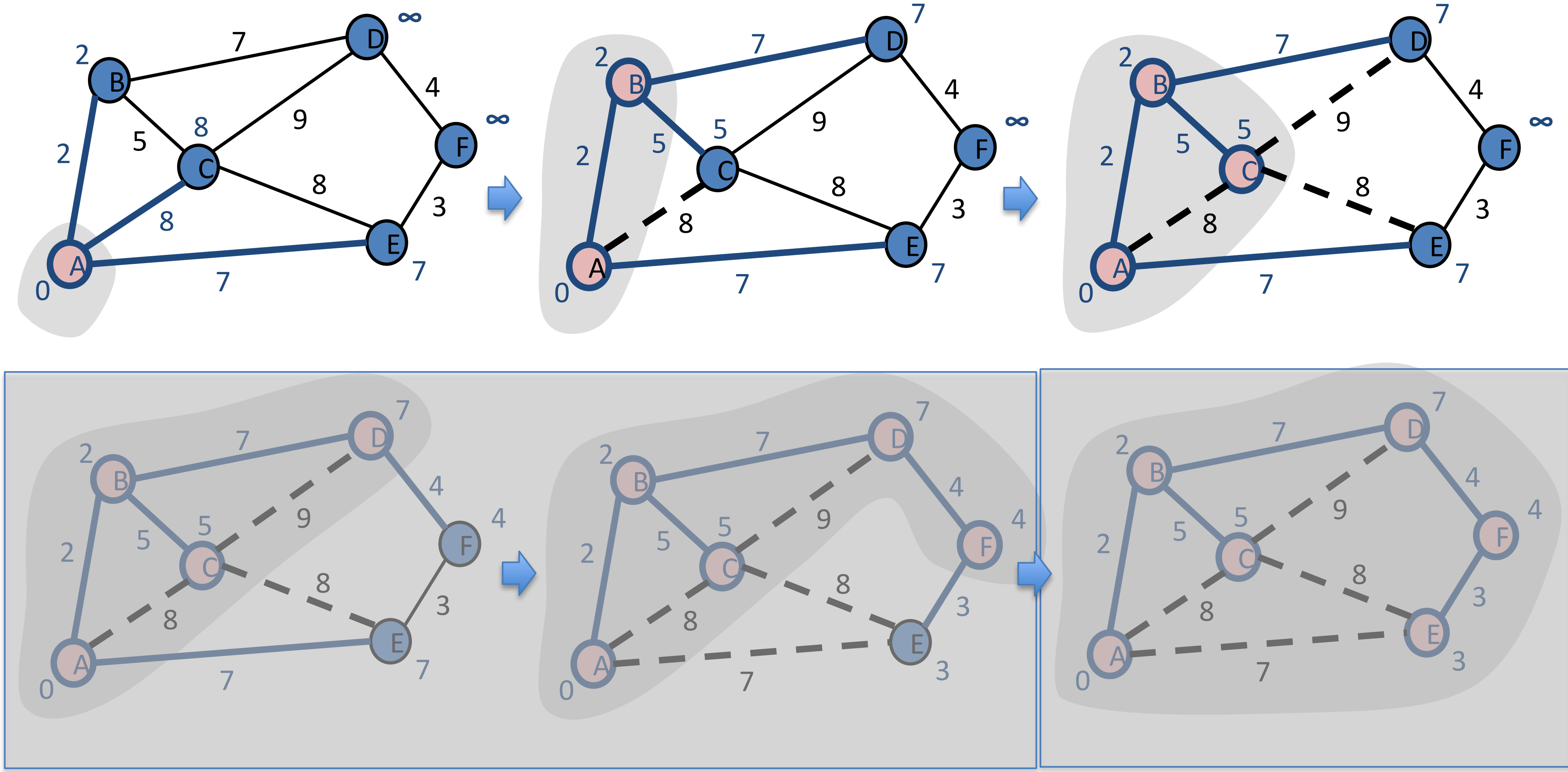




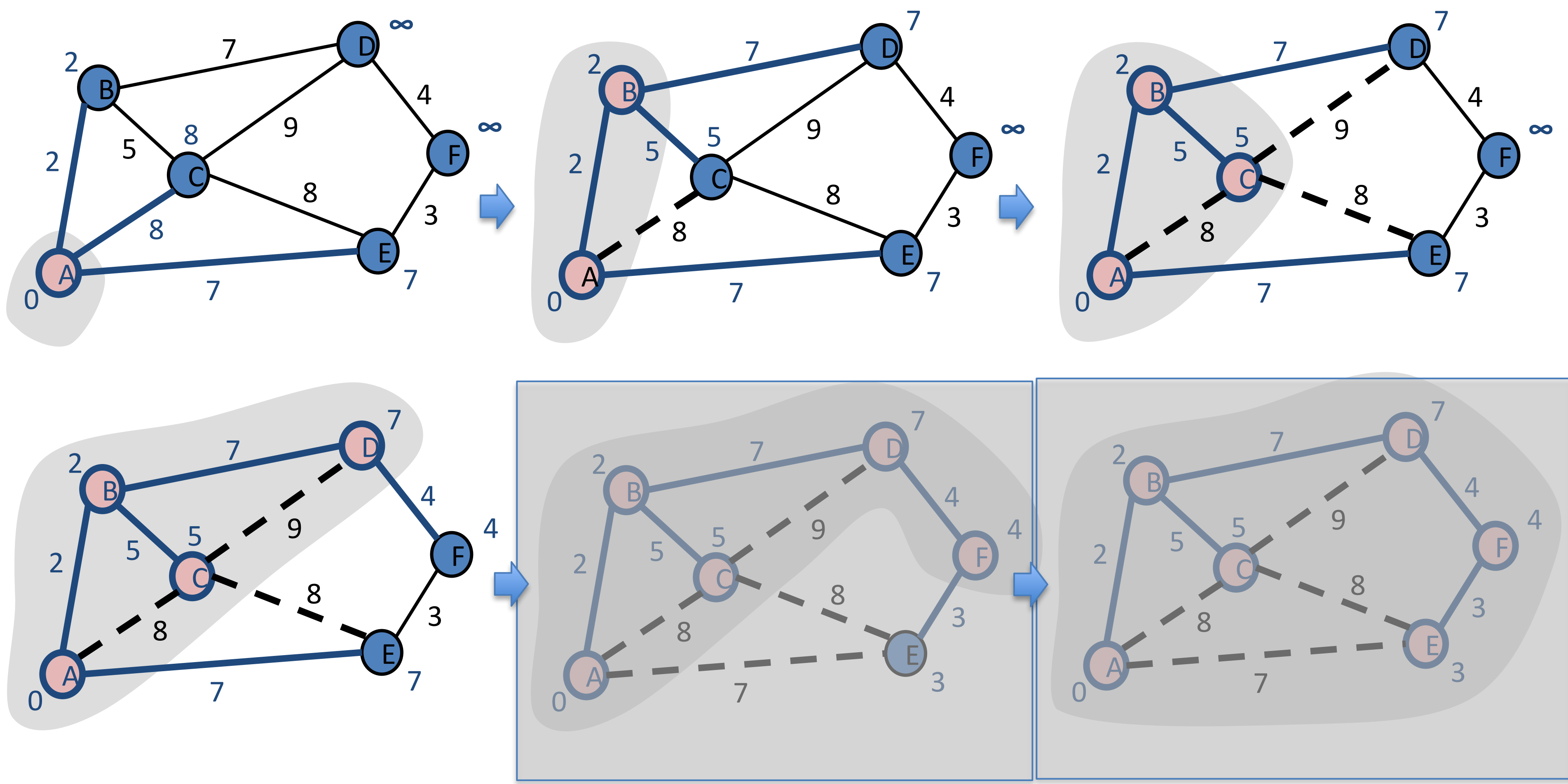
Prim's minimum spanning tree



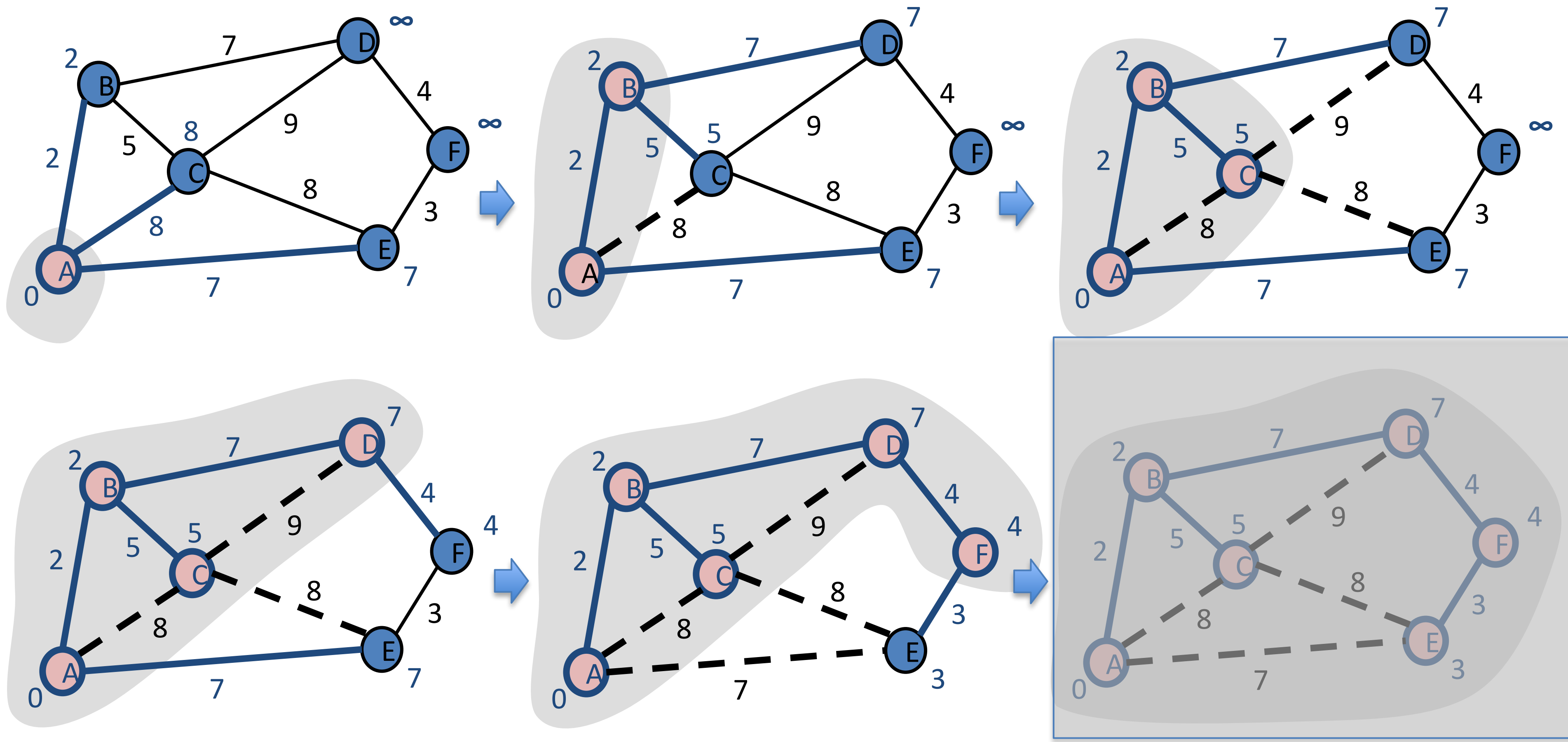
Prim’s minimum spanning tree



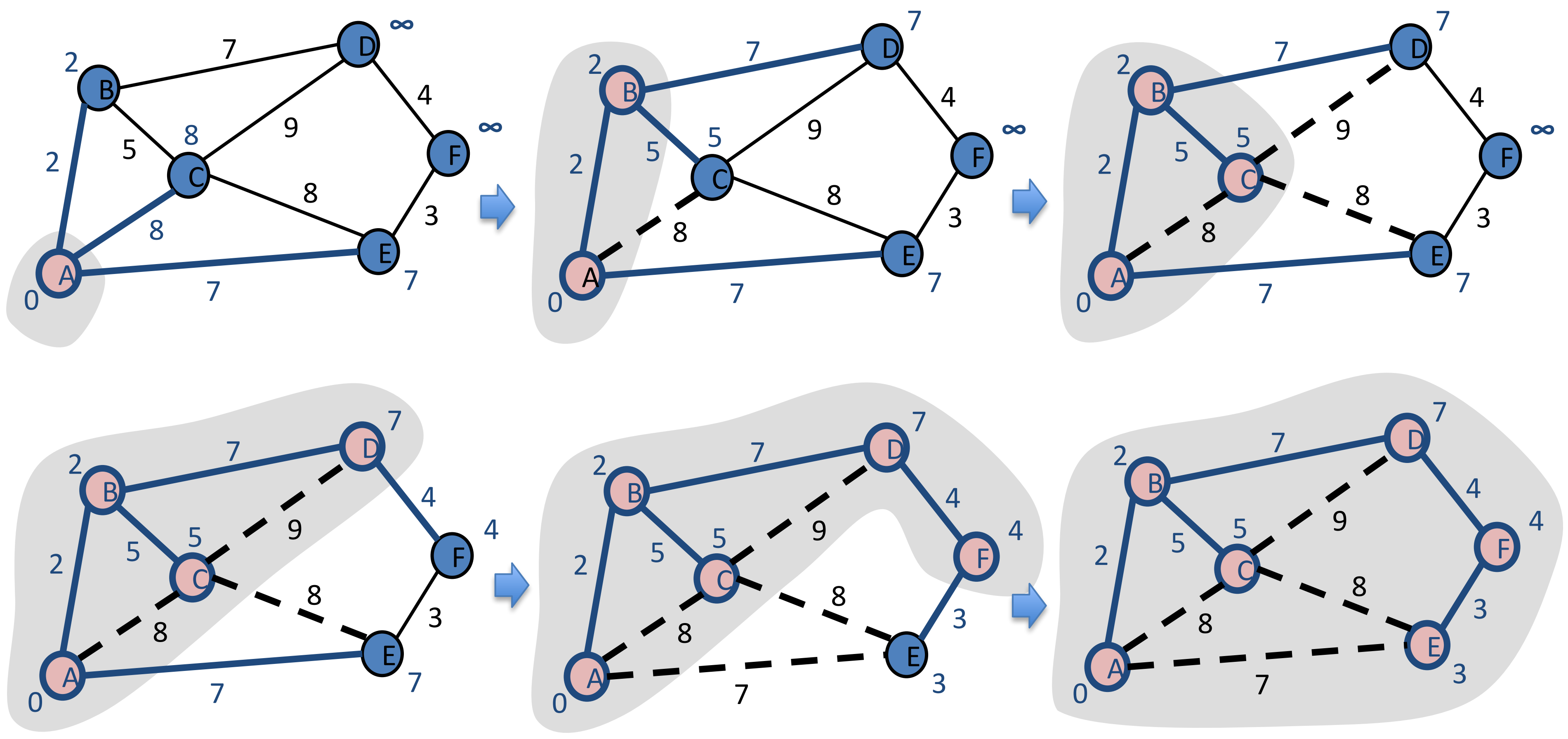
Prim’s minimum spanning tree



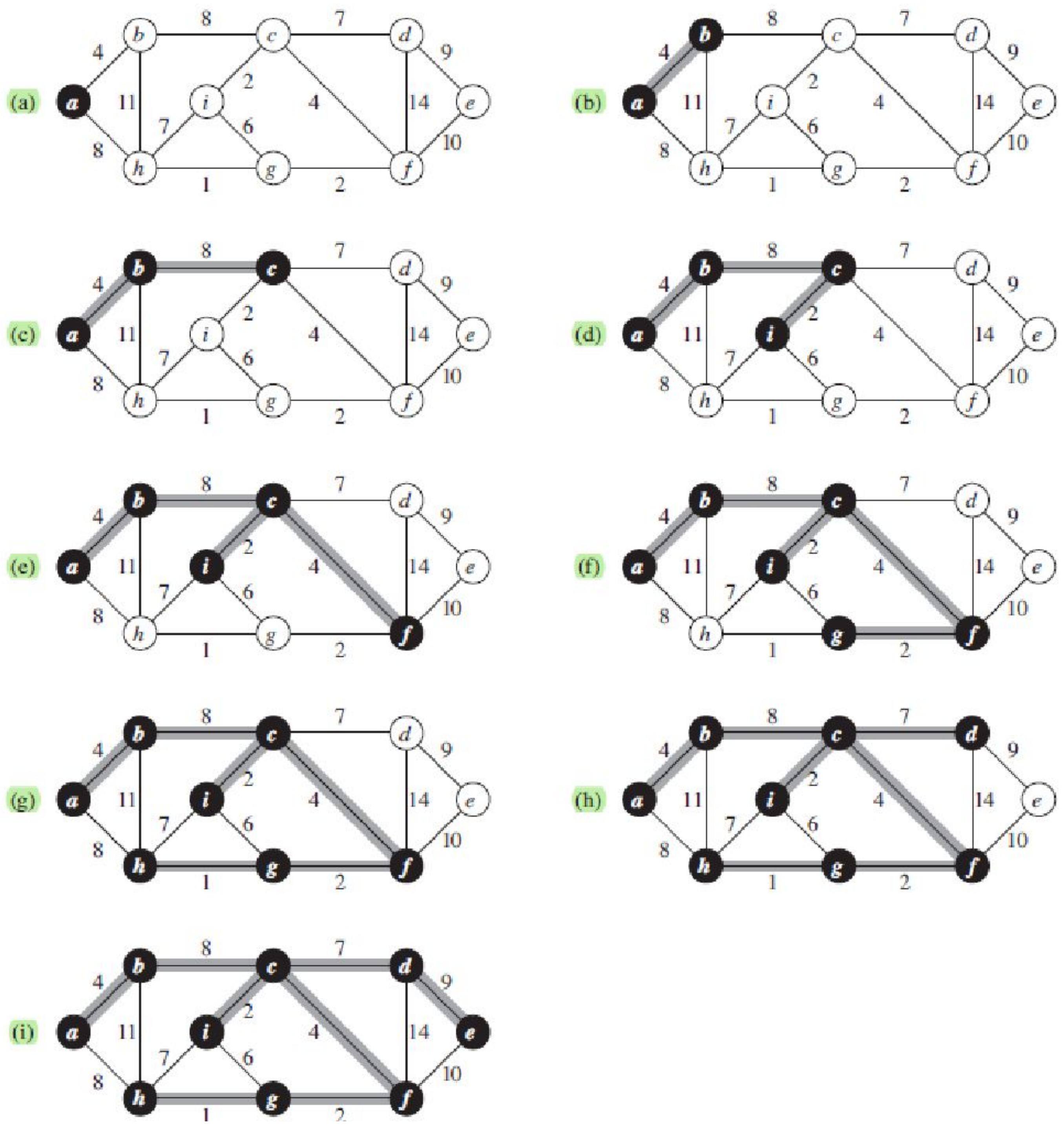
Prim’s minimum spanning tree



Prim’s minimum spanning tree



Prim’s MST algorithm

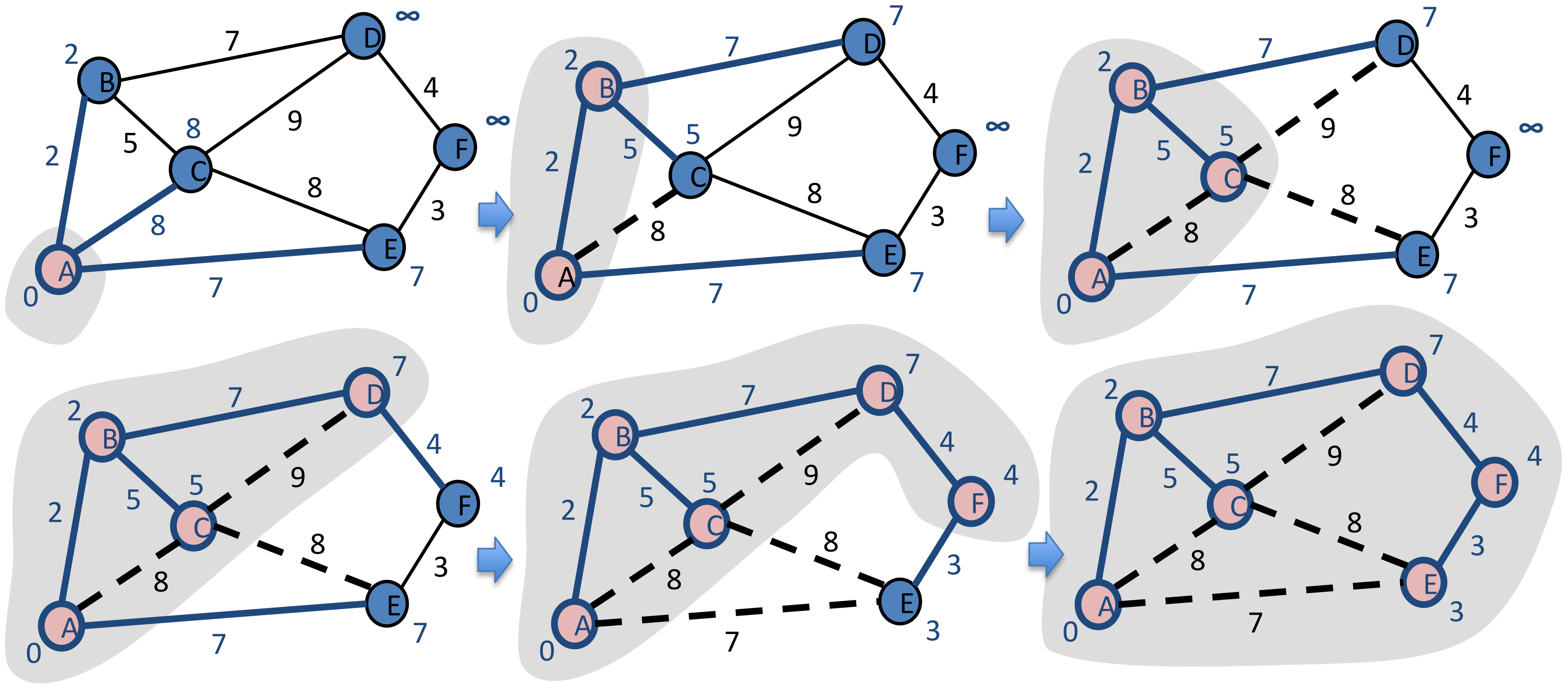


Prim’s MST algorithm

Prim’s algorithm on G = (V, E), with weights of edges in array W = [w(e)]

```
function MST-Prim(G, W, r)                  //G = (V, E)
    T = Φ                                   //list of all edges in MST
    X = Φ                                   // list of all vertices in MST
    for each vertex u ε V - {r}
        u.key = ⍰
        u.π = nil
    r.key = 0
    u.π = nil
    Q = V                                   //sort all in V as a min-binary heap Q based on u.key
    while Q ≠ Φ
        u = Extract-min(Q)                  //and delete the min node in Q
        T = T U {u, u.π}                    //the corresponding edge is added to T
        X = X U {u}                         //the corresponding vertex is added to X
        for each v in G.Adj[u]
            if v in Q and w(u, v) < v.key
                v.π = u                     //change its parent
                v.key = w(u, v)             //update its key
                Adjust-min(Q, v)            //re-adjust the min-heap
return T, X
```

Prim’s minimum spanning tree

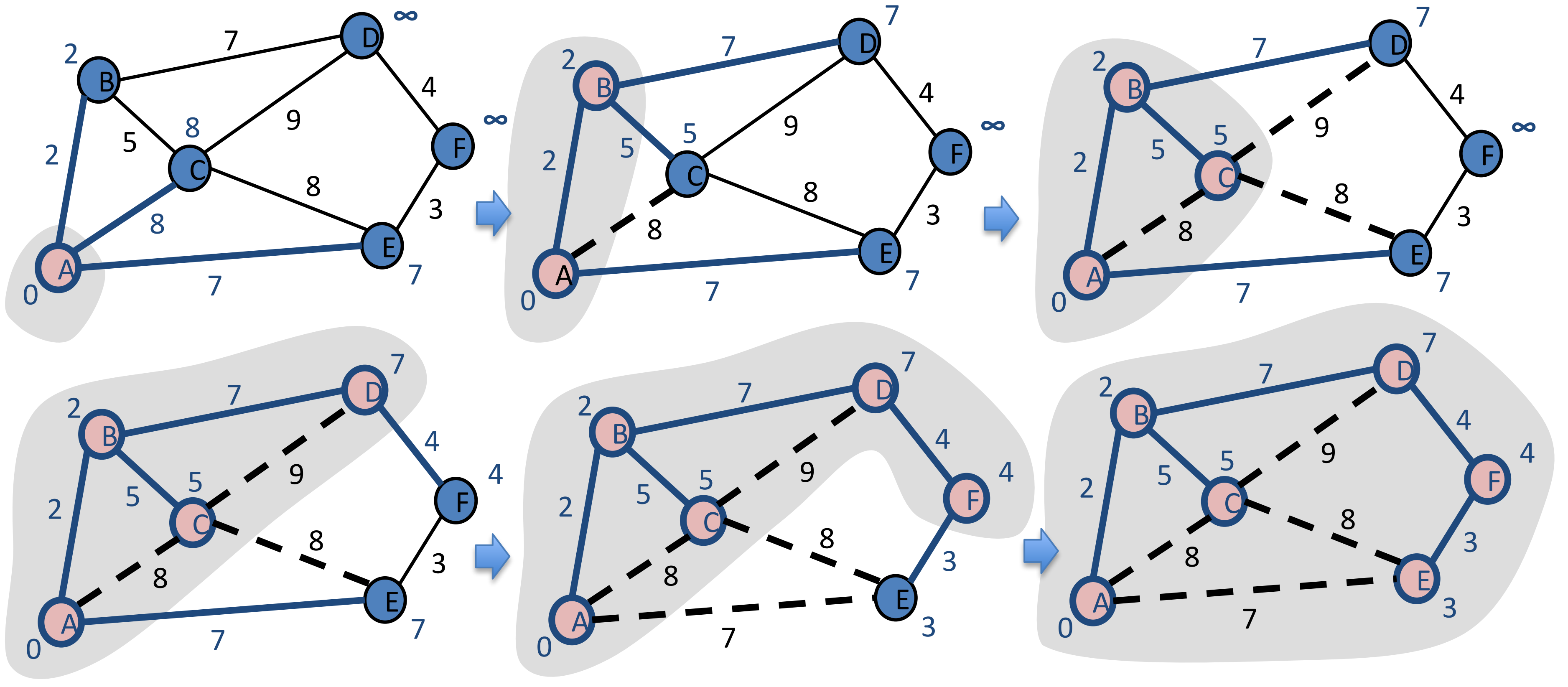


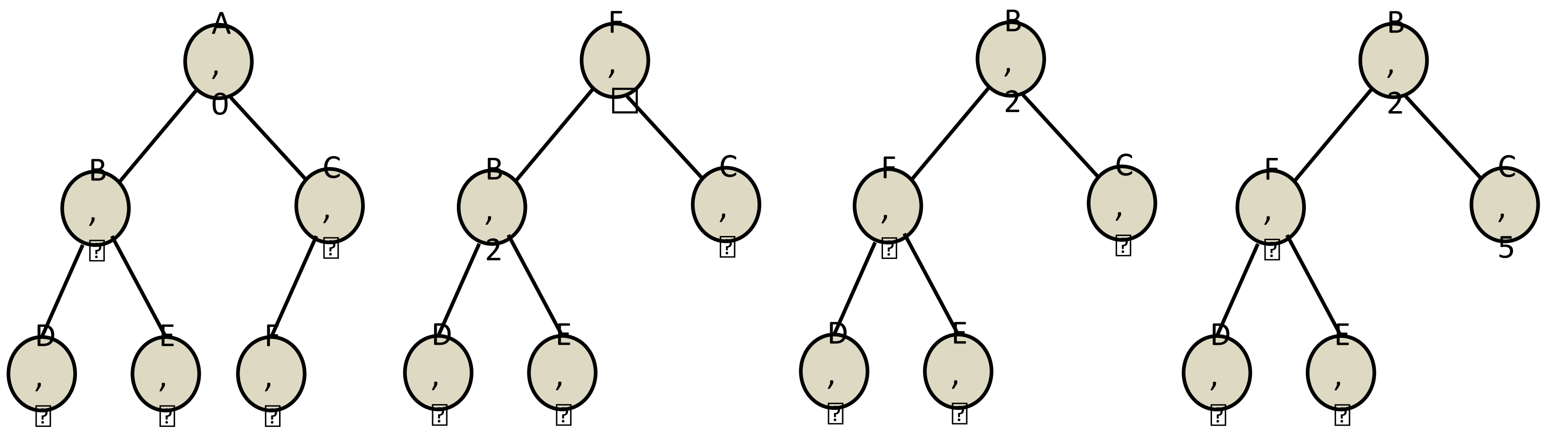
State of min binary heap

Initially:	X = []	T = []
	X = [A]	T = []
++ edge (A, B):	X = [A, B]	T = [(A, B)]
++ edge (B, C):	X = [A, B, C]	T = [(A, B), (B, C)]
++ edge (B, D):	X = [A, B, C, D]	T = [(A, B), (B, C), (B, D)]

Etc.

Prim’s minimum spanning tree





Etc.

Prim’s minimum spanning tree

Time complexity of Prim’s algorithm on G = (V, E), with weights W = [w(e)], n = |V|, m = |E|

```
function MST-Prim(G, W, r)                         //G = (V, E)
    T = Φ                                          //list of all edges in MST
    X = Φ                                          // list of all vertices in MST
    for each vertex u ε V - {r}
        u.key = □
        u.π = nil
    r.key = 0
    u.π = nil
    Q = V                                          //sort all in V as a min-binary heap Q based on u.key
    while Q ≠ Φ
        u = Extract-min(Q)                         //and delete the min node in Q
        T = T U {u, u.π}                           //the corresponding edge is added to T
        X = X U {u}                                //the corresponding vertex is added to X
        for each v in G.Adj[u]
            if v in Q and w(u, v) < v.key
                v.π = u                            //change its parent
                v.key = w(u, v)                    //update its key
                Adjust-min(Q, v)                   //re-adjust the min-heap
return T, X
```

O(1)

O(n)

O(1)

O(n)

O(m log n)

□ Time complexity of Prim’s algorithm: O(|E) log |V|)

Q&A