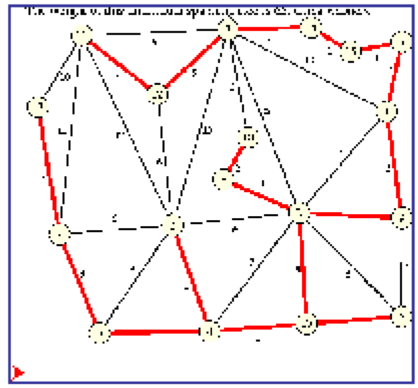
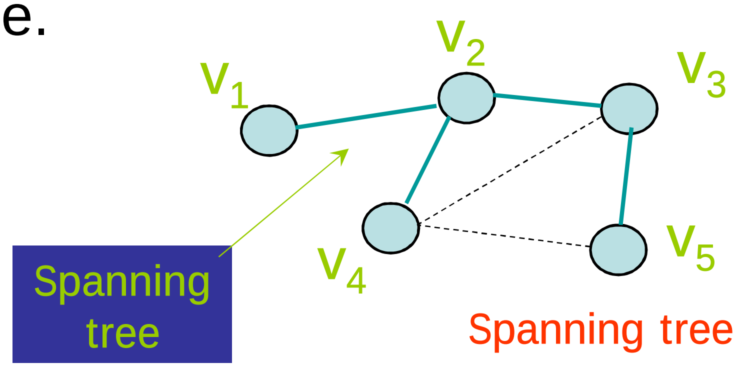
Minimum Spanning Tree



Spanning Tree

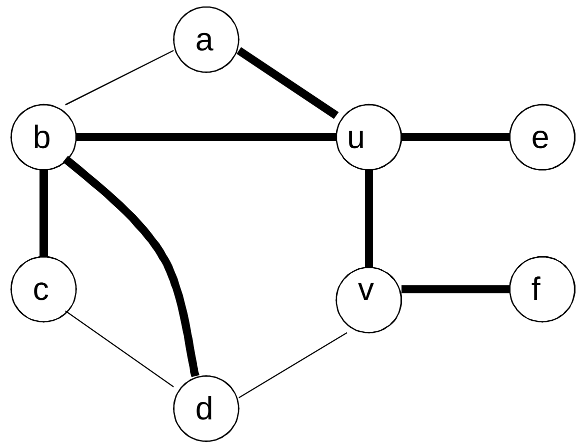
- Given a connected weighted undirected graph G, a spanning tree of G is a subgraph of G that contains all of G's nodes and enough of its edges to form a tree.



Spanning tree is not unique!

What is A Spanning Tree?

- A *spanning* tree for an undirected graph G=(V,E) is a subgraph of G that is a tree and contains all the vertices of G



- Can a graph have more than one spanning tree?

Yes

- Can an unconnected graph have a spanning tree?

No

DFS spanning tree

- Generate the spanning tree edge during the DFS traversal.

Algorithm dfsSpanningTree(v)

mark v as visited;

for (each unvisited node u adjacent to v) {

  mark the edge from u to v;

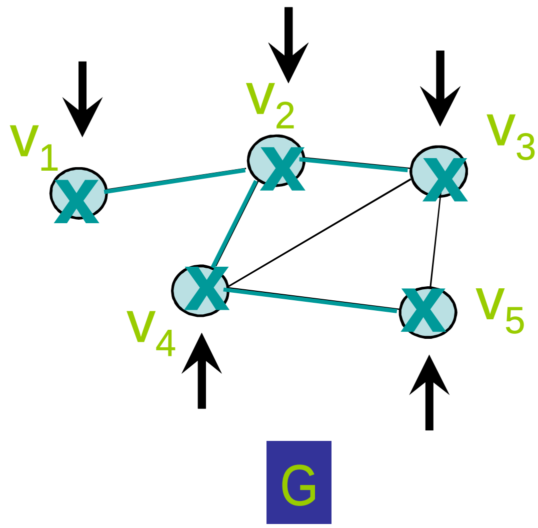
  dfsSpanningTree(u);

}

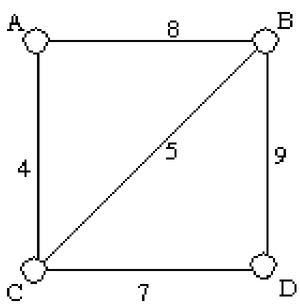
- Similar to DFS, the spanning tree edges can be generated based on BFS traversal.

Example of generating spanning tree based on DFS

	stack
v_3	v_3
v_2	v_3, v_2
v_1	v_3, v_2, v_1
backtrack	v_3, v_2
v_4	v_3, v_2, v_4
v_5	v_3, v_2, v_4 , v_5
backtrack	v_3, v_2, v_4
backtrack	v_3, v_2
backtrack	v_3
backtrack	empty



Spanning Tree



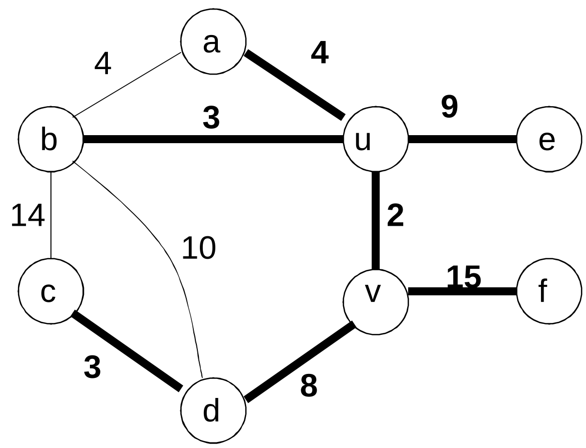
Use BFS and DFS

1. Find a spanning subgraph of G and draw it below.
2. Draw all the different spanning trees of G

Minimal Spanning Tree.

- The *weight* of a subgraph is the sum of the weights of it edges.
- A *minimum spanning tree* for a weighted graph is a spanning tree with minimum weight.
- Can a graph have more then one minimum spanning tree?

Yes, maybe



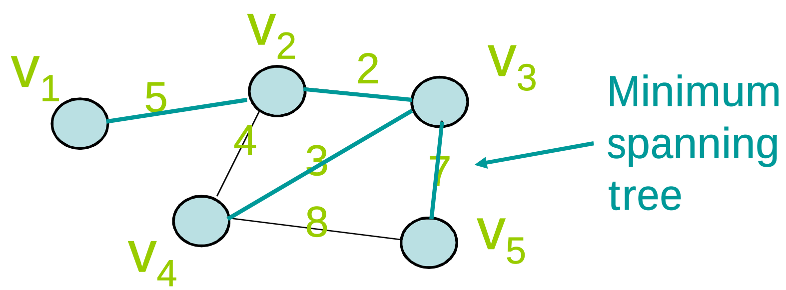
Mst T: $w(T) = \sum_{(u,v) \in T} w(u,v)$ is minimized

Minimum Spanning Tree

- Consider a connected undirected graph where
  - Each node x represents a country x
  - Each edge (x, y) has a number which measures the cost of placing telephone line between country x and country y
- Problem: connecting all countries while minimizing the total cost
- Solution: find a spanning tree with minimum total weight, that is, minimum spanning tree

Formal definition of minimum spanning tree

- Given a connected undirected graph G.
- Let T be a spanning tree of G.
- cost(T) = $\sum_{e \in T}$weight(e)
- The minimum spanning tree is a spanning tree T which minimizes cost(T)



Greedy Choice

We will show two ways to build a minimum spanning tree.

- A MST can be grown from the current spanning tree by adding the nearest vertex and the edge connecting the nearest vertex to the MST. (Prim's algorithm)

- A MST can be grown from a forest of spanning trees by adding the smallest edge connecting two spanning trees. (Kruskal's algorithm)

Notation

- Tree-vertices: in the tree constructed so far
- Non-tree vertices: rest of vertices

Prim’s Selection rule

- Select the minimum weight edge between a tree-node and a non-tree node and add to the tree

The Prim’s algorithm Main Idea

This algorithm starts with one node. It then, one by one, adds a node that is unconnected to the new tree to the new tree, each time selecting the node whose connecting edge has the smallest weight out of the available nodes’ connecting edges.

The steps are:

1. The new tree is constructed - with one node from the old graph.
2. While new tree has fewer than n nodes,
   1. Find the node from the old graph with the smallest connecting edge to the new tree,
   2. Add it to the new tree

Every step will have joined one node, so that at the end we will have one tree with all the nodes and it will be a minimum spanning tree of the original graph.

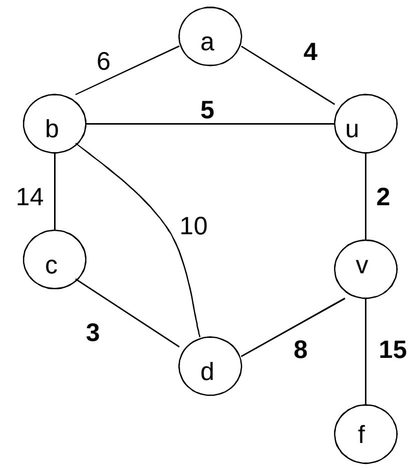
The Prim's algorithm Main Idea

Select a vertex to be a tree-node

`while` (there are non-tree vertices) {
if there is no edge connecting a tree node with a non-tree node
`return` "no spanning tree"

select an edge of minimum weight between a tree node and a non-tree node

add the selected edge and its new vertex to the tree
}
return tree



Prim's algorithm

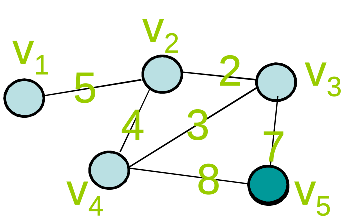
Algorithm PrimAlgorithm(v)

- Mark node v as visited and include it in the minimum spanning tree;
- while (there are unvisited nodes) {
  - find the minimum edge (v, u) between a visited node v and an unvisited node u;
  - mark u as visited;
  - add both v and (v, u) to the minimum spanning tree;

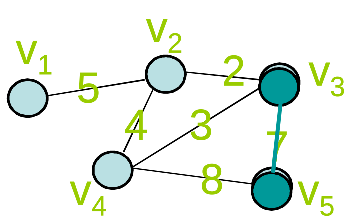
  }

Some Examples

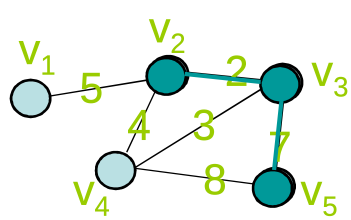
Example #01



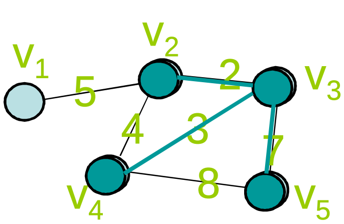
Start from v_5, find the minimum edge attach to v_5



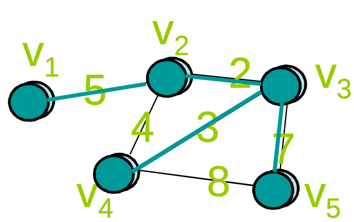
Find the minimum edge attach to v_3 and v_5



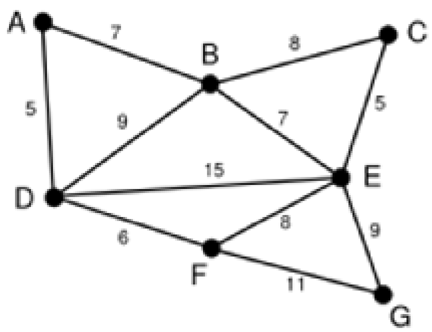
Find the minimum edge attach to v_2, v_3 and v_5



Find the minimum edge attach to v_2, v_3 , v_4 and v_5

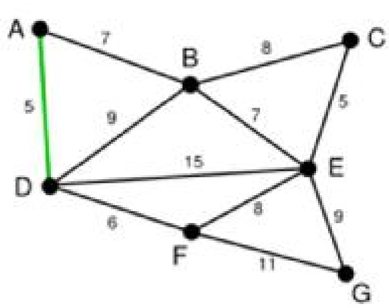


Example #02 - 1

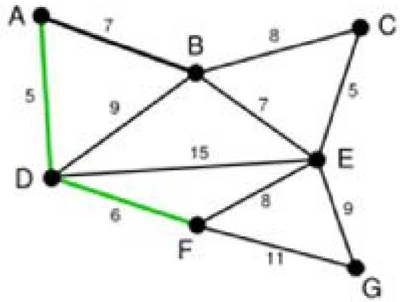


Description	Not seen	Fringe	Solution set
This is our original weighted graph. This is not a tree because the definition of a tree requires that there are no cycles and this diagram contains cycles. A more correct name for this diagram would be a graph or a network. The numbers near the arcs indicate their weight. None of the arcs are highlighted, and vertex **D** has been arbitrarily chosen as a starting point.	C, G	A, B, E, F	D

Example #02 - 2

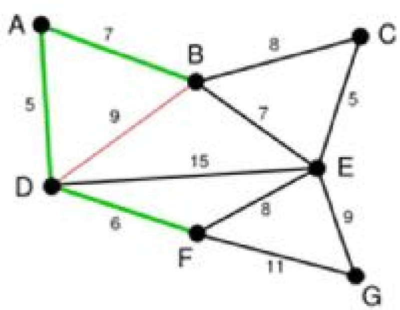


Description	Not seen	Fringe	Solution set
The second chosen vertex is the vertex nearest to **D**: **A** is 5 away, **B** is 9, **E** is 15, and **F** is 6. Of these, 5 is the smallest, so we highlight the vertex **A** and the arc **DA**.	C, G	B, E, F	A, D



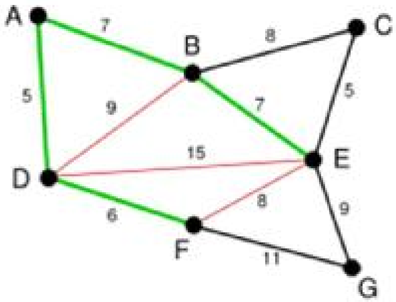
Example #02 - 3

Description	Not seen	Fringe	Solution set
The next vertex chosen is the vertex nearest to *either* **D** or **A**. **B** is 9 away from **D** and 7 away from **A**, **E** is 15, and **F** is 6. 6 is the smallest, so we highlight the vertex **F** and the arc **DF**.	C	B, E, G	A, D, F



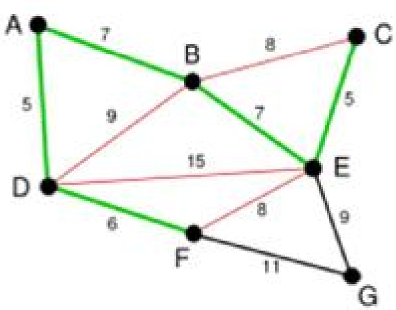
Example #02 - 4

Description	Not seen	Fringe	Solution set
The algorithm carries on as above. Vertex **B**, which is 7 away from **A**, is highlighted. Here, the arc **DB** is highlighted in red, because both vertex **B** and vertex **D** have been highlighted, so it cannot be used.	null	C, E, G	A, D, F, B



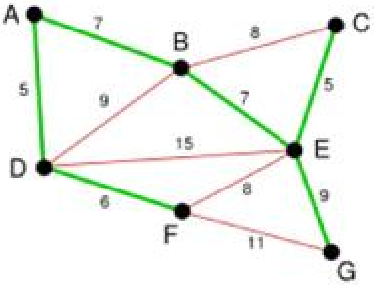
Example #02 - 5

Description	Not seen	Fringe	Solution set
In this case, we can choose between **C**, **E**, and **G**. **C** is 8 away from **B**, **E** is 7 away from **B**, and **G** is 11 away from **F**. **E** is nearest, so we highlight the vertex **E** and the arc **EB**. Two other arcs have been highlighted in red, as both their joining vertices have been used.	null	C, G	A, D, F, B, E



Example #02 - 6

Description	Not seen	Fringe	Solution set
Here, the only vertices available are **C** and **G**. **C** is 5 away from **E**, and **G** is 9 away from **E**. **C** is chosen, so it is highlighted along with the arc **EC**. The arc **BC** is also highlighted in red.	null	G	A, D, F, B, E, C

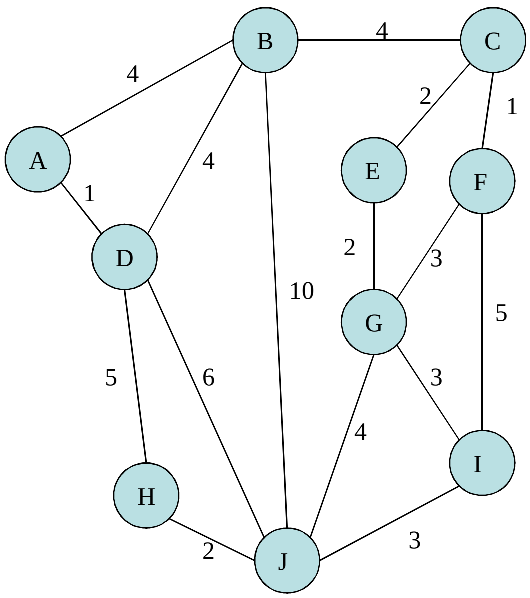


Example #02 - 7

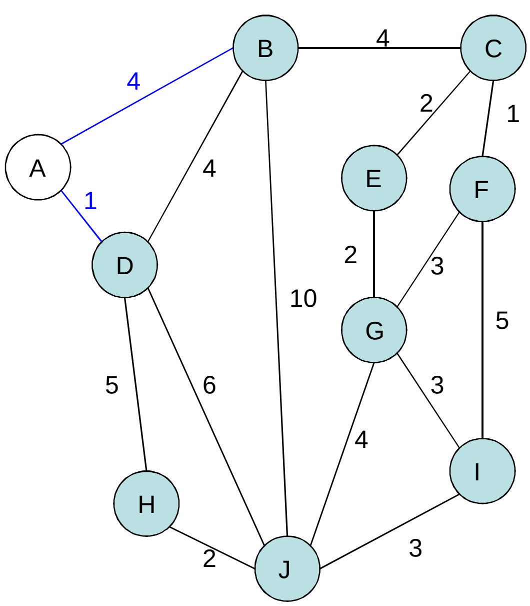
Description	Not seen	Fringe	Solution set
Vertex **G** is the only remaining vertex. It is 11 away from **F**, and 9 away from **E**. **E** is nearer, so we highlight it and the arc **EG**. Now all the vertices have been highlighted, the minimum spanning tree is shown in green. In this case, it has weight 39.	null	null	A, D, F, B, E, C, G

Example #03

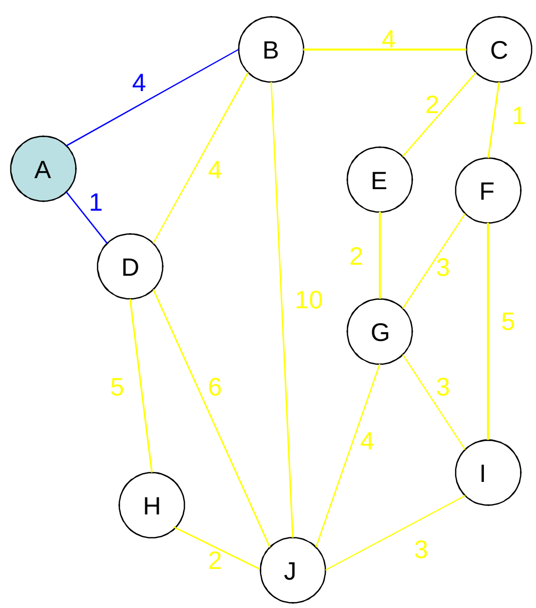
Complete Graph



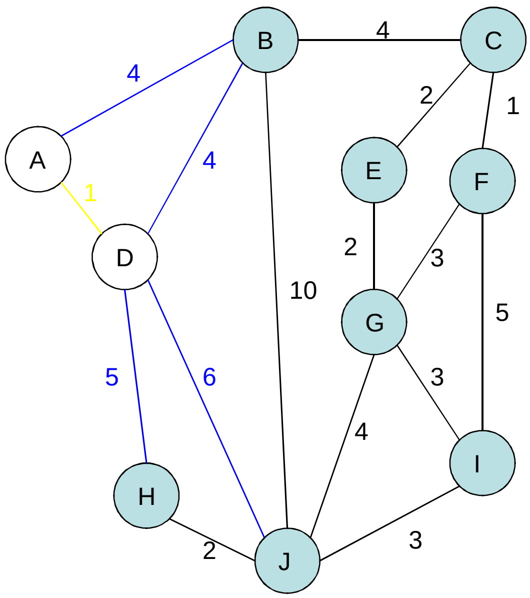
Old Graph



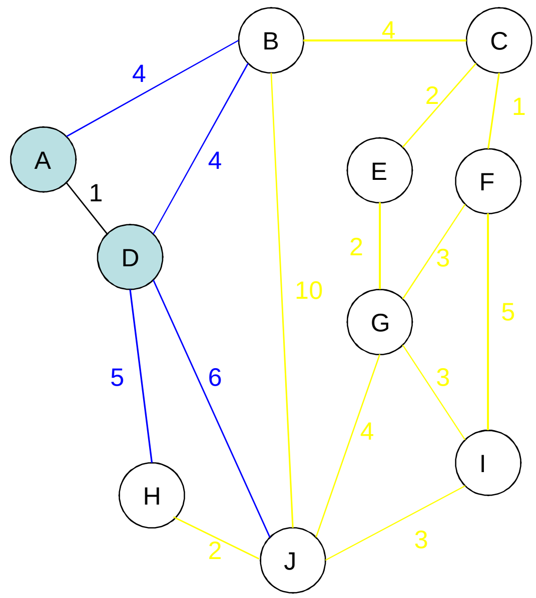
New Tree



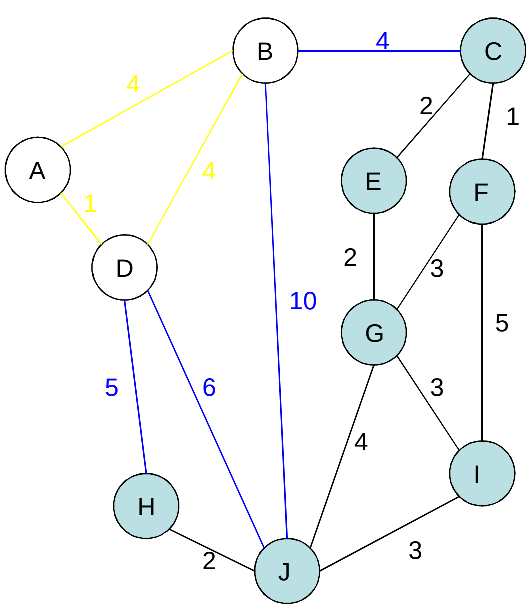
Old Graph



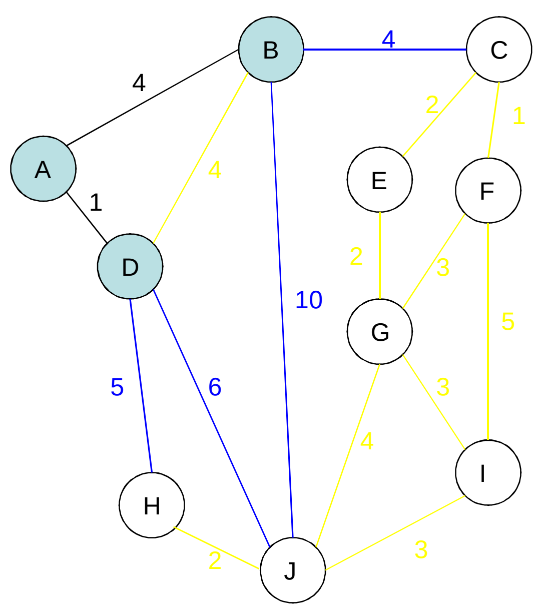
New Tree



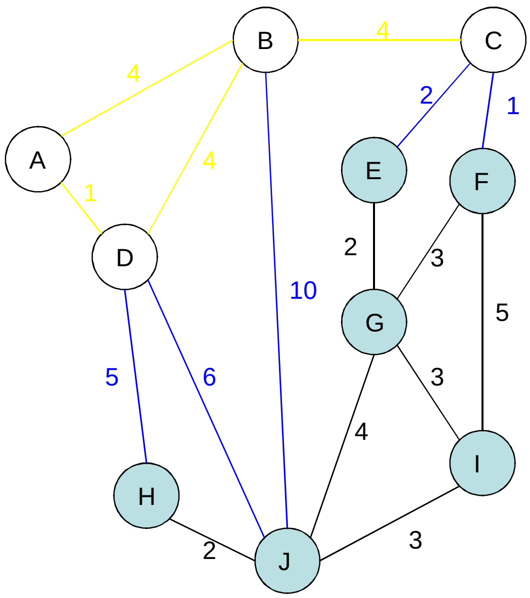
Old Graph



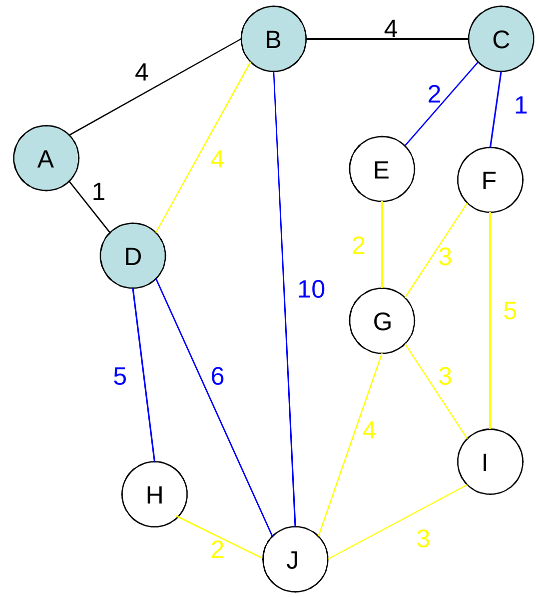
New Tree



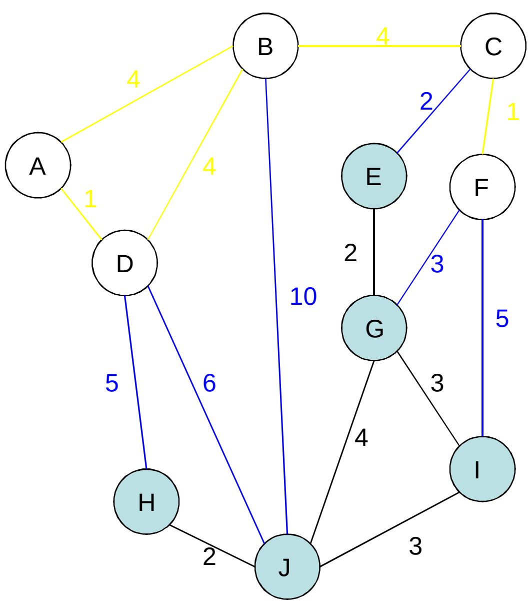
Old Graph



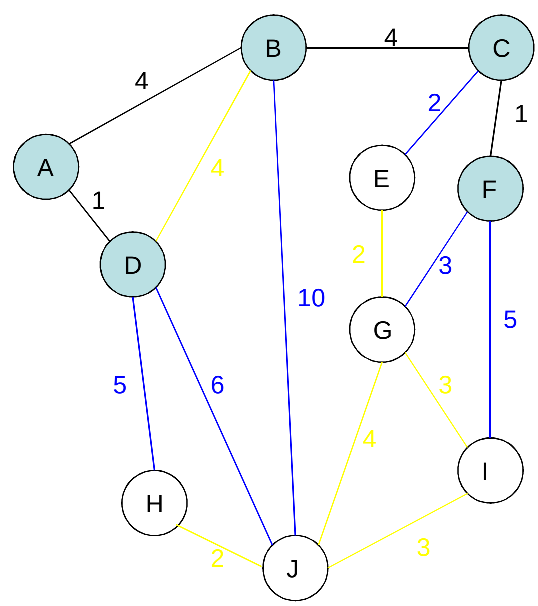
New Tree



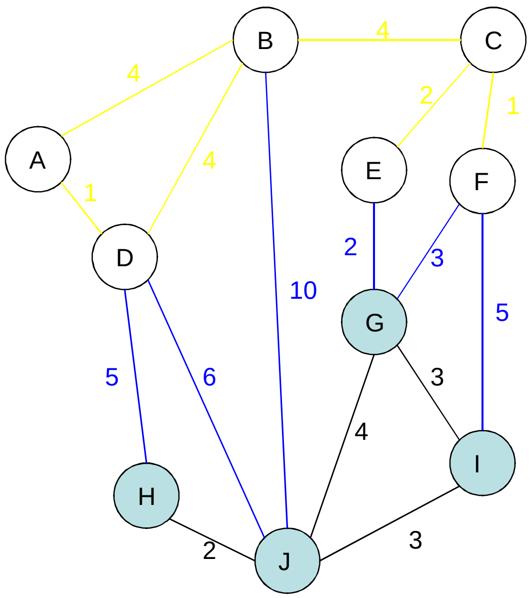
Old Graph



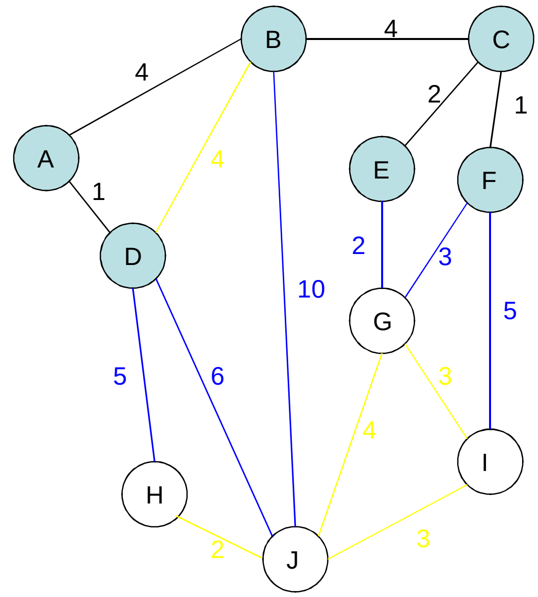
New Tree



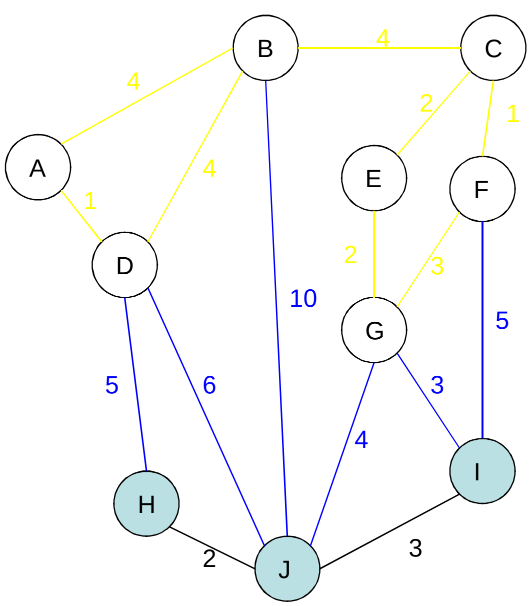
Old Graph



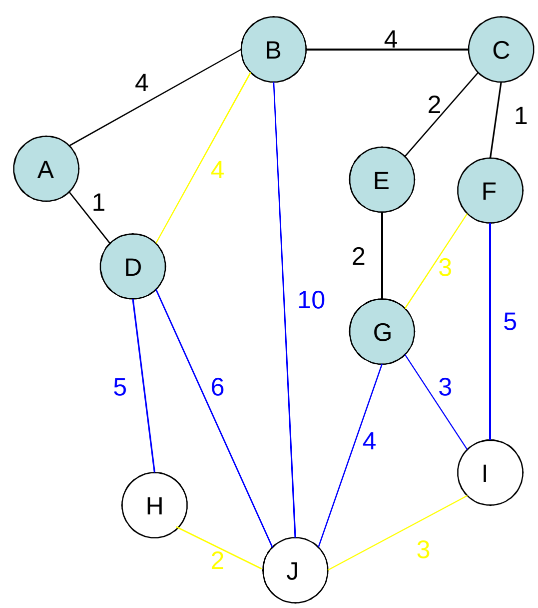
New Tree



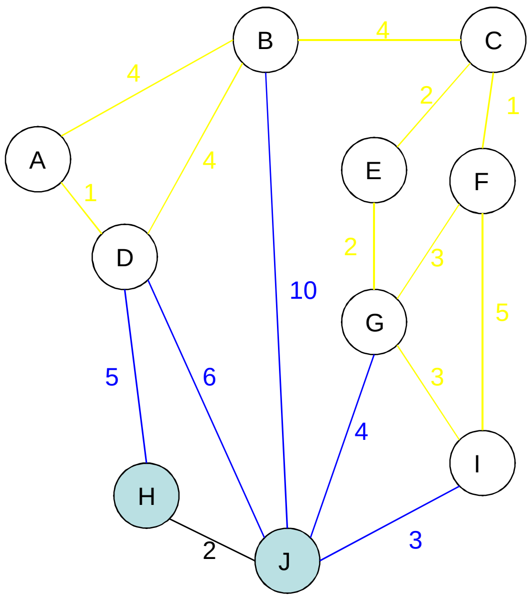
Old Graph



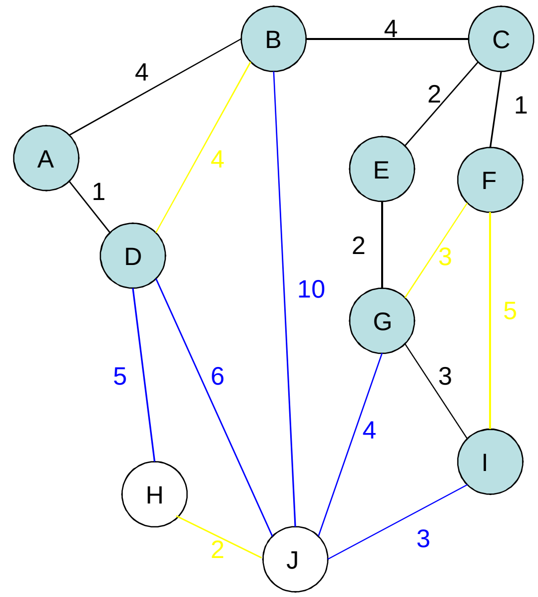
New Tree



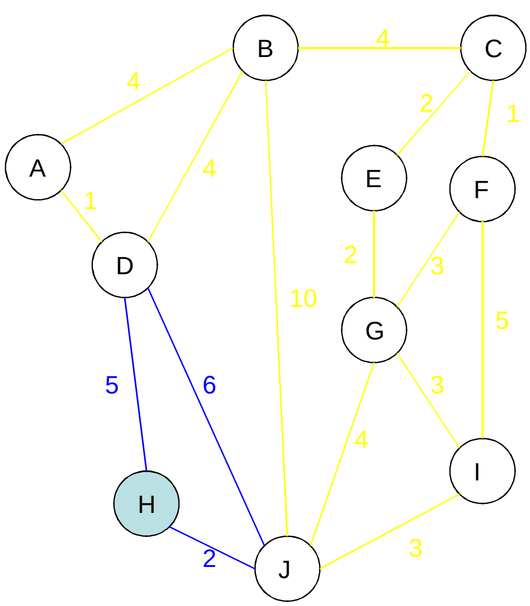
Old Graph



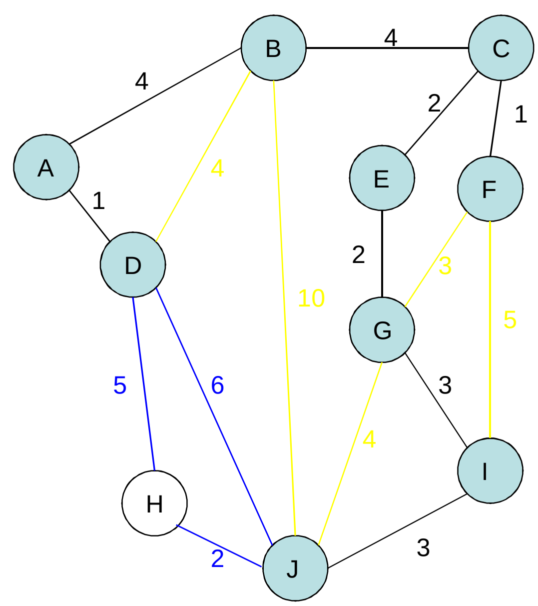
New Tree



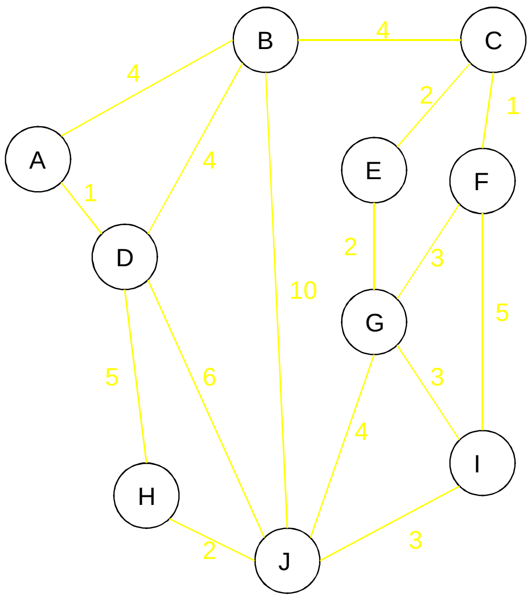
Old Graph



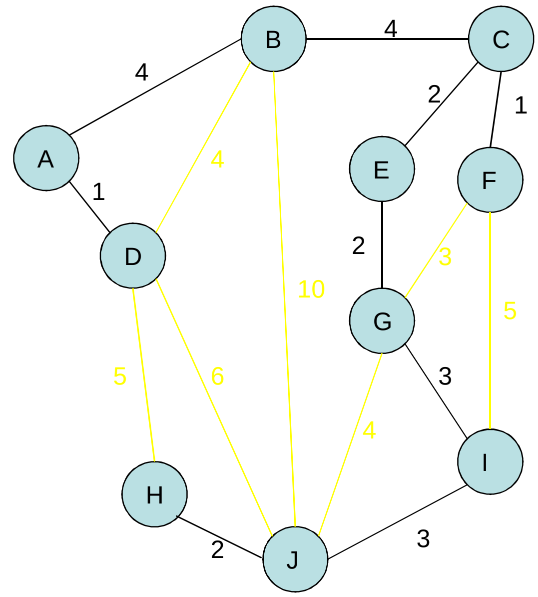
New Tree



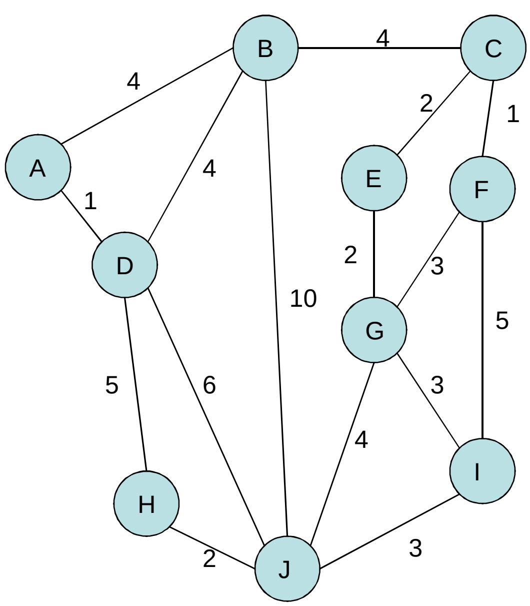
Old Graph



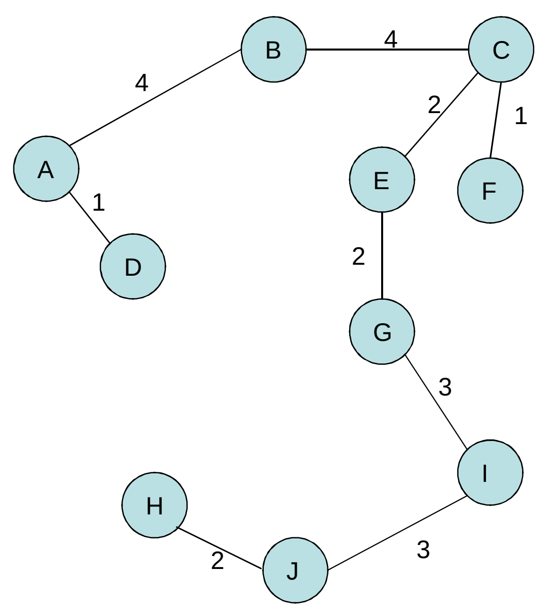
New Tree



Complete Graph



Minimum Spanning Tree



Implementation Issues

- How is the graph implemented?
  - Assume that we just added node u to the tree.
  - The distance of the nodes adjacent to u to the tree may now be decreased.
  - There must be fast access to all the adjacent vertices.
  - So using **adjacency lists** seems better
- How should the set of non-tree vertices be represented?
  - The operations are:
    - build set
    - delete node closest to tree
    - decrease the distance of a non-tree node from the tree
    - check whether a node is a non- tree node

Implementation Issues

- How should the set of non-tree vertices be represented?
  - A priority queue PQ may be used with the priority D[v] equal to the minimum *distance* of each non-tree vertex v to the tree.
  - Each item in PQ contains: D[v], the vertex v, and the shortest distance edge (v, u) where u is a tree node
- This means:
  - build a PQ of non-tree nodes with initial values -
    - fast build heap O **(*V*)**
    - building an unsorted list O(V)
    - building a sorted list O(V) (special case)

Implementation Issues

- delete node closest to tree (extractMin)
  - **O(lg *V*) if heap and**
  - **O(V) if unsorted list**
  - **O(1) sorted list**

- decrease the distance of a non-tree node to the tree
- We need to find the location i of node v in the priority queue and then execute (decreasePriorityValue(i, p)) where p is the new priority
- decreasePriorityValue(i, p)
  - O(lg V) for heap,
  - O(1) for unsorted list
  - O(***V*** **)** for sorted list (too slow)

Implementation Issues

- What is the location i of node v in a priority queue?
  - Find in Heaps, and sorted lists O(n)
  - Unsorted – if the nodes are numbered 1 to n and we use an array where node v is the v item in the array O(1)

Extended heap

- We will use extended heaps that contain a “handle” to the location of each node in the heap.
- When a node is not in PQ the “handle” will indicate that this is the case
- This means that we can access a node in the extended heap in O(1), and check $v \in PQ$ in O(1)
- Note that the “handle” must be updated whenever a heap operation is applied

Implementation Issues

2. Unsorted list
   – Array implementation where node v can be accesses as PQ[v] in O(1), and the value of PQ[v] indicates when the node is not in PQ.

Prim's Algorithm

1. **for** each $u \in V$
2. **do** $D[u] \leftarrow \infty$
3. $D[r] \leftarrow 0$
4. $PQ \leftarrow$make-heap(D,V, {})//No edges
5. $T \leftarrow \emptyset$
6.
7. **while** $PQ \neq \emptyset$ **do**
8. $(u,e) \leftarrow PQ$.extractMin()
9. add (u,e) to T
10. **for** each $v \in$ Adjacent (u)
    // execute relaxation
11. **do if** $v \in PQ$ && $w(u, v) < D[v]$
12. **then** $D[v] \leftarrow w(u, v)$
13. PQ.decreasePriorityValue (D[v], v, (u,v))
14. **return** T // T is a mst.

Lines 1-5 initialize the priority queue P*Q* to contain all Vertices. Ds for all vertices except *r*, are set to infinity.
r is the starting vertex of the *T*
The *T* so far is empty

Add closest vertex and edge to current *T*

Get all adjacent vertices v of u, update *D of each* non-tree vertex adjacent to u

Store the current minimum weight edge, and updated distance in the priority queue

Prim's Algorithm Initialization

Prim (G)

1. **for** each $u \in V$
2. **do** $D[u] \leftarrow \infty$
3. $D[r] \leftarrow 0$
4. $PQ \leftarrow$ make-heap(D,V, {})//No edges
5. $T \leftarrow \emptyset$

Building the *MST*

// solution check

7. **while** $PQ \neq \emptyset$ **do**

//Selection and feasibility

8. (u,e) $\leftarrow$ *PQ*.extractMin()

// T contains the solution so far .

9. add (u,e) to T
10. **for** each $v \in$ Adjacent (u)
11. **do if** $v \in PQ$ && $w(u, v) < D[v]$
12. **then** $D[v] \leftarrow w(u, v)$
13. PQ.decreasePriorityValue (D[v], v, (u,v))
14. **return** T

Time Analysis

1. **for** each $u \in V$
2. **do** $D[u] \leftarrow \infty$
3. $D[r] \leftarrow 0$
4. $PQ \leftarrow$make-PQ(D,V, {})//No edges
5. $T \leftarrow \varnothing$
6.
7. **while** $PQ \neq \varnothing$ **do**
8. (u,e) $\leftarrow$ PQ.extractMin()
9. add (u,e) to T
10. **for** each $v \in$ Adjacent (u)
11. **do if** $v \in PQ$ && $w(u, v) < D[v]$
12. **then** $D[v] \leftarrow w(u, v)$
13. PQ.decreasePriorityValue (D[v], v, (u,v))

15. **return** T // T is a mst.

Assume a node in PQ can be accessed in $O(1)$

** Decrease key for v requires O(lg$V$) provided the node in heap with v's data can be accessed in O(1)

Using Extended Heap implementation

Lines 1 -6 run in $O(V)$

Max Size of PQ is | V |

Count$_7$ $= O(V)$

Count$_{7(8)}$ $=$ O $(V) * O(\lg V)$

Count$_{7(10)}$ $=$ O(Σdeg(u)) $= O(E)$

Count$_{7(10(11))}$ $=$ O(1)$*$O(E)

Count$_{7(10(11(12)))}$ $=$ O(1) $*$O(E)

Count$_{7(10(13))}$ $= O(\lg V) *$ O(E) Decrease-Key operation on the extended heap can be implemented in $O(\lg V)$

So total time for Prim's Algorithm is $O(V \lg V + E \lg V)$

What is O(E) ?

Sparse Graph, E =O(V) , $O(E \lg V)$=O($V \lg V$)

Dense Graph, E=O(V^2), $O(E \lg V)$=O($V^2 \lg V$)

Time Analysis

1. **for** each $u \in V$
2. **do** $D[u] \leftarrow \infty$
3. $D[r] \leftarrow 0$
4. $PQ \leftarrow$make-PQ(D,V, {})//No edges
5. $T \leftarrow \varnothing$
6.
7. **while** $PQ \neq \varnothing$ **do**
8. (u,e) $\leftarrow$ PQ.extractMin()
9. add (u,e) to T
10. **for** each $v \in$ Adjacent (u)
11. **do if** $v \in PQ$ && $w(u, v) < D[v]$
12. **then** $D[v] \leftarrow w(u, v)$
13. PQ.decreasePriorityValue (D[v], v, (u,v))

15. **return** T // T is a mst.

Using unsorted PQ

Lines 1 - 6 run in $O(V)$

Max Size of PQ is | V |

Count$_7$ $= O(V)$

Count$_{7(8)}$ $=$ O $(V) * O(V)$

Count$_{7(10)}$ $=$ O(Σdeg(u)) $= O(E)$

Count$_{7(10(11))}$ $=$ O(1)$*$O(E)

Count$_{7(10(11(12)))}$ $=$ O(1) $*$O(E)

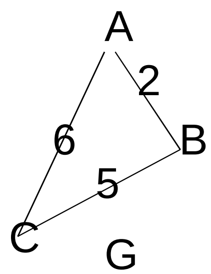
Count$_{7(10(13))}$ $= O(1) *$ O(E)

So total time for Prim's Algorithm is $O(V + V^2 + E) = O(V^2)$

For Sparse/Dense graph : O(V^2)

Note growth rate unchanged for adjacency matrix graph representation

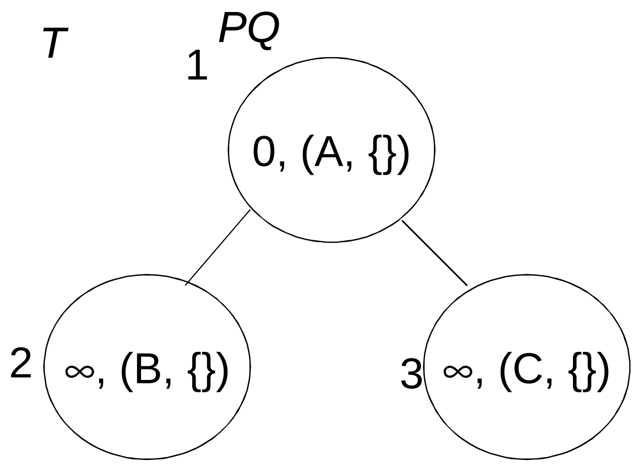
Prim - extended Heap
After Initialization



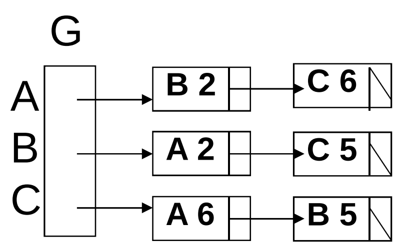
handle

A	1
B	2
C	3

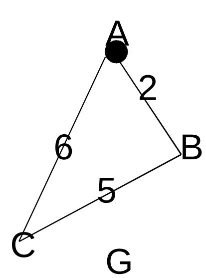
T



Prim (G, r)
1. for each $u \in V$
2. do $D[u] \leftarrow \infty$
3. $D[r] \leftarrow 0$
4. $PQ \leftarrow$make-heap(D,V, { })
5. $T \leftarrow \emptyset$



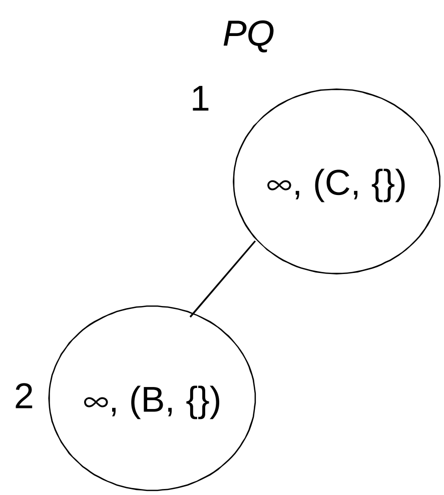
Prim - extended Heap
Build tree - after PQ.extractMin



handle

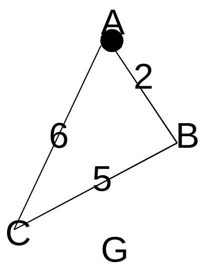
A	Null
B	2
C	1

T
(A, {})



7. **while** $PQ \neq \emptyset$ do
8. (u,e) $\leftarrow$ PQ.extractMin()
9. add (u,e) to T

Update B adjacent to A

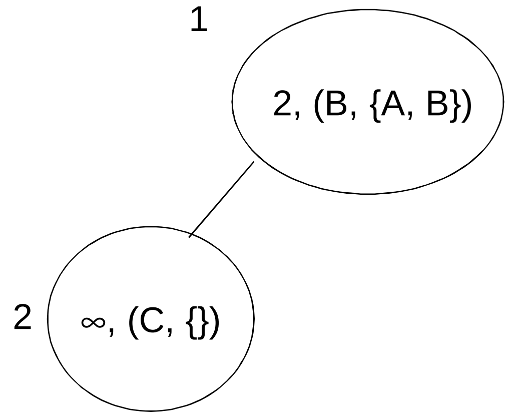


handle

A	Null
B	1
C	2

T
(A, {})

PQ



10. **for** each $v \in$ Adjacent (u)
11. // relaxation operation

// relaxation
11. **do if** $v \in PQ$ && $w(u, v) < D[v]$
12. **then** $D[v] \leftarrow w(u, v)$
13. PQ.decreasePriorityValue (D[v], v, (u,v))

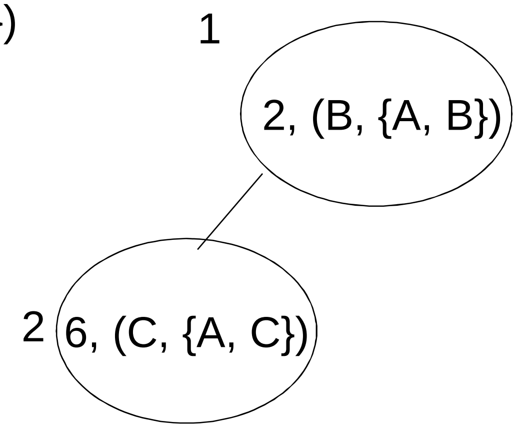
Update C adjacent to A

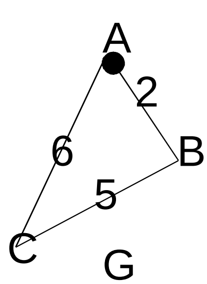
handle

A	Null
B	1
C	2

T
(A, {})

PQ



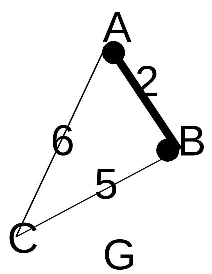


Build tree - after PQ.extractMin

	handle
A	Null
B	Null
C	1

T
(A, {})
(B, {A, B})

PQ
1 6, (C, {A, C})



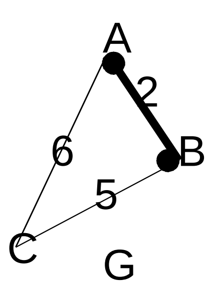
while *PQ* $\neq \varnothing$ do
(*u*,*e*) $\leftarrow$ *PQ*.extractMin()
add (*u*,*e*) to *T*

Update C adjacent to B

	handle
A	Null
B	Null
C	1

T
(A, {})
(B, {A, B})

PQ
1 5, (C, {B, C})



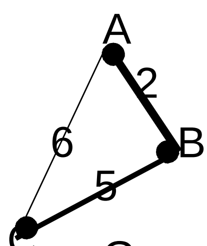
for each $v \in$ Adjacent (u)
// relaxation operation

Build tree - after PQ.extractMin

	handle
A	Null
B	Null
C	Null

T
(A, {})
(B, {A, B})
(C, {B, C})

PQ



G

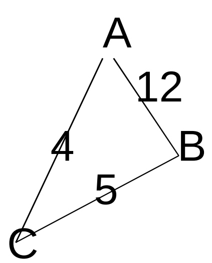
7. **while** $PQ \neq \emptyset$ do
8. $(u,e) \leftarrow PQ.$extractMin()
9.   add (u,e) to T

Prim - unsorted list After Initialization

T

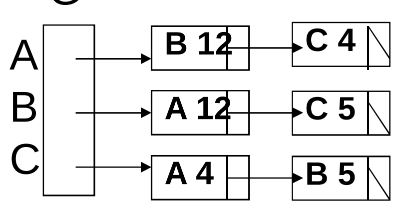
PQ

A	B	C
0, (A, {})	∞, (B, {})	∞, (C, {})



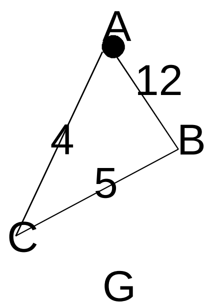
G

G



Prim (G, r)
1. for each $u \in V$
2.   do $D[u] \leftarrow \infty$
3. $D[r] \leftarrow 0$
4. $PQ \leftarrow$make-PQ(D,V, { })
5. $T \leftarrow \emptyset$

Build tree - after PQ.extractMin



T

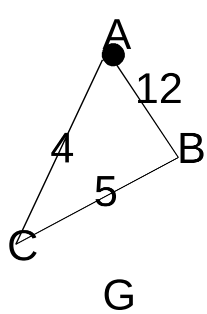
(A, {})

PQ

A	B	C
Null	∞, (B, {})	∞, (C, {})

```
while PQ ≠ ∅ do
   (u,e) ← PQ.extractMin()
               add (u,e) to T
```

Update B, C adjacent to A



T

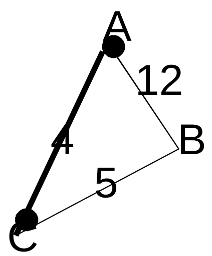
(A, {})

PQ

A	B	C
Null	12, (B, {A, B})	4, (C, {A, C})

```
       for each v ∈ Adjacent (u )
          // relaxation operation
```

Build tree - after PQ.extractMin



G

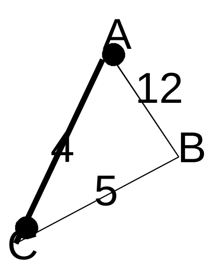
T
(A, {})
(C, {A, C})

PQ

A	B	C
Null	12, (B, {A, B})	Null

while $PQ \neq \emptyset$ do
    $(u,e) \leftarrow PQ$.extractMin()
        add (u,e) to T

Update B adjacent to C



G

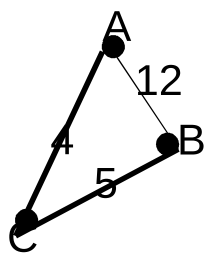
T
(A, {})
(C, {A, C})

PQ

A	B	C
Null	5, (B, {C, B})	Null

for each $v \in$ Adjacent (u)
    // relaxation operation

Build tree - after PQ.extractMin



G

T
(A, {})
(C, {A, C})
(B, {C, B})

PQ

A	B	C
Null	Null	Null

7. **while** $PQ \neq \emptyset$ do
8. $(u,e) \leftarrow PQ.$extractMin()
9. add (u,e) to T

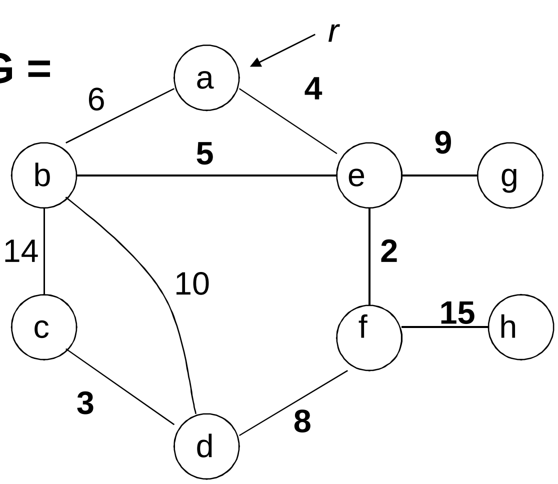
Prim (G)
1. for each $u \in V$
2. do $D[u] \leftarrow \infty$
3. $D[r] \leftarrow 0$
4. $PQ \leftarrow$make-heap(D,V, { })
5. $T \leftarrow \emptyset$

D = [0, ∞, …, ∞]

PQ = {(0,(a,∗)), (∞,(b,?)), ...(∞,(h,?))}

T = { }

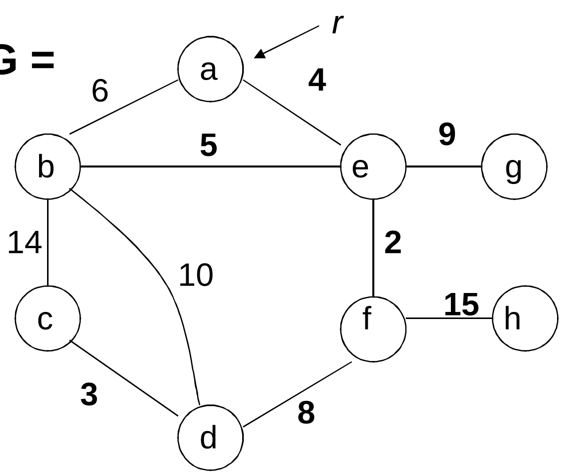
G =



```
7.  while  PQ ≠ ∅ do
8.      (u,e) ← PQ.extractMin()
9.              add (u,e) to T
10.     for each v ∈ Adjacent (u )
11.         // relaxation operation
15. return T
```

```
// relaxation
11.  do if v ∈ PQ  &&  w( u, v ) < D [ v ]
12.      then D [ v ]  ← w (u, v)
13.          PQ.decreasePriorityValue
                 ( D[v], v, (u,v))
```

G =



$D = [\, 0,$

$T = \{$

$PQ = \{$

Analysis of Prim's Algorithm

Running Time = O(m + n log n) (m = edges, n = nodes)
Acer Aspire 4920G-301G16Mi (008)
If a heap is not used, the run time will be O(n^2) instead of O(m + n log n). However, using a heap complicates the code since you're complicating the data structure. A Fibonacci heap is the best kind of heap to use, but again, it complicates the code.

Unlike Kruskal's, it doesn't need to see all of the graph at once. It can deal with it one piece at a time. It also doesn't need to worry if adding an edge will create a cycle since this algorithm deals primarily with the nodes, and not the edges.

For this algorithm the number of nodes needs to be kept to a minimum in addition to the number of edges. For small graphs, the edges matter more, while for large graphs the number of nodes matters more.

Kruskal's Algorithm: Main Idea

This algorithm creates a forest of trees. Initially the forest consists of n single node trees (and no edges). At each step, we add one edge (the cheapest one) so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected tree, so that this edge would not be needed.

Kruskal's Algorithm: Main Idea

The steps are:

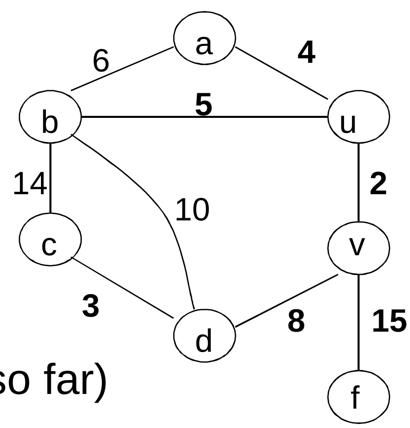
1. The forest is constructed - with each node in a separate tree.
2. The edges are placed in a priority queue.
3. Until we've added n-1 edges,
    1. Extract the cheapest edge from the queue,
    2. If it forms a cycle, reject it,
    3. Else add it to the forest. Adding it to the forest will join two trees together.

Every step will have joined two trees in the forest together, so that at the end, there will only be one tree in T.

Kruskal's algorithm

- **Step 1:** Find the cheapest edge in the graph (if there is more than one, pick one at random). Mark it with any given colour, say red.
- **Step 2:** Find the cheapest unmarked (uncoloured) edge in the graph that doesn't close a coloured or red circuit. Mark this edge red.
- **Step 3:** Repeat Step 2 until you reach out to every vertex of the graph (or you have N ; 1 coloured edges, where N is the number of Vertices.) The red edges form the desired minimum spanning tree.

Kruskal's Algorithm: Main Idea

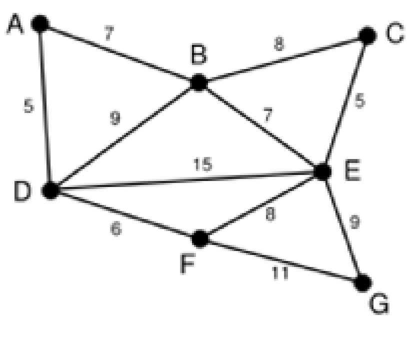


solution = { }
while (more edges in *E*) **do**
 // Selection
select minimum weight edge
remove *edge* from *E*
 // Feasibility
if (*edge* closes a cycle with *solution* so far)
  then reject *edge*
  else add *edge* to *solution*
// Solution check
if |*solution*| = |*V* | - 1 **return** *solution*

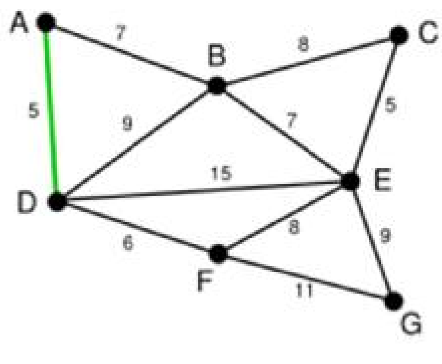
return null // when does this happen?

Some Examples

Example #01

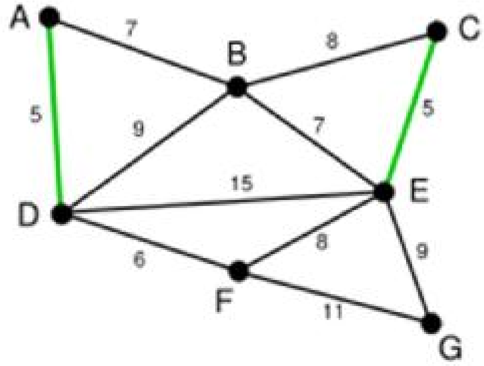


This is our original graph. The numbers near the arcs indicate their weight. None of the arcs are highlighted.

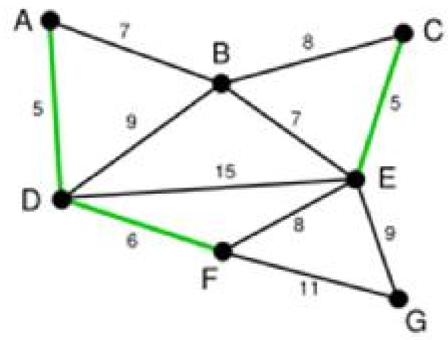


AD and **CE** are the shortest arcs, with length 5, and **AD** has been chosen, so it is highlighted.

Example #01

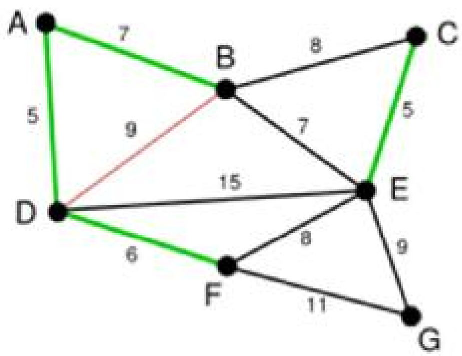


However, **CE** is now the shortest arc that does not form a cycle, with length 5, so it is highlighted as the second arc.

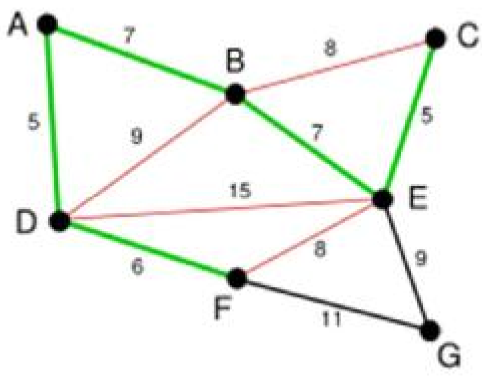


The next arc, **DF** with length 6, is highlighted using much the same method.

Example #01

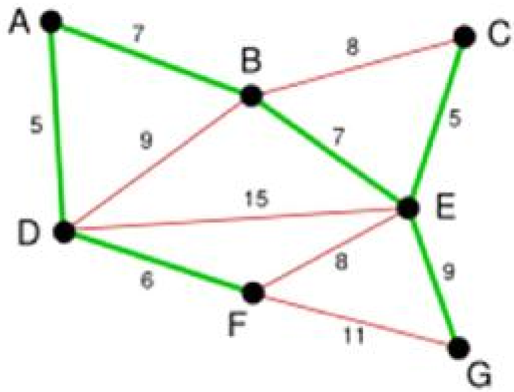


The next-shortest arcs are **AB** and **BE**, both with length 7. **AB** is chosen arbitrarily, and is highlighted. The arc **BD** has been highlighted in red, because it would form a cycle **ABD** if it were chosen.



The process continues to highlight the next-smallest arc, **BE** with length 7. Many more arcs are highlighted in red at this stage: **BC** because it would form the loop **BCE**, **DE** because it would form the loop **DEBA**, and **FE** because it would form **FEBAD**.

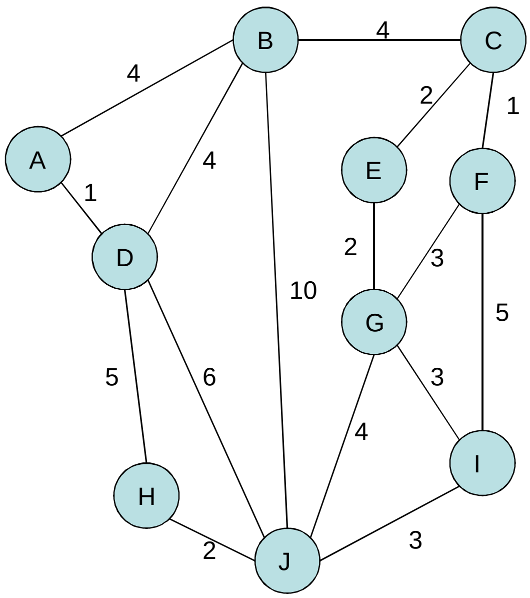
Example #01

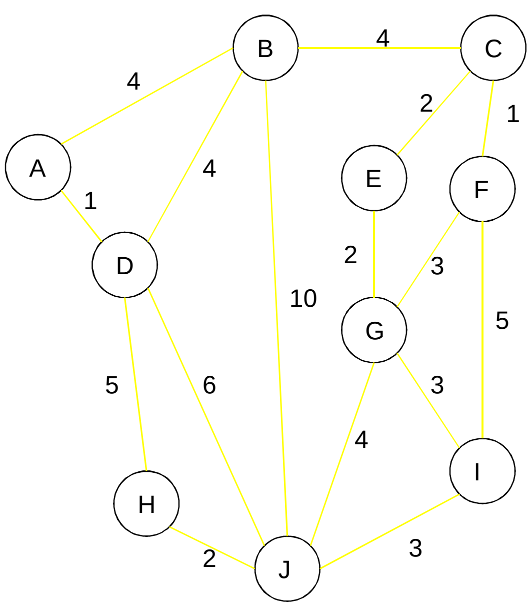


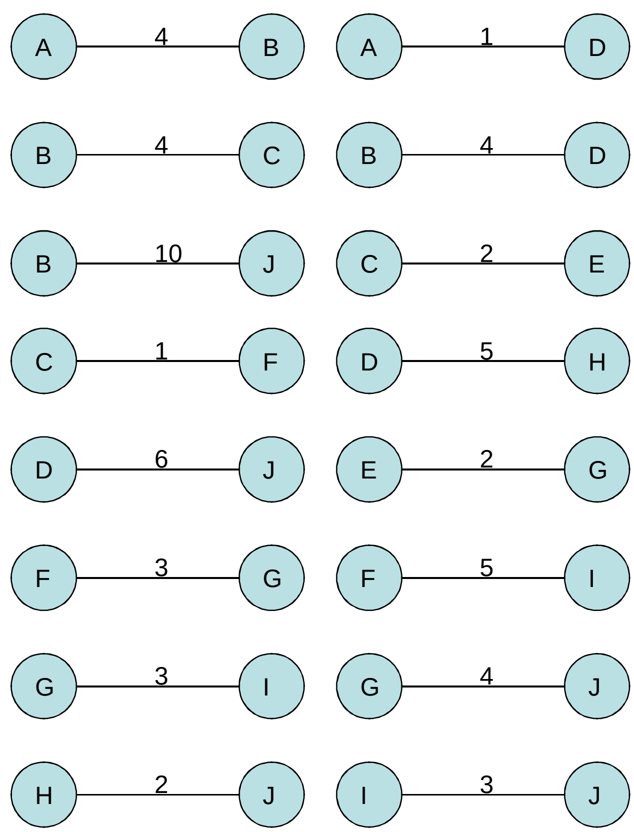
Finally, the process finishes with the arc **EG** of length 9, and the minimum spanning tree is found.

Example #02

Complete Graph

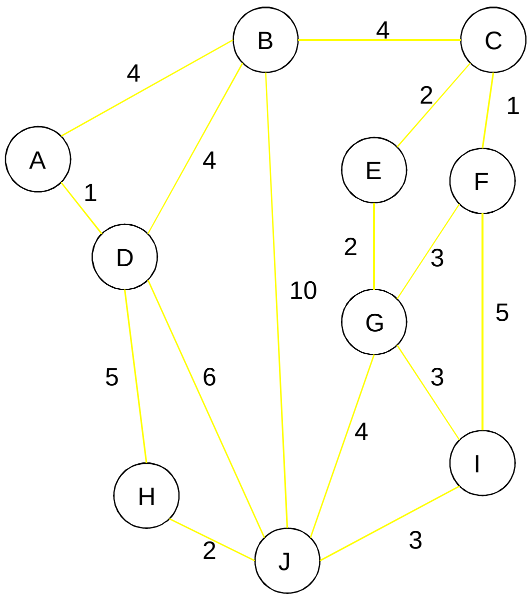


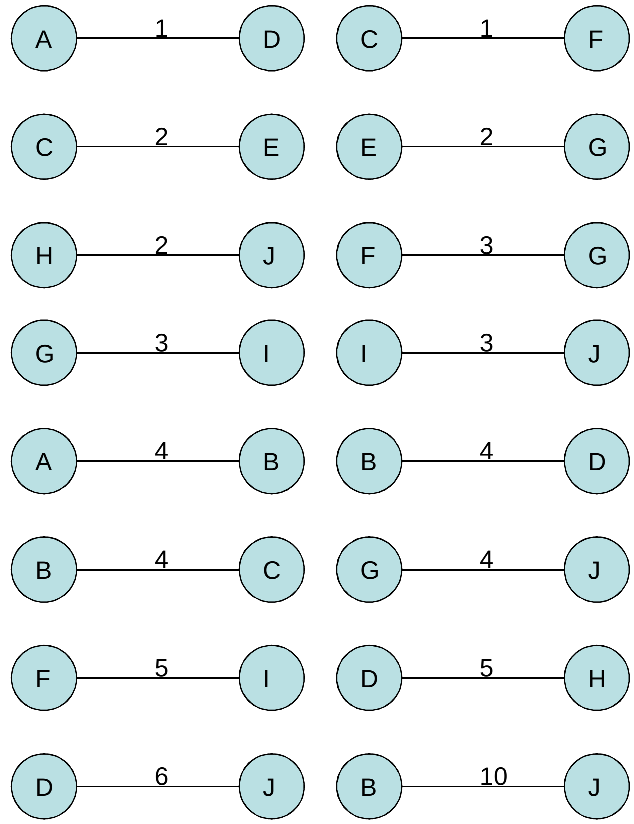




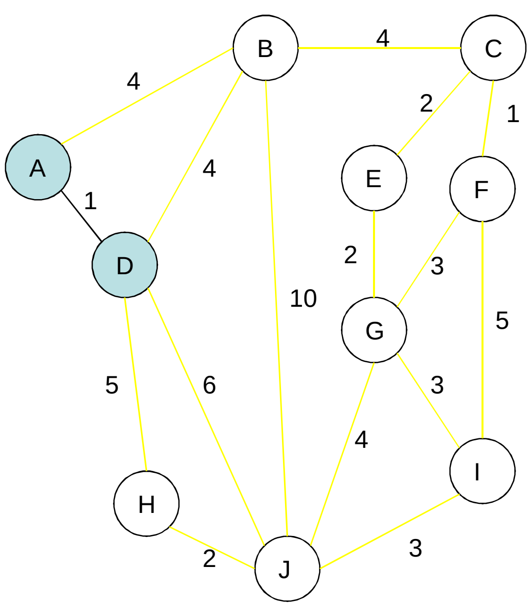
Sort Edges

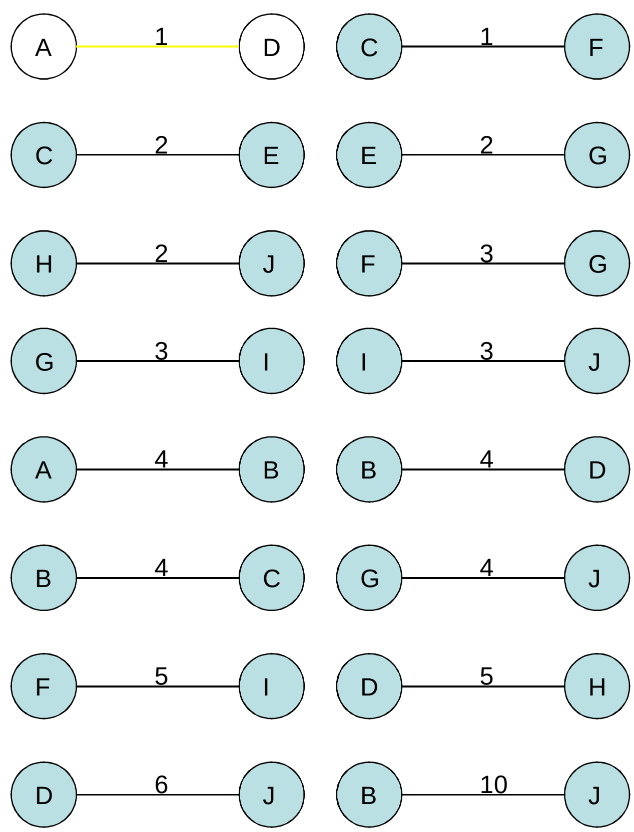
(in reality they are placed in a priority queue - not sorted - but sorting them makes the algorithm easier to visualize)



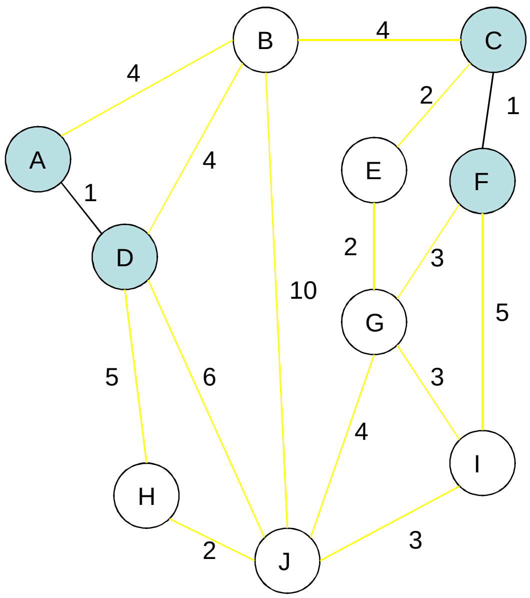


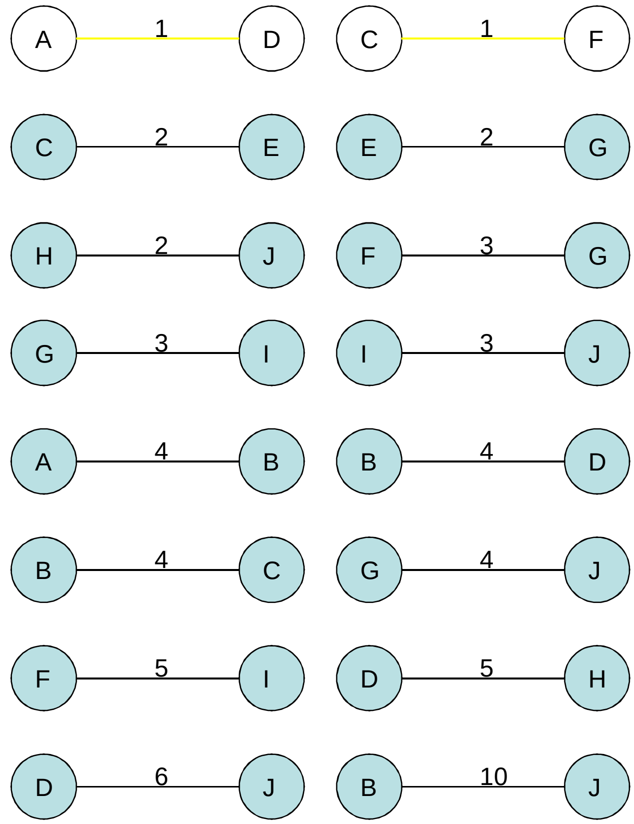
Add Edge



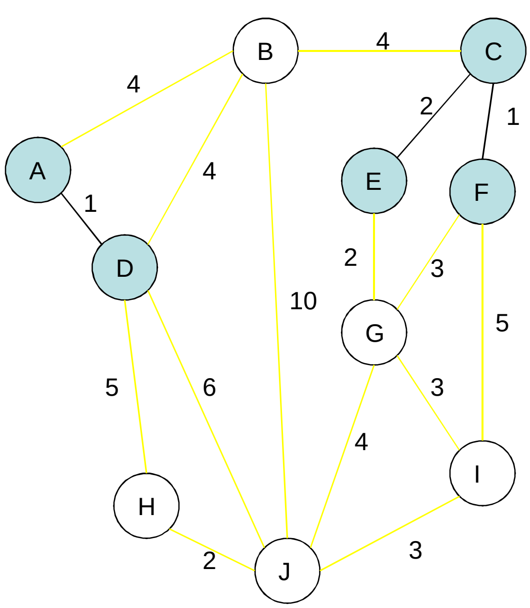


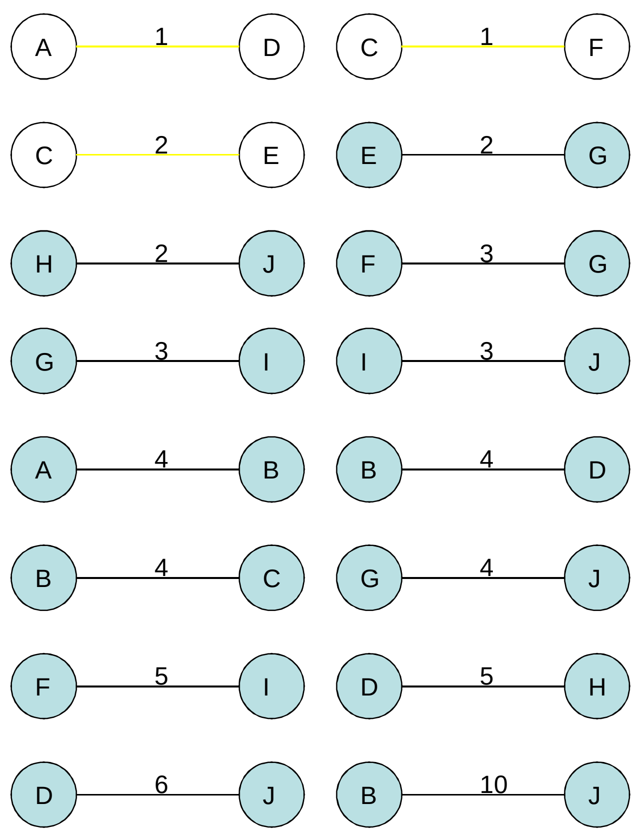
Add Edge



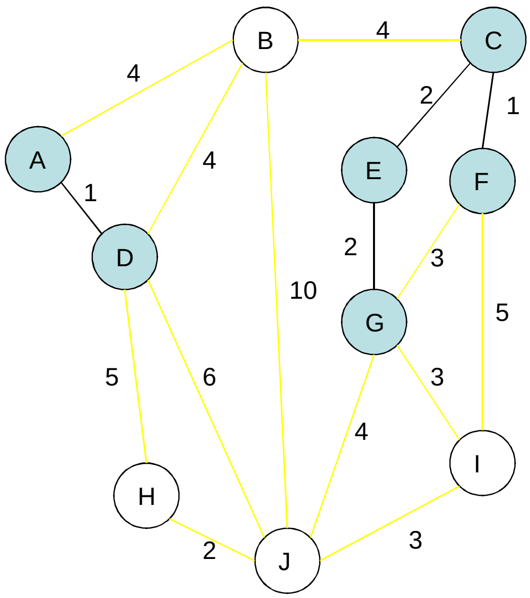


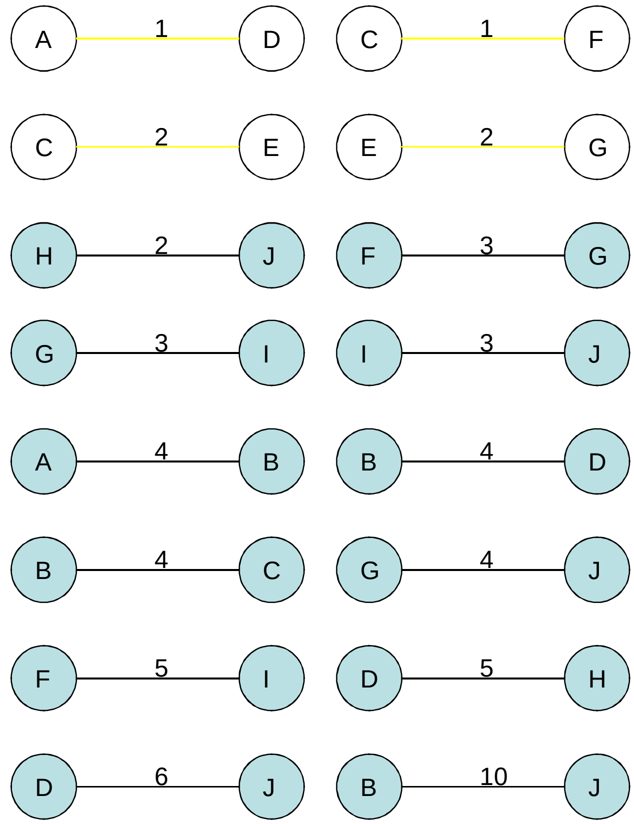
Add Edge



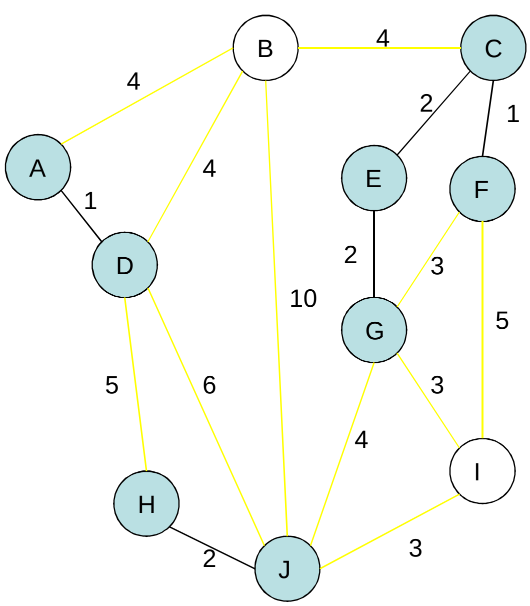


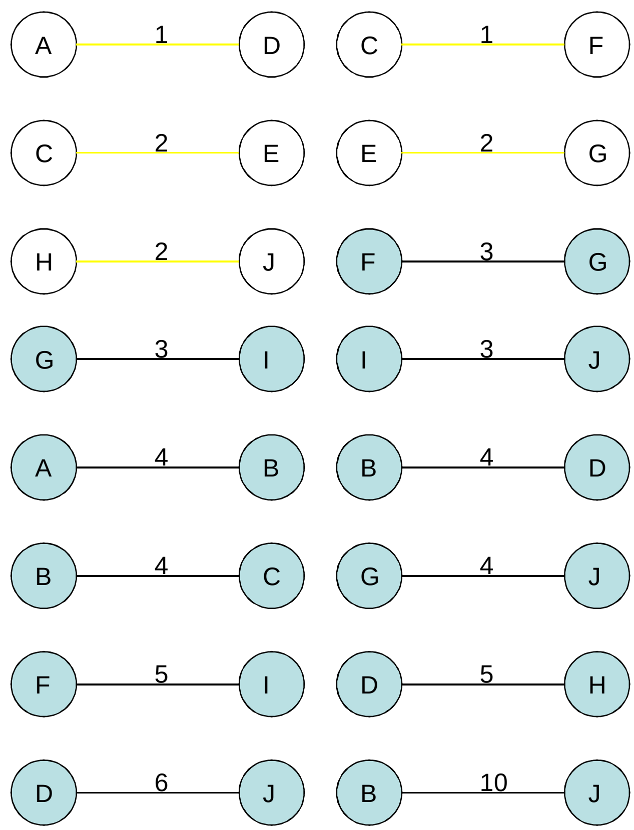
Add Edge





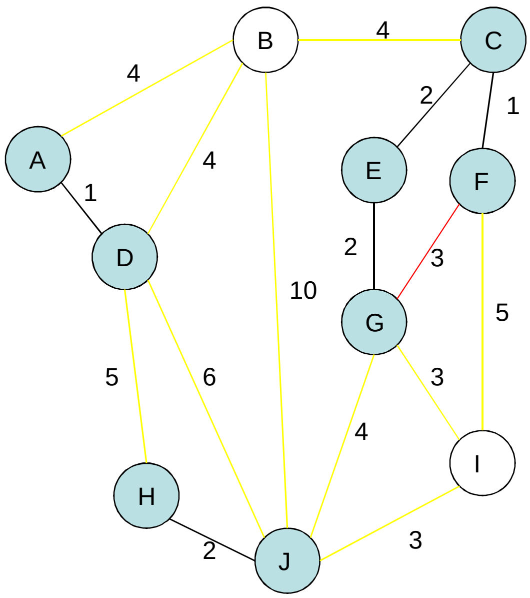
Add Edge

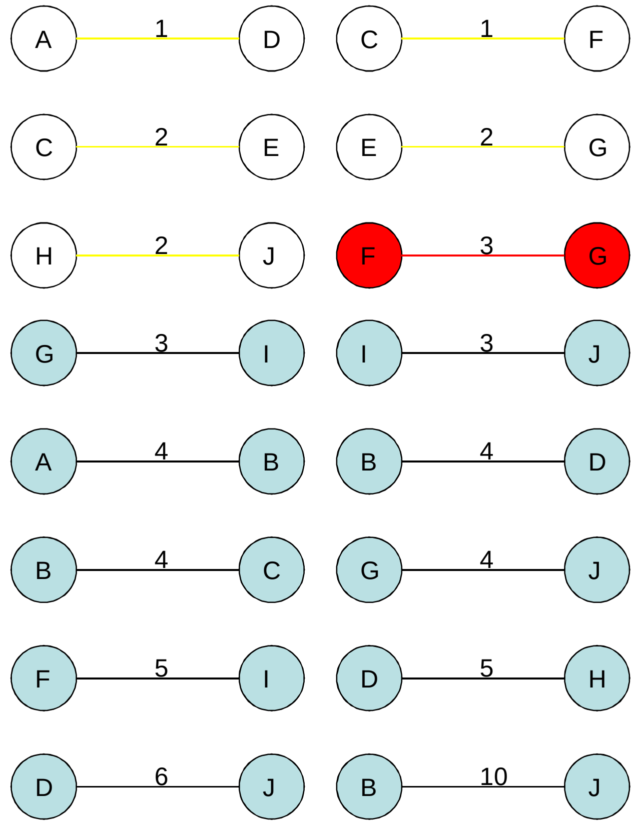




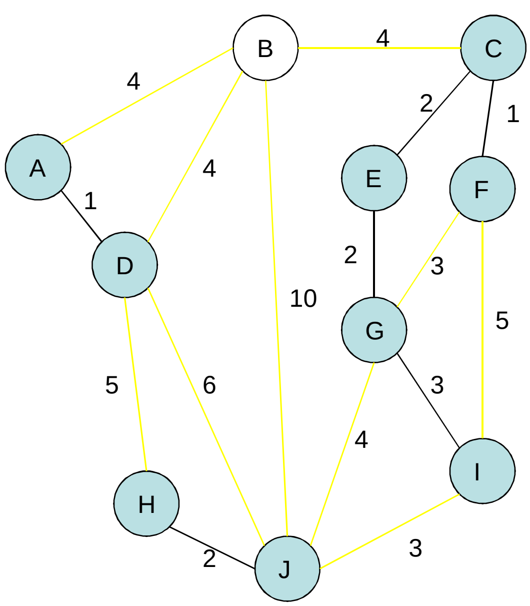
Cycle

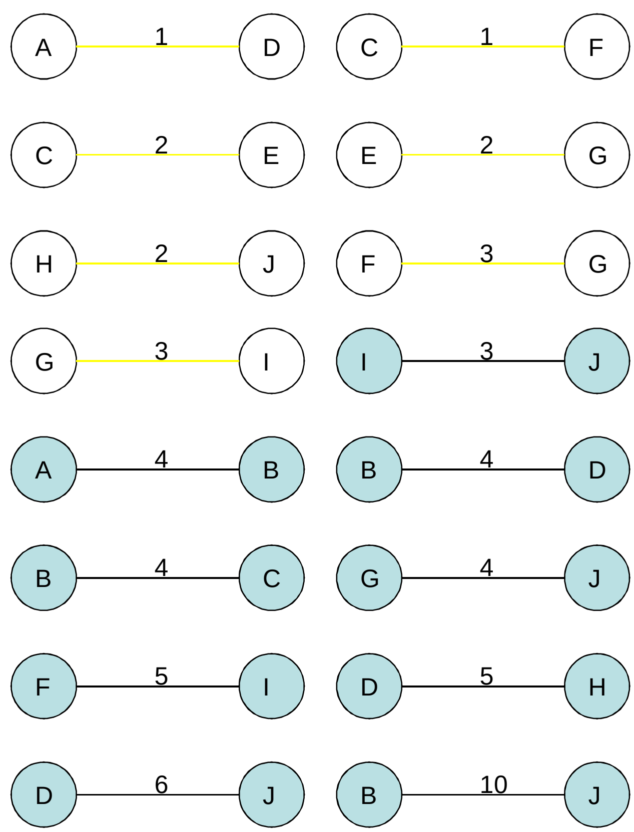
Don’t Add Edge



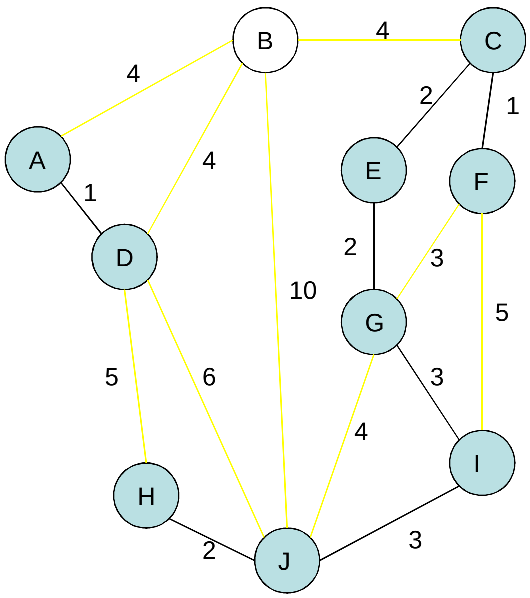


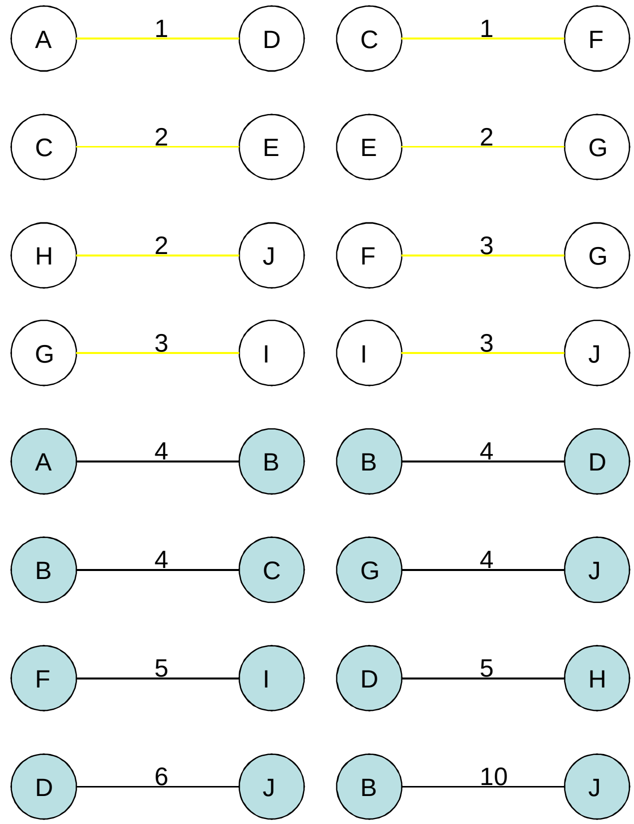
Add Edge



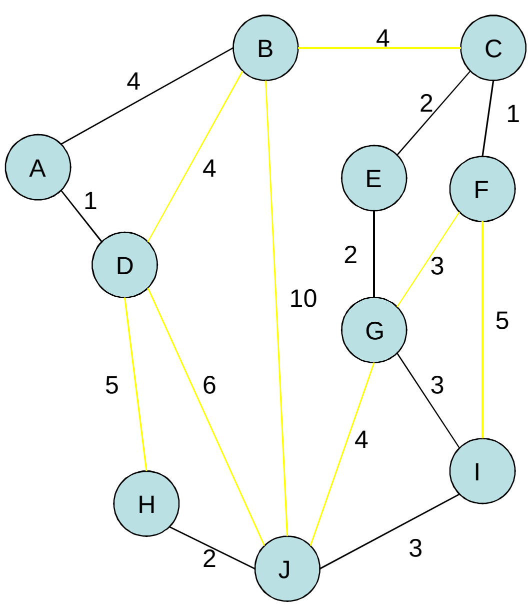


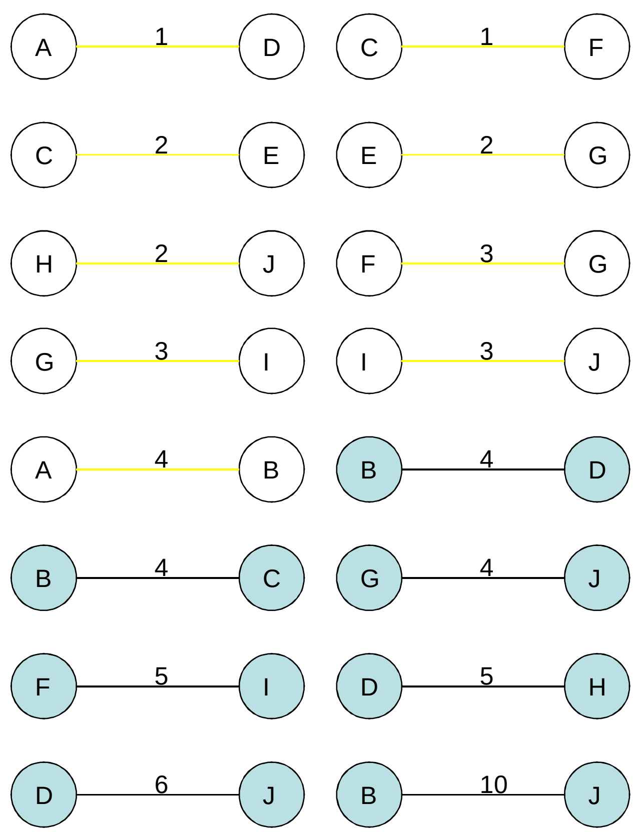
Add Edge





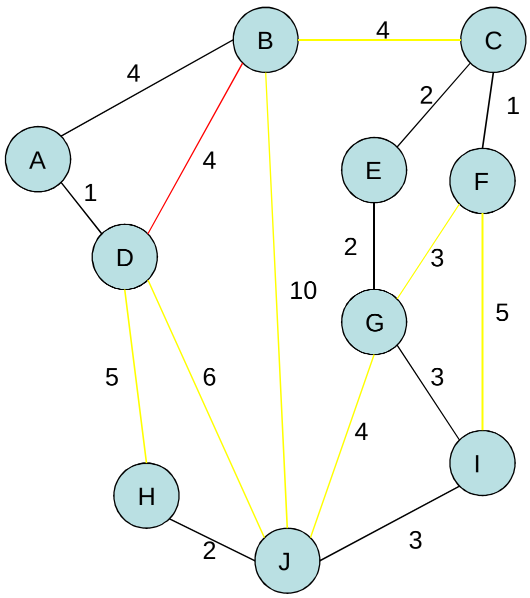
Add Edge

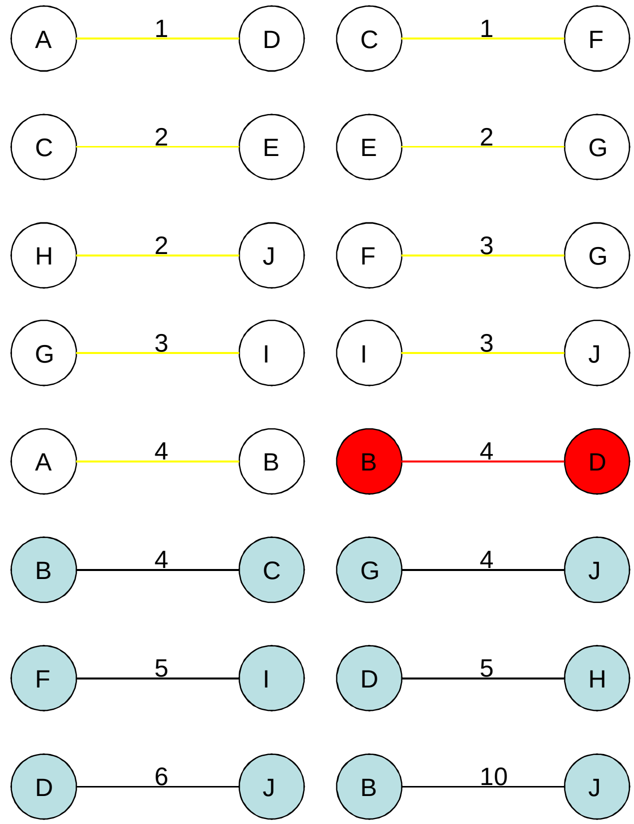




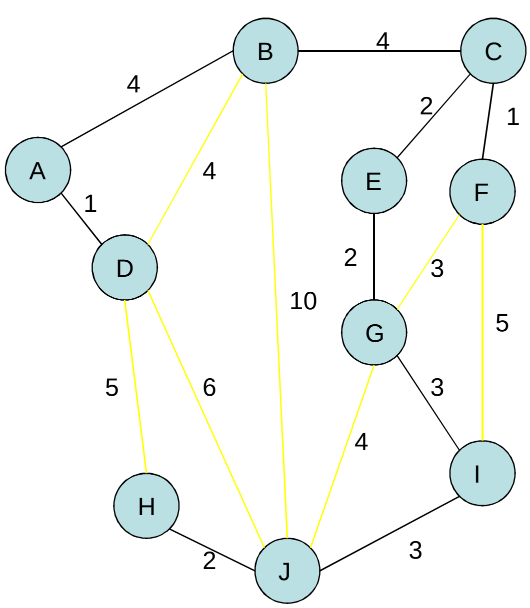
Cycle

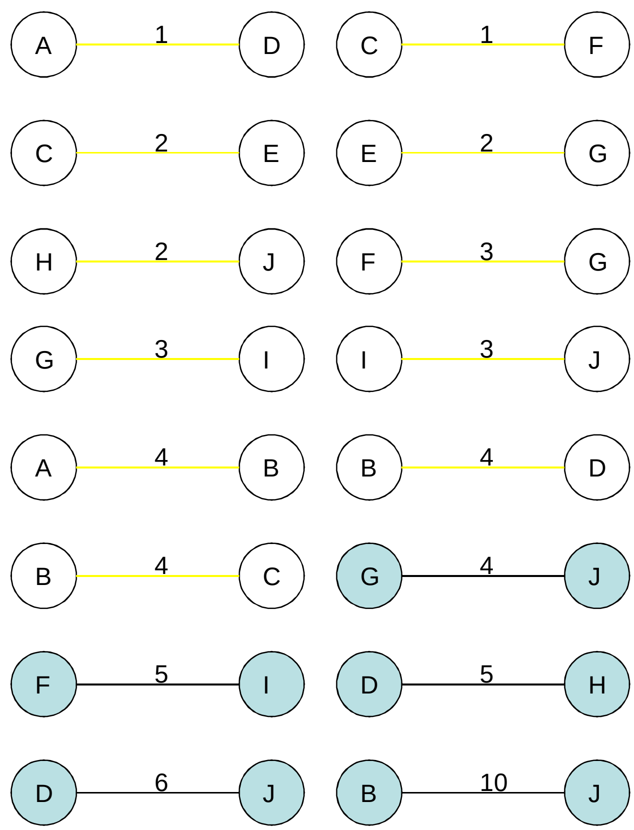
Don’t Add Edge



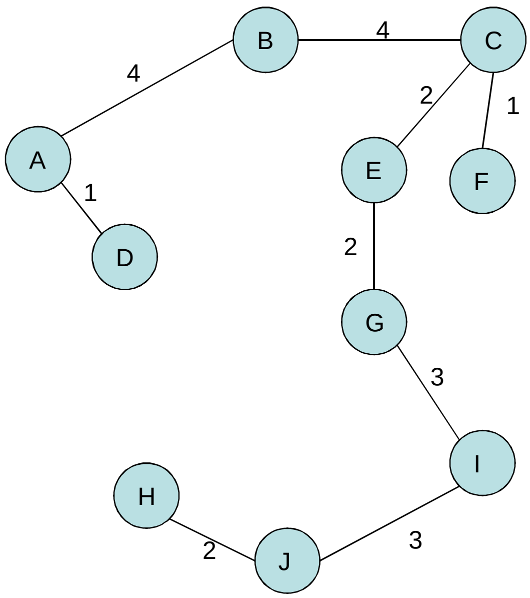


Add Edge

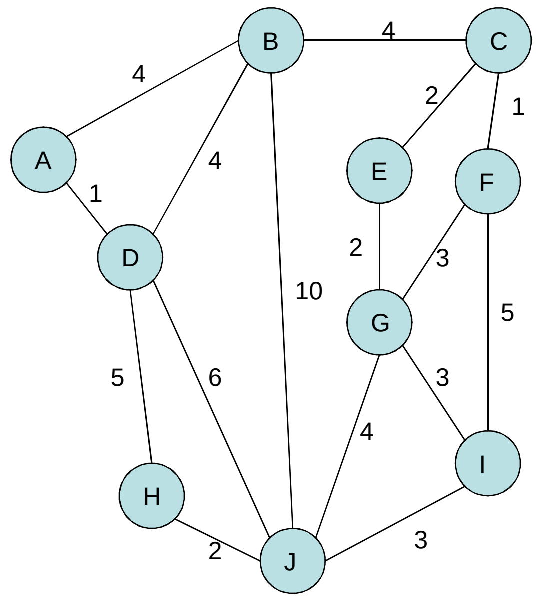




Minimum Spanning Tree

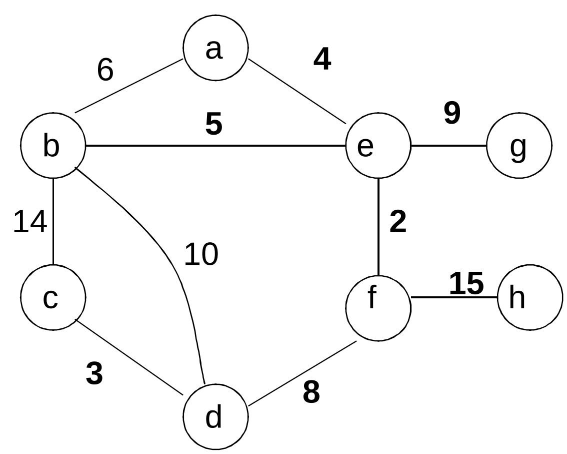


Complete Graph



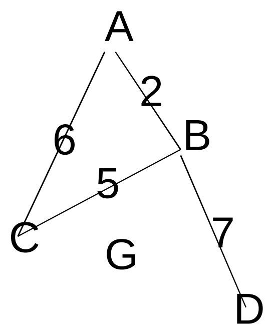
Kruskal's Algorithm:

1. Sort the edges *E* in non-decreasing weight
2. $T \leftarrow \varnothing$
3. For each $v \in V$ create a set.
4. repeat
5. Select next $\{u,v\} \in E$, in order
6. *ucomp* ← *find* (*u*)
7. *vcomp* ← *find* (*v*)
8. **if** *ucomp* ≠ *vcomp* **then**
8. add edge (*u,v*) to *T*
9. *union* (*ucomp,vcomp*)
10. **until** *T* contains |*V* | - 1 edges
11. **return** tree *T*



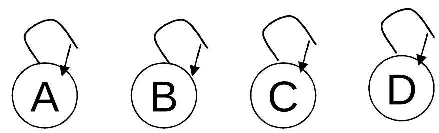
C = { {a}, {b}, {c}, {d}, {e}, {f}, {g}, {h} }
C is a forest of trees.

Kruskal - Disjoint set After Initialization



Sorted edges | *T*

A B 2
B C 5
A C 6
B D 7



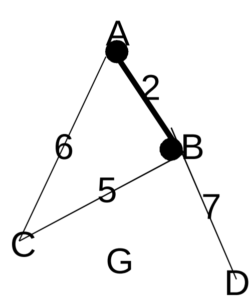
Disjoint data set for G

1. Sort the edges *E* in non-decreasing weight

2. $T \leftarrow \emptyset$

3. For each $v \in V$ create a set.

Kruskal - add minimum weight edge if feasible

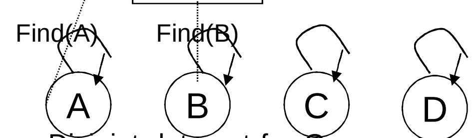


Sorted edges

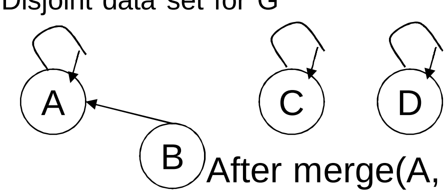
A B 2
B C 5
A C 6
B D 7

T

(A, B)



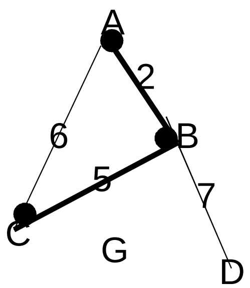
Disjoint data set for G

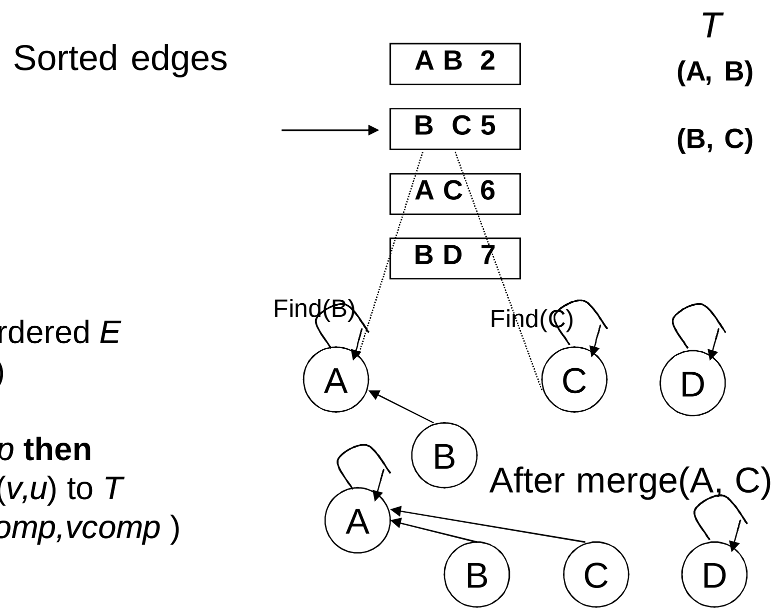


After merge(A, B)

5. for each $\{u,v\} \in$ in ordered *E*
6. *ucomp* $\leftarrow$ *find* (*u*)
7. *vcomp* $\leftarrow$ *find* (*v*)
8. **if** *ucomp* $\neq$ *vcomp* **then**
9. add edge (*v*,*u*) to *T*
10. *union*(*ucomp*,*vcomp*)

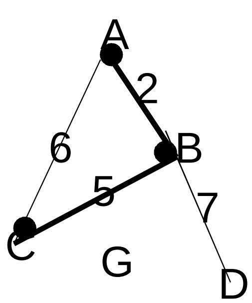
Kruskal - add minimum weight edge if feasible

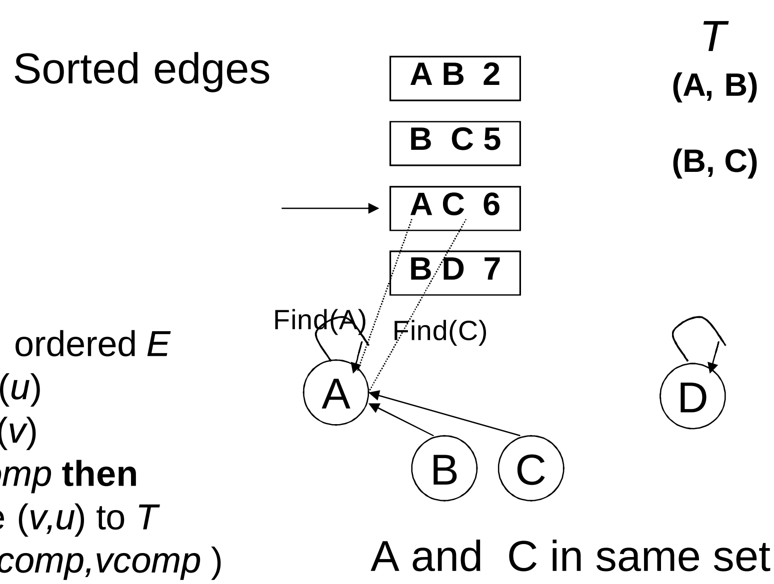




5. for each {*u*,*v*} ∈ in ordered *E*
6. *ucomp* ← *find* (*u*)
7. *vcomp* ← *find* (*v*)
8. **if** *ucomp* ¹ *vcomp* **then**
9. add edge (*v*,*u*) to *T*
10. *union* (*ucomp*,*vcomp*)

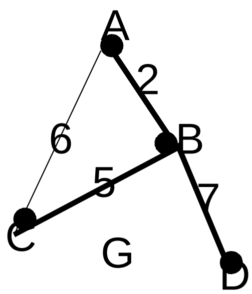
Kruskal - add minimum weight edge if feasible





5. for each {*u*,*v*} ∈ in ordered *E*
6. *ucomp* ← *find* (*u*)
7. *vcomp* ← *find* (*v*)
8. **if** *ucomp* ¹ *vcomp* **then**
9. add edge (*v*,*u*) to *T*
10. *union* (*ucomp*,*vcomp*)

Kruskal - add minimum weight edge if feasible



Sorted edges

A	B	2
B	C	5
A	C	6
B	D	7

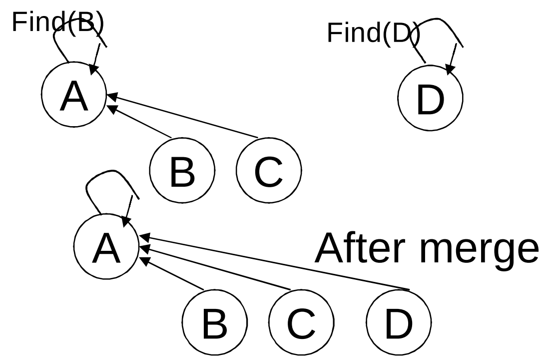
T

(A, B)

(B, C)

(B, D)

```
for each {u,v} ∈ in ordered E
    ucomp ← find (u)
    vcomp ← find (v)
    if ucomp ≠ vcomp then
        add edge (v,u) to T
        union ( ucomp,vcomp )
```



Analysis of Kruskal's Algorithm

Running Time = O(m log n) (m = edges, n = nodes)

Testing if an edge creates a cycle can be slow unless a complicated data structure called a “union-find” structure is used.

It usually only has to check a small fraction of the edges, but in some cases (like if there was a vertex connected to the graph by only one edge and it was the longest edge) it would have to check all the edges.

This algorithm works best, of course, if the number of edges is kept to a minimum.

Kruskal's Algorithm: Time Analysis

Kruskal (*G*)

1. Sort the edges *E* in non-decreasing weight
2. $T \leftarrow \emptyset$
3. For each $v \in V$ create a set.
4. repeat
5. $\{u,v\} \in E$, in order
6. *ucomp* $\leftarrow$ *find* (*u*)
7. *vcomp* $\leftarrow$ *find* (*v*)
8. **if** *ucomp* $\neq$ *vcomp* **then**
9. add edge (*v,u*) to *T*
10. *union* (*ucomp,vcomp*)
11. **until** *T* contains $|V|$ - 1 edges
12. **return** tree *T*

Count$_1$ = $\Theta(E \lg E)$

Count$_2$= $\Theta(1)$

Count$_3$= $\Theta(V)$

Count$_4$ = $O(E)$

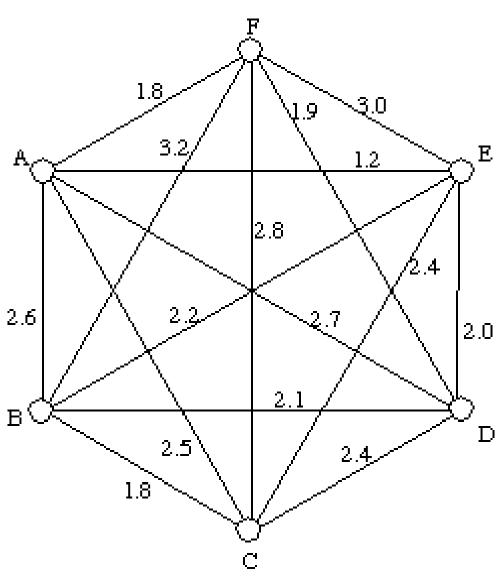
Using Disjoint set-height and path compression

Count$_{4(6+7+10)}$= $O((E+V)\,\alpha(V))$

Sorting dominates the runtime.
We get $T(E,V) = \Theta(E \lg E)$, so for a sparse graph we get $\Theta(V \lg V)$
for a dense graph we get $\Theta(V^2 \lg V^2) = \Theta(V^2 \lg V)$

Minimum Spanning Tree

Given the weighted graph below:



1. Use Kruskal's algorithm to find a minimum spanning tree and indicate the edges in the graph shown below: Indicate on the edges that are selected the order of their selection.

2. Use Prim's algorithm to find the minimum spanning tree and indicate the edges in the graph shown below. Indicate on the edges that are selected the order of their selection.

Minimum Spanning Tree

- http://www.cse.yorku.ca/~aaw/Ghiassi/MST/MSTAlg.htm