Data Structures and Algorithms

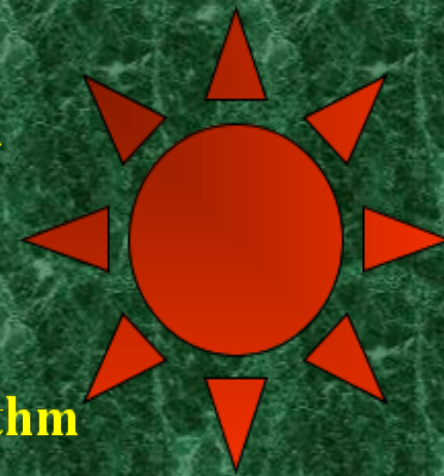
BSc. In Computing

Semester 5 Lecture 7

Lecturer: Dr. Simon McLoughlin

This Week:

- Minimum Spanning Tree (MST)
- The Dijkstra-Prim MST Algorithm
- Greedy Algorithms
- The Kruskal MST Algorithm
- Shortest Path Algorithms
- The Dijkstra Shortest Path Algorithm



Minimum Spanning Tree Algorithm

The Minimum Spanning Tree (**MST**) of a weighed connected graph is a **subgraph** that contains:

- **All** of the nodes of the original and
- A **subset** of the edges

so that:

- The **subgraph is connected** and
- The sum of the edge weights is the smallest possible.

The Dijkstra-Prim Algorithm

The following algorithm to find an MST was developed by Edsger Dijkstra and R.C Prim in the 1950s.

To find the MST, we will use what is called a **greedy algorithm**.

Greedy algorithms work by looking at a subset of the larger problem and making the best decision based on that information.

In this case, at each step of the process we will:

- Look at a collection of potential edges to add to the spanning tree and
- Pick the one with the smallest weight.

By repeatedly doing this, we will grow a spanning tree that has a minimum overall total.

To accomplish this process, we will consider the nodes of the graph to be in **one of three categories:**

- In the tree
- On the fringes of the tree
- Not yet considered

We begin by picking one node of the graph and putting that into the spanning tree

- Because our result is an unrooted tree, the choice of an initial node has no impact on out final results

We then place **all of the nodes** that are **connected to this initial one** into the **fringe category**.

- We **loop** through the process of **picking the smallest weighed edge** connecting a tree with a fringe node,
- **adding the new node to the tree**, and then
- **updating** the nodes in the fringe category.

When all the nodes have been added to the tree, we are completed.

Our general algorithm for this process is:

select a starting node;
build the initial fringe from nodes connected to the starting node
while (there are nodes left)
{
    choose the edge to the fringe of the smallest weight;
    add the associated node to the tr**e**;
    update the fringe by:
        {
            adding nodes to the fringe connected to the new node
            updating the edges to the fringe so that they are the smallest.
        }
}

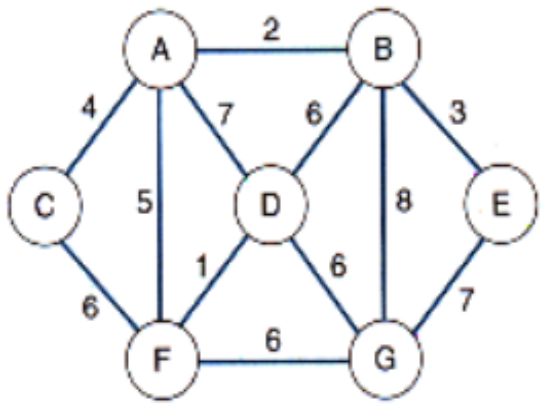
Diagram of progress of Algorithm

We choose node A to start the process.

All of the nodes directly connected to A become the starting fringe.

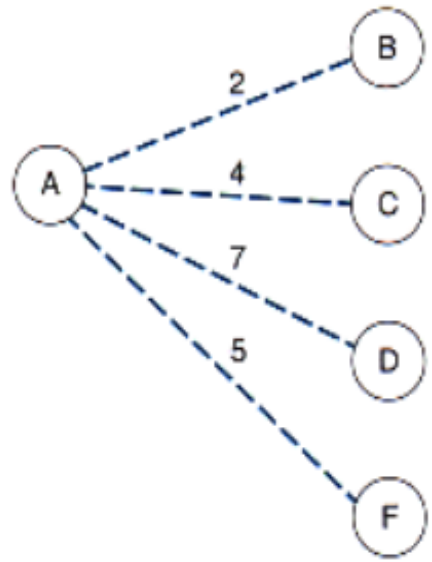
The edge with the smallest weight connects nodes A and B, so B is added to the MST along with edge AB.

A



The original graph

B

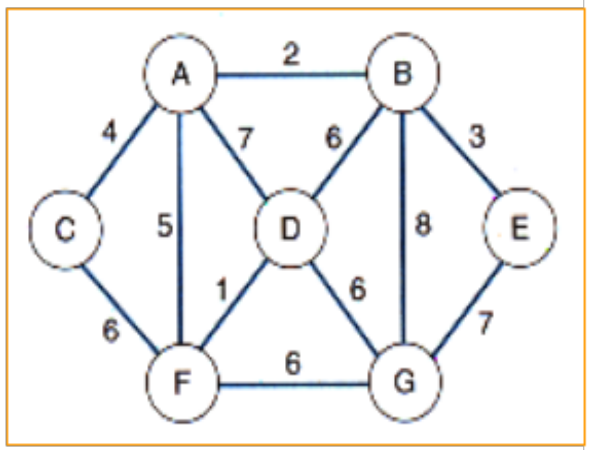


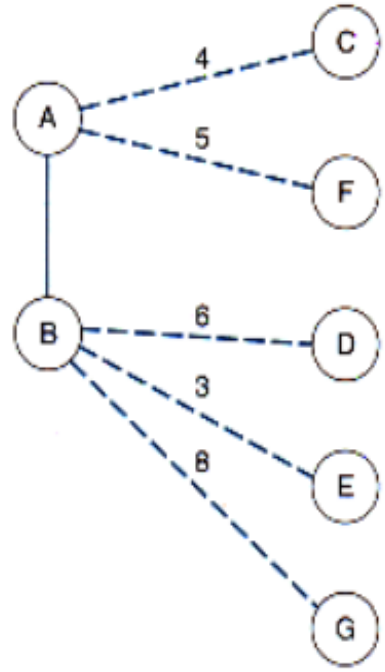
First node added. (Dashed lines show edges to fringe nodes.)

When node B is added to the tree we need to determine if there are any nodes that need to be added to the fringe set, and we find that nodes E and G must be added.

Because the only tree node they are connected to is node B, we add those edges to the ones we will consider next.

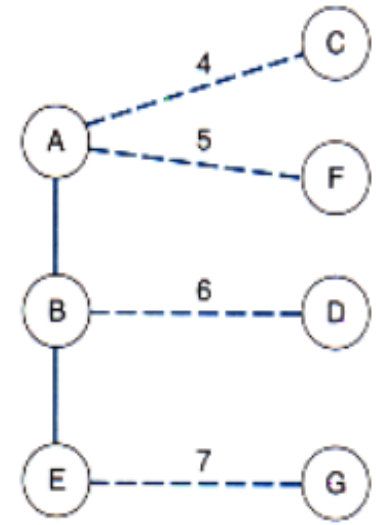
C





Second node added. Edges to nodes D, E, and G updated. (Solid lines show edges in the MST.)

D



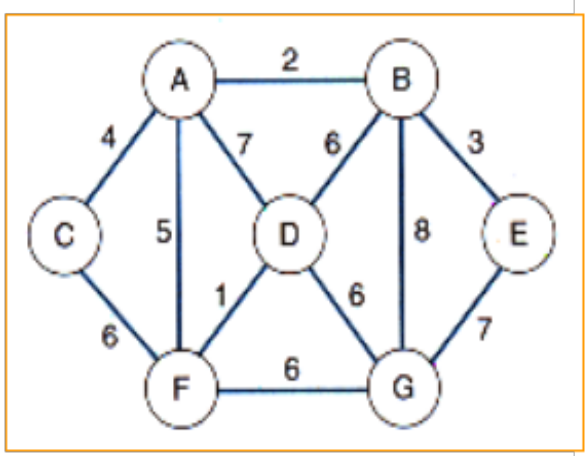
Third node added. Edge to node G updated.

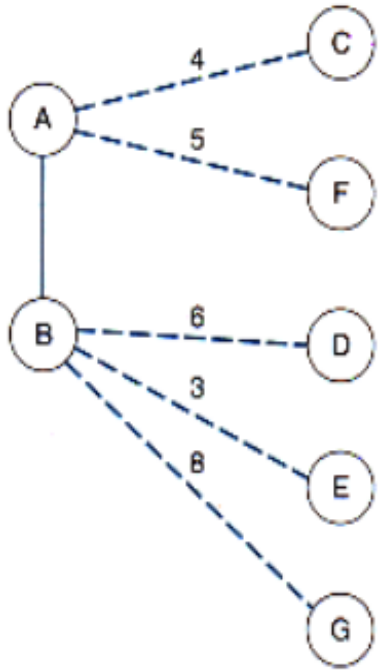
We need to check if the edges from **node A** to **nodes C, D and F** are still the shortest route, or if there are better edges from**B** to these three nodes.

In the original graph, there are no direct connecti ons from node B to nodes C and F, so these will not change.

But the edge from node B to node D has a smaller weight that the one from node A, and so the edge BD now replaces the edge AD.

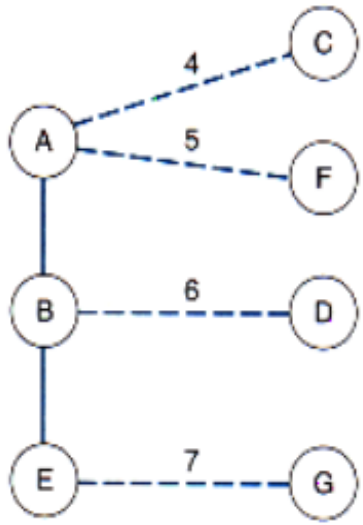
C





Second node added. Edges to nodes D, E, and G updated. (Solid lines show edges in the MST.)

D



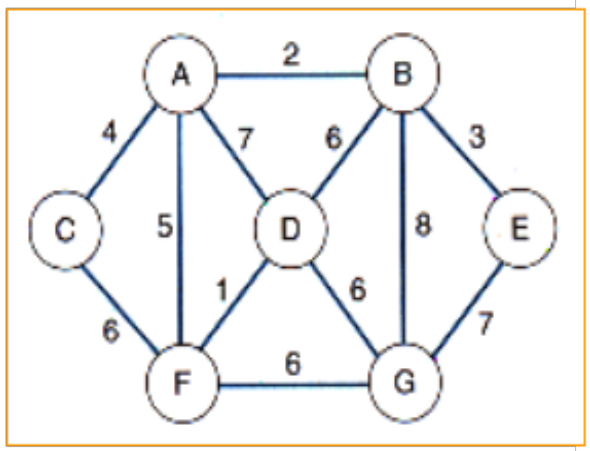
Third node added. Edge to node G updated.

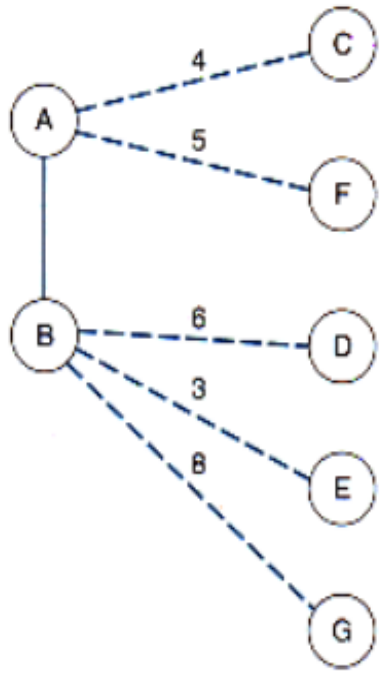
Of the five edges to fringe nodes, we see that **BE** has the smallest weight, and so it and node **E** are added to the tree

The edge **EG** has a smaller weight that the edge **BG**, so it is now used.

Of the four current edges to the fringe, we see that **AC** has the smallest weight and so it is added next.

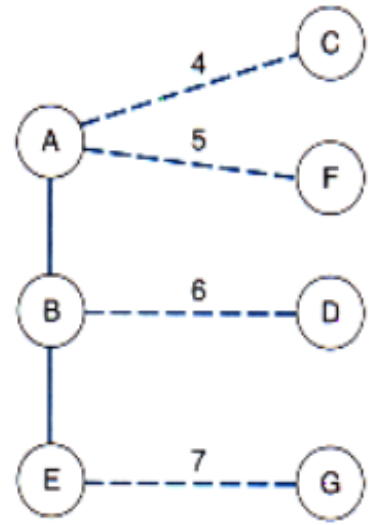
C





Second node added. Edges to nodes D, E, and G updated. (Solid lines show edges in the MST.)

D

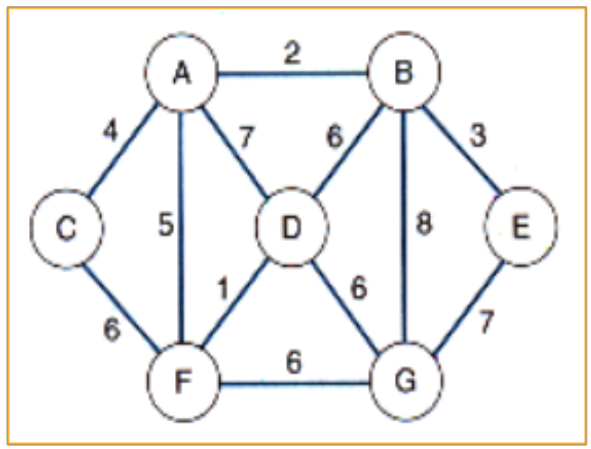


Third node added. Edge to node G updated.

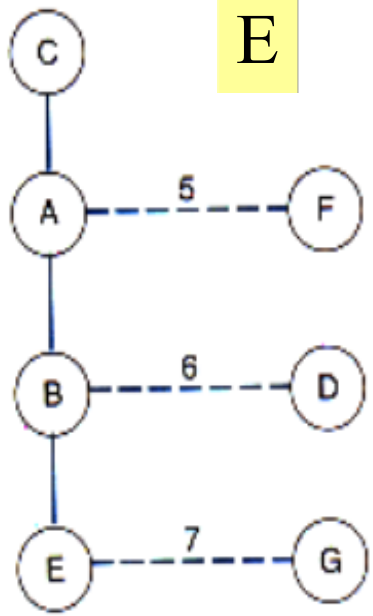
The addition of node C and edge **AC** to the spanning tree did not cause any edges to be updated.

We next choose edge **AF** so it and the node F are added to the tree.

We also update the links because the edge **FD** has a smaller weight then **BD** and edge **FG** has a smaller weight the **EG**.

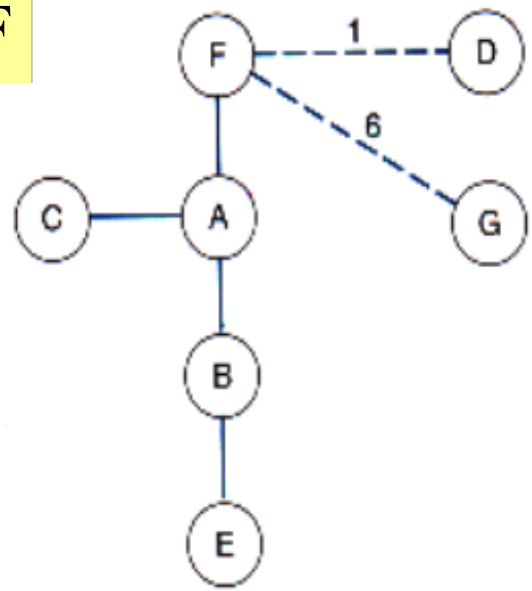


E



Node C added to the tree

F



Node F added to the tree and edges to nodes D and G are updated

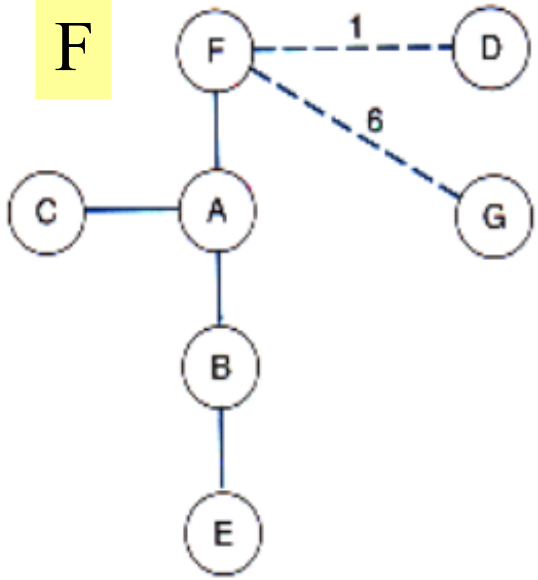
In the resulting fringe, we see that the edge FD is now the remaining edge with the smallest weight, so it is added next.

We now have just one node that has not been added to the tree.

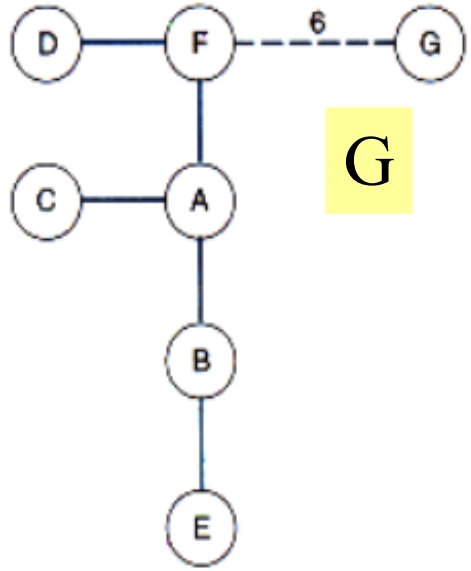
When it is added, the process is complete.

We have determined the MST rooted at node A

F



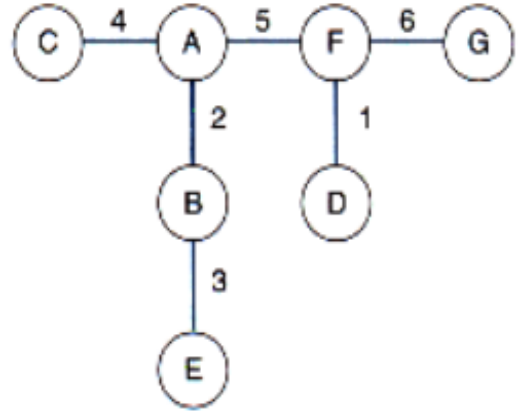
Node F added to the tree and edges to nodes D and G are updated



G

Only one node is left in the fringe

H

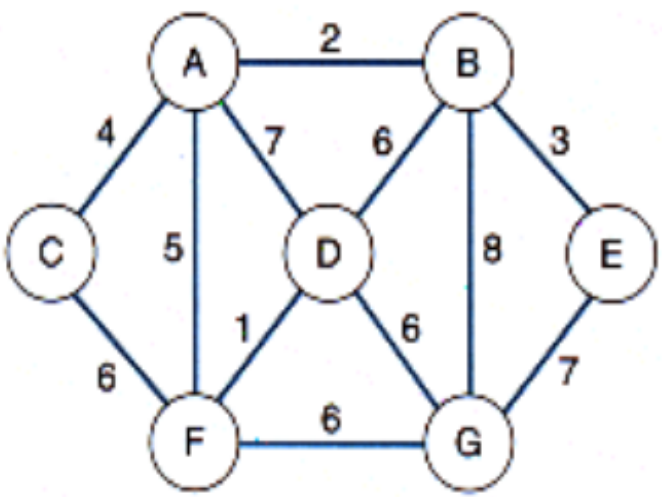


The complete minimum spanning tree rooted at node A

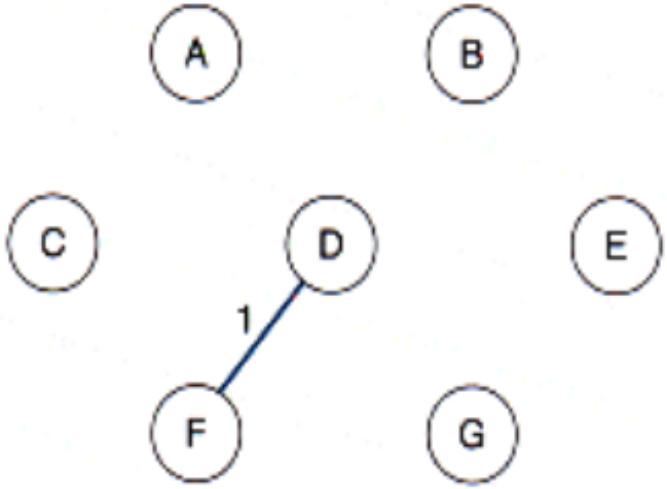
The Kruskal Minimum Spanning Tree Algorithm

The **Dijkstra-Prim MST algorithm** began at a particular node and built the minimum spanning tree outward, but…

Kruskal's algorithm concentrates instead on the edges of a graph.



The original graph



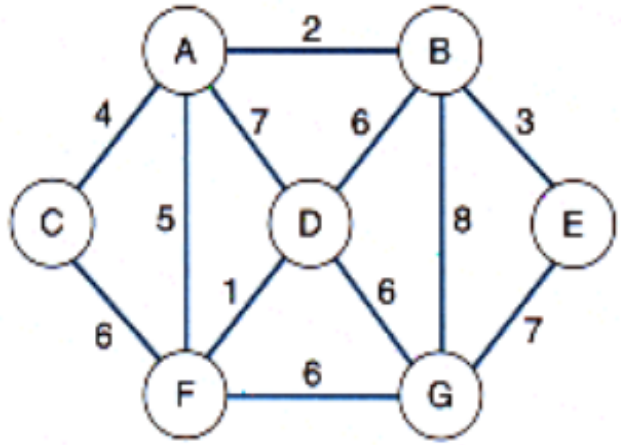
First edge added

In this algorithm, we begin with an **empty spanning tree** and **add edges in order of increasing weight**until **all nodes are connected to the graph**

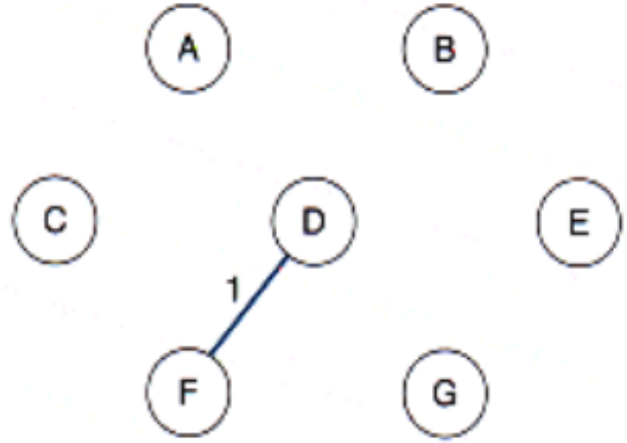
If we **run out of edges before all of the nodes are connected** , the **original graph was not connected** and the result we have generated is the MSTs of each of the connected components of the original graph.

We begin with the same graph that we used with the **Dijkstra-Prim algorithm**

In this case, we first add the edge with the lowest weight , which is the one between nodes D and F, giving the partial result below



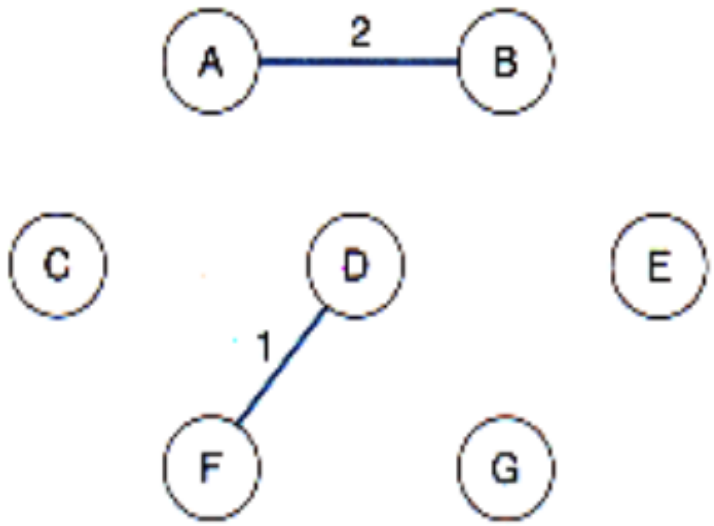
The original graph



First edge added

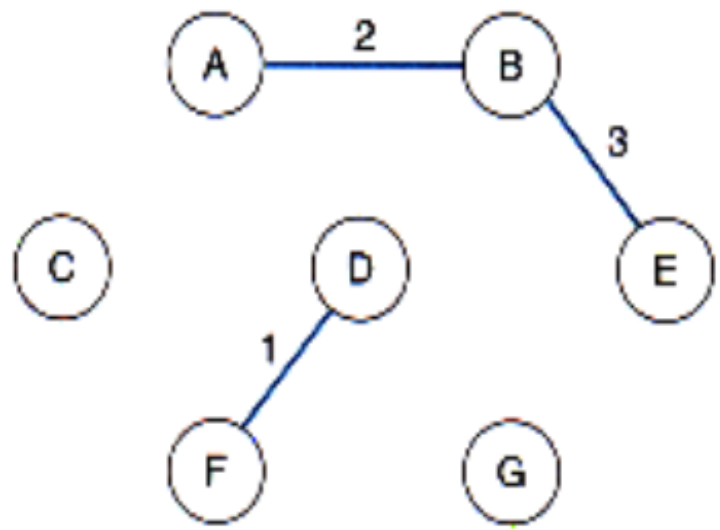
The edge with weight 2 is added next between the **nodes A and B**, and then the edge with weights 3 is added.

C

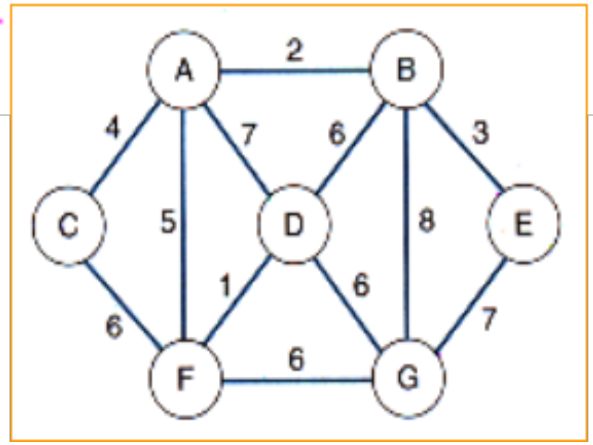


Second edge added

D

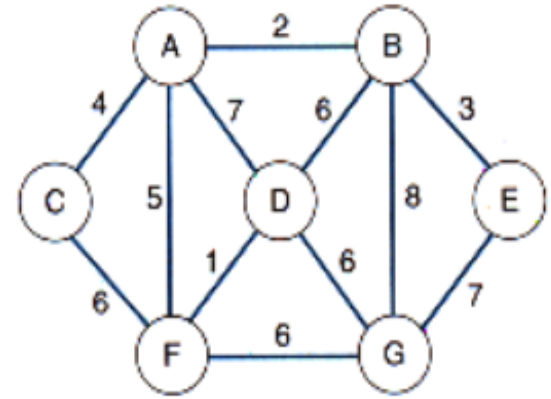


Third edge added

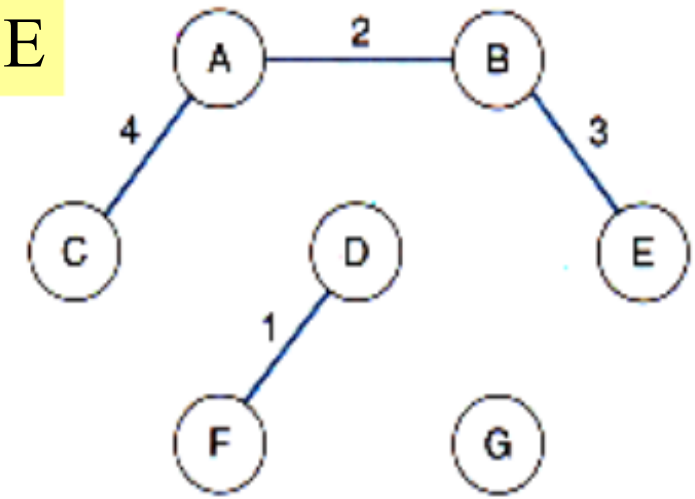


The edges with weights of 4 and 5 are added next to our result.

Only node G is still unconnected.

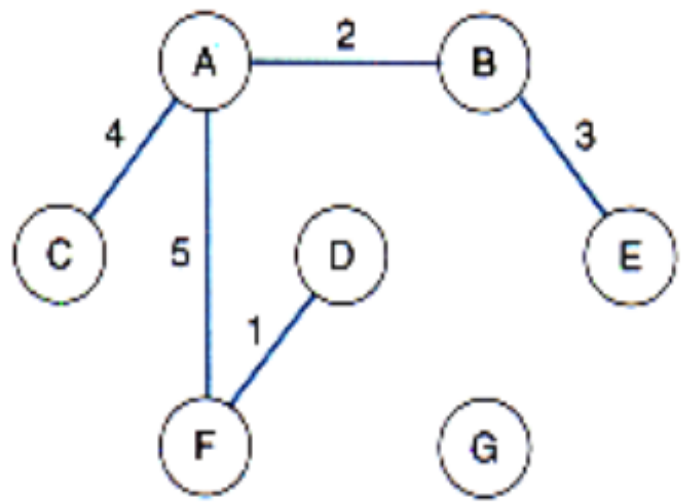


E



Fourth edge added

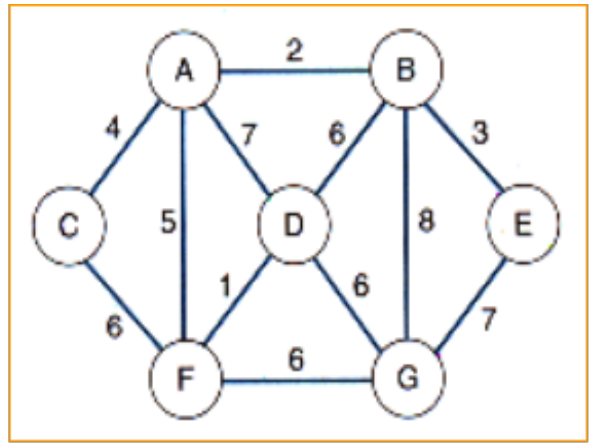
F



Fifth edge added

The next edge to consider is those with a weight of 6.

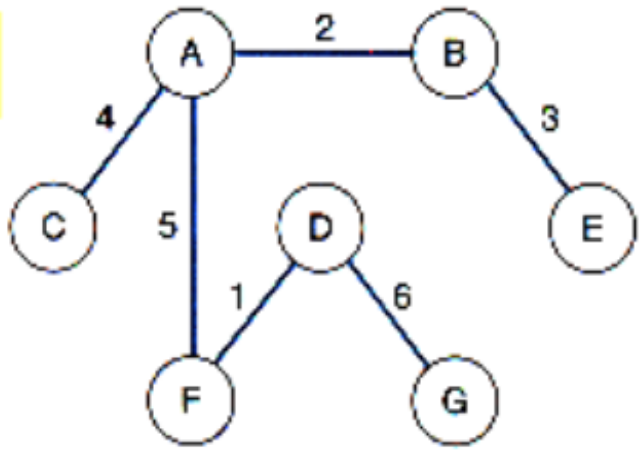
- Of the **four edges with a weight of 6**, two are discarded because they would form a cycle with edges already part of the MST.



The edge between nodes C and F would form a cycle that includes node A, and the edge between node B and D would form a cycle that includes A and F.

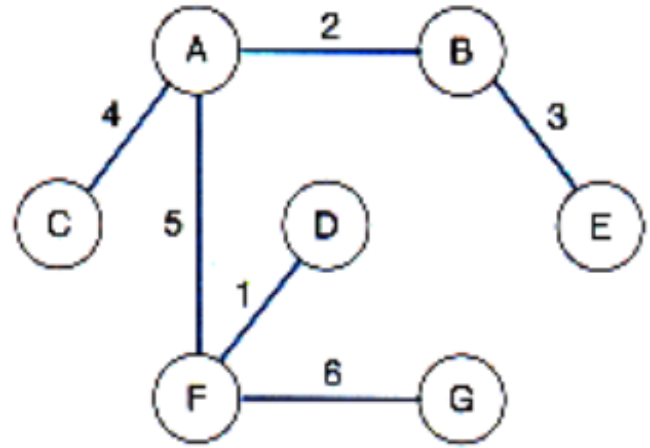
The other two nodes are good alternatives, and depending on the one chooses, we get the MST below

G



A minimum spanning tree

H

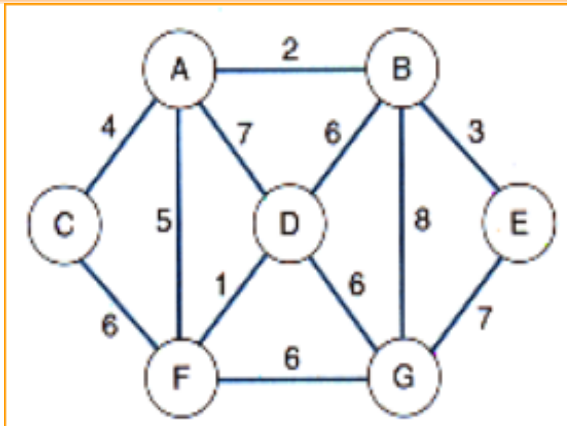


An alternative minimum spanning tree

The general algorithm that will accomplish this is:

```
E    // the number of edges
N    // the number of vertices
sort the edges in non-decreasing order by weight
initialise partition structure
edgeCount = 1;
includedCount = 0;
while (edgeCount <=  E && includedCount <= N-1)
{
      parent1 = FindRoot(edge[edgeCount].start )
      parent2 = FindRoot(edge[edgeCount].end )
      if (parent1 != parent2)
      {
            add edge[edgeCount] to spanning tree
            included Count = includedCount + 1;
            Union (parent1, parent2)
      }
      edgeCount = edgeCount + 1;
}
```

The pseudocode for FindRoot and Union is in the McConnell

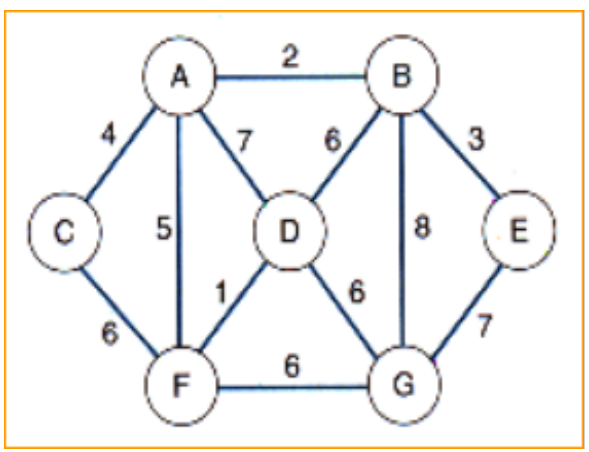


The main loop will continue until the **edgeCount** variable indicates that we have looked at all the edges, or the **includedCount** indicates that we have added enough edges to create the spanning tree.

If we have N nodes in the graph, a spanning tree would have one less edge than nodes.

The main loop will continue until the **edgeCount** variable indicates that we have looked at all the edges, or the **includedCount** indicates that we have added enough edges to create the spanning tree.

If we have N nodes in the graph, a spanning tree would have one less edge than nodes.



- Inside the loop, we first find the parents of the two nodes that are connected by the next edge we are considering.

- If those nodes are in partitions with different roots, adding an edge between them will not create a cycle, so this current edge can be added to the MST and those two pieces can be joined so that they have the same root.

The **complexity of the algorithm** will be the complexity of the sort algorithm used because the while loop is linearly related to the number of edges.

This makes the complexity of the Kruskal MST algorithm **O(E log E)**, where E represents the number of edges.

That is, Kruskal MST algorithm is **O(n log n)**,

Shortest Path Algorithms

The shortest path algorithm will always find for two nodes the series of edges between them that will result in the smallest total weight.

- The **Minimum Spanning Tree Algorithm** will not work to find the shortest path because **it just considers the weight of one edge at each pass**.

- We must use an algorithm that chooses an edge that is part of the shortest **entire** path from the starting node. Then we will get the result we want.

Although we use the term shortest, we must be aware that the weights could be a measure other than distance.

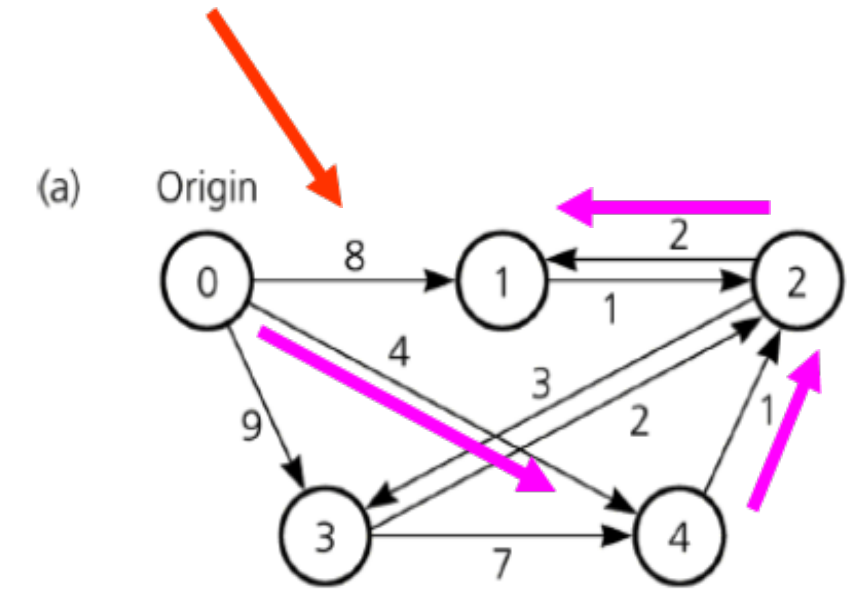
The weight could, for example, represent the **euro** cost, or a **time** duration.

The sum of the weights of the edges of a path is called the path's

- **length** or
- **weight** or
- **cost**

The shortest path fromvertex 0 to vertex 1 in the graph here is not the edge between 0 and 1 (cost = 8), but ratherthe path from 0 – 4 – 2 – 1(cost = 7).

a) A weighted directed graph and b) its adjacency matrix



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

Dijkstra Shortest Path Algorithm

An algorithm attributed to E. Dijkstra determines the shortest path between a given origin and all other vertices.

The algorithm uses a set **vertexSet** of selected vertices and an array **weight**, where:

weight[v] is the weight of the **current** shortest/cheapest path
- from **vertex 0**
- to **vertex v**
- that passes through vertices in **vertexSet**.

Initially, **vertexSet** contains only 0, and the weight array contains the weights of the single-edge paths from **vertex** 0 to all other vertices.

- We will see how this works soon, in a trace of the algorithm

That is, **weight[v]** equals **matrix[0][v]** for all **v**, where **matrix** is the adjacency matrix.

Therefore initially **weight** is the first row of matrix.

After this **initialisation**step, we find a vertex **v** that is not in **vertexSet** and has the smallest path weight i.e. min(weight[]) for all v not in **vertexSet**

We add **v** to **vertexSet**.

For all vertices **u** not in **vertexSet**, ..

we check to see if we can reduce **weight[u]** by following the path to our newly added vertex **v** onto **u.** If this path is shorter then we replace **weight[u]** with the weight of this path

To make the necessary determination,

1. break their path from **0** to **u** into pieces and

2. find their weights as follows:

   weight[v] = weight of the shortest path from **0** to **v**;
   matrix[v][u] = weight of the edge from **v** to **u**;

3. Then compare **weight[u]** with **weight[v] + matrix[v][u]** and

4. let:

   weight[u] = min(weight[u], weight[v] + matrix[v][u])

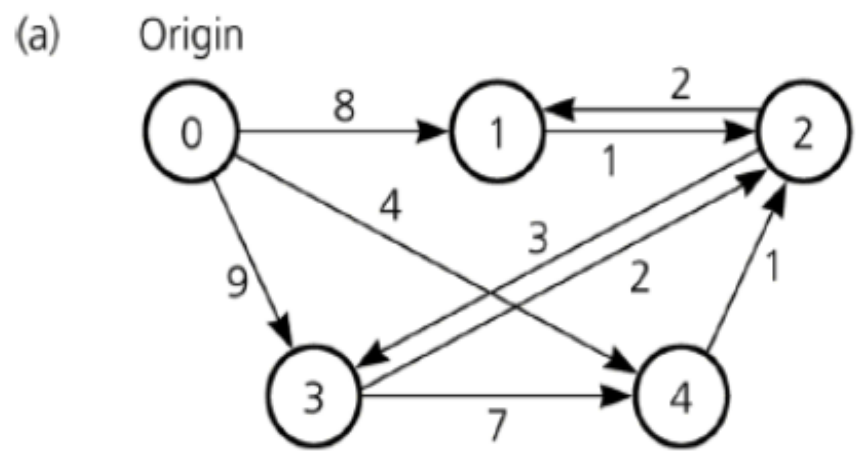
Trace of the Dijkstra shortest path algorithm.

The algorithm takes the following steps:

Step 1

The **vertexSet** initially contains vertex 0, and weight is initially the first row of the graphs adjacency matrix

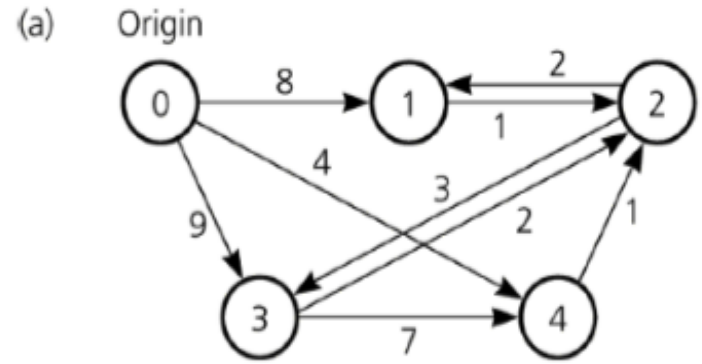
a) A weighted directed graph and b) its adjacency matrix



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

a) A weighted directed graph and b) its adjacency matrix



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

A trace of the shortest path applied to the above graph

Step	v	vertexSet	weight[0]	weight[1]	weight[2]	weight[3]	weight[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

The **vertexSet** initially contains vertex 0, and weight is initially the first row of the graphs adjacency matrix

Step 2

We have weight[4] = 4 as the smallest value in weight, ignoring weight[0] because 0 is in **vertexSet**.

Step	v	vertexSet	weight[0]	weight[1]	weight[2]	weight[3]	weight[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

Therefore, **v** = 4, so add 4 to **vertexSet**.

For vertices not in **vertexSet**, that is, for u = 1, 2, and 3, check whether it is shorter to go from 0 – 4, and then along an edge to **u** instead of directly from 0 to **u** along an edge.

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

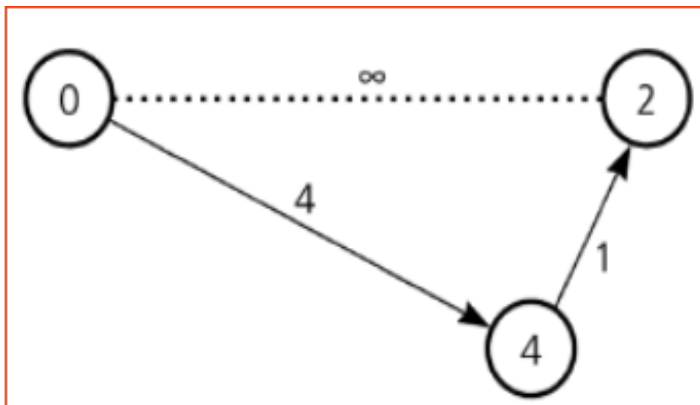
For vertices 1 and 3, it is not shorter to include vertex 4 in the pat

However, for vertex 2, we can notice that

weight[2] = ∞ > weight[4] + matrix[4][2] = 4 + 1 = 5.

Therefore,

replace weight[2] with 5.



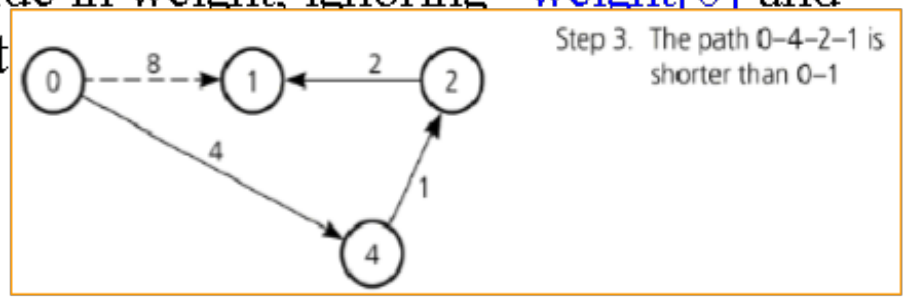
Step 2. The path 0–4–2 is shorter than 0–2

We can verify this by examining the graph directly

Step	v	vertexSet	weight[0]	weight[1]	weight[2]	weight[3]	weight[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

Step 3

We have weight[2] = **5** as the smallest value in weight, ignoring weight[0] and weight[4] because **0** and **4** are in **vertexSet**



Step 3. The path 0–4–2–1 is shorter than 0–1

Therefore, **v** = 2, so add **2** to **vertexSet**.

For vertices not in **vertexSet**, that is, for **u** = **1** and **3**, check whether it is shorter to take the path from **0** to **2** and them along an edge to u instead of the current path from **0** to **u**.

Step	v	vertexSet	weight[0]	weight[1]	weight[2]	weight[3]	weight[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

Notice that:

weight[1] = **8** > weight[2] + matrix[2][1] = 5 + 2 = 7

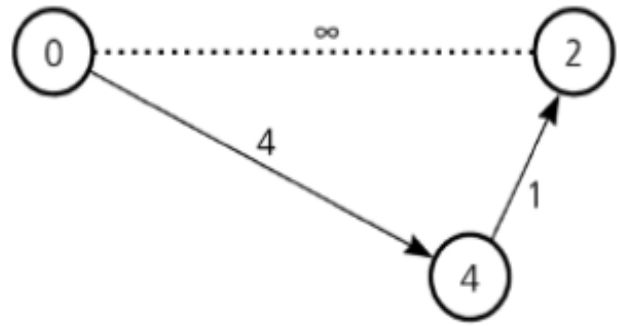
Therefore, we replace weight[1] with **7**.

weight[3] = **9** > weight[2] + matrix[2][3] = 5 + 3 = 8

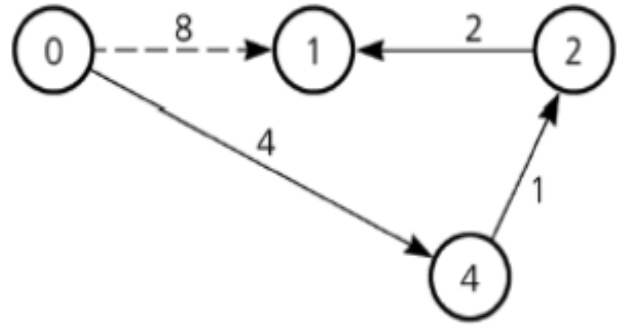
Therefore, we replace weight[3] with **8**.

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

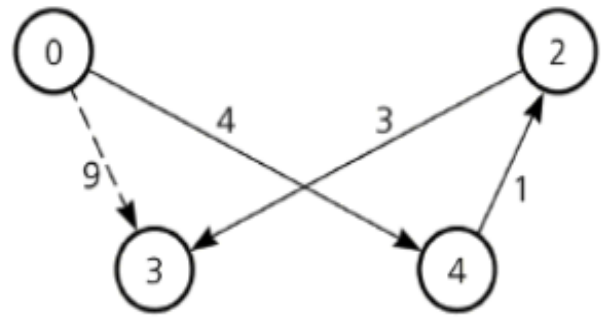
Step	v	vertexSet	weight[0]	weight[1]	weight[2]	weight[3]	weight[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4



Step 2. The path 0–4–2 is shorter than 0–2



Step 3. The path 0–4–2–1 is shorter than 0–1

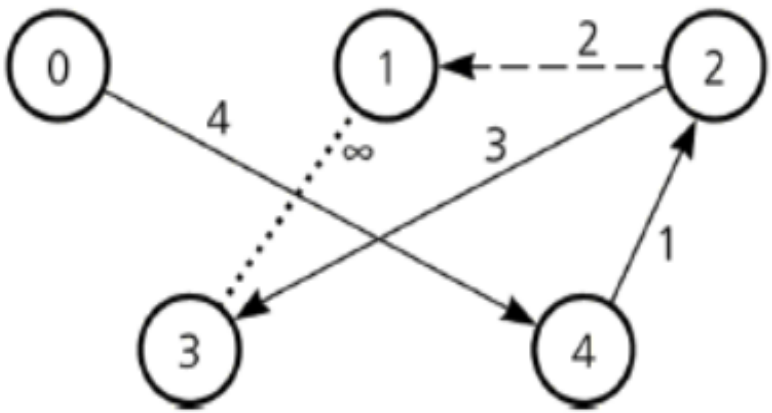


Step 3 continued. The path 0–4–2–3 is shorter than 0–3

Step 4

We have weight[1] = 7 as the smallest value in weight, ignoring weight[0], weight[2] and weight[4] because 0, 2, 4 are in **vertexSet**.

Therefore, v = 1, so add 1 to **vertexSet**.



Step 4. The path 0–4–2–3 is shorter than 0–4–2–1–3

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

For vertex 3, which is the only vertex not in **vertexSet**,
notice that weight[3] = 8 < weight[1] + matrix[1][3] = 7 + α.

Therefore, leave weight[3] as it is.

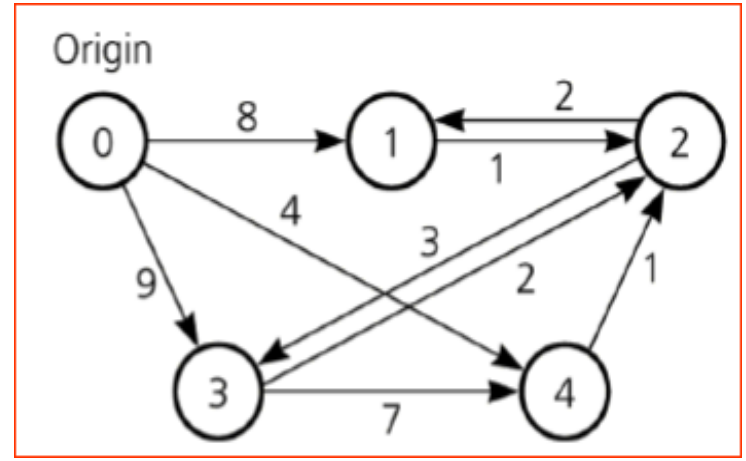
Step	v	vertexSet	weight[0]	weight[1]	weight[2]	weight[3]	weight[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

Step	v	vertexSet	weight[0]	weight[1]	weight[2]	weight[3]	weight[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

Step 5

The only remaining vertex not in **vertexSet** is 3, so add it to **vertexSet**.

Stop the process.

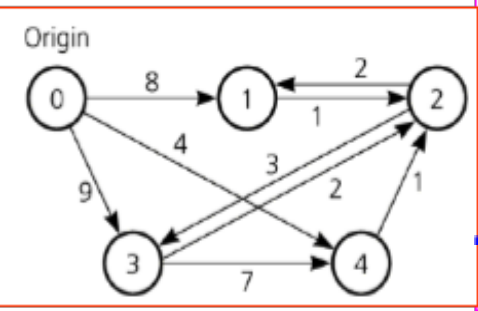


Step	v	vertexSet	weight[0]	weight[1]	weight[2]	weight[3]	weight[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

The final values in weight are the weights of the shortest paths.

These values appear in the last line of the trace.

For example, the shortest path from vertex 0 to vertex 1 has a cost of weight[1], which is 7.



Step	v	vertexSet	weight[0]	weight[1]	weight[2]	weight[3]	weight[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

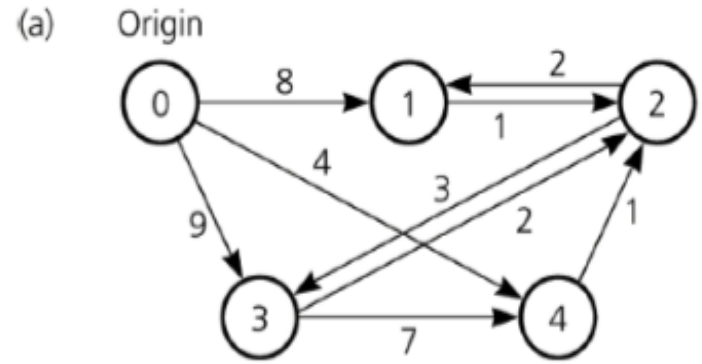
This result agrees with our earlier observation in relation to the graph diagram.

We saw that the shortest path is from 0 – 4 – 2 – 1.

Also, the shortest path from vertex 0 to vertex 2 has a cost of weight[2], which is 5. This is from 0 – 4 – 2.

The weights in weight are the smallest possible, as long as the algorithm's loop invariant is true.

a) A weighted directed graph and b) its adjacency matrix



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

A trace of the shortest path applied to the above graph

Step	v	vertexSet	weight[0]	weight[1]	weight[2]	weight[3]	weight[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

The **vertexSet** initially contains vertex 0, and weight is initially the first row of the graphs adjacency matrix

The pseudocode for the Dijkstra shortest path algorithm is:

The Dijkstra Shortest Path Algorithm

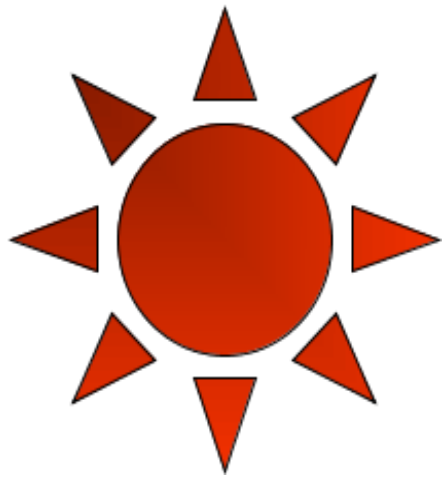
```
ShortestPath( theGraph, weight)
create a set of vertexSet that contains only vertex 0;
n = number of vertices in theGraph;

for (v = 0 though n – 1)
{
      weight[v] = matrix[0][v];
} //end for
for (step = 2 through n)
{
      find the smallest weight[v] such that v is not in  vertexSet;
      add v to vertexSet;

      // check weight[u] for all u not in vertexSet
      for (all vertices  u not in vertexSet)
      {
            if (weight[u] > weight[v] + matrix[v][u] )
            {
                  weight[u] = weight[v] + matrix[v][u];
            } // end if
      } //end for
} // end for
```

Summary of Graphs

The two most common implementations of a graph are the **adjacency matrix** and the **adjacency list.**

Each has its relative advantages and disadvantages.

The choice should depend on the needs of a given application.



Graph searching is an important application of stacks and queues.

- **Depth first searching** is a graph traversal algorithm that uses a stack to keep track of the sequence of visited vertices.
  - It goes as deep into the graph as it can before backtracking.
- **Breadth first search** uses a queue to keep track of the sequence of visited vertices.
  - It visits all possible adjacent vertices before traversing further into the graph.
  - When searching a graph, remember that the algorithm might take wrong turns. For example, we must always **eliminate the possibility of cycling** within the algorithm; the **algorithm must be able to backtrack** when it hits a dead end.

Trees are connected undirected graphs without cycles.

- A **spanning tree** of a connected undirected graph is a subgraph that contains all of the graph's vertices and enough of its edges to form a tree.

  DFS and **BFS** traversals produce DFS and BFS spanning trees.

- A **minimum spanning tree** for a weighted undirected graph is a spanning tree whose edge-weight sum in minimal.

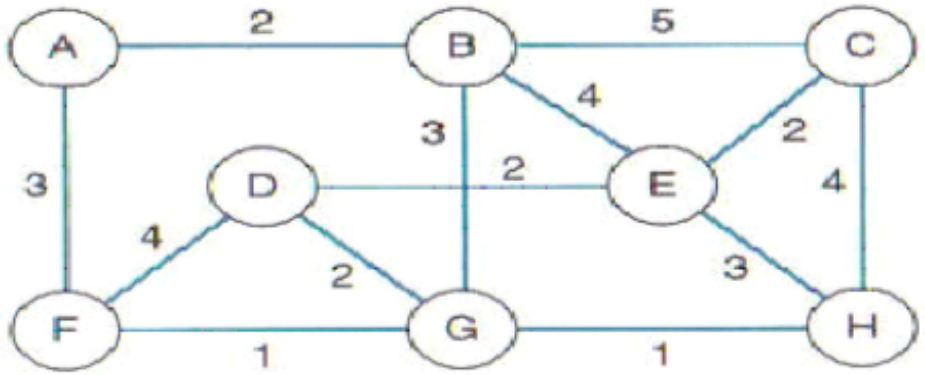
  Although a particular graph can have several minimum spanning trees, their edge-weight sums will be the same.

- The **shortest path** between two vertices in a weighted directed graph is the path that has the smallest sum of its edge weights.

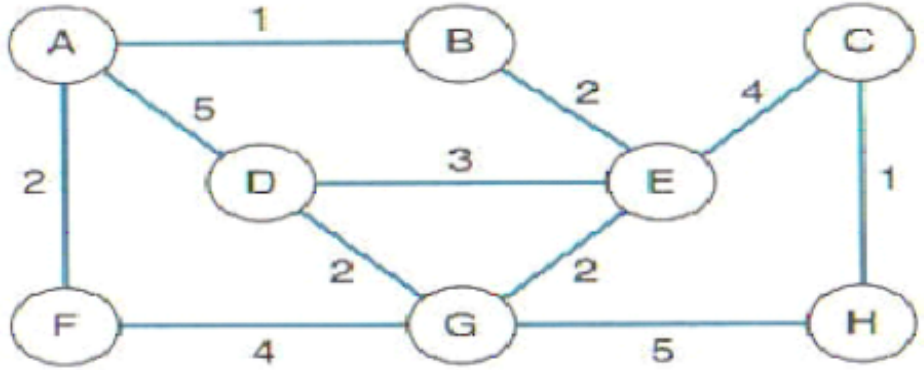
Programs to do this week:

1. Find the minimum spanning tree using the Dijkstra-Prim algorithm for the following graphs starting at node A. Show all steps.

a.



c.



2. Find the minimum spanning tree using the Kruskal algorithm for the graphs in question 1. Show all steps.

Read Chapter 6 Graph Algorithms
in Analysis of Algorithms
by McConnell

Read Chapter 13 Graphs.
in
Data Abstraction and Problem Solving with Java
Walls and Mirrors
by Carrano and Pritchard