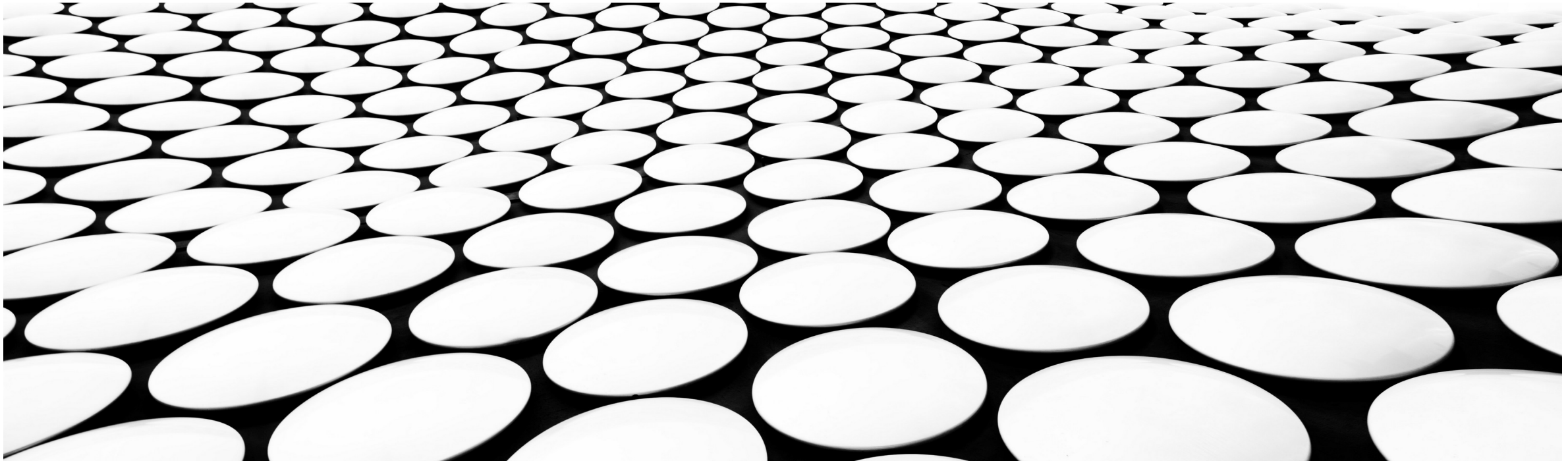
DSA 456 – WEEK 8

GRAPHS



GRAPHS

A graph are made up of a set of vertices and edges that form connections between vertices. If the edges are directed, the graph is sometimes called a digraph. Graphs can be used to model data where we are interested in connections and relationships between data.

```
* flight routes between cities
* social media follower counts
* and more...
```

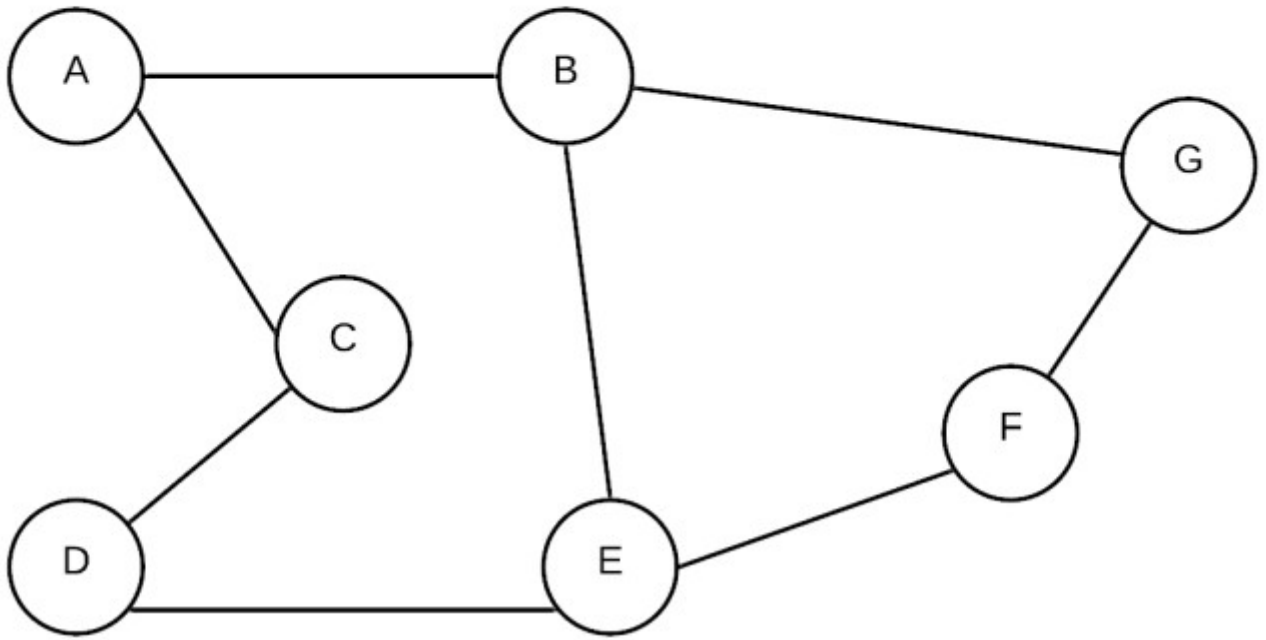
In this section of the notes we will define what a graph is and how the graph can be represented programmatically.

Graph Definition

The formal definition of a graph is as follows:

A Graph $G = (V, E)$ is made up made of a set of vertices $V = v_1, v_2, v_3...$ and edges $E = e_1, e_2, e_3....$ Each edge in E defines a connection between (u, v), where $u, v \in V$

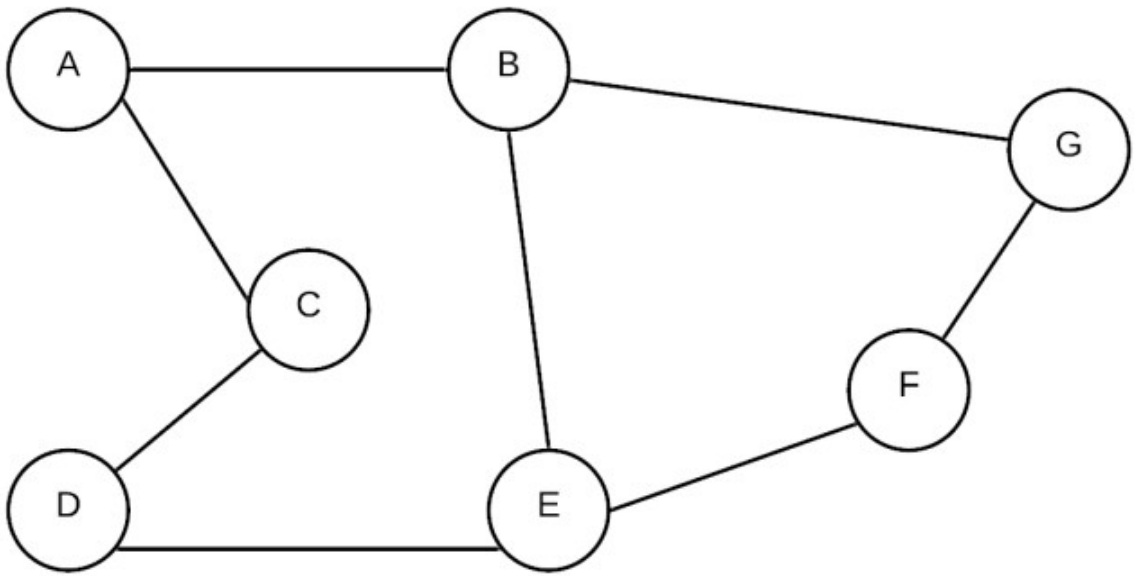
Basically this means that a graph is made up of vertices and edges. Each edge connects two of the vertices together. A graph is represented visually as follows:



Vertices

A vertex in a graph represents some type of object. The full graph is about relationships between these objects. For example, suppose you wanted to represent the flights between various airports for an airline. the airports would each be represented by a vertex.

When we look at a graph, we typically label each vertex. For example:



This graph has vertices A, B, C, D, E, F and G. If we were to represent the objects involved with a specific problem, we would store information about each object into records and store them into a table.

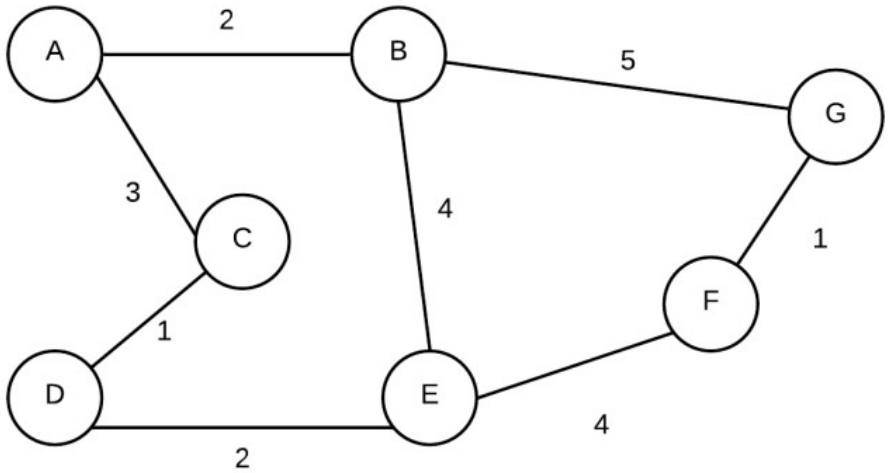
::: info

Note that the vertex labels A,B,C etc. are just labels. These labels are used to refer to the vertices in the diagram... in reality we don't necessarily store/name labels in this manner. Often vertices are just identified by a number or has some other identifying feature.

Edges

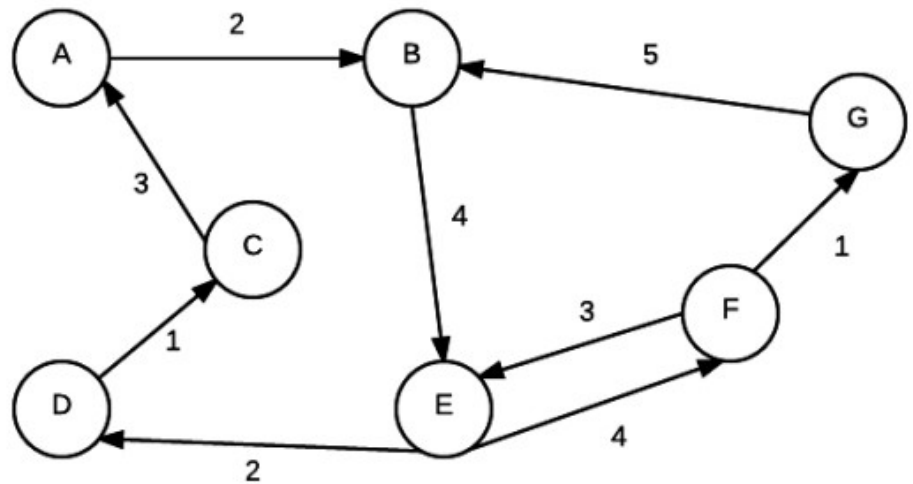
Edges represent a connection between two vertices.

Aside from identifying a connection between vertices, edges can also have weights. For example:



The weights on an edge can act in a way to serve as information about the nature of the connection between two vertices. For example, each vertex can represent a city and the weights, the distance between the cities.

Aside for weights, an edge may also include a direction. Graphs where edges have directions are also called a digraph. The following graph has both weights and direction:

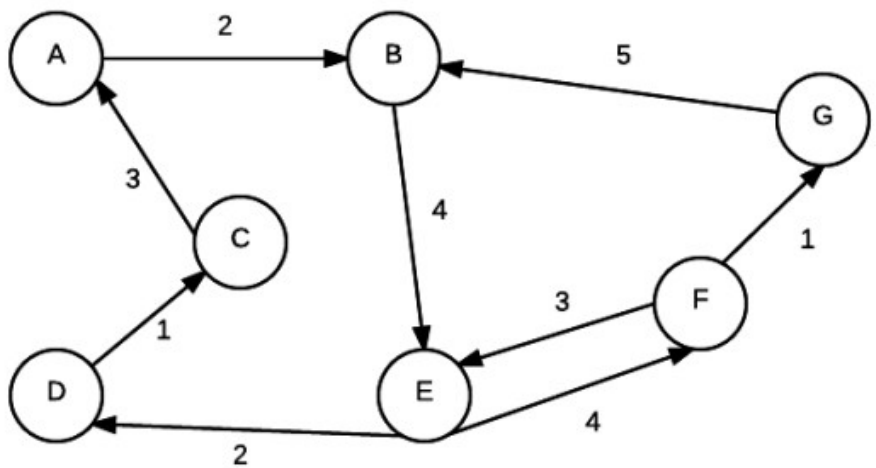


Other Definitions

- adjacent - Given two nodes A and B. B is adjacent to A if there is a connection from A to B. In a digraph if B is adjacent to A, it doesn't mean that A is automatically adjacent to B.
- edge weight/edge cost - a value associated with a connection between two nodes
- path - a ordered sequence of vertices where a connection must exist between consecutive pairs in the sequence.
- simplepath - every vertex in path is distinct
- path length - the number of the edges in a path.
- cycle - a path where the starting and ending node is the same
- strongly connected - If there exists some path from every vertex to every other vertex, the graph is strongly connected.
- weakly connected - if we take away the direction of the edges and there exists a path from every node to every other node, the digraph is weakly connected.

Representation

To store the info about a graph, there are two general approaches. We will use the following digraph in for examples in each of the following sections.



Adjacency Matrix

An adjacency matrix is in essence a 2 dimensional array. Each index value represents a node. When given 2 nodes, you can find out whether or not they are connected by simply checking if the value in corresponding array element is 0 or not. For graphs without weights, 1 represents a connection. 0 represents a non-connection.

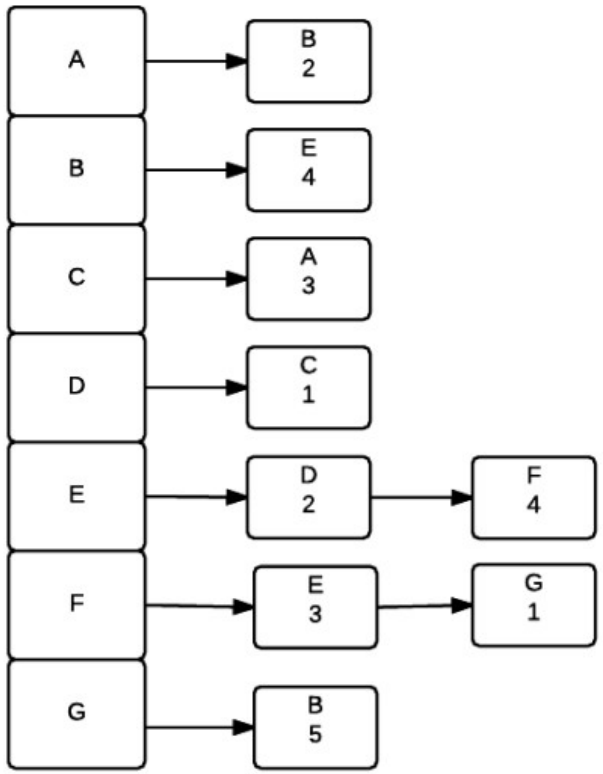
The graph adjancey matrix is a good represenation if the graph is dense. It is not good if the graph is sparse as many of the values in the array will be 0.

An adjacency matrix representation can provide an O(1) run time response to the question of whether two given vertices are connected (just look up in table)

	A-0	**B-1**	**C-2**	**D-3**	**E-4**	**F-5**	**G-6**
A-0	0	2	0	0	0	0	0
B-1	0	0	0	0	4	0	0
C-2	3	0	0	0	0	0	0
D-3	0	0	1	0	0	0	0
E-4	0	0	0	2	0	4	0
F-5	0	0	0	0	3	0	1
G-6	0	5	0	0	0	0	0

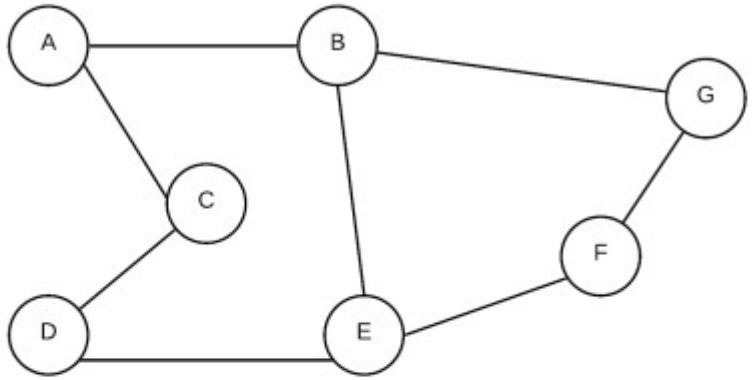
Adjacency List

An adjacency list uses an array of linked lists to represent a graph. Each element represents a vertex and for each (other) vertex it is connected to, a node is added to its linked list. For graphs with weights each node also stores the weight of the connection to the node. Adjacency lists are much better if the graph is sparse. It takes longer to answer the question of two given vertices are connected (Must go through list to check each node) but it can better answer the question given a single vertex, which vertices are directly reachable from that vertex.

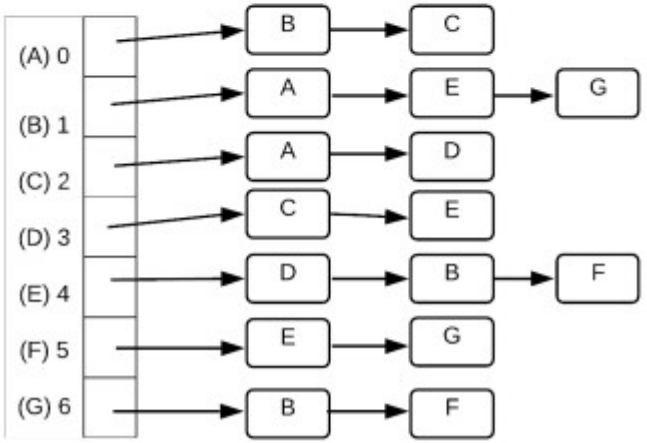


EXAMPLES

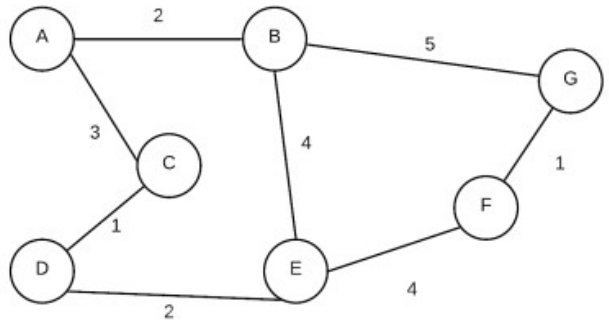
A graph with no weights or directions



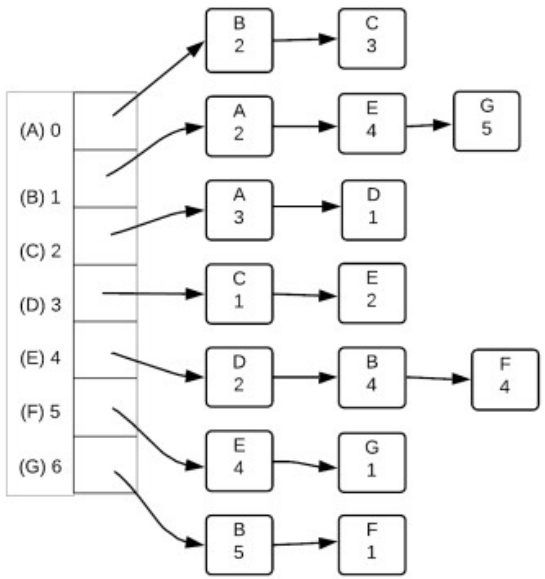
	(A) 0	(B) 1	(C) 2	(D) 3	(E) 4	(F) 5	(G) 6
(A) 0	0	1	1	0	0	0	0
(B) 1	1	0	0	0	1	0	1
(C) 2	1	0	0	1	0	0	0
(D) 3	0	0	1	0	1	0	0
(E) 4	0	1	0	1	0	1	0
(F) 5	0	0	0	0	1	0	1
(G) 6	0	1	0	0	0	1	0



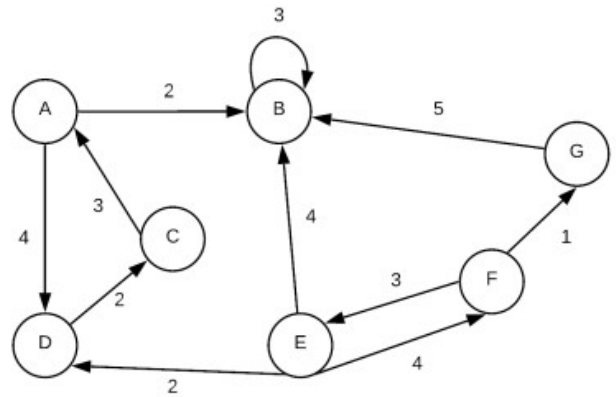
A graph with weights but no directions



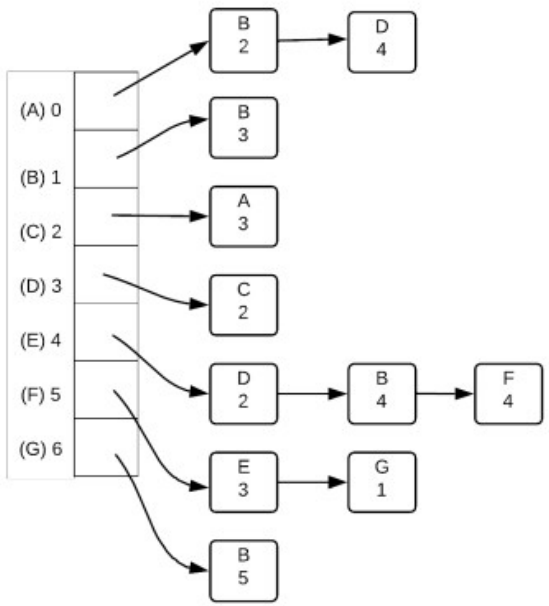
	(A) 0	(B) 1	(C) 2	(D) 3	(E) 4	(F) 5	(G) 6
(A) 0	0	2	3	0	0	0	0
(B) 1	2	0	0	0	4	0	5
(C) 2	3	0	0	1	0	0	0
(D) 3	0	0	1	0	2	0	0
(E) 4	0	4	0	2	0	4	0
(F) 5	0	0	0	0	4	0	1
(G) 6	0	5	0	0	0	1	0



A graph with weights and directions



from \ to:	(A) 0	(B) 1	(C) 2	(D) 3	(E) 4	(F) 5	(G) 6
(A) 0	0	2	0	4	0	0	0
(B) 1	0	3	0	0	0	0	0
(C) 2	3	0	0	0	0	0	0
(D) 3	0	0	2	0	0	0	0
(E) 4	0	4	0	2	0	4	0
(F) 5	0	0	0	0	3	0	1
(G) 6	0	5	0	0	0	0	0



DIJKSTRA’S ALGORITHM

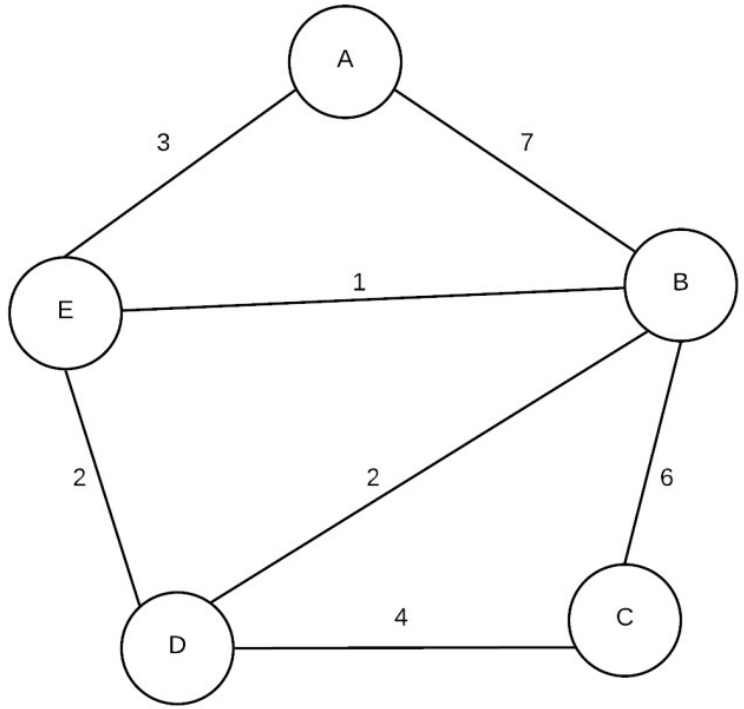
Dijkstra's Algorithm is an algorithm for finding the shortest path from one vertex to every other vertex. This algorithm is an example of a ***greedy algorithm.*** Greedy algorithms are algorithms that find a solution by picking the best solution encountered thus far and expand on the solution. Dijkstra's Algorithm was first conceived by Edsger W. Dijkstra.

The general algorithm can be described as follows:

1. Start at the chosen vertex (we'll call it v1).
2. Store the cost to travel to each vertex that can be reached directly from v1.
3. Next look at the vertex that is least costly to reach from v1 (we'll call this vertex v2 for this example).
4. For each vertex that is reachable from v2, find the total cost to that vertex starting from v1 (we want the total cost, not just the lowest cost from v2). Update the cost of travel to these vertices if it is lower than the current known cost.
5. Pick the next lowest cost and continue.

Example

Consider the following graph (non-directed)



Initial state

We will use Dijkstra's algorithm to find the shortest distance from **A** to every other vertex. To track how we are doing, we will create a table that will track the shortest distances from A we have encountered so far as well as the "previous" node in the path (so if we follow all previous nodes, we end up back at **A**). We also track whether we "know" about the shortest path from **A** to each node... that is, have we already been able to find the shortest path to that node from **A**. Currently (at the beginning,) we do not know which nodes are reachable from A so we store infinity into every spot other than A. The cost of getting to A from A is 0. Initially the table looks like this:

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		false
B	∞		false
C	∞		false
D	∞		false
E	∞		false

Expand A

Now, starting from **A**, there are two vertices that reachable, **B** and **E**

We store the total distance from **A** into the table for both these vertices if it is less than the shortest distance so far. At this point, **A** is now known as we have considered all its neighbours

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		**true**
B	**7**	**A**	false
C	∞		false
D	∞		false
E	**3**	**A**	false

Expand E

Now, we continue from here by considering the unknown vertex that has the shortest distance to **A**. In this case it is 3 (smallest in distance table with a false value in known). In implementation, we actually would use a priority queue (heap) to store the vertex with its distance and simply dequeue the smallest distance and check that the vertex was still unknown.

In any case, we are going to now consider **E**. From **E** the unknown neighbours are **B** and **D**. The distance to **D** (from A) is 5 (3 + 2). The distance to **B** is 4 (3 + 1). Now, The value stored previously for D was ∞ so we replace that with 5. The distance to **B** was 7, so we replace that also. We also mark that E is now known

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		true
B	**4**	**E**	false
C	∞		false
D	**5**	**E**	false
E	3	A	**true**

Expand B

The next vertex we consider is the one with the shortest distance to A that is still not known. This is now B.

From B, we can reach unknown neighbours C and D. The distance to C (from A) is 10 (4+6). The distance to D (from A) is 6 (4 + 2). The value stored in the table for C was ∞ so we replace that. The value stored in D was 5, so that entry is not replaced because this new distance is NOT smaller.

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		true
B	4	E	**true**
C	**10**	**B**	false
D	5	E	false
E	3	A	true

Expand D

The next vertex that with the smallest distance to A that is unknown is D. From D, there is only one unknown vertex (C). The distance to C is 9 (5 + 4). As this is smaller than what is stored, we will update the table

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		true
B	4	E	true
C	**9**	**D**	false
D	5	E	**true**
E	3	A	true

Expand C

The only vertex left is C. From C, there are no unknown neighbours so we mark C as known and we are done.

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		true
B	4	E	true
C	9	D	true
D	5	E	true
E	3	A	true

Shortest path

To find the shortest distance to A, we simply need to look it up in the final table.

However, if we want to find the shortest path it isn't so simple as a look up. We actually need to work away backwards by looking up the previous vertices. For example, to find shortest path from A to C, we see that C's previous was D. D's previous was E. E's previous was A. Thus, the path is:

A --> E --> D --> C

MINIMUM SPANNING TREES

A spanning tree is a connected, acyclic subgraph of a graph $G = (V, E)$. That is it is the subset of edges that are connected and acyclic. If G itself is not connected, then we can generalize this to a spanning forest.

A minimum spanning tree (MST) of a graph $G = (V, E)$ with weight function $w(e)$ for each $e \in E$, is a spanning tree that has the smallest total weight.

How to find a minimum spanning tree

To find an MST, we can either start with every edge in the graph and remove edges till we get an MST or we can select edges till we form an MST.

For the removal method, we can start out with every edge in the graph and start eliminating the edges with the highest weights. An edge can be removed as long as we don't disconnect some vertex (ie if that is the only edge to a vertex, we can't remove it). We keep removing edges until we have an MST. However, this is not practical. If there are n nodes in a graph, a spanning tree has n-1 edges. potentially there are $\frac{(n)(n-1)}{2}$ edges in total. This means that any algorithm that finds an MST in this manner would end up with worst case $\Omega(n^2)$

Instead of removing edges, we can find the MST by building the MST instead. The general generic algorithm for doing this is as follows:

```
greedyMST(G=(V,E)){
    T=[];  //empty set of edges
    while(T is not an MST){
        find an edge e from E that is safe for T
        add e to T
    }
}
```

Now... this is a very generic algorithm and there are parts not yet defined. Lets start with looking for an e being safe for T. An edge e is safe for T iff adding e to T makes T subset of some MST of G. Of course... problem is how do we know if e is part of some MST of G when that is exactly what this function is suppose to find?

INFO

Theorem: If G is a connected, undirected weighted graph and T is a subset of some MST of G and e is any edge of minimum weight that are in different connected components of T, then adding e is safe for T (See text for proof)

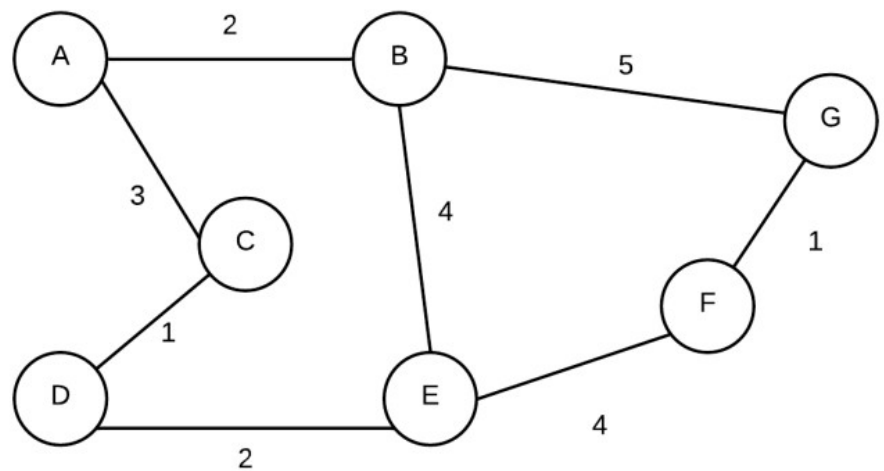
Kruskal's Algorithm

Kruskal's algorithm starts by sorting edges according to weights. It then picks the smallest edge out and adds it to T only if the end points are in different connected components. If we use something like a BFS or DFS to find connected components, it will be very slow. Instead, what we can do is use a disjoint set

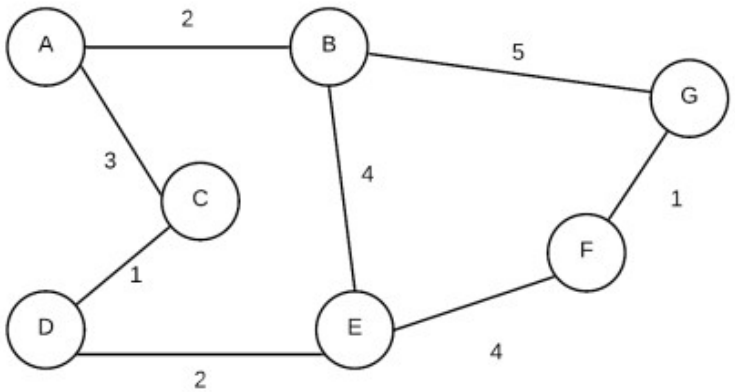
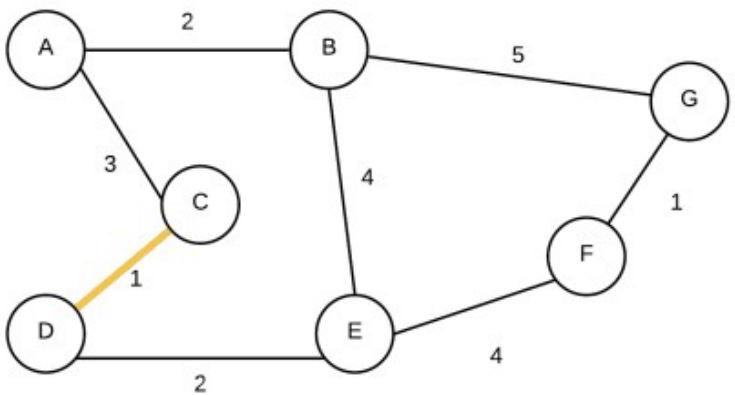
```
KruskalsMST(G=(V,E)){
    T=[];  //set of edges that form MST, initially empty
    sort edges in E by its weight

    //create a set for each vertex
    for(each vertex v in V) {
        makeSet(v);
    }

    for(each edge e=(u,v) in E){
        //find reps for endpoints of edge
        ep1=findSet(u);
        ep2=findSet(v);

        //if different reps then they are not in same set
        if(ep1!=ep2){
            //union them the set together
            union(ep1,ep2);

            //add e to the result
            add e to T;
        }
    }
}
```

Example

So let us consider the following graph:



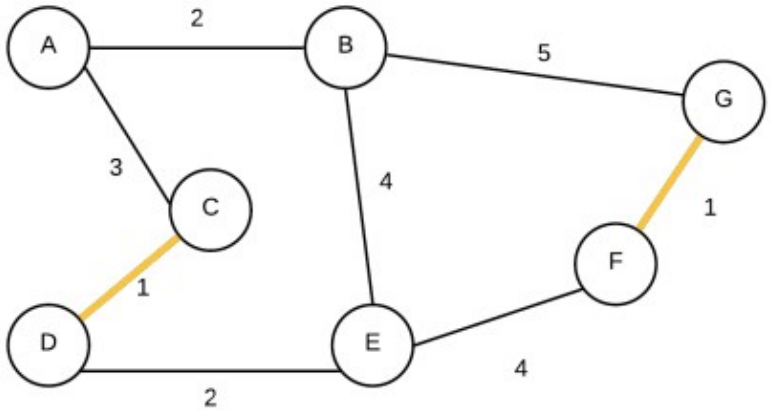
Given the graph above, the edges sorted in non-descending order by weight are: (C,D), (F,G), (A,B),(D,E),(A,C), (B,E), (E,F),(B,G). We will exam the edges in this order in the example below

Step #	graph, T shown in yellow	Disjoint Sets	Comments
Initial	 T= {}	{A}, {B}, {C}, {D},{E},{F}, {G}	every vert is in its own disjoint set, initial MST is empty
1	 T={(C,D)}	{A}, {B}, {CD}, {E},{F},{G}	(C,D) is added first. They were in different disjoint sets

2

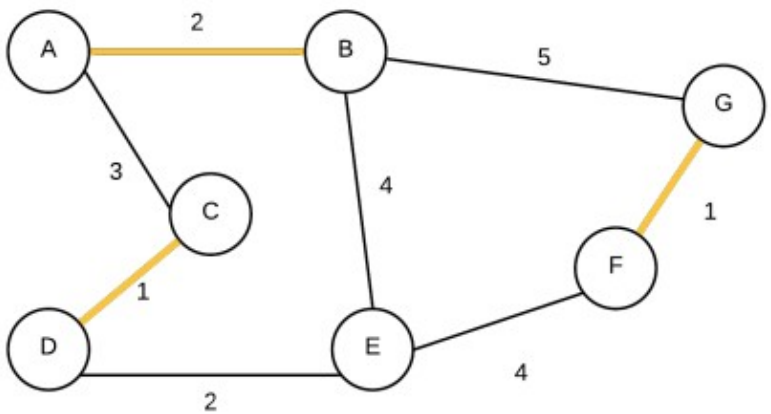
{A}, {B}, {C,D}, {E},{F,G}

(F,G) is added next. They were not in the same disjoint set



T={(C,D), (F,G)}

3

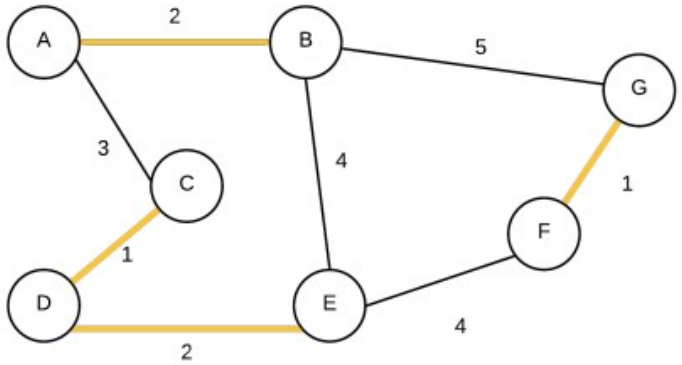


{A,B}, {C,D}, {E},{F,G}

(A,B) is added next as they were not in the same disjoint set

T={(C,D), (F,G),(A,B)}

4

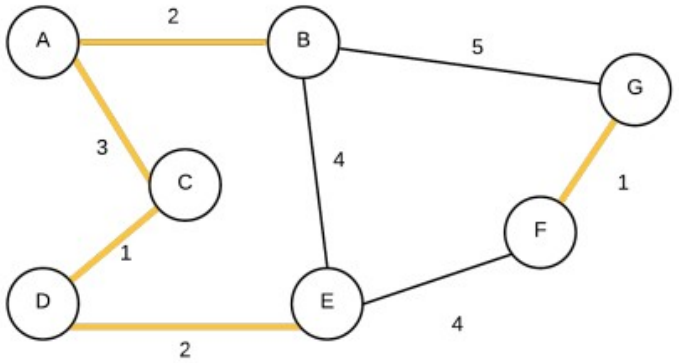


{A,B}, {C,D,E}, {F,G}

(D,E) is added next as they were not in the same disjoint set

T={(C,D), (F,G),(A,B),(D,E)}

5

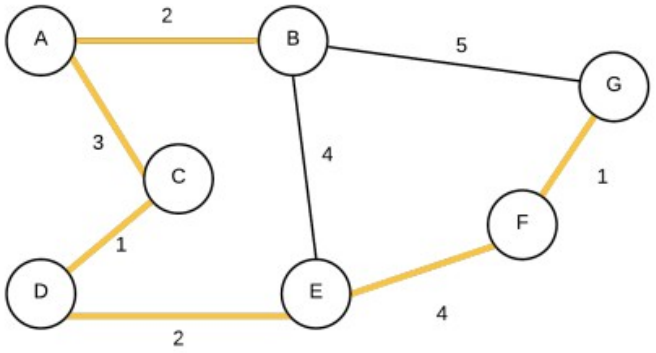


{A,B,C,D,E}, {F,G}

(A,C) is added next, they were not in the same disjoint set

T={(C,D), (F,G),(A,B),(D,E),(A,C)}

6



{A,B,C,D,E,F,G}

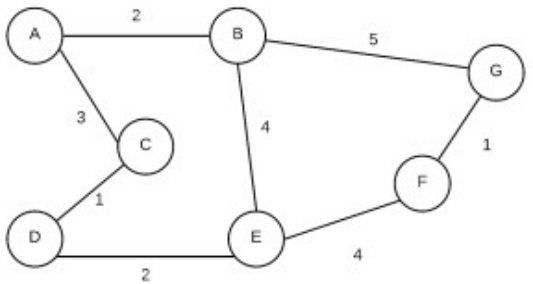
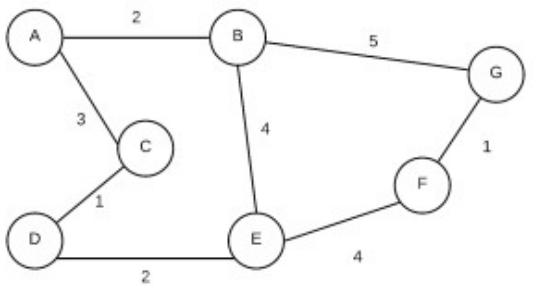
(B,E) was next but not added because B and E are in same disjoint instead.

T={(C,D), (F,G),(A,B),(D,E),(A,C)}

We next consider (E,F) and as they are in different sets we connect them. (B,G) are in same disjoint set so we are now done |

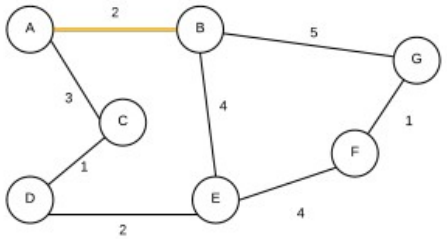
Prim's Algorithm

We pick a vertex $v \in V$ to be the "root" of the MST. After that we simply grow the tree by joining isolated vertices one at a time. An isolated vertex is any vertex that isn't part of the MST yet picking the smallest edge weight. To support this, we will use a MinHeap. We queue into this heap edges that will connect an isolated vertex with the current MST. We use infinity if there is no direct edge yet to any vertex in the MST

Step	Graph	Heap (vertex, parent, weight to parent), listed with by priority (smaller weight)	Comments
Initial		(A, NIL,0),(B, NIL,-),(C, NIL,-), (D, NIL,-),(E, NIL,-),(F, NIL,-), (G, NIL,-)	
Initial state. pick A as root			
1		(B, A,2),(C, A,3), (D, NIL,-),(E, NIL,-),(F, NIL,-), (G, NIL,-)	update edge weights, parents of B and C

T={}

2



(C, A, 3),(E, B, 4), (G, B, 5),(D, NIL,-),(F, NIL,-)

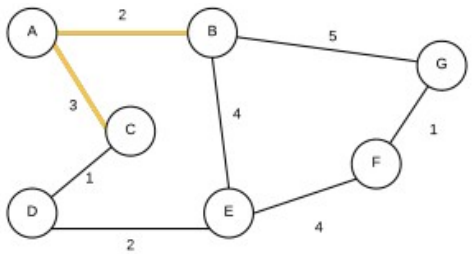
Take out vertex with smallest weight(B) and add edge to its parent to the MST. update edge weights and parents of E and G

T={(A,B)}

3

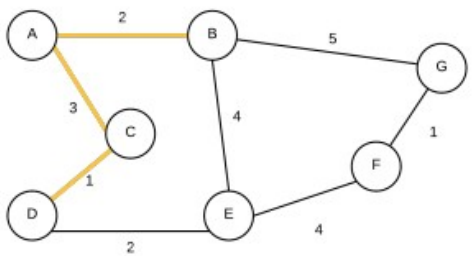
(D, C, 1),(E, B, 4), (G, B, 5),(F, NIL,-)

Take out vertex with smallest weight (C) and add edge to its parent to the MST. update edge weights and parent of D.



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

4

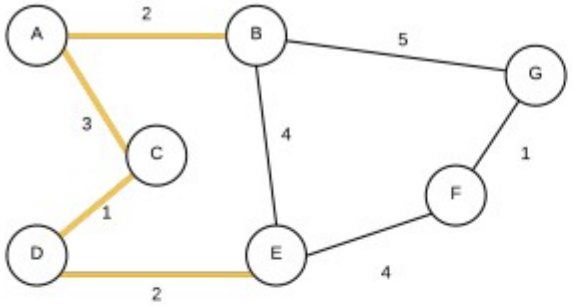


(E, D, 2),(G, B, 5), (F, NIL,-)

Take out vertex with smallest weight (D) and add edge to its parent to the MST. update edge weights and parent of E because cost to E is less going through D and not B.

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

5

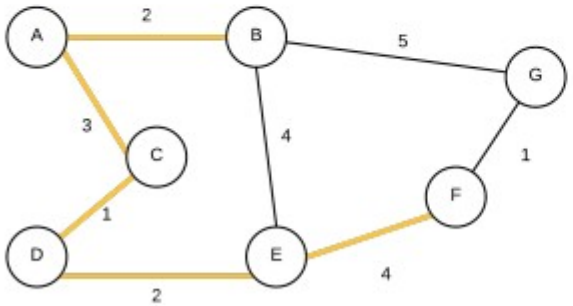


(F, E, 4),(G, B, 5)

Take out vertex with smallest weight (E) and add edge to its parent to the MST. update edge weights and parent of F

T={(A,B),(A,C),(C,D),(D,E)}

6

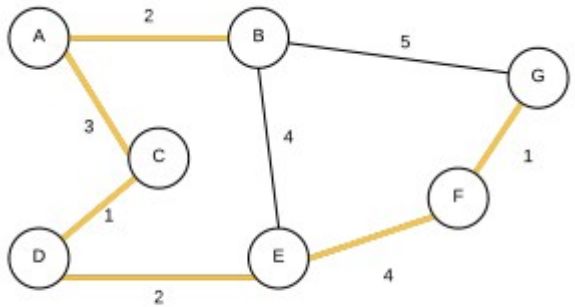


(G, F, 1)

Take out vertex with smallest weight (F) and add edge to its parent to the MST. update edge weights and parent of G because cost to G is less going through F and not B.

T={(A,B),(A,C),(C,D),(D,E),(E,F)}

7



empty

Take out vertex with smallest weight (G) and add edge to its parent to the MST.

T={(A,B),(A,C),(C,D),(D,E),(E,F),(F,G)}