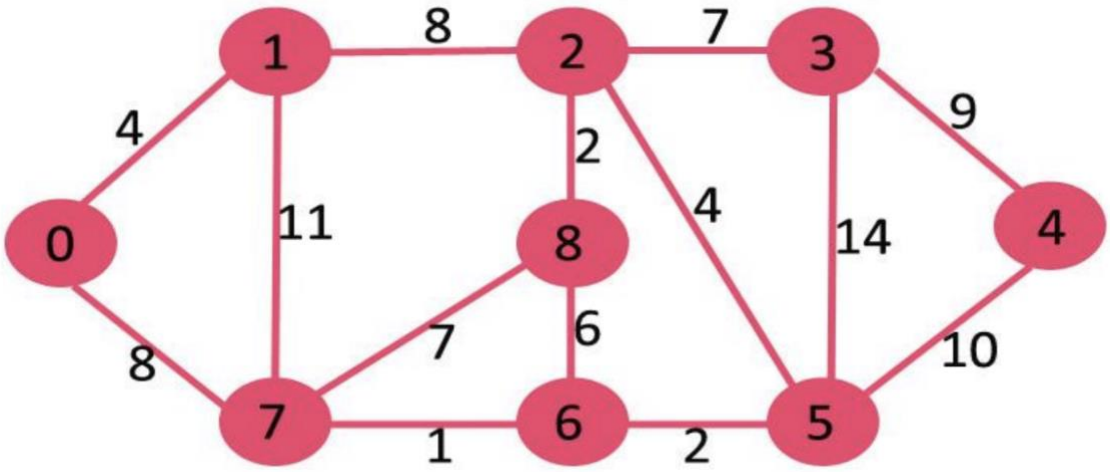
How does Kruskal's Algorithm Work? – Treats every node as an independent tree and connects 1 with another only if it has the lowest compared to all other options available (always select to not make a cycle)

Time Complexity for Kruskal's → O(ElogV)

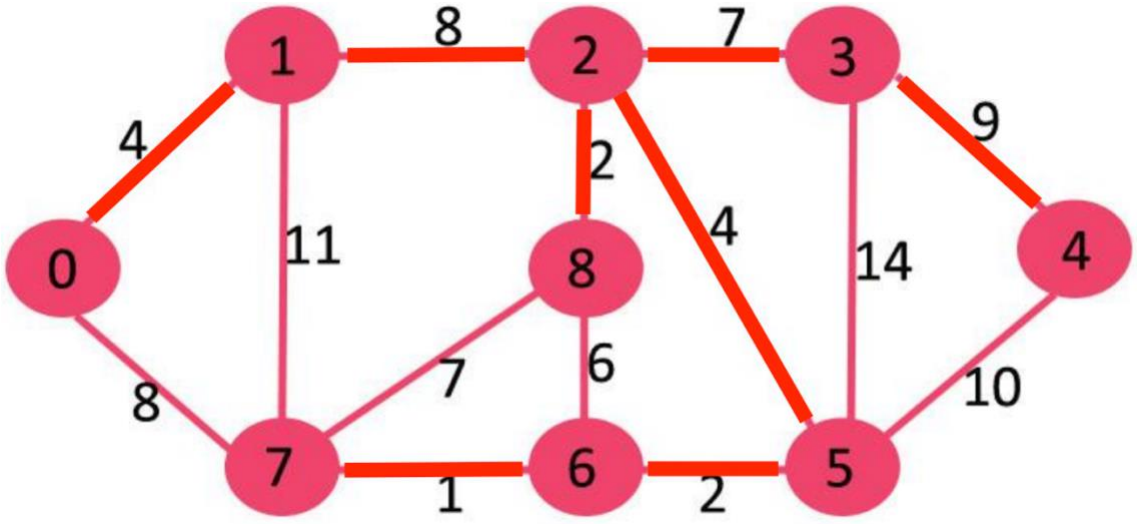
How does Prim's Algorithm Work? – Treats the nodes as a single tree and keeps on adding new nodes to spanning tree, a spanning tree means all vertices must be connected, so the 2 disjoint vertices of vertices must be connected to make a Spanning Tree, and they must be connected with the minimum weight to make it a Minimum Spanning Tree.

Time Complexity for Prim's → O(ElogV)

General Picture:



1. **a.** MST for Kruskal's Algorithm in picture and tabular form



Graph contains 9 vertices and 14 edges, so MST formed will be 8 edges (9-1) .

One Way

MST Edges	**6-7**	**2-8**	**5-6**	**2-5**	**1-0**	6-8	7-8	**3-2**	**1-2**	0-7	**3-4**	4-5	1-7	3-5
Edge weight	**1**	**2**	**2**	**4**	**4**	6	7	**7**	**8**	8	**9**	10	11	14

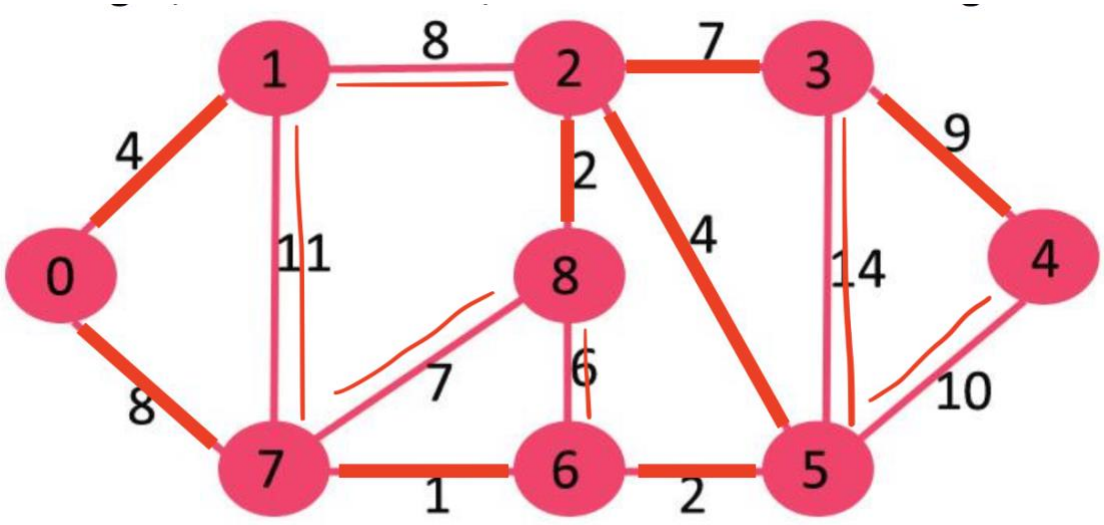
- bold is traversable path

Another Way

MST Edges	7-6	6-5	2-8	0-1	2-5	2-3	1-2	3-4
Edge weight	1	2	2	4	4	7	8	9

Total Edge weight for either way = 1+2+2+4+4+7+8+9 = 37

b. MST for Prim's Algorithm in picture and tabular form



One Way

MST Edges	8-2	2-5	5-6	6-7	7-0	0-1	2-3	3-4
Edge weight	2	4	2	1	8	4	7	9

Another Way

MST Edges	2-8	2-5	6-5	7-6	2-3	7-0	0-1	3-4
Edge weight	2	4	2	1	7	8	4	9

Total Edge weight for either way = 2+4+2+1+7+8+4+9 = 37

2. **a.** Kruskal's Algorithm implement cycle detection

Sort all edges in non-decreasing order of their weight

Pick smallest edge and check if it forms a cycle with the spanning tree formed so far and if cycle is not formed, include this edge (else, discard it)
Repeat the previous step until there are V-1 edges in the spanning tree

Cycle detection is also possible through Disjoint Set data structures. In this structure, every node belongs to a set. Initially, that set is just the node itself and the way we do this is that each node starts off with itself as its root. We start off at a vertex, with a specific root and as we go through the algorithm, every time we link to a new vertex, we set the root of that vertex equal to the root of the first vertex. This way, as we continue through algorithm, the roots will 1 by 1 change to the original root. We can detect a cycle when the next vertex's root is equal to the current vertex's root. When the root is equal, we do not choose that path, preventing a cycle to be formed.

Cycle detection with an concept similar to that of an adjacency matrix. You create a matrix of length V-1 and index each vertex from 0 to V-1. Any time an edge is added, you update the matrix to say edge V and W have the same value or both edges visited. (like a union find data structure)

b. Prim's Algorithm implement cycle detection

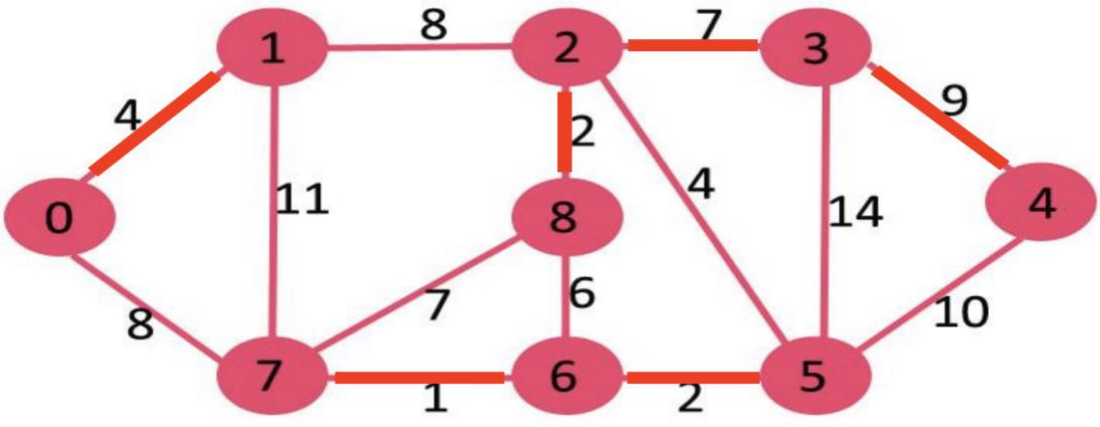
Cycle detection by using priority queue to sort the edge in descending order of edge weights. Then, you delete each edge one at a time to detect whether the graph is connected. If the graph is connected, then the edge contributed to that the graph is in a cycle.

Prim's algorithm can detect cycles by utilizing a priority queue. Sort all of the edges in descending order according to their respective edge weights. From here we delete an edge for each iteration and then check to see if graph is still connected and if it is still connected, there is a cycle.

c. If you tried to use Prim's cycle detection in Kruskal's algorithm then the graph would first use Kruskal's algorithm to check edge weights. Then for every edge found it will mark both vertices so that they cannot be visited again and it will then repeat this for as many edges as there are in the graph that can be visited.

When using Prim's Algorithm of cycle detection in Kruskal's algorithm, the algorithm fails because we are marking vertices as visited each time we add them into the priority queue. We cannot finish Kruskal's because you are not allowed to link things that are already marked. We find errors when we have already marked values as linked, since Kruskal's algorithm works by first listing out all the edge weights from lowest to highest and then linking the vertices.

Weight	Edges	Marked
1	7-6	6 & 7
2	6-5	5
2	2-8	2 & 8
4	2-5	Not possible (cannot link)
4	0-1	0 & 1
7	7-8	Not possible (cannot link)
7	2-3	3
8	1-2	Not possible (cannot link)
9	3-4	4
10	4-5	Not possible (cannot link)



Another table and explanation:

Edge	Weight	Visited
7-6	1	6
6-5	2	7
8-2	2	5
0-1	4	8
2-3	7	2
3-4	9	0
7	2-3	1
		3
		4

With Prim's cycle detection, edge 2-5 cannot be connected since 2 and 5 are both marked as visited nodes, therefore this edge is not eligible

Prim's method of cycle detection with Kruskal's algorithm is because Kruskal's algorithm is an edge first algorithm, which means that we organize the weights of our edges in order in a priority queue. Kruskal's algorithm would want to connected an edge but Prim's cycle detection will prevent that connection.

As what happened to edge 2-5, we see the same happen for edge 0-7, 1-2, and 8-6, which results in disjoint sets that cannot possibly connect.

3. If one applies Kruskal and Prim's algorithm to a directed graph, Prim's algorithm does not work, but can work for Kruskal's algorithm. In order for Kruskal's algorithm to work for a directed graph, one has make sure to ignore loops because if there is/are loops with small weight, the algorithm will count it on the MST. For Prim's Algorithm,

1) Create a set that keeps track of vertices already included in MST
2) Assign a key value (which picks the minimum weight edge from cut) to all vertices in input graph and initialize all key values as infinite, and assign key value as 0 for the 1st vertex for that it is picked first
3) While set does not include all vertices
    a. Pick a vertex w which is not there in set and has minimum key value
    b. Include w to set
    c. Update key value of all adjacent vertices of w and to update the key values, iterate through all adjacent vertices and for every adjacent matrix, if weight of edge w-v is less than the previous key value of w, update the key value as weight of w-v

Prim's algorithm cannot work for a directed graph, because in the highlight part of the algorithm above, the algorithm assumes every node is reachable from every node which may not be true for digraphs, and also if there is no directed edge from explored nodes of MST to remain

unexplored, the algorithm gets in a problem even though there are edges from unexplored nodes to explored nodes in MST.

Prim's algorithm fails because it assumes that every node is reachable from every node which is valid for undirected graphs but is not true for directed graphs. This means that when it finds a path of least weight, it goes to that vertex but since directed graphs do not allow to go opposite to the way the arrow points, some vertices are unreachable to the algorithm.

Kruskal's algorithm also fails for a similar reason. If an edge with a weight that is less than all other connected edges is found. Kruskal's algorithm will take that to be the next path. However, if this edge was a directed edge, the path cannot be taken, so the algorithm fails as well.

Pf. [Case 1] Kruskal's algorithm adds edge $e = v$–w to T.

- Vertices v and w are in different connected components of T.
- Cut = set of vertices connected to v in T.
- By construction of cut, no edge crossing cut is in T.
- No edge crossing cut has lower weight. Why?
- Cut property $\Rightarrow$ edge e is in the MST.

The red boxed part of this code would cause the error when this algorithm is used for directed edges. Similarly, if an edge is found that has lower weight, but the directed arrow is pointing away from that vertex, the algorithm cannot pick that path, which causes the algorithm to fail.

If you apply Kruskal's algorithm to a directed graph, the algorithm would not be able to work properly since the algorithm works based on increasing value of edge weights and not a specific traversal of a graph. The use of a directed graph would void the usage of Kruskal's as a result.

If you apply Prim's Algorithm to a directed graph, the algorithm would not be able to use the lowest edge weight of the traversed vertices since directed graphs does not let Prim's traverse in any direction. Thus Prim's Algorithm will be limited to edge weighted that point in a particular direction.

4. Lazy Prim code

Lazy:

```
public class LazyPrimMST
{
private boolean[] marked; // MST
vertices
```

```
private Queue<Edge> mst; // MST
edges
private MinPQ<Edge> pq; // crossing (and ineligible) edges
public LazyPrimMST(EdgeWeightedGraph G)
{
pq = new MinPQ<Edge>();
marked = new boolean[G.V()];
mst = new Queue<Edge>();
visit(G, 0); // assumes G is connected (see Exercise 4.3.22)
while (!pq.isEmpty())
{
Edge e = pq.delMin(); // Get lowest-weight
int v = e.either(), w = e.other(v); // edge from pq.
if (marked[v] && marked[w]) continue; // Skip if ineligible.
mst.enqueue(e); // Add edge to tree.
if (!marked[v]) visit(G, v); // Add vertex to tree
if (!marked[w]) visit(G, w); // (either v or w).
}
}
private void
visit(EdgeWeightedGraph G, int v)
{ // Mark v and add to pq all edges from v to unmarked vertices.
marked[v] = true;
for (Edge e : G.adj(v))
if (!marked[e.other(v)]) pq.insert(e);
}
public Iterable<Edge> edges()
{ return mst; }
public double weight() // See Exercise 4.3.31.
}
```

Eager Prim Code

```
public class PrimMST
{
private Edge[] edgeTo; // shortest edge from tree vertex
private double[] distTo; // distTo[w] = edgeTo[w].weight()
private boolean[] marked; // true if v on tree
private IndexMinPQ<Double> pq; // eligible crossing edges
public PrimMST(EdgeWeightedGraph G)
{
edgeTo = new Edge[G.V()];
distTo = new double[G.V()];
marked = new boolean[G.V()];
for (int v = 0; v < G.V(); v++)
```

```
distTo[v] = Double.POSITIVE_INFINITY;
pq = new IndexMinPQ<Double>(G.V());
distTo[0] = 0.0; pq.insert(0, 0.0); // Initialize pq with 0, weight 0.
while (!pq.isEmpty())
visit(G, pq.delMin()); // Add closest vertex to tree.
}
private void visit(EdgeWeightedGraph G, int v)
{ // Add v to tree; update data structures.
marked[v] = true;
for (Edge e : G.adj(v))
{
int w = e.other(v);
 if (marked[w]) continue; // v-w is ineligible.
if (e.weight() < distTo[w]) { // Edge e is new best connection from tree to w.
edgeTo[w] = e;
distTo[w] = e.weight();
if (pq.contains(w)) pq.change(w, distTo[w]);
else pq.insert(w, distTo[w]);
}
}
}
public Iterable<Edge> edges() // See Exercise 4.3.21.
public double weight() // See Exercise 4.3.31.
}
```

**difference* →

Lazy Prim's: BFS and bound first → *better bounds faster*
Eager Prim's: DFS and branch first → *better solutions faster*

One Explanation

The difference between eager and lazy prim's algorithm is that lazy runs the algorithm through the usage of a priority queue which stores the edge weights and edges in increasing order and a MST. This is done by listing all the adjacent vertices of a given vertex and storing the edges and their edge weights in the priority queue. Then, the minimum among these edges are found and deleted from the PQ and inserted into the MST. This is done again to the next vertex, followed by traversing the graph by finding the next minimum edge. While completing the process, we also eliminate any redundant edges, i.e. edges that would connect 2 vertices that already have an edge connecting the 2.

On the other hand, eager Prim's algorithm does something similar to lazy prim algorithm in that it still uses PQ which sorts by increasing edge weights and an MST. But, instead of ONLY

constantly finding the minimum edge to traverse through, it stores relevant connections to a given vertex using the same technique as lazy Prim's Algorithm

Unlike the lazy prim method, eager prim does not enqueue any extra values into the PQ, and while it is doing this, it can update the shortest path found for any given vertex (similar to the process done in number 1 part b).

Another Explanation

Lazy Prim's runs the algorithm through the usage of a priority queue which sorts the edges in increasing weights and an MST using only the edges and their respective edge weights. This is done by listing all of the adjacent vertices of a giver vertex and storing the edges and their edge weights in a priority queue. Then, the minimum among these edges are found and deleted from the priority queue and inserted into the MST. This is done again to the next vertex, and then traversing the graph by finding the next minimum edges. During this, it also eliminates any repeated edges that would connect two vertices that already have an edge connecting them.

Eager Prim's algorithm is similar because it uses a Priority queue which sorts the edges by increasing weights and a MST. However, the eager algorithms only stores the relevant paths to a given vertex using the same method as lazy Prim's algorithm. While it does this, it updates the shortest paths to the given vertex. Eager doesn't enqueue extra values into the Priority Queue, which means Eager Prim uses less space than Lazy Prim.

Difference in the code are highlighted above.

The time complexity for Lazy Prim and Eager Prim is E log V because both Lazy and Eager implement a priority queue.

However, the space complexity for eager is less than that of Lazy because it does not store every single vertex in the priority queue.

Time complexity for Lazy Prim and Eager Prim is E log V, where E is the number of edges and V is the vertices, and the space complexity is going to be O(V).

Furthermore:

Average Time complexity of Prim's Algorithm is $O(V\log V + E\log V) = O(|E|\log|V|)$

Worst Time complexity of Prim's Algorithm is $O(|V|^2)$

Worst Space complexity of Prim's Algorithm is $O(|V|+|E|)$

5. a. pseudocode (in event "an edge is added to the graph G")

If we want to add an edge to a MST, we have to find a way to optimize the existing MST. For that to be the case, we have to "replace" the existing maximum in the MST with the potential new edge. If the new edge is larger than the existing maximum weight in the MST, the MST stays the same.

1. For a given MST, S (random variable), check the maximum edge weight in the tree

    a. Do this by running a maximum function Max(S), finding the maximum edge weight

2. Find the new edge's edge weight.

3. Add edge to the PQ.

    a. If new edge has edge weight < Max(T)
        Add the new edge into the MST
    b. Else
        Don't add in MST

a. another pseudocode (in event "an edge is added to the graph G")

1. Run a BFS on the MST to find the maximum edge weight
2. Compare that edge weight with the edge weight of the added edge
3. If the edge already in the tree is smaller than the edge weight of the new graph then the MST is already optimized
4. Else replace the edge in the tree with the newly added edge to create the most optimal path
5. The runtime is going to be O(V).

Explanation:

when a new edge is added into the graph, it will affect in MST, only if the cycle is formed by taking edges of minimum spanning tree along with new edge. IF the new edge has maximum weights in the cycle, then minimum spanning tree will remain the same, otherwise add this edge to MST and remove the edge with maximum weight in the cycle from tree.

Time complexity:

Detecting the formed cycle is MST after adding new edge will take O(V) time and comparing the edge weight in the cycle will take O(V) time because there at most V-1 edges in the tree, so the overall complexity will be O(V).

b. pseudocode (in event "an edge of not in the tree in made smaller")

1. Run a BFS on the MST to find the maximum edge weight
2. Compare that edge weight with the edge weight in the graph that was made smaller
3. If the edge in the tree is smaller than edge weight in the graph than the tree is already optimized
4. Else replace the edge in the tree with the edge from the graph to create the new MST.
5. The runtime is going to be O(V)

Explanation:

If the cycle form by taking MST along with the edge whose weight have been reduced and if reduced edge has maximum edge in the cycle, then MST remains the same otherwise replace maximum weight edge in this cycle from MST and add this reduced edge in the MST.

Time complexity:

This task is same as part a and it will also take O(V) time.

c. pseudocode (in event "an edge in the tree is made larger")

1. Run a BFS on the graph to find the minimum edge weight
2. Compare that edge weight with the edge weight that was increased in the tree
3. If the edge in the tree is smaller than the edge weight of the graph then the minimum spanning tree is already optimized
4. Else replace the edge in the tree with the edge from the graph to create the new MST.
5. The runtime is going to be O(V)

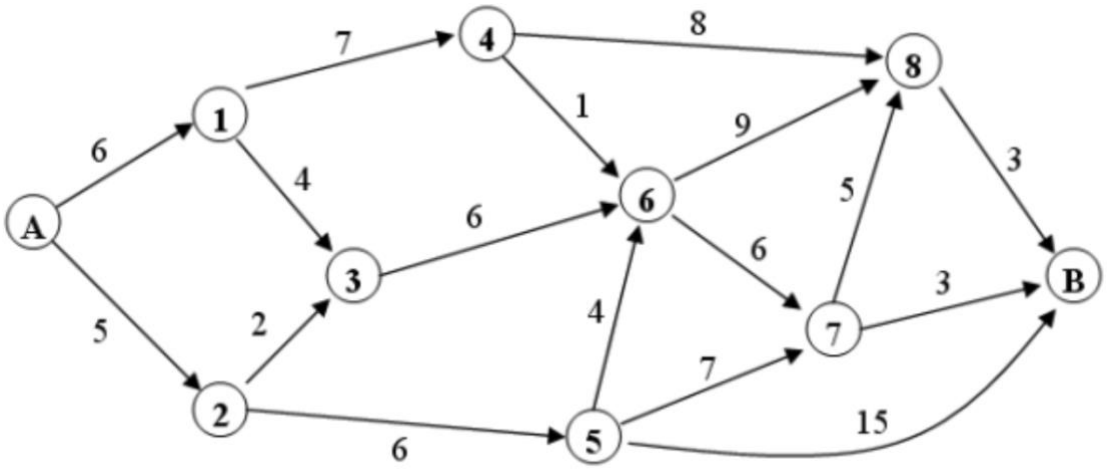
Explanation:

If the edge in the tree becomes larger, we need to see if all the cycles in the original graph which contains this edge. If this edge IS maximum weighted edge in at-least 1 cycle, then MST will delete this edge and add the minimum edge in that cycle and if this edge is NOT maximum weighted edge among any cycle, then MST will not change.

Time Complexity:

The complexity to detect cycle and compare with all edge in the cycle formed in graph could take O(V+E) time.

6. General Picture:



a. Bellman Ford Algorithm (Time complexity → O(VLogV))

An algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. (does work with all negative edge weights unlike Dijkstra's algorithm).

A-1, A-2, 1-3, 1-4, 2-3, 2-5, 3-6, 4-6, 4-8, 5-6, 5-7, 5-B, 6-7, 6-8, 7-8, 7-B, 8-B

One-way

Iteration 1
1st :

V	A	1	2	3	4	5	6	7	8	B
distTo	0	6	5	7	13	11	13	18	21	21
edgeTo	-	A-1	A-2	2-3	1-4	2-5	3-6	5-7	4-8	7-B

Iteration 2
2nd :

V	A	1	2	3	4	5	6	7	8	B
distTo	0	6	5	7	13	11	13	18	21	21
edgeTo	-	A-1	A-2	2-3	1-4	2-5	3-6	5-7	4-8	7-B

Another-way

Iteration 1
1st :

V	A	1	2	3	4	5	6	7	8	B

distTo	0	6	5	7	13	11	13	18	21	21
edgeTo	-	A-1	A-2	2-3	1-4	2-5	3-6	5-7	4-8	7-B

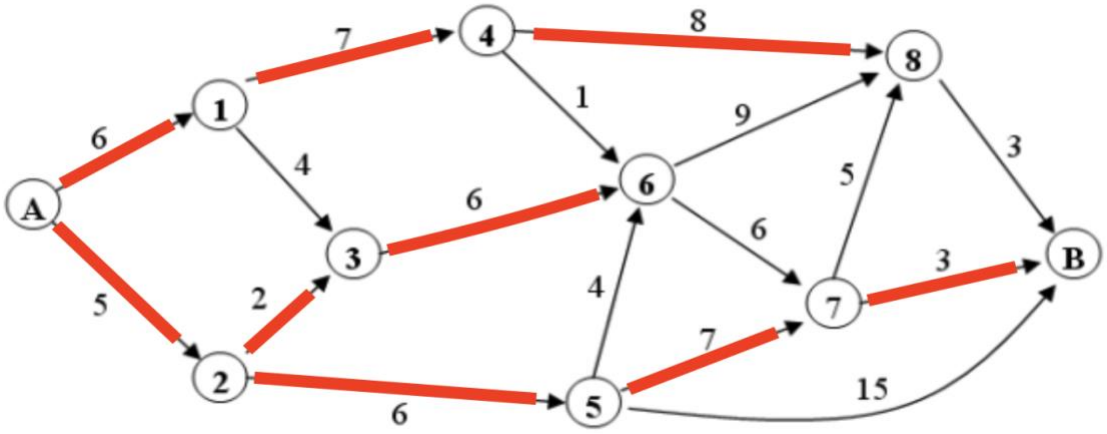
Iteration 2
2nd :

V	A	1	2	3	4	5	6	7	8	9
distTo	0	6	5	7	13	11	13	18	21	26
edgeTo	-	A-1	A-2	2-3	1-4	2-5	3-6	5-7	4-8	5-B

Iteration 2 results in NO changes → STOP

Iteration 2 results in no changes to the optimal path, which means there will be no changes to the tables in coming iterations as well, so this is where we can stop

The graph results in the following (no changes so this is optimal path) →



A-1, A-2, 1-3, 1-4, 2-3, 2-5, 3-6, 4-6, 4-8, 5-6, 5-7, 5-B, 6-7, 6-8, 7-8, 7-B, 8-B

** If there is/are (a) negative weight cycle(s), Bellman Ford Algorithm fails*

b. Dijkstra's Algorithm (based on lowest to high weight, Time complexity (with Fibonacci heap) → O(|E| + |V|log|V|)

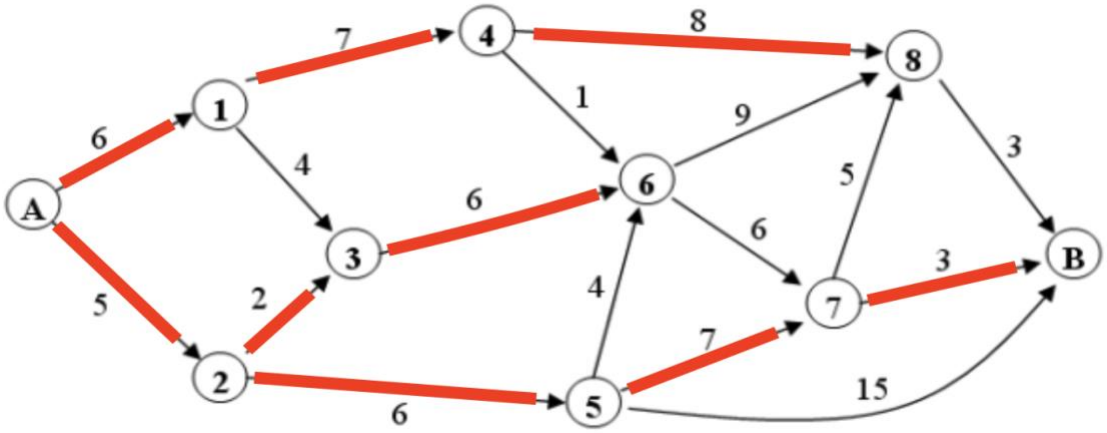
An algorithm for finding the shortest paths between nodes in a graph. (does NOT work with some negative edge weights, but Bellman Ford Algorithm works for all negative edge weights)

v	A	1	2	3	4	5	6	7	8	B

distTo	0	6	5	7	13	11	13	18	21	21
edgeTo	-	A-1	A-2	2-3	1-4	2-3	3-6	5-7	4-8	7-B
relaxCount	0	1	1	1	1	1	1	1	1	2

A, 2, 1, 3, 5, 4, 6, 7, B, 8

A-1, A-2, 1-3, 1-4, 2-3, 2-5,3-6,4-6,4-8,5-6,5-7,6-7,6-8,7-8,7-B,8-B



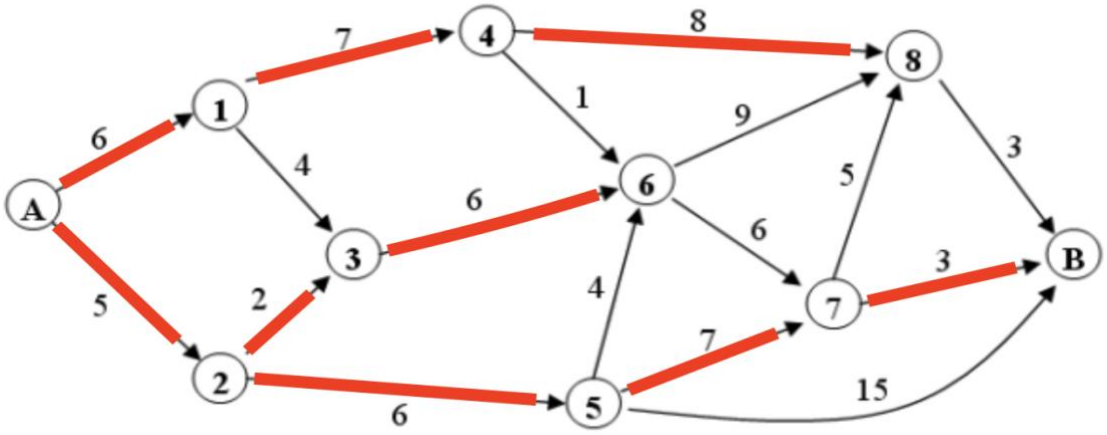
** With some negative edge weight(s), Dijkstra's Algorithm fails*

c. Topological Sort Algorithm (same Time Complexity as DFS → O(V+E))

An Algorithm of a directed graph that is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

v	A	1	2	3	4	5	6	7	8	B
distTo	0	6	5	7	13	11	13	18	21	21
edgeTo	-	A-1	A-2	2-3	1-4	2-5	3-6	5-7	4-8	7-B
relaxCount	0	1	1	2	1	1	1	1	1	2

Topological Sort order → A,1,2,3,4,5,6,7,8,B



**If the graph IS not a DAG, Topological Sort Algorithm fails*

7. Topological Sort Algorithm requires a DAG (directed acyclic graph), since topological ordering is possible if and only if the graph has no directed cycles. Topological sorted for DAG is a linear ordering of vertices such that for every directed edge vw, vertex v comes before w in the order. When running algorithm on a non-DAG there are errors because with directed cycles, no matter which point in the cycle one arrives at, there is no possible way to satisfy the topological sort the algorithm will not possibly work.

The topological sort algorithm requires a directed acyclic graph to run. This is because of the fact that we need to first order the vertices in topological order. This is only possible if and only if the graph has no directed cycles.

Another cause of algorithm's failure is because of the fact that in undirected graphs we can go backwards through the graph and visit vertices that have already been visited. This would interfere with our topologically ordered vertices because once they are ordered, we go through them one by one without going backwards in order to find the minimal path. If we are allowed to go backwards we can even go back to the source vertex and create cycles without finding the minimal path.

Code:

```
public class AcyclicSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;

   public AcyclicSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      Topological topological = new Topological(G);
      for (int v : topological.order())
         for (DirectedEdge e : G.adj(v))
            relax(e);

   }
}
```

When the code above is run for a non-DAG, the topological order is affected. When the topological order is created, we have to maintain that order, however when the graph is undirected this is not what happens. The DirectedEdge also causes an error because the edges are no longer directed, which means the algorithm is allowed to go backwards and forwards without regards to the topological order.

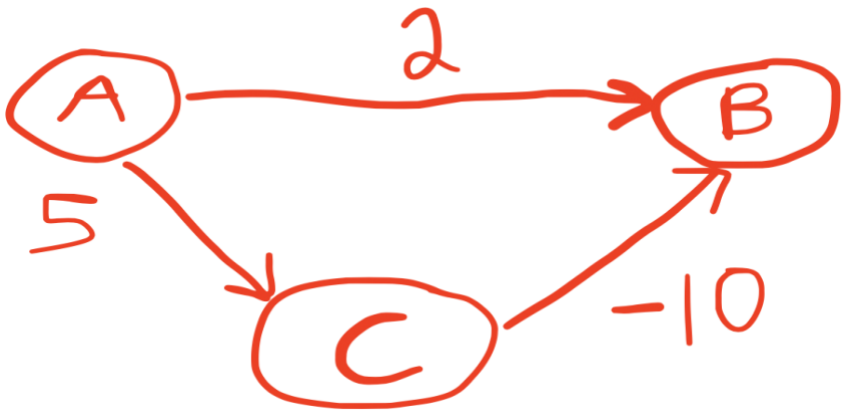
8. Prim's Algorithm works by expanding the tree by finding the shortest path from the source vertex to the tree and on the other hand, Dijkstra's Algorithm works by expanding the tree by finding the shortest path from the source to the next adjacent vertex. Since Prim's Algorithm looks at the entire tree when expanding, it goes backwards as well to find shorter paths. When this motion happens, there is a possibility of creating a cycle, which is why it is necessary to have cycle detection.

Dijkstra's algorithm always chooses the shorter one out of the possible paths from the source and then goes forward from that vertex. Due to this, there is no possibility of it going backwards,

which means there would not be any cycles being formed, so, therefore, you wouldn't need the cycle detection in Dijkstra's algorithm.

9. Unlike Bellman Ford Algorithm which is capable to handle negative weights, Dijkstra's Algorithm fails for some negative edge weights since it selects a next edge to add to the shortest part tree (spanning tree such weight of paths is minimum from given source node to all other nodes) it selected the one which minimum weight and is reachable from already reached nodes. The greed part comes when a next edge with minimum weight is added, the weight of the path from source to new node of that next edge will be minimum, so it becomes the shortest path and the greed property will only hold if weights are positive. Using some negative weights, will no longer guarantee an optimal path and so therefore the idea of minimum no longer exists.

Dijkstra's Algorithm will fair for negative edge weights because in the first step of the algorithm. It will take the minimal weight path to the node. When this happens, Dijkstra's assumes that it has already found the minimal weight path, but when there are negative edge weights in the graph they could be completely omitted.



For example, in this graph above, initially we could choose vertex A and relax all adjacent vertices. From there we could choose the minimal edge weight, so we could choose between A-B and A-C which have edge weights of 2 and 5, respectively. We could choose A-B because it has an edge weight of 2 which is less than an edge weight of 5. This is where it will fail because we have completely omitted the C-B edge weight of -10, which would actually give us the true minimal path.

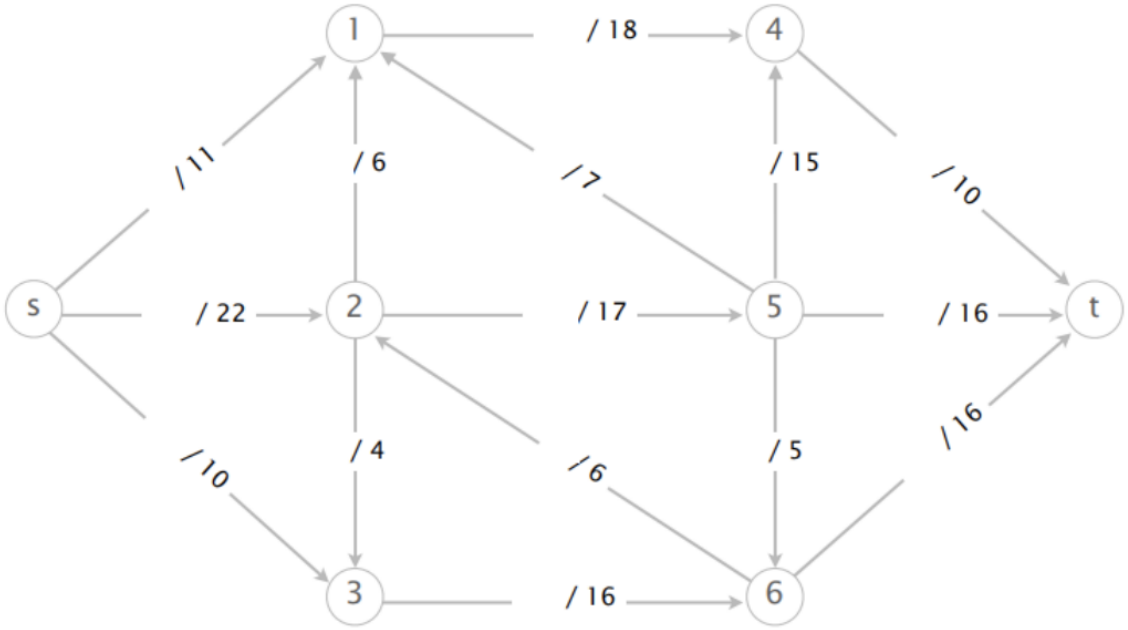
Code:

```
public class DijkstraSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(G.V());

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      pq.insert(s, 0.0);
      while (!pq.isEmpty())
      {
         int v = pq.delMin();
         for (DirectedEdge e : G.adj(v))
            relax(e);
      }

   }
}
```

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
      distTo[w] = distTo[v] + e.weight();
      edgeTo[w] = e;
      if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
      else                pq.insert     (w, distTo[w]);
   }
}
```

The red boxed part of the code above is what causes the error in the Dijkstra's Algorithm because initially when the distance of v is less than w, it chooses that path without regard for the other ones. This is where the omission happens, where the other path might lead to a shorter path because it leads to a negative edge weight which reduces the minimal path.

10. a.



Using Ford Fulkerson's Algorithm find Maximum Flow

1st iteration: s→1→4→t

Vertex	S	1	4	T
Flow/Capacity	-	10/11	10/18	10/10

t = 10

2nd iteration: s→3→6→t

Vertex	S	3	6	T
Flow/Capacity	-	10/10	10/16	10/16

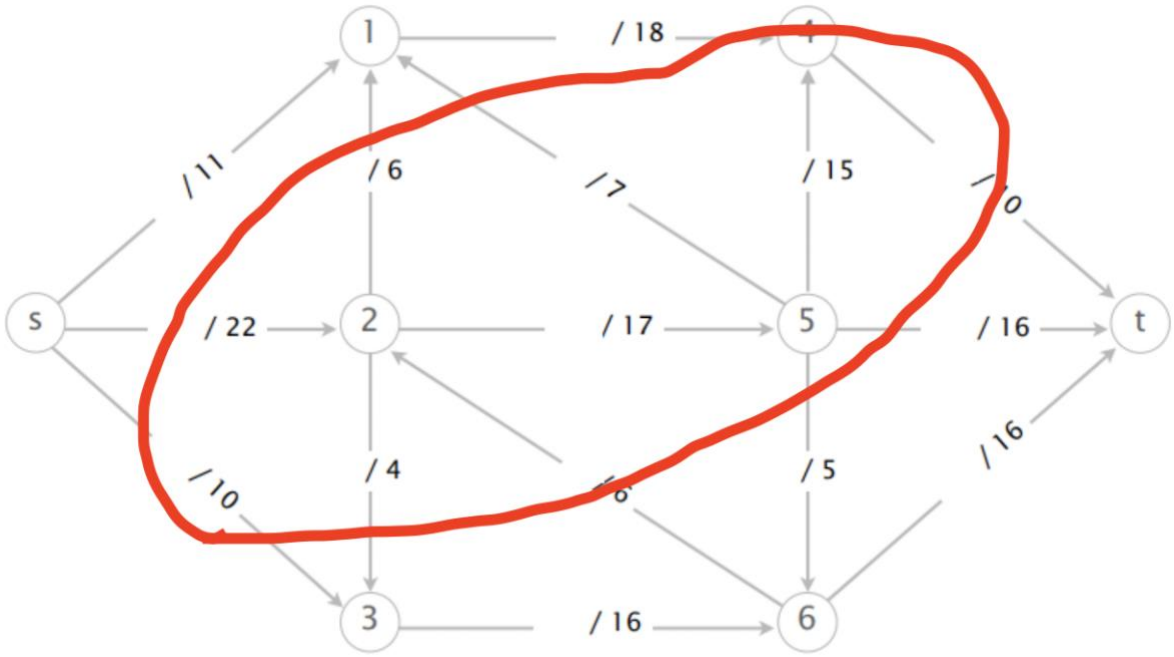
t = 20

3rd iteration: s→2→3→4→6→t→5→t

Vertex	S	2	3	5	New 6	New t	T from 5
Flow/Capacity	-	21/22	4/4	17/17	14/16	14/16	16/16

t = 41

b.



So the set that contains the mincut is going to contain S-3, 2-3, 4-t, 2-5

c. Mincut capacity = max flow value = 41