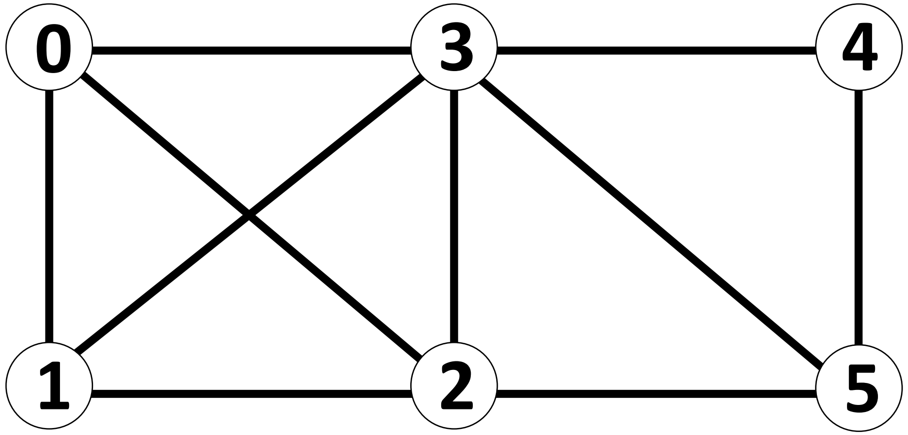
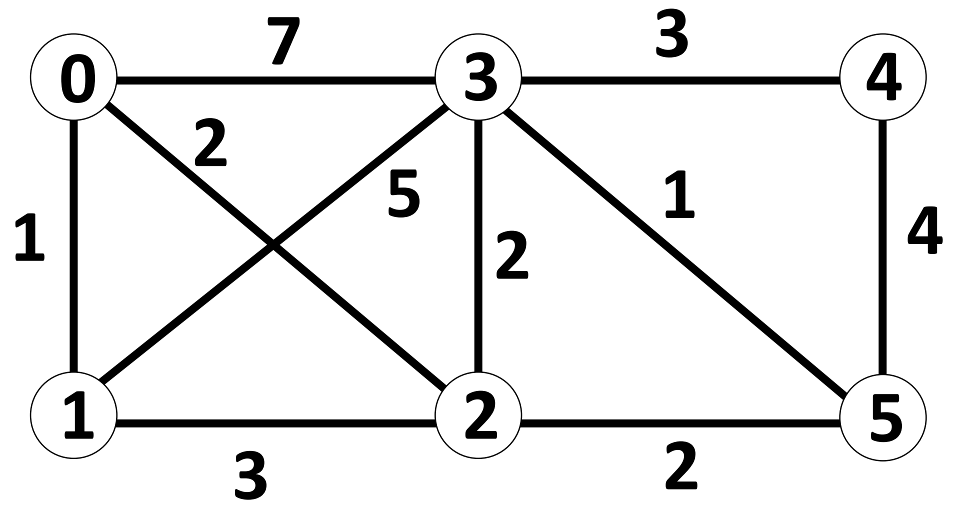
Minimum Spanning Trees

CSCI 232

Minimum Spanning Tree

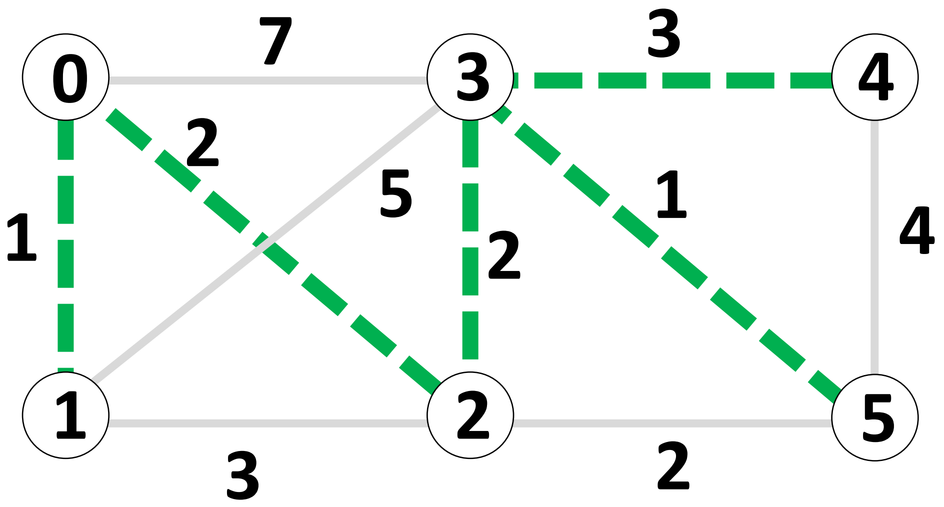


Minimum Spanning Tree



Edge-weighted graph: A graph where each edge has a weight (cost).

Minimum Spanning Tree

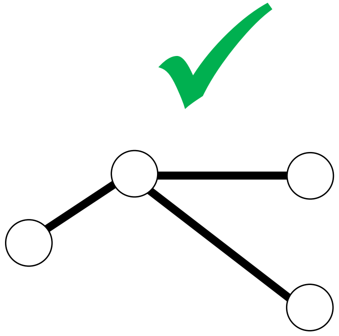


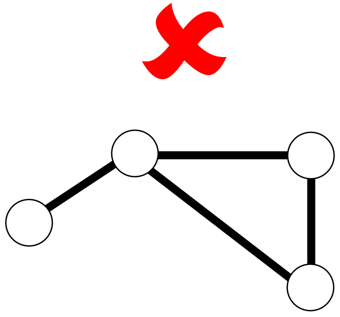
Edge-weighted graph: A graph where each edge has a weight (cost).

MST Goal: Connect all vertices to each other with a minimum weight subset of edges.

Minimum Spanning Tree

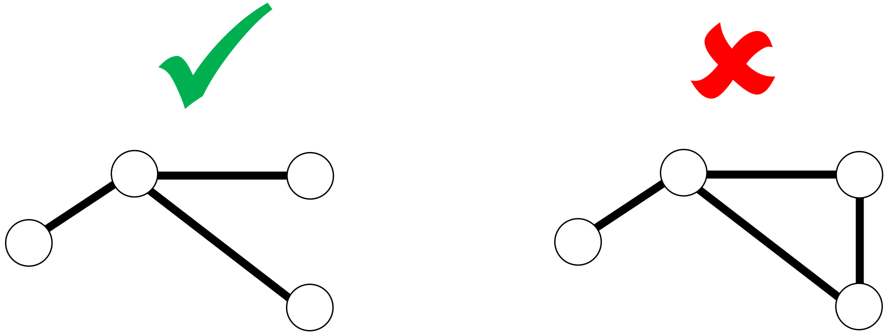
Tree – connected graph with no loops.



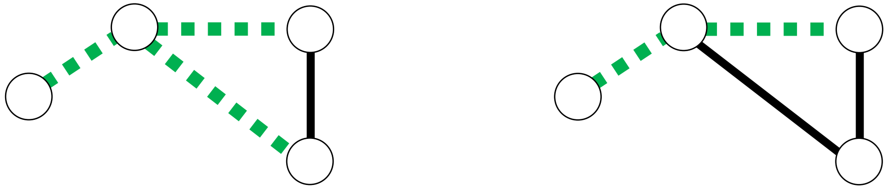


Minimum Spanning Tree

Tree – connected graph with no loops.

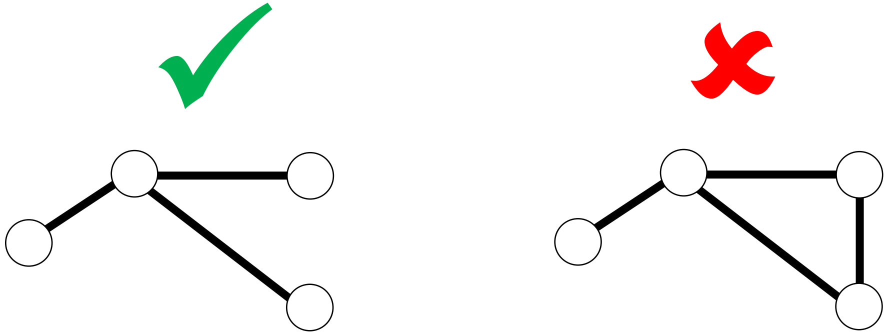


Spanning tree – tree that includes all vertices in a graph.

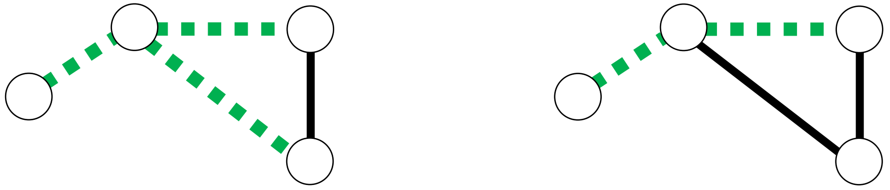


Minimum Spanning Tree

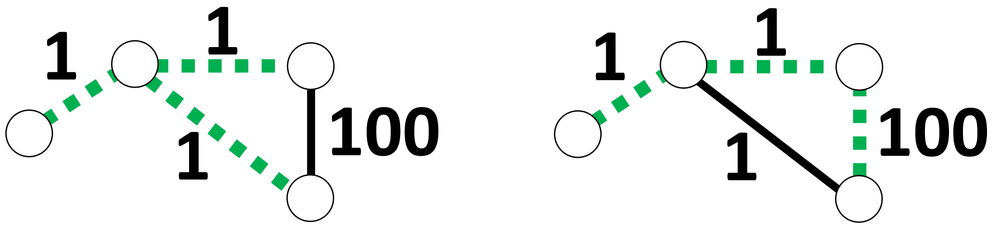
Tree – connected graph with no loops.



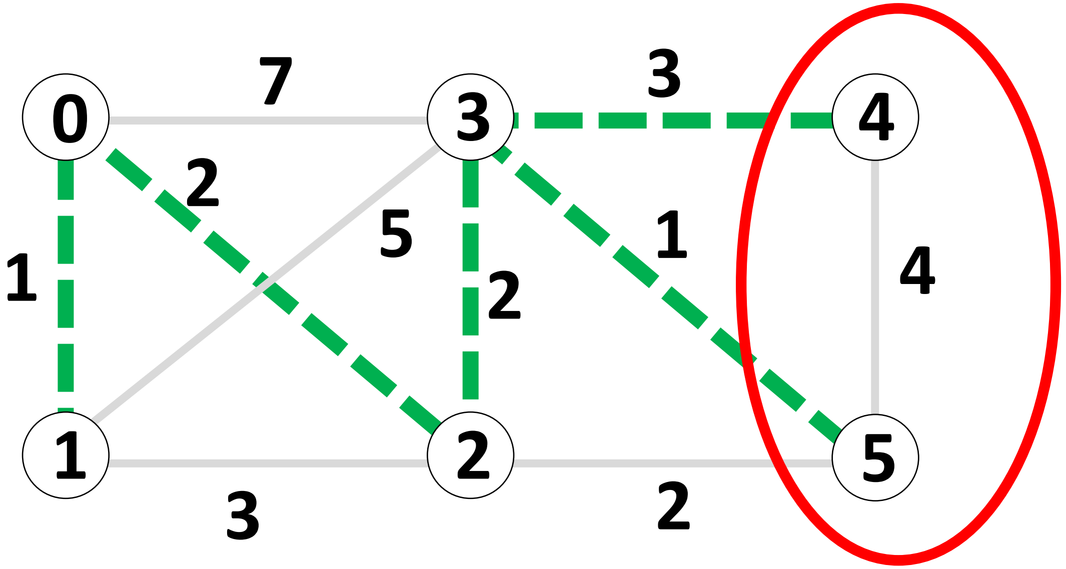
Spanning tree – tree that includes all vertices in a graph.



Minimum spanning tree – spanning tree whose sum of edge costs is the minimum possible value.



Minimum Spanning Tree

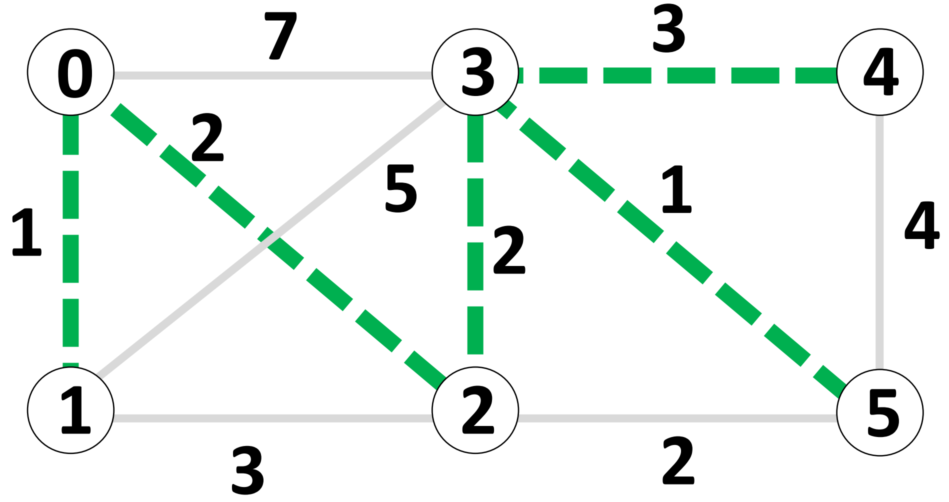


Does it ever make sense to have a cycle?

MST Goal: Connect all vertices to each other with a minimum weight subset of edges.

Must be a tree!

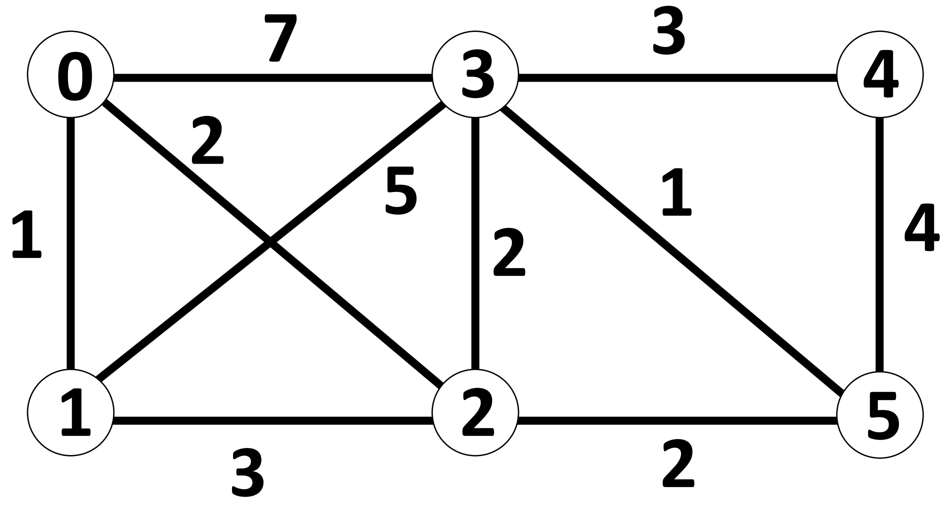
Minimum Spanning Tree



How to find MSTs?

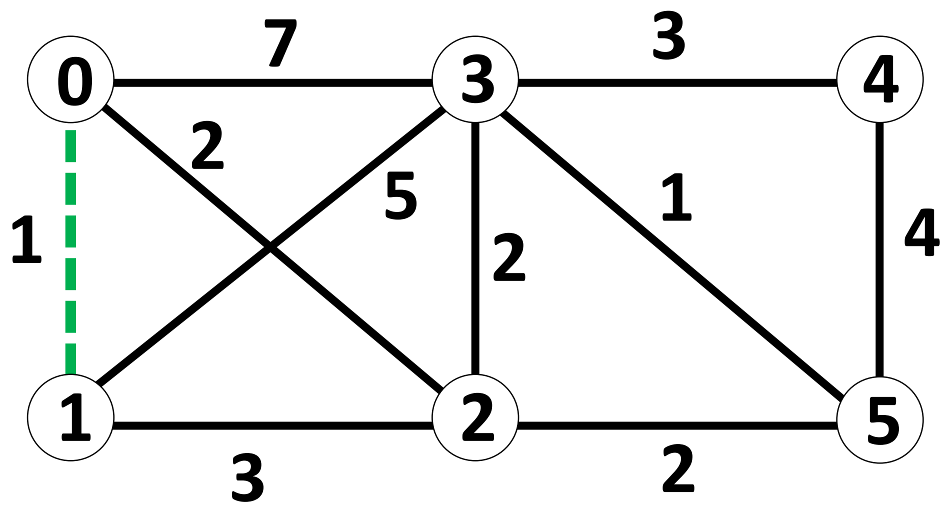
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



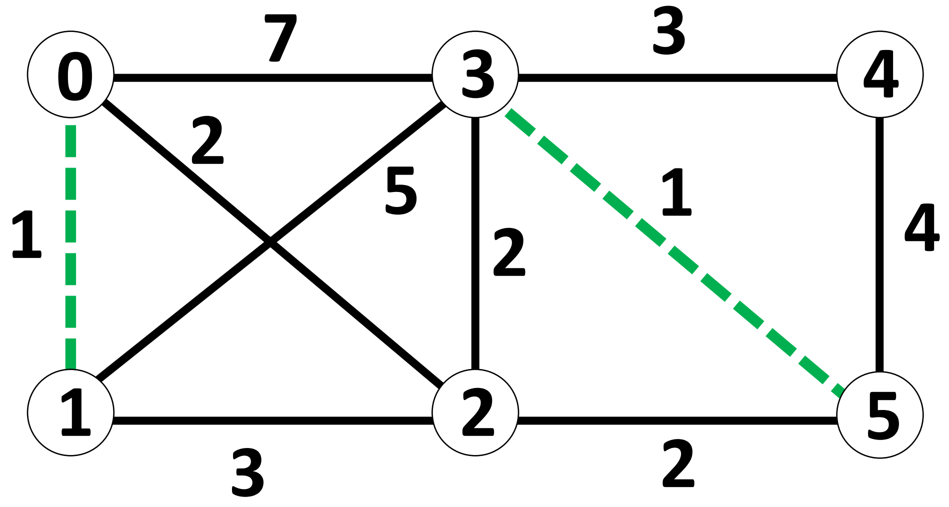
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



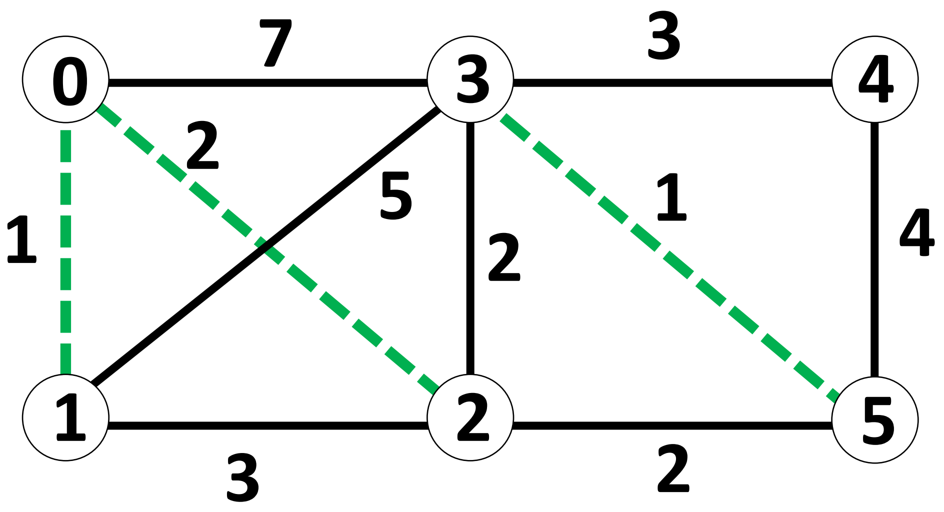
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



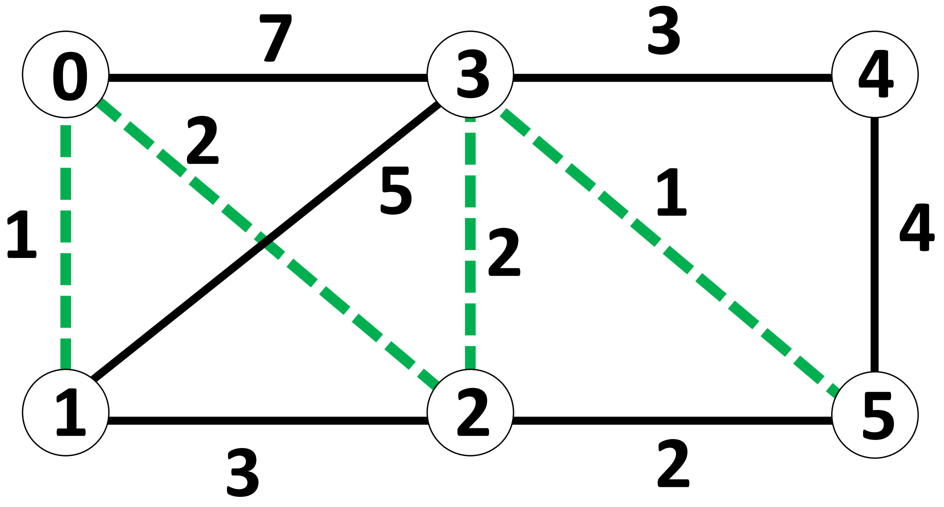
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



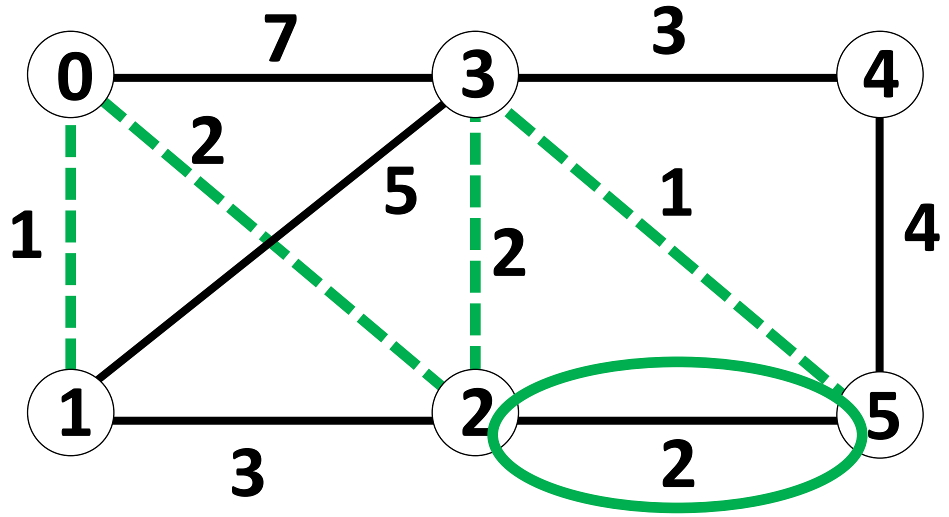
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



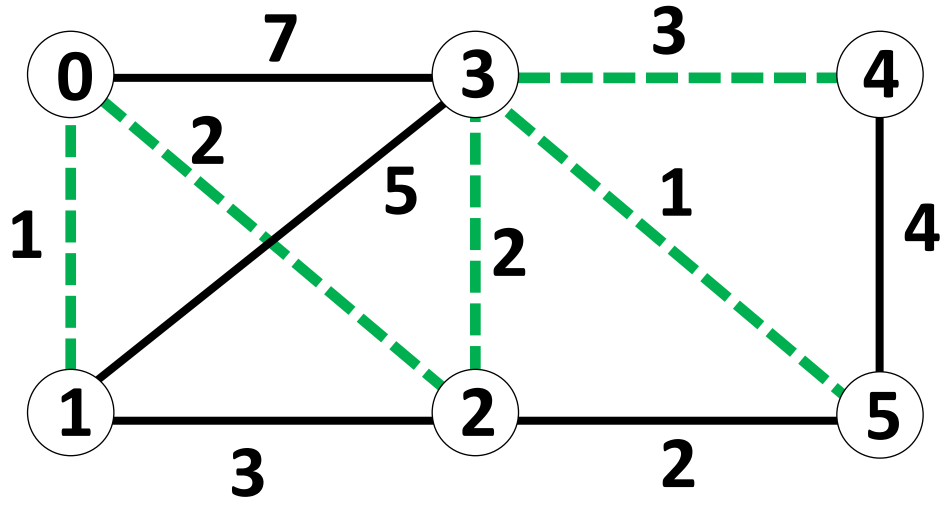
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



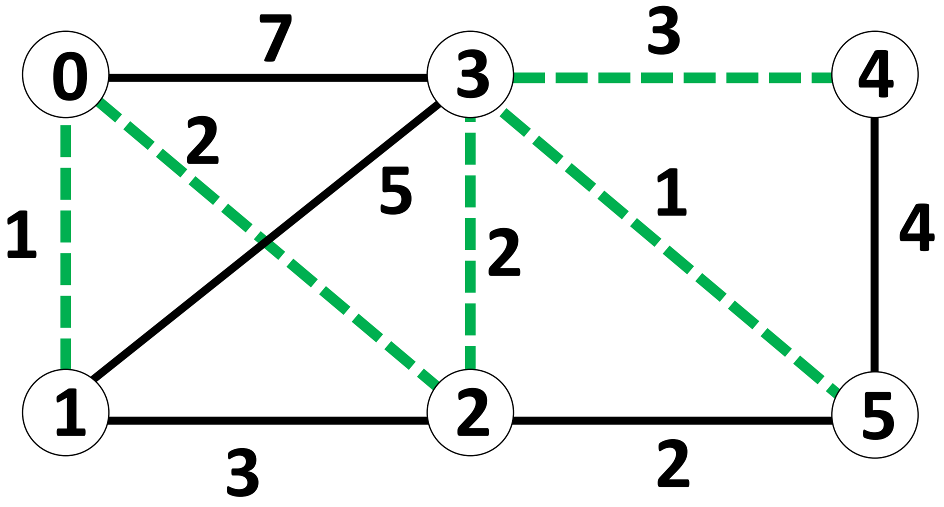
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



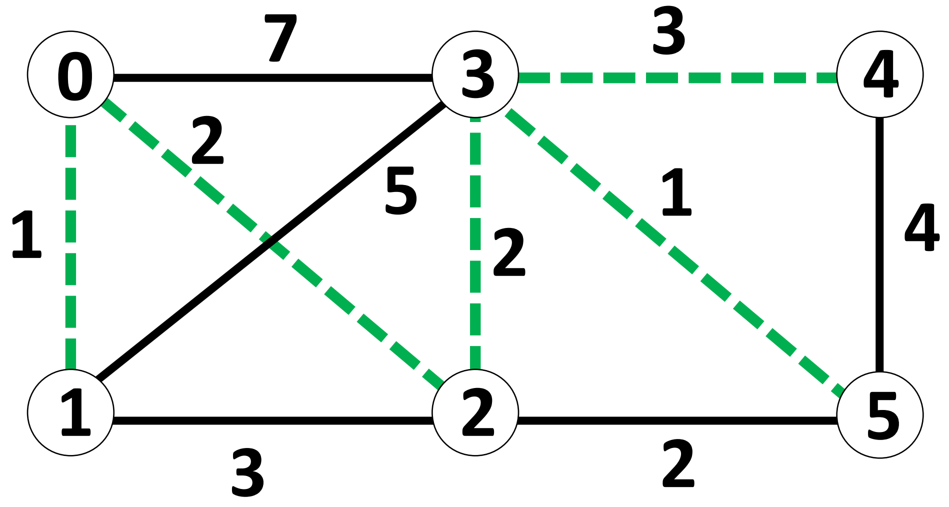
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



Kruskal’s MST Algorithm

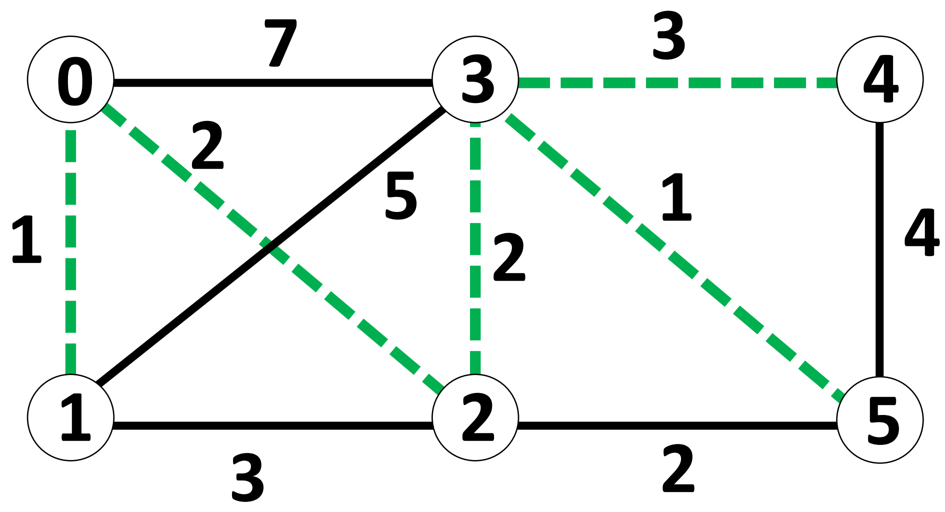
At each iteration, add the edge with smallest weight, that does not create a cycle.



Three questions for any algorithm:

Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.

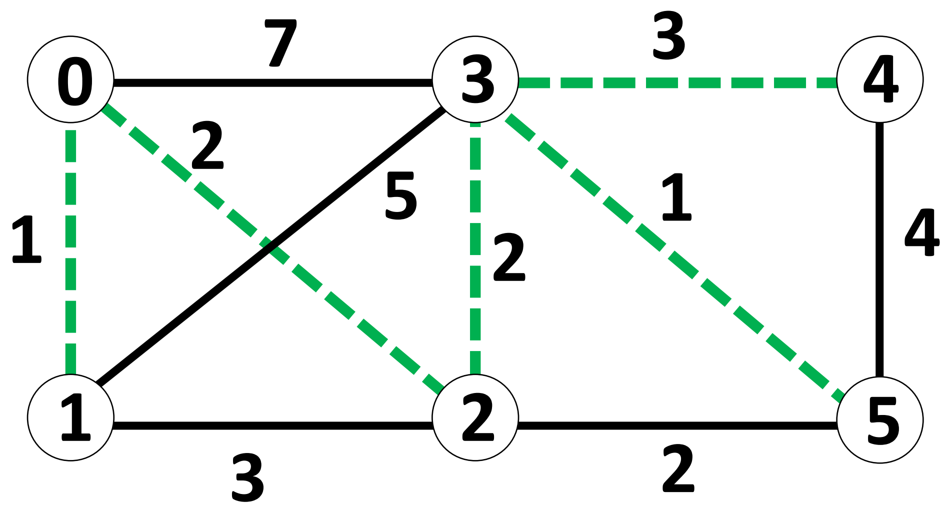


Three questions for any algorithm:

1. **Running time?**
2. **Validity/correctness?**
3. **Performance/accuracy?**

Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



Three questi for any al m:

1. Run e?

2 y/correctness?

rformance/accuracy?

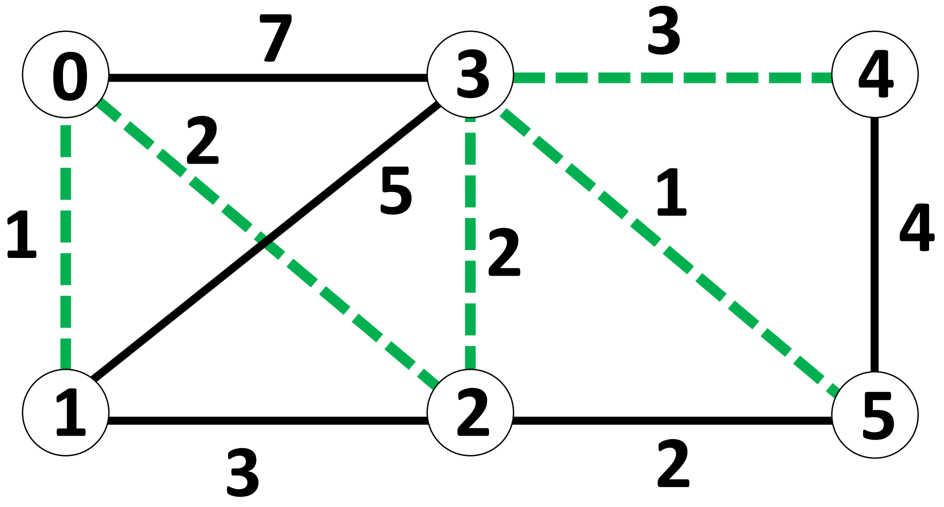
How to implement this?

Minimum Spanning Trees

CSCI 232

Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



```
public MinimumSpanningTree(EdgeWeightedGraph graph) {

}
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




















}
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    // Get the set of edges.















}
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    // Get the set of edges.
    HashSet<Edge> Edges = graph.getEdges();












}
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    // Get the set of edges, in order of increasing weight.
    ~~HashSet<Edge> Edges = graph.getEdges();~~













}
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    // Get the set of edges, in order of increasing weight.
    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();













}
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    // Get the set of edges, in order of increasing weight.
    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

}
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();




    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    // Run Kruskal’s algorithm.




}
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();





    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {








}   }
```

```java
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();






    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();









}   }
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();





    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        mst.add(edge);







}   }
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();






    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        ~~mst.add(edge);~~





}   }
```

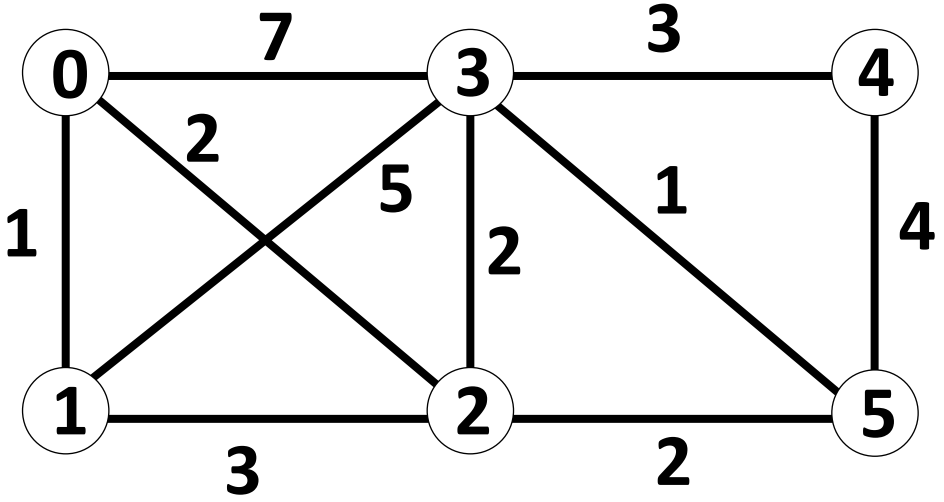
Need to check if adding edge adds a loop!

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();





    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        ~~mst.add(edge);~~




}   }
```

Need to check if adding edge adds a loop!

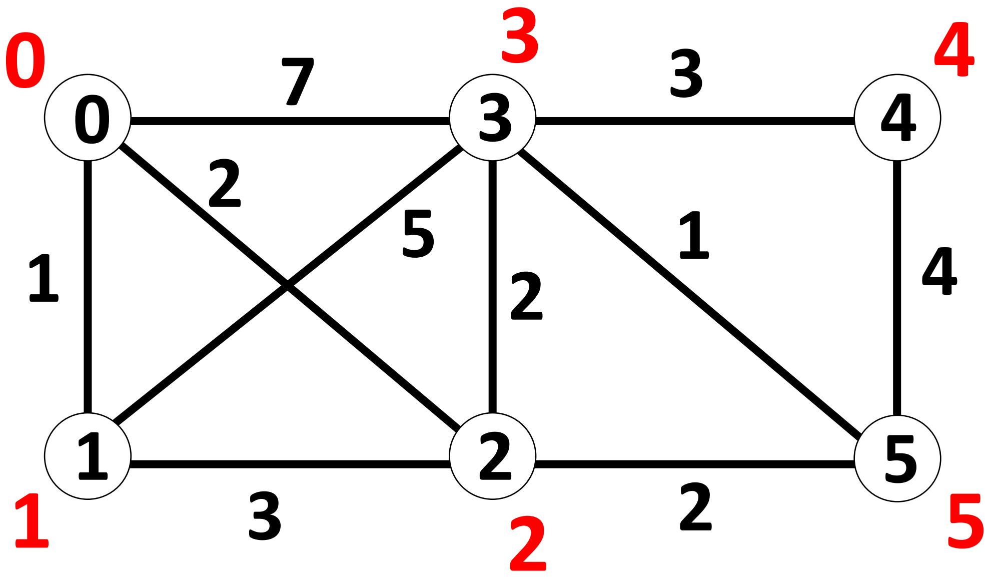
How?

Cycle Finding



How can we determine if adding an edge puts in a cycle?

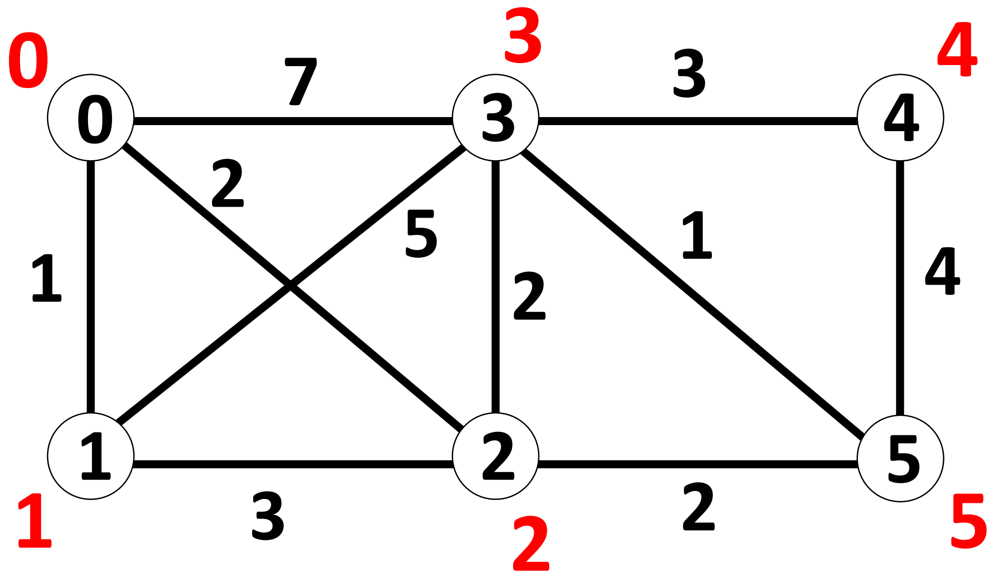
Cycle Finding



Rules: Only add edge if vertices have different connected component markers.

Connected component (in the tree) marker.

Cycle Finding

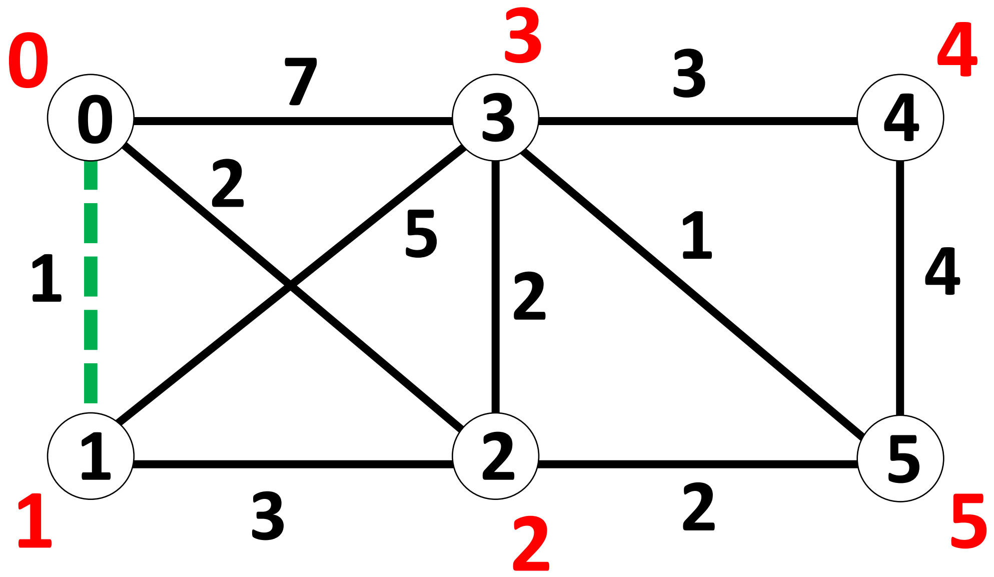


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

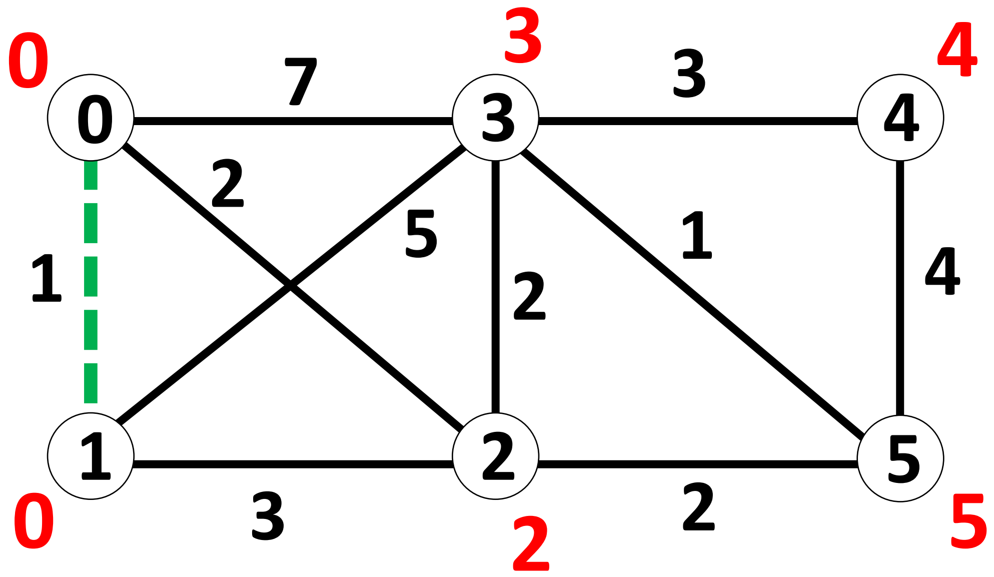


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

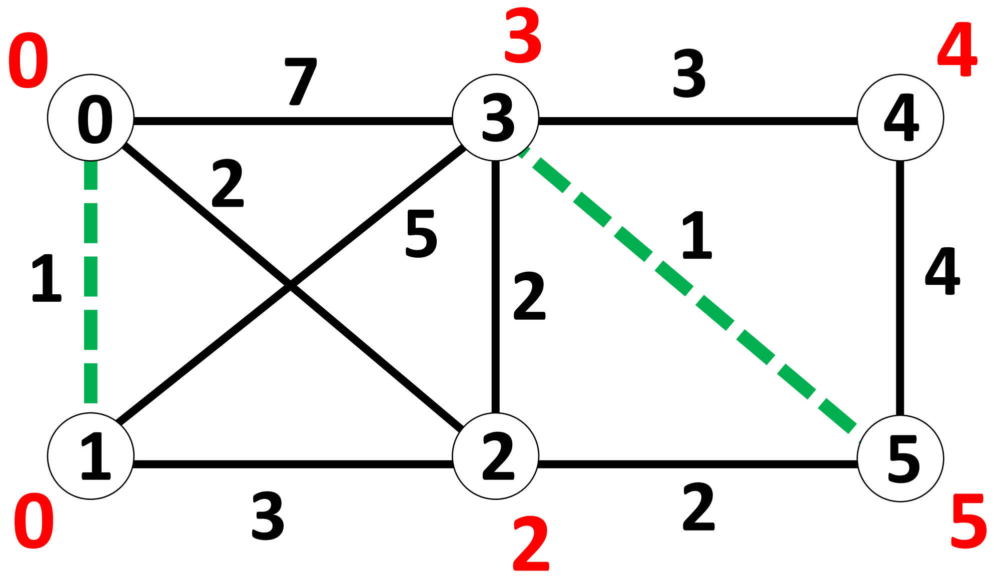


Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Connected component (in the tree) marker.

Cycle Finding

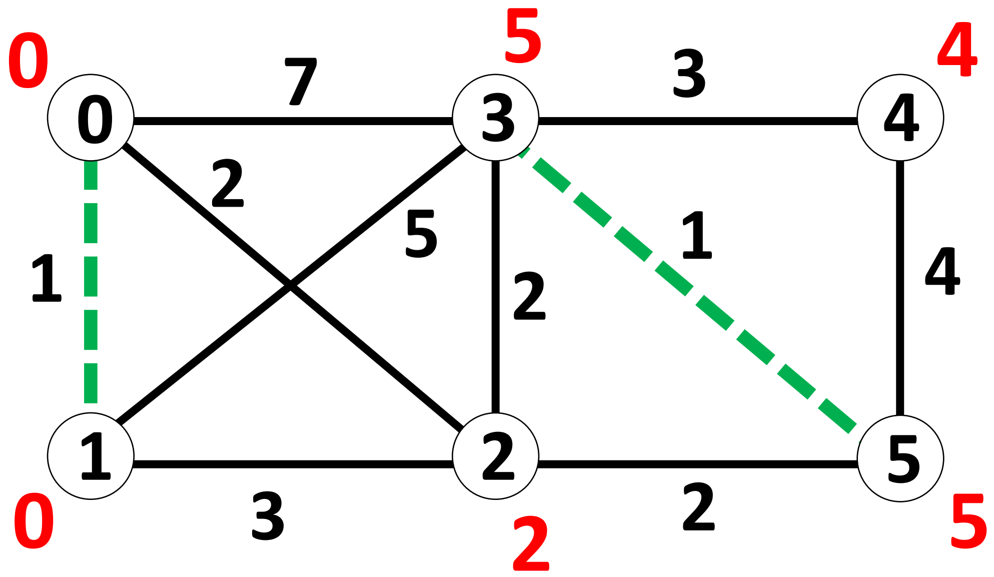


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **<u>all</u>** vertices with the other marker.

Cycle Finding

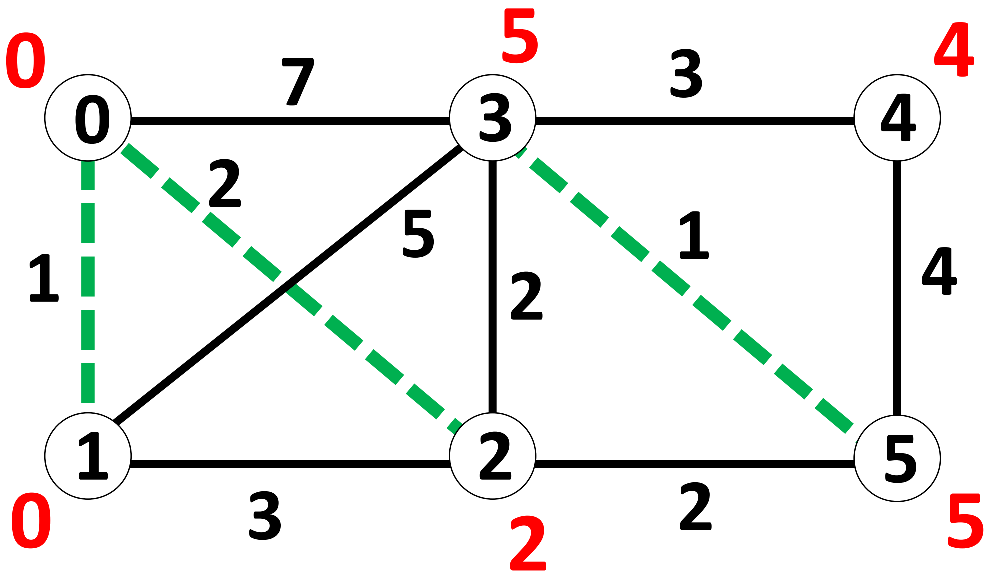


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Cycle Finding

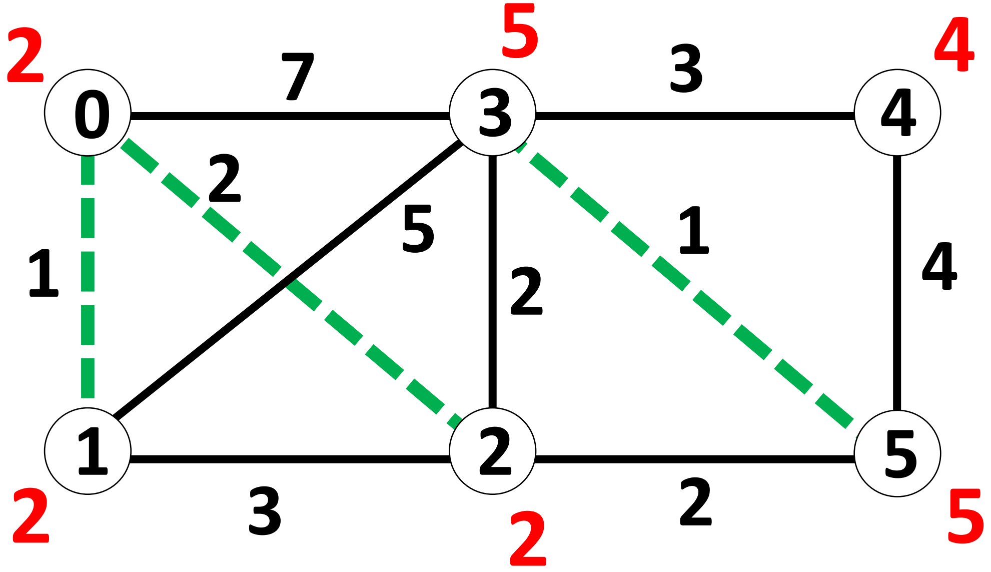


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **<u>all</u>** vertices with the other marker.

Cycle Finding

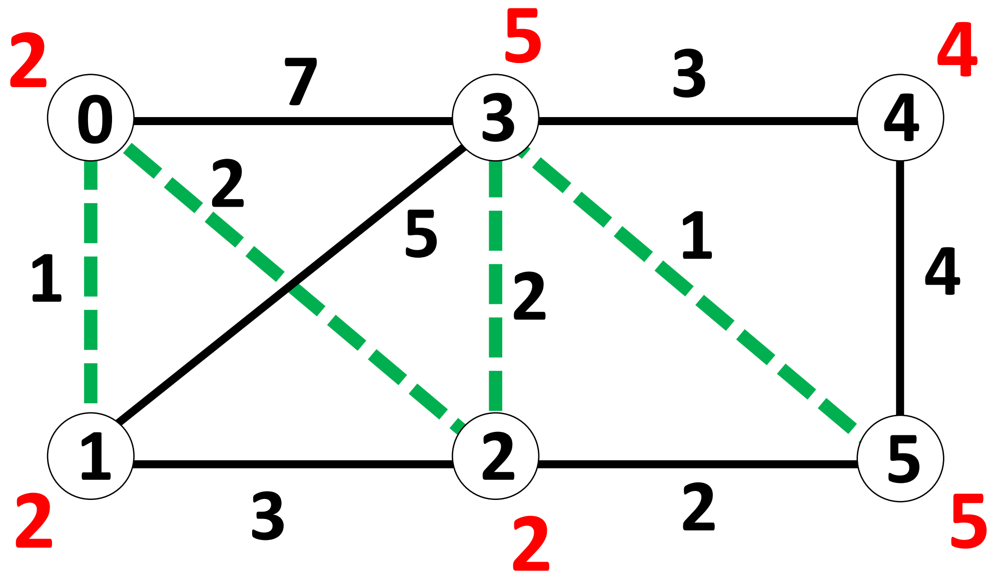


Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **<u>all</u>** vertices with the other marker.

Cycle Finding

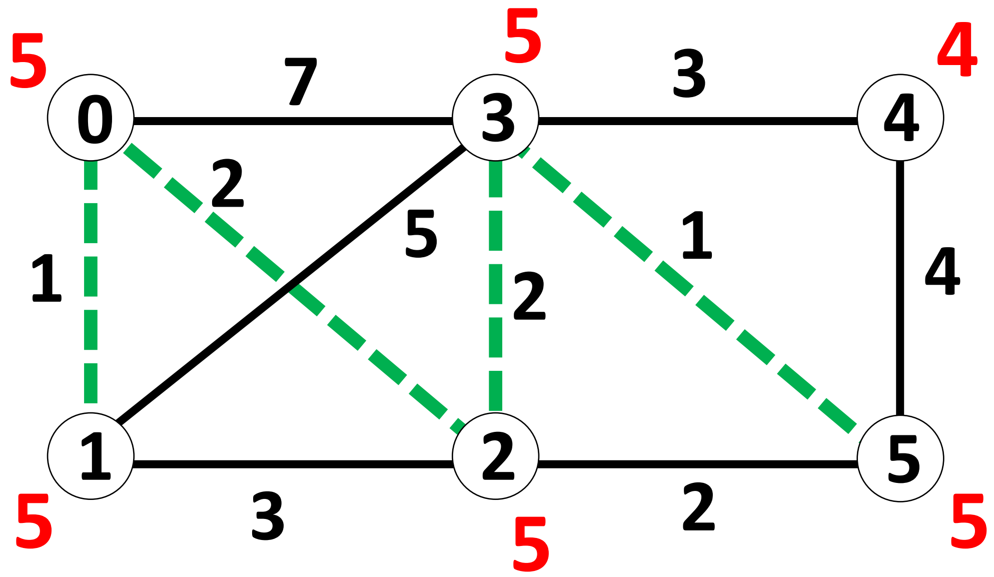


Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **<u>all</u>** vertices with the other marker.

Connected component (in the tree) marker.

Cycle Finding

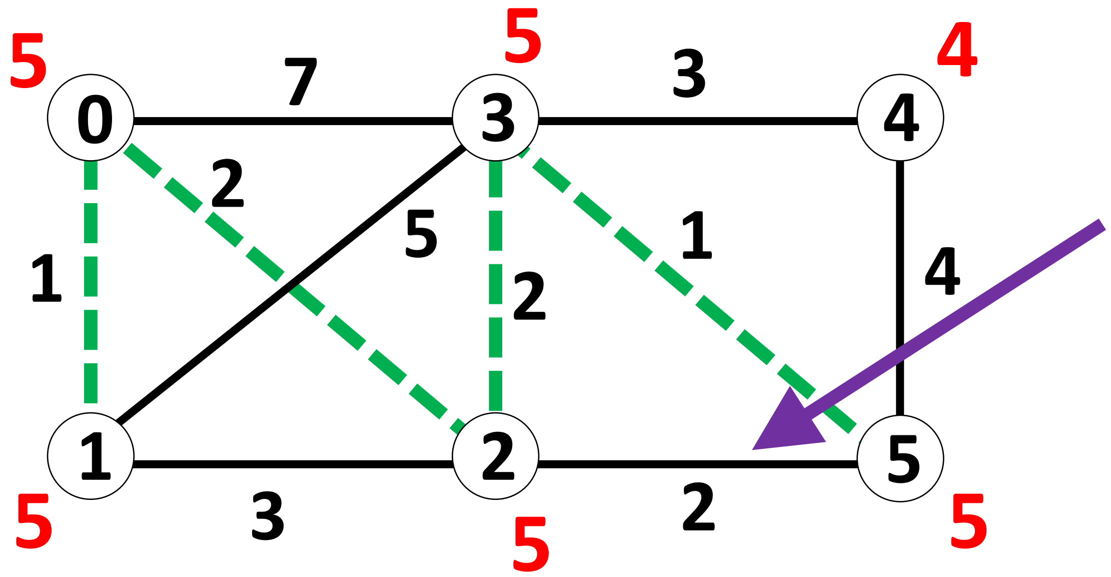


Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **<u>all</u>** vertices with the other marker.

Connected component (in the tree) marker.

Cycle Finding



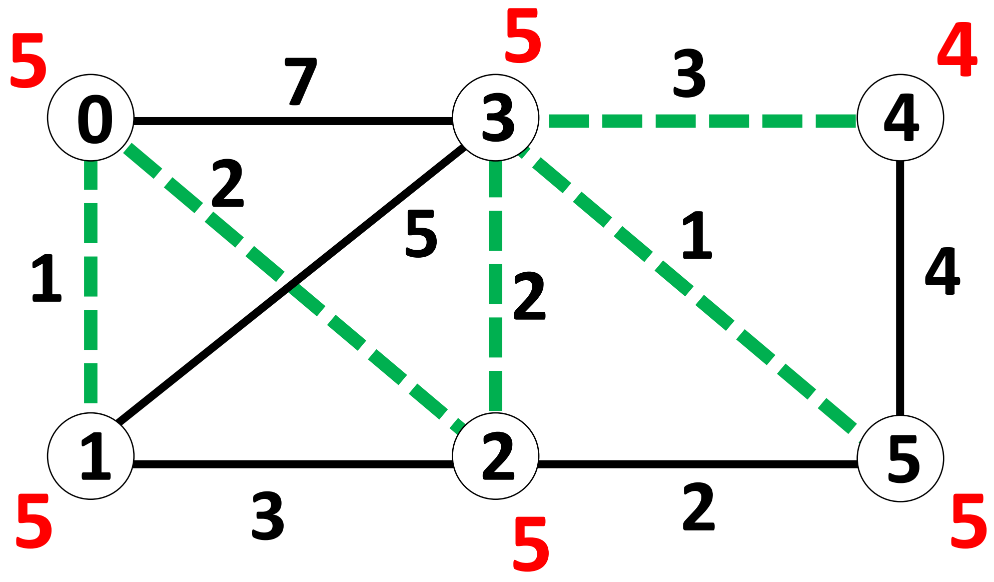
Why is adding this edge a bad idea?

Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **<u>all</u>** vertices with the other marker.

Cycle Finding

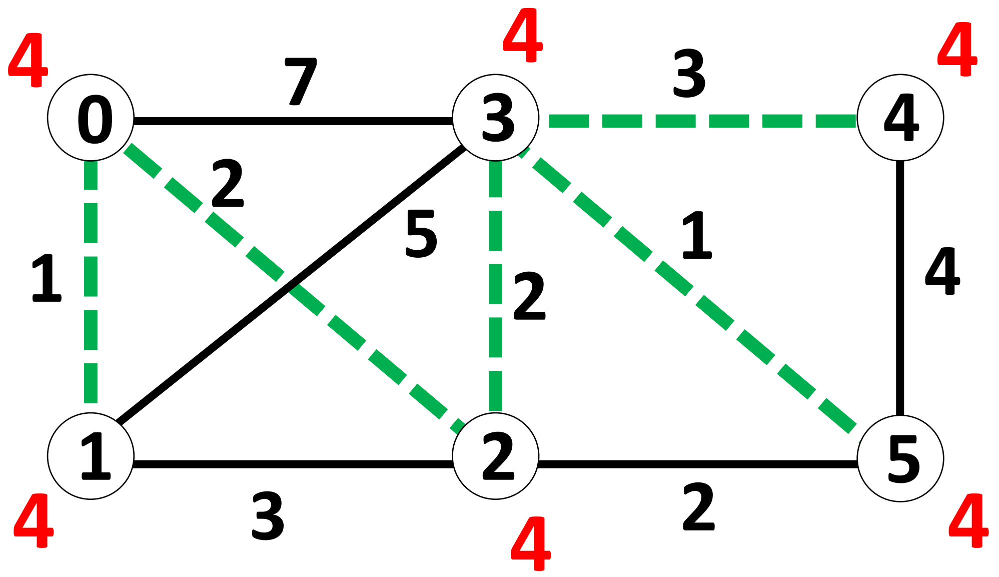


Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Connected component (in the tree) marker.

Cycle Finding



Connected component (in the tree) marker.

Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        ~~mst.add(edge);~~

}   }
```

Need to check if adding edge adds a loop!

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();





    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();

        // Need to check if adding edge adds a loop.






}   }
```

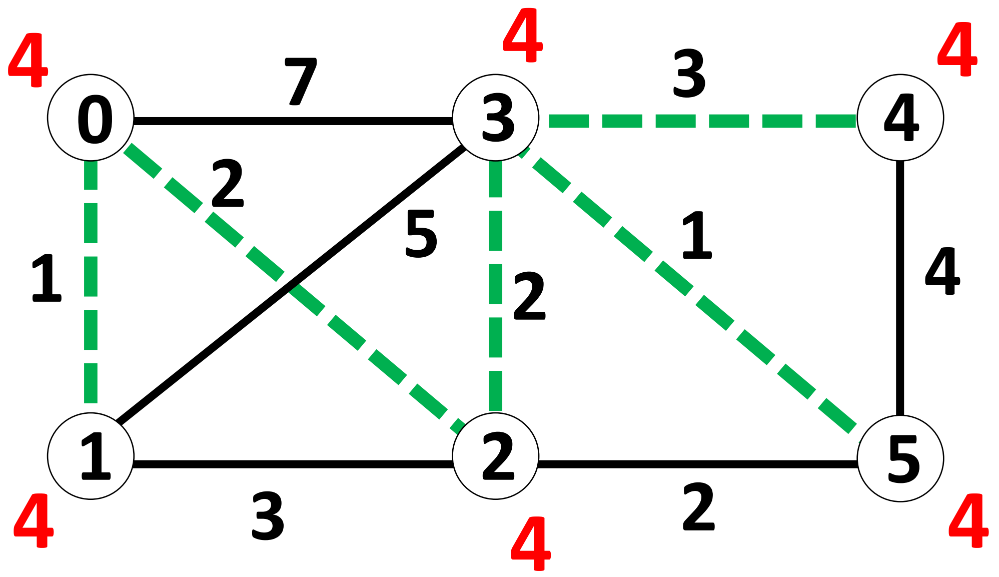
```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];




    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();

        // Need to check if adding edge adds a loop.






}   }
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();

        // Need to check if adding edge adds a loop.





}   }
```

```java
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                   != connectedComponentMarker[edge.getVertices()[1]]) {





}   }   }
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                   != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);






}   }   }
```

Cycle Finding



Rules: Only add edge if vertices have different connected component markers.

To add edge, pick one vertex's marker and change **all** vertices with the other marker.

Connected component (in the tree) marker.

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();
    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }
    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }
    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                   != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
}   }   }
```

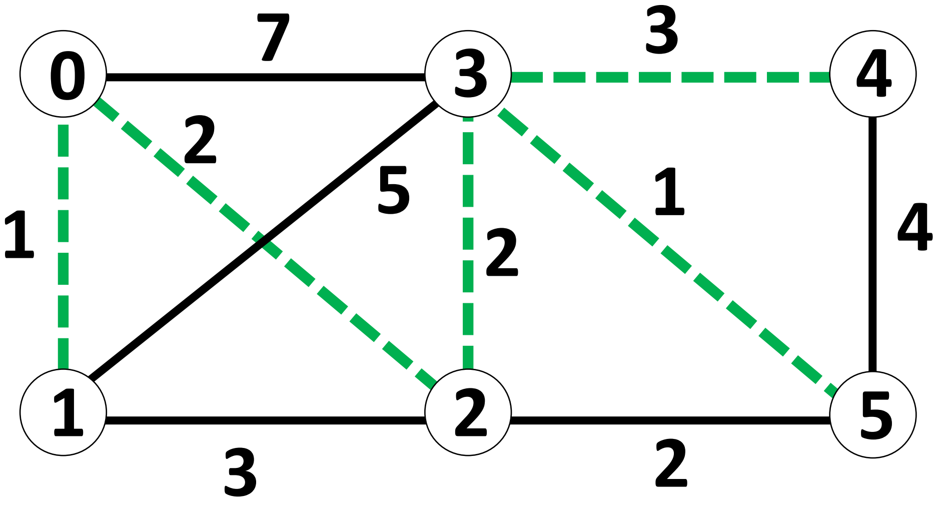
```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                   != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];



}   }   }
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                   != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {



}   }   }   }
```

```
public MinimumSpanningTree(EdgeWeightedGraph graph) {
    HashSet<Edge> mst = new HashSet<>();

    int[] connectedComponentMarker = new int[graph.getNumVertices()];
    for (int i = 0; i < connectedComponentMarker.length; i++) {
        connectedComponentMarker[i] = i;
    }

    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();
    for (Edge edge : graph.getEdges()) {
        edgeQueue.add(edge);
    }

    while (!edgeQueue.isEmpty()) {
        Edge edge = edgeQueue.poll();
        if (connectedComponentMarker[edge.getVertices()[0]]
                   != connectedComponentMarker[edge.getVertices()[1]]) {
            mst.add(edge);
            int newMarker = connectedComponentMarker[edge.getVertices()[0]];
            int oldMarker = connectedComponentMarker[edge.getVertices()[1]];
            for (int i = 0; i < connectedComponentMarker.length; i++) {
                if (connectedComponentMarker[i] == oldMarker) {
                    connectedComponentMarker[i] = newMarker;
}   }   }   }   }
```

Kruskal’s MST Algorithm

At each iteration, add smallest weight edge that doesn’t create a cycle.



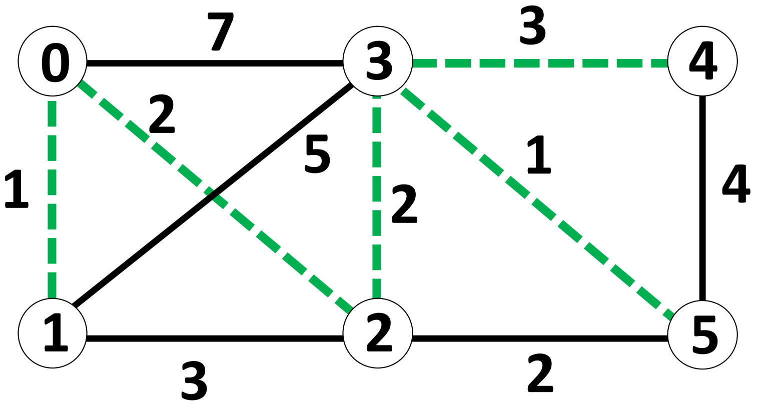
Is the solution valid (i.e., is the output a spanning tree)?

Is the solution optimal (i.e., is the output a <u>minimum</u> spanning tree)?

Kruskal’s MST Algorithm

At each iteration, add smallest weight edge that doesn’t create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal’s algorithm.

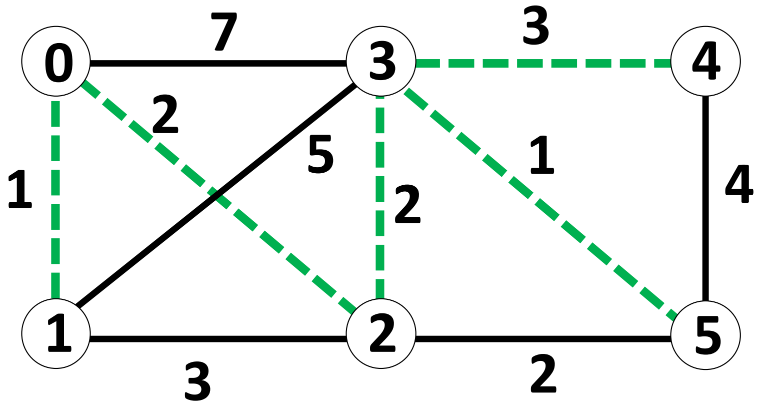


Kruskal’s MST Algorithm

At each iteration, add smallest weight edge that doesn’t create a cycle.

<u>Proof of validity:</u> Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal’s algorithm.

T is a tree because…?

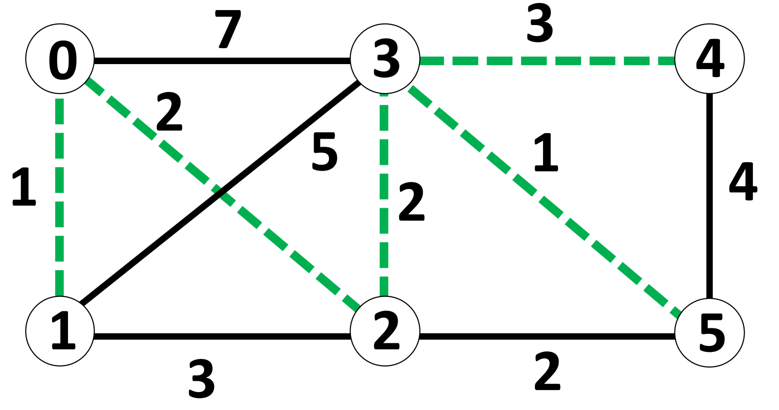


Kruskal’s MST Algorithm

At each iteration, add smallest weight edge that doesn’t create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal’s algorithm.

T is a tree because it is connected (otherwise we could have added more edges without creating cycles) and there are no cycles.



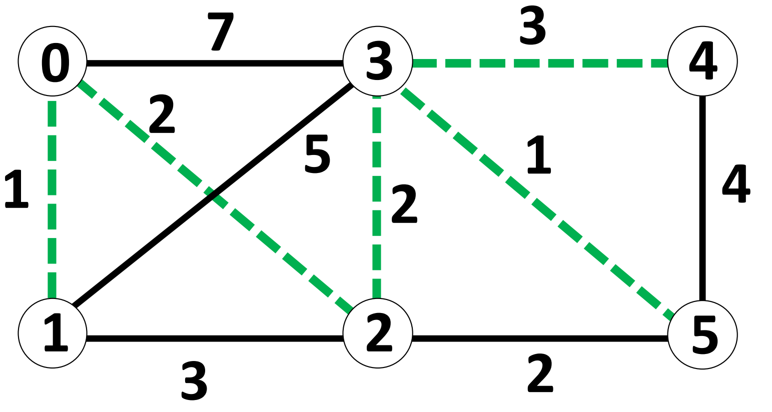
Kruskal’s MST Algorithm

At each iteration, add smallest weight edge that doesn’t create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal’s algorithm.

T is a tree because it is connected (otherwise we could have added more edges without creating cycles) and there are no cycles.

T spans G because…?



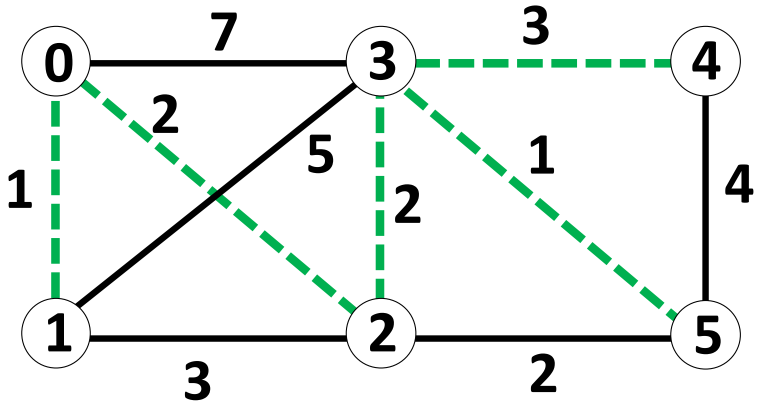
Kruskal’s MST Algorithm

At each iteration, add smallest weight edge that doesn’t create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal’s algorithm.

T is a tree because it is connected (otherwise we could have added more edges without creating cycles) and there are no cycles.

T spans G because if it did not, we could have added more edges to connected unreached nodes without creating cycles.



Kruskal’s MST Algorithm

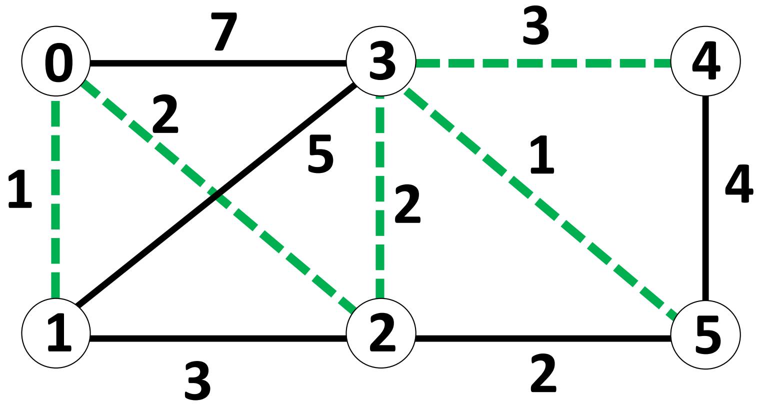
At each iteration, add smallest weight edge that doesn’t create a cycle.

Proof of validity: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal’s algorithm.

T is a tree because it is connected (otherwise we could have added more edges without creating cycles) and there are no cycles.

T spans G because if it did not, we could have added more edges to connected unreached nodes without creating cycles.

$\therefore T$ is a spanning tree of G



Kruskal’s MST Algorithm

At each iteration, add smallest weight edge that doesn’t create a cycle.

Proof of optimality: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal’s algorithm.

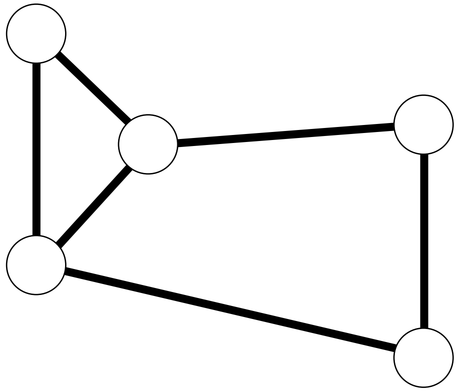
T is an MST because???

MST Cut Property

Assume unique edge costs.

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V \backslash S$ is part of every MST.

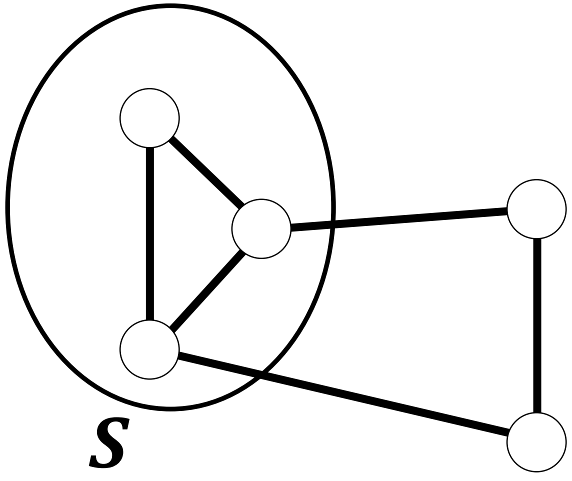
Proof:



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

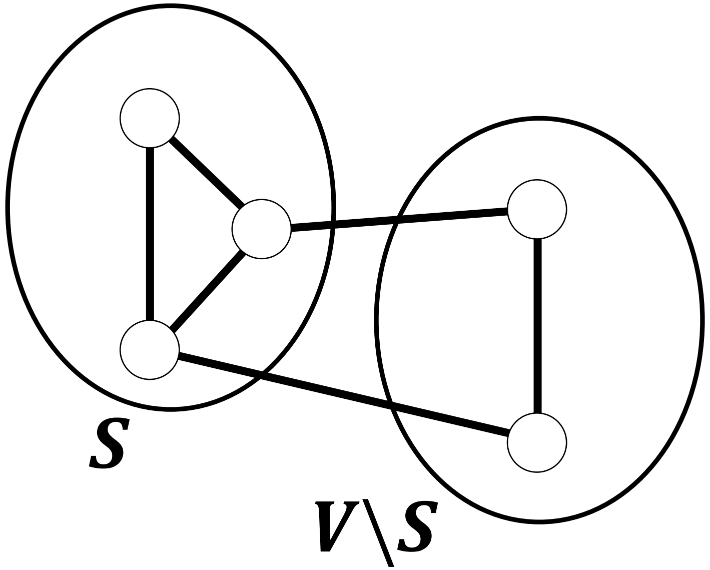
Proof:



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\setminus S$ is part of every MST.

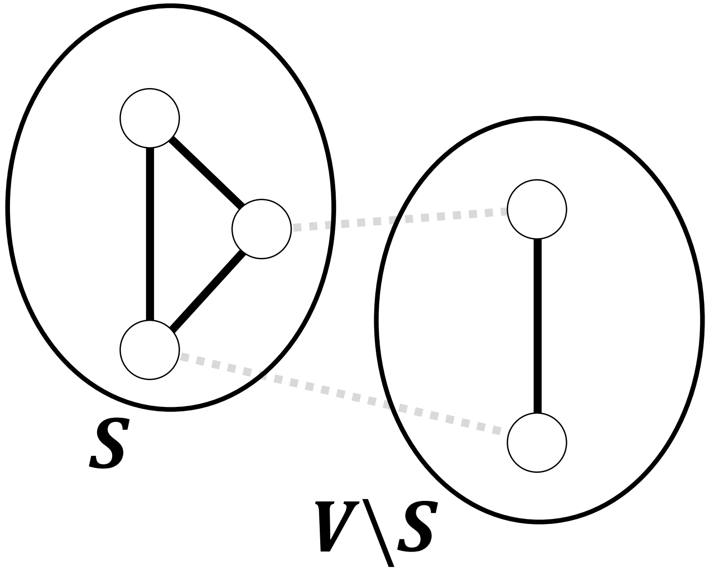
Proof:



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

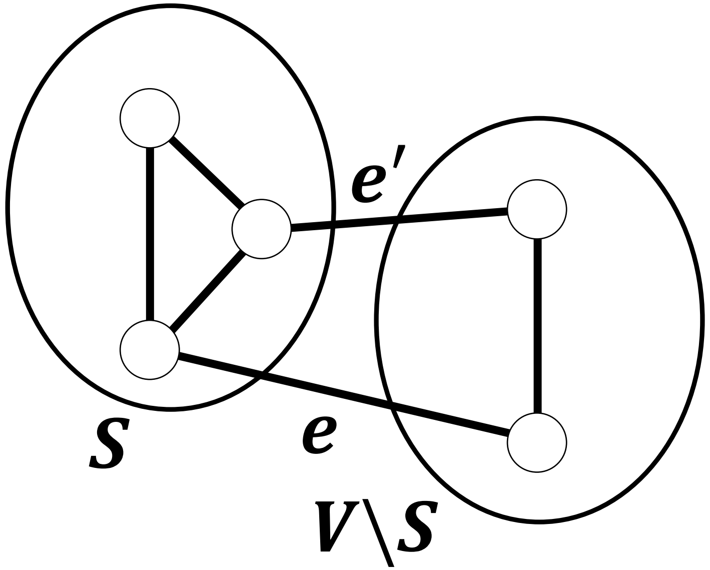


MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V \backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V \backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V \backslash S$.



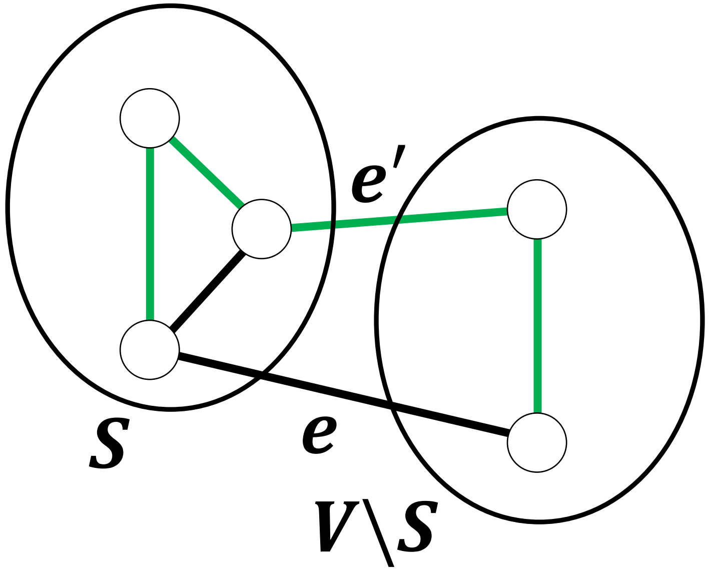
MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e.



MST Cut Property

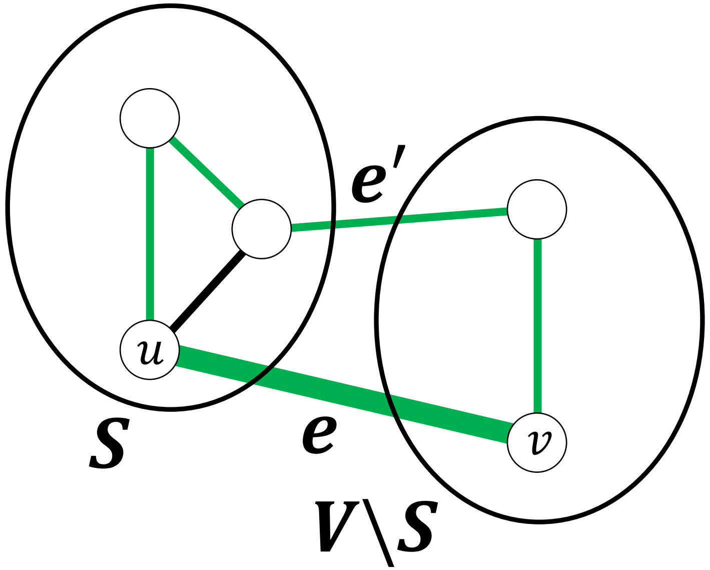
Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e. Then:

1. $T \cup \{e\}$ must have a cycle.
2. That cycle must have another edge e' between S and $V\backslash S$.



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

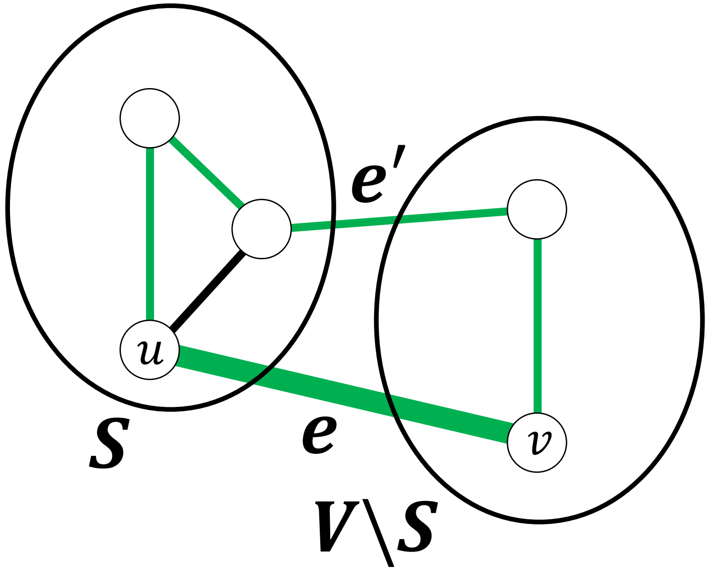
Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e. Then:

1. $T \cup \{e\}$ must have a cycle. Because?

2. That cycle must have another edge e' between S and $V\backslash S$.



MST Cut Property

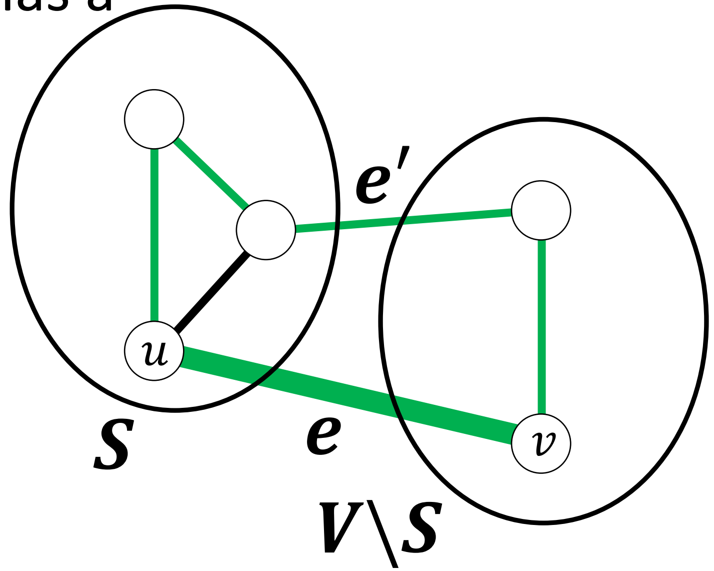
Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e. Then:

1. $T \cup \{e\}$ must have a cycle. (Since spanning tree T already has a path between u and v, so adding e will create a cycle)
2. That cycle must have another edge e' between S and $V\backslash S$. Because?



MST Cut Property

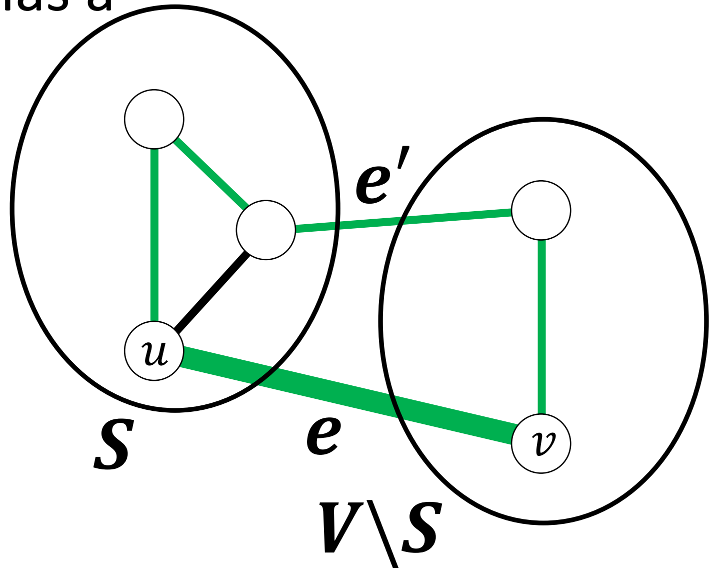
Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e. Then:

1. $T \cup \{e\}$ must have a cycle. (Since spanning tree T already has a path between u and v, so adding e will create a cycle)
2. That cycle must have another edge e' between S and $V\backslash S$. (Since there must be a path from $u \in S$ to $v \in V\backslash S$ in T)



MST Cut Property

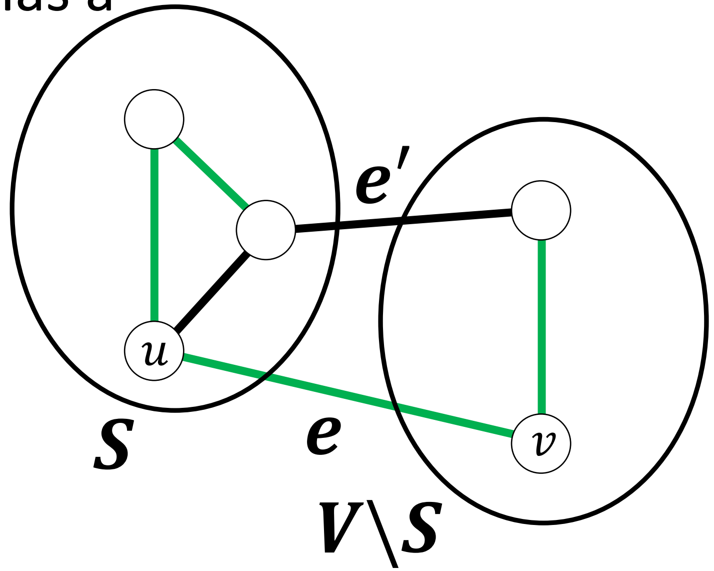
Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V \backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V \backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V \backslash S$.

Suppose T is a ST that does not include e. Then:

1. $T \cup \{e\}$ must have a cycle. (Since spanning tree T already has a path between u and v, so adding e will create a cycle)
2. That cycle must have another edge e' between S and $V \backslash S$. (Since there must be a path from $u \in S$ to $v \in V \backslash S$ in T)



Thus, removing e' and including e results in a cheaper spanning tree!

MST Cut Property

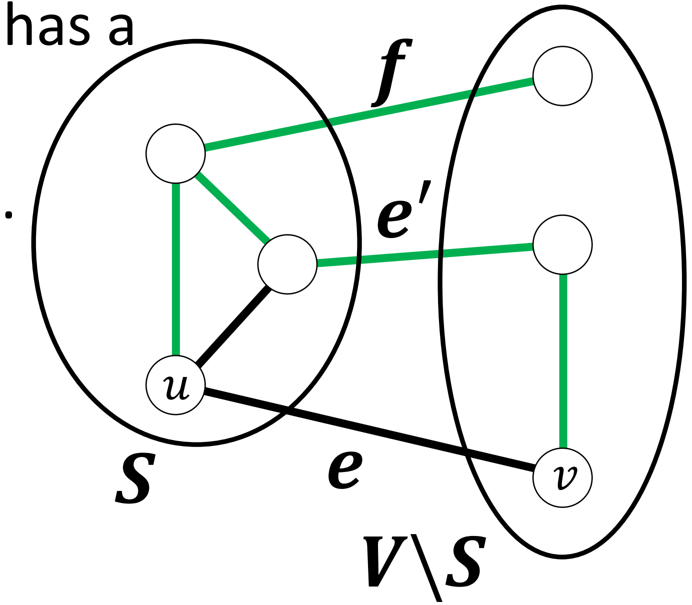
Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e. Then:

1. $T \cup \{e\}$ must have a cycle. (Since spanning tree T already has a path between u and v, so adding e will create a cycle)
2. That cycle must have another edge e' between S and $V\backslash S$. (Since there must be a path from $u \in S$ to $v \in V\backslash S$ in T)



Need to make sure we pick the edge between S and $V\backslash S$ on the cycle!

MST Cut Property

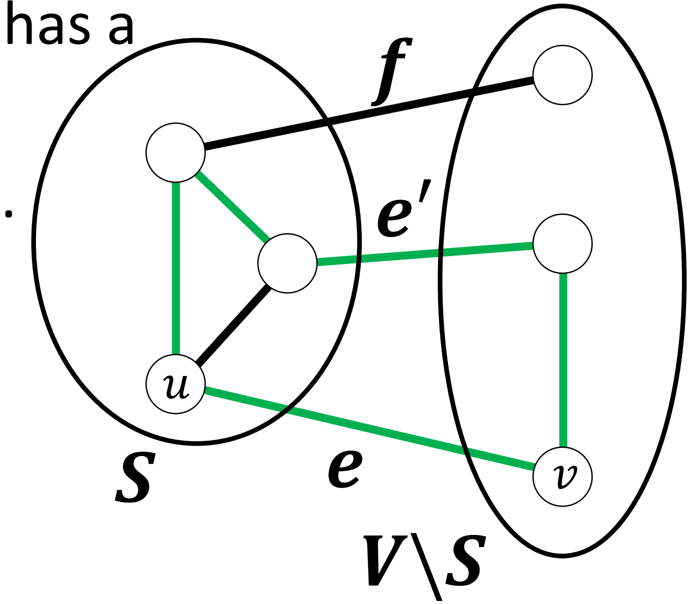
Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V \backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V \backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V \backslash S$.

Suppose T is a ST that does not include e. Then:

1. $T \cup \{e\}$ must have a cycle. (Since spanning tree T already has a path between u and v, so adding e will create a cycle)
2. That cycle must have another edge e' between S and $V \backslash S$. (Since there must be a path from $u \in S$ to $v \in V \backslash S$ in T)



Need to make sure we pick the edge between S and $V \backslash S$ on the cycle!

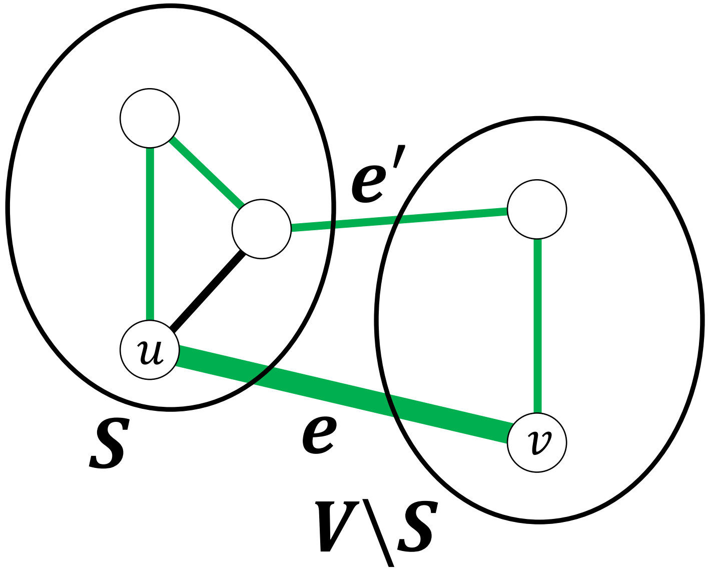
MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e. $T \cup \{e\}$ must have a cycle and that cycle must have another edge e' between S and $V\backslash S$.



MST Cut Property

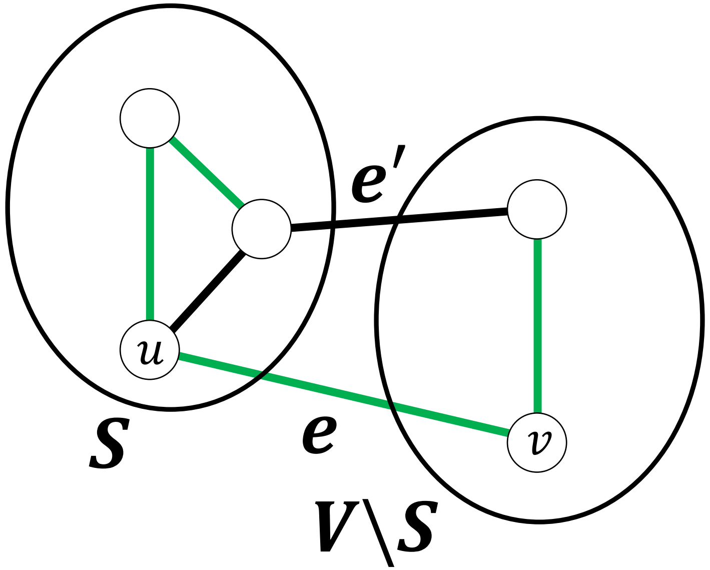
Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e. $T \cup \{e\}$ must have a cycle and that cycle must have another edge e' between S and $V\backslash S$.

Remove e' to form $T' = T \cup \{e\}\backslash\{e'\}$.



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

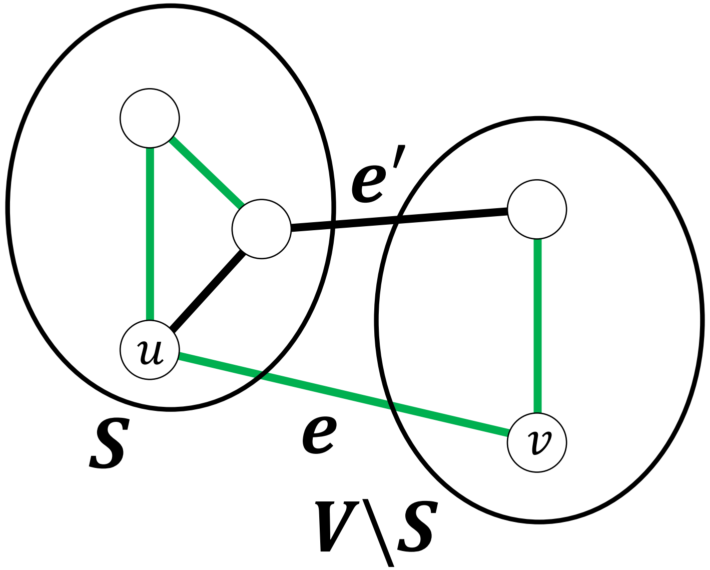
Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e. $T \cup \{e\}$ must have a cycle and that cycle must have another edge e' between S and $V\backslash S$.

Remove e' to form $T' = T \cup \{e\}\backslash\{e'\}$.

T' is a tree (removing edge from cycle cannot disconnect graph)



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V \backslash S$ is part of every MST.

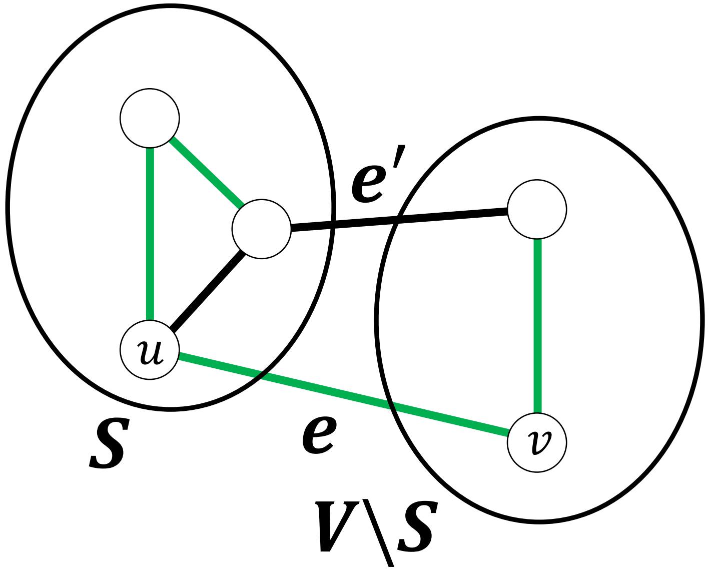
Proof: Any MST of G must include some edge between S and $V \backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V \backslash S$.

Suppose T is a ST that does not include e. $T \cup \{e\}$ must have a cycle and that cycle must have another edge e' between S and $V \backslash S$.

Remove e' to form $T' = T \cup \{e\} \backslash \{e'\}$.

T' is a tree (removing edge from cycle cannot disconnect graph)
T' spans V (same number of edges as ST T)



MST Cut Property

<u>Lemma:</u> Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

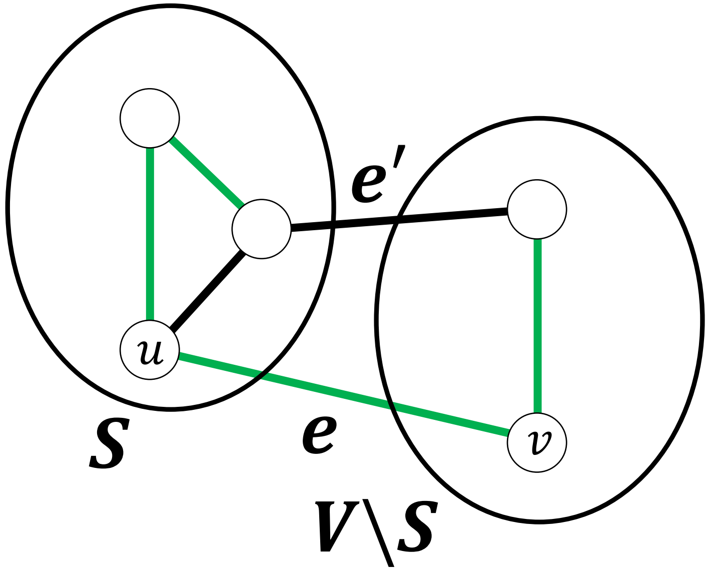
<u>Proof:</u> Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e. $T \cup \{e\}$ must have a cycle and that cycle must have another edge e' between S and $V\backslash S$.

Remove e' to form $T' = T \cup \{e\}\backslash\{e'\}$.

T' is a tree (removing edge from cycle cannot disconnect graph)
T' spans V (same number of edges as ST T)
weight(T') = weight(T) + weight(e) – weight(e').



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V\backslash S$ is part of every MST.

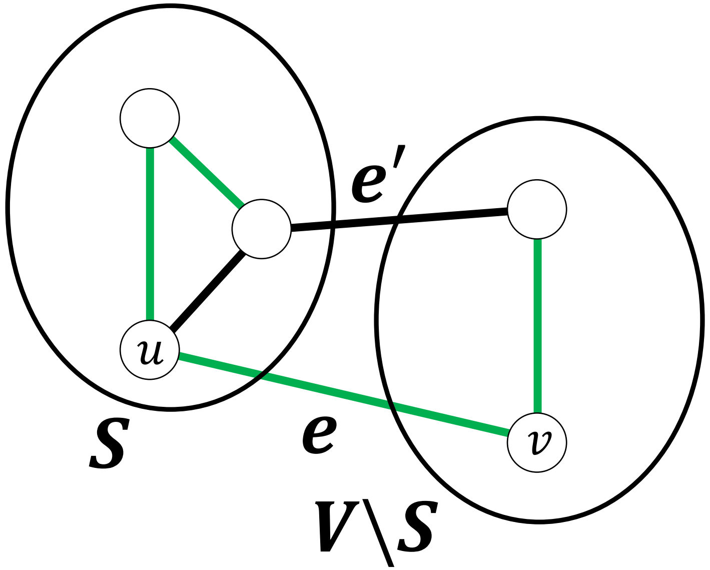
Proof: Any MST of G must include some edge between S and $V\backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V\backslash S$.

Suppose T is a ST that does not include e. $T \cup \{e\}$ must have a cycle and that cycle must have another edge e' between S and $V\backslash S$.

Remove e' to form $T' = T \cup \{e\}\backslash\{e'\}$.

T' is a tree (removing edge from cycle cannot disconnect graph)
T' spans V (same number of edges as ST T)
weight(T') = weight(T) + weight(e) – weight(e').
⇒ weight(T') < weight(T), since weight(e) < weight(e').



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V \backslash S$ is part of every MST.

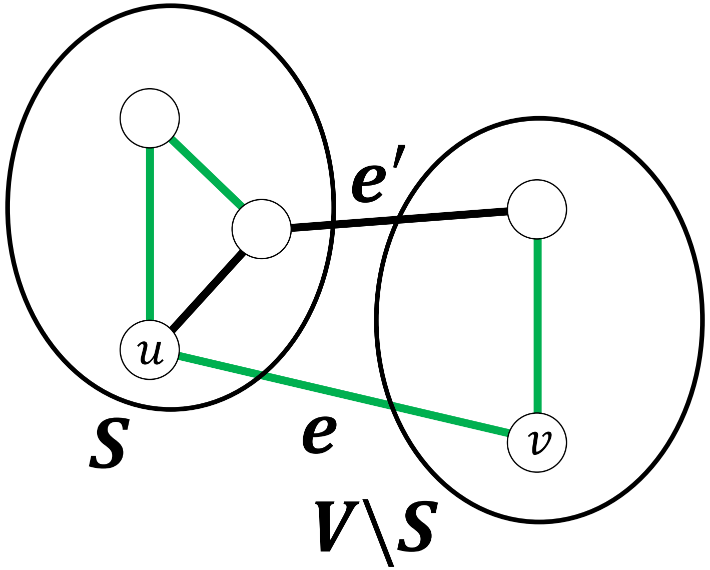
Proof: Any MST of G must include some edge between S and $V \backslash S$ (otherwise it would not be a tree).

Let e be the cheapest edge between S and $V \backslash S$.

Suppose T is a ST that does not include e. $T \cup \{e\}$ must have a cycle and that cycle must have another edge e' between S and $V \backslash S$.

Remove e' to form $T' = T \cup \{e\} \backslash \{e'\}$.

T' is a tree (removing edge from cycle cannot disconnect graph)
T' spans V (same number of edges as ST T)
weight(T') = weight(T) + weight(e) – weight(e').
⇒ weight(T') < weight(T), since weight(e) < weight(e').
⇒ T' is a cheaper ST.



MST Cut Property

Lemma: Suppose that S is a subset of nodes from $G = (V, E)$. Then, the cheapest edge e between S and $V \backslash S$ is part of every MST.

Proof: Any MST of G must include some edge between S and $V \backslash S$ (otherwise it would not be a tree).

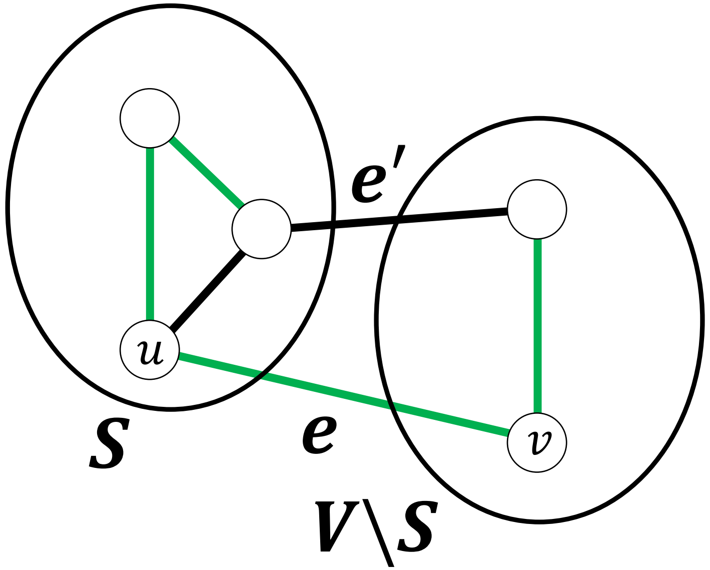
Let e be the cheapest edge between S and $V \backslash S$.

Suppose T is a ST that does not include e. $T \cup \{e\}$ must have a cycle and that cycle must have another edge e' between S and $V \backslash S$.

Remove e' to form $T' = T \cup \{e\} \backslash \{e'\}$.

T' is a tree (removing edge from cycle cannot disconnect graph)
T' spans V (same number of edges as ST T)
weight(T') = weight(T) + weight(e) – weight(e').
  $\Rightarrow$ weight(T') < weight(T), since weight(e) < weight(e').
  $\Rightarrow$ T' is a cheaper ST.

So, e is part of every MST.

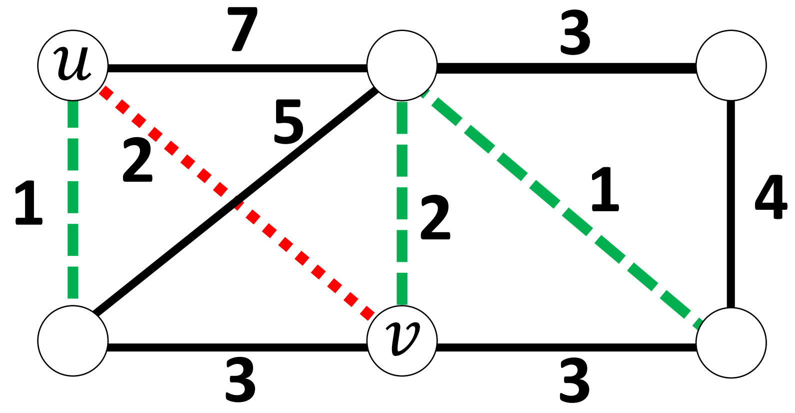


Kruskal’s MST Algorithm

At each iteration, add smallest weight edge that doesn’t create a cycle.

Proof of optimality: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal’s algorithm.

Consider the iteration that edge $e = (u, v)$ is added by Kruskal’s algorithm.



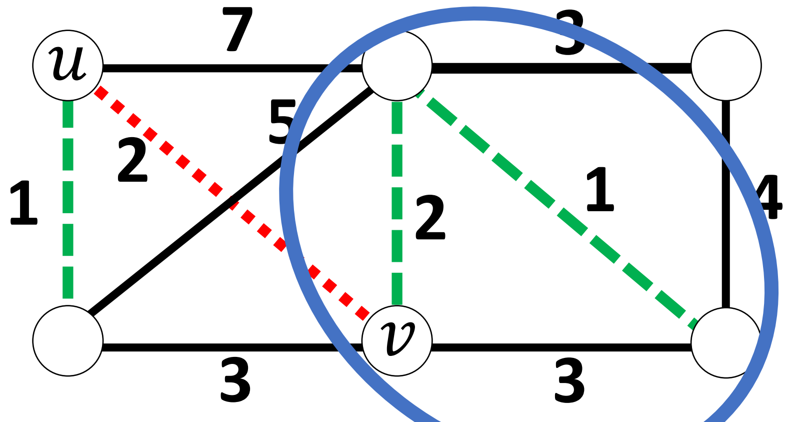
Lemma: The cheapest edge between $S \subseteq V$ and $V \backslash S$ is part of every MST.

Kruskal’s MST Algorithm

At each iteration, add smallest weight edge that doesn’t create a cycle.

Proof of optimality: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal’s algorithm.

Consider the iteration that edge $e = (u, v)$ is added by Kruskal’s algorithm. Let S be the set of v and nodes connected to v. (or u and all nodes connected to u)



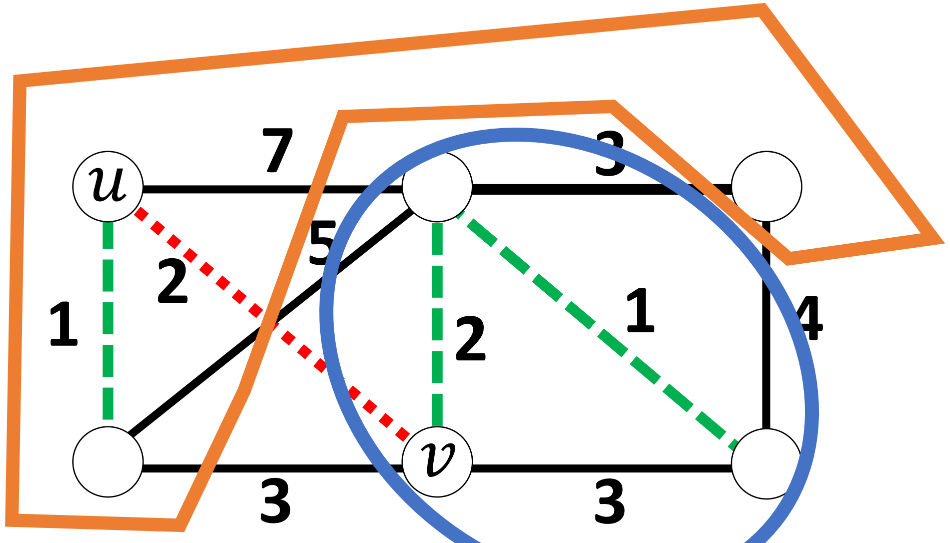
Lemma: The cheapest edge between $S \subseteq V$ and $V \backslash S$ is part of every MST.

Kruskal’s MST Algorithm

At each iteration, add smallest weight edge that doesn’t create a cycle.

Proof of optimality: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal’s algorithm.

Consider the iteration that edge $e = (u, v)$ is added by Kruskal’s algorithm. Let S be the set of v and nodes connected to v. Clearly $v \in S$ and $u \in V \backslash S$ (otherwise adding e would have created a cycle).



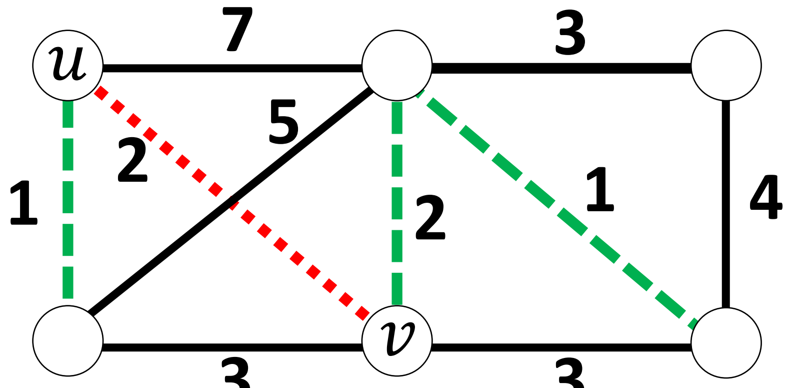
Lemma: The cheapest edge between $S \subseteq V$ and $V \backslash S$ is part of every MST.

Kruskal's MST Algorithm

At each iteration, add smallest weight edge that doesn't create a cycle.

Proof of optimality: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal's algorithm.

Consider the iteration that edge $e = (u, v)$ is added by Kruskal's algorithm. Let S be the set of v and nodes connected to v. Clearly $v \in S$ and $u \in V\backslash S$ (otherwise adding e would have created a cycle). We are picking the cheapest such edge (otherwise the cheaper edge would have been selected since it would not have created a cycle either).



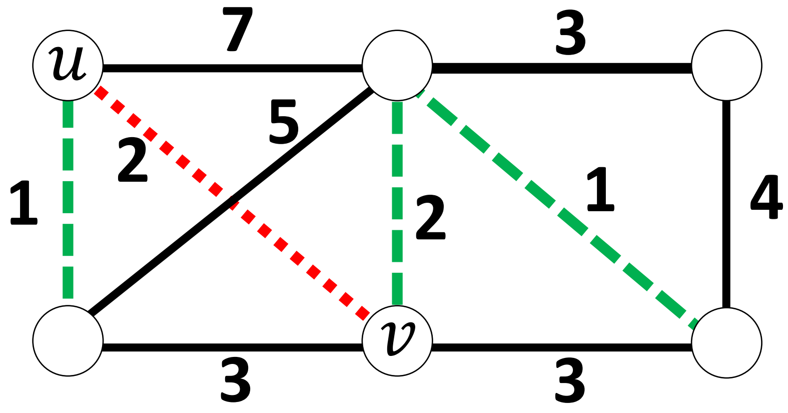
Lemma: The cheapest edge between $S \subseteq V$ and $V\backslash S$ is part of every MST.

Kruskal's MST Algorithm

At each iteration, add smallest weight edge that doesn't create a cycle.

Proof of optimality: Let $G = (V, E)$ be connected, and $T \subseteq E$ be the set of edges resulting from Kruskal's algorithm.

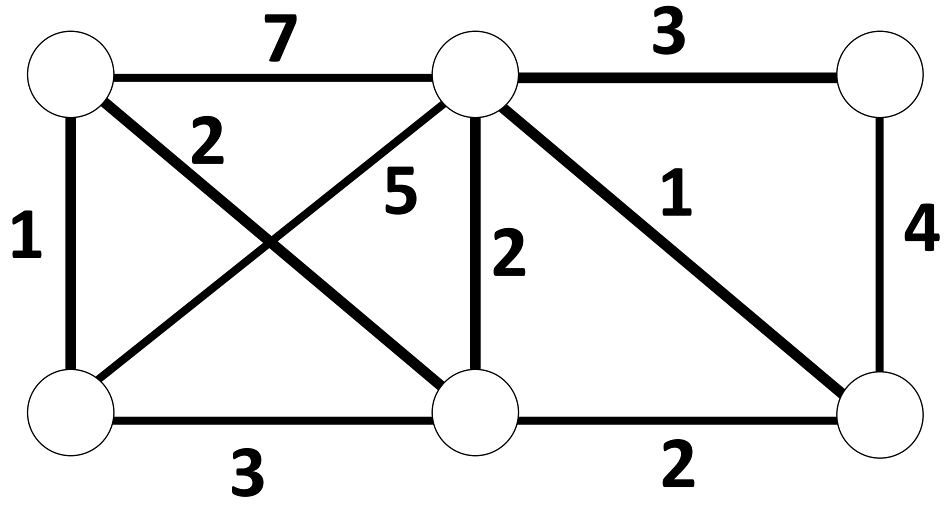
Consider the iteration that edge $e = (u, v)$ is added by Kruskal's algorithm. Let S be the set of v and nodes connected to v. Clearly $v \in S$ and $u \in V \backslash S$ (otherwise adding e would have created a cycle). We are picking the cheapest such edge (otherwise the cheaper edge would have been selected since it would not have created a cycle either). By the cut property lemma, this edge must be part of the MST.



Lemma: The cheapest edge between $S \subseteq V$ and $V \backslash S$ is part of every MST.

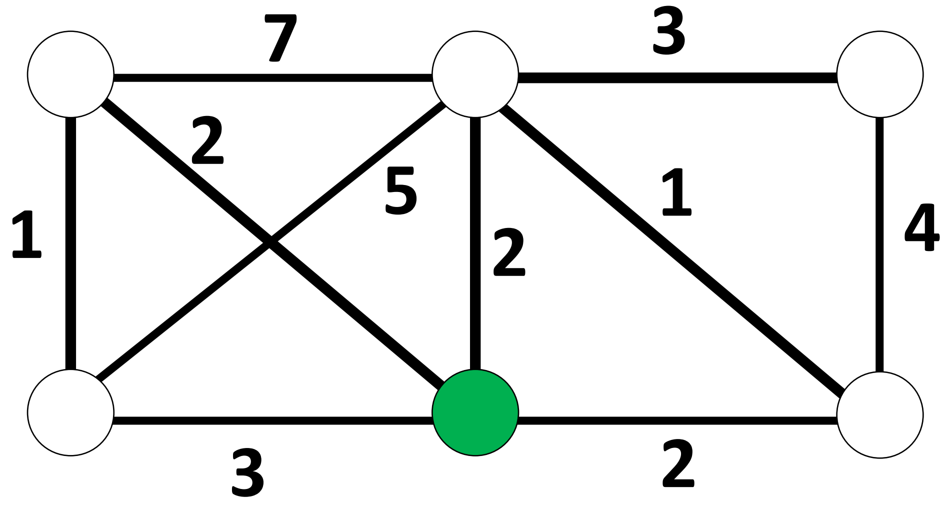
Prim’s MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



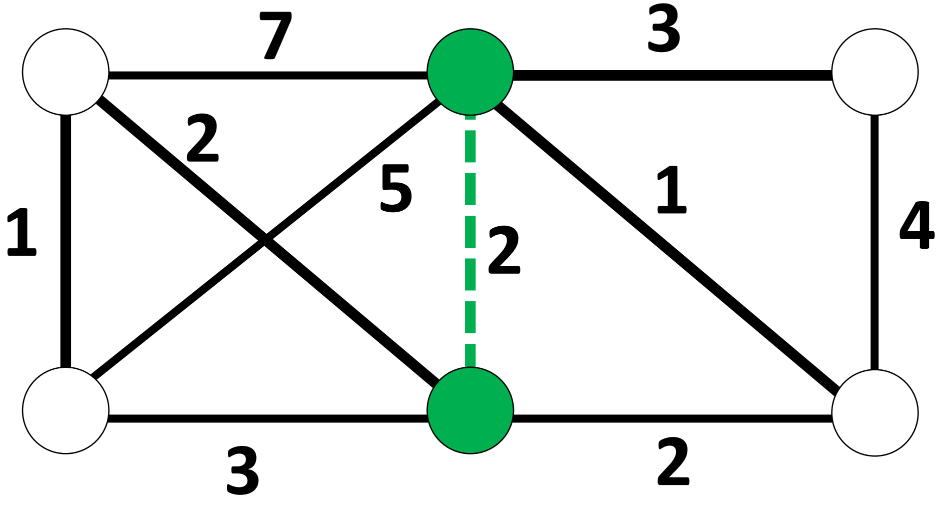
Prim’s MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



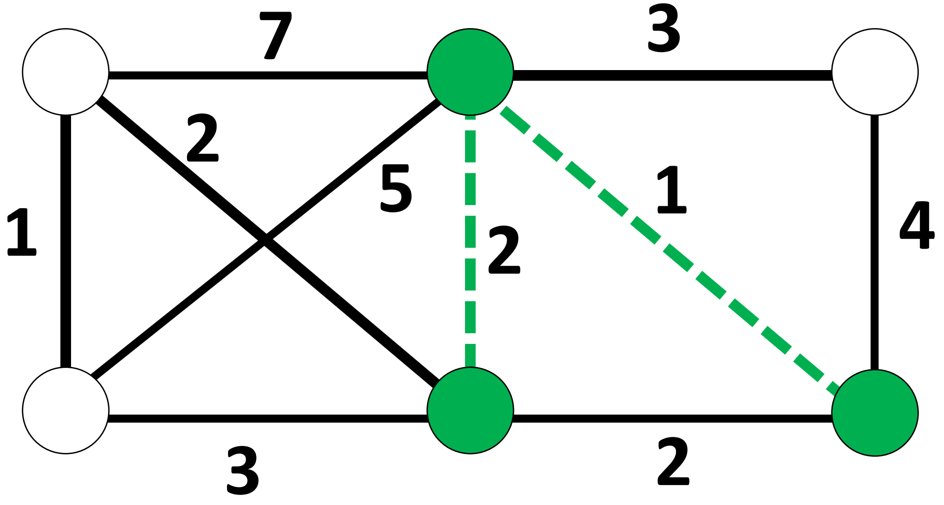
Prim’s MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



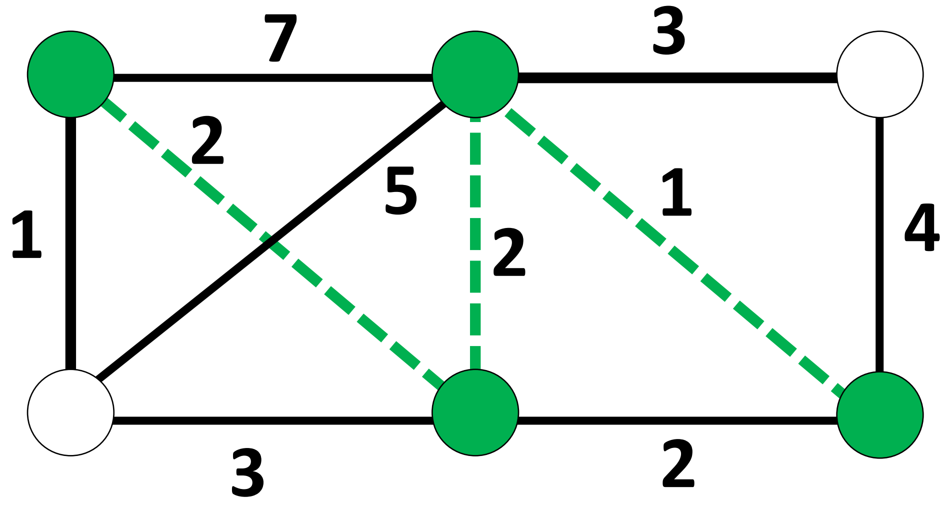
Prim’s MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



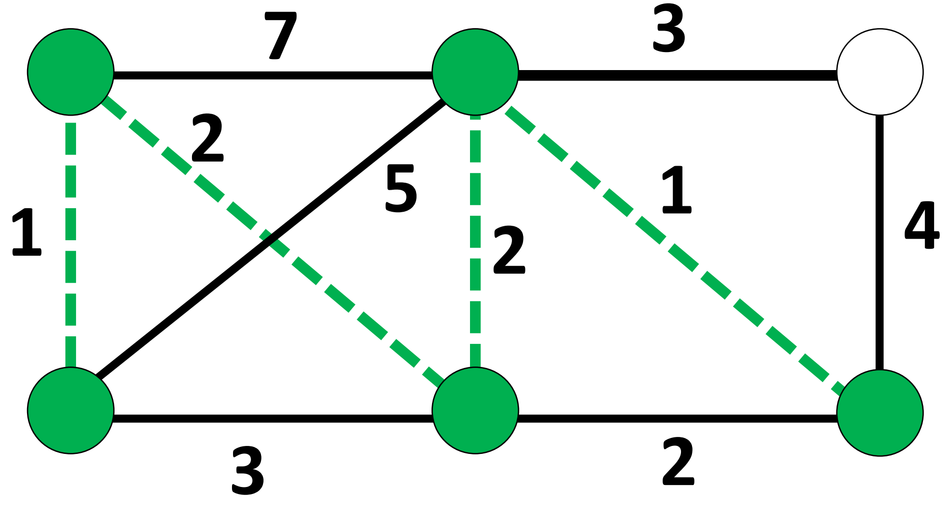
Prim’s MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



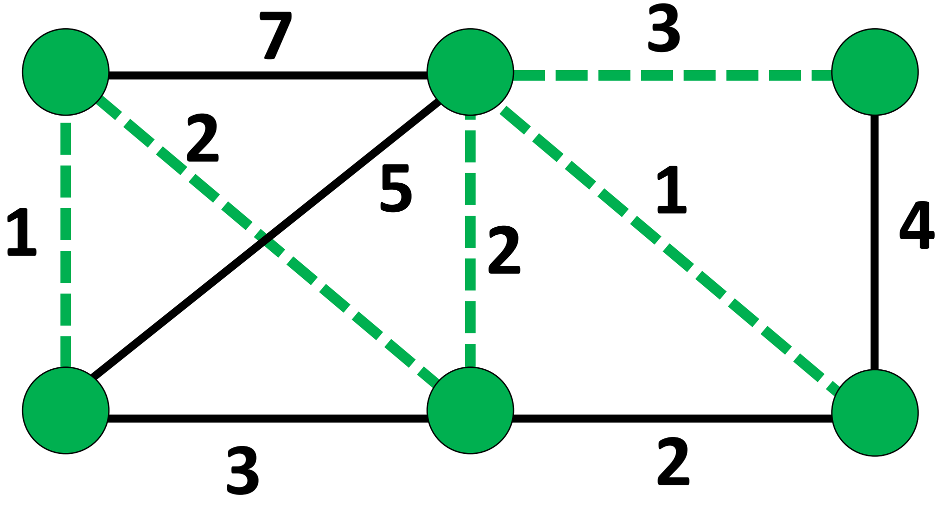
Prim’s MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.



Prim’s MST Algorithm

Beginning at any node, add the node that can be connected as cheaply as possible to the tree we are building.


7
3
2
5
1
1
2
4
3
2