Minimum Spanning Trees

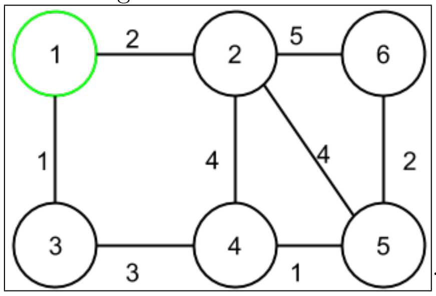
1 Minimum Spanning Trees: Prim's Algorithm

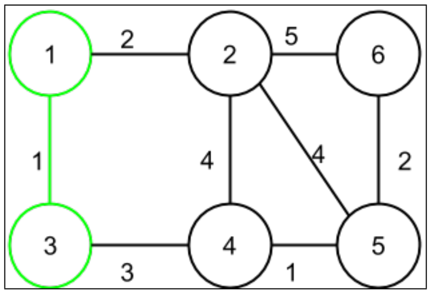
A Spanning Tree is tree (a graph with no cycles) that includes all the vertices. The **Minimum Spanning Tree** is a Spanning Tree that has the minimum weight. To find the Minimum Spanning Tree, we can use 1 of 2 algorithms, **Prim's Algorithm** or **Kruskal's Algorithm**.
Before we get started on these Algorithms, we will discuss the Cut Property. The cut property says that,

given two sets, the smallest edge that connects the two edges will be a part of the Minimum Spanning Tree.

Prim's Algorithm is very similar to Dijkstra's Algorithm in general. You have a starting vertex and at each vertex that you travel to, you pick to go the the closest vertex that does not cause a cycle. The vertex that is chosen does not need to be the one that you just visited, it just has to be a neighbor to a vertex that you have visited during the algorithm.
Prim's Algorithm uses a Priority Queue in order to pick the minimum edges that are adjacent to the current

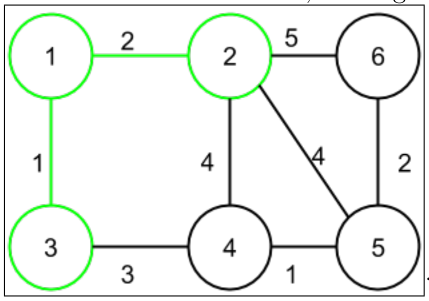
vertex. This is very similar to Dijkstra's algorithm except vertices are not given the priority based off distance from the starting point, it is given the weight of the minimum edge connected to it. Let's apply Prim's Algorithm.

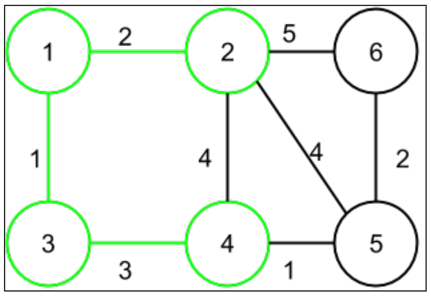


$\longrightarrow$


We started off at the vertex 1. We enqueue the edges that are next to 1, which are of weights 2 and 1. We

know that 1 is less than 2, so we go to vertex 3.

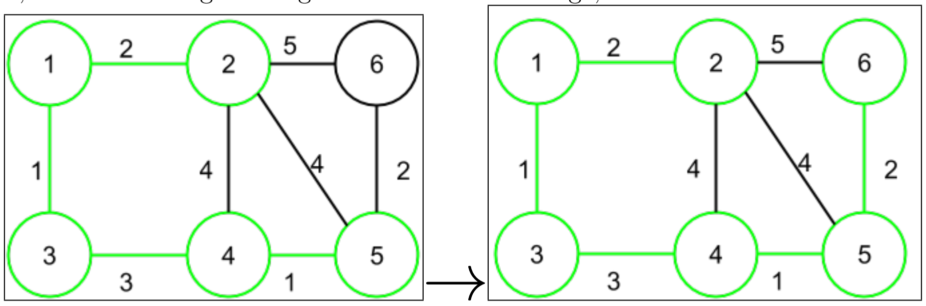


$\longrightarrow$


We enqueue the neighbors of 3 at this point, and we compare its neighboring edge of 3. Currently, we have

2 edges of weights 2 and 3, 2 is less than 3 so we go visit 2. We enqueue it's neighboring edges which are 4,

4, and 5. The edge of length 3 is the smallest edge, so we traverse it and visit vertex 4.

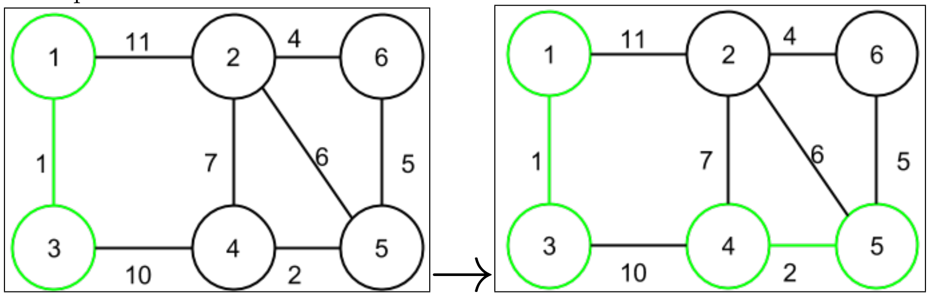


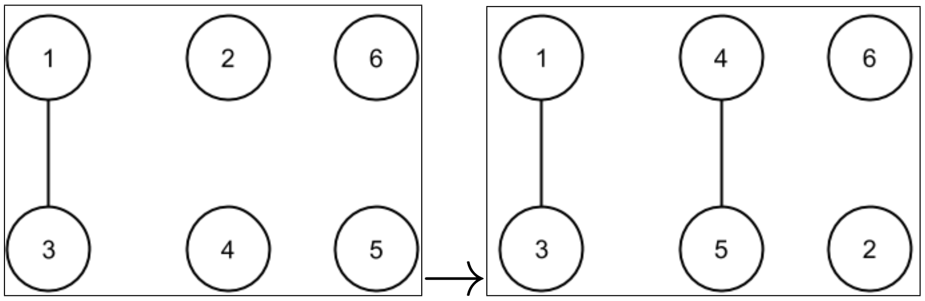
We enqueue 4's neighbors which have not yet been enqueued, which leads us to add an edge of 1 that is between 4 and 5. This edge is currently the smallest edge so we move to 5. 5 has one edge not enqueued yet and it has a value of 2, so we put it in, see it's lower than the others and we are done.
The runtime of Prim's Algorithm is $E\log(V)$ because we have $Vlog(V)$ time for inserting all the vertices into the heap, $V\log(V)$ for the deleting the min and $E\log(V)$ for changing the priority of the vertices. We sum these up to get $O(E\log(V)+V\log(V)+V\log(V))$ this can be simplified to $O(E\log(V))$.

2 Minimum Spanning Trees: Kruskal's Algorithm

Kruskal's Algorithm runs slightly differently than Prim's Algorithm. Instead of starting from some vertex, you put all the edges into a priority queue and add the ones that are the smallest and don't cause a cycle to your Minimum Spanning Tree.
The difference that Kruskal's algorithm has with Prim's requires that a different data structure be used.

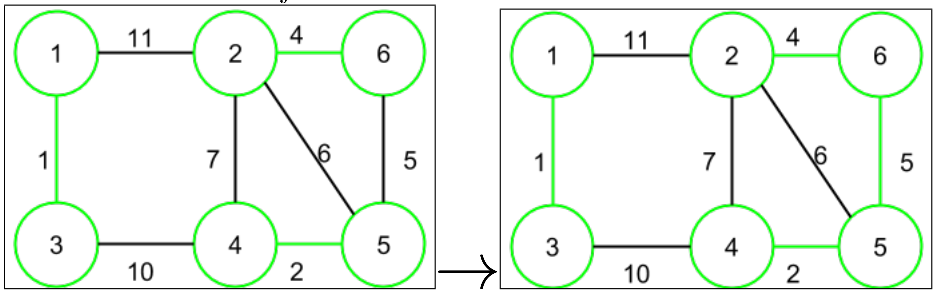
The edges are sorted from least to greatest, and a core Union Data Structure is used. In this application of a Union Data Structure, each vertex is it's own "tree" and it gets connected to another vertex. This Union Data Structure allows us to ensure that we do not have cycles in a very quick manner. Here is a step by step illustration of how to Union Data Structure works for Kruskals. We will go through an example of how kruskal's works. Underneath each graph will be the Weighted Quick Union with Path Compression that corresponds to it.

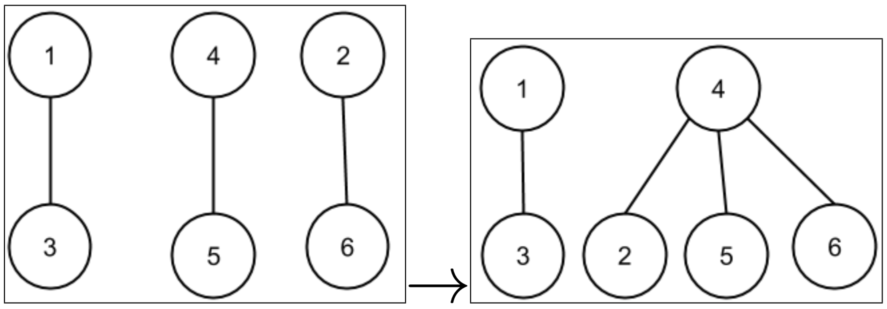




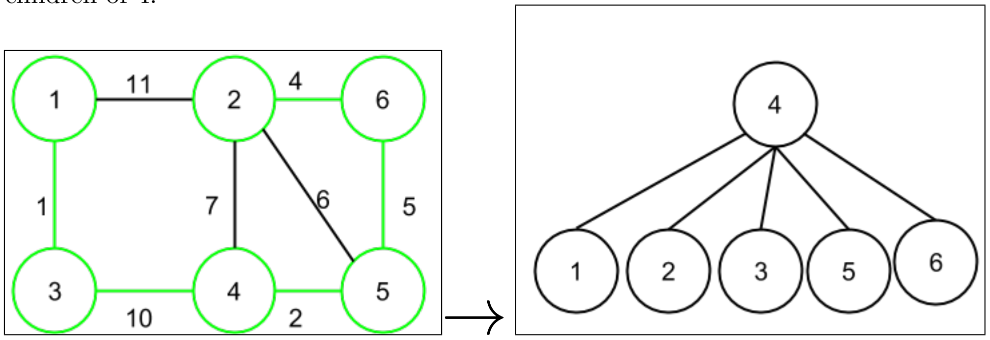
In this initial step, we find the smallest edge, which lays between 1 and 3. We then connect them in our

disjoint set data structure. After this, we find the next smallest edge which lays between 4 and 5, we also connect these in our disjoint set structure.





In this step, we again found the next smallest edge which is between 2 and 6 and once again we connect them in our union data structure. At this point, there are no vertices that are not connected to one other vertex. We find that the next smallest edge is between 5 and 6. We connect the two in our data structure. Since 6 is the item that is connecting to 4, we call find on it and 2 and then make them both immediate children of 4.



In this final step we see that the next smallest edge is of weight 6 and lays between 2 and 5; however, since

this would cause a cycle, we ignore it. We do the same thing for the edge weight of 7, our next choice, of weight 10 will not cause a cycle so we can go to it. We connect it in the Weighted Quick Union with Path Compression, and since 3 connects to the vertex 4, it calls find on itself and 1 which makes them both immediate children of 4.
The runtime for Kruskal's algorithm depends on what assumptions you make. If you make no assumptions,

the runtime is $O(E\log(E))$. This is because you would need to make a priority queue of all the edges which takes $O(E\log(E)$ time and deleting the min will take $O(E\log(E))$ time. the sum of this is $O(E\log(E) + E\log(E))$If you make the assumption that the edges are already sorted, it will take $E\log(V)$ time because deleting the minimum will take constant time. However union and connecting in the Weighted Quick Union With Path Compression will take worst case $lg(V)$ time and this is done E times so the final result will be $E\log(V)$ time for presorted edges.