**Greedy Algorithms**

Table of Contents

[Minimum Spanning Tree 3](#_Toc63011262)

[Naïve Approach 4](#_Toc63011263)

[Greedy Approach 5](#_Toc63011264)

[Prim’s Algorithm 11](#_Toc63011265)

[Kruskal’s Algorithm 16](#_Toc63011266)

In a greedy approach, we take the best local solution and we expect that it will lead us to the best global solution. In contrast to dynamic programming, where we try all possible choices, in the greedy approach, we make our choices intelligently.

## Minimum Spanning Tree

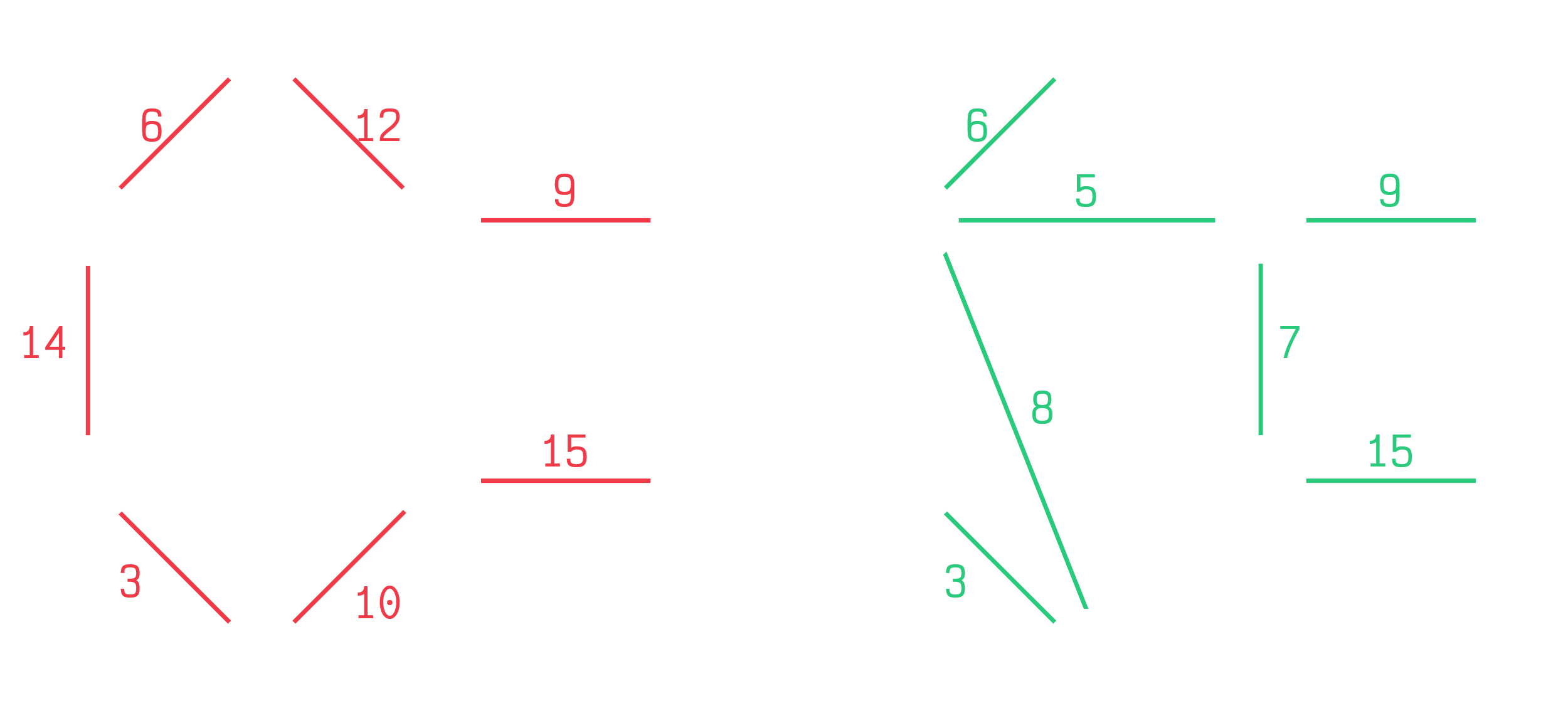
Formally, a tree is a connected acyclic graph, meaning there are no cycles and all the vertices are reachable from all other vertices. For vertices, there are exactly edges.

The spanning tree for a graph is a subset of the edges of such that a tree is formed. It must include all the vertices of . Since it is a tree, for vertices, we will be taking edges. Spanning trees can be formed for either undirected or directed graphs, but we will not be considering the directed cases.

In a weighted graph, , every edge has some weight . The weight of a tree, , can be defined as the sum of the weights of all the edges, i.e. .

We can create multiple spanning trees from a single graph. The minimum spanning tree is the one which has the minimum weight.

Consider the example below:



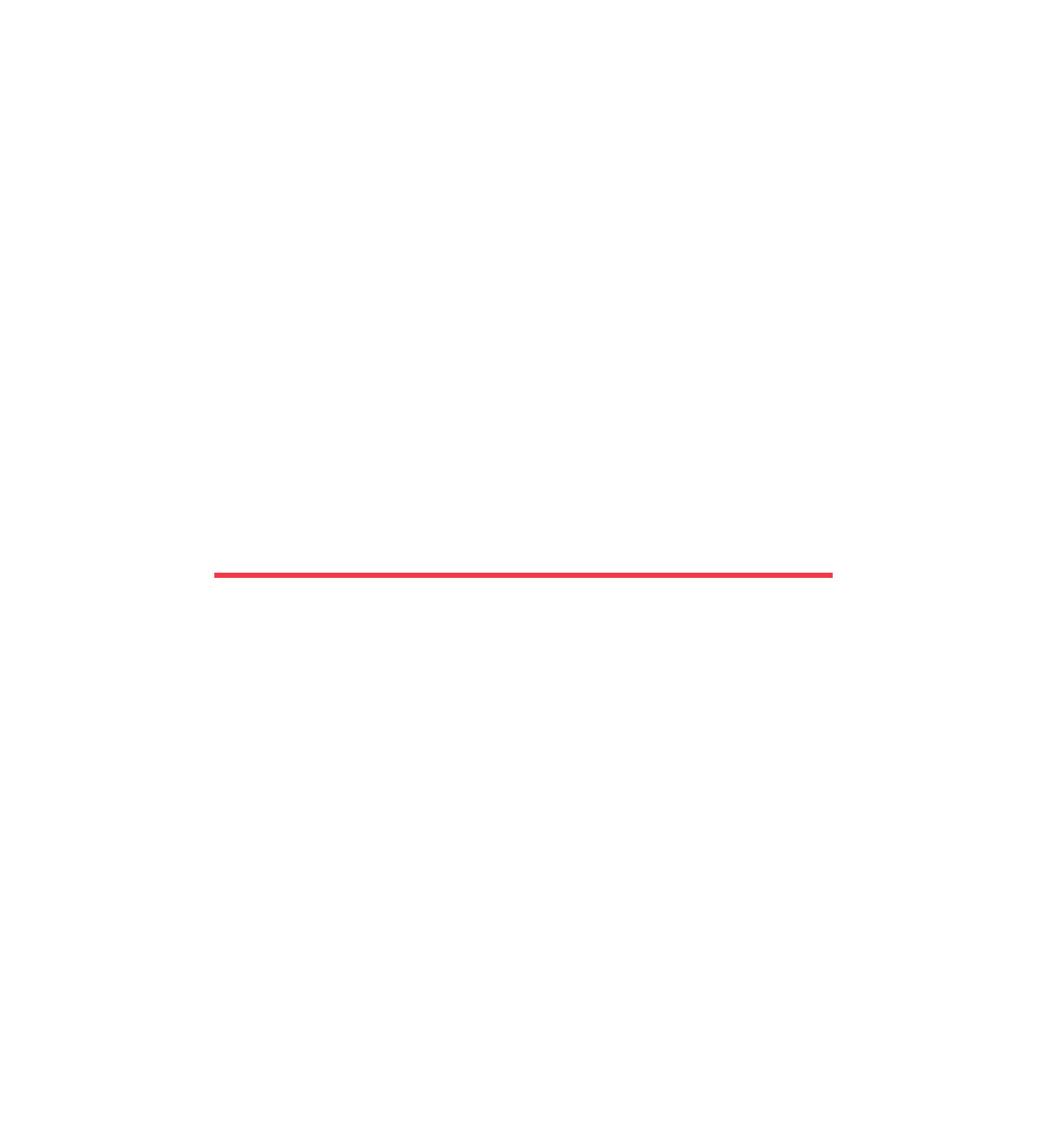
Both of these are spanning trees for the same graph. All the vertices are reached, and the edges used never form a cycle. However, the tree on the left has a total weight of and the tree on the right has a total weight of . There could of course be many more spanning trees for the same graph, but we will find that that tree formed on the right is actually the one with the least weight. There could be more spanning trees with the same weight though. Thus, this is (one of) the minimum spanning tree(s) for this graph.

Now we just need to figure out how to find the minimum spanning tree.

### Naïve Approach

The naïve approach is the brute force approach. We check all the possible spanning trees and take the one with the minimum weight. The problem is, this has an exponential runtime.

Consider this graph:



Say we take the central edge first, the one marked in red. There are other vertices, each of which are connected to both of the vertices we have already taken with one edge each. Thus, we only need to consider one of the two edges for each of those vertices. We have to make a choice between two options. For these vertices, we can make combinations of choices.

If we had such vertices, each connected to both of the original vertices in the same way, we would have choices. Thus, there would be different trees here, which means we have an exponential time complexity.

### Greedy Approach

Now for the one we will actually use. Before we dive into this, we need to understand how Greedy algorithms work

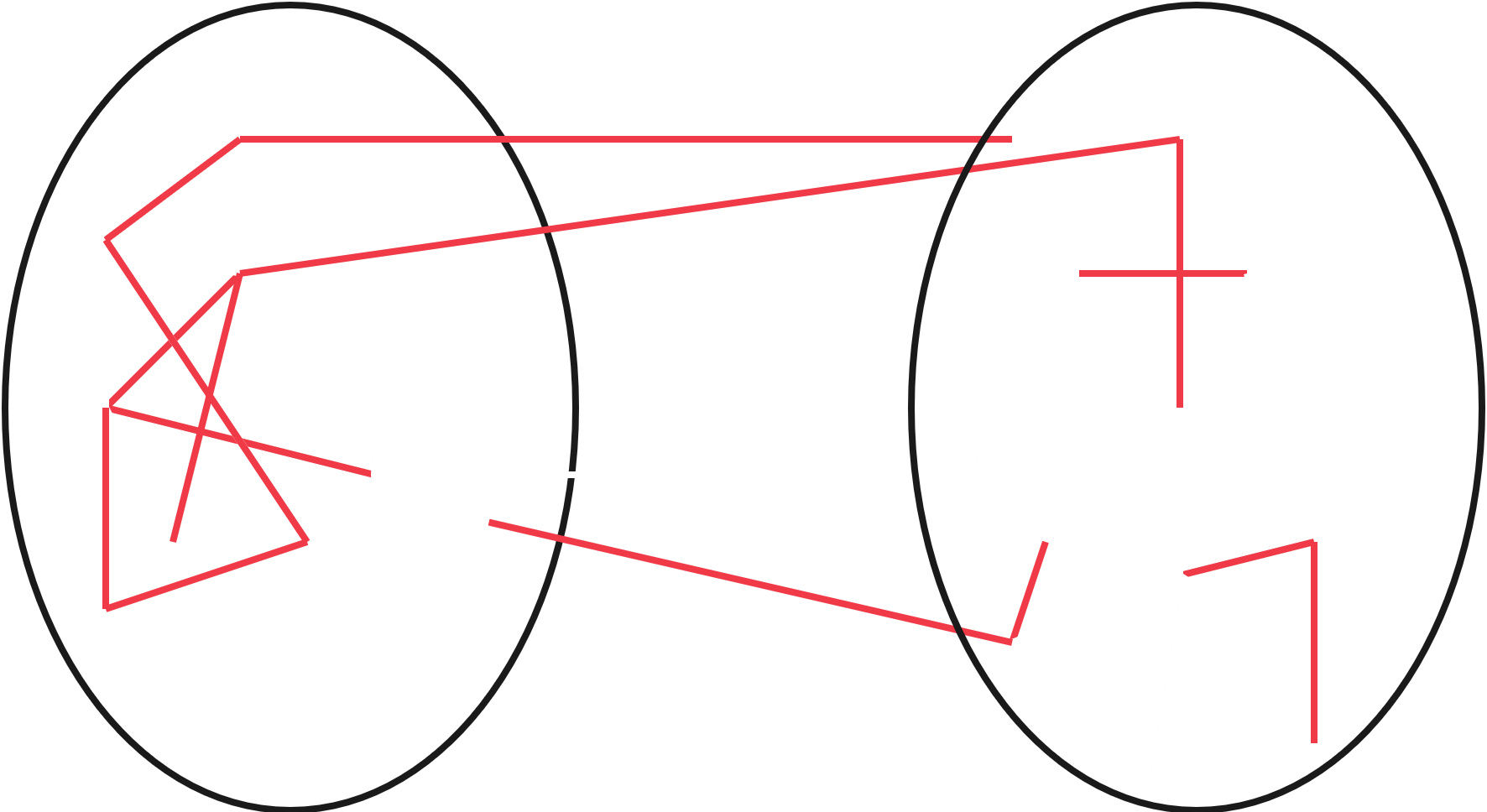
#### Greedy Properties

The first property is the Optimal Substructure property. This means that if we divide the problem into subproblems, the optimal solution to the entire problem will consist of the optimal solutions for all the subproblems. This is essentially what we did in dynamic programming.

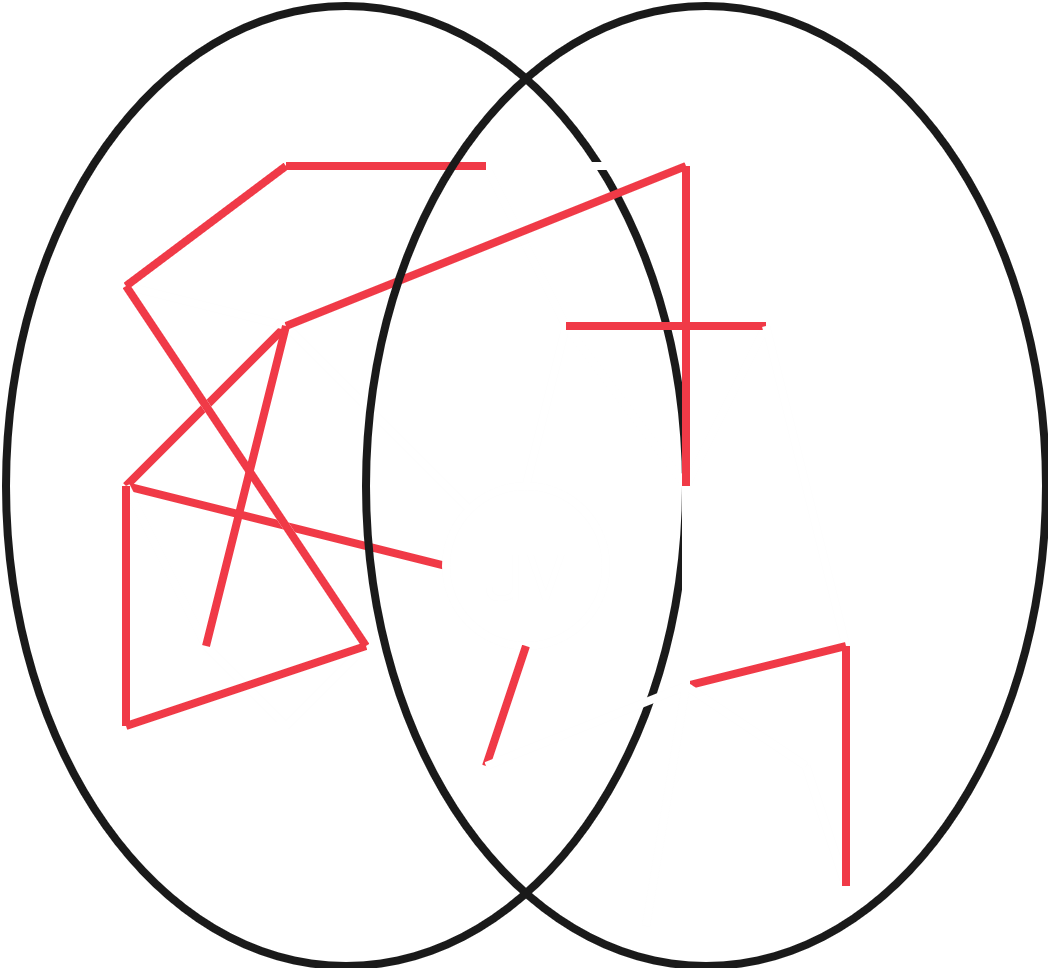
The second property is the Greedy-Choice property. This means that locally optimal solutions will lead to a globally optimal solution. Unlike dynamic programming, we do not try all the possible choices from a given state. Instead, we only take the optimal solution at that state. We make the decision for this local solution based on some rule, which ensures that the global optimal solution will be reached.

#### Optimal Substructure of Minimum Spanning Tree

To be able to create a Greedy algorithm to solve the Minimum Spanning Tree (MST) problem, we need to prove that an MST has these properties. To do this, we will use the graph below:



Remember that the optimal substructure property will be held if we can show that the globally optimal solution consists of the locally optimal solutions. Say the locally optimal solution is to take the edge between and . Once we have found this, we can contract that edge, merging and . This will give us the graph below:



In the original graph, there was a vertex that was joint to both and via separate edges. In the merged graph, we can only have one edge from the merged vertex to that vertex. This edge should be the one that was smaller between the two edges from the original graph, seeing as though we want to minimize the weight. All other edges remain the same.

Notice that we managed to create a smaller subproblem, . can also be denoted as , literally without .

Now we need to repeat the process on this smaller substructure. We keep going until we have found all the edges that are part of the MST. Once this is done, we come back, unmerging all the merged vertices. Doing this will return all the edges that we found, which will give us the MST.

Say we did the whole process and unmerged nodes and got back to . We now have the MST for . Let’s denote this as . All we have left to do is unmerge the vertex . Once we do this, we will just be adding to . Then we can say that we have the MST for , , where .

We can prove this. Let’s say is the MST for and . will give us , with the two vertices connected by merged together. Remember that a tree must be acyclic, so there cannot be any more paths connecting the vertices that connected. Thus, must be a spanning tree for . We do not know for sure yet that this is the MST.

We do know that is the MST for . For to be the MST, must be true. If it were not, then could not be the MST.

We previously defined

Since is an MST, cannot have a weight less than . Thus, their weights must be equal. As such, must be an MST.

#### Dynamic Programming Solution

We just proved that the optimal substructure property holds. Thus, we can use dynamic programming. The only difference in this approach is that we are not going to be smart about picking edges. We just try everything.

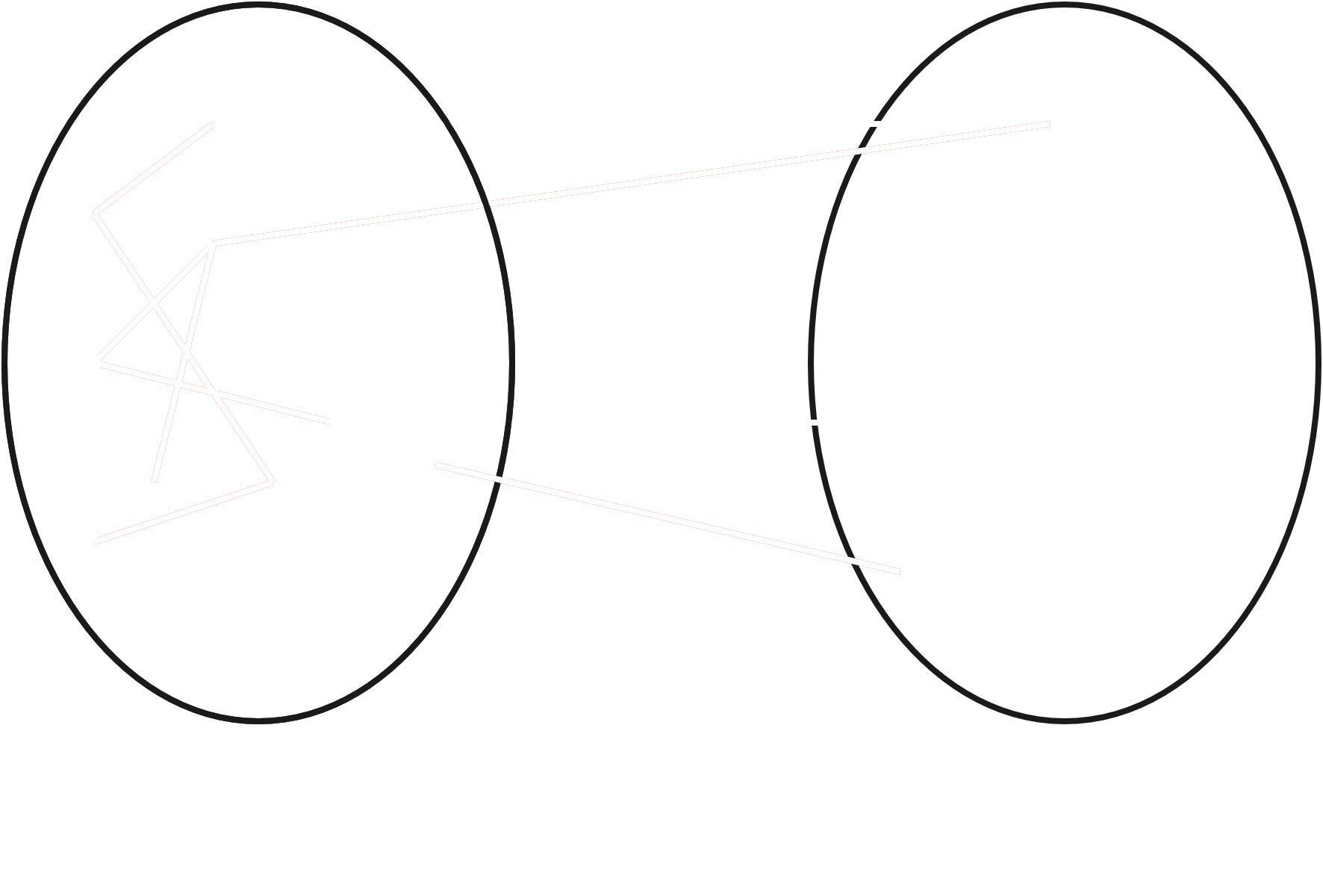
We guess an edge . We contract the edge. We get a new graph. We recursively repeat our steps for this new graph. We find the MST in the remaining graph. We decontract (not a word) the edge we contracted from the original graph. We repeat for all edges and find the minimum weight.

We will be memoizing repeated edges here, so that part is all good. However, the subproblems are all possible sets of edges, of which there are . Thus, no matter how fast we solve each subproblem, the time complexity will be .

Now, we will find a Greedy solution which has a polynomial runtime. Achieving this polynomial runtime is the entire reason why we are trying to find a Greedy solution, because a Dynamic Programming approach cannot give us this runtime.

#### Greedy Choice Property of Minimum Spanning Tree

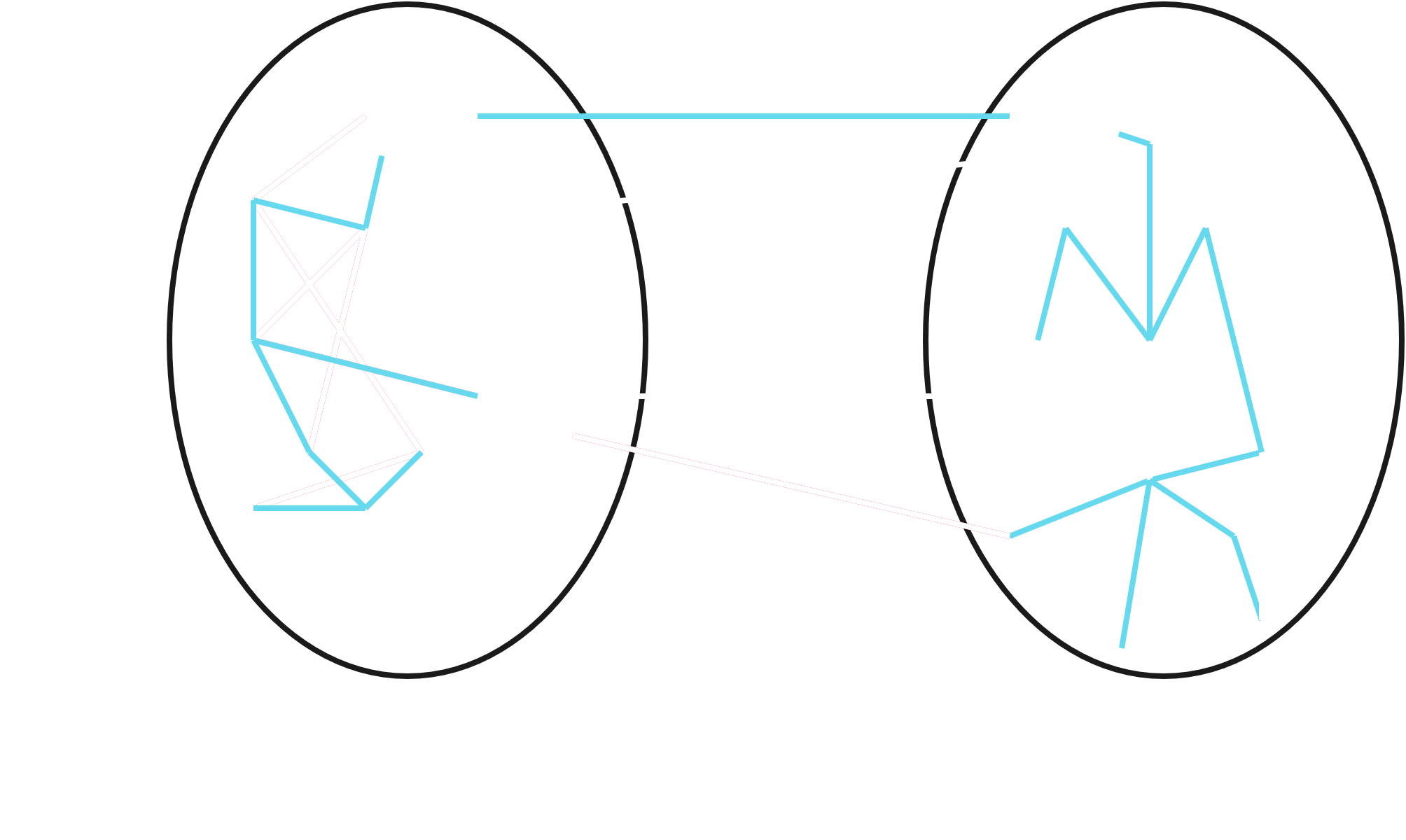
Consider this graph:



The nodes and edges have been divided into two distinct groups. Say we make a cut, separating the groups into two subgroups, one containing a set of vertices , and another containing a set of vertices ( without ). The edges that are cut are called the crossing edges.

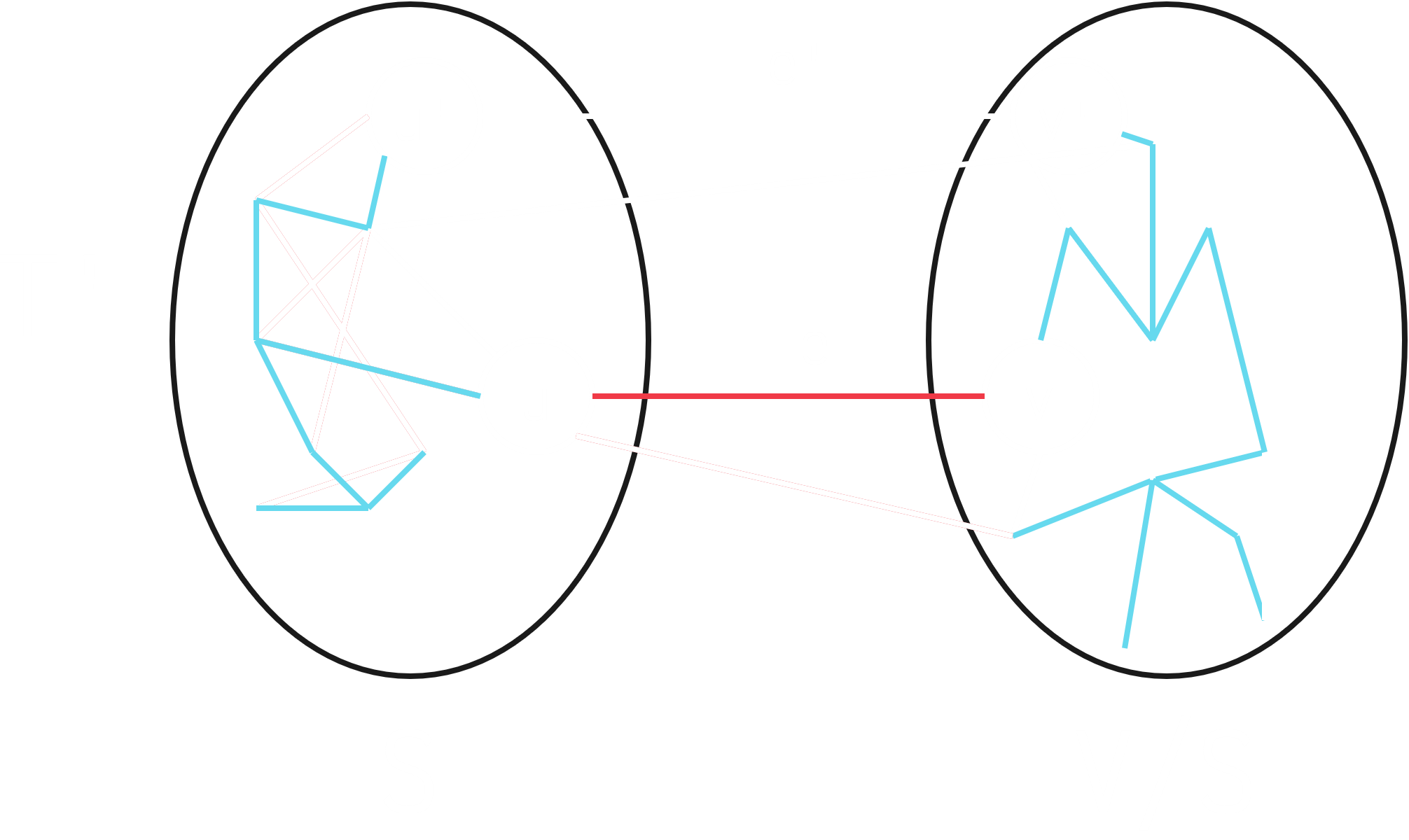
Let be the least-weight crossing edge such that , where is a vertex from and is a vertex from . This edge, , will be part of the minimum spanning tree. This is the greedy choice property.

We can prove this property. Consider this graph:



Here, the blue lines make up the MST, . We are considering that the least-weight edge, , is not a part of the MST. If not a part of the MST, there has to be some crossing edge that is part of the MST, since otherwise there would be unconnected vertices. Let this crossing edge that is part of the MST be .

Our argument is that we can use the Cut and Paste method, i.e. we can replace with and still get an MST. In that case, it would be justified to take . We can do this replacement since the entire point of a spanning tree is that there is exactly one path connecting any pair of vertices, meaning and were connected before through another path. We are just using a different path now. The result will still be a spanning tree. Let this spanning tee be , i.e.



For , the weight can be calculated as follow:

Since the only difference in weight between and is that we removed one edge and added another, we only need to be concerned with these two edges. Since , .

If is a minimum spanning tree, we cannot have another spanning tree that has a lesser weight, thus, , meaning is also a minimum spanning tree.

### Prim’s Algorithm

Prim’s algorithm is one of the algorithms that uses the Greedy properties we just saw to solve the minimum spanning tree problem. It will take subgroups, take the edge with the least weight amongst the crossing edges and continue repeating the process on the subgroups.

Prim’s algorithm uses a priority queue, . The priority queue will store the group of vertices that we previously labelled .

Initially, we will have one group, , that is empty. We will pick a single vertex and place it into . All the other vertices will thus be part of and will go to the priority queue.

All the outgoing edges from the single vertex in are now crossing edges. The priority queue will store the vertices in based on the weight of their crossing edges. Thus, we will be able to pick out the vertex that has the crossing edge with the least weight. We will add this vertex to and repeat the process, making the cut along the outgoing edges of the new vertex.

For every vertex, we will also store the ‘parent’. Thus, once we have chosen the first vertex, made a cut and found the second vertex, the parent of the second vertex will be the first one.

#### Formal Definition

1. Initially, and .
2. Now, we set for some arbitrary .
3. For every , that is every other vertex except , we set .
4. Next, we extract the minimum value vertex, , from . The first time we do this, .
5. is placed in . We can keep track of the fact that has been added to using a Boolean array.
6. For every , that is every vertex that is a neighbour of , we need to now check two conditions:
   1. Is , that is, is ?
   2. Is , the weight of the edge between and , less than .

If both of these conditions are met, it means we must decrease the key value of from the existing value to . We will also set the parent of to be .

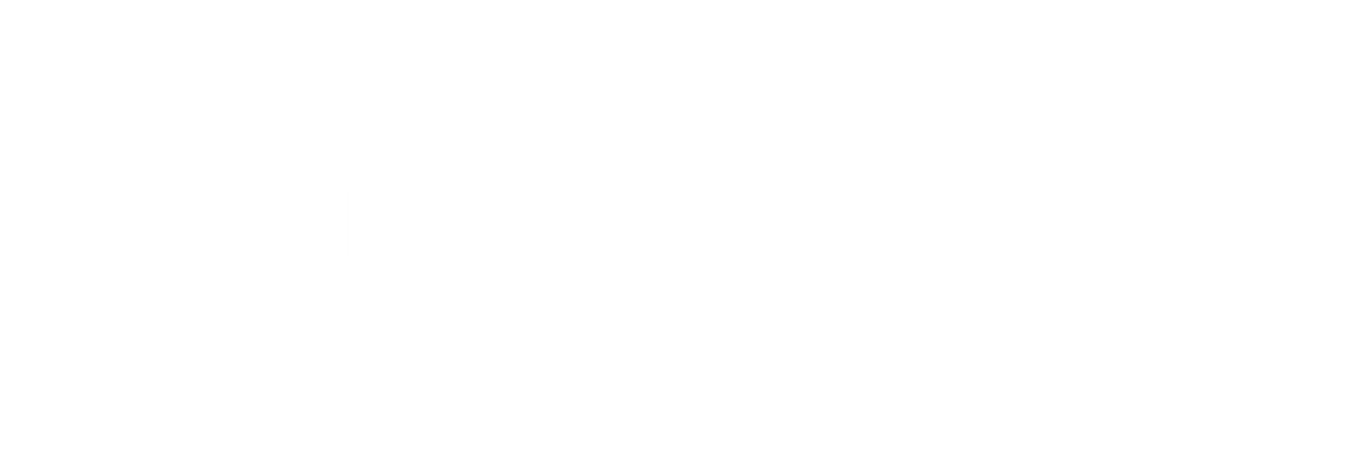
1. Steps 4, 5 and 6 are repeated in a loop while is not empty. Once this is done, we return , which is the set of pair values for every vertex from the MST and their parent.

#### Time Complexity

We inserted every vertex into the priority queue and extracted them one by one. If we use a Fibonacci heap, we will end up with a time complexity of for that part. Additionally, we visited all the neighbours for each vertex. In total, this means neighbours. Thus, the total time complexity is .

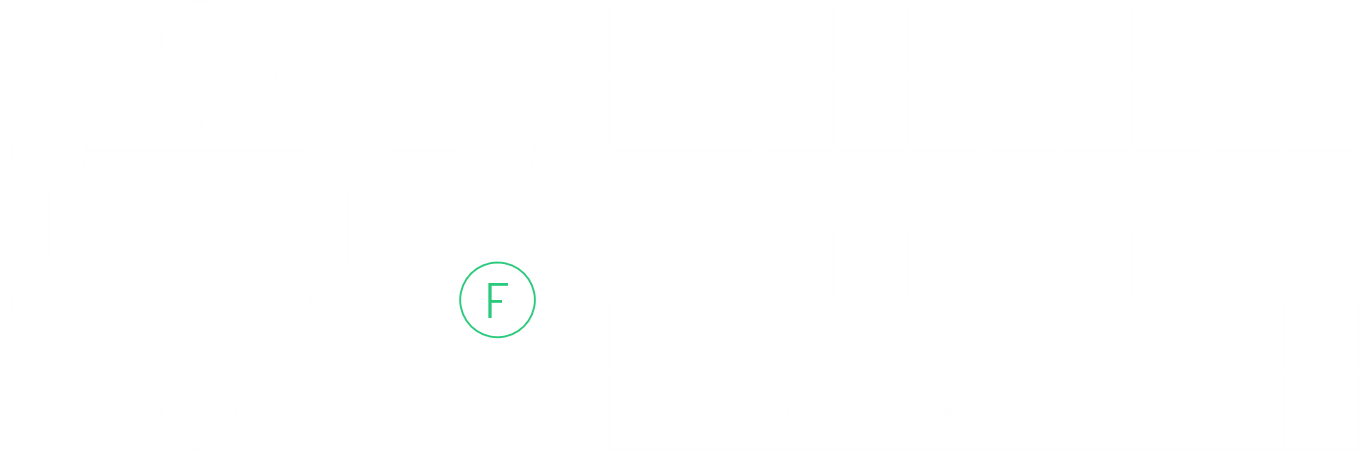
Prim’s algorithm is similar to Dijkstra’s algorithm and also has the same time complexity.

Example



Initially, everything is in the priority queue and is empty.

Say we pick as the initial node. We set the key for to and everything else to .

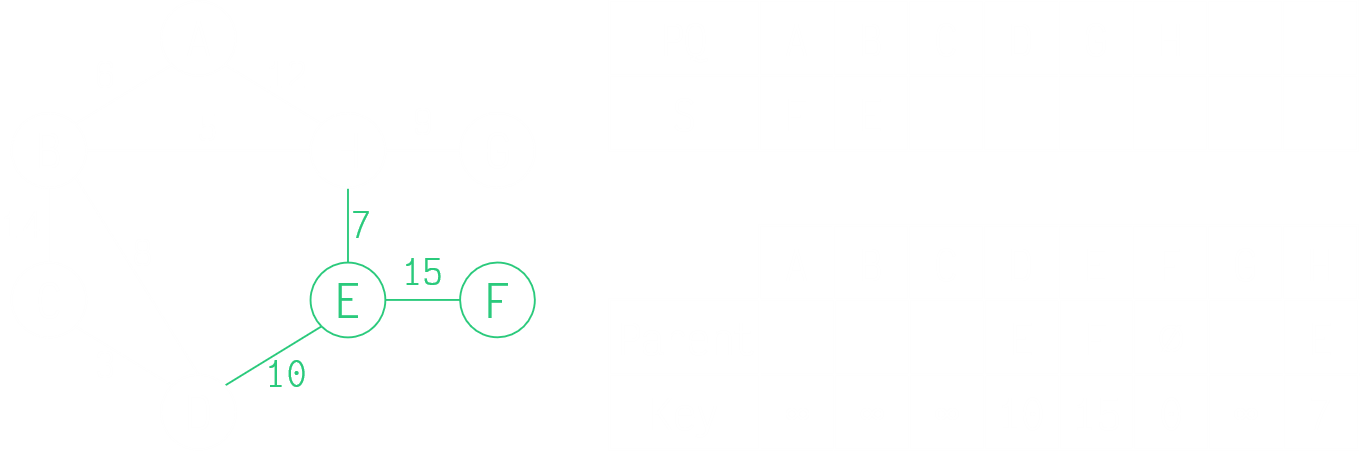


Thus, we extract . We can safely set the parent of to be now.

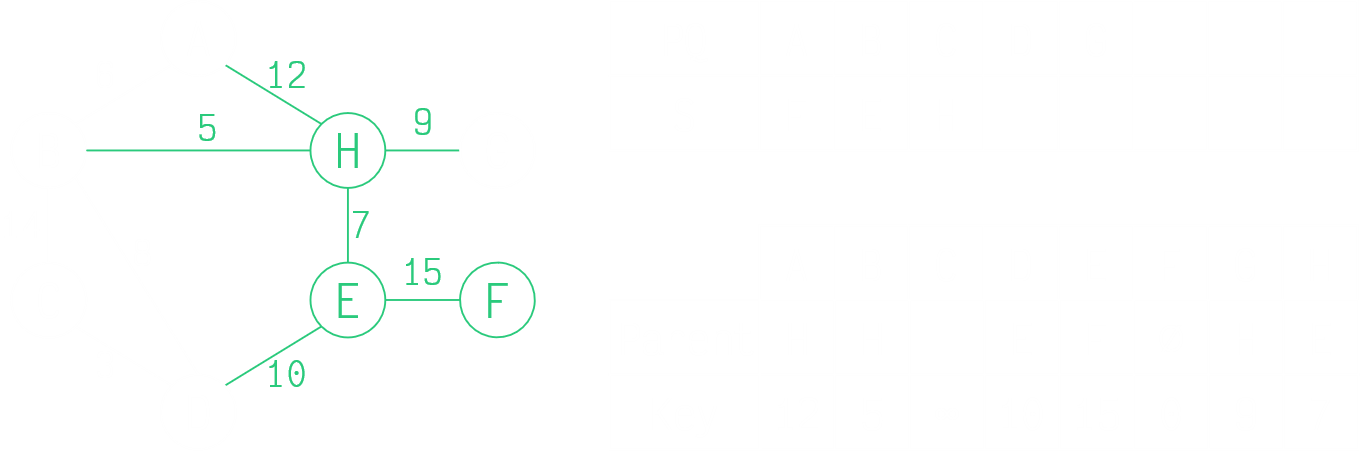
For every neighbour of , we first check if it is in and if its current key value is less than the weight of the edge if makes with . The only neighbour of is , and it does meet these conditions. Thus, we change its key value and set its parent to .

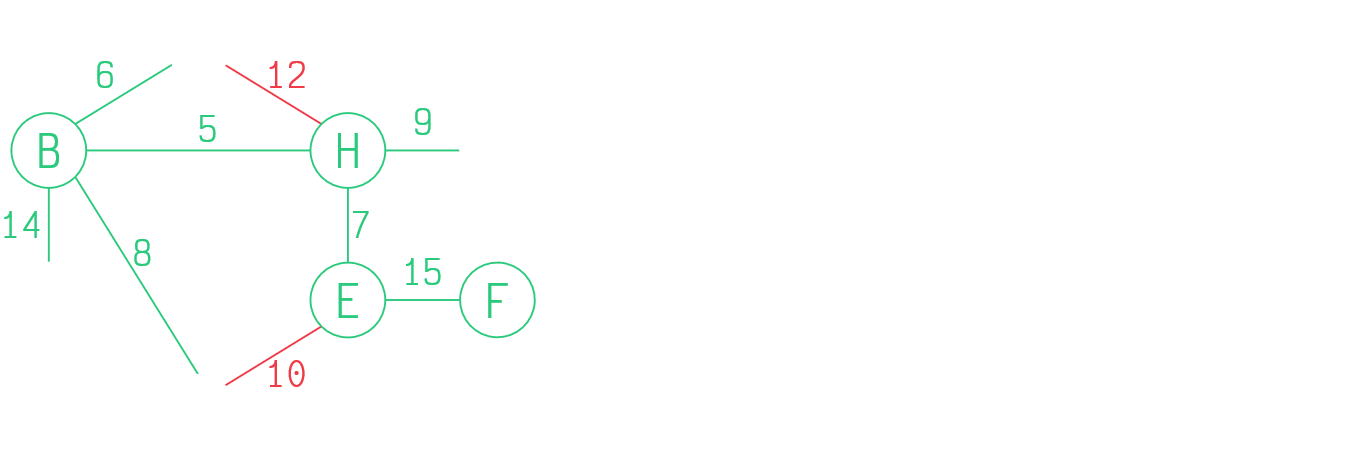


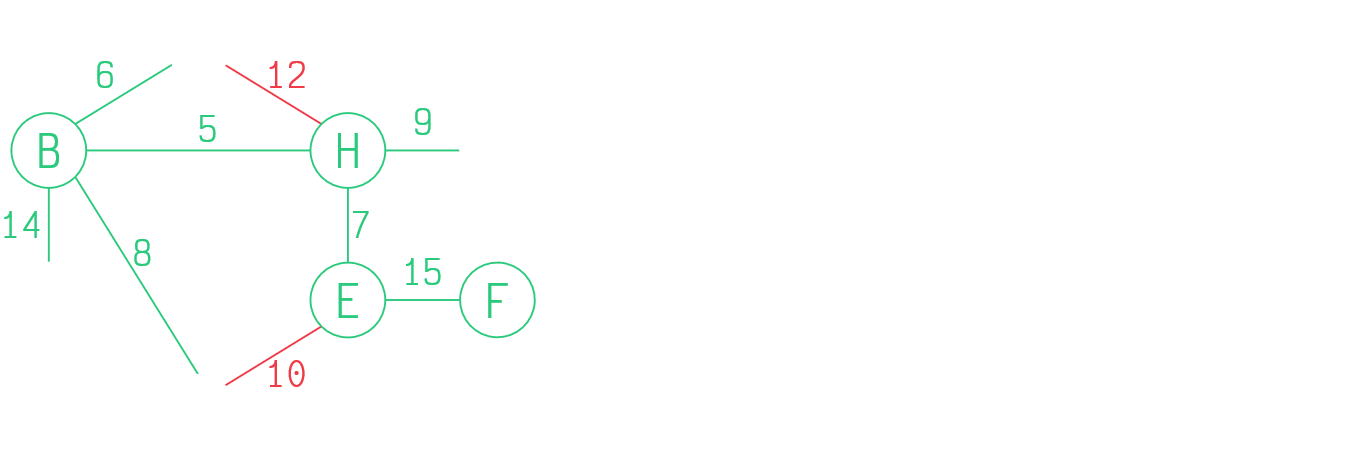
Now we extract . For every neighbour of , we again check the conditions and change key values and parents accordingly.

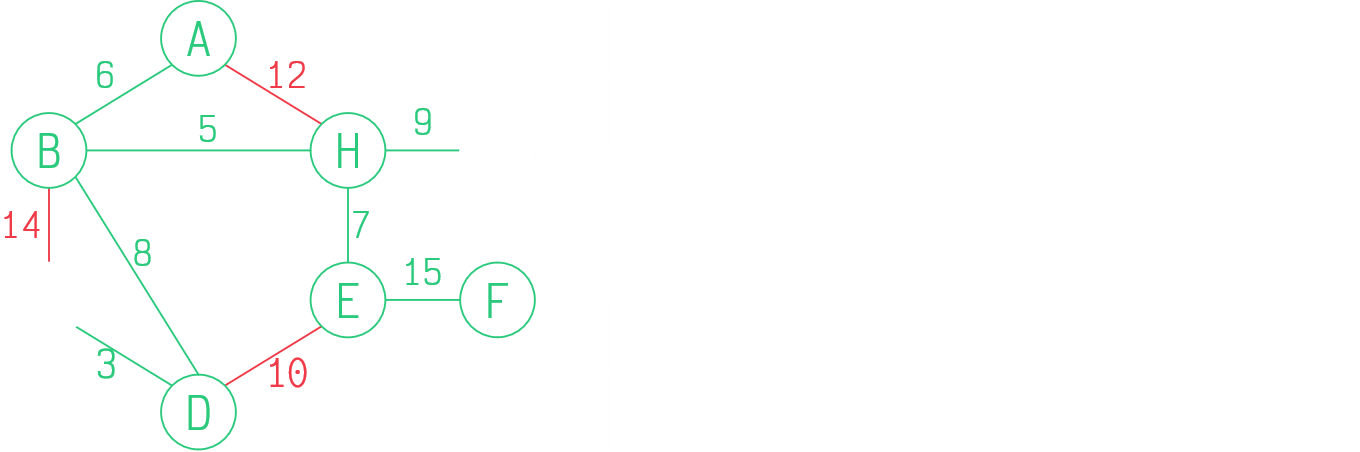


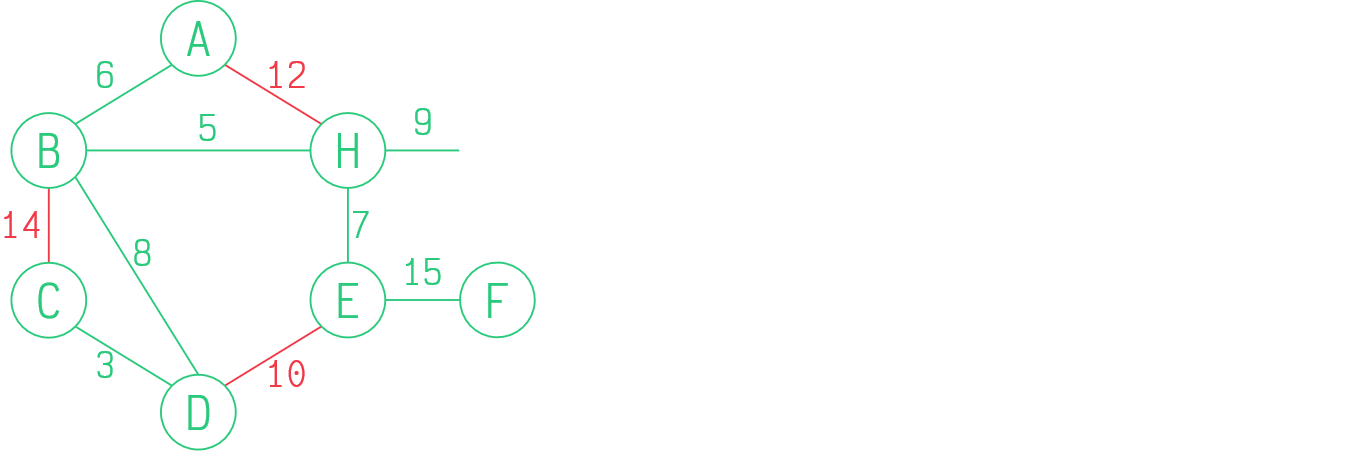
This process now continues.

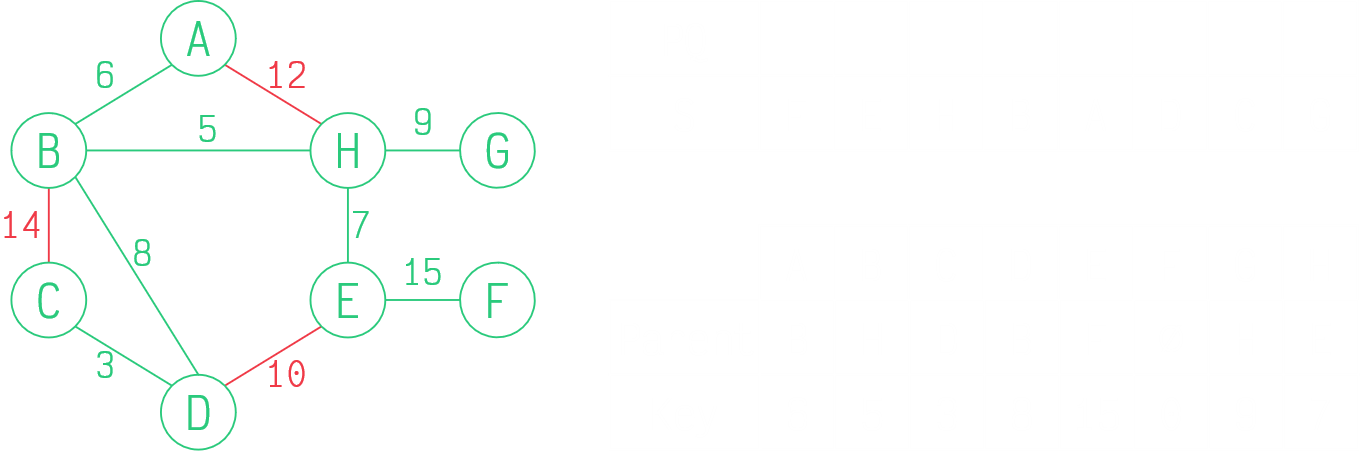












Thus, the MST is defined as:

### Kruskal’s Algorithm

In Kruskal’s algorithm, we will be using the DSU data structure to find minimum spanning trees. The actual process is very simple.

Let be the set of edges in the MST so far.

1. Initially, we do not know any of the edges, so .
2. For every vertex, , we call the make\_set(v) function, which creates a set with the vertex as the representative.
3. We sort in ascending order by weight.
4. For every , starting with the smallest one, if find\_set(u) != find\_set(v),
   1. We add the edge to the MST, i.e. .
   2. We put both the vertices in the same edge using union\_set(u, v)

The reason this works is because we are choosing which edges to add to the MST and ‘creating’ the tree as we go along.

#### Time Complexity

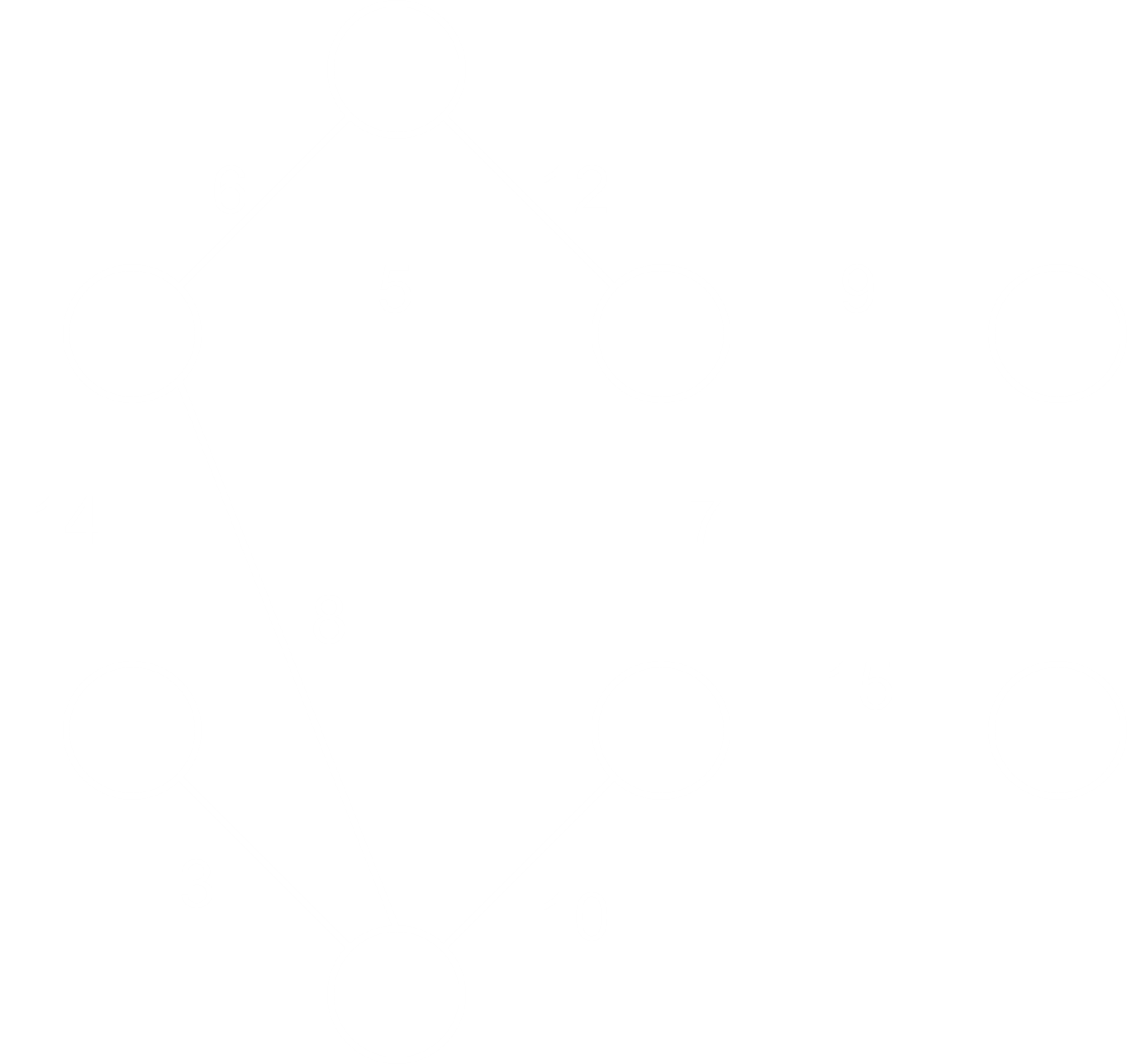
We need to perform the make\_set operation times, so we have a time complexity of there.

We need to sort the edges. If the edge weights are integers, we can use radix sort to get a time complexity of . This is the best case. We will need to use slower sorting algorithms if the edge weights are not integers.

For every edge, we have to perform the find\_set operation, which has a constant time complexity, and the union\_set operation, which has a time complexity of , making the total time complexity for this part .

Thus, the total time complexity is in the best case. This algorithm has a linear time complexity.

Example



The edges we will consider are, in order:

1. and are connected.
2. and are connected.
3. and are connected.
4. and are connected.
5. and are connected.
6. and are connected.
7. and are not connected, since they are from the same set.
8. and are not connected, since they are from the same set.
9. and are not connected, since they are from the same set.
10. and are connected.