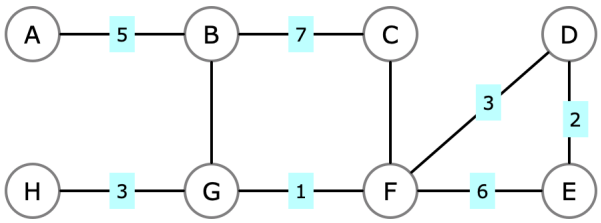
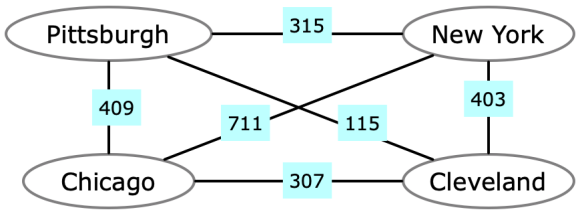
Graph Algorithms 3: the MST

1 Weighted graphs and the Minimum Spanning Tree

In this lecture we study algorithms on *weighted* graphs, those are graphs where each undirected edge has an associated weight. The figure below is an example of a weighted graph G. We use the notation $w(u, v)$ for the weight of the edge from vertex u to vertex v. In this lecture we assume all weights are positive.

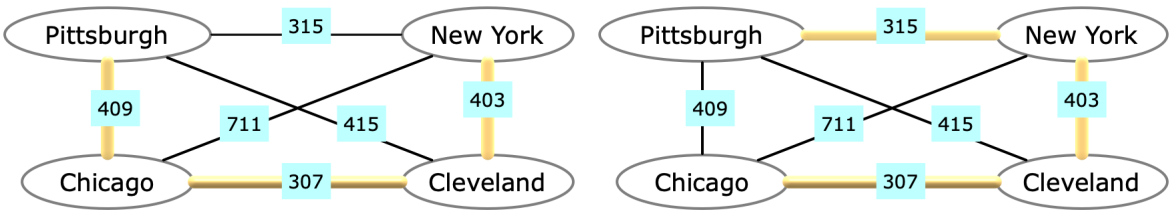


Weighted graphs are often used to model situations where a connection (an edge) has an associated weight that could equal its cost, its length, etc. For example, a set of cities could be represented as vertices of a graph, and the distances between any pair of cities is a weighted edge. Modelling this situation with a weighted graph is shown below:



1.1 Minimum Spanning Tree

Following the above analogy, suppose we wish to connect the cities with a set of telephone wires in such a way that every city could make a call to any other city along a path of telephone wires. If the cost of each wire is proportional to its length, we would like to do this is such a way that **minimizes** the total cost of the wires. This problem can be solved by computing the Minimum Spanning Tree (MST). In the figure below we show two ways of connecting the cities. The choice on the left has a total cost of 1119 and the choice on the right has a cost of 1025. The choice on the right represents the *Minimum spanning tree*: it is the minimum of all possible ways of connecting the cities.



Minimum spanning tree
Let G be a weighted undirected graph. The minimum spanning tree is a set of edges that connects all the vertices of G and whose total weight is **minimized**.

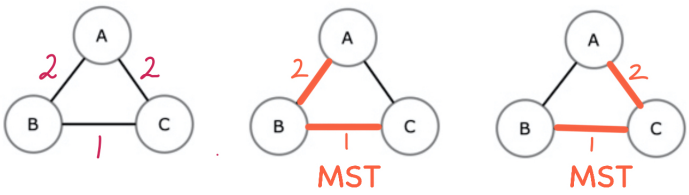
We begin this lecture by presenting specific properties of the MST, and then provide two algorithms for determining the MST: Prim's and Kruskal's.

1.2 Properties of the MST

The definition of the Minimum spanning tree results in several important properties.

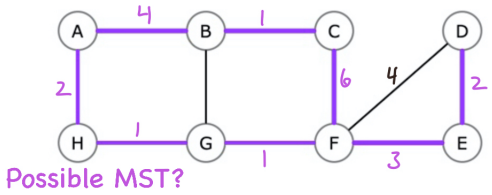
1. MST is not necessarily unique

The minimum spanning tree on a graph G is not necessarily unique. There may be several ways to connect the vertices that result in the same overall weight. The example below has two different minimum spanning trees each of total weight 3.



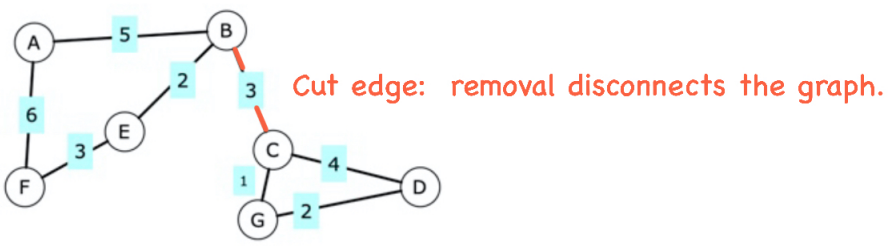
2. MST is a *tree*

The MST does not contain any cycles - therefore it is a tree. This may seem obvious from its name, but it is important to point out why this is true. Recall that we assume all weights are positive. Imagine for a moment that the minimum spanning tree contained a *cycle*, as in the figure below. Then certainly removing one of the edges on that cycle would *decrease* the total weight, and the vertices would still be connected. Therefore the MST does not contain a cycle, because removing it would decrease the total weight. In the example below, the selected edges contain a cycle, and therefore cannot possibly be the MST. Removing any edge on the cycle would still connect the vertices and would have smaller weight.



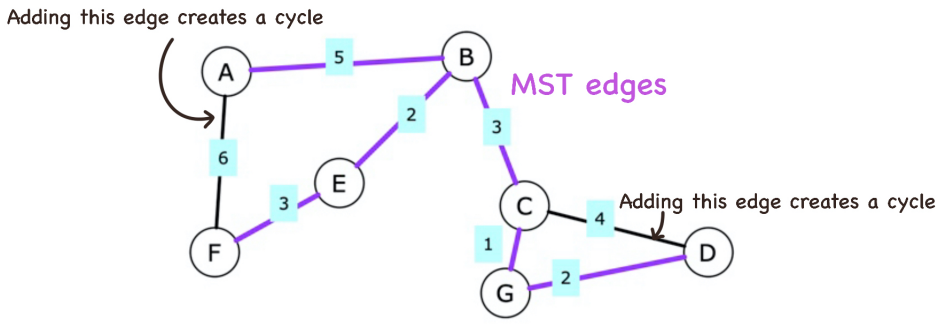
3. Any cut edge is in the MST

A cut edge is an edge of the graph G whose removal **disconnects** the graph. Any cut edge of G *must* be part of any MST. Why is this true? Again, imagine for a moment that were not the case, that some cut edge e were not part of the MST. Then since edge e is not included, G must be *disconnected*, therefore it is impossible that the MST actually connects all vertices of G since the edge e is missing.



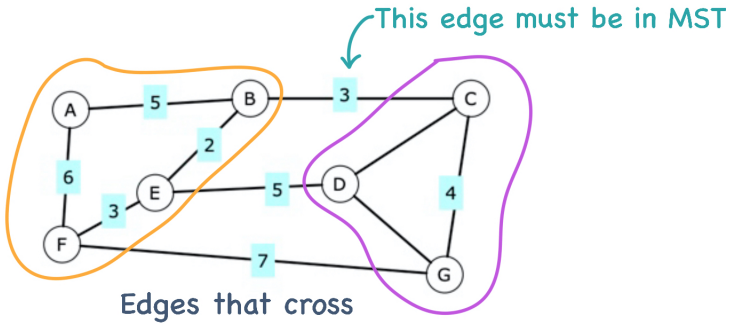
4. Adding any other edge from E to the MST creates a cycle

Since the MST is actually a *tree*, then if we add any additional edge, we create a cycle. This is actually a property of trees. As can see in the figure below, the MST edge already connect all the vertices. For example, there is a path from vertex A to vertex F in the MST. If we *add* the edge from A to F, then we create a cycle (the path from A to F plus the edge F to A).



5. The lightest-edge across any partition must be in the MST

Given any graph G, suppose we partition them (in any way) into two groups. There may be some edges of the graph that "cross" this partition. The *lightest* edge out of all the edges that cross from one "side" to the other, must be part of the MST.

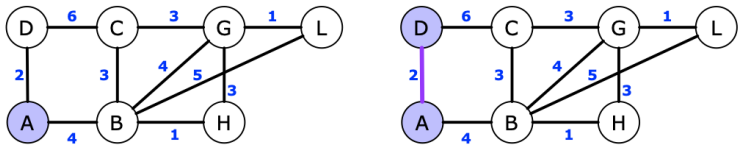


In the next section, we look at the first algorithm for constructing the MST.

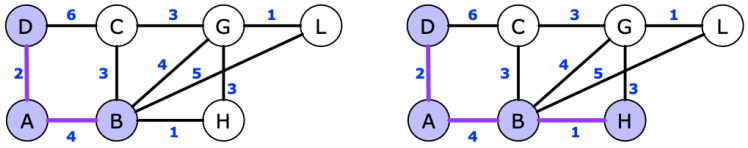
2 Prim's algorithm

The first algorithm that we study in this lecture is called *Prim's* algorithm. This algorithm "grows" the minimum spanning tree from an initial vertex, adding one edge at a time, until the entire tree is built. The algorithm is based on property 5 above. In other words, at every step, Prim's algorithm is able to chose with certainly which edge must be included in the MST, and therefore it is selected and added to the current tree.

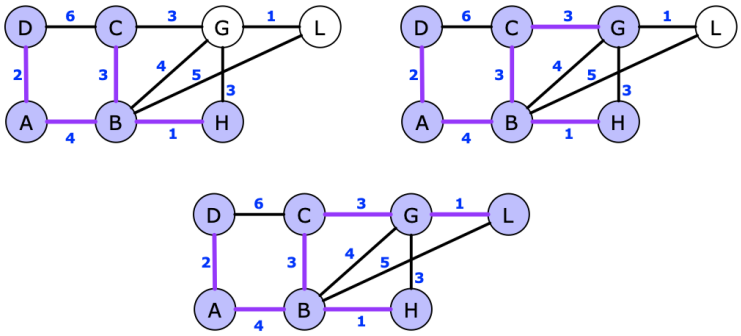
We begin with an example of how this algorithm works before diving into the details. The basic idea is that we keep track of how "far away" each vertex is from the current tree. The edge that is added next is always the edge that connects the next vertex that is "closest" to our tree. In the example below, Prims' algorithm begins at vertex A. The closest vertex that can be reached from A is vertex D. The tree now consists of two vertices as shown on the right.



The next vertex that is closest to the current tree (the current tree contains vertices A and D) is vertex B, at distance 4. Next, vertex H is now only distance 1 from the tree, and it is added next.



At this stage, both vertices C and G are at distance 3 from the tree. One of them is chosen next, suppose vertex C. From C, next vertex G is added, and finally vertex L which is only at distance 1 from the tree at the last step.



Prim's algorithm selects vertices based on their current distance to the tree. That distance *changes* as the tree grows. In the above example, note the vertex C is originally at distance 6 from the tree of vertices A and D. But later, as the tree grows and B is added, vertex C is only distance 3 from the tree. Therefore we need a data structure that can maintain these distances and update them efficiently.

2.1 The Priority Queue and distances

Each vertex requires a **distance attribute** which stores its current distance from the MST. This attribute is called $v.d$ for vertex v. Vertices are chosen one by one and added to the MST based on their distance attribute. We require a structure that enables us to quickly determine which vertex is the "closest" to the MST.

Prims algorithm uses a **Priority Queue**. This is not the same as the Queue we used to implement BFS. The Priority Queue that we use here is a **Min Heap**, with the additional operation *decrease-key*. The heap itself contains vertex *objects*, but the heap structure is built on the property $v.d$. Recall that we can delete the min from a Heap in time $O(\log n)$, and we also saw how to decrease the value of a node in the heap in time $O(\log n)$. We also saw previously in the course that we can build a heap on n items in $O(n)$ time. These operations together with the Heap structure are called a *Priority Queue*, Q. We will use the following pseudocode operations:

- Extract-min(Q): Extracts the vertex object with minimum $v.d$
- Decrease-key(Q, v, w): The attribute $v.d$ is set to w and the heap structure is updated.

We now move on to the details of Prims' algorithm.

2.2 The algorithm

Prim's algorithm begins building the MST from any vertex v of the graph. The starting vertex may affect the final tree that results. However, the resulting tree is guaranteed to have minimum total weight.

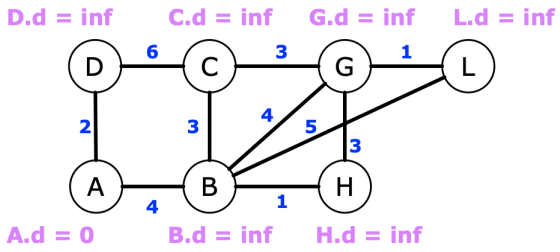
We start with the initialization step:

Prim(G,s):

Step 1: Initialize the variables

- Set all $v.d = \infty$ for each vertex v and set parent points to NIL for all vertices in the tree.
- Initialize an empty tree T.
- Set $s.d = 0$.
- Insert all vertices into the priority queue Q.

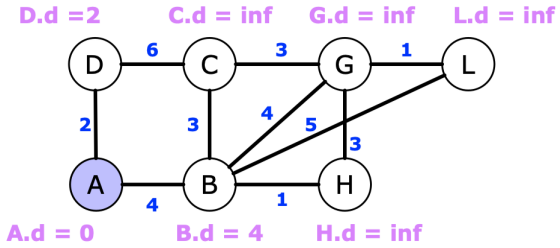
The graph below shows the result of the first step of Prims algorithm where $s = A$. We do not "draw" the priority queue, but instead write next to each vertex the value of $v.d$. Note that all vertices except A have $v.d = \infty$.



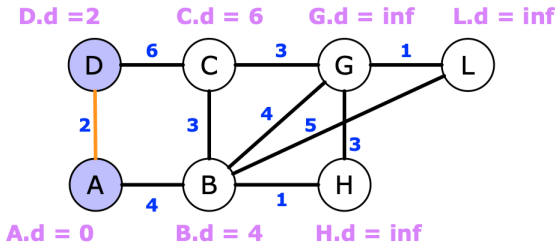
Step 2: Remove the minimum-distance item from the queue one at a time until the queue is empty. The vertex u with minimum distance is added to the MST. For any neighbor v of u, we check if its distance to u is less than the current value of $v.d$. This is equivalent to saying that vertex v is now "closer" to the MST.

```
while Q ≠ NIL
    u = Extract-min(Q)
    for each v in Adj [u]
        if v ∈ Q and v.d > weight(u, v)      *Update the distance to v from the tree
            Decrease-key(Q, v, w(u, v))
            v.parent = u                     *Set this node as the child of u
```

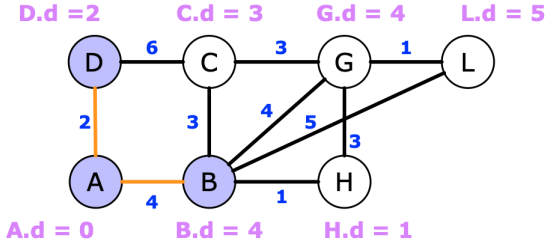
Let's look at the first iteration of this while loop. The node with the minimum distance is A because $A.d = 0$. Vertex A is deleted from Q (which can be interpreted as it now being included in the MST). The neighbors of A are updated to $D.d = 2$ and $B.d = 4$. At this stage, all white vertices are in the priority queue, and the purple vertices are in the tree.



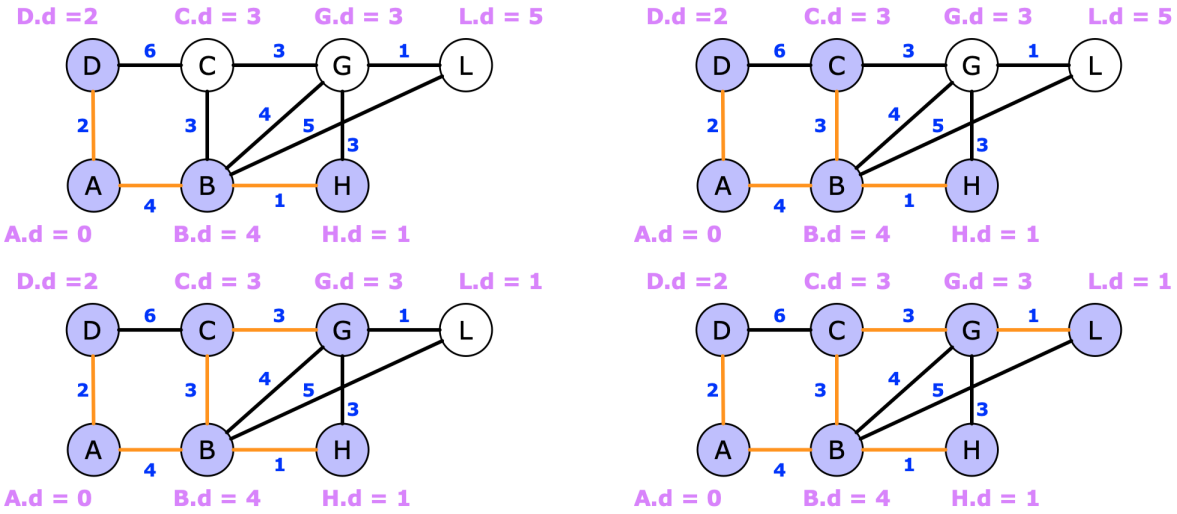
The next iteration of the while loop removes vertex D (since it has the minimum value of $D.d = 2$). The neighbor C is updated:



The next iteration removes vertex B (again, its value $B.d = 4$ is the minimum of all distances in the queue). The neighbors G, L, C and H all have their distances updated:



The next few iterations of the algorithm are shown below:



The last figure on the right represents the case where the queue is now empty. The while loop terminates.

One issue we didn't address above was how the edges were added to the MST. Note that the above pseudo-code sets **parent pointers** whenever the distance is updated in the priority queue. These parent pointers may be updated *several* times for a particular vertex v. Let's take vertex C for example. When vertex D was added to the MST, the value of $C.d$ was updated to 6 and its parent pointer was actually set to D. Later, when vertex B was added to the MST, the value of $C.d$ was updated again to 3, and its parent pointer was reset to B. When C was finally deleted from the queue and added to the MST, the parent pointer was B. Therefore at the moment when a vertex is deleted from Q, its parent pointer remains fixed. This corresponds to the edge in the MST.

Runtime:

Step 1 above takes $O(V)$ time, since a constant amount of work is done per vertex, and a heap of size V is built (taking time $O(V)$).

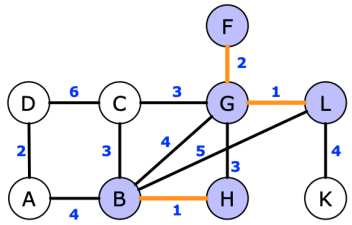
Step 2 carries out several operations. Let's focus first on the *Delete-min()* operations. Over the entire course of the algorithm, a vertex is removed from the queue exactly once. Recall that the priority queue carries out a delete in $O(\log n)$ time, and in this case $n = |V|$. Therefore each delete in the priority queue takes time $O(\log V)$. Since there are V vertices, this accounts for a runtime of $O(V \log V)$. Next, let's consider the *Decrease-key()* operations. The for loop that examines the adjacency list of each vertex examines each edge exactly twice during the entire algorithm - and each examination may carry out a Decrease-key() operation in (taking time $O(\log V)$). This accounts for a total runtime of $O(E \log V)$. Therefore the overall runtime of step 2 is therefore $O(E \log V + V \log V) = O(E \log V)$, and thus the runtime of Prim is $O(E \log V)$

3 Kruskal's algorithm

Kruskal's algorithm operates in a similar way, except that it greedily selects the edges to add to the MST in order of increasing edge size. The idea is quite simple: edges are added to the tree from smallest to largest. An edge is added *unless* the edge connects vertices that are already connected. Therefore the tree is not grown from a source vertex s, instead each vertex starts out as its own component, and components are slowly merged together as new edges are added.

Kruskal's algorithm doesn't use a priority queue, but instead requires a structure that allows the algorithm to keep track of the different components of the tree that have currently been established. For example, the figure below is a snapshot of an intermediary phase of Kruskal, where the MST is growing by *"components"*. One component consists of vertices G, F, L and the other consists of vertices B, H. As

Kruskal's algorithm continues, these components are slowly merged together until there is one resulting tree.

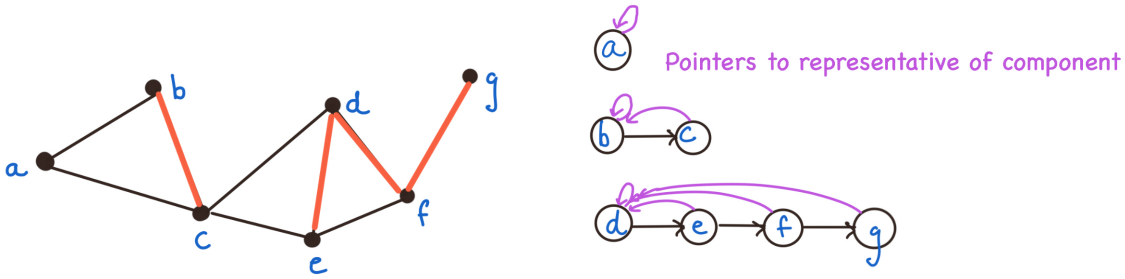


3.1 Union and Find

In this section we focus on how we can efficiently keep track of which vertices are part of which components during the execution of Kruskal. We require two main operations:

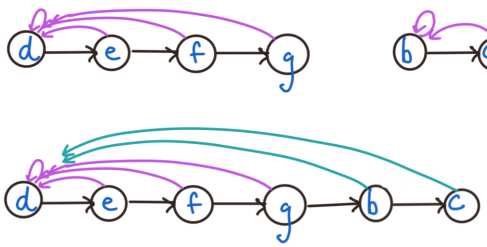
- A **Merge(u,v)** operation which efficiently takes the component containing vertex v and merges it with the component containing vertex v.
- A **Find(v)** operation, which returns the component containing vertex v. This is important, since we certainly don't need to concern ourselves with merging vertices that are already in the same component.

The solution is quite simple. We store each component as a linked-list, where the front of the list is used to identify the component. Each vertex in the list has a pointer (for example, called *mycaptain*) that points to the front of the list.



Find(u):
The Find(u) operation works by simply looking up the *mycaptain* attribute. Two vertices are in the same component if they have the same *mycaptain* variable. This takes $O(1)$ time.

Merge(u,v):
In order to merge two components together, we add the smaller list to the back of the bigger list. Then we reassign the *mycaptain* pointers for all vertices in the smaller list. The runtime of this depends on the number of vertices in the smaller list. Notice that when two lists merge, the size of the new list is at least *twice* the size of the smaller list. Therefore for each merge operation the smaller list at least doubles, and for a set of n elements, something can double at most $O(\log n)$ times. The figure below shows the result of merging a component of length 4 with one of length 2. Only two pointers are reassigned.



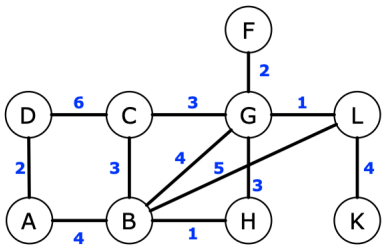
3.2 The Algorithm

Kruskal's algorithm simply sorts the edges of the graph and processes them one at a time. Processing the edges from smallest to largest, the algorithm checks if the edge can be safely added to the tree. If the edge is between two vertices that are *already* connected in the same component, then the edge is *not* added. Otherwise, the edge is added.

Kruskal's

Step 1: Initialize the variables and sort the edges

- Initialize an empty tree T.
- Sort the edges E of the graph from smallest to biggest.
- Each vertex v is placed in its own component.



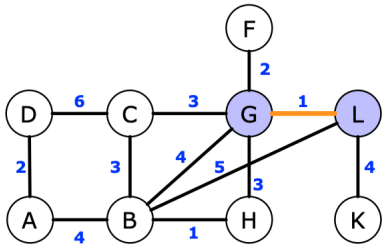
Sorted List:

1,1,2,2,3,3,3,4,4,4,5,6

Step 2: Go through the edges in sorted order. Whenever an edge connects two different components, *merge* those components together.

For each each $e = (u, v)$ in sorted order:
  If $Find(u) \neq Find(v)$
    Add edge (u, v) to T.
    $Merge(u, v)$

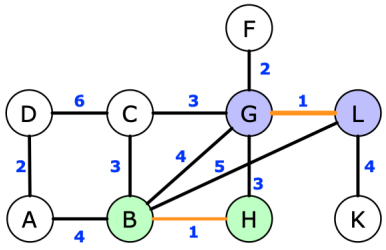
The first edge that is processed above has weight 1. Suppose this is the edge from G to L. These 2 vertices are merged into the same component. Note that the sorted list is now shorter:



Sorted List:

1,2,2,3,3,3,4,4,4,5,6

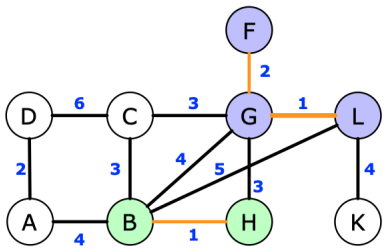
The next shortest edge is from B to H. These two vertices are merged into the same component. We use a different color to indicate that they are currently in different components.



Sorted List:

2,2,3,3,3,4,4,4,5,6

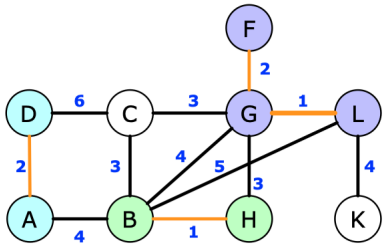
The next shortest edge has length 2. Suppose that edge is form G to F. Vertex F is merged into the component containing G and L:



Sorted List:

2,3,3,3,4,4,4,5,6

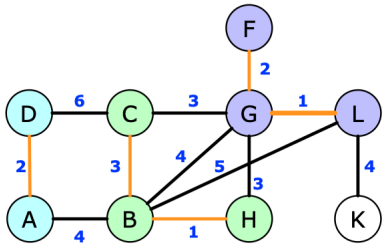
The next shortest edge has length 2, from vertex A to D:



Sorted List:

3,3,3,4,4,4,5,6

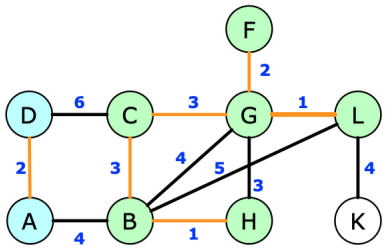
The next shortest edge has length 3, and suppose this edge is from C to B. Therefore vertex C gets merged with the component containing vertex B and H.



Sorted List:

3,3,4,4,4,5,6

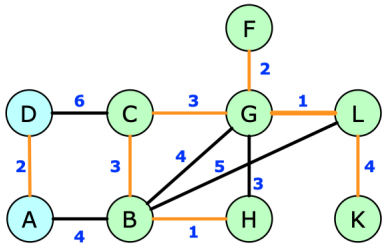
The next shortest edge has length 3, and suppose this edge is from C to G. Therefore the green and purple components get merged



Sorted List:

3,4,4,4,5,6

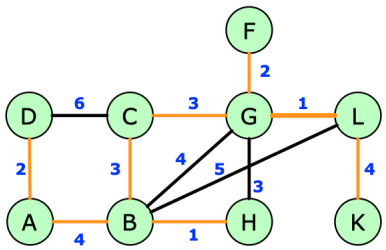
The next shortest edge has length 3. However, this edge is between two vertices that are already in the same component. Therefore we look next at the edge of length 4. Suppose this edge is between vertex K and L:



Sorted List:

4,4,5,6

And finally the next edge of length 4 merges the last two components. Although there are still edges left in the list after the merge is complete, they are emptied out (without any merges) and Kruskal's algorithm terminates.



Sorted List:

4,5,6

Runtime:

In step 1, we sort the edges. This takes time $O(E \log E)$, assuming we use Mergesort. In step 2, we carry out both *Find* and *Merge* operations. Each Find operation is constant, and we perform $O(E)$ finds, one for each edge in the tree. We showed above that the Merge operation is carried out at most $O(\log V)$ times for any one vertex, so over V vertices, the total of all the merges is $O(V \log V)$. The total runtime is then $O(V \log V + E \log E) = O(E \log E)$. Note however that $E \leq V^2$ and so $O(E \log E) = O(E \log(V^2)) = O(E \log V)$. The runtime is the same as that for Prim's.