Like Prim’s algorithm, Kruskal’s algorithm finds MSTs. The only difference between them is in their greedy choice. Prim’s starts from a single vertex and continually picks the lightest weight edge from the set of processed vertices to a vertex outside the set.

In contrast, Kruskal’s starts with each vertex as a connected component and repeatedly selects the lightest weight edge between two unconnected components.

Ex]
Kruskals \((G = (V, E), \omega)\)

Let \(T = (V, \emptyset)\)

Let \(P\) be a priority queue ordered by increasing edge weight

Put \(E\) into \(P\)

Let \(k = 0\) \((k = |V|)\)

While \(k < |V|-1\) \((\text{breaks when } T \text{ is a ST})\)

Poll edge \((u,v)\) from \(P\)

If \(\text{Find}(u) \neq \text{Find}(v)\)

Add \((u,v)\) to \(T\)

Union \((u,v)\)

\(k++\)

Return \(T\)

The runtime of this is certainly \(\Omega(|E| \log |E|) = \Omega(|E| \log |V|)\) in the WC since stuffing all of \(E\) into \(P\) means we may need to remove every edge from \(P\) to build \(T\) \((\text{this is a heap sort})\).

The upper bound on the WC runtime depends on how efficient \text{Find} and \text{Union} are. This is where the disjoint set data structure comes in.

The Problem: We want to know if \(u\) is connected to \(v\) (in the same component)

The idea: Each component has a representative vertex, so \(u\) and \(v\) are connected if they have the same representative.

\text{Initialize} \((V)\)

For each \(u \in V\)

\(u.\text{rep} = u\)

\text{Find} \((u)\)

If \(u.\text{rep} = u\)

Return \(u\)

Return \(\text{Find}(u.\text{rep})\)

Improvement!

Return \(u.\text{rep} = \text{Find}(u.\text{rep})\)

\text{Union} \((u,v)\)

\(u.\text{rep} = v\)

Improvement!

\text{Union} \((u,v)\)

\(\text{Find}(u).\text{rep} = \text{Find}(u)\)

If \(u, v\) are in the same component, this new operation is redundant.
The worst-case runtime of Union-Find is $O(n)$ when we call find at the bottom of a linked list, but this is uninteresting. Union-Find is inherently called many times, so we want to look at the amortized time. The best implementation of Union-Find leaves both algorithms with amortized time $O(\alpha(n))$.

What is $\alpha(n)$? $\alpha(n) = A^1(n,n)$.

Okay, but what is $A(n,n)$? It's the Ackermann function ($\alpha$ is the inverse Ackermann function).

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & n = 0, m > 0 \\ A(m-1,A(m,n-1)) & n, m > 0 \end{cases}$$

Which is...

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ 2^{m-2}(n+3) - 3 & m > 0 \end{cases}$$

What is that ↑? Knuth's up-arrow notation.
\[ 2 \uparrow 4 = 2 \times (2 \times (2 \times (2))) = 2^4 = 16 \]
\[ 2 \uparrow \uparrow 4 = 2 \uparrow (2 \uparrow (2 \uparrow (2))) = 2 \uparrow (2 \uparrow 2) = 2 \uparrow (2^2) = 2^{2^2} = 2^4 = 65536 \]
\[ 2 \uparrow \uparrow \uparrow 4 = 2 \uparrow \uparrow (2 \uparrow \uparrow (2 \uparrow \uparrow (2))) = \ldots = 2^{2^{^{^{\ldots}}}} = \text{very, very, big} \]
\[
\]
So \( A(n,n) \) gets huge really fast, so \( \alpha(n) \) grows at a crawl. \( \alpha(n) \) is less than 5 for any reasonable \( n \), where \( n \) is almost the number of atoms in the universe.

Proving that this is the runtime is beyond the scope of this course, but it is interesting to checkout.

Regardless, Union-Find is thus effectively constant time, so Kruskal's runs in \( O(E \log V) \).