Prim's Implementation - Continued.

Let's assume that there is an away
Entry [E] for each v ∈ V − S,
Entry [v] points to priority queue entry (v, δ(v), c(v, δ(v))) of v.
For v ∈ S, Entry [v] is null.

T ← ∅
Entry [s] ← null  // Take S = 1.

For each v ∈ Adj [s]
Set Entry [v] to (v, s, c(v, s))
and insert into priority queue.

For each v ∈ V \ {s},
if Entry [v] = null (priority queue entry for v has not been created),
set Entry [v] to (v, −, ∞) and
insert into priority queue.
For \( i \leftarrow 1 \) to \( n-1 \) do

Do Extract-min on priority queue.

Suppose this yields \((v, u, c(v,u))\).

For each \( w \in \text{Adj}[v] \)

if \( \text{Entry}[w] \neq \text{null} \)

Suppose \( \text{Entry}[w] \) points to

\((w, _, c)\). If \( c(v, w) < c \),

set \( \text{Entry}[w] \) to \((w, v, c(w,v))\)

and update priority queue.

\( \text{Entry}[v] \leftarrow \text{null} \)

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

---

Running time is \( O(nm) \) plus time
for \( n-1 \) insert, \( n-1 \) extract min,
and \( m \) (why?) update keys.
Since each operation on a priority queue can be performed in $O(\log n)$ time, running time is $O(m \log n)$.

(We assume $n = O(m)$, which holds because the input graph is connected.)

**Kruskal's implementation:**

We need a data structure that maintains connected components. Suppose the graph constructed by Kruskal so far is:

```
1 --o 2
  |   |
  7   4
  |   |
  o   0
```

This graph has 3 connected components:

$$\{1, 7, 5, 6\}, \{2, 4\}, \{3\}$$
Suppose edge examined is (5,6) at this point. It needs to know if 5 and 6 are in the same connected component. When it discovers they are, it doesn’t add edge (5,6). So connected components remain unchanged.

Suppose next edge examined is (5,2). Algorithm will add (5,2). The connected components change—the components containing 5 and 2 have to be merged:

\[
\{1, 7, 5, 6, 2, 4\} \quad \{3\}
\]

A Union-Find data structure is one that maintains disjoint sets under the union operation and that supports the find operation: given an element
return the set that it contains.

For the name of a set, we'll use one of its elements. The choice is arbitrary but should be consistent across all elements of the set.

Here are the operations supported by the Union-Find Structure:

Make Union Find (V) - Create a data structure with each element of V in a separate set.

Find (u) - Return name of set containing u

Union (A, B) - Merge the sets A and B into a single set.
With these primitives, here is Kruskal again. We can assume that an edge \( e = (u,v) \) is represented as a triple \((u,v, c(u,v))\).

Suppose \( E \subseteq e_1, e_2, \ldots, e_n \) in order of increasing costs.

Make Union Find \((V)\)

\[ T \leftarrow \emptyset \]

For \( i \leftarrow 1 \) to \( n \) do

Suppose \( E_i = (u, v, c(u,v)) \).

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

Add \( E_i \) to \( T \).

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

end if

end for

Running time is \( O(m \log m) \) plus time for \( 2m \) find operators, \( n-1 \) union operations (why?), and one Make Union find operation on \( V \).
How to implement the data structure, let's first consider a first, array-based implementation. There is an array $\text{set}[1..n]$. $\text{set}(u)$ identifies the set to which $u$ belongs.

In the example, before components are merged, the array would look like:


<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

After the two components are merged, the array looks like:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

For such a data structure, $\text{Union}$, $\text{MakeUnion}$, $\text{Find}(u)$ takes $O(n)$ time, $\text{Find}()$ takes $O(1)$ time.
The problem is that each union operation requires us to scan the entire array, and thus takes \( \Omega(n) \) time.

A better idea is to represent sets using linked lists. Corresponding to each \( v \in V \), there is an object which has a Set field, \( \text{Set}(v) \) identifies the set to which \( v \) belongs. This object should be accessible from \( v \).

In the example, just be hue the merge:

\[
\begin{array}{c|c}
\text{Name:} & 1 \\
\text{Size:} & 4 \\
\end{array} \quad \rightarrow \quad \begin{array}{c|c}
v & \text{Set}(v) \\
\end{array} \\
\begin{array}{c|c}
2 & 2 \\
\end{array} \quad \rightarrow \quad \begin{array}{c|c}
4 & 2 \\
\end{array} \quad \rightarrow \quad \begin{array}{c|c}
2 & 2 \\
\end{array} \\
\begin{array}{c|c}
3 & 1 \\
\end{array} \quad \rightarrow \quad \begin{array}{c|c}
3 & 3 \\
\end{array}
\]
After the merge:

\[
\begin{array}{c}
3 \\
1
\end{array} \rightarrow \begin{array}{c}
8 \\
3
\end{array}
\]

In a union operation, we merge the smaller set into the larger set. We can use the size field of the lists to determine the smaller set.

In this implementation,

Make Union-End(V) takes \(O(n)\) time.
Find() takes \(O(1)\) time.
Union(A, B) could still take \(\Omega(n)\) time in worst case. For
example when \(|A| = |B| = \frac{n}{2}\).

The key is to look at the combined running time of all \(n-1\) unions. (Why are there \(n-1\)?) Note that a single union operation takes time equal to the size of the smaller set (ignoring constant factors). Charge this running time to the elements of the smaller set, assigning a charge of 1 to each element.

Thus, running time for all the unions

\[
\sum_{v \in V} \text{Total Charge to } v
\]

\[
= \sum_{v \in V} \text{No. of }
\]

\(v \in V\)
Notice that each time v is changed, the size of the set in which v resides doubles (at least).

After v is changed for i-th time, the size of set in which v resides is at least \(2^i\).

Now, we must have

\[
\left(\text{No. of times } v \text{ is changed}\right) \leq n, 
\]

so No of times v is changed \(\leq \log_2 n\).

We conclude that running time for \(n-1\) unions = \(O(n \log_2 n)\).

We also conclude that our implementation of Kruskal takes

\[
O(m \log m + m + n + n \log n) = O(m \log n) \text{ time.}
\]