Union-Find: Improved Algorithms for Minimum Spanning Trees

Milda Zizyte
adapted from notes by Kathi Fisler

March 16, 2022

Objectives

By the end of this lecture, you will know:

- The disjoint-sets data structure
- The union-find operations on disjoint sets for improving MST construction

1 Revisiting Kruskal's Algorithm

For today's notes, we continue with the high-level descriptions from section 19.8 the Programming and Programming Languages textbook (written by Brown CS Professor Shriram Krishnamurthi), which you may use as a reference.

`https://papl.cs.brown.edu/2019/graphs.html`

1.1 Kruskal's Pseudocode

Let's examine what it may look like to write code for Kruskal's algorithm. Here, we use the notation (u, v) to mean the *edge between vertex u and vertex v*:

```
inputs: E set of edges, V set of vertices
sortedE = list(E).sort()
retTree = empty graph

while |retTree.E| < |V| - 1:
 (u, v) = sortedE.removeFirst()
 if not(retTree.hasPath(u, v):
   retTree.addEdge(u, v)

return retTree
```

We begin by sorting the set of edges E by weight. We then consider the smallest edge that we haven't considered thus far (by removing it from `sortedE`). We then check if introducing that edge would create a cycle. We do this for an edge (u, v) by asking if there *already* exists a path from u to v in the spanning tree we are building. If so, we would introduce a cycle by adding the edge (u, v) (the cycle from vertex u back to itself would consist of the existing path from u to v plus the edge from v to u). Otherwise, we adding this edge would not introduce a cycle, so we add it to the tree we are building.

The end condition of our loop is done when we have constructed a spanning tree. Mathematically, this is true when there are $|V| - 1$ edges in our graph, because every tree with $|V|$ vertices will have $|V| - 1$

edges (this can be proved mathematically by induction, which you will see if you take a course such as 22). We could have very well checked that `sortedE` was empty to end our loop, but this might result in unnecessary computation – if we've finished building the tree before we've exhausted all of the edges to check, why bother checking all of the edges? Similarly, we cannot use the condition $|retTree.V| == |V|$ to know if we've finished building our tree, because it might be that all of the vertices from the original graph are in the tree we are building, but they have not all been connected, so we are not done (consider adding edges (A, E), (A, B), and (C, D) to the spanning tree for the graph pictured below. All of the vertices will be in the spanning tree, but there is no way to get from A to C or D!). Thus, we use the stopping condition on edges.

1.2 Runtime of This Pseudocode

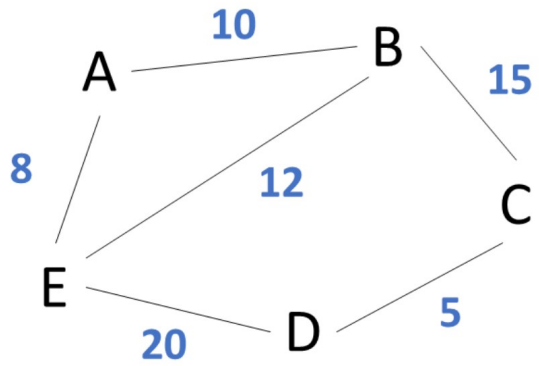
What is the runtime of this pseudocode? Let's take it piece by piece, omitting the operations that take constant time from our analysis:

- The line `sortedE = list(E).sort()` will take $O(|E|log|E|)$ time, because the best sorting algorithms run in $O(nlog(n))$ time and n here is the number of edges.
- The **`while`** loop will run $O(|E|)$ times in the worst case, if we end up having to examine all of the edges before we are done building the tree.
- `retTree.hasPath(u,v)` will take $O(|E| + |V|)$ time if we use DFS here, as discussed in the previous lecture

Taking into account that `retTree.hasPath(u,v)` runs every time we enter the body of the **`while`** loop, this means that the worst-case analysis of this code is $O(|E|log|E| + |E| * (|E| + |V|)) = O(|E|log|E| + |E|^2 + |E||V|)$. Because a connected graph with E edges has at most $|E| + 1$ vertices, $|E||V|$ will be at most $|E|^2 + |E|$. This means that the costly term here is $O(|E|^2)$. Can we do better?

2 Disjoint Sets

The idea of improving the runtime comes from speeding up the line of code that checks if adding the edge under consideration would introduce a cycle in the graph. We do this *as we run our code*, by storing some additional data. In particular, we store a notion of which vertices have formed *clusters*, that is, which vertices are already connected to each other. If the edge under consideration is between two vertices that are already in the same cluster, we know that there is already a path between those two vertices in the cluster, so adding the edge would introduce a cycle.



As a concrete example, consider running Kruskal's algorithm on the graph above. The list of edges, sorted by their weights, will be as follows (we use the $(u, v) : n$ notation to say that the edge between u and v has weight n):

```
(C, D):5, (A,E):8, (A,B):10, (B,E):12, (B,C):15, (D,E):20
```

At the beginning, each vertex is in its own cluster, because we have not added any edges to the spanning tree. Hence, our clusters look like:

```
{A}, {B}, {C}, {D}, {E}
```

The first edge we consider is (C, D). Because C and D are in different clusters, we know they are not connected and we are able to add the edge (C, D) to the spanning tree. We also update our knowledge of the clusters to say that C and D are now in the same cluster, because we connected them by adding the edge (C, D):

```
{A}, {B}, {C, D}, {E}
```

Next, we consider (A, E). Again, A and E are in different clusters, so we add the edge (A, E) to the spanning tree and update our knowledge of the clusters:

```
{A, E}, {B}, {C, D}
```

The edge with the next-lowest weight is (A, B). A is in the cluster $\{A, E\}$, while B is in the cluster $\{B\}$, which are distinct, so again we add the edge (A, B) to the spannign tree and update our clusters:

```
{A, B, E}, {C, D}
```

Now we consider the edge (B, E). The issue here is that B and E are already known to be in the same cluster, so adding the edge (B, E) to our spanning tree would create a cycle (indeed, looking back at the figure, the cycle would be made up of the existing edges (A, E) and (A, B) and the new edge (B, E). So, we do *not* add (B, E) to the spanning tree.

We next consider the edge (B, C). B and C are in distinct clusters, so it is safe to add this ege to the tree. But now, we have 4 edges in our spanning tree, when our original graph had 5 vertices, so because $4 = 5 - 1$, we know we are done with our algorithm. Our spanning tree includes the edges

```
(A, B), (A, E), (B, C), (C, D)
```

Thinking about the procedure, there are two key ideas here: checking to see if two vertices are in the same cluster (to see if we can add an edge), and keeping our knowledge of the clusters up-to-date (when we do add an edge and need to combine clusters). In computer science, these clusters are called *disjoint sets*, because no two clusters have an element in common (that is, they are all distinct). Checking to see if two vertices are in the same cluster is done using a disjoint set operation called *find* – if the two vertices are *found* to be in different clusters, we know that they are not connected and therefore adding the edge between them will not create a cycle. Combinding clusters is done using a disjoint set operation called *union* – when we add an edge that connects vertices in two different clusters, the clusters are now connected and should be *united* into one cluster.

2.1 Updated Kruskal's Pseudocode with Union/Find

What does that look like in our Kruskal's code? We keep track of all of the clusters and update them as necessary as we loop through the edges:

```
inputs: E set of edges, V set of vertices
sortedE = list(E).sort()
retTree = empty graph

for v in V:
```

```
  v.cluster = {v}

while |retTree.E| < |V| - 1:
 (u, v) = sortedE.removeFirst()
 if not(u.find() == v.find()):
   retTree.addEdge(u, v)
   union(u, v)

return retTree
```

The **for** loop with `v.cluster = {v}` means that we first initialize each vertex to be in a cluster by itself.

The line with `u.find == v.find` checks to see if u and v are in the same cluster. That is, if their clusters are equal, that means they are in the same cluster.

The line `union(u,v)` merges u's and v's clusters.

2.2 Keeping Track of Disjoint Sets

How are `union` and `find` implemented in practice? It seems inefficient to have each vertex associated with a set in memory (such as a `HashSet` in Java). Indeed, one approach is much simpler: give each cluster a "name," such as a unique numerical ID. We can match each vertex to the name of its cluster using a structure like a `HashMap`. `u.find()` would then return the value of the `HashMap` keyed on the specific vertex, and `union(u, v)` would make sure that every vertex in u's and v's cluster is updated to have the same name.

As an example, consider the sequence of adding edges to the graph above, and how they changed what our clusters looked like:

```
init:
{A}, {B}, {C}, {D}, {E}

add edge (C, D):
{A}, {B}, {C, D}, {E}

add edge (A, E):
{A, E}, {B}, {C, D}

add edge (A, B):
{A, B, E}, {C, D}

do not add edge (B, E)

add edge (B, C):
{A, B, C, D, E}
```

Using a `HashMap` to store names of clusters as numerical IDs instead, this computation will look like:

```
init:
A -> 1, B -> 2, C -> 3, D -> 4, E -> 5

add edge (C, D):
A -> 1, B -> 2, C -> 3, D -> 3, E -> 5

add edge (A, E):
A -> 5, B -> 2, C -> 3, D -> 3, E -> 5
```

```
add edge (A, B):
A -> 2, B -> 2, C -> 3, D -> 3, E -> 2

do not add edge (B, E)

add edge (B, C):
A -> 3, B -> 3, C -> 3, D -> 3, E -> 3
```

Notice that the computation that made us decide that (B, E) should not be added to the spanning tree was that $B.find()$ would return 2 and $E.find()$ would also return 2, which means they are both in the same cluster.

Also notice that whenever we add an edge (such as (C, D)), we update all of the vertices in that cluster to have the same name. In this case, we chose to give C's cluster name (3) to D, but we could have very well done it the other way around. Also notice that when we added edge (A, B), both A and E took on B's cluster name, because A and B were both in the same cluster. This is one approach to the `union` operation – when merging two clusters, update the name of every item in one of the clusters to be the name of the other cluster.

What are the implications on runtime? This means that `union` still has to loop through all of the vertices, check to see if they were in the old cluster, and update their cluster names. This means that the runtime every time we enter the **`while`** loop of Kruskal's algorithm is now $O(|V|)$ rather than the $O(|V| + |E|)$ from DFS, which means we gained some improvement for *dense* graphs (graphs with a lot of edges). Can we do even better?

2.3 A Faster Union

It turns out that we can speed up `union` by deferring the computation of updating names to `find`. We do this by using *vertices*, instead of numerical IDs, for the names of clusters.

To `find` the name of the cluster of u, we traverse the data structure holding all of the vertices' clusters' names (for example, the `HashMap` that goes from a vertex to its cluster name) until we get to a vertex whose cluster name is itself. We then update each of the vertices we traversed to to have that cluster name. If we do this using a recursive function, we can actually get this done in very few lines, as we will see soon.

To `union` the two clusters of vertices u and v, we `find` the vertex that names u's cluster (sometimes called u's *parent*) and change that vertice's cluster to v (or the other way around – we will see a way to track the optimal choice in a few paragraphs). This becomes a constant-time operation.

To rewrite our example yet again, this looks like:

```
init:
  A -> A, B -> B, C -> C, D -> D, E -> E

check edge (C, D):
  after C.find():
  A -> A, B -> B, C -> C, D -> D, E -> E (C.find() = C)
  after D.find():
  A -> A, B -> B, C -> C, D -> D, E -> E (D.find() = D)
  after union(C, D):
  A -> A, B -> B, C -> D, D -> D, E -> E (C's parent is C, change C's parent to D)

check edge (A, E):
  after A.find():
  A -> A, B -> B, C -> D, D -> D, E -> E (A.find() = A)
  after E.find():
  A -> A, B -> B, C -> D, D -> D, E -> E (E.find() = E)
```

```
  after union(A, E):
  A -> E, B -> B, C -> D, D -> D, E -> E (A's parent is A, change A's parent to E)

check edge (A, B):
  after A.find():
  A -> E, B -> B, C -> D, D -> D, E -> E (A.find() = E)
  after B.find():
  A -> E, B -> B, C -> D, D -> D, E -> E (B.find() = B)
  after union(A,B):
  A -> E, B -> A, C -> D, D -> D, E -> E (B's parent is B, change B's parent to A)

check edge (B, E):
  after B.find():
  A -> E, B -> E, C -> D, D -> D, E -> E (B.find() = E)
  after E.find():
  A -> E, B -> E, C -> D, D -> D, E -> E (E.find() = E)
  do not add edge (B, E)

check edge (B, C):
  after B.find():
  A -> E, B -> E, C -> D, D -> D, E -> E (B.find() = E)
  after C.find():
  A -> E, B -> B, C -> D, D -> D, E -> E (C.find() = D)
  after union(B,C):
  A -> E, B -> A, C -> D, D -> E, E -> E (C's parent is D, change D's parent to B)
```

Pay special attention to how `B.find()` changed the mapping when we checked the edge (B, E). We first found that B's parent is A, and then we found that A's parent is E, and then we found that E's parent is E. We traversed back through E, A, and B, and updated B's parent to E, all while doing the `find` operation. This is what we mean by *deferring* the update to `find`.

Also notice that, in the very last step (`union(B, C)`), we found C's parent, which was D, and changed D's parent to B, not C's. Had we changed C's parent to B instead, we would have a situation where we no longer maintain D's connection to C (that is, `D.find()` would return D but `C.find()` would return A, even though they should be in the same cluster). This is why we need to find a vertex's parent when doing the `union` operation.

2.4 Even More Optimized Union/Find Pseudocode and Example

In the above example, we were doing the `union` operation by changing u's parent's parent to v or v's parent's parent to u, arbitrarily. It turns out that we can be more systematic about this decision – if we keep track of the sizes of the clusters that u and v belong to, we can choose to change what the *smaller* cluster's parent points to. Effectively, this means that the larger cluster subsumes the smaller cluster, and not the other way around. We won't go into the math here, but it works out that this results in a shorter runtime for the `find` operations, on average.

Assuming we use a data structure called `clusterMap` to keep track of the the parents that identify the clusters of the vertices and a `sizeMap` to keep track of the cluster sizes, the definitions above lead to the following pseudocode for `union`:

```
union(u, v):
  u_par = u.find()
  v_par = v.find() (consider u's and v's parents instead)
```

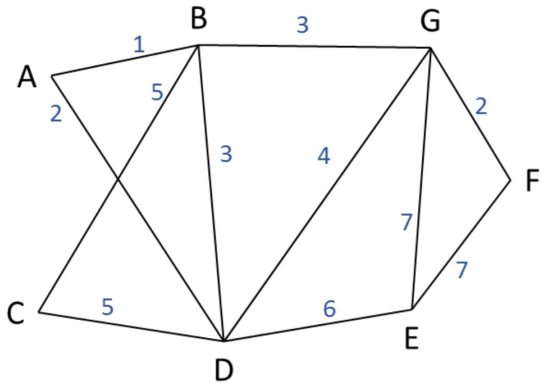
```
  if sizeMap.get(u_par) >= sizeMap.get(v_par):
    clusterMap.put(v_par, u_par)
    sizeMap.put(u_par, sizeMap.get(u_par) + sizeMap.get(v_par))
  else:
    clusterMap.put(u_par, v_par)
    sizeMap.put(v_par, sizeMap.get(u_par) + sizeMap.get(v_par))
```

Notice how we update the `sizeMap` here. If *u_par* is added as *v_par*'s parent, `v_par.find()` will produce *u_par*, so we only need to update *u_par*'s size, and vice versa.

The following pseudocode is for `find`:

```
u.find():
  parent = clusterMap.get(u)
  if parent == u:
    return u
  else:
    new_parent = parent.find()
    clusterMap.put(u, newParent)
```

Let's run through a bigger example to verify that the `union` and `find` don't look so bad:




```
initial clusterMap:
  A -> A, B -> B, C -> C, D -> D, E -> E, F -> F, G -> G

initial sizeMap:
  A -> 1, B -> 1, C -> 1, D -> 1, E -> 1, F -> 1, G -> 1

check edge (A, B):
  after A.find():
  A -> A, B -> B, C -> C, D -> D, E -> E, F -> F, G -> G (A.find() = A)
  after B.find():
  A -> A, B -> B, C -> C, D -> D, E -> E, F -> F, G -> G (B.find() = B)
  after union(A, B):
  A -> A, B -> A, C -> C, D -> D, E -> E, F -> F, G -> G
  updated sizeMap:
  A -> 2, B -> 1, C -> 1, D -> 1, E -> 1, F -> 1, G -> 1

check edge (F, G):
  after F.find():
```

```
  A -> A, B -> A, C -> C, D -> D, E -> E, F -> F, G -> G (F.find() = F)
  after G.find():
  A -> A, B -> A, C -> C, D -> D, E -> E, F -> F, G -> G (G.find() = G)
  after union(F, G):
  A -> A, B -> A, C -> C, D -> D, E -> E, F -> F, G -> F
  updated sizeMap:
  A -> 2, B -> 1, C -> 1, D -> 1, E -> 1, F -> 2, G -> 1

check edge (A, D):
  after A.find():
  A -> A, B -> A, C -> C, D -> D, E -> E, F -> F, G -> F (A.find() = A)
  after D.find():
  A -> A, B -> A, C -> C, D -> D, E -> E, F -> F, G -> F (D.find() = D)
  after union(A, D):
  A -> A, B -> A, C -> C, D -> A, E -> E, F -> F, G -> F
  updated sizeMap:
  A -> 3, B -> 1, C -> 1, D -> 1, E -> 1, F -> 2, G -> 1

check edge (B, G):
  after B.find():
  A -> A, B -> A, C -> C, D -> A, E -> E, F -> F, G -> F (B.find() = A)
  after G.find():
  A -> A, B -> A, C -> C, D -> A, E -> E, F -> F, G -> F (G.find() = F)
  after union(B, G):
  A -> A, B -> A, C -> C, D -> A, E -> E, F -> A, G -> F
  updated sizeMap:
  A -> 5, B -> 1, C -> 1, D -> 1, E -> 1, F -> 2, G -> 1

check edge (B, D):
  after B.find():
  A -> A, B -> A, C -> C, D -> A, E -> E, F -> A, G -> F (B.find() = A)
  after D.find():
  A -> A, B -> A, C -> C, D -> A, E -> E, F -> A, G -> F (D.find() = A)
  do not add edge (B, D)

check edge (D, G):
  after D.find():
  A -> A, B -> A, C -> C, D -> A, E -> E, F -> A, G -> G (D.find() = A)
  after G.find():
  A -> A, B -> A, C -> C, D -> A, E -> E, F -> A, G -> A (G.find() = A)
  do not add edge (D, G)

check edge (B, C):
  after B.find():
  A -> A, B -> A, C -> C, D -> A, E -> E, F -> A, G -> A (B.find() = A)
  after C.find():
  A -> A, B -> A, C -> C, D -> A, E -> E, F -> A, G -> A (C.find() = C)
  after union(B, C):
  A -> A, B -> A, C -> A, B -> A, E -> E, F -> A, G -> A
  updated sizeMap:
  A -> 6, B -> 1, C -> 1, D -> 1, E -> 1, F -> 2, G -> 1
```

```
check edge (C, D):
  after C.find():
  A -> A, B -> A, C -> A, D -> A, E -> E, F -> A, G -> A (C.find() = A)
  after D.find():
  A -> A, B -> A, C -> A, D -> A, E -> E, F -> A, G -> A (D.find() = A)
  do not add edge (C, D)

check edge (D, E):
  after D.find():
  A -> A, B -> A, C -> A, D -> A, E -> E, F -> A, G -> A (D.find() = A)
  after E.find():
  A -> A, B -> A, C -> A, D -> A, E -> E, F -> A, G -> A (E.find() = E)
  after union(D, E):
  A -> A, B -> A, C -> A, D -> A, E -> A, F -> A, G -> A
  updated sizeMap:
  A -> 7, B -> 1, C -> 1, D -> 1, E -> 1, F -> 2, G -> 1

6 edges added; done
```

As you might notice in this example, with this version of union/find, the number of recursive calls to get to the parent drops in subsequent calls to `find` on the same vertex. Doing this optimization (called *path-compression* along with our optimization in `union` that merges the smaller set into the larger one, yields an amortized running time for `find` that is nearly constant ("nearly" because it isn't constant, but the growth is so small that it is effectively constant). The textbook chapter we referenced at the start of the notes provides a link to the proof, whose math is quite involved.