Union-Find: Improved Algorithms for Minimum Spanning Trees

Kathi Fisler and Milda Zizyte

October 28, 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 = E as a sorted list
retTree = empty graph

while |retTree.E| < |V| - 1:
 (u, v) = sortedE.removeFirst()
 if !retTree.hasPath(v, u)):
   retTree.addEdge(u, v)

return retTree
```

In this pseudocode, 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 v to u 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 the rest 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

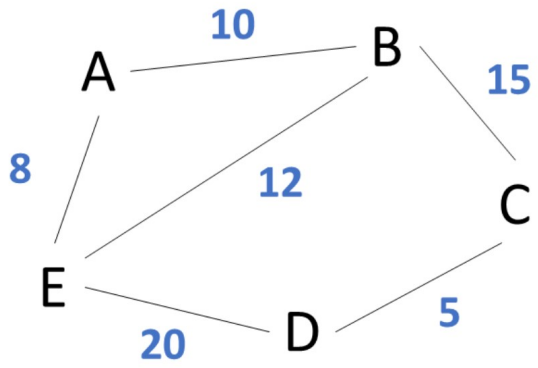
What is the runtime of this pseudocode? Let's take it piece by piece, omitting the operations that take constant time from our analysis:

- 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 or BFS, as discussed in the previous notes.

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 $O(|E|log|E|+|E|*(|E|+|V|)) = O(|E|log|E|+|E|^2+|E||V|)$. Simplifying this to take the biggest exponent into consideration leads us to $O(|E|^2 + |E||V|)$. This means that the costly term of this runtime 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 *connected groups*, 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 connected group, we know that there is already a path between those two vertices in the connected group, 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 a notation of $(u, v) : n$ 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 connected group, because we have not added any edges to the spanning tree. Hence, our connected groups look like:

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

The first edge we consider is (C, D). Because C and D are in different connected groups, 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 connected groups to say that C and D are now in the same connected group, 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 connected groups, so we add the edge (A, E) to the spanning tree and update our knowledge of the connected groups:

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

The edge with the next-lowest weight is (A, B). A is in the connected group $\{A, E\}$, while B is in the connected group $\{B\}$. Again, we add the edge (A, B) to the spanning tree and update our connected groups:

```
{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 connected group (i.e. a path already exists to get between B and E), 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 different connected groups, 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 connected group (to see if we can add an edge), and keeping our knowledge of the connected groups up-to-date (when we do add an edge and need to combine connected groups). In computer science, these connected groups are called *disjoint sets*, because no two connected groups have an element in common.

Checking to see if two vertices are in the same connected group is done using a disjoint set operation called *find* – if the two vertices are *found* to be in different connected groups, we know that they are not connected and therefore adding the edge between them will not create a cycle.

Combining connected groups is done using a disjoint set operation called *union* – when we add an edge that connects vertices in two different connected groups, the connected groups are now connected and should be *united* into one connected group.

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 connected groups and update them as necessary as we loop through the edges:

```
initialize each vertex to be in its own connected group

while |retTree.E| < |V| - 1:
  (u, v) = sortedE.removeFirst()
  if find(u) != find(v):
    retTree.addEdge(u, v)
    union(u, v)
```

The line with `find(u) != find(v)` checks to see if u and v are in the same connected group. That is, if their connected groups are equal, that means they are in the same connected group.

The line `union(u,v)` merges u's and v's connected groups when we have added an edge connecting them.

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 connected group a "name," such as a unique numerical ID. We can match each vertex to the name of its connected group using a structure like a `HashMap`. `find(u)` 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 connected group is updated to have the same name. The simplest way to do this is to update every vertex in v to have u's name.

As an example, consider the sequence of adding edges to the graph above, and how they changed what our connected groups 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 connected groups 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 -> 1, B -> 2, C -> 3, D -> 3, E -> 1

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

do not add edge (B, E)

add edge (B, C):
A -> 1, B -> 1, C -> 1, D -> 1, E -> 1
```

Notice that the computation that made us decide that (B, E) should not be added to the spanning tree was that $find(B)$ would return 1 and $find(E)$ would also return 1, which means they are both in the same connected group.

Also notice that whenever we add an edge (such as (C, D), we update all of the vertices in that connected group to have the same name. In this case, we chose to give C's connected group name (3) to D, but we could have very well done it the other way around.

What are the implications on runtime? Take a look at how we added the edge (B, C) – `union` still has to loop through all of the vertices, check to see if they were in the old connected group, and update their connected group names. This means that the runtime every time we enter the **`while`** loop of Kruskal's algorithm is now $O(|V|)$ rather than the $O(|V| + |E|)$ we got from DFS or BFS. We gained some improvement for *dense* graphs (graphs with a lot of edges), but can we do even better?

2.3 A Faster Union

It turns out that we can speed up `union` by using *vertices*, instead of numerical IDs, for the names of connected groups, and deferring when we update the names of vertices' groups. The *key idea* is that, instead of having the `HashMap` associate each vertex in a connected group with its group name, we want to have a *chain* of vertices that we can follow to get back to the group name.

Here is our *initial* pseudocode for `find` and `union`, where `groupMap` is the `HashMap` from a vertex to the name of its group

```
find(u):
  if (groupMap.get(u)) == u:
     return u
  else:
      return find(groupMap.get(u))

union(u, v):
   groupMap.put(find(u), find(v))
```

How does this get us an optimization? Consider the following initial `groupMap` of some set of vertices $M - P$:

```
M -> M, N -> N, O -> O, P -> P
```

Consider running `union(M, N)`. `find` goes through the `groupMap` until it finds a vertex whose name is itself. `find(M)` and `find(N)` will return M and N, respectively, so calling `union(M, N)` would update the `groupMap` to be:

```
  M -> N, N -> N, O -> O, P -> P
```

Now consider running `union(N, P)`. `find(N)` will return N and `find(P)` will return P, so calling `union(N, P)` would update the `groupMap` to become:

```
  M -> N, N -> P, O -> O, P -> P
```

Note that, even though M and N were in the same connected group, we did not have to update M in the `groupMap` – but M will still see the update of being in a connected group with N and P by following the chain of group names! **Verify for yourself** that calling `find` on M, N and P will all return P, meaning that they are all in the same connected group.

Why do we have to call `find` on u and v every time we run union? In the words of Fleetwood Mac, this is so that we never break the chain. Consider calling `union(M, O)` without first calling `find` on M, i.e. by simply running `groupMap.put(M, find(O))` (or even `groupMap.put(M, O)`). Then, **incorrectly**, `groupMap` would be:

```
  M -> O, N -> P, O -> O, P -> P
```

Even though all of M, N, O, and P should now be in the same connected group, if we call `find` on M and O, we get back O, and if we call `find` on N or P, we get back P. So, we need to `find` the end of the chain when we want to create the union of the connected groups.

2.3.1 Path compression/Final `find` pseudocode

The code above still has an inefficiency that we can fix – calling `find` means we might have to traverse a whole chain of vertices until we reach the connected group's name. `union` also calls `find`, which means that both operations have this ineffiency. But take a look at the recursive nature of `find` – what if, when we traverse the chain of vertices in the recursion, we update the `groupMap` after each recursive call, such that each vertex in the chain now maps to the new answer? This wouldn't create additional runtime for the find operation, since updating a `HashMap` is constant time, and could benefit *future* calls to `union` and `find`.

We can do this by updating our `find` code with a few lines (here we use *parent* as a variable that names the vertex that u maps to in `groupMap`).

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

To rewrite our original example yet again, the algorithm 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

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

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 -> B, C -> D, D -> D, E -> B

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

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

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

Notice that, for this example, there was no path compression, since every `find` operation ended after at most one recursive call. The example at the end of these notes does feature path compression.

Verify for yourself that, after the last step, `X.find()` should return D for every vertex X.

2.4 Even More Optimized Union/Final `union` Pseudocode and Example

In the above example, we were doing the `union(u,v)` operation by changing u's parent to v's parent. We could have also changed v's parent's to u's parent, arbitrarily. It turns out that we can be more systematic about this decision – if we keep track of the sizes of the connected groups that u and v belong to, we can choose to change what the *smaller* connected group's parent points to. Effectively, this means that the larger connected group subsumes the smaller connected group, and not the other way around. We won't go into the math here (it's fairly sophisticated and beyond the scope of what you need to know for 200), but it works out that this results in a shorter runtime for the `find` operations, on average.

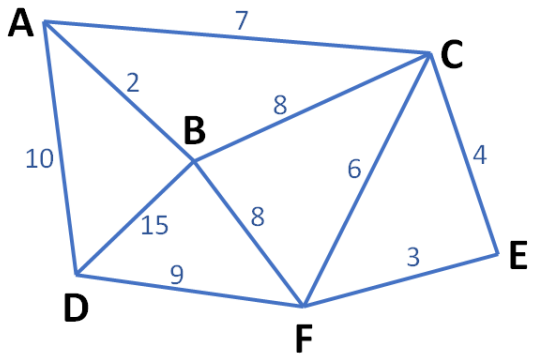
Assuming we use a `sizeMap` to keep track of the connected group sizes, the definitions above lead to the following pseudocode for `union`:

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

  if sizeMap.get(u_par) >= sizeMap.get(v_par):
    groupMap.put(v_par, u_par)
    sizeMap.put(u_par, sizeMap.get(u_par) + sizeMap.get(v_par))
    sizeMap.remove(v_par)
  else:
    groupMap.put(u_par, v_par)
    sizeMap.put(v_par, sizeMap.get(u_par) + sizeMap.get(v_par))
    sizeMap.remove(u_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. At this point, we can remove the other vertice's size, since we have merged the two connected groups. This is an optional step that doesn't affect the runtime, but makes the below example a little more readable by showing that one connected group was subsumed by the other.

Let's run through a bigger example to verify that the `union` and `find` don't take very long at all:



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

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

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

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

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

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

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

```
  A -> E, B -> A, C -> E, D -> D, E -> E, F -> E
  updated sizeMap:
  D -> 1, E -> 5

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

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

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

5 edges added; done
```

Take a look at how `find` updated B's parent when it checked the edge (B, C) – while the `find` itself took three steps (B maps to A, A maps to E, E maps to E), it then updated B's parent to E. On the subsequent check of edge (B, F), finding B's parent only took two steps, because we had done the path compression on the previous step! 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 (to take just two steps in the vast majority of cases). All of the optimization steps combined yield an amortized (average across all operations) 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.