CMSC 420: Disjoint-Set Data Structures

Justin Wyss-Gallifent

April 22, 2024

1 Introduction

Disjoint-set data structures, also called union-find data structures, are a class of data structure which stores elements according to disjoint sets they are in in a way which allows us to do things such as:

- Discover if two elements are in the same disjoint set.
- Merge two disjoint sets to form a new one.

Disjoint-set data structures are heavily used in Kruskal's Algorithm for finding a minimal spanning tree, specifically for checking if there is a cycle in the graph.

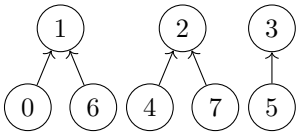
2 Implementation

The most common way to implement a disjoint-set data structure is a forest, or a collection of trees. We'll call this a *disjoint set forest* or just a forest for short. Each tree will correspond to a subset and each node within that tree to an element. The trees will be a bit non-traditional in that each node will only contain a pointer to its parent. For each subset, one node will be chosen as the root. This is a *representative* of the subset. The root nodes have no parent of course and it's traditional to have their parent pointers point to themselves.

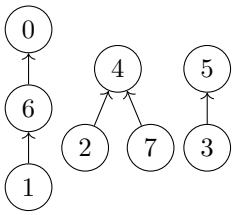
There is no real limit to the number of children a node could have since we aren't storing child data.

Example 2.1. Consider the set of elements $\{0, 1, 2, 3, 4, 5, 6, 7\}$ divided into three disjoint subsets $\{0, 1, 6\}$, $\{2, 4, 7\}$, and $\{3, 5\}$.

We'll store this by creating three trees, one for each subset. Here they are. In these pictures I've not drawn the parent pointers (pointing to themselves) out of habit:



In such case we chose the representatives 1, 2, and 3 but this was arbitrary. In fact this forest would work, too:



In addition the structure of these trees can be stored in a simple list A indexed with the number of elements whereby $A[i]$ is equal to the parent index. In such

a case for a root r we assign $A[r] = r$.

Example 2.2. The second picture in the above example would be stored easily as:

$$A = [0, 6, 4, 5, 4, 5, 0, 7]$$

This is because:

- $A[0] = 0$ since 0 is a root.
- $A[1] = 6$ since $1 \to 6$.
- $A[2] = 4$ since $2 \to 4$.
- And so on...

Observe that $A[0] = 0$ and $A[4] = 4$ and $A[5] = 5$ as they are the roots.

Before proceeding, we have:

Definition 2.0.1. For an element x in the set, define the *root* of x, denoted $root(x)$, to be the root of the tree containing x.

In this sense the representative of a subset can be found by taking $root(x)$ for any x in the subset.

3 Three Basic Operations Version 1

3.1 Introduction

There are three basic operations we can easily perform with our disjoint subset data structure.

3.2 Creating a New Subset with a New Element

If a new element is introduced we simply create a node (or list entry) which points to itself.

3.3 Finding Subset Representatives

To find the representative for a subset means to find the root. The following pseudocode will do this. This pseudocode is premised on the fact that the root's parent is itself.

The pseudocode would look like:

```
function findrep(x)
  if x.parent == x
    return(x)
  else
    return(findrep(x.parent))
```

```
    end if
  end function
```

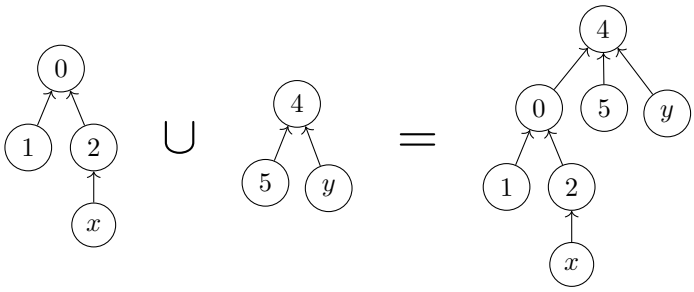
Observe that finding subset representatives allows us to easily see if two elements are in the same subset. Given two elements x and y we can check if they're in the same set by checking if `findrep(x) == findrep(y)`.

Note 3.3.1. You might wonder why we didn't just call this function `root`, since we're finding the root. The answer is that in a bit we'll tweak it so that it adjusts the tree and we'd like to keep the $root(x)$ so that it simply returns the root.

3.4 Union of Subsets

Given two subsets it's easy to merge them. Suppose x and y are elements and we wish to merge the subsets which contain them. We check $root(x)$ and $root(y)$. If they're the same then there's nothing to do. If they're not the same then we simply set the parent of $root(x)$ (which was originially $root(x)$) to be $root(y)$.

Example 3.1. For example:



The pseudocode would look like:

```
function union(x,y)
  if findrep(x) != findrep(y)
    findrep(x).parent = findrep(y)
  end if
end function
```

3.5 Time Complexity

The major issue with the above operations is that it's possible for the trees representing the subsets to get very unbalanced. For example if we have n elements total we might end up with one subset with n elements for which the tree is a list of length n, or we might end up with two subsets with $n/2$ elements for which the trees are lists of length $n/2$.

In such cases the second two operations run with time $\mathcal{O}(n)$ which is less than ideal.

4 Three Basic Operations Version 2

4.1 Introduction

We'd like to speed up our operations!

We might suggest some ideas like - keeping the trees balanced, but that takes time itself. Can we do something else?

The answer is yes and it's anchored in the fact that our trees have two interesting properties. First, there is no child limit, and second, we have parent pointers. This allows us to modify our second two operations so that they keep the trees "short".

4.2 Finding Subset Representatives Revisited

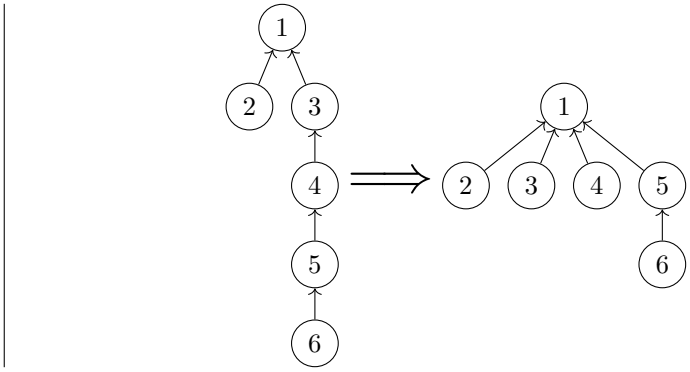
When we are finding a subset representative of the subset containing the element x we follow the graph from x to $root(x)$. While we're doing this we can actually easily modify all the nodes along the root so that their parents are $root(x)$.

Here is the modified pseucode. Note that it runs just as fast as it did before.

```
function findrep(x)
  if x.parent == x
    return(x)
  else
    return(x.parent = findrep(x.parent))
  end if
end function
```

Here's an example to see what it does:

Example 4.1. This example demonstrates what our updated `findrep(5)` will do:



It turns out that if we do this every time we look for the representative we keep the tree short enough to reduce the time complexity significantly. We call this *path compression*.

4.3 Union of Subsets Revisited

Taking the union of subsets is what can lead to trees getting rather tall so perhaps there's a way to carefully join the trees so that this doesn't happen. There are, and we'll look at one of them.

Given two elements x and y When we took the union earlier we simply set $root(x) = y$. However suppose the tree containing x has height h_x and the tree containing y has height h_y. If we set $root(x) = y$ then the new tree has height $h_x + 1$ whereas if we set $root(y) = x$ then the new tree has height $h_y + 1$. We could of course record the heights of the trees and choose the shorter option but keeping track of tree heights takes time and care, especially given the fact that we're repeatedly messing with them in this case.

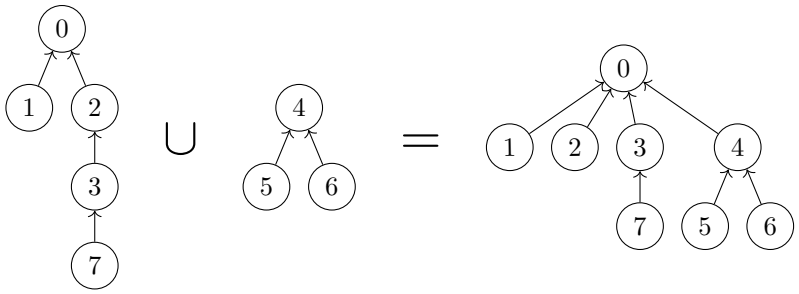
Instead a reasonable proxy for tree height is the number of elements. A tree with more elements tends to be higher. Moreover keeping track of element counts is easy. When we create a tree with one element we store its size as 1 and when we merge two trees we add their sizes.

So what we will do is pick the tree with fewer elements and attach the root of that tree to the root of the tree with more elements. We call this the *weighted union*. It turns out that this small change has a massive impact.

Here is the modified pseucode. Note that it runs just as fast as it did before. Also note that when are finding the root in order to complete the union we use path compression.

```
function union(x,y)
  if findrep(x) != findrep(y)
    if x.size >= size.y
      findrep(y).parent = findrep(x)
    else
      findrep(x).parent = findrep(y)
  end if
end function
```

Example 4.2. For example here is the result of `union(3,5)`. Notice that `findrep(3)` performs path compression:



4.4 Time Complexity Revisted

These small changes punch above their weight. If we implement them then amortized analysis shows that any series of our three operations runs in $\mathcal{O}(\alpha(n))$ amortized time, where α is the inverse of the Ackermann function which is "essentially constant". The inverse of the Ackermann function is an increasing function which is less than 4 for all n less than approximately 10^{600}.

This means that our set operations essentially run in constant amortized time!

The proof of this amortized time complexity is not trivial. If you are interested you can find it in the classic CLRS (Cormen, Leiserson, Rivest, Stein) algorithms testbook.

5 Simple Graph Management

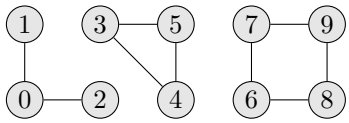
5.1 Introduction

Here are some tools for managing simple graphs. In what follows we'll assume that we have some fixed number n of vertices and the only thing that changes is the edges.

5.2 Representing Connected Components

Given a (not necessarily connected) graph with n vertices suppose we partition the set of vertices into subsets according to which connected component. These subsets will be disjoint and their union will be the set of all vertices.

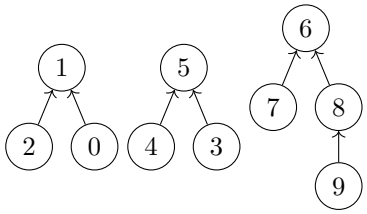
Example 5.1. Consider the following graph:



The disjoint sets of vertices are:

$$\{0, 1, 2\}, \{3, 4, 5\}, \{6, 7, 8, 9\}$$

These can be represented by the forest:



5.3 Edge Information and Addition

First observe that the disjoint set data structure can only tell us if two vertices u and v are in the same connected component, it cannot tell us if an edge exists between them.

Consequently the impact of adding an edge between vertices u and v would simply be to ensure that the two components are connected. This simply means taking the union of the two components with `union(x,y)`. The result will be possibly some path compression as well as a union of the components if they are disjoint.

5.4 Cycle Detection

The disjoint set data structure cannot detect whether a cycle exists in the graph because it only stores information about connected components.

5.5 Does Adding an Edge Form a Cycle?

All hope is not lost, however. For two vertices u and v, suppose we know the following two things:

- There is no edge between u and v
- There are no cycles in the graph.

Then we can detect if adding an edge (u, v) will create a cycle. We do this simply by check if $root(u) = root(v)$, or in code if `findrep(u)==findrep(v)`. If this is true then u and v are in the same component and since they are not already connected by an edge then adding such an edge will create a cycle. If this is false then they are not in the same component and adding such an edge will simply union those components.

Note that we have seen that we can check this in $\mathcal{O}(\alpha(n))$ amortized time, which is almost constant.

Example 5.2. Returning to our example:

Does addition of the edge $(3, 6)$ form a cycle? Well the root of 3 is 5 and the root of 6 is 6 so no.

Does addition of the edge $(1, 2)$ form a cycle? Well the root of 1 and the root of 2 are both 1 so yes.

6 Kruskal's Algorithm

6.1 Introduction

Kruskal's Algorithm finds a minimum weight spanning tree in a weighted, connected, and undirected graph.

It works by first ignoring all the edges and then by progressively adding back a minimum-weight edge which does not form a cycle until the graph is connected.

The difficult part of Kruskal's Algorithm is the cycle-detection step. When we pick a minimum-weight edge we need to know if adding it to the graph will form a cycle or not.

6.2 Using a Disjoint Set Data Structure

We now discuss how Kruskal's algorith can work in $\mathcal{O}(E\alpha(V))$ amortized time. This demands that we have information for the graph stored very specifically.

In the following pseudocode we assume:

- The graph has V vertices and E edges
- `EL` is a list of the edges in increasing order by weight.
- `F` is a disjoint set forest with V isolated vertices.
- `F` uses path compression and weighted union.
- `K` is an empty list.

When the code ends `K` will contain all the edges in the minimal spanning tree.

```
for each edge(u,w) in EL:
    if F.findrep(u) != F.findrep(w)
        K.append(edge(u,w))
        F.union(u,v)
    end if
end for
```

6.3 Time Complexity

The `for` loop iterates E times.

In the body of the `for` loop we need to calculate `findrep(u)` and `findrep(v)` and with V vertices it takes time $\alpha(V)$ for each. If the `if` is satisfied then we also need to do `F.union(u,w)` which takes time $\alpha(V)$.

Consequently the body of the `for` loop takes $\mathcal{O}(\alpha(V))$ time so the entire algorithm takes $\mathcal{O}(E\alpha(V))$ time.

7 Maze Generation

7.1 Introduction

Consider the problem of generating an $n \times n$ maze. Ideally a maze should have the following properties:

- No circular routes (cycles).
- Every cell is reachable from every other square.

7.2 Using a Disjoint Set Data Structure

Suppose we begin by creating an $n \times n$ grid of cells labeled like an array, so the cells will be $(1,1), ..., (1,n), (2,1), ..., 2,n), ..., (n,1), ..., (n,n)$.

For now let's assume that every cell is isolated so every wall between them is filled-in. We put these walls in a list W and shuffle it randomly. Treat it like a queue. Note that a wall has the form $\{c_1, c_2\}$ where c_1 and c_2 are the two cells it separates.

We also create a disjoint set forest which contains one tree (a single node) for each cell. What we will do is remove the walls between cells which will progressively link the cells together like components of a graph.

We proceed as follows:

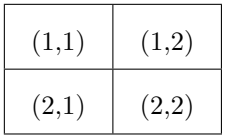
1. Pop a wall $\{c_1, c_2\}$ off W.
2. If the cells c_1 and c_2 are not in the same tree then union their trees.
3. Continue until we have one tree.

In pseudocode this looks like:

```
F = Disjoint set forest with one tree for each cell.
    Each tree is just that cell.
W = shuffled queue of walls
K = empty list
while F contains more than one tree:
    (c1,c2) = W.dequeue
    if F.findrep(c1) != F.findrep(c2)
        F.union(c1,c2)
        K.append( (c1,c2) )
    end
end
```

When this ends the list `K` will contain all the walls which were removed.

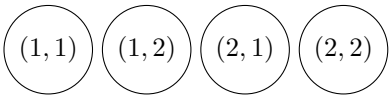
Example 7.1. Let's create a 2×2 maze. We start with the grid:



There are 4 edges which we shuffle to get: We shuffle to get:

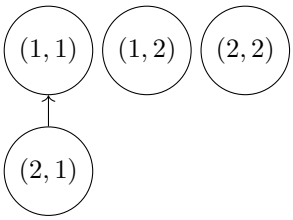
$$W = \{\{(1,1),(2,1)\},\{(1,1),(1,2)\},\{(1,2),(2,2)\},\{(2,1),(2,2)\}\}$$

Our disjoint forest is:

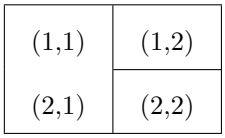


This represents the fact that our four cells are all disconnected.

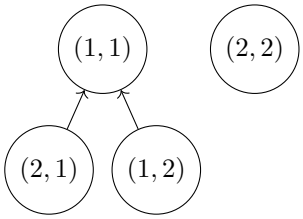
We dequeue the wall $\{(1,1),(2,1)\}$ and since $(1,1)$ and $(2,1)$ are not in the same tree we union the trees. We also have $K = [\{(1,1),(2,1)\}]$, the wall which was removed:



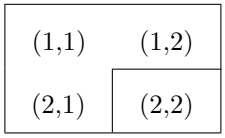
The maze is now:



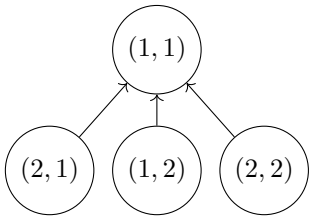
We dequeue the wall $\{(1,1),(1,2)\}$ and since $(1,1)$ and $(1,2)$ are not in the same tree we union the trees. We also have $K = [\{(1,1),(2,1)\},\{(1,1),(1,2)\}]$, the wall which were removed:



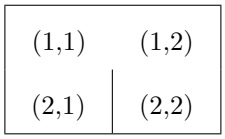
The edges that remain in $W = \{\{(1,2),(2,2)\},\{(2,1),(2,2)\}\}$ are the edges that remain in the maze:



We dequeue the wall $\{(1,2),(2,2)\}$ and since $(1,2)$ and $(2,2)$ are not in the same tree we union the trees: We also have $K = [\{(1,1),(2,1)\},\{(1,1),(1,2)\},\{(1,2),(2,2)\}]$, the walls which were removed:



We now have just one tree and the resulting maze is:



Okay it's a pretty boring maze but it works!

7.3 Time Complexity

It takes $\mathcal{O}(n^2)$ to set up the Disjoint Set Forest F.. In an $n \times n$ grid of cells there will be $2n(n-1)$ walls which a Fisher-Yates shuffle can shuffle in $\mathcal{O}(n^2)$.

The `while` loop then iterates $\mathcal{O}(n^2)$ times and the body of the loop is $\mathcal{O}(\alpha(n^2))$.

Overall we then have time complexity $\mathcal{O}(n^2\alpha(n^2))$.