

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 8 | Part 1

Today's Lecture

Disjoint Sets

- ▶ Often need to keep a collection of **disjoint sets**.
 - ▶ Example: $\{\{4, 6, 2, 0\}, \{1, 3\}, \{5\}\}$
- ▶ May need to union disjoint sets.
- ▶ May need to check if two items are in same set.

Use Case

- ▶ We are given a **stream** of nodes, edges.
- ▶ Want to keep track of CCs at every step.
- ▶ BFS/DFS take $\Theta(V + E)$ time; efficient to compute CCs once, but then need to recompute.

Use Cases

- ▶ Used in Kruskal's algorithm for MST.
- ▶ Used in single linkage clustering.
- ▶ Used in Tarjan's algorithm to find LCA in a tree.

Disjoint Sets, Abstractly

- ▶ A **disjoint sets** ADT represents a collection of disjoint sets.
 - ▶ Example: $\{\{4, 6, 2, 0\}, \{1, 3\}, \{5\}\}$
- ▶ Supports three operations:
 - ▶ `.make_set()`, `.find_set(x)`, `.union(x, y)`
- ▶ Sometimes called a **Union-Find** data type.

Assumption

- ▶ Elements are consecutive integers.
 - ▶ Example: $\{\{4, 6, 2, 0\}, \{1, 3\}, \{5\}\}$
- ▶ Not really a limitation.
 - ▶ Keep dictionary mapping, e.g., string ids to integers.

.make_set()

- ▶ Create a new singleton set.
- ▶ Element “id” automatically inferred, returned.

```
»» ds = DisjointSet()
»» ds.make_set()
0
»» ds.make_set()
1
»» ds.make_set()
2
```

.union(x, y)

- ▶ Union sets containing x and y.
- ▶ Updates data structure in-place.

```
»» ds = DisjointSet()
»» ds.make_set()
0
»» ds.make_set()
1
»» ds.make_set()
2
»» ds.union(0, 2)
```


.find_set(x)

- ▶ Find **representative** of set containing x.
- ▶ Representative is arbitrary, but same for all items in same set.
- ▶ Used to test if two nodes in same set.
- ▶ Guaranteed to not change unless a union is performed.

```
»» # ds is {{0}, {1}, {2}}
»» ds.union(0, 2)
»» ds.find_set(0)
0
»» ds.find_set(2)
0
»» ds.union(0, 1)
»» ds.find_set(0)
1
»» ds.find_set(1)
1
»» ds.find_set(2)
1
```

Today's Lecture

- ▶ How do we implement a disjoint set?
- ▶ We'll introduce the **disjoint set forest** data structure.
- ▶ Talk about two heuristics that make it very efficient.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 8 | Part 2

Disjoint Set Forests

Implementing Disjoint Sets

- ▶ First idea: a **list** of **sets**.

`[{2, 4, 3}, {1, 5}, {0}]`

- ▶ **Problem**: unioning two **sets** takes time linear in size of smaller.

Looking Ahead

- ▶ We'll design data structure so that all operations, including union, take (practically) $\Theta(1)$ time.

The Idea

- ▶ Represent collection as a forest of trees, called a **Disjoint Set Forest**.
- ▶ Example:
 $\{\{2, 4, 3, 6\}, \{1, 5\}, \{0\}\}$
- ▶ Not unique!

Tree Structure

- ▶ Each node has reference to **parent**.
- ▶ Not a binary tree!

Representing Forests

- ▶ We have several choices:
- ▶ 1) Each node is own **object** with parent attribute.
- ▶ 2) Keep a **list** containing parent of each element.

Approach #1

```
class DSFNode:
```

```
    def __init__(self, parent=None):  
        self.parent = parent
```

- ▶ `make_set` becomes `DSFNode()`
- ▶ `find_set` and `union` are functions, not methods.
- ▶ They accept `DSFNode` objects.

Approach #2

```
class DisjointSetForest:
    def __init__(self):
        # self._parent[i] is
        # parent of element i
        self._parent = []

    def make_set(self):
        ...

    def find_set(self, x):
        ...

    def union(self, x, y):
        ...
```

Implementation Notes

- ▶ We'll use the second approach.
- ▶ We can use second representation because elements are consecutive integers.
- ▶ For cache locality, use numpy array, not `list`.

.make_set

```
def make_set(self):  
    # infer new element's "id"  
    x = len(self._parent)  
    self._parent.append(None)  
    return x
```

```
»» dsf = DisjointSetForest()  
»» dsf.make_set()  
0  
»» dsf.make_set()  
1  
»» dsf.make_set()  
2  
»» dsf._parent  
[None, None, None]
```

`.find_set(x)`

- ▶ Idea: use the “root” as the representative.

.find_set

```
def find_set(self, x):  
    if self._parent[x] is None:  
        return x  
    else:  
        return self.find_set(self._parent[x])
```

`.union(x, y)`

- ▶ Idea: make one root the parent of the other.

.union(x, y)

```
def union(self, x, y):  
    x_rep = self.find_set(x)  
    y_rep = self.find_set(y)  
    if x_rep != y_rep:  
        self._parent[y_rep] = x_rep
```

```
»» # dsf is {{0}, {1}, {2}}  
»» dsf._parent  
[None, None, None]  
»» dsf.union(0, 1)  
»» dsf._parent  
[None, 0, None]  
»» dsf.union(1, 2)  
»» dsf._parent  
[None, 0, 0]
```


Analysis

- ▶ `.make_set`: $\Theta(1)$ time¹
- ▶ `.union`: depends on `.find_set`
- ▶ `.find_set`: $O(h)$, where h is height of tree

¹Amortized, since we're using a dynamic array. But truly $\Theta(1)$ with an over-allocated static array or in the object representation.

Tree Height

- ▶ Trees can be very deep, with $h = O(n)$.
 - ▶ `.find_set` and `.union` can take $\Theta(n)$ time!

- ▶ Example:

```
# dsf is {{0}, {1}, {2}, {3}, {4}}
»» dsf.union(1, 0)
»» dsf.union(2, 1)
»» dsf.union(3, 2)
»» dsf.union(4, 3)
```

Tree Height

- ▶ But trees can also be shallow, with $h = O(1)$.

- ▶ Example:

```
# dsf is {{0}, {1}, {2}, {3}, {4}}  
»» dsf.union(0, 1)  
»» dsf.union(1, 2)  
»» dsf.union(2, 3)  
»» dsf.union(3, 4)
```

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 8 | Part 3

Path Compression and Union-by-Rank

The Bad News

- ▶ We saw that the tree can become very deep.
- ▶ In worst case, `.find_set` and thus `.union` take $\Theta(n)$ time.

Heuristics

- ▶ Now: two heuristics helping trees stay shallow.
- ▶ **Union-by-Rank** and **Path Compression**
- ▶ Together, these result in a massive speed up.

Path Compression

- ▶ Idea: if we find a long path during `.find_set`, “compress” it to (possibly) reduce height.

.find_set

```
def find_set(self, x):  
    if self._parent[x] is None:  
        return x  
    else:  
        root = self.find_set(self._parent[x])  
        self._parent[x] = root  
        return root
```


Union-by-Rank

- ▶ Should we `.union(x, y)` or `.union(y, x)`?

Union-by-Rank

- ▶ Placing deeper tree under shallower tree increases height by one.
- ▶ But placing shallower tree under deeper tree doesn't increase height.
- ▶ **Idea:** always place shallower tree under deeper.

Rank

- ▶ We need to keep track of height (**rank**) of each tree.
- ▶ Store rank attribute.
- ▶ $\text{rank}[i]$ is height² of tree rooted at node i .

²Exactly the height if path compression isn't used, but upper bound if it is.

Rank

```
class DisjointSetForest:

    def __init__(self):
        self._parent = []
        self._rank = []

    def make_set(self):
        # infer new element's "id"
        x = len(self._parent)
        self._parent.append(None)
        self._rank.append(0)
        return x
```

.union

```
def union(self, x, y):  
    x_rep = self.find_set(x)  
    y_rep = self.find_set(y)  
  
    if x_rep == y_rep:  
        return  
  
    if self._rank[x_rep] > self._rank[y_rep]:  
        self._parent[y_rep] = x_rep  
    else:  
        self._parent[x_rep] = y_rep  
        if self._rank[x_rep] == self._rank[y_rep]:  
            self._rank[y_rep] += 1
```

Note

- ▶ With path compression, rank is no longer *exactly* the height – it is an upper bound.
- ▶ But this is good enough.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 8 | Part 4

Analysis

Analysis of DSF

- ▶ A DSF with path compression and union-by-rank ensures trees are shallow.
- ▶ How does this affect runtime?

Answer

- ▶ Assuming union-by-rank and path compression...
- ▶ In a sequence of m operations, n of which are `.make_sets...`
- ▶ Amortized cost of a single operation is $O(\alpha(n))$.
- ▶ α is the **inverse Ackermann function**, and it is essentially constant.

Inverse Ackermann

$\alpha(n)$	n
0	$n \in [0, 1, 2]$
1	$n = 3$
2	$n \in [4, \dots, 7]$
3	$n \in [8, \dots, 2047]$
4	$n \in [2048, \dots, 2^{2048}]$ and beyond

Proof

- ▶ The formal analysis is quite involved.
- ▶ But we'll provide some intuition.

Union-by-rank Alone

- ▶ Union-by-rank alone ensures height is $O(\log n)$.

```
# dsf is {{0}, {1}, {2}, {3}}  
»» dsf.union(0, 1)  
»» dsf.union(2, 3)  
»» dsf.union(0, 2)
```

Union-by-rank Alone

- ▶ Union-by-rank alone ensures `.find_set` is $O(\log n)$.

Path Compression + U-by-R

- ▶ With path compression, individual `.find_set` calls can take $O(\log n)$.
- ▶ But they massively improve subsequent calls.
 - ▶ For other nodes, too!