Chapter 4

Greedy Algorithms
A Disjoint set data structure implements three functions:

- **MakeSet(x)**: creates a set with one element x in it.
  
  MakeSet(V), creates n sets each with one element of V.

- **Find(x)**: returns a set-canonical member for set-of(x).

- **Union(x,y)**: merges the set containing x and the set containing y into one new set with one canonical element.

Let V={1,2,3,4,5}

MakeSet(V) = \{ \{1\}, \{2\}, \{3\}, \{4\}, \{5\} \}

Find(4) = 4

Union(1,2) changes the sets to \{ \{1,2\}, \{3\}, \{4\}, \{5\} \}

Find(2) = 1.

Union(2,5) changes the sets to \{ \{1,2,5\}, \{3\}, \{4\} \}

Find(5) = 1
A first try using simple array organization

MakeSet(V)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Union(2,5)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Find(2) = Find(7) = 2

2 = Find(5) ≠ Find(6) = 6

Union(5,7)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- A first try using simple array organization

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Find is $O(1)$
- Makeset is $O(1)$ or $O(n)$
- But Union is $\Omega(n)$
- We have to change all elements that point to 2 when an element that returns 2 is unionized with another.
A second try using array and linked list

**MakeSet(V)**

<table>
<thead>
<tr>
<th>0</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>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>

**Union(2,5)**

<table>
<thead>
<tr>
<th>0</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>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</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>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</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>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</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>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>
A second try using array and linked list

Union(5,7)
A second try using array and linked list

Union(5,7)
A second try using array and linked list

How long does it take to do Union(5,1)?

- You need to update all the pointers in the main array to the new combined list.
- Hence, in the worst case, Union requires $\Omega(n)$ time.
- Find and MakeSet still require constant time.
A second try using array and linked list

- There are sequences of Union that require $\Omega(n^2)$ amortized time. Just take $\text{Union}(0, i+1)$ for all $1 \leq i \leq n$. If the second list always gets moved into the first one, then after $n$ Union, we have modified $\Omega(n^2)$ values in the main array.

- But there is a heuristic that we can use to counter this bad side effect. It is called Union-by-rank.
Union-Find DS: Union-by-rank

- A second try using array and linked list

Union(5,1)

Union(6,7)
- A second try using array and linked list

**Union(5,1)**

```
0  nil
1
2
3
4
5 nil
6 nil
7 nil
... nil
... nil
```

**Union(6,7)**

```
1
5
3 nil
2
7 nil
6 nil
```
A second try using array and linked list

Union-Find DS: Union-by-rank 2

Union(5, 6)
- A second try using array and linked list

Union-Find DS: Union-by-rank 2

Union(5, 6)
- Each time a list is added at the back of another list, its size doubles (the smallest).
- How many times can the size of a list double if the maximum size it can achieve is \( n \)?
- \( \log(n) \) (no element can be updated more than once)
- Yes, in the final step we work \( \Omega(n) \), but we can work for only \( \log(n) \) times.
- Hence the total work done to do \( n-1 \) Union and \( m \) Find is \( O(m+n \log(n)) \)
- A definite improvement over \( O(n^2) \).
- Find(x) takes $O(1)$
- $n$ Union(x,y) takes in the worst case $O(n \log n)$
- MakeSet(V) takes $O(n)$
- Hence $m$ Find operations and $n-1$ Union operations require in the worst case $O(m+n \log n)$
- But we can still do better: Almost linear in the worst case for $m$ Find operations and $n-1$ Unions
**Union-Find DS: Path compression**

**Makeset\{V\}**

- 1
- 2
- 3
- 4
- 5
- 6
- 7

**Union(2,5)**

- 1
- 2
- 3
- 4
- 5
- 6
- 7

**Union(6,7)**

- 1
- 2
- 3
- 4
- 5
- 6
- 7
Union-Find DS: Path compression 2

Union(1,2) → Union(5,6)

2 = Find(5) = Find(6) 2 = Find(7) ≠ Find(4) = 4
- How long does it take on average to do a Find(x) operation?
- Union(x,y) operation?
- Makeset(V)?
- We can use Union-by-rank.
- Just keep a pointer in the array or in each cell.
**Union-Find DS: Path compression 4**

**Find(x)::**
while \( x \neq \text{prev}(x) \)
  \( x=\text{prev}(x) \)
return \( x \)

**Union(x,y)::**
rx = Find(x)
ry = Find(y)
if rx==ry
  return
if rank(rx)>rank(ry)
  prev(ry)=rx
else
  prev(rx)=ry
if rank(rx)==rank(ry)
  rank(ry)=rank(ry)+1

Here rank means depth of the tree rooted at rx
Union-Find DS: Path compression

1. Union(6,7)
2. Union(8,2)
3. Union(7,5)
4. Union(4,5)
5. Union(3,7)
A few observations:
- $\text{rank}(x) < \text{rank}(\text{prev}(x))$
- Any sub-tree of rank $k$ has at least $2^k$ nodes.

Pf: At the beginning all nodes have rank 0. $2^0 = 1$.

When does the rank change?

*When we unionize equal rank trees.*

Hence if both trees had at least $2^k$ nodes, then the new tree has $2^k + 2^k = 2^{k+1}$ nodes and rank $k+1$.

Finally, over $n$ nodes, only $n/2^k$ nodes can have rank $k$.

From this we conclude that the maximum depth of a tree is $\log(n)$ since $n/2^{\log n} = 1$. 

Since the maximum dept of a tree is at most $\log n$, we can conclude that:

- Find($x$) takes at most $O(\log n)$ time,
- n Union($x,y$) takes at most $O(n \log n)$ time,
- Makeset($V$) is still linear.

But Find($x$) for our linked list implementation used $O(1)$ and we are not quicker for Union and Makeset.

**What happened?**
Well, we are not done.
This data structure needs another heuristic in order to be competitive.
Find(x)::
if x ≠ prev(x) then
    prev(x) = Find(prev(x))
return prev(x)

Find(x)::
r = x
while r ≠ prev(r)
    r = prev(r)
while x ≠ prev(x)
    x' = prev(x)
    prev(x) = r
    x = x'
return x
Find(x)::
if x ≠ prev(x) then
    prev(x) = Find(prev(x))
return prev(x)

Union(x,y)::
rx = Find(x)
ry = Find(y)
if rx==ry
    return
if rank(rx)>rank(ry)
    prev(ry)=rx
else
    prev(rx)=ry
if rank(rx)=rank(ry)
    rank(ry)=rank(ry)+1
Union-Find DS: Path compression

Makeset\{V\}

1 -> 2 -> 3
1 -> 2 -> 5
Union(2, 5)

1 -> 2 -> 3
1 -> 2
1

3
4
5
6
7

Union(6, 7)

6
7
Union-Find DS: Path compression 10

1. Union(1,2)
2. Union(5,6)
3. Union(3,4)

2 = Find(5) = Find(6) 2 = Find(7) ≠ Find(4) = 3
Union-Find DS: Path compression 11

After Union(4,1) and Find(7):
- Intuitively, the tree cannot get very deep as we constantly make it flatter.
- MakeSet(V) takes $O(n)$.
- In fact, any sequence of $m$ operations (MakeSet, Union and Find) $n$ of which are MakeSet(x) operations require

$$\Theta(m \cdot \alpha(m, n))$$ operations.

- The function $\alpha(m,n)$ is the extremely slowly growing inverse of Ackermann’s function. For all practical (and un-practical) problem instances, it is at most 4.
- So in Practice, it requires $\Theta(4m)$ time for $m$ operations, $n$ of which are Union.
Input: \( G = (V, E, \text{cost}(E)) \)
output: \( T \)

for all \( u \) in \( V \)
   Makeset(u)

\( T = \{ \} \)

sort the edges \( E \) by \( \text{cost}(E) \)

for each edges \( (u, v) \) in (sorted) \( E \)
   if \( \text{Find}(u) \neq \text{Find}(v) \)
      \( T = T + (u, v) \)
      Union(u, v)
   if(size(T) = |V| - 1)
      done

Hence, our data structure operations will require
\( \Theta( (m+n) \cdot \alpha((m+n), n) ) \) or roughly \( \Theta( m+n ) \).
Chapter 4

Greedy Algorithms