Union-Find

Robin Visser

IOI Training Camp
University of Cape Town

3 December 2016

Overview

Definition

We want a data structure which keeps track of a set of *elements* partitioned into *disjoint* subsets, which implement the following three operations:

Definition

We want a data structure which keeps track of a set of *elements* partitioned into *disjoint* subsets, which implement the following three operations:

- **MakeSet**: Constructs a subset containing a single element.

Definition

We want a data structure which keeps track of a set of *elements* partitioned into *disjoint* subsets, which implement the following three operations:

- **MakeSet**: Constructs a subset containing a single element.
- **Find**: Determine which subset a particular element is in.

Definition

We want a data structure which keeps track of a set of *elements* partitioned into *disjoint* subsets, which implement the following three operations:

- **MakeSet**: Constructs a subset containing a single element.
- **Find**: Determine which subset a particular element is in.
- **Union**: Join two subsets into a single subset.

Definition

We want a data structure which keeps track of a set of *elements* partitioned into *disjoint* subsets, which implement the following three operations:

- **MakeSet**: Constructs a subset containing a single element.
- **Find**: Determine which subset a particular element is in.
- **Union**: Join two subsets into a single subset.

For each subset, **Find** will usually return a *representative* element of that set, and **Union** will take two representative elements as its arguments.

Linked List Approach

- One easy solution is to use a linked list approach, where the head of the linked list is the representative element.

Linked List Approach

- One easy solution is to use a linked list approach, where the head of the linked list is the representative element.
- Each element is initially represent by a single linked list containing just that element.

Linked List Approach

- One easy solution is to use a linked list approach, where the head of the linked list is the representative element.
- Each element is initially represent by a single linked list containing just that element.
- The **Union** operation simply concatenates two linked lists.

Linked List Approach

- One easy solution is to use a linked list approach, where the head of the linked list is the representative element.
- Each element is initially represent by a single linked list containing just that element.
- The **Union** operation simply concatenates two linked lists.
- The **Find** operation traverses through the linked list until it reaches the head.

Linked List Approach

- One easy solution is to use a linked list approach, where the head of the linked list is the representative element.
- Each element is initially represent by a single linked list containing just that element.
- The **Union** operation simply concatenates two linked lists.
- The **Find** operation traverses through the linked list until it reaches the head.

Drawback: **Union** takes $O(1)$ time (assuming pointers to the tail), but **Find** takes $O(n)$ time.

Linked List Approach

- One easy solution is to use a linked list approach, where the head of the linked list is the representative element.
- Each element is initially represent by a single linked list containing just that element.
- The **Union** operation simply concatenates two linked lists.
- The **Find** operation traverses through the linked list until it reaches the head.

Drawback: **Union** takes $O(1)$ time (assuming pointers to the tail), but **Find** takes $O(n)$ time.

Alternatively: Keeping pointers to the head in each node allows us to have **Find** in $O(1)$ time, but **Union** takes $O(n)$ time.

Tree Approach

- We can alternatively represent the subsets as trees, where each element simply holds a reference to its parent node.

Tree Approach

- We can alternatively represent the subsets as trees, where each element simply holds a reference to its parent node.
- **Find** follows parents until it reaches a root.

Tree Approach

- We can alternatively represent the subsets as trees, where each element simply holds a reference to its parent node.
- **Find** follows parents until it reaches a root.
- **Union** attaches the root of the one tree to the root of the other.

Tree Approach

- We can alternatively represent the subsets as trees, where each element simply holds a reference to its parent node.
- **Find** follows parents until it reaches a root.
- **Union** attaches the root of the one tree to the root of the other.

Drawback: This is essentially the same as a linked list, as the trees could become highly unbalanced, with the **Find** operation still taking $O(n)$ time worst case.

Code

Pseudocode:

```
def MakeSet(x):
    parent[x] = x

def Find(x):
    if parent[x] == x:
        return x
    return Find(parent[x])

def Union(x, y):
    xRoot, yRoot = Find(x), Find(y)
    parent[xRoot] = yRoot
```

First optimisation

We can optimise this approach by always attaching the *smaller* tree to the root of the *larger* tree. This is called Union by rank

First optimisation

We can optimise this approach by always attaching the *smaller* tree to the root of the *larger* tree. This is called Union by rank

- We simply keep an additional parameter, the *depth* of each node, which denotes the size of the tree it represents.

First optimisation

We can optimise this approach by always attaching the *smaller* tree to the root of the *larger* tree. This is called Union by rank

- We simply keep an additional parameter, the *depth* of each node, which denotes the size of the tree it represents.
- Each node is initialised with a depth of 0.

First optimisation

We can optimise this approach by always attaching the *smaller* tree to the root of the *larger* tree. This is called Union by rank

- We simply keep an additional parameter, the *depth* of each node, which denotes the size of the tree it represents.
- Each node is initialised with a depth of 0.

This will ensure each tree stays balanced, therefore resulting in a worst case time of only $O(\log n)$ for the **Find** operation.

Code

```
def MakeSet(x):
    parent[x] = x
    rank[x] = 0

def Union(x, y):
    xRoot, yRoot = Find(x), Find(y)
    if (xRoot == yRoot): return

    if rank[xRoot] < rank[yRoot]:
        parent[xRoot] = yRoot
    elif rank[xRoot] > rank[yRoot]:
        parent[yRoot] = xRoot
    else:
        parent[yRoot] = xRoot
        rank[xRoot] += 1
```

Second optimisation

We can optimise even further by flattening the tree whenever we call the **Find** operation on it (noting that we might as well have each node pointing directly to its representative). This is called *path compression*.

Second optimisation

We can optimise even further by flattening the tree whenever we call the **Find** operation on it (noting that we might as well have each node pointing directly to its representative). This is called *path compression*.

This speeds up future **Find** operations for those elements, as well as other elements referencing them

Code

```
def Find(x):
    if parent[x] != x:
        parent[x] = Find(parent[x])
    return parent[x]
```

Optimised Running Time

Using both optimisation techniques, one obtains an amortised time per operation of $O(\alpha(n))$, where $\alpha(n)$ denotes the inverse Ackermann function. Since this function grows *very* slowly, it's practically constant for all reasonable values of n.

Optimised Running Time

Using both optimisation techniques, one obtains an amortised time per operation of $O(\alpha(n))$, where $\alpha(n)$ denotes the inverse Ackermann function. Since this function grows *very* slowly, it's practically constant for all reasonable values of n.

Note: $\alpha\left(2^{2^{2^{65536}}}\right) = 4$.

Optimised Running Time

Using both optimisation techniques, one obtains an amortised time per operation of $O(\alpha(n))$, where $\alpha(n)$ denotes the inverse Ackermann function. Since this function grows *very* slowly, it's practically constant for all reasonable values of n.

Note: $\alpha\left(2^{2^{2^{65536}}}\right) = 4$.

Quite remarkably, one can prove that we cannot do any better. $O(\alpha(n))$ is the tightest bound we can obtain for a disjoint set data structure.