DMA 7g

Carl Dybdahl, Patrick Hartvigsen, Emil Chr. Søderblom

December 5, 2016

Part 1

(1)

We have been tasked with writing a pseudocode implementation of union-find where an array `A` is kept with the property that `A[i]` is the representative for `i`.

Our implementation of the initialization algorithm is the following:

```
A = []
init(n)
    A = new Array(n)
    for i = 0 to n - 1
        A[i] = i
```

To execute `find`, we simply look up in the array:

```
find(n)
    return A[n]
```

With `union`, we have to iterate through the entire array to change the relevant fields:

```
union(i, j)
        if find(i) == find(j)
                return
        repI = find(i)
        repJ = find(j)
        for k = 0 to A.Length - 1
                if A[k] == repJ
                        A[k] = repI
```

(2)

We have been tasked with manually computing a sequence of operations with our union-find. Here is our results:

```
        A = []
init(7)
        A = [0, 1, 2, 3, 4, 5, 6]
union(3, 4)
        A = [0, 1, 2, 3, 3, 5, 6]
union(5, 0)
        A = [5, 1, 2, 3, 3, 5, 6]
```

```
union(4, 5)
        A = [3, 1, 2, 3, 3, 3, 6]
union(4, 3)
        A = [3, 1, 2, 3, 3, 3, 6]
union(0, 1)
        A = [3, 3, 2, 3, 3, 3, 6]
union(2, 6)
        A = [3, 3, 2, 3, 3, 3, 2]
union(0, 4)
        A = [3, 3, 2, 3, 3, 3, 2]
union(6, 0)
        A = [2, 2, 2, 2, 2, 2, 2]
```

(3)

Our implementation maintains a stricter invariant on the union-find array than the fast implementation in the notes does. The notes require that if you follow the path `i -> A[i] -> A[A[i]] -> ...`, you eventually end up at the element that represents the set. However, our implementation requires that this path has at most length `1`, so that `A[i]` represents the set containing `i`.

Maintaining this invariant sometimes requires extra work, as we always have to iterate through the entire array when `union`ing. This means that our `union` implementation always runs in $O(n)$ time, whereas the notes only do this in a few pathological cases.

Part 2

(1)

We have been asked to prove the following sentence:

Theorem 1. *Let t be a tree constructed by `union`ing two trees t_1 and t_2 using the union-by-rank heuristic. If $rank(t_1) \neq rank(t_2)$ then $rank(t) \leq max(rank(t_1), rank(t_2))$. If $rank(t_1) = rank(t_2)$ then $rank(t) = 1 + rank(t_1)$.*

Proof. We will consider three cases: $rank(t_1) > rank(t_2)$, $rank(t_1) < rank(t_2)$ and $rank(t_1) = rank(t_2)$. In the first case, the root of t_2 is assigned as a child to the root of t_1, and the rank of the root of t_1 is not changed. This means that the root of t is the root of t_1, and so $rank(t) = rank(t_1) = max(rank(t_1), rank(t_2))$. A similar argument applies for case two. Therefore, if $rank(t_1) \neq rank(t_2)$, we have that $rank(t) \leq max(rank(t_1), rank(t_2))$. In case three, the root of t_1 is assigned as a child to the root of t_2. This means that the root of t is the same as the root of t_2. Then, the rank of the root of t is incremented. This means that $rank(t) = 1 + rank(t_1)$. □

(2)

We have been asked to fill out the missing parts of a proof.

The first missing part is proving the base case of a proof by strong induction. The proposition $P(n)$ that is being proven inductive is that trees of size n have a *rank* of at most $\log_2(n)$.

Lemma 1. *$P(1)$ holds.*

We use the implementation in CLRS page 571. We have disproven the above lemma with the following counterexample:

Proof. First, use the `Make-Set` algorithm to create a one-element set `x`. Then do `Union(x, x)`, which increases its rank to `1`. That is, we now have $rank(x) = 1 \geq 0 = \log_2 size(x)$, contradicting 1 $\square$

To avoid this, we propose changing the `Union` algorithm to the following:

```
Union(x, y):
    found-x = find(x)
    found-y = find(y)
    if found-x != found-y
        Link(found-x, found-y)
```

From now on, we will assume that `Union` does not increase the rank when called as `Union(x, x)`, with a correction along the lines of the listing above. We will now prove that in this case, $P(1)$ holds.

Proof. Consider the last operation applied to the tree t of size 1. We can without loss of generality assume that this is not `Union(t, t)`, since this has no effect. It cannot be the union of two other trees, because then it would have a size greater than 1. Therefore, it must be newly constructed using `Make-Set`. But `Make-Set` assigns a rank of $0 = log_2 1$, and therefore the property holds. $\square$

Next, we need to prove a part of the induction step.

Lemma 2. *A tree t with n nodes that has been constructed by a union of two trees t_1 and t_2 of size i and j respectively where $rank(t_1) \neq rank(t_2)$ has $rank(t) \leq \log_2 n$.*

Proof. In theorem 1 we proved that that $rank(t) \leq max(rank(t_1), rank(t_2))$. Observe that $rank(t_1) \leq \log_2 i \leq \log_2 n$, and similarly $rank(t_2) \leq \log_2 j \leq \log_2 n$. From that we see that $max(rank(t_1), rank(t_2)) \leq \log_2 n$, which means that $rank(t) \leq \log_2 n$. $\square$

(3)

Next we need to prove the following theorem:

Theorem 2. *The height of a tree t constructed with union-by-rank is less than or equal to $\log_2 size(t)$.*

Proof. The union-by-rank algorithm has been implemented so it maintains the invariant that $height(t) \leq rank(t)$, and we already know that $rank(t) \leq \log_2 size(t)$, which must imply that $height(t) \leq \log_2 size(t)$. $\square$