Chapter 21

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 21.1-1

EdgeProcessed											
initial	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$	$\{k\}$
(d,i)	$\{a\}$	$\{b\}$	$\{c\}$	$\{d,i\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$		$\{j\}$	$\{k\}$
(f,k)	$\{a\}$	$\{b\}$	$\{c\}$	$\{d,i\}$	$\{e\}$	$\{f,k\}$	$\{g\}$	$\{h\}$		$\{j\}$	
(g,i)	$\{a\}$	$\{b\}$	$\{c\}$	$\{d,i,g\}$	$\{e\}$	$\{f,k\}$		$\{h\}$		$\{j\}$	
(b,g)	$\{a\}$	$\{b,d,i,g\}$	$\{c\}$		$\{e\}$	$\{f,k\}$		$\{h\}$		$\{j\}$	
(a,h)	$\{a,h\}$	$\{b,d,i,g\}$	$\{c\}$		$\{e\}$	$\{f,k\}$				$\{j\}$	
(i,j)	$\{a,h\}$	$\{b,d,i,g,j\}$	$\{c\}$		$\{e\}$	$\{f,k\}$					
(d,k)	$\{a,h\}$	$\{b,d,i,g,j,f,k\}$	$\{c\}$		$\{e\}$						
(b,j)	$\{a,h\}$	$\{b,d,i,g,j,f,k\}$	$\{c\}$		$\{e\}$						
(d,f)	$\{a,h\}$	$\{b,d,i,g,j,f,k\}$	$\{c\}$		$\{e\}$						
(g,j)	$\{a,h\}$	$\{b,d,i,g,j,f,k\}$	$\{c\}$		$\{e\}$						
(a,e)	$\{a,h,e\}$	$\{b,d,i,g,j,f,k\}$	$\{c\}$								

So, the connected components that we are left with are $\{a,h,e\}$, $\{b,d,i,g,j,f,k\}$, and $\{c\}$.

Exercise 21.1-2

First suppose that two vertices are in the same connected component. Then there exists a path of edges connecting them. If two vertices are connected by a single edge, then they are put into the same set when that edge is processed. At some point during the algorithm every edge of the path will be processed, so all vertices on the path will be in the same set, including the endpoints. Now suppose two vertices u and v wind up in the same set. Since every vertex starts off in its own set, some sequence of edges in G must have resulted in eventually combining the sets containing u and v. From among these, there must be a path of edges from u to v, implying that u and v are in the same connected component.

Exercise 21.1-3

Find set is called twice on line 4, this is run once per edge in the graph, so, we have that find set is run $2|E|$ times. Since we start with $|V|$ sets, at the end

only have k, and each call to UNION reduces the number of sets by one, we have that we have to of made $|V| - k$ calls to UNION.

Exercise 21.2-1

The three algorithms follow the english description and are provided here. There are alternate versions using the weighted union heuristic, suffixed with WU.

Algorithm 1 MAKE-SET(x)

```
Let o be an object with three fields, next, value, and set
Let L be a linked list object with head = tail = o
o.next = NIL
o.set = L
o.value = x
return L
```

Algorithm 2 FIND-SET(x)

```
return o.set.head.value
```

Algorithm 3 UNION(x,y)

```
L1= x.set
L2 = y.set
L1.tail.next = L2.head
z = L2.head
while z.next ≠ NIL do
    z.set = L1
end while
L1.tail = L2.tail
return L1
```

Exercise 21.2-2

Originally we have 16 sets, each containing x_i. In the following, we'll replace x_i by i. After the for loop in line 3 we have:

$$\{1,2\},\{3,4\},\{5,6\},\{7,8\},\{9,10\},\{11,12\},\{13,14\},\{15,16\}.$$

After the for loop on line 5 we have

$$\{1,2,3,4\},\{5,6,7,8\},\{9,10,11,12\},\{13,14,15,16\}.$$

Line 7 results in:

Algorithm 4 MAKE-SET-WU(x)

```
L = MAKE-SET(x)
L.size = 1
return L
```

Algorithm 5 UNION-WU(x,y)

```
L1= x.set
L2 = y.set
if L1.size ≥ L2.size then
    L = UNION(x,y)
else
    L = UNION(y,x)
end if
L.size = L1.size + L2.size
return L
```

$$\{1,2,3,4,5,6,7,8\},\{9,10,11,12\},\{13,14,15,16\}.$$

Line 8 results in:

$$\{1,2,3,4,5,6,7,8\},\{9,10,11,12,13,14,15,16\}.$$

Line 9 results in:

$$\{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16\}.$$

FIND-SET(x_2) and FIND-SET(x_9) each return pointers to x_1.

Exercise 21.2-3

During the proof of theorem 21.1, we concluded that the time for the n UNION operations to run was at most $O(n\lg(n))$. This means that each of them took an amortized time of at most $O(\lg(n))$. Also, since there is only a constant actual amount of work in performing MAKE-SET and FIND-SET operations, and none of that ease is used to offset costs of UNION operations, they both have $O(1)$ runtime.

Exercise 21.2-4

We call MAKE-SET n times, which contributes $\Theta(n)$. In each union, the smaller set is of size 1, so each of these takes $\Theta(1)$ time. Since we union $n-1$ times, the runtime is $\Theta(n)$.

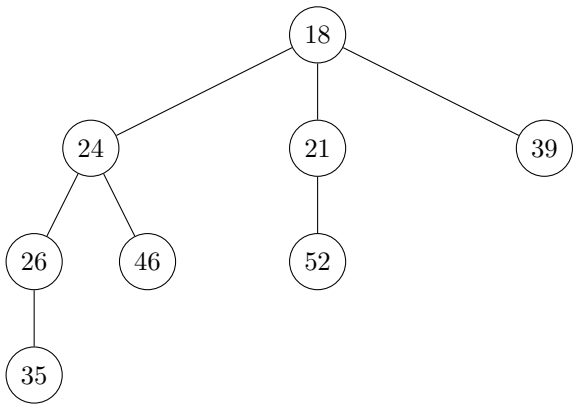
Exercise 21.2-5

For each member of the set, we will make its first field which used to point back to the set object point instead to the last element of the linked list. Then, given any set, we can find its last element by going ot the head and following the pointer that that object maintains to the last element of the linked list. This only requires following exactly two pointers, so it takes a constant amount of time. Some care must be taken when unioning these modified sets. Since the set representative is the last element in the set, when we combine two linked lists, we place the smaller of the two sets before the larger, since we need to update their set representative pointers, unlike the original situation, where we update the representative of the objects that are placed on to the end of the linked list.

Exercise 21.2-6

Instead of appending the second list to the end of the first, we can imagine splicing it into the first list, in between the head and the elements. Store a pointer to the first element in S_1. Then for each element x in S_2, set $x.head = S_1.head$. When the last element of S_2 is reached, set its next pointer to the first element of S_1. If we always let S_2 play the role of the smaller set, this works well with the weighted-union heuristic and don't affect the asymptotic running time of UNION.

Exercise 21.3-1



Exercise 21.3-2

To implement FIND-SET nonrecursively, let x be the element we call the function on. Create a linked list A which contains a pointer to x. Each time we most one element up the tree, insert a pointer to that element into A. Once the root r has been found, use the linked list to find each node on the path from the root to x and update its parent to r.

Exercise 21.3-3

Suppose that $n' = 2^k$ is the smallest power of two less than n. To see that this sequences of operations does take the required amount of time, we'll first note that after each iteration of the for loop indexed by j, we have that the elements $x_1, \ldots, x_{n'}$ are in trees of depth i. So, after we finish the outer for loop, we have that $x_1 \ldots x_{n'}$ all lie in the same set, but are represented by a tree of depth $k \in \Omega(\lg(n))$. Then, since we repeatedly call FIND-SET on an item that is $\lg(n)$ away from its set representative, we have that each one takes time $\lg(n)$. So, the last for loop alltogther takes time $m \lg(n)$.

Algorithm 6 Sequence of operations for Exercise 21.3-3

```
for i=1..n do
    MAKE-SET x_i
end for
for i = 1..k do
    for j = 1..n' − 2^{i=1} by 2^i do
        UNION(x_i, x_{i+2^{j−1}})
    end for
end for
for i = 1..m do
    FIND-SET(x_1)
end for
```

Exercise 21.3-4

In addition to each tree, we'll store a linked list (whose set object contains a single tail pointer) with which keeps track of all the names of elements in the tree. The only additional information we'll store in each node is a pointer $x.l$ to that element's position in the list. When we call MAKE-SET(x), we'll also create a new linked list, insert the label of x into the list, and set $x.l$ to a pointer to that label. This is all done in $O(1)$. FIND-SET will remain unchanged. UNION(x, y) will work as usual, with the additional requirement that we union the linked lists of x and y. Since we don't need to update pointers to the head, we can link up the lists in constant time, thus preserving the runtime of UNION. Finally, PRINT-SET(x) works as follows: first, set s = FIND-SET(x). Then print the elements in the linked list, starting with the element pointed to by x. (This will be the first element in the list). Since the list contains the same number of elements as the set and printing takes $O(1)$, this operation takes linear time in the number of set members.

Exercise 21.3-5

Clearly each MAKE-SET and LINK operation only takes time $O(1)$, so, supposing that n is the number of FIND-SET operations occuring after the making and linking, we need to show that all the FIND-SET operations only

take time $O(n)$. To do this, we will ammortize some of the cost of the FIND-SET operations into the cost of the MAKE-SET operations. Imagine paying some constant amount extra for each MAKE-SET operation. Then, when doing a FIND-SET(x) operation, we have three possibilities. First, we could have that x is the representative of its own set. In this case, it clearly only takes constant time to run. Second, we could have that the path from x to its set's representative is already compressed, so it only takes a single step to find the set representative. In this case also, the time required is constant. Lastly, we could have that x is not the representative and it's path has not been compressed. Then, suppose that there are k nodes between x and its representative. The time of this find-set operation is $O(k)$, but it also ends up compressing the paths of k nodes, so we use that extra amount that we paid during the MAKE-SET operations for these k nodes whose paths were compressed. Any subsequent call to find set for these nodes will take only a constant amount of time, so we would never try to use the work that amortization amount twice for a given node.

Exercise 21.4-1

The initial value of x.rank is 0, as it is initialized in line 2 of the MAKE-SET(x) procedure. When we run LINK(x,y), whichever one has the larger rank is placed as the parent of the other, and if there is a tie, the parent's rank is incremented. This means that after any LINK(y,x), the two nodes being linked satisfy this strict inequality of ranks. Also, if we have that $x \neq x.p$, then, we have that x is not its own set representative, so, any linking together of sets that would occur would not involve x, but that's the only way for ranks to increase, so, we have that x.rank must remain constant after that point.

Exercise 21.4-2

We'll prove the claim by strong induction on the number of nodes. If $n = 1$, then that node has rank equal to $0 = \lfloor \lg 1 \rfloor$. Now suppose that the claim holds for $1, 2, \ldots, n$ nodes. Given $n + 1$ nodes, suppose we perform a UNION operation on two disjoint sets with a and b nodes respectively, where $a, b \leq n$. Then the root of the first set has rank at most $\lfloor \lg a \rfloor$ and the root of the second set has rank at most $\lfloor \lg b \rfloor$. If the ranks are unequal, then the UNION operation preserves rank and we are done, so suppose the ranks are equal. Then the rank of the union increases by 1, and the resulting set has rank $\lfloor \lg a \rfloor + 1 \leq \lfloor \lg(n+1)/2 \rfloor + 1 = \lfloor \lg(n+1) \rfloor$.

Exercise 21.4-3

Since their value is at most $\lfloor \lg(n) \rfloor$, we can represent them using $\Theta(\lg(\lg(n)))$ bits, and may need to use that many bits to represent a number that can take that many values.

Exercise 21.4-4

MAKE-SET takes constant time and both FIND-SET and UNION are bounded by the largest rank among all the sets. Exercise 21.4-2 bounds this from about by $\lfloor \lg n \rfloor$, so the actual cost of each operation is $O(\lg n)$. Therefore the actual cost of m operations is $O(m \lg n)$.

Exercise 21.4-5

He isn't correct, suppose that we had that $rank(x.p) > A_2(rank(x))$ but that $rank(x.p.p) = 1 + rank(x.p)$, then we would have that $level(x.p) = 0$, but $level(x) \geq 2$. So, we don't have that $level(x) \leq level(x.p)$ even though we have that the ranks are monotonically increasing as we go up in the tree. Put another way, even though the ranks are monotonically increasing, the rate at which they are increasing (roughly captured by the level vales) doesn't have to, itself be increasing.

Exercise 21.4-6

First observe that by a change of variables, $\alpha'(2^{n-1}) = \alpha(n)$. Earlier in the section we saw that $\alpha(n) \leq 3$ for $0 \leq n \leq 2047$. This means that $\alpha'(n) \leq 2$ for $0 \leq n \leq 2^{2046}$, which is larger than the estimated number of atoms in the observable universe. To prove the improved bound of $O(m\alpha'(n))$ on the operations, the general structure will be essentially the same as that given in the section. First, modify bound 21.2 by observing that $A_{\alpha'(n)}(x.rank) \geq A_{\alpha'(n)}(1) \geq \lg(n+1) > x.p.rank$ which implies $level(x) \leq \alpha'(n)$. Next, redefine the potential replacing $\alpha(n)$ by $\alpha'(n)$. Lemma 21.8 now goes through just as before. All subsequent lemmas rely on these previous observations, and their proofs go through exactly as in the section, yielding the bound.

Problem 21-1

a.

index	*value*
1	4
2	3
3	2
4	6
5	8
6	1

b. As we run the for loop, we are picking off the smallest of the possible elements to be removed, knowing for sure that it will be removed by the next unused EXTRACT-MIN operation. Then, since that EXTRACT-MIN operation is used up, we can pretend that it no longer exists, and combine the set of things that were inserted by that segment with those inserted by the next, since we know that the EXTRACT-MIN operation that had separated the

two is now used up. Since we proceed to figure out what the various extract operations do one at a time, by the time we are done, we have figured them all out.

c. We let each of the sets be represented by a disjoint set structure. To union them (as on line 6) just call UNION. Checking that they exist is just a matter of keeping track of a linked list of which ones exist(needed for line 5), initially containing all of them, but then, when deleting the set on line 6, we delete it from the linked list that we were maintaining. The only other interaction with the sets that we have to worry about is on line 2, which just amounts to a call of FIND-SET(j). Since line 2 takes amortized time $\alpha(n)$ and we call it exactly n times, then, since the rest of the for loop only takes constant time, the total runtime is $O(n\alpha(n))$.

Problem 21-2

a. MAKE-TREE and GRAFT are both constant time operations. FIND-DEPTH is linear in the depth of the node. In a sequence of m operations the maximal depth which can be achieved is $m/2$, so FIND-DEPTH takes at most $O(m)$. Thus, m operations take at most $O(m^2)$. This is achieved as follows: Create $m/3$ new trees. Graft them together into a chain using $m/3$ calls to GRAFT. Now call FIND-DEPTH on the deepest node $m/3$ times. Each call takes time at least $m/3$, so the total runtime is $\Omega((m/3)^2) = \Omega(m^2)$. Thus the worst-case runtime of the m operations is $\Theta(m^2)$.

b. Since the new set will contain only a single node, its depth must be zero and its parent is itself. In this case, the set and its corresponding tree are indistinguishable.

Algorithm 7 MAKE-TREE(v)

```
v = Allocate-Node()
v.d = 0
v.p = v
Return v
```

c. In addition to returning the set object, modify FIND-SET to also return the depth of the parent node. Update the pseudodistance of the current node v to be $v.d$ plus the returned pseudodistance. Since this is done recursively, the running time is unchanged. It is still linear in the length of the find path. To implement FIND-DEPTH, simply recurse up the tree containing v, keeping a running total of pseudodistances.

d. To implement GRAFT we need to find v's actual depth and add it to the pseudodistance of the root of the tree S_i which contains r.

Algorithm 8 FIND-SET(v)

```
if v ≠ v.p then
    (v.p, d) = FIND − SET(v.p)
    v.d = v.d + d
    Return (v.p, v.d)
else
    Return (v, 0)
end if
```

Algorithm 9 GRAFT(r,v)

```
(x, d1) = FIND-SET(r)
(y, d2) = FIND-SET(v)
if x.rank > y.rank then
    y.p = x
    x.d = x.d + d2 + y.d
else
    x.p = y
    x.d = x.d + d2
    if x.rank == y.rank then
        y.rank = y.rank + 1
    end if
end if
```

e. The three implemented operations have the same asymptotic running time as MAKE, FIND, and UNION for disjoint sets, so the worst-case runtime of m such operations, n of which are MAKE-TREE operations, is $O(m\alpha(n))$.

Problem 21-3

a. Suppose that we let $\leq_{LCA}$ to be an ordering on the vertices so that $u \leq_{LCA} v$ if we run line 7 of $LCA(u)$ before line 7 of $LCA(v)$. Then, when we are running line 7 of $LCA(u)$, we immediately go on to the for loop on line 8. So, while we are doing this for loop, we still haven't called line 7 of LCA(v). This means that v.color is white, and so, the pair {u,v} is not considered during the run of LCA(u). However, during the for loop of LCA(v), since line 7 of LCA(u) has already run, u.color = black. This means that we will consider the pair {u,v} during the running of LCA(v).

   It is not obvious what the ordering $\leq_{LCA}$ is, as it will be implementation dependent. It depends on the order in which child vertices are iterated in the for loop on line 3. That is, it doesn't just depend on the graph structure.

b. We suppose that it is true prior to a given call of LCA, and show that this property is preserved throughout a run of the procedure, increasing the number of disjoint sets by one by the end of the procedure. So, supposing

that u has depth d and there are d items in the disjoint set data structure before it runs, it increases to d+1 disjoint sets on line 1. So, by the time we get to line 4, and call LCA of a child of u, there are d+1 disjoint sets, this is exactly the depth of the child. After line 4, there are now $d + 2$ disjoint sets, so, line 5 brings it back down to $d + 1$ disjoint sets for the subsequent times through the loop. After the loop, there are no more changes to the number of disjoint sets, so, the algorithm terminates with d+1 disjoint sets, as desired. Since this holds for any arbitrary run of LCA, it holds for all runs of LCA.

c. Suppose that the pair u and v have the least common ancestor w. Then, when running $LCA(w)$, u will be in the subtree rooted at one of w's children, and v will be in another. WLOG, suppose that the subtree containing u runs first. So, when we are done with running that subtree, all of their ancestor values will point to w and their colors will be black, and their ancestor values will not change until $LCA(w)$ returns. However, we run LCA(v) before $LCA(w)$ returns, so in the for loop on line 8 of LCA(v), we will be considering the pair $\{u, v\}$, since $u.color == BLACK$. Since u.ancestor is still w, that is what will be output, which is the correct answer for their LCA.

d. The time complexity of lines 1 and 2 are just constant. Then, for each child, we have a call to the same procedure, a UNION operation which only takes constant time, and a FIND-SET operation which can take at most amortized inverse Ackerman's time. Since we check each and every thing that is adjacent to u for being black, we are only checking each pair in P at most twice in lines 8-10, among all the runs of LCA. This means that the total runtime is $O(|T|\alpha(|T|) + |P|)$.