Maintaining a Partition

Jonathan Turner

January 31, 2013

This note is largely based on the analysis in Chapter 2 of *Data Structures and Network Algorithms* by Robert Tarjan, SIAM Press, 1985. The presentation has been expanded and adapted to provide a more step-by-step path leading up to the final result.

1 Introduction

A *partition* of a set is a collection of subsets that are pair-wise mutually exclusive and whose union includes every member of the set. In many graph algorithms it's useful to have a data structure for maintaining a partition on the vertices of the graph that allows us to quickly determine if two vertices are in the same subset, and to quickly combine two subsets. More precisely, the *partition data structure* maintains a partition on an underlying set S and supports the following operations.

- *partition*(S). Create a partition on the set S with each element of S belonging to a distinct set.
- *find*(x). Return the *canonical element* of the set containing x.
- *link*(x, y). Merge the two sets whose canonical elements are x and y. The original sets are destroyed, and the canonical element of the new set is returned.

The canonical element of a set is just some element that is used by the data structure as a representative of the set. The canonical element of a set is not changed by the *find* operation. We can determine if two elements are in the same set by comparing their canonical elements. This data structure is easy to implement and has a wide range of applications. It is often referred to as the *union-find* data structure. We will describe a particularly efficient implementation and analyze its performance.

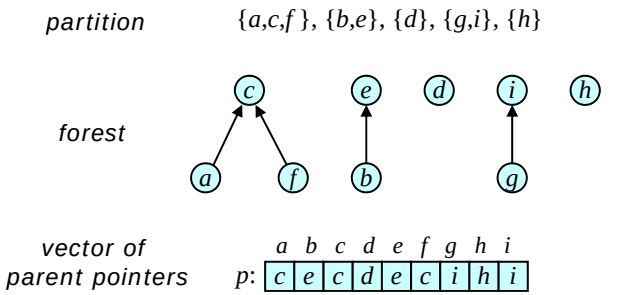

Figure 1: Example of partition data structure

2 Representing a Partition as a Collection of Trees

The partition data structure can be represented by a collection of trees or *forest*. Each element in the set is represented by a node in the forest and the trees are defined by *parent pointers* which identify the parent of each non-root node. To simplify the expression of the analysis, we define the parent pointer of a root node to be the node itself. Figure 1 shows an example of a partition, its conceptual representation as a collection of trees and its concrete implementation as a vector of parent pointers.

If we define the root of a tree to be the canonical element of its set, we can find the canonical element of a set by following parent pointers, and we can combine two sets by re-directing the parent pointer of one tree root to point to the other tree root. To get the most efficient implementation, we add two refinements to this basic idea. The first is called *path compression*. During each *find* operation, we re-direct each of the parent pointers along the path followed during the operation to point to the root of the tree. This requires a second pass along the path, but helps speed up later operations. The second refinement, called *linking by rank*, requires the addition of a new field to each tree node called its *rank*. For each node x, $rank(x)$ is initialized to zero and may be modified as a side effect of *link* operations. The ranks are used during link operations to decide which of the two tree roots becomes the child of the other. If one node has larger rank, then it becomes the new root and the ranks don't change. If both nodes have the same rank, one is chosen to be the root (arbitrarily) and its rank is increased by 1.

The following program implements the three operations.

```
procedure partition(set S);
        for x ∈ S ⇒ p(x) := x; rank(x) := 0; rof;
end;

int function find(int x);
        r := x;
        do p(r) ≠ r ⇒ r := p(r); od;
        do p(x) ≠ r ⇒ x, p(x) := p(x), r; od;
        return x;
end;

int function link(int x, y);
        if rank(x) > rank(y) ⇒
                p(y) := x; return x;
        | rank(x) = rank(y) ⇒
                rank(y) := rank(y) + 1; p(x) := y; return y;
        fi;
end;
```

3 Elementary Analysis

The initialization of the data structure takes $O(n)$ time and the link operation takes constant time. So, the only question is how much time is required for the find operation? This depends on the depth of the tree. We start by noting that the link-by-rank heuristic ensures that $rank(x)$ is an upper bound on the height of x in the tree. So, if x is a tree root, the time for a find operation starting within x's tree is at most $rank(x)$. So, if we can find an upper bound on $rank(x)$, we can bound the running time of the find operation. The key to bounding the rank is to observe that for any tree root x with $rank(x) = k$ the number of nodes in x's tree is at least 2^k.

This can be proved by induction on the number of link operations, as illustrated in Figure 2, giving us the following lemma.

Lemma 1 *The number of nodes in a tree with a root that has rank $= k$ is at least 2^k.*

This implies that if a tree root has rank k, then $2^k \leq n$ and so $k \leq \lg n$. This gives us an $O(\log n)$ bound on the time required for the find operation.

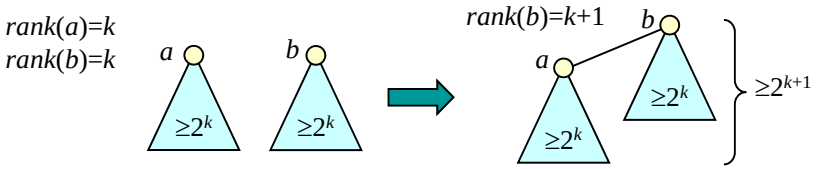

Figure 2: Relationship between rank and tree size

Note that this result does not depend on the path compression heuristic. Also, note that some find operations can take time proportional to $\log n$, so we can't get a better bound on the time for an arbitrary find operation. However, it turns out that we can get a better bound on a whole sequence of find operations, using a more sophisticated analysis that takes into account the effects of path compression.

4 A Better Analysis

As a first step in our more detailed analysis, we make a few observations about the ranks. First, note that the rank of a non-root node is strictly less than the rank of its parent. Second, the rank of a node can increase as a result of link operations, but it can never decrease. Third, once a node becomes a child of another node, its rank can no longer change. Finally, $rank(p(x))$ can increase (as a result of either link operations or find operations that change $p(x)$), but it can never decrease. These observations are collected in the next lemma.

Lemma 2 *If x is any node, $rank(x) \leq rank(p(x))$ with the inequality strict if $p(x) \neq x$. The value of $rank(x)$ is initially 0 and increases with time until $p(x)$ is assigned a value other than x. Subsequently, $rank(x)$ does not change. The value of $rank(p(x))$ is a nondecreasing function of time.*

To facilitate the analysis, we consider each find operation as consisting of a series of *find steps*, one for each node along the path followed by the operation. Our objective is to put an upper bound on the total number of find steps. We note that the number of find steps at root nodes is at most m, so in our analysis, we focus only on find steps at non-root nodes.

We define $\Delta(x) = rank(p(x)) - rank(x)$ and note that $\Delta(x) \leq \lg n$ and that $\Delta(x) = 0$ initially and increases, as x acquires a parent and participates

in find operations. Also, define a node x to be *dominant* if $\Delta(x) > 2\Delta(y)$ for all proper ancestors y of x.

We will break the analysis into two cases, one covering find steps that occur at dominant nodes, and the other covering find steps that occur at non-dominant nodes. The analysis of the first case rests on the observation that there can be at most $\lg \lg n$ dominant nodes along the path from any node to the root of its tree. This means that, each find operation can include at most $\lg \lg n$ find steps at dominant nodes. Consequently, in a sequence of m find operations, at most $m \lg \lg n$ find steps occur at dominant nodes.

Proceeding to the second case, note that if x is not dominant, then x must have some proper ancestor y with $\Delta(y) \geq \Delta(x)/2$. This implies that $rank(p(y)) - rank(x) \geq 1.5 \times rank(x)$ before the find. After the find completes, x's parent has rank at least as large the original value of $rank(p(y))$. Consequently, the find operation causes $\Delta(x)$ to increase by a factor of at least 1.5. This is illustrated in Figure 3.

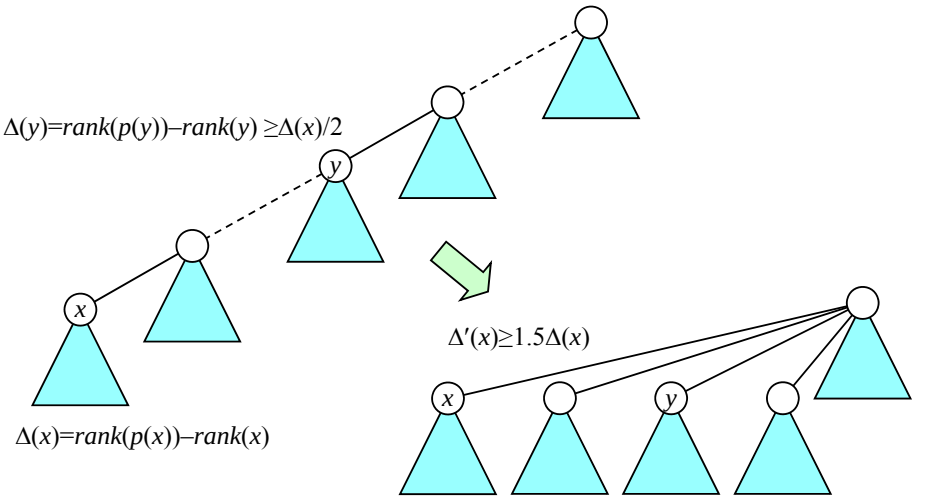

Figure 3: Find at a non-dominant node x increases $\Delta(x)$ by at least 50%

Since every find step that occurs at x at times when x is not dominant causes $\Delta(x)$ to increase by a factor of 1.5, there can be fewer than $2 \lg \lg n$ find steps at a node x when it is not dominant. Consequently, the total number of find steps at non-dominant nodes is $O(n \log \log n)$ and the total number of find steps is $O((m + n) \log \log n)$.

5 A Still Better Analysis

From a practical perspective, there is little reason to carry the analysis further, since for any practical value of n, $\lg \lg n$ is very small. However, it is possible to do better and it's instructive to see how.

The new analysis requires some additional technical machinery. Rather than go straight to the final analysis, we will build up to it in several steps, with each step providing some improvement on what has gone before. In the first step, we will show that the number of find steps is at $O(m+n \log \log n)$.

We start by dividing the non-negative integers into a series of *blocks* by defining $block(0) = [0, 1]$, $block(1) = [2, 3]$, $block(2) = [4, 5, 6, 7]$ and in general for $j \geq 1$, $block(j) = [2^j, ..., 2^{j+1} - 1]$. We say that a node x is on *level 1* if there is some $block(j)$ that contains both $rank(x)$ and $rank(p(x))$. All other nodes are on level 2. Note that once a node becomes a level 2 node, it can never again become a level 1 node (since $rank(x)$ does not change once x acquires a parent and $rank(p(x))$ never decreases). In the example shown in Figure 4, nodes a and d are on level 1, while nodes b and c are on level 2 We say that a node is *singular* if none of its proper ancestors is on the same level as it is. So for example, in Figure 4, nodes c and d are singular.

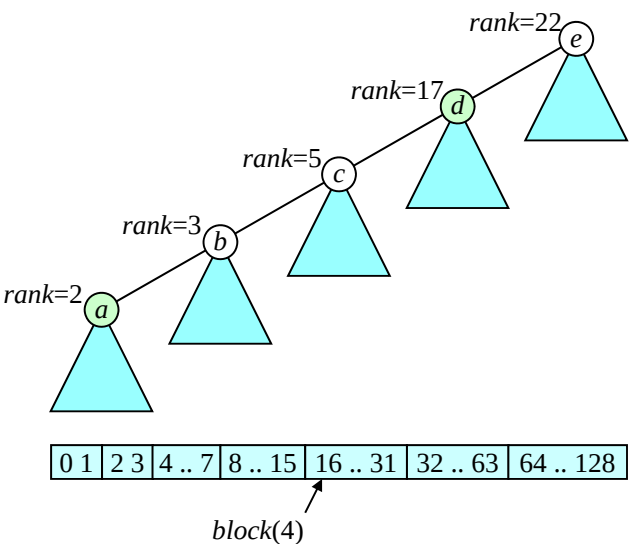

Figure 4: Example showing levels

As before, we divide the analysis into two cases. The first counts the find

steps that occur at singular nodes, while the second counts the find steps that occur at non-singular nodes. For the first case, note that any path from a node to the root of its tree can contain at most one singular node on each of the two levels. Hence, each find operation includes at most two find steps at singular nodes.

We divide the find steps at non-singular nodes into two sub-cases, one for the find steps involving level 1 nodes and another for the find steps involving level 2 nodes. Let's consider the second case first. Suppose x is a non-singular node on level 2. Because x is non-singular, it has some ancestor y that is also on level 2. Because x is on level 2, $rank(p(x))$ is in a different block than $rank(x)$. Similarly, $rank(p(y))$ is in a different block than $rank(y)$. Consequently, if we let z be the root of the tree, then $rank(z)$ is in a different block than $rank(p(x))$. After the find, $z = p(x)$, so the find operation causes $rank(p(x))$ to move into a different (and larger) block than it was in before the find. Since successive blocks double in size, this can happen at most $\lg \lg n$ times before $rank(p(x))$ exceeds $\lg n$, so the number of find steps that can occur at node x while x is a non-singular node on level 2 is at most $\lg \lg n$. Hence, the total number of find steps that occur at non-singular nodes on level 2 is $O(n \log \log n)$.

This leaves the case of non-singular nodes on level 1. Let's suppose that x is such a node and assume that $block(j)$ is the subset of the integers that includes $rank(x)$. A find involving x will assign a new parent to x causing $rank(p(x))$ to increase by at least 1. Since $block(j)$ contains 2^j values, this can happen no more than 2^j times while x is on level 1.

To bound the total number of find steps at non-singular level 1 nodes, we need a bound on the number of nodes in $block(j)$ for each value of j. To get this bound, we need one more piece of information, which is expressed in the following lemma.

Lemma 3 *For any integer $k > 0$, the number of nodes of rank k is at most $n/2^k$. Consequently, the number of nodes with rank $\geq k$ is less than $2n/2^k$.*

Proof. Suppose that when a node x is assigned a rank of k, all nodes in the subtree with root x are labeled (x, k). By Lemma 1, at least 2^k nodes are labeled when x receives a rank of k. If the root of the tree containing x changes, the new root must have rank at least $k + 1$, so no node is labeled more than once with a label of the form (y, k) for a particular value of k. Since there are n nodes, there are at most n labels of the form (y, k) for any particular k and there are at least 2^k such labels for each node of rank k. Hence, at most $n/2^k$ nodes can be assigned a rank of k. Summing $n/2^j$ for

all values of $j \geq k$ yields $2n/2^k$, so the number of nodes with rank $\geq k$ is at most $2n/2^k$. $\square$

Now, since a node with rank in $block(j)$ has rank at least 2^j, the number of nodes in $block(j)$ is at most $2n/2^{2^j}$. Consequently, the number of find steps experienced by non-singular on level 1 with rank in $block(j)$ is at most $2^j \left(2n/2^{2^j}\right)$. There are no non-singular nodes with rank in $block(0)$, so if we sum this expression over all values of $j \geq 1$, we get

$$2n \sum_{j>0} 2^j/2^{2^j} \leq 2n \sum_{j>0} j/2^j \leq 4n$$

The first inequality above holds because every term in the summation on left-hand side is also present in the summation on the right-hand side (along with many additional terms). The second inequality follows directly from the formula for an unbounded arithmetic-geometric series. Hence, the number of find steps at non-singular nodes on level 1 is $O(n)$. Combining this with the results from the other cases we get that the total number of find steps is $O(m + n \log\log n)$.

Before we proceed further, let's recap. In the analysis just completed, we divided the find steps into three categories.

- *Steps at singular nodes.* For each find operation, there are at most two of these, for a total of at most $2m$.
- *Steps at non-singular nodes on level 1.* The total number of these steps is $O(n)$.
- *Steps at non-singular nodes on level 2.* Each vertex experiences at most $\lg\lg n$ of these, for a total of at most $n \lg\lg n$.

To improve on this result, we need to get a better bound on the number of find steps in the last category. We can do that by dividing this case into two sub-cases. We first define a second, coarser partition on the integers, by letting

$$\begin{aligned} block(2,0) &= [0..3] \\ block(2,1) &= [4..15] = \left[2^2..2^{2^2} - 1\right] \\ block(2,2) &= [16..65535] = \left[2^{2^2}..2^{2^{2^2}} - 1\right] \end{aligned}$$

and in general

$$block(2, j) = \left[2^{[j]}..2^{[j+1]} - 1\right]$$

where $2^{[1]} = 2^2$ and $2^{[j]} = 2^{2^{[j-1]}}$ for $j > 1$. For notational consistency, we use $block(1, j)$ to denote our original partition of the integers.

Now, we say that a node x is on level 1 if there is some $block(1, j)$ that contains both $rank(x)$ and $rank(p(x))$. We say that x is on level 2 if it is not on level 1 and there is some $block(2, j)$ that contains both $rank(x)$ and $rank(p(x))$. Nodes that are on neither level 1 nor level 2 are on level 3. So, in Figure 4, nodes a and d are on level 1 and nodes b and c are on level 3. As before, we say a node is singular if none of its ancestors is on the same level that it is. So, in Figure 4, nodes c and d are singular.

As before, we proceed using a case analysis. We have four cases to consider.

- *Steps at singular nodes.* Since there are just three levels, each find operation can have at most three find steps at singular nodes. This gives a total of at most $3m$ find steps in this category.

- *Steps at non-singular nodes on level 1.* Our previous analysis applies directly to this case, so as before, the number of find steps for non-singular nodes on level 1 is $O(n)$.

- *Steps at non-singular nodes on level 2.* We show below that the number of find steps in this group is $O(n)$.

- *Find steps at non-singular nodes on level 3.* We show below that the number of find steps in this group is $O(n\beta(n))$, where $\beta(n)$ is a very slowly growing function.

Let's proceed with the analysis of the third case. Let x be a non-singular node with $rank(x)$ in $block(2, j)$. Because x is on level 2, its parent's rank is in a different level 1 block than its rank. Because it is non-singular, it has a proper ancestor y that is also on level 2. Consequently, the root of the tree has rank that is in a different level 1 block than $rank(p(x))$. This means that a find involving x causes $rank(p(x))$ to move to a different level 1 block. Hence, the number of find steps experienced by x while it is on level 2 and non-singular is less than the number of level 1 blocks that intersect $block(2, j)$, and this at most $2^{[j]} - 2^{[j-1]} < 2^{[j]}$, as illustrated in Figure 5.

Since the number of nodes with rank in $block_2(j)$ is $\leq 2n/2^{2^{[j]}} = 2n/2^{[j+1]}$ for $j > 0$, the number of find steps at non-singular nodes on level 2 is at

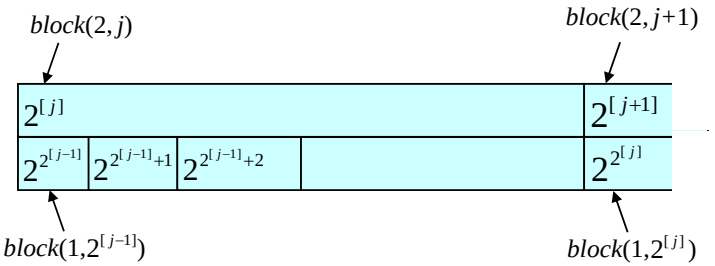

Figure 5: Intersections of $block_1$ and $block_2$

most $2^{[j]}\left(2n/2^{[j+1]}\right)$ for $j > 0$. There can be no non-singular nodes on level 2 with rank in $block(2, 0)$, so, the total number of find steps that occur at non-singular nodes on level 2 is at most

$$2n \sum_{j>0} 2^{[j]}/2^{2^{[j]}} \leq 2n \sum_{j>0} j/2^j \leq 4n$$

That leaves us with just the fourth case. Let x be a non-singular node with $rank(x)$ in $block(2, j)$. Because x is on level 3, its parent's rank is in a different level 2 block than its rank. Because it is non-singular, it has a proper ancestor y that is also on level 3. Consequently, the root of the tree has a rank that is in a different level 2 block than $rank(p(x))$. This means that a find involving x causes $rank(p(x))$ to move to a different level 2 block. This means that the number of find steps that x can experience while it is non-singular and on level 3 is bounded by the number of level 2 blocks that include values $\leq \lg n$. If we define $\beta(n)$ to be the smallest integer $i > 0$ for which $2^{[i]} > \lg n$, then the number of level 2 blocks that includes values $\leq \lg n$ is $\beta(n)$ and the number of find steps at any non-singular node on level 3 is at most $\beta(n)$. The function $\beta(n)$ grows extremely slowly with n; in general $\beta(2^{[k]}) = k$.

Putting together the results from the four cases above, we find that the total number of find steps is $O(m + n\beta(n))$. Now it turns out that even this analysis can be improved, by further sub-dividing the fourth case.

6 The Final Analysis

The key to this final analysis of the number of find steps is to allow the number of distinct cases to grow further. To do this, we define a series of

partitions on the integers, *block*$(1,*)$, *block*$(2,*)$, *block*$(3,*)$, and so forth. Each partition is coarser than the previous one with the last partition requiring relatively few blocks to span the range up to $\lg n$. Specifically, for all $i > 0$, *block*$(i,0)$ starts at 0 and for all $j > 0$, *block*(i,j) starts with $A(i,j)$, where A is *Ackerman's function* and is defined as follows. $A(1,j) = 2^j$ for all $j > 0$, $A(i,1) = A(i-1,2)$ for all $i > 1$ and

$$A(i,j) = A(i-1, A(i,j-1))$$

for all $i > 1$ and $j > 1$. Notice that with this definition, $A(2,j) = 2^{[j]}$ for all $j \geq 1$. Consequently this definition of the blocks is consistent with our earlier definition of *block*$(1,*)$ and *block*$(2,*)$. We also define a functional inverse α of Ackerman's function.

$$\alpha(m,n) = \min\{i \geq 1 | A(i, \lfloor m/n \rfloor) > \lg n\}$$

and note that *block*$(\alpha(m,n), j)$ contains no values $\leq \lg n$ for all $j \geq \lfloor m/n \rfloor$.

We use the blocks to define levels, much as before. Specifically, we say that a non-root node x is on level 1 if *rank*(x) and *rank*$(p(x))$ are both in some *block*$(1,j)$. For $1 < i \leq \alpha(m,n)$, we say that x is on level i if it is not on level $i-1$ and there is some *block*(i,j) that contains both *rank*(x) and *rank*$(p(x))$. Finally, a node x is on level $\alpha(m,n)+1$ if there is no *block*$(\alpha(m,n),j)$ that contains both *rank*(x) and *rank*$(p(x))$. We say that x is *singular* if it has no proper ancestor on the same level as x.

As before, the analysis can be broken down into cases.

- *Steps at singular nodes.* Since there are $\alpha(m,n)+1$ levels, the number of find steps at singular nodes is at most $m(\alpha(m,n)+1)$.
- *Steps at non-singular nodes on level 1.* Our previous analysis applies directly to this case, so the number of find steps in this case is $O(n)$.
- *Steps at non-singular nodes on level i for $1 < i \leq \alpha(m,n)$.* We show below that the number of find steps covered by this case is $O(n)$.
- *Steps at non-singular nodes on level $\alpha(m,n)+1$.* A node can experience at most $\lfloor m/n \rfloor$ find steps while it is non-singular and on level $\alpha(m,n)+1$. Hence the total number of find steps covered by this case is $\leq m$.

Proceeding with the analysis of the third case, let x be a non-singular node on level i ($1 < i \leq \alpha(m,n)$) and assume that *block*(i,j) includes *rank*(x). Because x is on level i, its parent's rank is in a different level $i-1$ block than its rank. Because it is non-singular, it has a proper ancestor

y that is also on level i. Consequently, the root of the tree has rank that is in a different level $i-1$ block than $rank(p(x))$. Hence, a find involving x causes $rank(p(x))$ to move to a different level $i-1$ block. This means that the number of find steps that x can experience while it is non-singular and on level i is bounded by the number of level $i-1$ blocks that intersect $block(i,j)$. Call this number $b_{i,j}$ and note that $b_{i,0}=2$ for all i and that for $j>0$, $b_{i,j} \leq A(i,j)-A(i,j-1) < A(i,j)$, as illustrated in Figure 6.

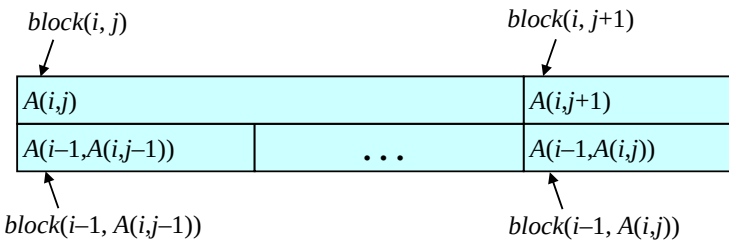

Figure 6: Level $i-1$ blocks that intersect $block(i,j)$

Since the number of nodes with rank in $block(i,j)$ is $\leq 2n/2^{A(i,j)}$, the number of find steps experienced by non-singular nodes on level 2 whose rank is in $block(i,j)$ is at most $A(i,j)\left(2n/2^{A(i,j)}\right)$ for $j>0$. There are no non-singular nodes with rank in $block(i,0)$ for all i, so the total number of find steps experienced by non-singular nodes on levels 2 through $\alpha(m,n)$ is at most

$$\begin{aligned} 2n \sum_{2\leq i\leq \alpha(m,n)} \sum_{j>0} A(i,j)/2^{A(i,j)} &\leq 2n \sum_{2\leq i\leq \alpha(m,n)} \sum_{j>0} j/2^j \\ &\leq 2(\alpha(m,n)-1)n \sum_{j>0} j/2^j \\ &\leq 5(\alpha(m,n)-1)n \end{aligned}$$

Combining the bounds in the four cases gives $O((m+n)\alpha(m,n))$ find steps.

To compare this result with the previous one, we need to compare the growth rates of α and β. Note that α grows most quickly when $m=n$. In this case, $\alpha(n,n)=\min\{i\geq 1 | A(i,1) > \lg n\}$ and applying the definition $\alpha(4,4)=1$, $\alpha(16,16)=\alpha(2^{[2]},2^{[2]})=2$, $\alpha(2^{[3]},2^{[3]})=3$ and $\alpha(2^{[17]},2^{[17]})=4$. Since $\beta(2^{[5]})=4$, we can see that although β is a very slowly growing function, α grows even more slowly. Also, it's worth noting that for $m>$

$n \lg \lg n$, $\alpha(m, n) = 1$. So, when the number of find operations grows even a little bit faster than n, the total number of find steps is $O(m + n)$.