CS 225 - Disjoint Sets

Scribe : Harsha Srimath Tirumala

1 Learning Objectives

↪ Disjoint Sets' significance : MST building

↪ Union-find data structure : Overview

↪ Union by size, Union by height

↪ Union by rank

↪ Path Compression : motivation, analysis

2 Motivation - MST building

Disjoint sets are used to keep track of, well, sets that are disjoint! As a motivating example, consider the task of creating a Minimum Spanning Tree (MST) for a given weighted undirected graph G. Recall that a spanning tree must :

↪ Span the entire vertex set (*connectivity*)

↪ Avoid any cycles (*Tree*)

We build connectivity by adding edges to form the MST that we seek. While adding edges, we need to make sure to avoid cycles as that would break the tree. Specifically, when processing an edge (u, v), we need to check if u and v are already connected - if they are, we must avoid adding the edge (u, v). Since this test is necessary[1], the efficiency of the MST building algorithm depends on being able to efficiently determining if u and v are already connected in the partial MST built so far. This can be done in $O(n)$ time,[2]but that is rather time consuming.

Instead, we will frame the same test in a different sense - we must check if u and v are in the same connected component in the partial MST. If we think of connected components as sets, then we are essentially asking "are u and v in the same disjoint set?" We will now see a data structure that helps keep track of connections and efficiently answers queries on disjoint sets.

3 Union-find data structure

The disjoint set data structure (also called union-find data structure) maintains disjoint sets and supports the following operations -

↪ $makeset(x)$: makes a set containing only x

↪ $find(x)$: returns the representative of the disjoint set containing x

↪ $union(u, v)$: merges the disjoint sets containing u and v if they were not in the same initial disjoint set

Observe that these three operations suffice to support the maintenance of disjoint sets undergoing union operations. $makeset(x)$ is rather straightforward, so we will focus on the more interesting $find(x)$ and $union(u, v)$ operations.

[1]Beware the use of the word necessary - it is quite dangerous to use it without sufficient justification.

[2]How? You tell me! Pick your favorite graph search algorithm..

3.1 Union by size

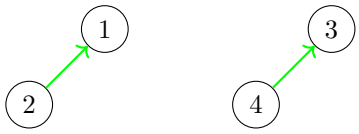
As the name suggests, in this case we implement $union(u, v)$ by having the smaller set point to the larger set. The advantage of doing this is that fewer elements undergo an increase in height (as compared to arbitrarily assigning pointers or having the larger set point to the smaller).

3.1.1 Tie-Break

If both sets are of the same size, we can break ties arbitrarily. In MP-Mazes for example, if union is called on disjoint sets of same size, then the set containing the second parameter points to the set containing the first - i.e. on $union(u, v)$ - the set containing v points to the set containing u.

Let us consider a bunch of union operations and how union-by-size impacts the data structure -

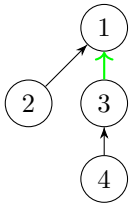
$union(1, 2), union(3, 4)$



1	2	3	4
-2	1	-2	3

On $union(1, 2)$, we first check if they belong to the same disjoint set. Since $find(1) = 1 \neq find(2) = 2$, the union should be executed. $union(1, 2)$ is asking for a union of two sets of equal size ($\{1\}$ and $\{2\}$), we break the tie by having second set point to first. Likewise for $union(3, 4)$. The parent of the root is set to $-|$up-tree$|$ - for example for parent information for 1 and 3.

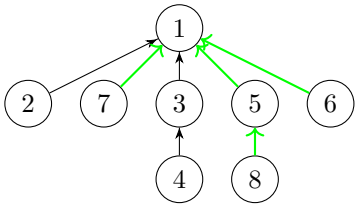
$union(1, 4)$



1	2	3	4
-4	1	1	3

On $union(1, 4)$, we first check if 1 and 4 belong to different sets by calling $find(1)$ and $find(4)$. Since $find(1) = 1$, $find(4) = 3$, and both sets have same size {1,2} and {3,4}, we again break tie by having 4's set (i.e. 3) point to 1's set (i.e. 1).

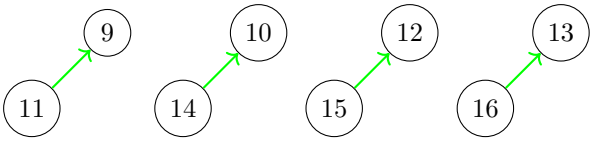
$union(2,4), union(5,8), union(3,6), union(7,3), union(8,2)$



1	2	3	4	5	6	7	8
-8	1	1	3	1	1	1	5

The operation $union(2,4)$ first has to check if 2 and 4 are in the same set. Since $find(2) = find(4) = 1$, 2 and 4 are in the same set {1,2,3,4}, and $union(2,4)$ is not executed. $union(5,8)$ is straightforward. For $union(3,6)$, since $find(3) = 1 \neq find(6) = 6$, and 3 is in the larger set {1,2,3,4}, we have 6 pointing to 1. Likewise for $union(7,3)$. For $union(8,2)$, since $find(8) = 5 \neq find(2) = 1$, and 2 is in the larger set, we have 5 pointing to 1.

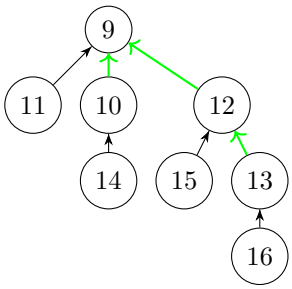
$union(9,11), union(10,14), union(12,15), union(13,16)$



9	10	11	12	13	14	15	16
-2	-2	9	-2	-2	10	12	13

Straightforward unions as was the case with the original bunch.

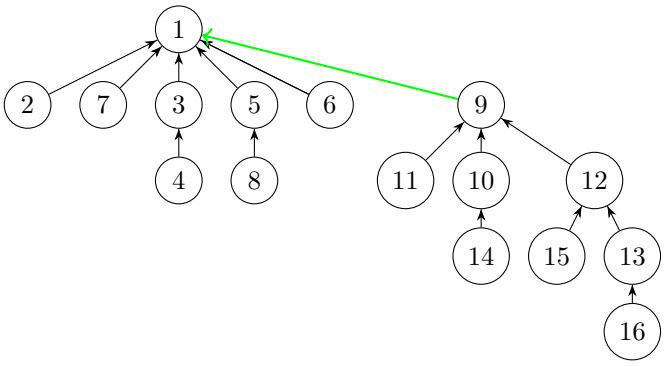
$union(9,14), union(12,16), union(13,15), union(9,12)$



9	10	11	12	13	14	15	16
-8	9	9	9	12	10	12	13

$union(9,14)$ and $union(12,16)$ are similar to $union(1,4)$ on the previous page. $union(13,15)$ does not execute as $find(13) = 12 = find(15)$. For $union(9,12)$, we have $find(9) = 9 \neq find(12) = 12$ - so $union(9,12)$ is executed. Since both sets have equal size (4 each), we break tie by having the second set point to first, i.e. 12's set (12) points to 9's set (9). At this point, the height of the up-tree of 9 has increased to 3.

$union(7, 13)$



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-16	1	1	3	1	1	1	5	1	9	9	9	12	10	12	13

Finally, for $union(7, 13)$, we have $find(7) = 1 \neq find(13) = 9$. Since both sets again have equal sizes (8 each), we break tie by having 13's set (9) point to 7's set (1). Observe that the height of the up-tree of 1 has increased to 4.

Now that we have seen union-by-size in action, it's time to understand the time complexity of operations. $makeset(x)$ takes $O(1)$ time. $find(x)$, $union(u, v)$ operations essentially take the time needed for $find$ - once the "root" of each set is clear, operations (actual union, declining union due to same find etc.) can be done in $O(1)$ time. Observe that the time taken by $find$ is $O(h)$ - where h is the height of the specific up-tree we are looking up.

Claim 1. *Any up-tree of height h has at least 2^h nodes.*

Proof. Before we write the proof, let us think about what we want to say. We want to make a claim about the minimum number of nodes in an up-tree of height h. Can we think of when an up-tree of height h has minimum size? Possibly when it first reaches height h. Good, so when does it first reach height h? Well, when there is a union of *at least two* up-trees each of height $h - 1$. But, what are the minimum sizes of these two?
By now, some of you must be thinking "Induction" while the rest are thinking "recurrence". Indeed, both techniques work. Let's take a look at both :

Induction
Let $p(i)$: Any up-tree of height i has at least 2^i nodes.
Base case - $p(0)$: Any up-tree of height 0 has at least $2^0 = 1$ nodes.
We note that $p(0)$ is trivially true.
By induction hypothesis, we assume that $p(i)$ is true $\forall i \leq k$. And we now need to prove $p(k + 1)$.
$p(k + 1)$: Any up-tree of height $(k + 1)$ has at least 2^{k+1} nodes.
Any up-tree of height $(k + 1)$ contains at least two up-trees each of height (atleast) k - since that is the only way an up-tree of height $(k + 1)$ can come to exist. By Induction hypothesis, these two up-trees each have at least 2^k nodes. So, there are at least $2^k + 2^k = 2^{k+1}$ nodes in any up-tree of height $(k + 1)$. This proves $p(k + 1)$.
By the principle of mathematical induction, $p(i)$ is true for all $i \geq 0$ and we conclude that any up-tree of height h has atleast 2^h nodes.

Recurrence

Let $a(i)$ denote the minimum number of nodes in an up-tree of height i. Since any up-tree of height i contains at least two up-trees of height $(i-1)$, and each of these have atleast $a(i-1)$ nodes, we have :

$$a(i) \geq a(i-1) + a(i-1) = 2a(i-1)$$

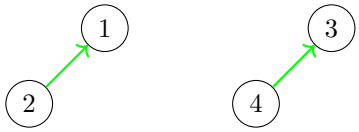
Since $a(0) = 1$, this gives us $a(1) \geq 2a(0) = 2$, $a(2) \geq 2a(1) = 4$ and in particular $a(i) \geq 2^i$. □

3.2 Union by height

As the name suggests, in this case we implement $union(u, v)$ by having the shorter set point to the taller set i.e. the set with lower height pointing to the one with higher height. The advantage of doing this is that it ensures minimal increase in height (as compared to arbitrarily assigning pointers or having the taller set point to the shorter).

Let us consider the same set of union operations we have previously seen and how union-by-height impacts the data structure -

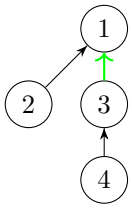
$union(1, 2), union(3, 4)$



1	2	3	4
-2	1	-2	3

On $union(1, 2)$, we first check if they belong to the same disjoint set. Since $find(1) = 1 \neq find(2) = 2$, the union should be executed. $union(1, 2)$ is asking for a union of two sets of equal height ($\{1\}$ and $\{2\}$), we break the tie by having second set point to first. Likewise for $union(3, 4)$.

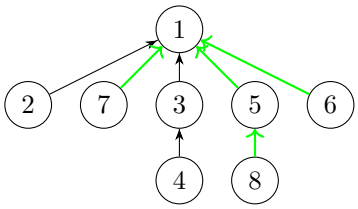
$union(1, 4)$



1	2	3	4
-4	1	1	3

On $union(1, 4)$, we first check if 1 and 4 belong to different sets by calling $find(1)$ and $find(4)$. Since $find(1) = 1$, $find(4) = 3$, and both sets have same height ($\{1,2\}$ and $\{3,4\}$ have height 1), we again break tie by having 4's set (i.e. 3) point to 1's set (i.e. 1).

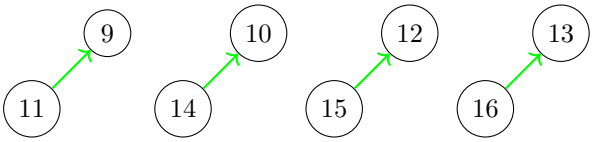
$union(2,4), union(5,8), union(3,6), union(7,3), union(8,2)$



1	2	3	4	5	6	7	8
-8	1	1	3	1	1	1	5

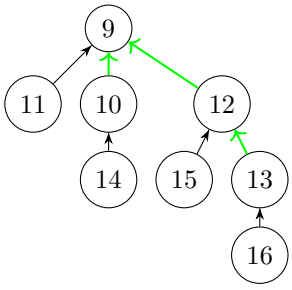
The operation $union(2,4)$ first has to check if 2 and 4 are in the same set. Since $find(2) = find(4) = 1$, 2 and 4 are in the same set {1,2,3,4}, and $union(2,4)$ is not executed. $union(5,8)$ is straightforward. For $union(3,6)$, since $find(3) = 1 \neq find(6) = 6$, and 3 is in the taller set {1,2,3,4}, we have 6 pointing to 1. Likewise for $union(7,3)$. For $union(8,2)$, since $find(8) = 5 \neq find(2) = 1$, and 2 is in the taller set, we have 5 pointing to 1.

$union(9,11), union(10,14), union(12,15), union(13,16)$



Straightforward unions as was the case with the original bunch.

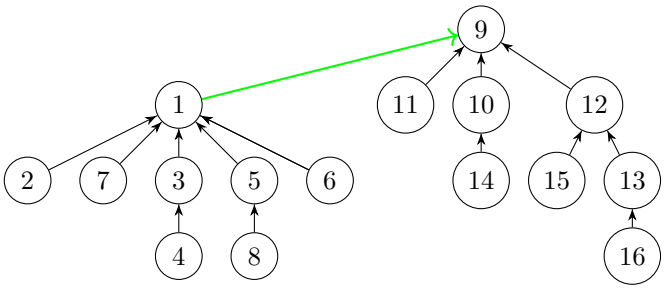
$union(9,14), union(12,16), union(13,15), union(9,12)$



9	10	11	12	13	14	15	16
-2	-2	9	-2	-2	10	12	13

$union(9,14)$ and $union(12,16)$ are similar to $union(1,4)$ on the previous page. $union(13,15)$ does not execute as $find(13) = 12 = find(15)$. For $union(9,12)$, we have $find(9) = 9 \neq find(12) = 12$ - so $union(9,12)$ is executed. Since both sets have equal height (2 each), we break tie by having the second set point to first, i.e. 12's set (12) points to 9's set (9). At this point, the height of the up-tree of 9 has increased to 3.

$union(7, 13)$



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
9	1	1	3	1	1	1	5	-16	9	9	9	12	10	12	13

Finally, for $union(7, 13)$, we have $find(7) = 1 \neq find(13) = 9$. Unlike the previous case, these both sets have different heights - 13's set (9) has height 3 whereas 7's set (1) has height 2. So, 7's set (1) now points to 13's set (9). Observe that the height of the up-tree of 9 has NOT changed while the height of the up-tree of 1 is now 3 (and the up-tree now "belongs" to 9).

Now that we have seen union-by-size in action, it's time to understand the time complexity of operations. Again, we focus on the height of the up-trees as $find, union$ have complexity $O(h)$ for an up-tree of height h.

Claim 2. *Any up-tree of height h following union-by-height has at least 2^h nodes.*

Proof. Essentially the same as union-by-size. Since an up-tree of height h is first formed by union of two up-trees of height *at least* $(h-1)$ each, the same recurrence holds. Let $a(i)$ denote the minimum number of nodes in an up-tree of height i. Then :

$$a(i) \geq 2a(i-1)$$

which gives us

$$a(i) \geq 2^i a(0) = 2^i$$

since $a(0) = 1$. □

4 Time Complexity analysis

Coming back to the big picture, we were looking to build an MST for (say) a graph G with n vertices and m edges. Following either union-by-size or union-by-height, we have seen that any up-tree of height h contains at least 2^h nodes. For $2^h = n$, this gives us $h = O(\log n)$, that is, any up-tree on n nodes has height at most $O(\log n)$ (more precisely, $h \leq \log n$).

Let us analyze time complexity by operation type -

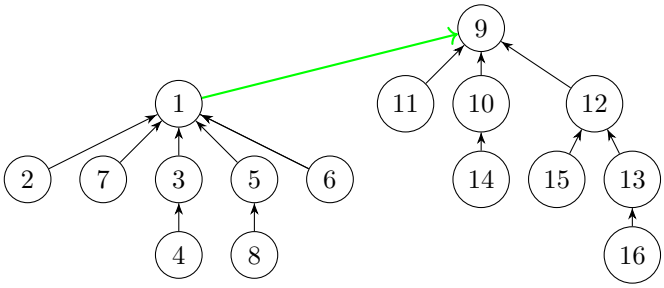
- *makeset* - Since we run *makeset* on each of the n vertices to start with, and each call takes $O(1)$ time, so this takes $O(n)$ time.

- $union(u, v)$ - Since each edge is potentially checked for being added to the partial MST, $union$ is called at most m times. Since $union(u, v)$ first calls $find(u), find(v)$ to test for disjointedness, and each $find$ takes $O(h) = O(\log n)$ in the worst case, the total time taken by these $union$ calls is $O(m)O(\log n) = \mathbf{O(m \log n)}$.

There is a sequence of union operations that cause the overall time complexity to be $\Theta(m \log n)$ - which is not the most efficient implementation for this application. Intuitively, on average, it seems that $O(\log n)$ time is being spent per each $find$, which is actually the worst case cost of $find$ as $h = O(\log n)$. Can we do better? Yes, we can.

5 Path Compression

Consider the state of the data structure as we last saw it. We have an up-tree on 16 nodes of height 3.

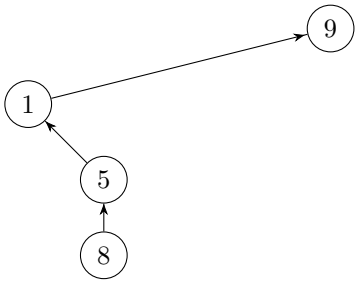


Suppose the next set of operations is -

$union(8, 16), union(8, 13), union(4, 16), union(4, 14), union(1, 17)$

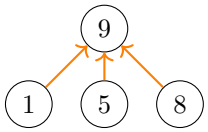
The first four union operations will not be executed as all these nodes belong to the same set. Let us look at the execution of the $find$ operations that lead to this conclusion -

- $union(8, 16)$ - $find(8)$ goes through $8 \to 5 \to 1 \to 9$ while $find(16)$ goes through $16 \to 13 \to 12 \to 9$.
- $union(8, 13)$ - $find(8)$ goes through $8 \to 5 \to 1 \to 9$ while $find(13)$ goes through $13 \to 12 \to 9$.
- $union(4, 16)$ - $find(4)$ goes through $4 \to 3 \to 1 \to 9$ while $find(16)$ goes through $16 \to 13 \to 12 \to 9$.
- $union(4, 14)$ - $find(4)$ goes through $4 \to 3 \to 1 \to 9$ while $find(14)$ goes through $14 \to 10 \to 9$.



Observe that after the first call to $find(8)$, any subsequent call to $find(8)$ can only result in 9 or a node that is a (future) predecessor of 9 as the answer. In particular, there is no need to keep following the entire path $8 \to 5 \to 1 \to 9$

- as 5,1 will NEVER be the answer to $find(8)$. So, we can use **path compression** to compress this path and save on time needed to go through 5,1 en route to 9. In addition, since any calls to $find(5)$ and $find(1)$ can also only result in 9 or a (future) predecessor of 9 as the answer, we can update 5's and 1's parent to 9.
And after path compression -



This will ensure that future calls to $find(8)$ directly go to 9. When do we update the paths in accordance with path compression? Well, whenever a $find(x)$ operation is called and we traverse through the path to reach the up-tree's root, we can update the pointers for all nodes on this path to have them point to the root. This additional work will lead to savings for any future calls to $find$ on x and its ancestors.

5.1 Analysis - Path Compression

The height of an up-tree no longer gives a good approximation for the number of nodes contained in it. Case in point - consider the up-tree of 9 above. Its new height is 1 (after path compression), but it contains 4 elements. In order to get better insight into the size of up-trees, we consider the notion of *rank*. We will also alter our union operations as union-by-height may no longer result in "smaller" up-trees pointing to larger up-trees (due to height changes). We will consider union-by-rank instead of union-by-height to get around the problem of varying heights due to path compression.

5.2 Union by rank

First, after the *makeset* operations, the *rank* of each node is 0. Hereafter, the rule to change *rank* is simple - *rank* changes only when $union(u, v)$ is called on two up-trees with roots of equal rank. Without loss of generality, assume that x and y are the respective roots of these up-trees. Then, we can have v's uptree (y) point to u's uptree (x) and $rank(x)$ increases by 1 (while $rank(y)$ remains unchanged). Does this look familiar? We used the same idea for union by height previously; now, we are using the parameter *rank* instead.

Claim 3. *An up-tree with root (say x) having $rank(x) = k$ has at least 2^k nodes in it.*

Proof. The proof is so similar to claim 2 that we will avoid repeating it and just note that at the instant x became a root with $rank(x) = k$, it must have been because *union* was called on two up-trees of *rank* at least $(k-1)$ each (one of which was x's up-tree). The remainder follows by induction on k. □

Claim 3 immediately implies that *rank* of any node in a n sized universe can be at most $\log_2 n$.

Is it true that any node z with $rank(z) = k$ contains at least 2^k nodes in its "subtree"? Not necessarily, since z may lose some of its descendants after path compression. This makes analysis challenging. Let us first look at a few properties of *rank* :

- **Predecessor** *rank* - For any node u, $rank(u) < rank(\pi(u))$ where $\pi(u)$ is the predecessor of u in any up-tree.

  This is because during any union we have the element with smaller rank point to the element with larger rank (in case of tie, we increase the rank of one of them and make it the predecessor of the other). Note that

even after path compression this property continues to hold because the new predecessor is an ancestor of the original predecessor (and hence has a higher rank).

- *Rank* **count** - If there are n nodes overall, then there are at most $n/2^k$ nodes of *rank* k.

  Since each node of *rank* k had at least 2^k nodes under it at some point (in particular when it first reached *rank* k) and none of the nodes have multiple predecessors, it is impossible to have more than $n/2^k$ nodes of *rank* k.

5.2.1 The Iterative log function - $\log^* n$

↪ Definition of $\log^* n$ -

$$\log^* n = \begin{cases} 0, & \text{if } n \leq 1 \\ 1 + \log^*(\log_2 n), & \text{if } n > 1 \end{cases}$$

↪ $\log^* n$ essentially counts the number of times $\log_2$ must be applied to bring n down to 1 (or some value less than 1).

↪ For example, $\log^*(3) = 2$ as $\log(\log(3)) \leq 1$, $\log^*(10) = 3$ as $\log(\log(\log(10))) \leq 1$ and $\log^*(13) = 3$ as $\log(\log(\log(13))) \leq 1$.

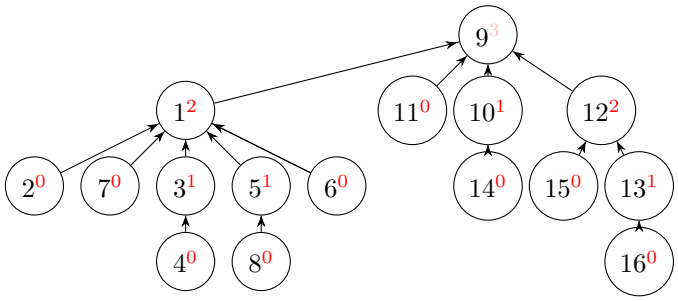
↪ $\log^* 1 = 0, \quad \log^*(2^1) = 1, \quad \log^*(2^2) = 2, \quad \log^*(2^{2^2}) = 3, \quad \log^*(2^{2^{2^2}}) = 4.$

↪ Important property - $\log^*(n)$ depends on n, but it is essentially $O(1)$ for practical considerations as it is a *very very very very* slow growing function. This can be evidenced by considering $n = 10^{80}$[3] which has $\log^*(10^{80}) = 6$. So, for all practical considerations, $\log^* n \leq 6 = O(1)$

Theorem 4. *The overall time complexity of the union-find data structure following union by rank (with path compression) is* $\mathbf{O(m \log^* n + n \log^* n)}$.

Before we prove this theorem, let us understand what this says about the data structure. Since there are n makeset operation and m *union* operations to be performed (total : $m + n$ operation), the amortized time per operation is $O(\log^* n)$ - which we have seen is practically $O(1)$. This is a big improvement on the previous cases without path compression.

Proof. Since the *rank* of a node cannot change once it ceases to be the root, we will fix the *rank* of such nodes to be this particular value. In the diagram below, the fixed *rank* values of all such nodes are indicated in *red*. The node whose *rank* value can potentially change is 9, whose *rank* is indicated in *pink*.



[3] which is the estimated number of atoms in the entire universe (based on [2])

Since *find* operations can cost varying amounts, we have to use a creative counting method to account for the total work done across all *find* operations. The total work done for *find* operations is proportional to the number of pointers followed in the data structure.
Consider *rank* intervals classified by the value of $\log^*(rank)$ as follows -

$$\{1\}, \{2\}, \{3, 4\}, \{5, 6, .., 2^4 = 16\}, \{17, 18, .., 2^{2^4} = 65536\}, \{65537, ..., 2^{65536}\}$$

Each pointer in the data structure is either among *rank* values in the same interval ("same" pointers) or different intervals ("different pointers). We will count the total number of pointers followed based on this classification -

↪ "Different pointers" - Observe that the number of different *rank* intervals is at most $\log^*(n)$. So, on any *find* operation, the number of "Different pointers" followed is at most $O(\log^*(n))$. Since there are at most $2m$ *find* operations in total, the total time taken by "different pointers" is $O(m \log^* n)$.

↪ "Same pointers" - This case is not as easy since there are varying number of "same pointers" for *find* operations. So, we will count these in a different way. For any node z with $rank(z)$ in the interval $\{k+1, k+2, ..., 2^k\}$, assign an amount of 2^k dollars. We will pay for "same pointers" followed by using these dollars. Observe that each time a dollar is paid by a node z, its predecessor changes to one with a *rank* value at least 1 higher than z's original predecessor (due to path compression). That is, before we end up exhausting the 2^k amount held by node z, we will have updated the predecessor of z to be a node with *rank* at least 2^k higher than z's original predecessor. This new predecessor belongs to a different *rank* interval as z's *rank* interval only had $2^k - k$ number of ranks. The outgoing pointer for z is now a "different pointer" (and any subsequent predecessor update for z will continue to be a "different pointer").
Since the dollars paid out account for "same pointers" followed, we are left with counting how many dollars have been paid out.

- For any node z in *rank* interval $[k+1, k+2, ..., 2^k]$, we pay 2^k dollars.
- The number of nodes in *rank* interval $[k+1, k+2, ..., 2^k]$ is at most $2^n/(k+1)$ based on the *Rank count* property. So, the total amount given to this interval is at most $2^k(n/2^{k+1}) < n$.
- Since there are $\log^*(n)$ *rank* intervals and each interval receives at most n dollars, the total number of dollars given out is $n \log^*(n)$.

This concludes that the number of "same" pointers followed overall is at most $O(n \log^* n)$.

So, the total number of pointers followed for the entire sequence of *find* operations is at most $\mathbf{O(m \log^* n + n \log^* n)}$. □

This analysis was based on the section on Path compression in [1].

References

[1] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. 1st ed. USA: McGraw-Hill, Inc., 2006. ISBN: 0073523402.

[2] Planck Collaboration et al. "Planck 2015 results - XIII. Cosmological parameters". In: *AA* 594 (2016), A13. DOI: 10.1051/0004-6361/201525830. URL: https://doi.org/10.1051/0004-6361/201525830.