Union Find with Constant Time Deletions

Seminar on Advanced Algorithms and Data Structures - Student Report

Sandro Rüegge

April 14, 2018

We will have a look at the most interesting aspects of a paper discussing union find with constant time deletions [1]. First we look at a general implementation of the union find data structure and then at what changes have to be made in order to support fast deletions. The figures are color coded. Green nodes and arrows are added, red ones removed. Orange nodes in tree graphs are vacant (a property explained later in the report.)

1 Union Find

A Union Find data structure manages disjoint sets under the following operations. Let v be any element and A, B set references.

- make-set(v): Takes an element, creating a singleton set
- union(A, B): Takes two different set references, unifying the two sets into a new one, destroying the old sets
- find(v): Takes an element and returns a reference to the set it is in

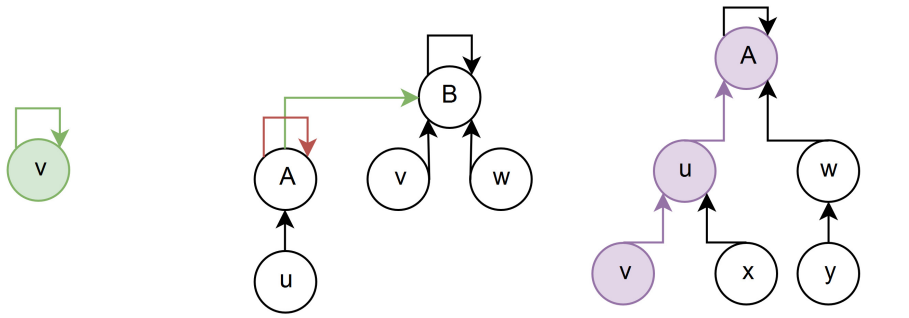
There are different approaches to implementing these operations. For example one can choose to optimize union or find depending on the use case. In this report we will focus on the quick union implementation. We describe each set as a rooted tree. For this we store a parent p(v) for each element in the data structure. The root of the tree is used as the set reference.

Definition 1. $Height(v) = \begin{cases} 0 & \textit{if } v \textit{ is leaf} \\ max\{Height(\omega) | \omega \textit{ is child of } v\} + 1 & \textit{otherwise} \end{cases}$

Definition 2. $Rank(v) \in \mathbb{N}, Rank(v) < Rank(p(v))$
Rank(v) = 0 if v is leaf
Rank(v) = 1 if v is the root of a tree with height 1

We implement the rank calculation for an element in the following way: set to 0 when created with make-set. Increase for the root of the new set after a union, if the two initial sets had equal rank. The above means two things. Firstly on any path from a node to the root the ranks are strictly increasing. Secondly the rank of the root is an upper bound for the height of the tree. We describe the union find operations as follows.

- make-set(v): set $p(v) \leftarrow v$
- union(A, B): A and B are set references and thus root nodes of their respective set trees, If $rank(A) \geq rank(B)$ set $p(B) \leftarrow p(A)$ otherwise $p(A) \leftarrow p(B)$
- find(v): while not $p(v) = v, v \leftarrow p(v)$ afterwards return v. This means that we follow parent pointers up to the root of the set tree.



(a) create singleton set (b) unite two disjoint sets (c) follow parent pointers (in violet) to the root

Figure 1: Visuals of the three union find operations

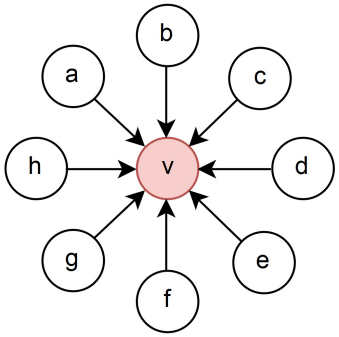


Figure 2: Each node on the circle needs to be reattached to another node

2 Goal

2.1 Example

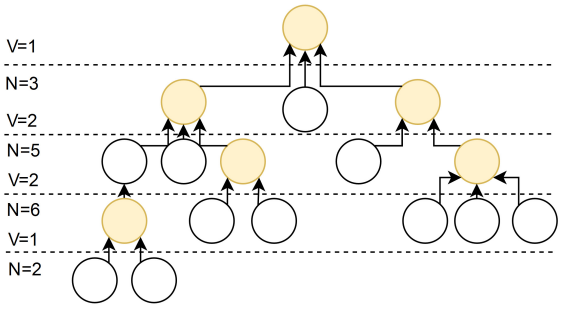
As a motivation we look at an example problem our data structure will be able to solve. In our world new businesses are created every now and then. Sometimes they join into bigger corporations. Occasionally those corporations terminate obsolete business. We formulate a task in which as input one gets either information about the creation of a new business, the union of two corporations, the destruction of a business or a request for affiliation information for a business. A solution program would have to adapt its internal state to the first three input types and return the corporation for the third input.

2.2 Problem

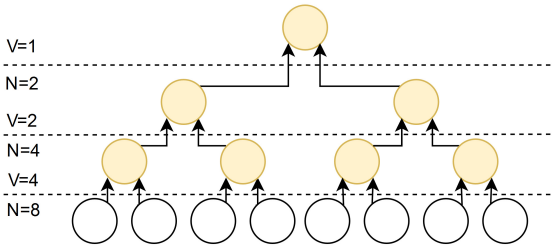
To solve this example a union find sounds like an almost perfect match. But we need a way to remove businesses again. This is no problem if the element is at a leaf position but as we see in Figure 2 we already run into a problem if our tree is a star. If we need to remove the root node, we have to reattach up to n-1 nodes to a new root. This slows our deletion time bounds to linear.

3 Idea

In the paper they present a neat and efficient way to solve the problem by separating the tree nodes from the set elements. This means that we build our tree with nodes to which we can attach an element. We call a node with an element attached to it occupied and vacant



(a) An example of a tidy tree



(b) A vacant node worst case example of a tidy tree

Figure 3: V is the number of vacant and N total nodes in a layer of nodes with equal distance form the root

otherwise. If we want to delete an element we just detach it from its node, preserving the tree structure. With this a new problem arises, we are polluting the tree, meaning that we can arrive at trees with few elements and a lot of vacant nodes. This slows the find operation and consumes large amounts of memory because we can have arbitrarily large trees. As a result we have to periodically clean our structure.

3.1 Tidy Trees

There is an easy way to provide linear space by keeping trees tidy.

Definition 3. *We call a tree tidy if every vacant non-root node has at least two children and every leaf is occupied.*

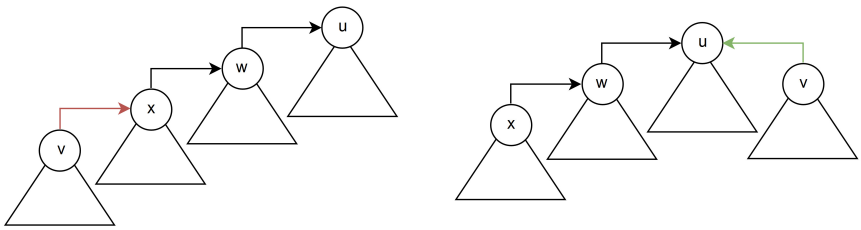
Lemma 1. *In a tidy tree at most half of the nodes are vacant.*

Each higher layer in the tree can have at most half as many vacant nodes as the previous layer had nodes in total. Above lemma follows directly from that. Worst case number of vacant nodes would be as shown in Figure 3b. For the rest of the report we will only operate on tidy trees. We will preserve tidiness with one of the two operations:

- deleting vacant leafs
- shortcutting from a node v, meaning that we give it an ancestor $u \neq p(v)$ as a new parent (Figure 4)

Definition 4. *A shortcut is called bypassing if the new parent is $p(p(v))$.*

Let's have a look at preserving tidiness. Assume we delete an element attached to a node v (making it vacant), we differ two cases. If v is a leaf delete it. If $p(v)$ is occupied, we are done. Otherwise if $p(v)$ is a leaf we can delete it. If it as a single child, we can bypass it. If v is not a leaf, we only have to check if it has a single child and if so bypass it.



(a) Before (b) After

Figure 4: An example of a shortcut from a node v to a node u

Definition 5. *We call a tree reduced if it is a star.*

Whenever we arrive at a reduced tree, we reduce the rank of the root to 1 if it has a child and to 0 otherwise.

3.2 Shallow Trees

Unfortunately tidy trees do not by themselves guarantee sub linear time bounds for find. We need to do some more work for that. In the paper they introduce a measurement they call "value" for each of the nodes and the sets to analyze the effect of deletions. The constants in the formula were selected to satisfy all further equations in the paper. Let A be any set, T_A the set of nodes representing set A and v any node in our data structure.

$$val(v) = \begin{cases} \left(\frac{5}{3}\right)^{rank(v)} & \text{if } v \text{ is occupied} \\ \left(\frac{1}{2}\right)\left(\frac{5}{3}\right)^{rank(v)} & \text{if } v \text{ is vacant} \end{cases}$$

$$val(A) = \sum_{v \epsilon T_A} val(v)$$

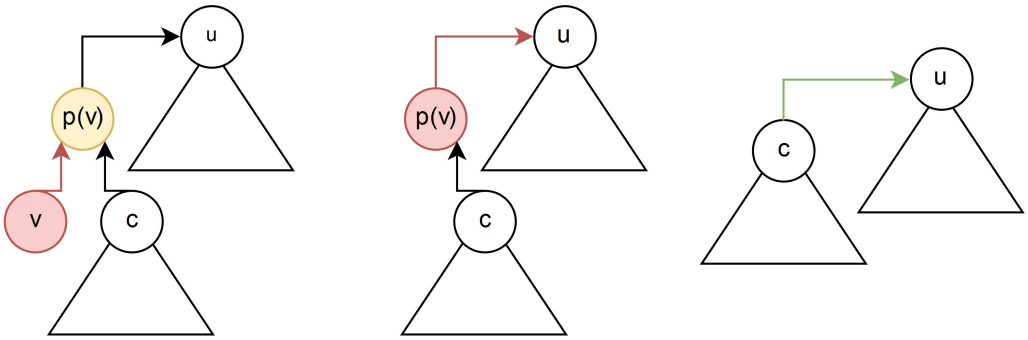
We can now show that for $val(A) \geq 2^{rank(A)}$ the rank of a set is logarithmic in the number of elements in the set. There are precisely |A| elements in a set A and at most |A| vacant nodes in T_A.

$$val(A) \geq |A| \left(\left(\frac{5}{3}\right)^{rank(A)} + \left(\frac{1}{2}\right)\left(\frac{5}{3}\right)^{rank(A)} \right)$$

$$= |A| \left(\frac{3}{2}\right)\left(\frac{5}{3}\right)^{rank(A)}$$

$$\Rightarrow |A| \geq \frac{2^{rank(A)}}{\left(\frac{3}{2}\right)\left(\frac{5}{3}\right)^{rank(A)}} = \left(\frac{2}{3}\right)\left(\frac{6}{5}\right)^{rank(A)}$$

Because the rank is an upper bound for the height of the tree this condition gives us a logarithmic time bound for the find operation. We note here that a reduced tree is trivially shallow because either its rank is zero giving it value $\left(\frac{5}{3}\right)^0 = 2^0$ or it has at least one child and thus value $2 \cdot \left(\frac{5}{3}\right)^1 \geq 2^1$. Let's have a look at the effect the original 3 operations have on the value. Make-set creates a reduced tree which is trivially shallow. Find does not change the structure of the tree. For the union we assume the trees representing two sets are shallow. If the two trees have different ranks the value of the set with higher rank is sufficient to satisfy the shallowness because the rank of the new set is not higher. If they have the same rank then they both had at least value $2^{rank(A)}$ giving us at least $2 \cdot 2^{rank(A)}$ value which is enough for shallowness of the new tree with rank $rank(A) + 1$.



(a) Step 1 (b) Step 2 (c) Step 3

Figure 5: Pictures illustrating the worst case tidying process

3.3 Local Rebuilding

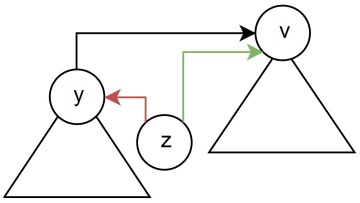
We now know that if we manage to implement deletions without losing value, we reached our primary goal. So let us analyze the loss of value if we delete an element attached to some node v. We get the highest possible loss when v is a leaf and its parent is bypassed (Figure 5). Let $k := rank(p(p(v)))$ then we loose at most $\left(\frac{5}{3}\right)^{k-1}$ for v and $\left(\frac{1}{2}\right)\left(\frac{5}{3}\right)^{k}$ for $p(v)$. The shortcut child of $p(v)$ gains value, namely at least $\left(\frac{1}{2}\right)\left(\left(\frac{5}{3}\right)^{k}-\left(\frac{5}{3}\right)^{k-1}\right)$. This all sums up to a maximum loss of $\left(\frac{9}{10}\right)\left(\frac{5}{3}\right)^{k}$. Assume we have a pointer to some node v in a tree that is not reduced. We also assume that v has grandchildren because otherwise we can get a node that does by moving up the tree at most two layers. For the rebuilding process we need two lists per node. The list $C(v)$ contains the children of v and the list $G(v)$ only the children that themselves have children. These two lists can be conveniently updated as the tree structures change. Let y be one of the nodes in $G(v)$.

- if y is occupied or has more than 2 children we shortcut any of its children to v (Figure 6a)
- if y is vacant with $|C(y)| = 2$
  - if both children are occupied, we shortcut both to v, deleting y
  - if at least one of the children is vacant, say z, then it has at least two children of its own. If it has more, then we can shortcut any child to v (Figure 6c). Otherwise we shortcut one to y and the other to v, deleting z (Figure 6d).

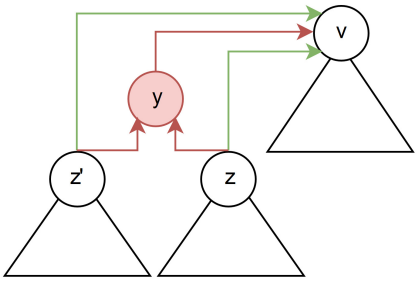
The minimum value we gain can be calculated and is $\frac{7}{50}\left(\frac{5}{3}\right)^{rank(v)}$. This means that to regain the loss of a maximum of $\left(\frac{9}{10}\right)\left(\frac{5}{3}\right)^{k}$ we only need to perform at most 7 rebuilds on nodes with rank at least $rank(v)$. We do this by continuing rebuilding with $p(v)$ when v has no more children with children. It is not always possible but the only trees without grandchildren to perform the operation on are reduced trees for which we already know that they are shallow. This concludes our search for a way to keep trees shallow when we delete elements.

4 Faster Bounds

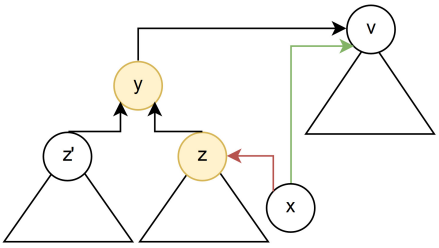
A highly effective optimization of our algorithm would be to perform path compression (Figure 7). This means shortcutting all nodes on a path from a node to the root node. This can be performed during a find operation, the time bound of which already depends on the length of the path thus we are only adding a constant. For union finds without



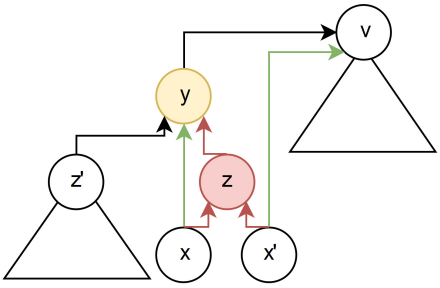
(a) Shortcut any child of y



(b) Shortcut both children of y

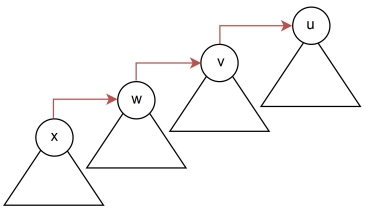


(c) Shortcut a child of z

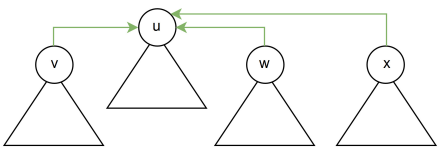


(d) Shortcut both children of z

Figure 6: The four possible rebuilding cases for a node v



(a) Before



(b) After

Figure 7: path compression

deletions there are proofs showing that find with path compression runs in $O(a(|A|))$ amortized time where $a(n)$ is a version of the inverse Ackerman function. The Ackerman function is basically just an extremely fast growing function. Its inverse is thus growing extremely slowly. So slow in fact that for any practical use it can be assumed to be constant. We would like those bounds to apply to our version of the algorithm too. The prove of amortized time is somewhat tedious and done extensively in the paper. We are only giving an intuition by looking at why deletions do not change the already proved bounds of a union find. The potentials for all nodes in the proof are ≥ 0 thus deletions and tidying only decrease the value and have no extra cost on the amortized time. The path compression is the critical part to look at. When shortcutting some vacant nodes on the path their parents might get reduced to one child meaning we have to do tidying. The maximum number of nodes is in O(length of the path) and the time required per node constant. So this is still in the time bounds of the find operation just adding another constant. The tidying does decrease the value of the tree. With some case distinction the paper shows that the value gained by the compression is not smaller than the value lost for tidying afterwards.

References

[1] S. Alstrup, I. L. Gørtz, T. Rauhe, M. Thorup, and U. Zwick. Union-find with constant time deletions. 2005.