Chapter 23

Union-Find

CS 573: Algorithms, Fall 2013
November 14, 2013

23.1 Union-Find

23.2 Union-Find

23.2.1 Requirements from the data-structure

23.2.1.1 Requirements from the data-structure

(A) Maintain a collection of sets.
(B) makeSet(x) - creates a set that contains the single element x.
(C) find(x) - returns the set that contains x.
(D) union(A, B) - returns set = union of A and B. That is A ∪ B.

... merges the two sets A and B and return the merged set.

23.2.2 Amortized analysis

23.2.2.1 Amortized Analysis

(A) Use data-structure as a black-box inside algorithm.
... Union-Find in Kruskal algorithm for computing MST.
(B) Bounded worst case time per operation.
(C) Care: overall running time spend in data-structure.
(D) amortized running-time of operation is the average time it takes to perform an operation on the data-structure.
(E) Amortized time of an operation is \( \frac{\text{overall running time}}{\text{number of operations}} \)
23.2.3 The data-structure

23.2.4 Reversed Trees

23.2.4.1 Representing sets in the Union-Find DS

The Union-Find representation of the sets $A = \{a, b, c, d, e\}$ and $B = \{f, g, h, i, j, k\}$. The set $A$ is uniquely identified by a pointer to the root of $A$, which is the node containing $a$.

23.2.5 Reversed Trees

23.2.5.1 Reversed Trees: $A$

(A) Every element is stored in its own node.

(B) A node has one pointer to its parent.

(C) A set is uniquely identified with the element stored in the root.

(B) makeSet: Create a singleton pointing to itself: $a$

(C) find($x$): Start from node containing $x$, traverse up the tree (using parent pointers), till arriving to root.

Thus, doing a find($x$) operation in the reversed tree shown on the right, involve going up the tree from $x \rightarrow b \rightarrow a$, and returning $a$ as the set.

23.2.6 Union operation in reversed trees

23.2.6.1 Just hang them on each other.

union($a, p$): Merge two sets.

(A) Hanging the root of one tree, on the root of the other.

(B) A destructive operation, and the two original sets no longer exist.
23.2.6.2 Pseudo-code of naive version...

```plaintext
makeSet(x)
\[ p(x) \leftarrow x \]

find(x)
\[
\text{if } x = p(x) \text{ then return } x \\
\text{return find(p(x))}
\]

union(x, y)
\[
A \leftarrow \text{find}(x) \\
B \leftarrow \text{find}(y) \\
p(B) \leftarrow A
\]
```

23.2.7 Example...

23.2.7.1 The long chain

After: makeSet(a), makeSet(b), makeSet(c), makeSet(d), makeSet(e), makeSet(f), makeSet(g), makeSet(h)

\[
\text{union}(g, h) \\
\text{union}(f, g) \\
\text{union}(e, f) \\
\text{union}(d, e) \\
\text{union}(c, d) \\
\text{union}(b, c) \\
\text{union}(a, b)
\]

23.2.7.2 Find is slow, hack it!

find might require \(\Omega(n)\) time.

So, the question is how to further improve the performance of this data-structure. We are going to do this, by using two “hacks”:

(i) **Union by rank**: Maintain for every tree, in the root, a bound on its depth (called rank). Always hang the smaller tree on the larger tree.

(ii) **Path compression**: Since, anyway, we travel the path to the root during a find operation, we might as well hang all the nodes on the path directly on the root.
23.2.7.3 Path compression in action...

(a) The tree before performing find(z), and (b) The reversed tree after performing find(z) that uses path compression.

23.2.7.4 Pseudo-code of improved version...

makeSet(x)
\[\begin{align*}
p(x) &\leftarrow x \\
\text{rank}(x) &\leftarrow 0
\end{align*}\]

find(x)
\[\begin{align*}
\text{if } x \neq p(x) \text{ then } \\
p(x) &\leftarrow \text{find}(p(x)) \\
\text{return } p(x)
\end{align*}\]

union(x, y )
\[\begin{align*}
A &\leftarrow \text{find}(x) \\
B &\leftarrow \text{find}(y) \\
\text{if } \text{rank}(A) > \text{rank}(B) \text{ then } \\
p(B) &\leftarrow A \\
\text{else} \\
p(A) &\leftarrow B \\
\text{if } \text{rank}(A) = \text{rank}(B) \text{ then } \\
\text{rank}(B) &\leftarrow \text{rank}(B) + 1
\end{align*}\]

23.3 Analyzing the Union-Find Data-Structure

23.3.0.5 Definition

Definition 23.3.1. A node in the union-find data-structure is a leader if it is the root of a (reversed) tree.

23.3.0.6 Lemma

Lemma 23.3.2. Once a node stop being a leader (i.e., the node in top of a tree), it can never become a leader again.

Proof: Note, that an element x can stop being a leader only because of a union operation that hanged x on an element y. From this point on, the only operation that might change x parent pointer, is a find operation that traverses through x. Since path-compression can only change the parent pointer of x to point to some other element y, it follows that x parent pointer will never become equal to x again. Namely, once x stop being a leader, it can never be a leader again. \(\blacksquare\)
Lemma 23.3.3. Once a node stop being a leader then its rank is fixed.

Proof: The rank of an element changes only by the union operation. However, the union operation changes the rank, only for elements that are leader after the operation is done. As such, if an element is no longer a leader, than its rank is fixed.

23.3.0.8 Ranks are strictly monotonically increasing

Lemma 23.3.4. Ranks are monotonically increasing in the reversed trees, as we travel from a node to the root of the tree.

Proof...

(A) Claim: \( \forall u \rightarrow v \) in DS: \( \text{rank}(u) < \text{rank}(v) \).

(B) Proof by induction. Base: all singletons. Holds.

(C) Assume claim holds at time \( t \), before an operation.

(D) If operation is union \((A, B)\), and assume that we hanged root\((A)\) on root\((B)\). Must be that \( \text{rank}(\text{root}(B)) \) is now larger than \( \text{rank}(\text{root}(A)) \) (verify!). Claim true after operation!

(E) If operation find: traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \).

By induction, \( \text{rank}(v) > \text{rank} \) of all other nodes of \( \pi \).

All the nodes that get compressed, the rank of their new parent, is larger than their own rank.

23.3.0.10 Trees grow exponentially in size with rank

Lemma 23.3.5. When a node gets rank \( k \) than there are at least \( \geq 2^k \) elements in its subtree.

Proof: The proof is by induction. For \( k = 0 \) it is obvious since a singleton has a rank zero, and a single element in the set. Next observe that a node gets rank \( k \) only if the merged two roots has rank \( k - 1 \). By induction, they have \( 2^{k-1} \) nodes (each one of them), and thus the merged tree has \( \geq 2^{k-1} + 2^{k-1} = 2^k \) nodes.

23.3.0.11 Having higher rank is rare

Lemma 23.3.6. \# nodes that get assigned rank \( k \) throughout execution of Union-Find DS is at most \( n/2^k \).

Proof: Again, by induction. For \( k = 0 \) it is obvious.

Charge a node \( v \) of rank \( k \) to two elements \( u \) and \( v \) of rank \( k - 1 \) that were leaders used to create new larger set.

After the merge \( v \) is of rank \( k \) and \( u \) is of rank \( k - 1 \) and it is no longer a leader. Thus, we can charge this event to the two (no longer active) nodes of degree \( k - 1 \). Namely, \( u \) and \( v \).

By induction: algorithm created at most \( n/2^{k-1} \) nodes of rank \( k - 1 \) \( \implies \) \# nodes of rank \( k \) created by algorithm is \( \leq \left( n/2^{k-1} \right) / 2 = n/2^k \).
23.3.0.12  Find takes logarithmic time

Lemma 23.3.7. The time to perform a single find operation when we perform union by rank and path compression is $O(\log n)$ time.

Proof: The rank of the leader $v$ of a reversed tree $T$, bounds the depth of a tree $T$ in the Union-Find data-structure. By the above lemma, if we have $n$ elements, the maximum rank is $\lg n$ and thus the depth of a tree is at most $O(\log n)$. 

23.3.0.13  $\log^*$ in detail

$\log^*(n)$: number of times one has to take $\lg$ of a number to get a number smaller than two.

Thus, $\log^* 2 = 1$ and $\log^* 2^2 = 2$. Similarly, $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$. Similarly, $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.

Things get really exciting, when one considers

$$\log^* 2^{2^{2^{2^2}}} = \log^* 2^{65536} = 5.$$ 

However, $\log^*$ is a monotone increasing function. And $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$ is a huge number (considerably larger than the number of atoms in the universe). Thus, for all practical purposes, $\log^*$ returns a value which is smaller than 5.

23.3.0.14  Can do much better!

Theorem 23.3.8. If we perform a sequence of $m$ operations over $n$ elements, the overall running time of the Union-Find data-structure is $O((n + m) \log^* n)$.

(A) Intuitively: (in the amortized sense) Union-Find data-structure takes constant time per operation (unless $n$ is larger than $\beta$ which is unlikely).

(B) Not quite correct if $n$ sufficiently large...

23.3.0.15  The tower function...

Definition 23.3.9. Tower($b$) = $2^{\text{Tower}(b-1)}$ and Tower(0) = 1.

Tower($i$): a tower of $2^{2^{2^{2^{2^2}}}}$ of height $i$.

Observe that $\log^*(\text{Tower}(i)) = i$.

Definition 23.3.10. For $i \geq 0$, let Block($i$) = $[\text{Tower}(i - 1) + 1, \text{Tower}(i)]$; that is

$$\text{Block}(i) = \left[z, 2^z - 1\right]$$  for  

$z = \text{Tower}(i - 1) + 1$.

Also Block(0) = [0, 1]. As such, 

Block(0) = [0, 1], Block(1) = [2, 2], Block(2) = [3, 4], Block(3) = [5, 16], Block(4) = [17, 65536], Block(5) = [65537, $2^{65536}$] ...
Running time of find...

(A) RT of \texttt{find}(x) proportional to length of the path from \(x\) to the root of its tree.

(B) ...since start from \(x\) and we visit the sequence:
\[ x_1 = x, x_2 = \texttt{p}(x) = \texttt{p}(x_1), \ldots, x_i = \texttt{p}(x_{i-1}), \ldots, x_m = \text{root of tree}. \]

(C) \( \text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m) \).

(D) RT of \texttt{find}(x) is \( O(m) \).

**Definition 23.3.11.** A node \( x \) is in the \( i \)th block if \( \text{rank}(x) \in \text{Block}(i) \).

(E) Looking for ways to pay for the \texttt{find} operation.

(other two operations take constant time).

Blocks and jumping pointers

(A) maximum rank of node \( v \) is \( O(\log n) \).

(B) # of blocks is \( O(\log^* n) \), as \( O(\log n) \in \text{Block}(c \log^* n) \), (\( c \): constant).

(C) \texttt{find} (\( x \)): \( \pi \) path used.

(D) partition \( \pi \) into each by rank.

(E) Price of \texttt{find} length \( \pi \).

(F) For node \( x \): \( \nu = \text{index}_B(x) \) index of block containing \( \text{rank}(x) \). \( \text{rank}(x) \in \text{Block}(\text{index}_B(x)) \).

(G) \text{index}_B(x): \textit{block of} \( x \)

The path of find operation, and its pointers

(A) During a \texttt{find} operation...

(B) \( \pi \): path traversed.
(C) Ranks of the nodes visited in $\pi$ monotone increasing.
(D) Once leave block $i$th, never go back!
(E) charge visit to nodes in $\pi$ next to element in a different block...
(F) to total number of blocks $\leq O(\log^* n)$.

23.3.0.20 Jumping pointers

Definition 23.3.12. $\pi$: path traversed by $\texttt{find}$. $x \in \pi$, $p(x)$ is in a different block, is a jump between blocks.

Jump inside a block is an internal jump (i.e., $x$ and $p(x)$ are in same block).

23.3.0.21 Not too many jumps between blocks

Lemma 23.3.13. During a single $\texttt{find}(x)$ operation, the number of jumps between blocks along the search path is $O(\log^* n)$.

Proof: Consider the search path $\pi = x_1, \ldots, x_m$, and consider the list of numbers $0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \ldots \leq \text{index}_B(x_m)$. We have that $\text{index}_B(x_m) = O(\log^* n)$. As such, the number of elements $x$ in $\pi$ such that $\text{index}_B(x) \neq \text{index}_B(p(x))$ is at most $O(\log^* n)$.

23.3.0.22 Benefits of an internal jump

(A) $x$ and $p(x)$ are in same block.
(B) $\text{index}_B(x) = \text{index}_B(p(x))$.
(C) $\texttt{find}$ passes through $x$.
(D) $r_{\text{bef}} = \text{rank}(p(x))$ before $\texttt{find}$ operation.
(E) $r_{\text{aft}} = \text{rank}(p(x))$ after $\texttt{find}$ operation.
(F) By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
(G) $\Rightarrow$ parent pointer of $x$ jump forward and the new parent has higher rank.
(H) Charge internal block jumps to this “progress”.

23.3.0.23 Your parent can be promoted only a few times before leaving block

Lemma 23.3.14. At most $|\text{Block}(i)| \leq \text{Tower}(i)$ $\texttt{find}$ operations can pass through an element $x$, which is in the $i$th block (i.e., $\text{index}_B(x) = i$) before $p(x)$ is no longer in the $i$th block. That is $\text{index}_B(p(x)) > i$.

Proof: By above discussion, the parent of $x$ increases its rank every-time an internal jump goes through $x$. Since there at most $|\text{Block}(i)|$ different values in the $i$th block, the claim follows. The inequality $|\text{Block}(i)| \leq \text{Tower}(i)$ holds by definition:

$$\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]$$

23.3.0.24 Few elements are in the bigger blocks

Lemma 23.3.15. There are at most $n/\text{Tower}(i)$ nodes that have ranks in the $i$th block throughout the algorithm execution.
Proof: By lemma, the number of elements with rank in the $i$th block

\[
\leq \sum_{k \in \text{Block}(i)} \frac{n}{2^k} = \sum_{k=\text{Tower}(i)-1+1}^{\text{Tower}(i)} \frac{n}{2^k} = n \cdot \sum_{k=\text{Tower}(i)-1+1}^{\text{Tower}(i)} \frac{1}{2^k} \leq \frac{n}{2^{\text{Tower}(i)-1}} = \frac{n}{\text{Tower}(i)}.
\]

23.3.0.25 Total number of internal jumps is $O(n)$

Lemma 23.3.16. The number of internal jumps performed, inside the $i$th block, during the lifetime of the union-find data-structure is $O(n)$.

Proof: An element $x$ in the $i$th block, can have at most $|\text{Block}(i)|$ internal jumps, before all jumps through $x$ are jumps between blocks, by lemma... There are at most $n/\text{Tower}(i)$ elements with ranks in the $i$th block, throughout the algorithm execution, by other Lemma. Thus, the total number of internal jumps is

\[
|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n.
\]

23.3.0.26 Total number of internal jumps

Lemma 23.3.17. The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.

Proof: Every internal jump can be associated with the block it is being performed in. Every block contributes $O(n)$ internal jumps throughout the execution of the union-find data-structures, by Lemma 23.3.16. There are $O(\log^* n)$ blocks. As such there are at most $O(n \log^* n)$ internal jumps.

23.3.0.27 Result...

Lemma 23.3.18. The overall time spent on $m$ find operations, throughout the lifetime of a union-find data-structure defined over $n$ elements, is $O((m + n) \log^* n)$.

Theorem 23.3.19. If we perform a sequence of $m$ operations over $n$ elements, the overall running time of the Union-Find data-structure is $O((n + m) \log^* n)$. 