Union-Find

Lecture 23
November 14, 2013

Requirements from the data-structure
- Maintain a collection of sets.
- `makeSet(x)` - creates a set that contains the single element `x`.
- `find(x)` - returns the set that contains `x`.
- `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.

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

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 \( a \).

Reversed Trees

1. Every element is stored in its own node.
2. A node has one pointer to its parent.
3. A set is uniquely identified with the element stored in the root.

\textbf{makeSet}: Create a singleton pointing to itself:

\textbf{find}(x): Start from node containing \( x \), traverse up the tree (using parent pointers), till arriving to root.

Thus, doing a \textbf{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.

Union operation in reversed trees

Just hang them on each other.

\textbf{union}(a, p): Merge two sets.

- Hanging the root of one tree, on the root of the other.
- A destructive operation, and the two original sets no longer exist.

Pseudo-code of naive version...

\begin{verbatim}
makeSet(x)
\textbf{p}(x) \leftarrow x

union(x, y)
A \leftarrow \textbf{find}(x)
B \leftarrow \textbf{find}(y)
\textbf{p}(B) \leftarrow A

find(x)
if x = \textbf{p}(x) then return x
return \textbf{find}(\textbf{p}(x))
\end{verbatim}
Example...
The long chain

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

- union(g, h)
- union(f, g)
- union(e, f)
- union(d, e)
- union(c, d)
- union(b, c)
- union(a, b)

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.

Path compression in action...

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

Pseudo-code of improved version...

```
makeSet(x)
    p(x) ← x
    rank(x) ← 0

find(x)
    if x ≠ p(x) then
        p(x) ← find(p(x))
    return p(x)

union(x, y)
    A ← find(x)
    B ← find(y)
    if rank(A) > rank(B) then
        p(B) ← A
    else
        p(A) ← B
        if rank(A) = rank(B) then
            rank(B) ← rank(B) + 1
```

Sariel (UIUC)
Part II

Analyzing the Union-Find Data-Structure

Definition

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

Lemma

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.

Another Lemma

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.
Ranks are strictly monotonically increasing

**Lemma**

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

---

Proof...

1. **Claim:** \( \forall u \rightarrow v \text{ in DS: } \text{rank}(u) < \text{rank}(v) \).
2. **Proof by induction. Base:** all singletons. Holds.
3. **Assume claim holds at time** \( t \), before an operation.
4. If operation is **union** \( (A, B) \), and assume that we hanged \( \text{root}(A) \) on \( \text{root}(B) \).
   Must be that \( \text{rank}(\text{root}(B)) \) is now larger than \( \text{rank}(\text{root}(A)) \) (verify!).
   Claim true after operation!
5. 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.

---

Trees grow exponentially in size with rank

**Lemma**

*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.

---

Having higher rank is rare

**Lemma**

*\# 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 \( \leq (n/2^{k-1})/2 = n/2^k \).
Find takes logarithmic time

**Lemma**

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 \(O(\log n)\).

Can do much better!

**Theorem**

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

- Intuitively: (in the amortized sense) Union-Find data-structure takes constant time per operation (unless \(n\) is larger than \(\beta\) which is unlikely).
- Not quite correct if \(n\) sufficiently large...

\(\log^*\) 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 \(\beta = 2^{2^{2^2}} = 2^{65536}\) 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.

The tower function...

**Definition**

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

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

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

**Definition**

For \(i \geq 0\), let Block\((i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]\); that is \(\text{Block}(i) = [z, 2^{z-1}]\) for \(z = \text{Tower}(i - 1) + 1\).

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

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

- RT of \texttt{find}(x) proportional to length of the path from x to the root of its tree.
- ...since start from x and we visit the sequence:
  \[ x_1 = x, x_2 = p(x) = p(x_1), ..., x_i = p(x_{i-1}), \ldots, x_m = \text{root of tree.} \]
- \( \text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m) \).
- RT of \texttt{find}(x) is \( O(m) \).

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

- Looking for ways to pay for the \texttt{find} operation. (other two operations take constant time).

The path of \texttt{find} operation, and its pointers

- During a \texttt{find} operation...
  - \( \pi \): path traversed.
  - Ranks of the nodes visited in \( \pi \) monotone increasing.
  - Once leave block \( i \)th, never go back!
  - charge visit to nodes in \( \pi \) next to element in a different block...
  - to total number of blocks \( \leq O(\log^* n) \).

Blocks and jumping pointers

- maximum rank of node v is \( O(\log n) \).
- \# of blocks is \( O(\log^* n) \), as \( O(\log n) \in \text{Block}(c \log^* n) \), (c: constant).
- \texttt{find} (x): \( \pi \) path used.
- partition \( \pi \) into each by rank.
- Price of \texttt{find} length \( \pi \).
- For node x: \( \nu = \text{index}_B(x) \) index of block containing \( \text{rank}(x) \). \( \text{rank}(x) \in \text{Block}(\text{index}_B(x)) \).
- \( \text{index}_B(x) \): block of x
Jumping pointers

Definition
π: path traversed by find.

x ∈ π, 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 the same block).

Not too many jumps between blocks

Lemma
During a single 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 in $\pi$ such that $\text{index}_B(x) \neq \text{index}_B(p(x))$ is at most $O(\log^* n)$.

Benefits of an internal jump

- x and p(x) are in the same block.
- $\text{index}_B(x) = \text{index}_B(p(x))$.
- find passes through x.
- $r_{\text{bef}} = \text{rank}(p(x))$ before find operation.
- $r_{\text{aft}} = \text{rank}(p(x))$ after find operation.
- By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
- $\implies$ parent pointer of x jumps forward and the new parent has higher rank.
- Charge internal block jumps to this “progress”.

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

Lemma
At most $|\text{Block}(i)| \leq \text{Tower}(i)$ 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:

$$\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]$$
Few elements are in the bigger blocks

**Lemma**

There are at most $n/Tower(i)$ 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

$$\leq \sum_{k \in Block(i)} \frac{n}{2^k} = \sum_{k=\text{Tower}(i)-1}^{\text{Tower}(i)} \frac{n}{2^k}$$

$$= n \cdot \sum_{k=\text{Tower}(i)-1}^{\text{Tower}(i)} \frac{1}{2^k} \leq \frac{n}{2^{\text{Tower}(i)-1}} = \frac{n}{\text{Tower}(i)}.$$

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

**Lemma**

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 $n/Tower(i)$ elements with ranks in the $i$th block, throughout the algorithm execution, by other Lemma. Thus, the total number of internal jumps is

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

Total number of internal jumps

**Lemma**

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.

Result...

**Lemma**

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**

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