9 Union Find

Union Find Data Structure \( \mathcal{P} \): Maintains a partition of disjoint sets over elements.

- \( \mathcal{P}. \text{makeset}(x) \): Given an element \( x \), adds \( x \) to the data-structure and creates a singleton set that contains only this element. Returns a locator/handle for \( x \) in the data-structure.
- \( \mathcal{P}. \text{find}(x) \): Given a handle for an element \( x \); find the set that contains \( x \). Returns a representative/identifier for this set.
- \( \mathcal{P}. \text{union}(x, y) \): Given two elements \( x \), and \( y \) that are currently in sets \( S_x \) and \( S_y \), respectively, the function replaces \( S_x \) and \( S_y \) by \( S_x \cup S_y \) and returns an identifier for the new set.

Applications:
- Keep track of the connected components of a dynamic graph that changes due to insertion of nodes and edges.
- Kruskals Minimum Spanning Tree Algorithm

List Implementation
- The elements of a set are stored in a list; each node has a backward pointer to the head.
- The head of the list contains the identifier for the set and a field that stores the size of the set.

Algorithm 16 Kruskal-MST\((G = (V, E), w)\)

1: \( A \leftarrow \emptyset; \)
2: for all \( v \in V \) do
3: \( v. \text{set} \leftarrow \mathcal{P}. \text{makeset}(v. \text{label}) \)
4: sort edges in non-decreasing order of weight \( w \)
5: for all \( (u, v) \in E \) in non-decreasing order do
6: if \( \mathcal{P}. \text{find}(u. \text{set}) \neq \mathcal{P}. \text{find}(v. \text{set}) \) then
7: \( A \leftarrow A \cup \{(u, v)\} \)
8: \( \mathcal{P}. \text{union}(u. \text{set}, v. \text{set}) \)

- \( \text{makeset}(x) \) can be performed in constant time.
- \( \text{find}(x) \) can be performed in constant time.
List Implementation

union(x, y)

- Determine sets $S_x$ and $S_y$.
- Traverse the smaller list (say $S_y$), and change all backward pointers to the head of list $S_x$.
- Insert list $S_y$ at the head of $S_x$.
- Adjust the size-field of list $S_x$.
- Time: $\min\{|S_x|, |S_y|\}$.

Running times:

- $\text{find}(x)$: constant
- $\text{makeset}(x)$: constant
- $\text{union}(x, y)$: $O(n)$, where $n$ denotes the number of elements contained in the set system.
Lemma 1
The list implementation for the ADT union find fulfills the following amortized time bounds:

- \textbf{find}(x): O(1).
- \textbf{makeset}(x): O(\log n).
- \textbf{union}(x, y): O(1).

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.
- Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.
- Whenever for an operation the actual cost exceeds the amortized time bound, the difference is charged to bank accounts of some of the elements involved.
- If we can find a charging scheme that guarantees that balances always stay positive the amortized time bounds are proven.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the \textit{excess} to the elements involved.
- In total we will charge at most \( O(\log n) \) to an element (regardless of the request sequence).
- For each element a \textbf{makeset} operation occurs as the first operation involving this element.
- We inflate the amortized cost of the \textbf{makeset}-operation to \( \Theta(\log n) \), i.e., at this point we fill the bank account of the element to \( \Theta(\log n) \).
- Later operations charge the account but the balance never drops below zero.

List Implementation

\textbf{makeset}(x): The actual cost is \( O(1) \). Due to the cost inflation the amortized cost is \( O(\log n) \).

\textbf{find}(x): For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: \( O(1) \).

\textbf{union}(x, y):
- If \( S_x = S_y \) the cost is constant; no bank accounts change.
- Otw. the actual cost is \( O(\min(|S_x|, |S_y|)) \).
- Assume wlog. that \( S_x \) is the smaller set; let \( c \) denote the hidden constant, i.e., the actual cost is at most \( c \cdot |S_x| \).
- Charge \( c \) to every element in set \( S_x \).
List Implementation

Lemma 2
An element is charged at most \( \lfloor \log_2 n \rfloor \) times, where \( n \) is the total number of elements in the set system.

Proof.
Whenever an element \( x \) is charged the number of elements in \( x \)'s set doubles. This can happen at most \( \lfloor \log n \rfloor \) times.

Implementation via Trees

- Maintain nodes of a set in a tree.
- The root of the tree is the label of the set.
- Only pointer to parent exists; we cannot list all elements of a given set.

Example:

Set system \{2, 5, 10, 12\}, \{3, 6, 7, 8, 9, 14, 17\}, \{16, 19, 23\}.

Implementation via Trees

\[ \text{makeset}(x) \]
- Create a singleton tree. Return pointer to the root.
- Time: \( O(1) \).

\[ \text{find}(x) \]
- Start at element \( x \) in the tree. Go upwards until you reach the root.
- Time: \( O(\text{level}(x)) \), where \( \text{level}(x) \) is the distance of element \( x \) to the root in its tree. Not constant.

\[ \text{union}(x, y) \]
- Perform \( a \leftarrow \text{find}(x); b \leftarrow \text{find}(y) \). Then: \( \text{link}(a, b) \).
- \( \text{link}(a, b) \) attaches the smaller tree as the child of the larger.
- In addition it updates the size-field of the new root.
- Time: constant for \( \text{link}(a, b) \) plus two find-operations.
Implementation via Trees

**Lemma 3**
The running time (non-amortized!!!) for \( \text{find}(x) \) is \( O(\log n) \).

**Proof.**
- When we attach a tree with root \( c \) to become a child of a tree with root \( p \), then \( \text{size}(p) \geq 2 \text{size}(c) \), where \( \text{size} \) denotes the value of the size-field right after the operation.
- After that the value of \( \text{size}(c) \) stays fixed, while the value of \( \text{size}(p) \) may still increase.
- Hence, at any point in time a tree fulfills \( \text{size}(p) \geq 2 \text{size}(c) \), for any pair of nodes \( (p, c) \), where \( p \) is a parent of \( c \).

Path Compression

**find(x):**
- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

\[ \begin{array}{c}
\text{Node} & \text{Size} \\
6 & 2 \\
5 & 1 \\
9 & 1 \\
7 & 1 \\
10 & 2 \\
1 & 2 \\
3 & 5 \\
\end{array} \]

- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Asymptotically the cost for a find-operation does not increase due to the path compression heuristic.

However, for a worst-case analysis there is no improvement on the running time. It can still happen that a find-operation takes time \( O(\log n) \).
Amortized Analysis

Definitions:
▶ \( \text{size}(v) \) = the number of nodes that were in the sub-tree rooted at \( v \) when \( v \) became the child of another node (or the number of nodes if \( v \) is the root).
Note that this is the same as the size of \( v \)'s subtree in the case that there are no find-operations.
▶ \( \text{rank}(v) = \lfloor \log(\text{size}(v)) \rfloor \).
▶ \( \Rightarrow \text{size}(v) \geq 2^{\text{rank}(v)} \).

Lemma 4
The rank of a parent must be strictly larger than the rank of a child.

Amortized Analysis

We define
\[
tow(i) := \begin{cases} 
1 & \text{if } i = 0 \\
2^{tow(i-1)} & \text{otherwise} 
\end{cases}
\]

and
\[
\log^*(n) := \min\{i \mid tow(i) \geq n\}.
\]

Theorem 6
Union find with path compression fulfills the following amortized running times:
▶ \( \text{makeset}(x) : O(\log^*(n)) \)
▶ \( \text{find}(x) : O(\log^*(n)) \)
▶ \( \text{union}(x,y) : O(\log^*(n)) \)

Amortized Analysis

Lemma 5
There are at most \( n/2^s \) nodes of rank \( s \).

Proof.
▶ Let’s say a node \( v \) sees node \( x \) if \( v \) is in \( x \)'s sub-tree at the time that \( x \) becomes a child.
▶ A node \( v \) sees at most one node of rank \( s \) during the running time of the algorithm.
▶ This holds because the rank-sequence of the roots of the different trees that contain \( v \) during the running time of the algorithm is a strictly increasing sequence.
▶ Hence, every node sees at most one rank \( s \) node, but every rank \( s \) node is seen by at least \( 2^s \) different nodes.

Amortized Analysis

In the following we assume \( n \geq 2 \).

rank-group:
▶ A node with rank \( \text{rank}(v) \) is in rank group \( \log^*(\text{rank}(v)) \).
▶ The rank-group \( g = 0 \) contains only nodes with rank 0 or rank 1.
▶ A rank group \( g \geq 1 \) contains ranks \( tow(g-1) + 1, \ldots, tow(g) \).
▶ The maximum non-empty rank group is \( \log^*((\log n)) \leq \log^*(n) - 1 \) (which holds for \( n \geq 2 \)).
▶ Hence, the total number of rank-groups is at most \( \log^* n \).
Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node $v$

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from $v$ to $\text{parent}[v]$ as follows:

- If $\text{parent}[v]$ is the root we charge the cost to the find-account.
- If the group-number of $\text{rank}(v)$ is the same as that of $\text{rank}(\text{parent}[v])$ (before starting path compression) we charge the cost to the node-account of $v$.
- Otherwise we charge the cost to the find-account.

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).
- After a node $v$ is charged its parent-edge is re-assigned. The rank of the parent strictly increases.
- After some charges to $v$ the parent will be in a larger rank-group. $\Rightarrow v$ will never be charged again.
- The total charge made to a node in rank-group $g$ is at most $\text{tow}(g) - \text{tow}(g-1) - 1 \leq \text{tow}(g)$.

What is the total charge made to nodes?

- The total charge is at most

$$\sum_g n(g) \cdot \text{tow}(g),$$

where $n(g)$ is the number of nodes in group $g$.

For $g \geq 1$ we have

$$n(g) \leq \sum_{s=\text{tow}(g-1)+1}^{\text{tow}(g)} \frac{n}{2^s} \leq \sum_{s=\text{tow}(g-1)+1}^{\infty} \frac{n}{2^s} = \frac{n}{2^{\text{tow}(g-1)+1}} \cdot 2 = \frac{n}{2^{\text{tow}(g-1)}} = \frac{n}{\text{tow}(g)}.$$ 

Hence,

$$\sum_{g} n(g) \cdot \text{tow}(g) \leq n(0) \cdot \text{tow}(0) + \sum_{g=1} n(g) \cdot \text{tow}(g) \leq n \log^*(n).$$
Amortized Analysis

Without loss of generality we can assume that all makeset-operations occur at the start. This means if we inflate the cost of makeset to $\log^* n$ and add this to the node account of $v$ then the balances of all node accounts will sum up to a positive value (this is sufficient to obtain an amortized bound).

Amortized Analysis

The analysis is not tight. In fact it has been shown that the amortized time for the union-find data structure with path compression is $O(\alpha(m, n))$, where $\alpha(m, n)$ is the inverse Ackermann function which grows a lot lot slower than $\log^* n$. (Here, we consider the average running time of $m$ operations on at most $n$ elements).

There is also a lower bound of $\Omega(\alpha(m, n))$.

Amortized Analysis

$A(x, y) = \begin{cases} 
  y + 1 & \text{if } x = 0 \\
  A(x - 1, 1) & \text{if } y = 0 \\
  A(x - 1, A(x, y - 1)) & \text{otherwise}
\end{cases}$

$\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log n\}$

- $A(0, y) = y + 1$
- $A(1, y) = y + 2$
- $A(2, y) = 2y + 3$
- $A(3, y) = 2^{y+3} - 3$
- $A(4, y) = 2^{2^{2^2y+3}} - 3$ $y+3$ times

Union Find

Bibliography


Union find data structures are discussed in Chapter 21 of [CLRS90b] and [CLRS90c] and in Chapter 22 of [CLRS90a]. The analysis of union by rank with path compression can be found in [CLRS90a] but neither in [CLRS90b] nor in [CLRS90c]. The latter books contains a more involved analysis that gives a better bound than $O(\log^* n)$.

A description of the $O(\log^* n)$ bound can also be found in Chapter 4.8 of [AHU74].