Datenstrukturen und Effiziente Algorithmen

Disjoint Set Data Structure

**Disjoint set data structure**

- maintains a set \( S = \{S_1, \ldots, S_k\} \) of disjoint sets (\( S_i \cap S_j = \emptyset \) for \( i \neq j \))
- \( S \) is dynamic: it can be changed by

**Disjoint set operations**

- **MAKE-SET(\( x \))**
  For an object \( x \), create a new set with only member \( x \).

- **UNION(\( x, y \))**
  Unite the two sets that contain elements \( x \) and \( y \). Let \( S_x, S_y \in S \) denote the two sets that contain \( x \) and \( y \), respectively. We require \( S_x \neq S_y \). Then update the set \( S \):

  \[
  S \leftarrow (S \setminus \{S_x, S_y\}) \cup \{S_x \cup S_y\}
  \]

- **FIND-SET(\( x \))**
  Return a pointer to the representative of the set containing object \( x \).
Disjoint Set Data Structure

**Representatives**

- each set $S$ will be identified through a representative element $x \in S$
- the representative is arbitrary but must remain the same until the set is removed from $S$

Actually, instead of removing $S_x$ and $S_y$ during $\text{UNION}(x, y)$, we will replace $S_x$ with $S_x \cup S_y$ and delete $S_y$ or vice versa.)

**Number of operations for time analyses**

- $n$: number of $\text{MAKE-SET}$ operations = total number of elements
- first $n$ operations are $\text{MAKE-SET}$
- $m \geq n$: total number of operations ($\text{MAKE-SET}$, $\text{UNION}$ or $\text{FIND-SET}$)
- number of $\text{UNION}$ operations is at most $n - 1$ (Why?)
Application for Disjoint Sets: Connected Components

Connected components

The **connected components** of an undirected graph are the equivalence classes under the “is reachable from” relation on the vertices.

Example 1

![Diagram of connected components](image)

Connected components:

\{1, 5, 7\},
\{2, 3, 6, 9\},
\{4, 8\},
\{10\}
**Application for Disjoint Sets: Connected Components**

**CONNECTED-COMPONENTS**($G$)

Computes the connected components of an undirected graph $G = (G.V, G.E)$.

1: **for all** vertices $v \in G.V$ **do**
2: \hspace{1em} **MAKE-SET**($v$)
3: **for all** edges $\{u, v\} \in G.E$ **do**
4: \hspace{1em} **if** ** FIND-SET**(u) ≠ **FIND-SET**(v) **then**
5: \hspace{2em} **UNION**(u, v)

**Example 2**

- **edge processed in line 4**
- **set $S$ of disjoint sets**

<table>
<thead>
<tr>
<th>edge</th>
<th>set $S$ of disjoint sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>${2, 3}$</td>
<td>${1}, {2, 3}, {4, 5, 6, 7, 8, 9, 10}$</td>
</tr>
<tr>
<td>${4, 8}$</td>
<td>${1}, {2, 3}, {4, 5, 6, 7, 8, 9, 10}$</td>
</tr>
<tr>
<td>${1, 5}$</td>
<td>${1, 5}, {2, 3}, {4, 8}, {6, 6}, {9, {10}$</td>
</tr>
<tr>
<td>${6, 9}$</td>
<td>${1, 5}, {2, 3}, {4, 6, 7, 9, 10}$</td>
</tr>
<tr>
<td>${1, 7}$</td>
<td>${1, 5, 7}, {2, 3}, {4, 8}, {6, 9}, {10}$</td>
</tr>
<tr>
<td>${2, 6}$</td>
<td>${1, 5, 7}, {2, 3, 6, 9}, {4, 8}, {10}$</td>
</tr>
<tr>
<td>${2, 9}$</td>
<td>${1, 5, 7}, {2, 3, 6, 9}, {4, 8}, {10}$</td>
</tr>
</tbody>
</table>
Application for Disjoint Sets: Connected Components

**SAME-COMPONENT(\(u, v\))**

Answers whether vertices \(u\) and \(v\) are in the same connected component.

1: if FIND-SET\((u)\) = FIND-SET\((v)\) then
2: return TRUE
3: else
4: return FALSE

This procedure can be called after preprocessing with CONNECTED-COMPONENTS.

**Application: minimum spanning trees**

Above operations for connected components of a graph are required by Kruskals algorithm that finds a minimum spanning tree in a weighted graph.
Sets as linked lists

- Each set $S$ is represented by an object that includes
  - The *head* pointer to the head of a linked list containing the elements of $S$
  - The *tail* pointer to the last element of that list.
  - (The number of elements $|S|$ in the list: *size*)
- Each list element contains
  - the object
  - a pointer to the *next list* element
  - a pointer *back* to the set object

Example 3
Linked Lists as Data Structure for Disjoint Sets

Example 4 (Disjoint sets as linked lists)

In panel (b) the union of the sets in (a) is shown: \( S_1 \leftarrow S_1 \cup S_2 \)
Linked Lists as Data Structure for Disjoint Sets

### Operations

- **MAKE-SET(x)** creates a new set object with a list with \(\text{head} = \text{tail} = x\) in time \(O(1)\).
- **FIND-SET(x)** returns \(x.\text{back}.\text{head}\), i.e. the first element of the list as representative in time \(O(1)\).
- **UNION(u, v)**
  
  Append the list of \(v\) to the list of \(u\):

  1. \(u.\text{back}.\text{tail}.\text{next} \leftarrow v.\text{back}.\text{head}\)
  2. **for** \((p = v.\text{back}.\text{head}; p \neq \text{NULL}; p \leftarrow p.\text{next} \text{ do})\)
  3. \(p.\text{back} = u.\text{back}\)
  4. \(u.\text{back}.\text{tail} \leftarrow v.\text{back}.\text{tail}\)
  5. remove \(v.\text{back}\) from \(S\)
Linked Lists as Data Structure for Disjoint Sets

**Running time of \( \text{UNION}(u, v) \)**

- The running time is proportional to the length of the list of \( v \): \( O(v.\text{back}.\text{size}) \).
- The worst-case running time is quadratic. Consider the following sequence of \( m = 2n - 1 \) operations: \( \text{MAKE-SET}(1), \ldots, \text{MAKE-SET}(n) \), \( \text{UNION}(n - 1, n) \), \( \text{UNION}(n - 2, n - 1) \), \ldots, \( \text{UNION}(1, 2) \)
- This sequence requires \( 1 + 2 + \cdots + n - 1 = \Theta(n^2) \) updates to \( \text{back} \) pointers plus \( \Theta(n) \) time for other updates. Therefore \( m \) operations require also \( \Theta(m^2) \) time.
Linked Lists as Data Structure for Disjoint Sets

A heuristic speedup for UNION

- The running time of $\text{UNION}(u, v)$ is dominated by length of second list.
- Idea: switch roles of $u$ and $v$, if set of $v$ is larger than set of $u$: Append shorter list to longer list.

Theorem 5

Using the linked-list representation of disjoint sets and above heuristics, a sequence of $m \text{MAKE-SET}$, $\text{UNION}$ and $\text{FIND-SET}$ operations, $n$ of which are $\text{MAKE-SET}$ operations, takes time $O(m + n \log n)$. 
Proof.

We first bound the number of times the `back` pointer is updated in line 3 of the pseudo code of `UNION`. `UNION` is called at most \( n - 1 \) times, because each time it is called the number of sets in \( S \) decreases by 1. Consider an arbitrary but fixed object \( x \). Each time \( x.back \) is updated, \( x \) is element of the smaller of the two merged lists. Therefore, after \( x.back \) has been updated for the \( k \)th time, \( x \) is in a list of at least \( 2^k \) elements. As any list has at most \( n \) elements, we must have that \( k \leq \log_2 n \). Therefore, the total number of `back` pointer updates is at most \( n \log_2 n \). All other required time is \( \Theta(1) \) per operation, and therefore amounts to a total of \( \Theta(m) \). Summarizing, we get a running time of \( O(m + n \log n) \).
Forests as Data Structure for Disjoint Sets

Disjoint set forests

- each set is represented by a rooted tree
- the vertices of the tree are the elements of the set
- each vertex $x$ has a pointer $x.p$ to its parent
- the representative of each set is the root of the tree (and has itself as parent)
- $\text{MAKE-SET}(x)$ creates a tree with just the node $x$ in time $O(1)$.
- $\text{FIND-SET}(x)$ recursively follows the parent pointers of $x$ until the root has been reached. The sequence of nodes traversed is called the find path.
- $\text{UNION}(u, v)$ the root of one tree is made the parent of the root of the other tree

Example 6

(chalk board)
Forests as Data Structure for Disjoint Sets

Nothing gained yet over linked lists

Without further effort, the running time would be quadratic just as in the naive linked list approach. When \textsc{UNION}s are done in the “wrong” order, then a tree may become a linear chain of \(n\) nodes and a single \textsc{FIND-SET} on the leaf then takes \(O(n)\) time.

Union by rank heuristic

Idea:
Keep height of the trees small by making the root of the “smaller” tree point to the root of the “larger” tree. An attribute \(x.\text{rank}\) of each node \(x\) will be an upper bound on the height of the node.

Path compression heuristic

Reduce tree heights by making all nodes on a find path point directly to the root. (chalk board)
## Make-Set($x$)

1: $x.rank \leftarrow 0$
2: $x.p \leftarrow x$

## Find-Set($x$)

1: if $x \neq x.p$ then
2: $x.p \leftarrow \text{Find-Set}(x.p)$
3: return $x.p$

---

## Link($x$, $y$)

1: if $x.rank > y.rank$ then
2: $y.p \leftarrow x$
3: else
4: $x.p \leftarrow y$
5: if $x.rank = y.rank$ then
6: $y.rank \leftarrow y.rank + 1$

## Union($x$, $y$)

1: $\text{Link}(\text{Find-Set}(x), \text{Find-Set}(y))$

---

**makes 2 passes:**

1. one pass up to find the root
2. one pass down to update each parent on the find path
### Example 7

<table>
<thead>
<tr>
<th>operations</th>
<th>disjoint set forest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>MAKE-SET(1), . . . , MAKE-SET(9)</strong></td>
<td><img src="image" alt="MAKE-SET" /></td>
</tr>
<tr>
<td>2. <strong>UNION(1,2), UNION(3,4), UNION(5,6), UNION(7,8)</strong></td>
<td><img src="image" alt="UNION" /></td>
</tr>
<tr>
<td>3. <strong>UNION(1,4), UNION(6,8)</strong></td>
<td><img src="image" alt="UNION" /></td>
</tr>
<tr>
<td>4. <strong>UNION(4,8)</strong></td>
<td><img src="image" alt="UNION" /></td>
</tr>
<tr>
<td>5. <strong>UNION(1,9)</strong></td>
<td><img src="image" alt="UNION" /></td>
</tr>
</tbody>
</table>
Time Analysis of Forests

Functional iteration notation

For a function $f$, let $f^{(i)}(n)$ denote the value, if $f$ is applied $i$ times iteratively on $n$. Formally,

$$f^{(i)}(n) := \begin{cases} 
n & \text{if } i = 0, \\
 f(f^{(i-1)}(n)) & \text{if } i > 0.
\end{cases}$$

Example 8

$$\log_{2}^{(3)} 2^{256} = \log_{2}(\log_{2}(\log_{2} 2^{2^{2^{3}}})) = 3$$
Time Analysis of Forests

Ackermann-like function

For integers $k \geq 0$ and $j \geq 1$ define the function $A_k(j)$ as

$$A_k(j) := \begin{cases} j + 1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1. \end{cases}$$

$k$ will be called the level of the function $A$.

Lemma 9

For $j \geq 1$, we have $A_1(j) = 2j + 1$.

Lemma 10

For $j \geq 1$, we have $A_2(j) = 2^{j+1}(j + 1) - 1$. 
**Proof of Lemma 9**

By definition

\[
A_1(j) = A_0^{(j+1)}(j) \\
= A_0(A_0(\cdots A_0(j))) \quad \text{\(j+1\) times} \\
= (\cdots ((j + 1) + 1) \cdots ) + 1 \quad \text{\(j+1\) times} \\
= 2j + 1
\]
Proof of Lemma 10

First, we show by induction on $i$ that

$$A_1^{(i)}(j) = 2^i(j + 1) - 1. \quad (1)$$

**Base case:** $i = 0$. $A_1^{(0)}(j) = j = 2^0(j + 1) - 1$

**Induction step:** $i \rightarrow i + 1$.

$$A_1^{(i+1)}(j) = A_1(A_1^{(i)}(j)) \quad \text{(by definition of functional iteration)}$$
$$= A_1(2^i(j + 1) - 1) \quad \text{(by induction hypothesis)}$$
$$= 2(2^i(j + 1) - 1) + 1 \quad \text{(by Lemma 9)}$$
$$= 2^{i+1}(j + 1) - 1$$

Now we can plug in (1) in the definition of $A_2(j)$ to get the claim:

$$A_2(j) = A_1^{(i+1)}(j) = 2^{i+1}(j + 1) - 1$$
## Time Analysis of Forests

### $A_3(1)$

<table>
<thead>
<tr>
<th>$A_3(1)$</th>
<th>$A_4(1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_3(1) = A_2^{(2)}(1)$</td>
<td>$A_4(1) = A_3^{(2)}(1)$</td>
</tr>
<tr>
<td>$= A_2(A_2(1))$</td>
<td>$= A_3(A_3(1))$</td>
</tr>
<tr>
<td>$= A_2(7)$</td>
<td>$= A_2^{(2048)}(2047)$</td>
</tr>
<tr>
<td>$= 2^8 \cdot 8 - 1$</td>
<td>$&gt; 2^{2\cdots 2^{2047}}$ <em>(as $A_2(2047) &gt; 2^{2047}$)</em></td>
</tr>
<tr>
<td>$= 2047$</td>
<td>$2048$ times</td>
</tr>
</tbody>
</table>

$$
\gg 2^{2047} \\
= 10^{2047 \cdot \log_2 / \log_{10}} \\
> 10^{616}
$$
Time Analysis of Forests

**Definition 11 (function $\alpha(n)$)**

Define $\alpha$ as the inverse of $k \mapsto A_k(1)$. For $n \geq 0$, let

$$\alpha(n) := \min\{k \mid A_k(1) \geq n\}.$$

**Example 12**

$$\alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq A_4(1). 
\end{cases}$$

$\alpha(n)$ is a very slowly growing function ...

... and can be assumed to be at most 4 in all practical problem sizes $n$. It is almost, but not quite, a constant.
Time Analysis of Forests

Theorem 13

A sequence of $m$ MAKE-SET, UNION and FIND-SET operations, $n$ of which are MAKE-SET, can be performed on a disjoint-set forest with union by rank and path compression in worst-case time $O(m \alpha(n))$.

Method of proof

• FIND-SET is only operation that can be expensive
• A single FIND-SET can take up to time $O(\log n)$.  
  ⇒ With naive analysis we can prove only $O(m \log n)$ bound.
• Rough idea for Theorem 13:
  • If FIND-SET($x$) is time-consuming, then $x$ is a deep node and – through path-compression – afterwards the forest has become more efficient for other FIND-SET operations.
  • Overall, there cannot be too many expensive calls to FIND-SET.
• Proof will require analysis technique amortized analysis.
Properties of Ranks

Lemma 14

For all nodes $x$ we have

- $x.rank < x.p.rank$ if $x.p \neq x$
- $x.rank$ monotonically increases over time, starts at 0 and does not change anymore when $x \neq x.p$
- $x.p.rank$ monotonically increases over time

Proof.

Induction on the number of iterations of the operations defined on page 25.

Lemma 15

Every node has rank at most $\lfloor \log_2 n \rfloor$.

Proof.

Aufgabe 1, Blatt 11.
Properties of Ranks

Example 16 (A disjoint set forest with ranks)
Time Analysis of Forests

Counting LINKS instead of UNIONS

- **UNION** consists of two **FIND-SET** and one **LINK** operation
- will analyse running time in terms of number \( m \) of **basic operations** (**FIND-SET, LINK, MAKE-SET**) instead of in terms of the number \( m' \) of original operations (**FIND-SET, UNION, MAKE-SET**)
- To prove theorem 13 it will be sufficient to show that the running time of \( m \) basic operations is \( O(m \alpha(n)) \).
  Theorem 13 then follows as there are at most three times as many basic operations as original operations:
  \[ m = \Theta(m') \Rightarrow O(m \alpha(n)) = O(m' \alpha(n)). \]
The potential function for amortized analysis is

$$\Phi_q = \sum_{\text{nodes } x \text{ in forest}} \phi_q(x)$$

after $q$ basic operations.
We set $\Phi_0 = 0$ (potential of the empty forest).
Will need to define potential of a node $\phi_q(x)$. 
Time Analysis of Forests: Potential Function

**Definition 17 (level)**

For a node $x$ with $x.rank \geq 1$ that is not a root define

$$level(x) := \max\{k \mid A_k(x.rank) \leq x.p.rank\}.$$  

**Range of Levels**

We have

$$0 \leq level(x) < \alpha(n).$$

**Proof.**

$$A_0(x.rank) = x.rank + 1$$

$$\leq x.p.rank \quad \text{(by lemma 14)}$$

Therefore, the maximum is defined and $level(x) \geq 0$. On the other hand

$$A_{\alpha(n)}(x.rank) \geq A_{\alpha(n)}(1) \quad \text{(because } A_k \text{ is increasing)}$$

$$\geq n \quad \text{(by definition of } \alpha)$$

$$> x.p.rank \quad \text{(by lemma 15)}$$

implies that $level(x) < \alpha(n)$. 

$\square$
**Remark**

`level(x)` increases over time because `x.p.rank` monotonically increases over time (by lemma 15).

**Example 18 (Level of a node)**
Definition 19 (iter)

For a node \( x \) with \( x.rank \geq 1 \) define

\[
iter(x) := \max\{ i \mid A^{(i)}_{\text{level}(x)}(x.rank) \leq x.p.rank \};
\]

the largest number of iterations of applying \( A_{\text{level}(x)} \) to the rank of \( x \) before the value gets larger than the rank of \( x \)'s parent.

Range of \( iter \)

We have

\[
1 \leq iter(x) \leq x.rank.
\]

Proof.

\[
x.p.rank \geq A_{\text{level}(x)}(x.rank) \quad \text{(by definition of } \text{level}(x))
\]

\[
= A^{(1)}_{\text{level}(x)}(x.rank) \quad \text{(by definition of functional iteration)}
\]

and therefore \( iter(x) \geq 1 \). To prove the upper bound consider

\[
A^{\text{iter}(x)+1}_{\text{level}(x)}(x.rank) = A_{\text{level}(x)+1}(x.rank) \quad \text{(by definition of } A) \\
> x.p.rank \quad \text{(by definition of } \text{level}(x))
\]

This implies – as \( A_k \) is monotonically increasing – that \( iter(x) \leq x.rank \).
Definition 20 (Potential function)

Let \( x \) be any node and let \( q \) be the number of operations after which the potential is evaluated. Let

\[
\phi_q(x) := \begin{cases} 
\alpha(n) \cdot x.\text{rank} & \text{if } x \text{ is a root or } x.\text{rank} = 0 \\
(\alpha(n) - \text{level}(x)) \cdot x.\text{rank} - \text{iter}(x) & \text{otherwise.}
\end{cases}
\]

Lemma 21

For all \( x \) and \( q \) we have

\[
0 \leq \phi_q(x) \leq \alpha(n) \cdot x.\text{rank}
\]

and when \( x \) is not a root and \( x.\text{rank} > 0 \) then we even have

\[
\phi_q(x) < \alpha(n) \cdot x.\text{rank}.
\]
Time Analysis of Forests: Potential Function

Proof.

For a root $x$ or when $x.rank = 0$, the bounds are obviously satisfied. Let $x$ not be a root and let $x.rank > 0$. The lower and upper bounds for $\phi_q(x)$ are an immediate consequence of the upper and lower bounds for $level(x)$ and $iter(x)$:

$$
\phi_q(x) = (\alpha(n) - level(x)) \cdot x.rank - iter(x)
$$

$$
\geq (\alpha(n) - (\alpha(n) - 1)) \cdot x.rank - x.rank
$$

$$
= 0
$$

$$
\phi_q(x) = (\alpha(n) - level(x)) \cdot x.rank - iter(x)
$$

$$
\leq \alpha(n) \cdot x.rank - 1
$$

$$
< \alpha(n) \cdot x.rank
$$
Lemma 22

The amortized cost of each MAKE-SET operation is $O(1)$.

Proof.

Suppose that the $q$th operation is MAKE-SET($x$). The amortized cost $\hat{c}_q$ of that operation is equal to the actual cost $c_q$ of that operation plus the change of the potential:

$$\hat{c}_q = c_q + \Phi_q - \Phi_{q-1}$$

The potentials of all the previously existing nodes do not change and the potential of the newly created node $x$ is $\phi_q(x) = 0$ as $x.rank = 0$. Therefore $\Phi_q = \Phi_{q-1}$ and the amortized costs equal the actual costs which are $O(1)$. 

Lemma 23

The amortized cost of each LINK operation is $O(\alpha(n))$.

Proof...

Suppose that the $q$th operation is LINK($x$, $y$). WLG assume that $x.rank \leq y.rank$ such that $y$ is made the parent of $x$. Because the potential of a node depends only on the nodes rank and on its parent rank, the only nodes that may change potential are $y$ and the children of $y$, including the new child $x$:

$$\phi_q - \phi_{q-1} = \sum_{z \in \{x, y\} \cup \{\text{children of } y\}} (\phi_q(z) - \phi_{q-1}(z))$$

We will separately bound the changes in node potentials from above: Let $y.rank_q$ and $y.rank_{q-1}$ denote the rank of $y$ after and before the $q$th operation. Then

$$\phi_q(y) - \phi_{q-1}(y) = \alpha(n) \cdot y.rank_q - \alpha(n) \cdot y.rank_{q-1} \quad \text{(because } y \text{ is and remains a root)}$$

$$\leq \alpha(n) \quad \text{(because } y.rank_q \leq y.rank_{q-1} + 1).$$

The rank of $x$ does not change during this operation. If $x.rank = 0$ then $\phi_q(x) - \phi_{q-1}(x) = 0 - 0 = 0$. If $x.rank > 0$ then

$$\phi_q(x) - \phi_{q-1}(x) = \phi_q(x) - \alpha(n) \cdot x.rank \quad \text{(because } x \text{ was a root before operation } q)$$

$$\leq 0 \quad \text{(because of lemma 21).}$$
...Proof.

Let \( z \neq x \) be any child of \( y \). The rank of \( z \) does not change with operation \( q \). If \( z.rank = 0 \), then, again, \( \phi_q(z) - \phi_{q-1}(z) = 0 - 0 = 0 \). Now, suppose \( z.rank > 0 \), let \( \text{iter}_q(z) \) and \( \text{iter}_{q-1}(z) \) denote the value of \( \text{iter}(z) \) after the \( q \)th and \( (q - 1) \)th operation and consider two cases:

1. \( \text{level}(z) \) does not change with operation \( q \). By definition of \( \text{iter} \) and because \( z.p.rank \) can not decrease (lemma 14), \( \text{iter}(z) \) cannot decrease. Therefore,

\[
\phi_q(z) - \phi_{q-1}(z) = -\text{iter}_q(z) + \text{iter}_{q-1}(z) \leq 0.
\]

2. \( \text{level}(z) \) increases with operation \( q \) at least by 1. By the remark on page 57 this is the only alternative if \( \text{level}(z) \) changes. Then

\[
\phi_q(z) - \phi_{q-1}(z) \leq -z.rank - \text{iter}_q(z) + \text{iter}_{q-1}(z)
\leq -1 \quad \text{(by the boundaries of \( \text{iter} \))}
\leq 0
\]

Summarizing, we get \( \Phi_q - \Phi_{q-1} \leq \alpha(n) \). The actual costs of \( \text{LINK} \) are \( O(1) \). Therefore, the amortized costs of \( \text{LINK} \) are at most \( O(1) + \alpha(n) = O(\alpha(n)) \). \qed
Lemma 24

The amortized cost of each FIND-SET operation is \(O(\alpha(n))\).

Proof...

Suppose that the \(q\)th operation is a FIND-SET and that the find path contains \(s\) nodes. If \(z\) is any node other than the root then the argument of the proof of lemma 23 (second page) shows that the potential of \(z\) does not increase. The potential of the root does not change.

Let

\[
B = \{x \mid \text{node } x \text{ is on the find path and } x.rank > 0 \text{ and there is a node } y \text{ on the way properly between } x \text{ and the root, such that } level(y) = level(x)\}
\]

\(B\) contains at least \(s - 2 - \alpha(n)\) elements: It contains all elements on the find path, except (possibly) the first node (if it has rank 0), the root, and for each possible level \(0 \leq k < \alpha(n)\) the highest node on the find path with that level (if it exists).

Claim: The potential of each of the nodes in \(B\) decreases by at least 1.

Proof. Let \(x \in B\), \(y\) be as in the definition of \(B\) and let \(k = level(x) = level(y)\).

\[
\begin{align*}
y.p.rank & \geq A_k(y.rank) & \text{(by definition of } level(y)) \\
& \geq A_k(x.p.rank) & \text{(by lemma 14)} \\
& \geq A_k(A_k^{(iter(x))}(x.rank)) & \text{(as, by definition of iter, } A_k^{(iter(x))}(x.rank) \leq x.p.rank) \\
& = A_k^{(iter(x)+1)}(x.rank) & \text{(by definition of functional iteration)}.
\end{align*}
\]
Time Analysis of Forests

...Proof.

\( x.\text{rank} \) does not change. After path compression, both \( x \) and \( y \) have the root as parent. Therefore, and because the root has at least the rank \( y.\text{p.rank} \), we get after path compression that

\[
x.\text{p.rank} \geq A_k^{(\text{iter}(x) + 1)}(x.\text{rank}).
\]

Therefore, either \( k \) remains the level of \( x \) after path compression, or \( \text{level}(x) \) increases. If \( k \) remains the level of \( x \), then \( \text{iter}(x) \) increases and therefore the potential of \( x \) decreases.

If \( \text{level}(x) \) increases, then as argued in the proof of lemma 23 (second page, item 2) the potential decreases, too. As the potential is an integer this concludes above claim.

Therefore, \( \Phi_q - \Phi_{q-1} \leq 2 + \alpha(n) - s \). The actual costs of the FIND-SET operation are \( O(s) \). As a result, we get that the amortized costs are \( O(s) - s + 2 + \alpha(n)) = O(s) - s + \alpha(n) \). As the units of amortized analysis can be scaled up if necessary, this yields amortized costs of \( O(\alpha(n)) \).
Theorem 13

A sequence of $m$ MAKE-SET, UNION and FIND-SET operations, $n$ of which are MAKE-SET, can be performed on a disjoint-set forest with union by rank and path compression in worst-case time $O(m\,\alpha(n))$.

Proof.

The theorem follows by amortized analysis as – by above three lemmas – the amortized cost of the $m$ operation is $O(\alpha(n))$ and by the above observation that we get the same time bound when counting LINK operations instead of UNION operations.