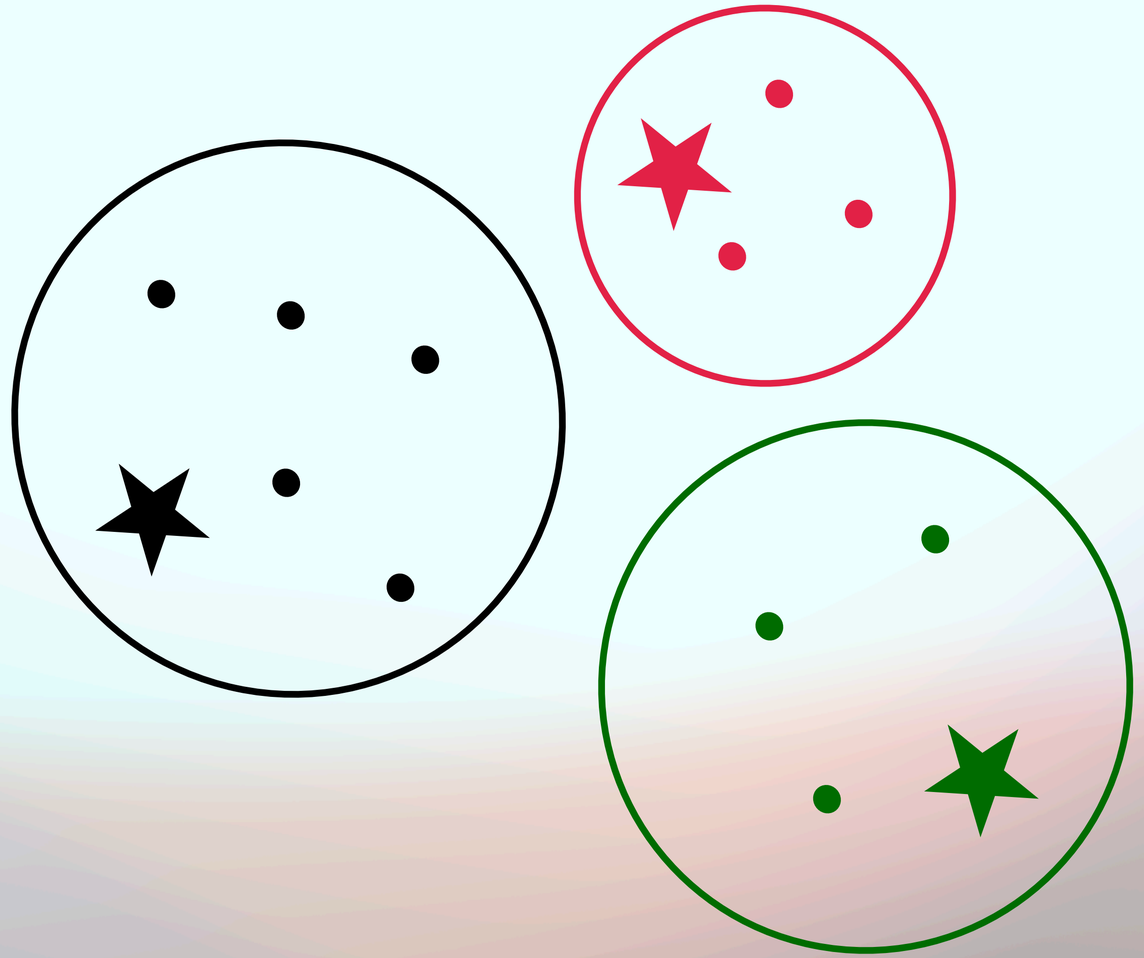
Lecture 6

Disjoint-set data structure
(and longest common substring)

Today’s plan

- Disjoint-set data structure and its applications
- Linked-list implementation
- Disjoint-set forest
- Time analysis

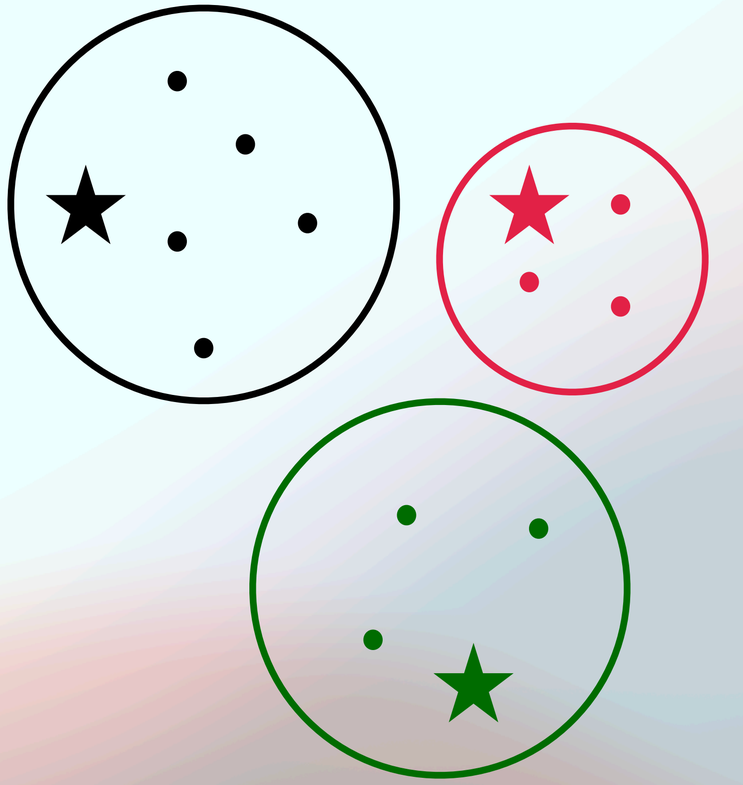
Disjoint-set data structure



Disjoint-set data structure

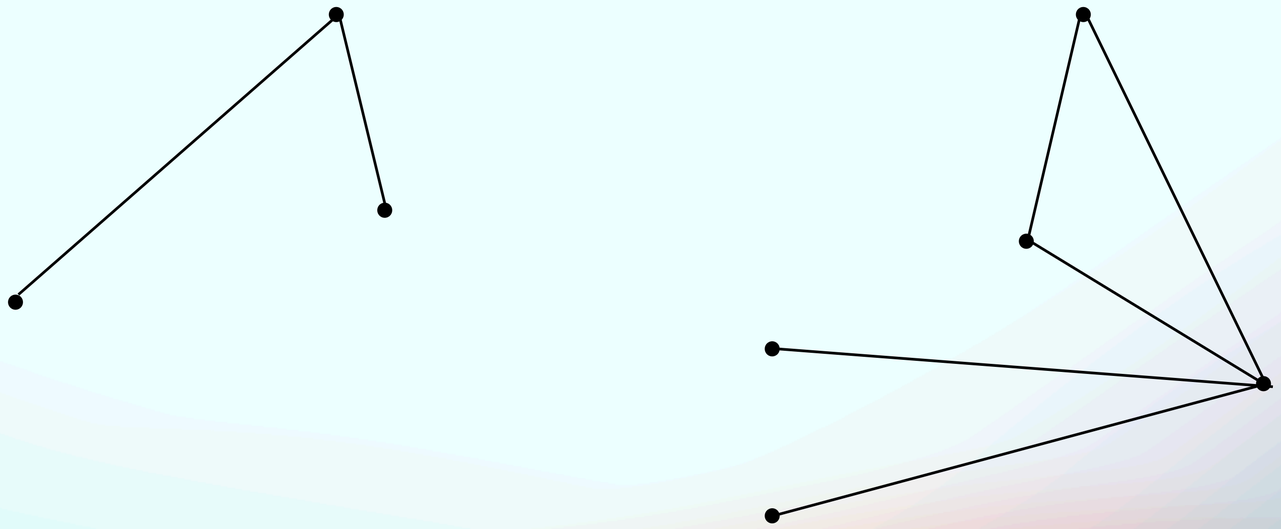
Maintain a collection of **disjoint sets** with 3 operations:

- **make_set(x):** create a new set containing one element, x
- **union_set(x, y):** union the sets containing x and y
- **find_set(x):** return a pointer to the representative of the set containing x



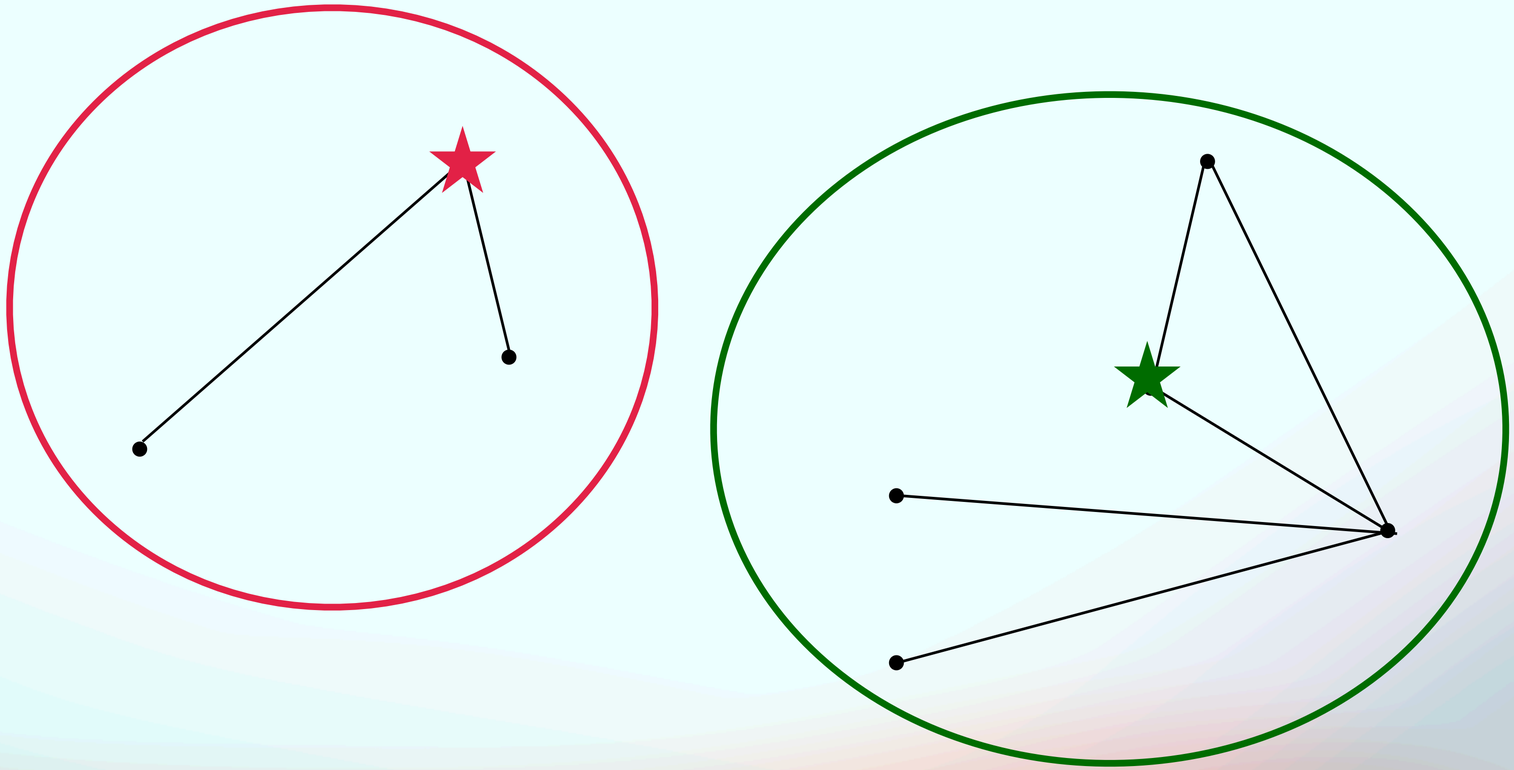
Two parameters: n = number of elements, m = number of operations

Application I: Connected components



Given a graph, preprocess it to answer the following queries fast: are two vertices u, v in the same connected component (= is there a path between them)?

Application I: Connected components



Preprocessing: add all vertices to the disjoint-set union data structure. For each edge (u, v), union the sets containing u and v. After the preprocessing, each set is a connected component of the graph $\Leftrightarrow$ u, v belong to the same set iff the representatives of the sets containing them are equal

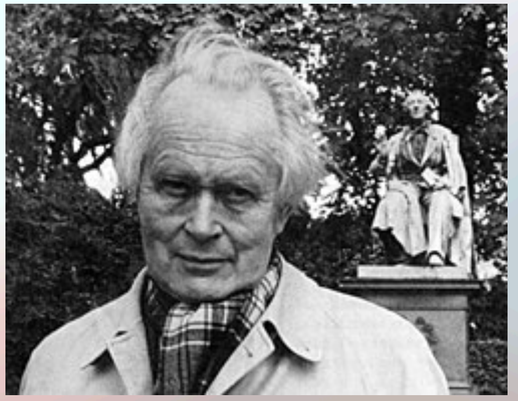
Application: HEX

Invented by Piet Hein in 1942 while a student at Niehls Bohr's Institute for Theoretical Physics, and independently by John Nash in 1948 while a math graduate student at Princeton.

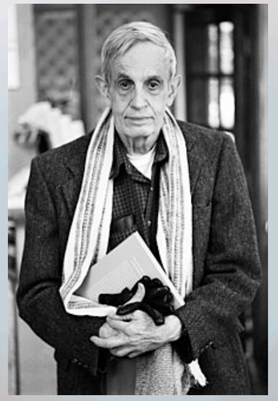
The game is also called "John or Nash" ("John" is another word for bathrooms, as the game used to be played on hexagon-shaped bathroom tiles).

The game has two players. Each player has an allocated color, white or black. Players take turns placing a stone of their color on a single cell within the overall playing board. Once placed, stones are not moved, captured or removed from the board. The goal for each player is to form a connected path of their own stones linking the opposing sides of the board marked by their colors, before their opponent connects his or her sides in a similar fashion. The first player to complete his or her connection wins the game.

Disjoint-set data structures can be used to decide who wins (maintain 2 data structures, one for each colour).



Piet Hein



John Nash

Linked-list representation

Baltic chain, August 1989. Copyright: Wikipedia

Linked-list representation

- Each set = a linked list, representative = the head of the list
- For every element, store a pointer to the node containing it. In the node, store a pointer to the representative.
- **make_set**(x): $O(1)$ time, **find_set**(x): O(1) time
- **union_set**(x, y) - requires more work

union_set(x, y): First attempt

- We can simply append the set (= a list) of y to the end of the set (= a list as well) of x
- Appending the list takes constant time, but we also need to update the pointers to the representatives: $\Theta(|\text{set of } y|)$ time
- Unfortunately, even amortised time per op. can be large:

make_set(x_1), **make_set**(x_2), ..., **make_set**(x_n);

union_set(x_2, x_1), **union_set**(x_3, x_2), ..., **union_set**(x_n, x_{n-1})

Total time $\Theta(n^2)$, amortised time per operation $\Theta(n)$

union_set(x, y): Second attempt

- We apply a **weighted-union** "heuristic":
  - if the set of x is larger than the set of y, append the latter to the former;
  - else append the set of x to the set of y.
- Worst-case time per operation is still $\Theta(n)$
- Amortised time is much better!

union_set(x, y): Second attempt

Theorem: Linked-list + weighted union heuristic $\Rightarrow$ m make_set, union_set, find_set operations, n of which are make_set, take $O(m + n \log n)$ time.

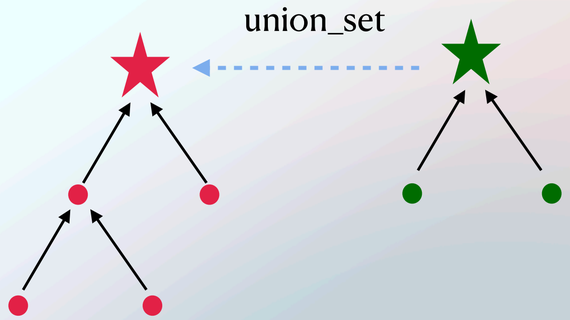
Proof: At any moment, the largest size of a set is bounded by n. Therefore, an element can change its set representative at most $\log_2 n$ times, as every time it happens the size of the set of x at least doubles in size. Therefore, the total time for updating the representatives is $O(n \log n)$. All other operations take $O(m)$ time.

Disjoint-set forests

Dismaland, Banksy. Photo: T. Starikovskaya

Disjoint-set forests

- Each set = tree, the representative = the root
- Each node stores a pointer to its parent
- make_set(x): create a tree containing one node x
- find_set(x): follow the path from x to the root
- union_set(x, y): The root of the tree of y becomes a child of the root of the tree of x



Two “heuristics”

Path compression

During **find_set**(x), make each node on the path from x to the root to be a child of the root

Union by rank

During **union_set,** the root of the tree with smaller rank becomes a child of the root of the tree with larger rank

Ranks

- Rank is an upper bound on the height of a node (= length of a longest path from the node to a leaf below it).
- When we add a new node x during **make_set**(x), we initialise the rank of x with 0.
- If we union two trees (i.e., two sets) of different ranks, ranks do not change. If we union two trees of equal ranks, the rank of the root of the resulting tree increases by one.

Pseudocode

make_set(x)

$x.p \leftarrow x$

$x.rank \leftarrow 0$

union_set(x, y)

link(find_set(x), find_set(y))

find_set(x)

if $x \neq x.p$:

$x.p \leftarrow$ find_set($x.p$)

return $x.p$

link(x, y)

if $x \neq y$:

if $x.rank > y.rank$:

$y.p \leftarrow x$

else

$x.p \leftarrow y$

if $x.rank = y.rank$:

$y.rank \leftarrow y.rank + 1$

Time analysis

Recall that

n = number of elements = number of make_sets

m = total number of operations.

Theorem [Tarjan, 1975]

Using both Union by Rank and Path Compression, the time complexity for any sequence of Find and Union is $O(m + n\alpha(n))$, where $\alpha(n)$ is the inverse of the Ackerman function. In any conceivable application, $\alpha(n) \leq 4$.

Fredman and Saks proved in 1989 that this is tight for any implementation of disjoint set data structure.

Tarjan's analysis is quite intricate. A much simpler analysis was given by Hopcroft and Ullman in 1973 showing:

$$O(m \log^* n) \textbf{ time}$$

where $\log^* n$ is the number of times you have to push the log button of a calculator to go from n to at most 1 ($\log^* 2^{65536} = 5$)

Forget about unions

Observation. Suppose we convert a sequence of m' operations make_set, union_set, find_set into m operations make_set, link, find_set. If we can execute the latter in $O(m \cdot \alpha(n))$ time, we can execute the former in $O(m' \cdot \alpha(n))$ time.

Proof: $m' \leq m \leq 3m'$.

From now on, we assume that we have only make_sets, links, and find_sets.

Properties of ranks

Observation 1. Rank of a node can only increase with time. Rank of non-root nodes are frozen forever.

Proposition 2. For every node x, $x\,.\,rank \leq x\,.\,p\,.\,rank$**.** The inequality is strict unless x is a root node.

Corollary 3. The ranks of nodes in any path strictly increase.

Observation 4. The rank of any node $\leq n - 1$.

Properties of ranks

Rank lemma: At any moment, for every $r \in \mathbb{N}$, there are $\leq n/2^r$ nodes with rank r.

Proof. By induction, the number of nodes in a tree with a root of rank r is at least 2^r.

Base case. $r = 0$, trivially true.

Induction step. After linking two trees with roots of ranks r_1, r_2:

- If $r_1 < r_2$, the claim is true by induction for r_2.
- If $r = r_1 = r_2$, then the root of the resulting tree has rank $r + 1$ and the tree has at least $2^r + 2^r = 2^{r+1}$ nodes.

find_set operation does not change the number of nodes in a tree.

Properties of ranks

Rank lemma: At any moment, for every $r \in \mathbb{N}$, there is at $\leq n/2^r$ nodes with rank r.

Proof. By induction, the number of nodes in a tree with a root of rank r is at least 2^r.

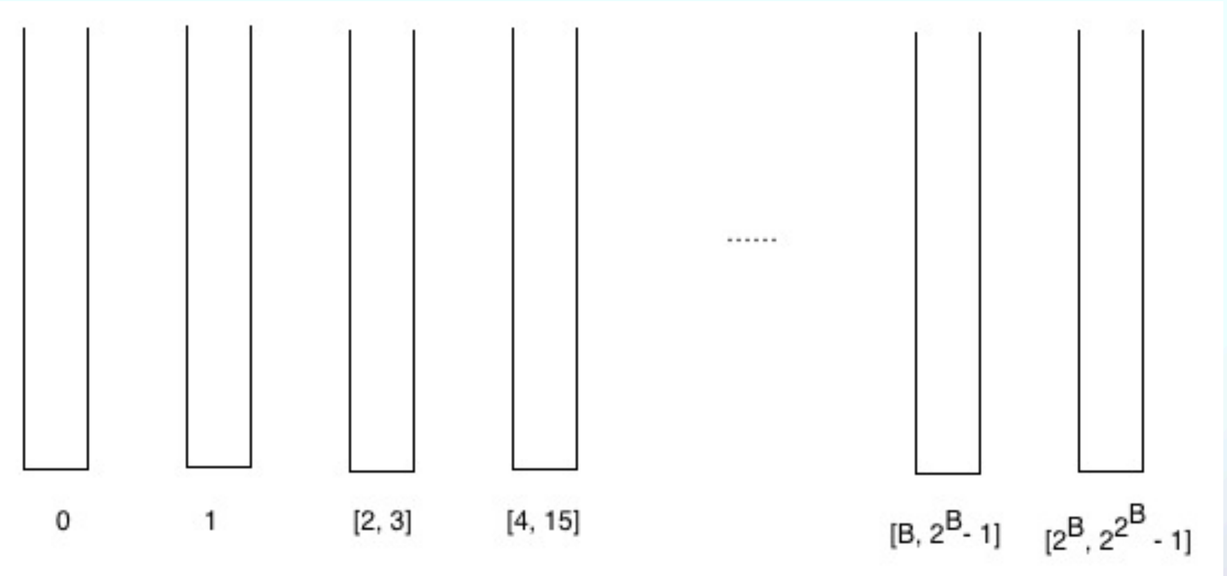
Consider all nodes of rank r.

None of them is not and was not an ancestor of another.

Hence, there are 2^r nodes corresponding to each of them in the forest, and these sets are independent.

As we have n nodes in total, the bound follows.

Buckets



By Qunwangcs157 - by omnigraphics, Public Domain, https://commons.wikimedia.org/w/index.php?curid=71684618

$B_0 = [0,0], B_1 = [1,1], B_2 = [2^1, 2^2 - 1], B_3 = [2^2, 2^{2^2} - 1], \ldots$

Buckets are not implemented, we only use them for the analysis. The total number of buckets is $\leq \log^* n$.

Buckets

Put a node with a rank r into the bucket $B \ni r$. The number of nodes in a bucket $B = [b, 2^b - 1]$ is $\leq \sum_{r \in B} n/2^r \leq 2n/2^b$.

Definition

At a given time, a node is **good** if:

- It is a root, or
- It is a child of a root, or
- It is in a smaller bucket than its parent.

It is **bad** otherwise.

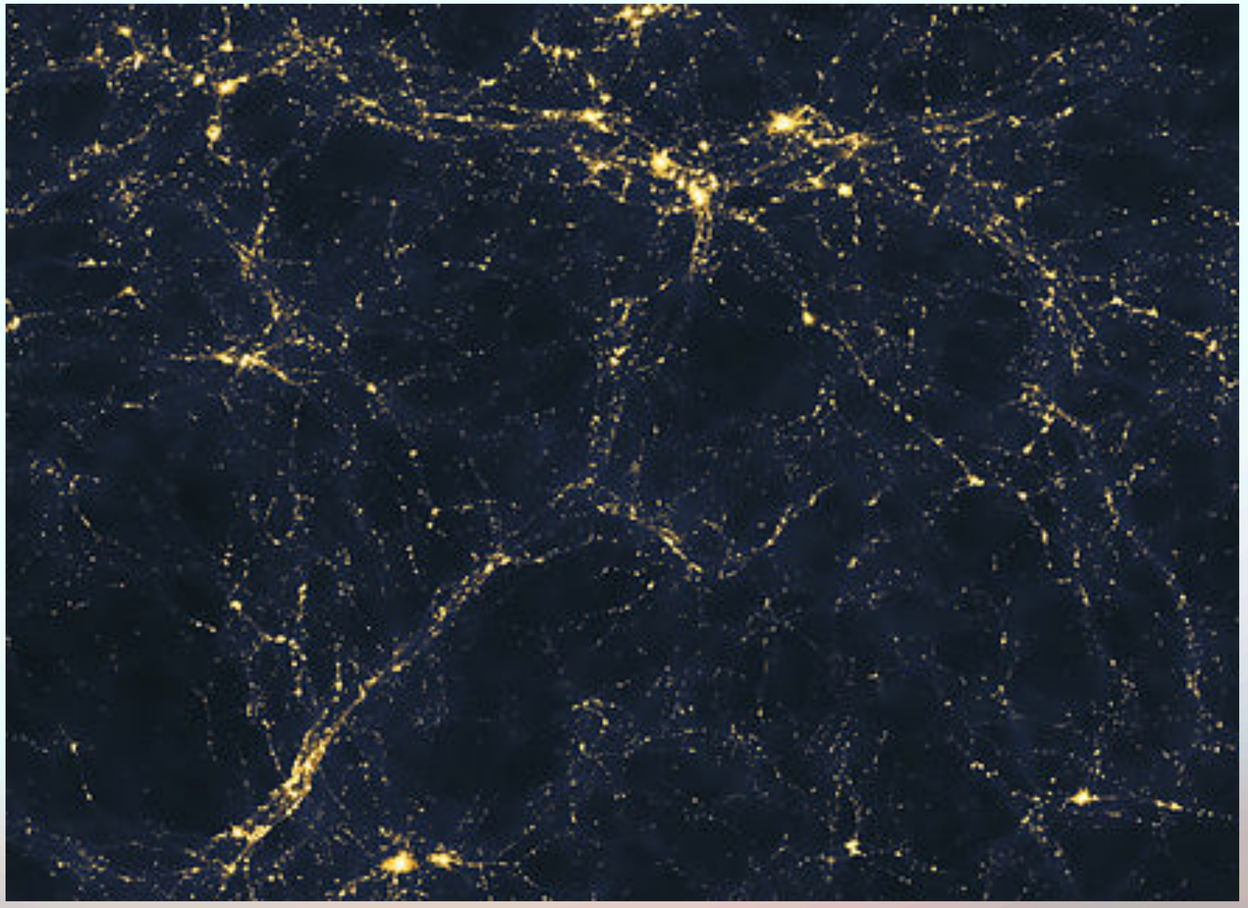
A find_set visits $O(\log^* n)$ good nodes. Therefore, the total time complexity is:

$$O(m \log^* n + \textbf{total number of bad nodes visited})$$

Bad nodes

- Bad nodes have frozen rank
- When a bad node x is visited by a find_set, the new x.p has rank strictly larger than the previous one
- Hence, for each bad node x in a block $B = [b, 2^b - 1]$, the number of times x is visited while x is bad is at most $2^b - 1$
- But the number of nodes in B is at most $2n/2^b$
- So the number of times we visit a bad node in B is O(n)
- Hence, the total number of visits of bad nodes is $O(n \log^* n)$

Ackermann's function



Computer simulated image of an area of space more than 50 million light-years across, presenting a possible large-scale distribution of light sources in the universe.

Definition

$k \geq 0, j \geq 1$ - integers. Ackermann's function is defined as

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

Here $A_{k-1}^{j+1}(j)$ is a shorthand for $\underbrace{A_{k-1}(A_{k-1}(\ldots(A_{k-1}}_{j+1 \text{ times}}(j)\ldots)$

Properties

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

Lemma 1. $\forall j \geq 1, A_1(j) = 2j + 1.$

Proof: By induction on i, $A_0^i(j) = j + i$. Hence, $A_1(j) = A_0^{j+1}(j) = 2j + 1.$

Lemma 2. $\forall j \geq 1, A_2(j) = 2^{j+1}(j + 1) - 1.$

Proof: By induction on i, $A_1^i(j) = 2^i(j + 1) - 1$. Hence,
$A_2(j) = A_1^{j+1}(j) = 2^{j+1}(j + 1) - 1.$

We finally obtain: $A_0(1) = 2, A_1(1) = 3, A_2(1) = 7,$
$A_3(1) = A_2^2(1) = A_2(A_2(1)) = A_2(7) = 2^{11} - 1 = 2047,$
$A_4(1) = A_3(A_3(1)) = A_3(2047) = A_2^{2048}(2047) \gg A_2(2047) = 2^{2048} \cdot 2048 - 1 > 2^{2048} \gg 10^{80}$
(estimated number of atoms in the observable universe)

Reverse Ackermann's function

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

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

Potential function

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

$$iter(x) = \max\{i : x\,.\,p\,.\,rank \geq A^{(i)}_{level(x)}(x\,.\,rank)\}$$

Def. [Potential function] For a node x, define

$$\Phi_q(x) = \begin{cases} \alpha(n) \cdot x\,.\,rank, & \text{if } x \text{ is a root or } x\,.\,rank = 0; \\ (\alpha(n) - level(x)) \cdot x\,.\,rank - iter(x), & \text{otherwise.} \end{cases}$$

For a forest F, define $\Phi_0(F) = 0$ and $\Phi_q(F) = \sum_{x \text{ – node of } F} \Phi_q(x)$

Potential function

In the next slides we'll show that $\Phi_q(F) \geq 0$ for any q.

t_q - time for the q-th operation (link, make_set, or find_set)

$$\sum_q t_q \leq \sum_q t_q + \Phi_q(F) - \underbrace{\Phi_0(F)}_{=0} = \sum_q t_q + \sum_q (\Phi_{q+1}(F) - \Phi_q(F)) = \sum_q \underbrace{(t_q + \Phi_{q+1}(F) - \Phi_q(F))}_{\text{amortised time}}$$

Potential function

$$level(x) = \max\{k : x\,.\,p\,.\,rank \geq A_k(x\,.\,rank)\}$$
$$iter(x) = \max\{i : x\,.\,p\,.\,rank \geq A^{(i)}_{level(x)}(x\,.\,rank)\}$$

Lemma 6. If $x \neq x\,.\,p$ and $x\,.\,rank \geq 1$, we have $0 \leq level(x) \leq \alpha(n) - 1$.

Proof: $x\,.\,p\,.\,rank \geq x\,.\,rank + 1 = A_0(x\,.\,rank)$. Hence, $level(x) \geq 0$.

$A_{\alpha(n)}(x\,.\,rank) \geq A_{\alpha(n)}(1)$ (monotonicity) $\geq n > x\,.\,p\,.\,rank$.
Hence, $level(x) \leq \alpha(n) - 1$.

Potential function

$$level(x) = \max\{k : x\,.\,p\,.\,rank \geq A_k(x\,.\,rank)\}$$
$$iter(x) = \max\{i : x\,.\,p\,.\,rank \geq A^{(i)}_{level(x)}(x\,.\,rank)\}$$

Lemma 7. If $x\,.\,rank \geq 1$, we have $1 \leq iter(x) \leq x\,.\,rank$.

Proof: $x\,.\,p\,.\,rank \geq A_{level(x)}(x\,.\,rank) = A^1_{level(x)}(x\,.\,rank)$.

Hence, $iter(x) \geq 1$.

On the other hand,
$A^{x.rank+1}_{level(x)}(x\,.\,rank) = A_{level(x)+1}(x\,.\,rank) > x\,.\,p\,.\,rank$ and
therefore $iter(x) \leq x\,.\,rank$.

Potential function

Lemma 8. ∀ node x and q, $0 \leq \Phi_q(x) \leq \alpha(n) \cdot x\,.\,rank$.

Proof: If x is the root or $x\,.\,rank = 0$, $\Phi_q(x) = \alpha(n) \cdot x\,.\,rank$, and we are done.

Otherwise, $\Phi_q(x) = (\alpha(n) - level(x)) \cdot x\,.\,rank - iter(x) \geq$

$(\alpha(n) - (\alpha(n) - 1)) \cdot x\,.\,rank - iter(x) \geq x\,.\,rank - x\,.\,rank = 0$

Also, $\Phi_q(x) \leq \alpha(n) \cdot x\,.\,rank - 1 < \alpha(n) \cdot x\,.\,rank$.

Corollary 9. If x is not a root and $x\,.\,rank > 0$, then $\Phi_q(x) < \alpha(n) \cdot x\,.\,rank$.

Potential function

Lemma 10. Node x is not a root, the q-th operation is link or find_set, then $\Phi_q(x) \leq \Phi_{q-1}(x)$. Moreover, if $x\,.\,rank \geq 1$ and either $level(x)$ or $iter(x)$ changes due to the q-th operation, then $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$.

Proof: x is not a root $\Rightarrow$ the rank of x does not change.

$$level(x) = \max\{k : x\,.\,p\,.\,rank \geq A_k(x\,.\,rank)\}$$
$$iter(x) = \max\{i : x\,.\,p\,.\,rank \geq A^{(i)}_{level(x)}(x\,.\,rank)\}$$
$$\Phi_q(x) = (\alpha(n) - level(x)) \cdot x\,.\,rank - iter(x)$$

1) If $x\,.\,rank = 0$, then $\Phi_q(x) = \Phi_{q-1}(x) = 0$

2) Assume $x\,.\,rank \geq 1$

- If $level(x)$ increases, $(\alpha(n) - level(x)) \cdot x\,.\,rank$ drops by at least $x\,.\,rank$, but $iter(x)$ can drop by at most $x\,.\,rank - 1$. Therefore, $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$.

- If $level(x)$ is unchanged, $iter(x)$ remains unchanged or increases. If it does not change, $\Phi_q(x) = \Phi_{q-1}(x)$, otherwise, $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$.

Time bound

Lemma 11. The amortised cost of each make_set is $O(1)$.

Proof: make_set creates a node x with $\Phi_q(x) = 0$.

Hence, $\Phi_{q-1}(F) = \Phi_q(F)$ and the amortised time is
$1 + \Phi_{q-1}(F) - \Phi_q(F) = 1.$

Time bound

Lemma 12. The amortised time of $link(x, y)$ is $O(\alpha(n))$.

Proof: W.l.o.g., y becomes the parent of x. By Lemma 10, the potentials of any node $\notin \{x, y\}$ cannot increase. Before $link(x, y)$, the node x is a root and $\Phi_{q-1}(x) = \alpha(n) \cdot x\,.\,rank$. If $x\,.\,rank = 0$, we have $\Phi_{q-1}(x) = \Phi_q(x) = 0$. Otherwise, by Corollary 9, $\Phi_q(x) < \alpha(n) \cdot x\,.\,rank = \Phi_{q-1}(x)$.

For y: either $\Phi_{q-1}(y) = \Phi_q(y)$ or $\Phi_q(y) = \Phi_{q-1}(y) + \alpha(n)$ (y is a root).

Hence, $\Phi_q(F) - \Phi_{q-1}(F) \leq \alpha(n)$. The actual time of $link(x, y)$ is 1, the lemma follows.

Time bound

Lemma 13. The amortised time of find_set is $O(\alpha(n))$.

Proof: Assume that the find path contains s nodes.

1) no potential increases (follows from Lemma 10 for non-roots. For a root x, the potential is $\alpha(n) \cdot x\,.\,rank$);

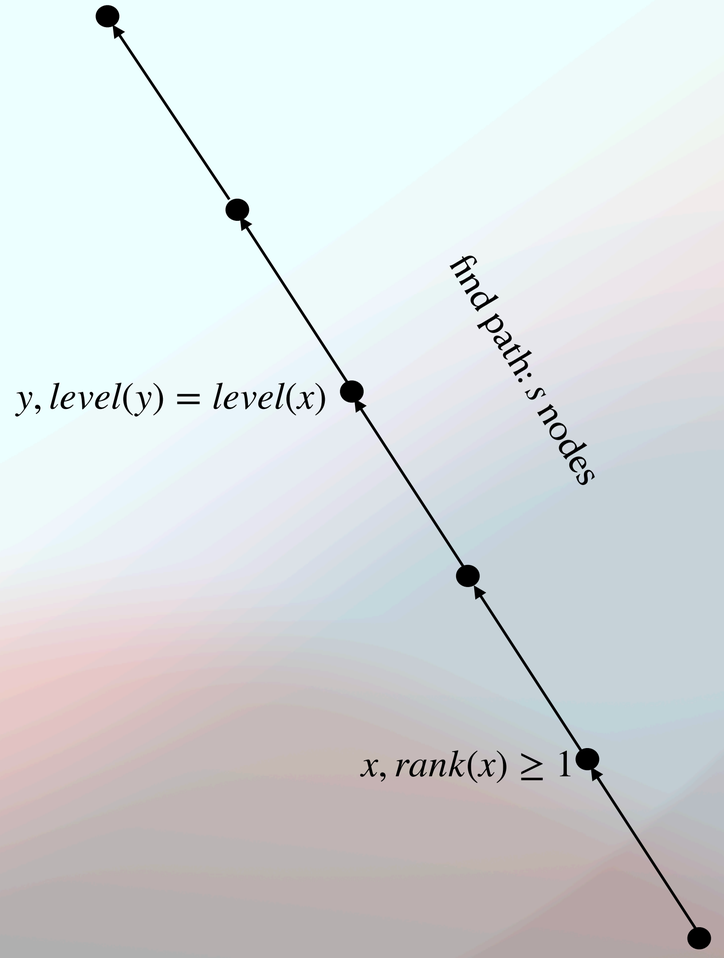
2) we show that the potential of at least $\max\{0, s - (\alpha(n) + 2)\}$ nodes decreases by at least one:

Time bound

Consider all nodes x on the find path such that $x\,.\,rank \geq 1$ and $\exists y$ that is an ancestor of x and $level(x) = level(y)$.

# of such nodes $\geq s - (\alpha(n) + 2)$
(+2 accounts for the first node of level = 0 and the root)

Consider such a node x. Let $k = level(x) = level(y)$



Time bound

Prior to path compression:

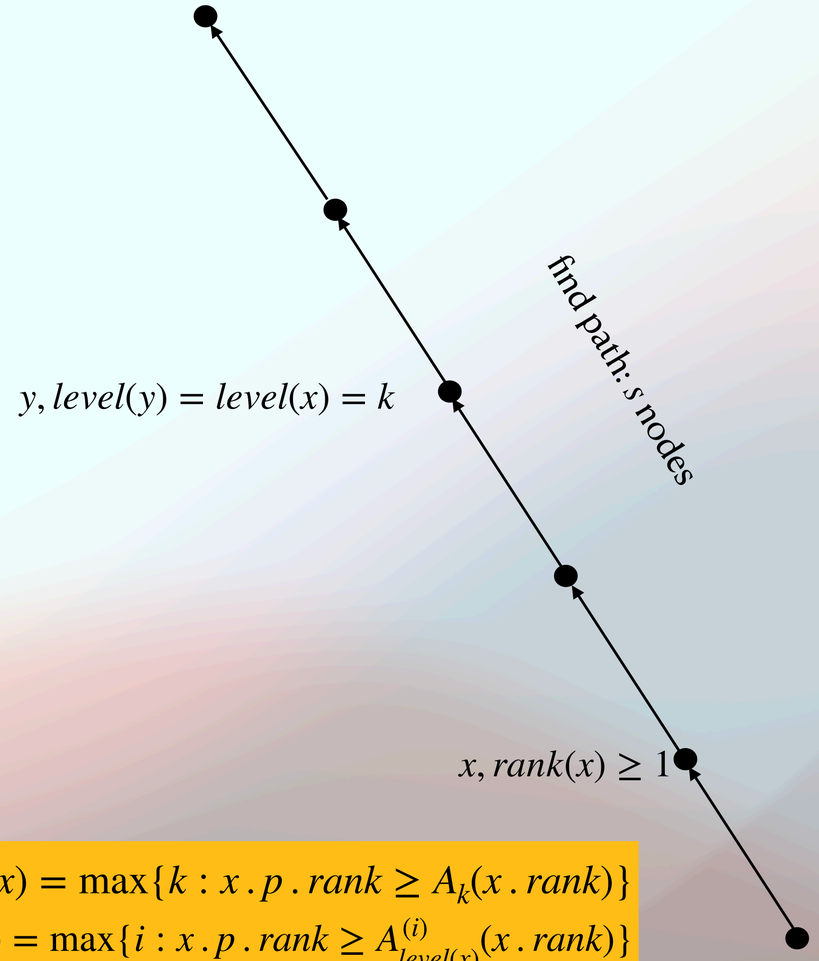
$x \,.\, p \,.\, rank \geq A_k^{iter(x)}(x \,.\, rank)$

$y \,.\, p \,.\, rank \geq A_k(y \,.\, rank)$

$y \,.\, rank \geq x \,.\, p \,.\, rank$ (rank is monotonically increasing along the path, y is above $x \,.\, p$)

Let $i := iter(x)$ - the old value

$y \,.\, p \,.\, rank \geq A_k(A_k^i(x \,.\, rank))$



$$level(x) = \max\{k : x \,.\, p \,.\, rank \geq A_k(x \,.\, rank)\}$$
$$iter(x) = \max\{i : x \,.\, p \,.\, rank \geq A^{(i)}_{level(x)}(x \,.\, rank)\}$$

Time bound

After path compression:

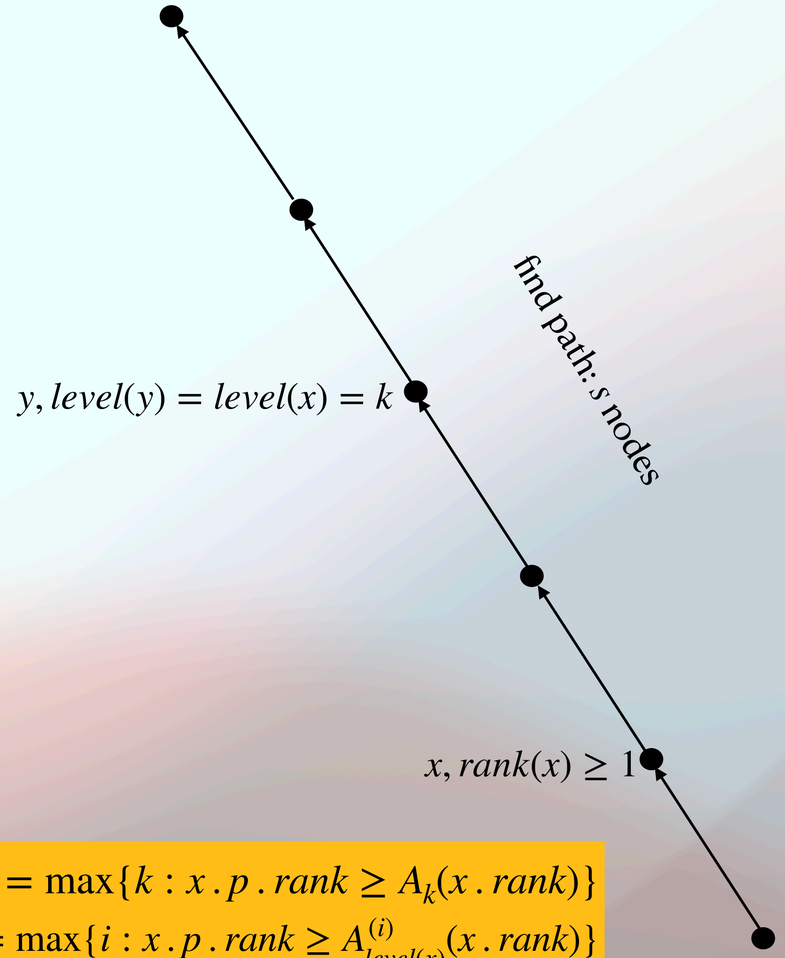
x and y have the same parent (the root)

$\Rightarrow x\,.\,p\,.\,rank = y\,.\,p\,.\,rank$

$x\,.\,p\,.\,rank = y\,.\,p\,.\,rank \geq A_k^{i+1}(x\,.\,rank)$

($x\,.\,rank$ does not change)

Therefore, either $iter(x)$ increases, or $level(x)$ increases, and hence $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$ (Lemma 10)



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

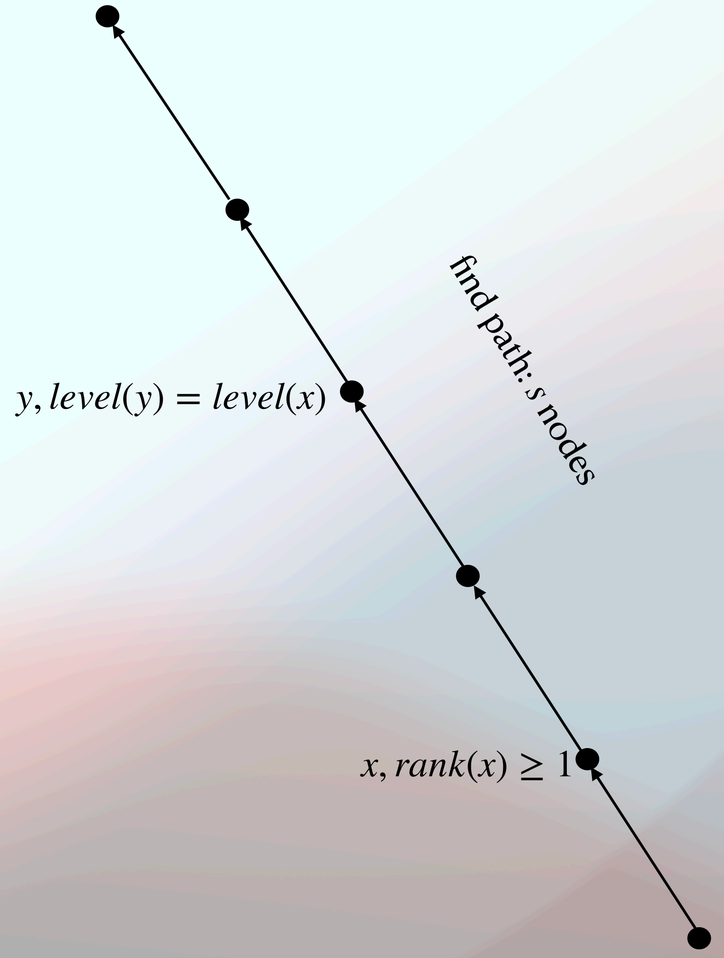
$iter(x) = \max\{i : x\,.\,p\,.\,rank \geq A^{(i)}_{level(x)}(x\,.\,rank)\}$

Time bound

As a corollary, we obtain that
$\Phi_q(F) \leq \Phi_{q-1}(F) - \max\{0, s - (\alpha(n) + 2)\}$

The actual time of find_path is s.

Therefore, the amortised time is
$O(\alpha(n) + 2)$.



Time bound: summary

Reminder: If we convert a sequence of m' operations make_set, union_set, find_set into m operations make_set, link, find_set and we can execute the latter in $O(m \cdot \alpha(n))$ time, we can execute the former in $O(m' \cdot \alpha(n))$ time.

Theorem: A sequence of m make_set, union_set, find_set (out of which n are make_set) takes $O(m \cdot \alpha(n))$. In other words, one operation takes $O(\alpha(n))$ amortised time.

Algorithms from the Book.

Asked 10 years, 2 months ago Active 1 year, 9 months ago Viewed 110k times

367

521

Paul Erdos talked about the "Book" where God keeps the most elegant proof of each mathematical theorem. This even inspired a book (which I believe is now in its 4th edition): Proofs from the Book.

If God had a similar book for algorithms, what algorithm(s) do you think would be a candidate(s)?

If possible, please also supply a clickable reference and the key insight(s) which make it work.

Only one algorithm per answer, please.

120

Union-find is a beautiful problem whose best algorithm/datastructure (**Disjoint Set Forest**) is based on a spaghetti stack. While very simple and intuitive enough to explain to an intelligent child, it took several years to get a tight bound on its runtime. Ultimately, its behavior was discovered to be related to the inverse Ackermann Function, a function whose discovery marked a shift in perspective about computation (and was in fact included in Hilbert's *On the Infinite*).

Wikipedia provides a good introduction to Disjoint Set Forests.

share cite improve this answer follow edited Sep 15 '10 at 8:37 community wiki
3 revs, 3 users 50%
Jukka Suomela

Source: https://cstheory.stackexchange.com/questions/189/algorithms-from-the-book

Today's lecture:

- Disjoint-set data structure and its applications
- Linked-list implementation
- Disjoint-set forest
- Iterated log analysis by Hopcroft and Ulmann
- Ackermann's function analysis by Tarjan