Union Find

Data Structures and Algorithms 2

Oswin Aichholzer

Union Find: Problem Definition

Union Find, setting:

- Set S of n elements: $S = \{s_1, s_2, \ldots, s_n\}$
- S is partitioned into subsets $S_1, S_2, \ldots, S_k$, that is, $S = \cup_{i=1,\ldots,k} S_i$ and $S_i \cap S_j = \emptyset$ for $i \neq j$.

Union Find: Problem Definition

Union Find, setting:

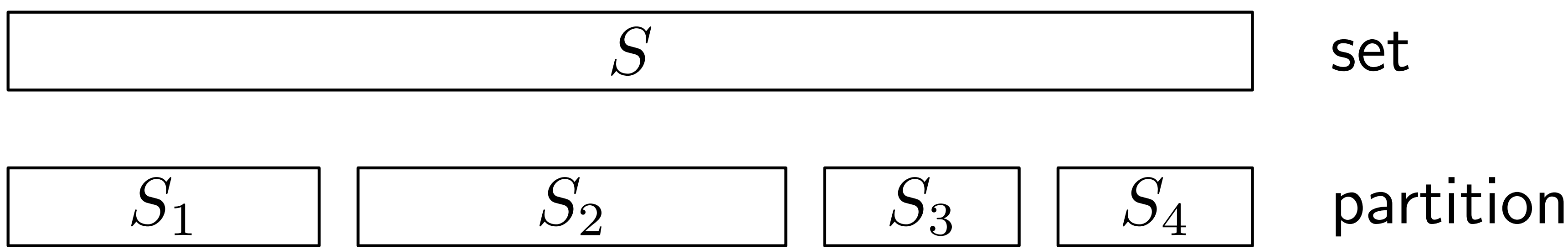
- Set S of n elements: $S = \{s_1, s_2, \ldots, s_n\}$
- S is partitioned into subsets $S_1, S_2, \ldots, S_k$, that is, $S = \cup_{i=1,\ldots,k} S_i$ and $S_i \cap S_j = \emptyset$ for $i \neq j$.

$\boxed{S}$ set

Union Find: Problem Definition

Union Find, setting:

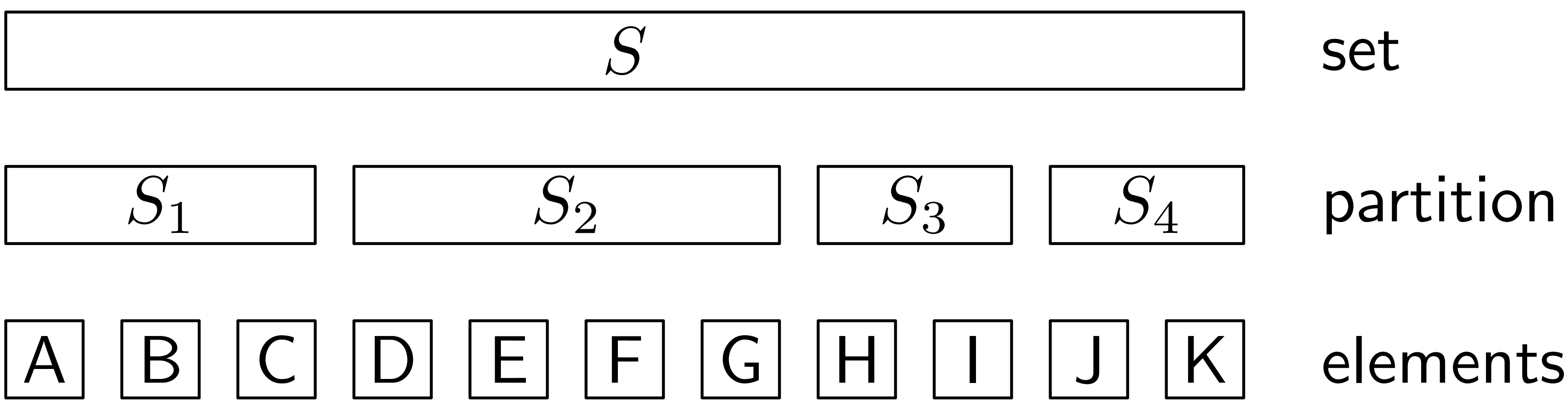
- Set S of n elements: $S = \{s_1, s_2, \ldots, s_n\}$
- S is partitioned into subsets $S_1, S_2, \ldots, S_k$, that is, $S = \cup_{i=1,\ldots,k} S_i$ and $S_i \cap S_j = \emptyset$ for $i \neq j$.



Union Find: Problem Definition

Union Find, setting:

- Set S of n elements: $S = \{s_1, s_2, \ldots, s_n\}$
- S is partitioned into subsets $S_1, S_2, \ldots, S_k$, that is, $S = \cup_{i=1,\ldots,k} S_i$ and $S_i \cap S_j = \emptyset$ for $i \neq j$.



Union Find: Problem Definition

Union Find, setting:

- Set S of n elements: $S = \{s_1, s_2, \ldots, s_n\}$
- S is partitioned into subsets $S_1, S_2, \ldots, S_k$, that is, $S = \cup_{i=1,\ldots,k} S_i$ and $S_i \cap S_j = \emptyset$ for $i \neq j$.

Two operations:

- **Union(S_i, S_j):** Combine two subsets S_i and S_j to a new subset: $S_{new} = S_i \cup S_j$.
- **Find(s_x):** Find the subset S_i to which s_x belongs

Union Find: Problem Definition

Union Find, setting:

- Set S of n elements: $S = \{s_1, s_2, \ldots, s_n\}$
- S is partitioned into subsets $S_1, S_2, \ldots, S_k$, that is, $S = \cup_{i=1,\ldots,k} S_i$ and $S_i \cap S_j = \emptyset$ for $i \neq j$.

Two operations:

- **Union(S_i, S_j):** Combine two subsets S_i and S_j to a new subset: $S_{new} = S_i \cup S_j$.
- **Find(s_x):** Find the subset S_i to which s_x belongs

Question:

- How fast can a sequence of $n-1$ unions and f finds (in arbitrarily interleaved order) be performed?

An Example

Initial setting:

- Initially there are n subsets $S_i = \{s_i\}$, $i = 1, \ldots, n$
- Exampe: $S = \{A, B, C, D, E\}$
  $S_1 = \{A\}$, $S_2 = \{B\}$, $S_3 = \{C\}$, $S_4 = \{D\}$, $S_5 = \{E\}$,

An Example

Initial setting:

- Initially there are n subsets $S_i = \{s_i\}$, $i = 1, \ldots, n$
- Exampe: $S = \{A, B, C, D, E\}$
  $S_1 = \{A\}$, $S_2 = \{B\}$, $S_3 = \{C\}$, $S_4 = \{D\}$, $S_5 = \{E\}$,

  FIND(D)=S_4

An Example

Initial setting:

- Initially there are n subsets $S_i = \{s_i\}$, $i = 1, \ldots, n$
- Exampe: $S = \{A, B, C, D, E\}$
  $S_1 = \{A\}$, $S_2 = \{B\}$, $S_3 = \{C\}$, $S_4 = \{D\}$, $S_5 = \{E\}$,

  FIND(D)=S_4
  UNION(S_3, S_4) %New set always gets smaller index

An Example

Initial setting:

- Initially there are n subsets $S_i = \{s_i\}$, $i = 1, \ldots, n$
- Exampe: $S = \{A, B, C, D, E\}$
  $S_1 = \{A\}$, $S_2 = \{B\}$, $S_3 = \{C\}$, $S_4 = \{D\}$, $S_5 = \{E\}$,

  FIND(D)=S_4
  UNION(S_3, S_4) %New set always gets smaller index
  FIND(A)=S_1

An Example

Initial setting:

- Initially there are n subsets $S_i = \{s_i\}$, $i = 1, \ldots, n$
- Exampe: $S = \{A, B, C, D, E\}$
  $S_1 = \{A\}$, $S_2 = \{B\}$, $S_3 = \{C\}$, $S_4 = \{D\}$, $S_5 = \{E\}$,

  FIND(D)=S_4
  UNION(S_3, S_4) %New set always gets smaller index
  FIND(A)=S_1
  FIND(D)=S_3

An Example

Initial setting:

- Initially there are n subsets $S_i = \{s_i\}$, $i = 1, \ldots, n$
- Exampe: $S = \{A, B, C, D, E\}$
  $S_1 = \{A\}$, $S_2 = \{B\}$, $S_3 = \{C\}$, $S_4 = \{D\}$, $S_5 = \{E\}$,

  FIND(D)=S_4
  UNION(S_3, S_4) %New set always gets smaller index
  FIND(A)=S_1
  FIND(D)=S_3
  UNION(S_1, S_3)

An Example

Initial setting:

- Initially there are n subsets $S_i = \{s_i\}$, $i = 1, \ldots, n$
- Exampe: $S = \{A, B, C, D, E\}$
  $S_1 = \{A\}$, $S_2 = \{B\}$, $S_3 = \{C\}$, $S_4 = \{D\}$, $S_5 = \{E\}$,

  FIND(D)=S_4
  UNION(S_3, S_4) %New set always gets smaller index
  FIND(A)=S_1
  FIND(D)=S_3
  UNION(S_1, S_3)
  UNION(S_2, S_5)

An Example

Initial setting:

- Initially there are n subsets $S_i = \{s_i\}$, $i = 1, \ldots, n$
- Exampe: $S = \{A, B, C, D, E\}$
  $S_1 = \{A\}$, $S_2 = \{B\}$, $S_3 = \{C\}$, $S_4 = \{D\}$, $S_5 = \{E\}$,

  FIND(D)=S_4
  UNION(S_3, S_4) %New set always gets smaller index
  FIND(A)=S_1
  FIND(D)=S_3
  UNION(S_1, S_3)
  UNION(S_2, S_5)
  FIND(D)=S_1

An Example

Initial setting:

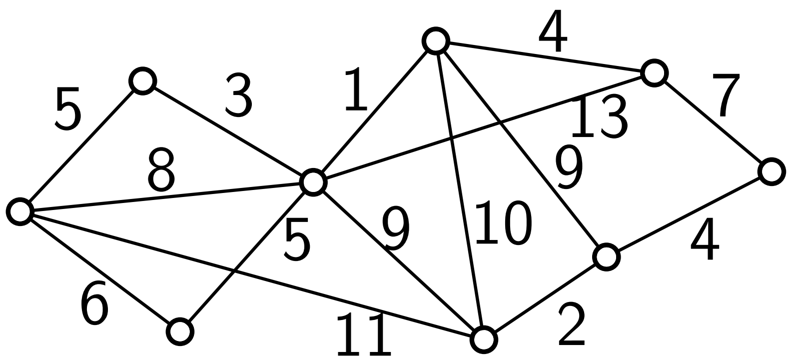
- Initially there are n subsets $S_i = \{s_i\}$, $i = 1, \ldots, n$
- Exampe: $S = \{A, B, C, D, E\}$
  $S_1 = \{A\}$, $S_2 = \{B\}$, $S_3 = \{C\}$, $S_4 = \{D\}$, $S_5 = \{E\}$,

FIND(D)=S_4
UNION(S_3, S_4) %New set always gets smaller index
FIND(A)=S_1
FIND(D)=S_3
UNION(S_1, S_3)
UNION(S_2, S_5)
FIND(D)=S_1
FIND(E)=S_2 $\Rightarrow$ $S_1 = \{A, C, D\}, S_2 = \{B, E\}$

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight

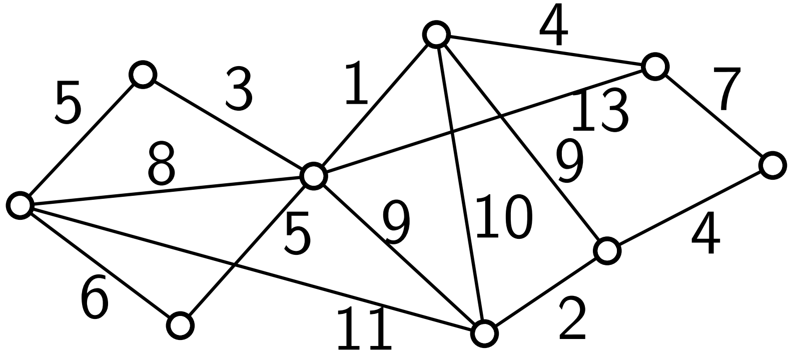


More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight



More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

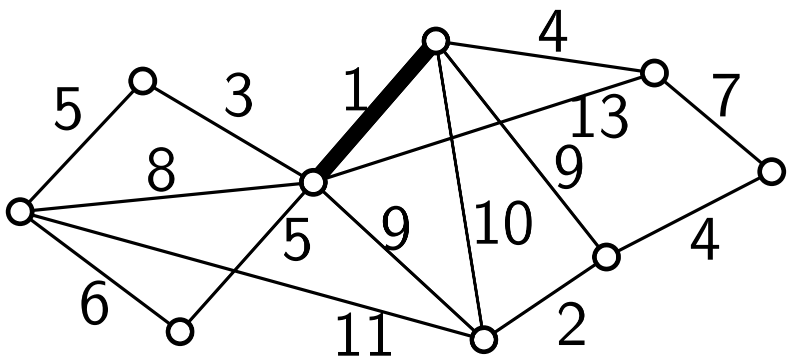
Algorithm:

- Greedy Algorithm: Add short edges (in increasing order), but only if they **don't close a cycle**, until we get a tree.
- We need to check if an edge $e = p_i, p_j$ closes a cycle!

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight



More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

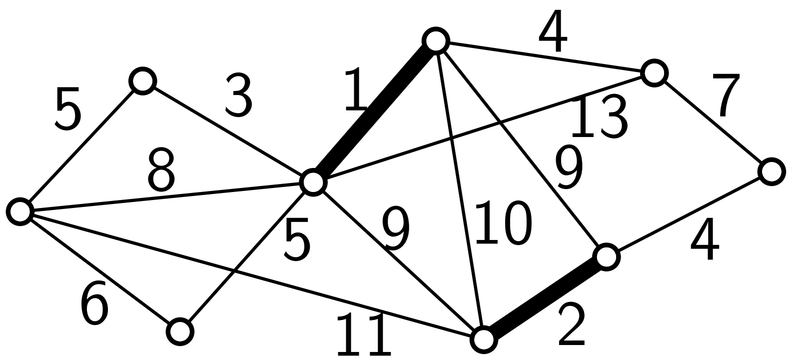
Algorithm:

- Greedy Algorithm: Add short edges (in increasing order), but only if they **don't close a cycle**, until we get a tree.
- We need to check if an edge $e = p_i, p_j$ closes a cycle!

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight



More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

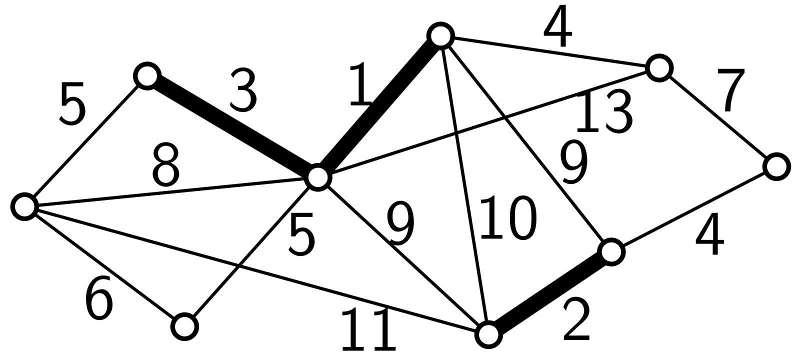
Algorithm:

- Greedy Algorithm: Add short edges (in increasing order), but only if they **don't close a cycle**, until we get a tree.
- We need to check if an edge $e = p_i, p_j$ closes a cycle!

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight



More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

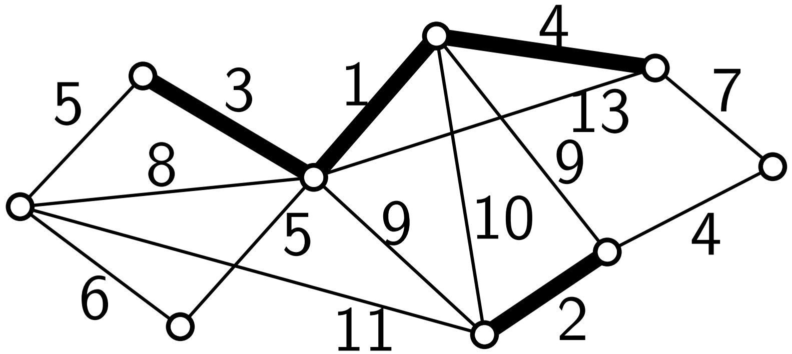
Algorithm:

- Greedy Algorithm: Add short edges (in increasing order), but only if they **don't close a cycle**, until we get a tree.
- We need to check if an edge $e = p_i, p_j$ closes a cycle!

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight



More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

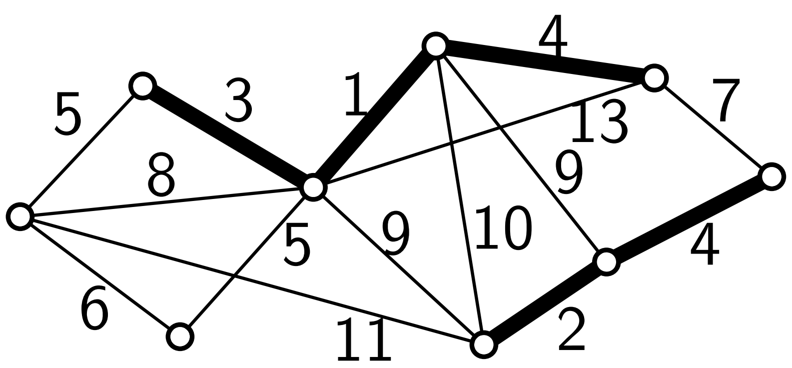
Algorithm:

- Greedy Algorithm: Add short edges (in increasing order), but only if they **don't close a cycle**, until we get a tree.
- We need to check if an edge $e = p_i, p_j$ closes a cycle!

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight



More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

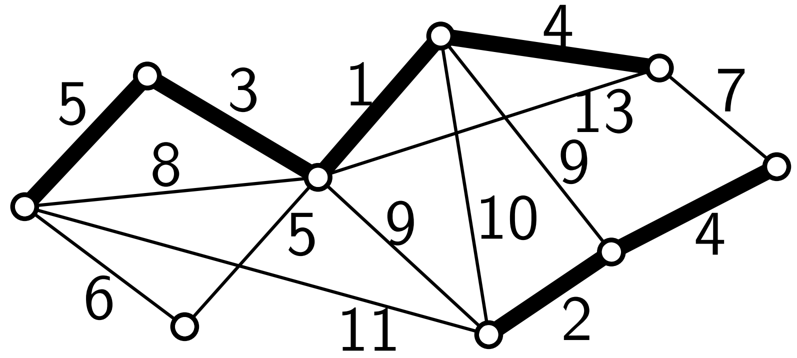
Algorithm:

- Greedy Algorithm: Add short edges (in increasing order), but only if they **don't close a cycle**, until we get a tree.
- We need to check if an edge $e = p_i, p_j$ closes a cycle!

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight



More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

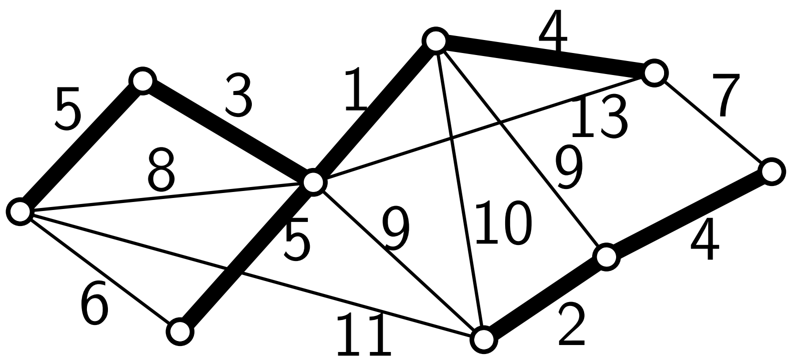
Algorithm:

- Greedy Algorithm: Add short edges (in increasing order), but only if they **don't close a cycle**, until we get a tree.
- We need to check if an edge $e = p_i, p_j$ closes a cycle!

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight



More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

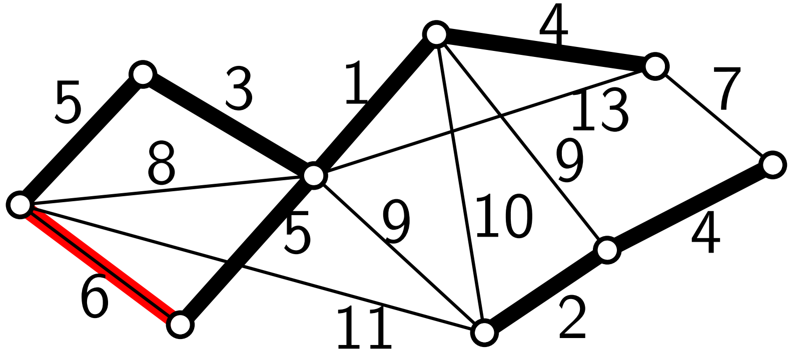
Algorithm:

- Greedy Algorithm: Add short edges (in increasing order), but only if they **don't close a cycle**, until we get a tree.
- We need to check if an edge $e = p_i, p_j$ closes a cycle!

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight



More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

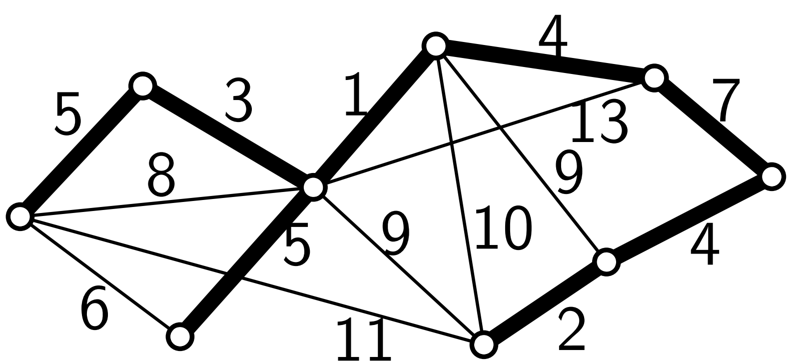
Algorithm:

- Greedy Algorithm: Add short edges (in increasing order), but only if they **don't close a cycle**, until we get a tree.
- We need to check if an edge $e = p_i, p_j$ closes a cycle!

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight



More details about minimum spanning trees (algorithms, correctness etc.) in the lecture *Design and Analysis of Algorithms*

Algorithm:

- Greedy Algorithm: Add short edges (in increasing order), but only if they **don't close a cycle**, until we get a tree.
- We need to check if an edge $e = p_i, p_j$ closes a cycle!

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight

How to check if an edge $e = p_i, p_j$ closes a cycle?

- Use UNION-FIND
- Initially each point is in its own set: $S_i = \{p_i\}$
- When inserting the edge $e = p_i, p_j$: UNION(S_{p_i},S_{p_j})
- Edge $e = p_i, p_j$ closes a cycle $\Leftrightarrow$ FIND(p_i)=FIND(p_j)

Motivation: Minimum Spanning Tree

Minimum Spanning Tree:

- Let G be a weighted graph.
- Compute a spanning tree $\in G$ of minimum weight

How to check if an edge $e = p_i, p_j$ closes a cycle?

- Use UNION-FIND
- Initially each point is in its own set: $S_i = \{p_i\}$
- When inserting the edge $e = p_i, p_j$: UNION(S_{p_i},S_{p_j})
- Edge $e = p_i, p_j$ closes a cycle $\Leftrightarrow$ FIND(p_i)=FIND(p_j)

Runtime:

- If G has n vertices and m edges, then we need $n-1$ unions and $O(m)$ finds

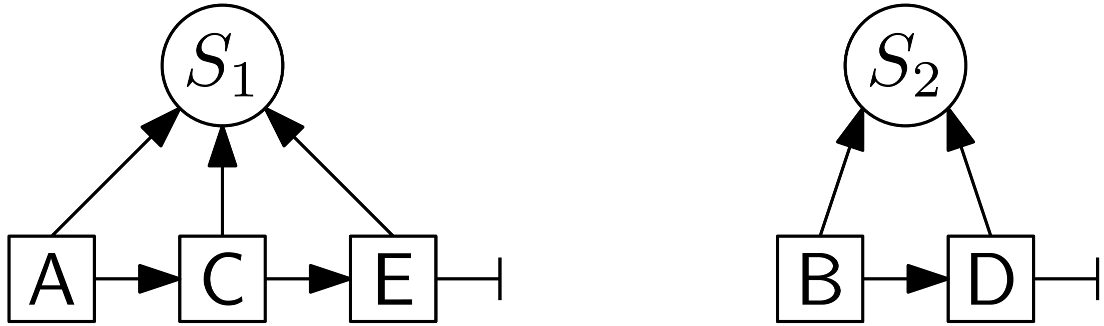
Version I: Fast FIND

Constant time FIND operation

Fast FIND

Store each set as a tree:

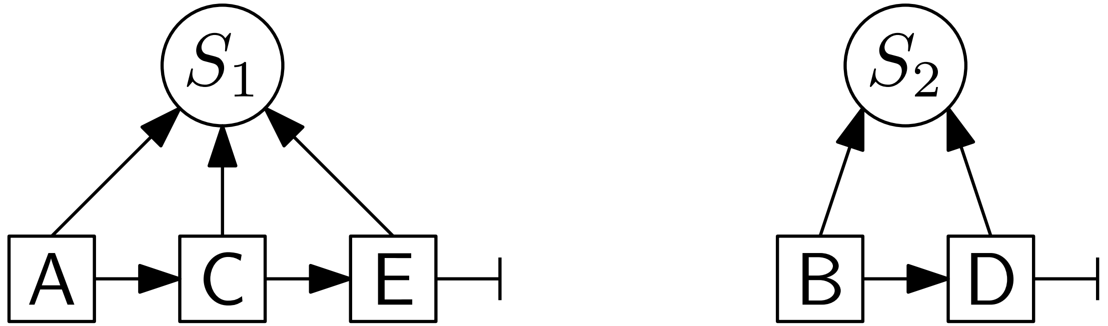
- Each set S_i is a tree of height 1
- Leaves of the tree are the elements (in arbitrary order)
- Each leaf points to the root and to the next element



Fast FIND

Store each set as a tree:

- Each set S_i is a tree of height 1
- Leaves of the tree are the elements (in arbitrary order)
- Each leaf points to the root and to the next element

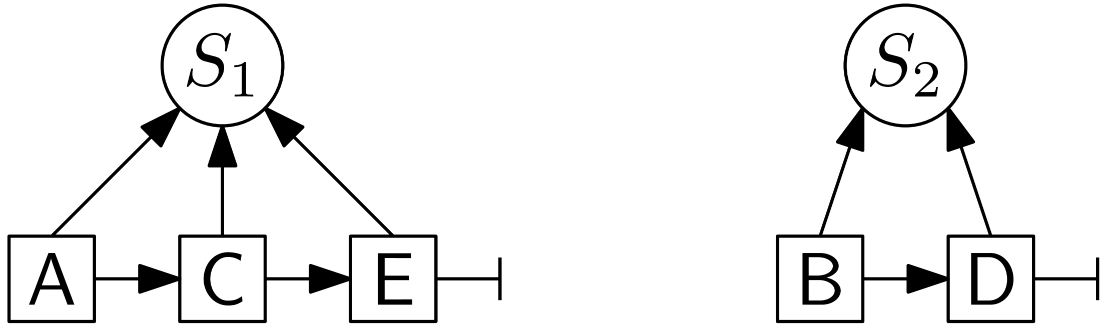


- FIND(x): inspect pointer to root in $\Theta(1)$ time

Fast FIND

Store each set as a tree:

- Each set S_i is a tree of height 1
- Leaves of the tree are the elements (in arbitrary order)
- Each leaf points to the root and to the next element

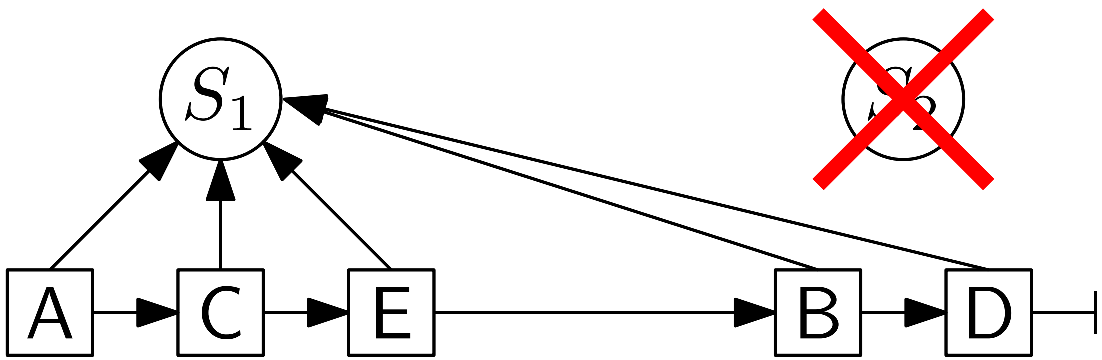


- FIND(x): inspect pointer to root in $\Theta(1)$ time
- UNION(S_i, S_j): update pointers of one set and link elements in time $\Theta(|S_i|)$ or $\Theta(|S_j|)$

Fast FIND

Store each set as a tree:

- Each set S_i is a tree of height 1
- Leaves of the tree are the elements (in arbitrary order)
- Each leaf points to the root and to the next element



Remark: root contains pointers to first and last element

- FIND(x): inspect pointer to root in $\Theta(1)$ time
- UNION(S_i, S_j): update pointers of one set and link elements in time $\Theta(|S_i|)$ or $\Theta(|S_j|)$

Fast FIND

Analysis UNION operation:

- Assume UNION(S_i, S_j) updates the pointers of S_j
- The new sets gets index i

Fast FIND

Analysis UNION operation:

- Assume UNION(S_i, S_j) updates the pointers of S_j
- The new sets gets index i

Then the sequence UNION(S_{i-1}, S_i) for $i = n$ down to 2 needs $1 + 2 + 3 + \ldots + (n-1) = \Theta(n^2)$ pointer updates

- Then a UNION() operation needs $\Theta(n)$ time in average.

Fast FIND

Analysis UNION operation:

- Assume UNION(S_i, S_j) updates the pointers of S_j
- The new sets gets index i

Then the sequence UNION(S_{i-1}, S_i) for $i = n$ down to 2 needs $1 + 2 + 3 + \ldots + (n-1) = \Theta(n^2)$ pointer updates

- Then a UNION() operation needs $\Theta(n)$ time in average.

Idea: update smaller set (make u UNION() operations)

- For a fixed element $x \in S$ in each UNION() step which 'moves' x the size of the resulting set at least doubles
- After u_x UNION() steps of x we get $2^{u_x} \leq |S_{u_x}| \leq u$
- $u_x = O(\log u)$: each element is updated $O(\log u)$ times
- So u UNION() operations need $O(u \log u)$ time

Fast FIND

Lemma (Fast FIND):

- A sequence of $n-1$ unions and f finds can be performed in $O(n \log n + f)$ time.

Fast FIND

Lemma (Fast FIND):

- A sequence of $n-1$ unions and f finds can be performed in $O(n \log n + f)$ time.

Remark: The minimum spanning tree of a graph with n vertices and m edges can be computed in $O(n \log n + m)$ time.

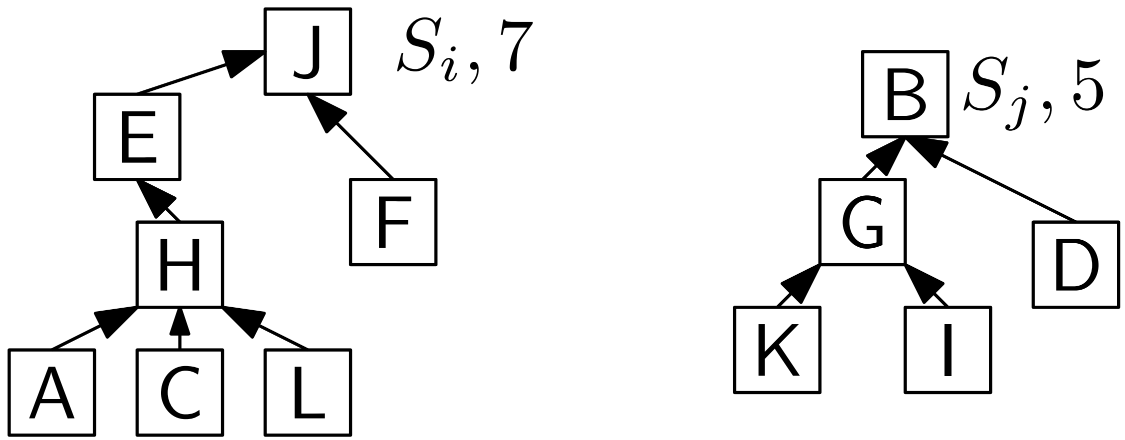
Version II: Fast UNION

Constant time UNION operation

Fast UNION

Store each set as a tree:

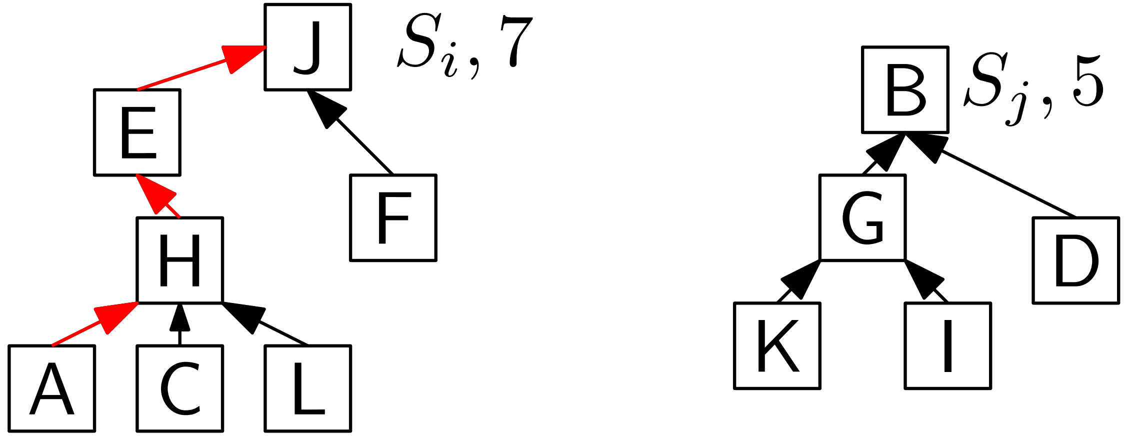
- Each set S_i is a tree of height h_i
- Nodes are the elements with pointers towards the root
- Root contains the id S_i and the number of nodes



Fast UNION

Store each set as a tree:

- Each set S_i is a tree of height h_i
- Nodes are the elements with pointers towards the root
- Root contains the id S_i and the number of nodes

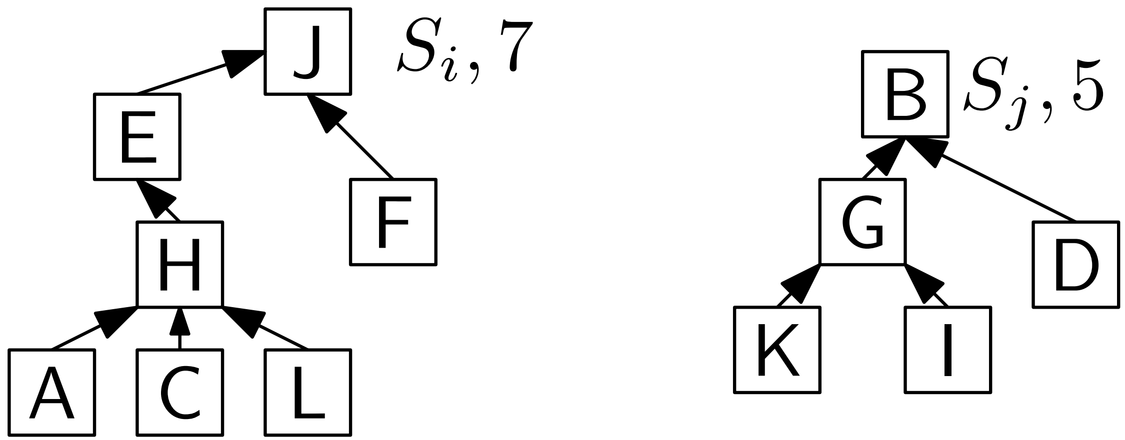


- FIND(x): follow pointers towards the root, then inspect root. Time needed: $\Theta(h_i)$

Fast UNION

Store each set as a tree:

- Each set S_i is a tree of height h_i
- Nodes are the elements with pointers towards the root
- Root contains the id S_i and the number of nodes

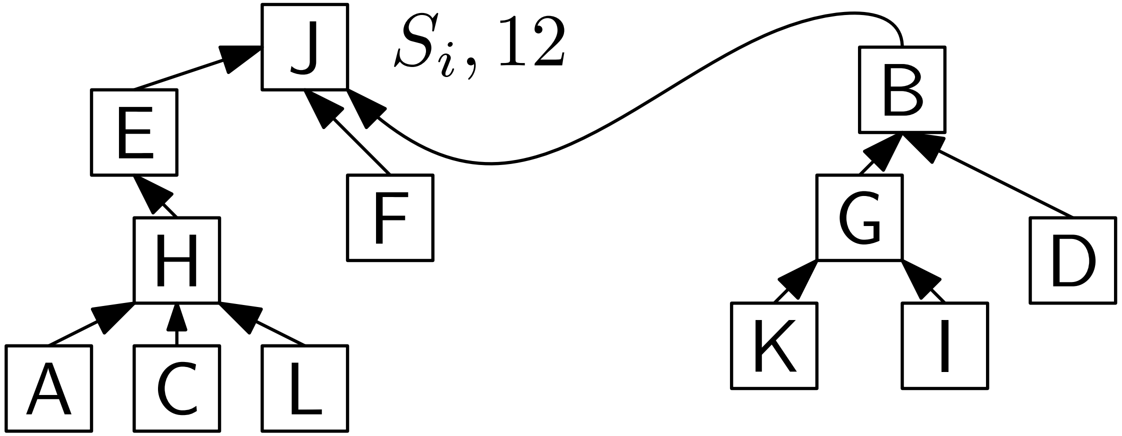


- FIND(x): follow pointers towards the root, then inspect root. Time needed: $\Theta(h_i)$
- UNION(S_i, S_j): connect root of the smaller tree as child to the root of the larger tree, update treesize. Time $\Theta(1)$

Fast UNION

Store each set as a tree:

- Each set S_i is a tree of height h_i
- Nodes are the elements with pointers towards the root
- Root contains the id S_i and the number of nodes



- FIND(x): follow pointers towards the root, then inspect root. Time needed: $\Theta(h_i)$
- UNION(S_i, S_j): connect root of the smaller tree as child to the root of the larger tree, update treesize. Time $\Theta(1)$

Fast UNION

Analysis of depth of tree:

- For a (sub)tree T let h_T be its height, and $|T|$ its size
- Claim: $|T| \geq 2^{h_T}$

Fast UNION

Analysis of depth of tree:

- For a (sub)tree T let h_T be its height, and $|T|$ its size
- Claim: $|T| \geq 2^{h_T}$

- Proof by induction over h_T
- Induction base: $h_T = 0$ (single node), $|T| = 1 \geq 2^0 = 1$

Fast UNION

Analysis of depth of tree:

- For a (sub)tree T let h_T be its height, and $|T|$ its size
- Claim: $|T| \geq 2^{h_T}$

- Proof by induction over h_T
- Induction base: $h_T = 0$ (single node), $|T| = 1 \geq 2^0 = 1$

- Induction step: $h_T > 0$
- Let s be the son with heighest subtree T_s, $h_{T_s} = h_T - 1$
- By induction we have $|T_s| \geq 2^{h_{T_s}} = 2^{h_T - 1}$
- Consider UNION() when s got a son of the root of $T \setminus T_s$: by the 'smaller' rule we know $|T \setminus T_s| \geq |T_s|$
- Thus $|T| \geq 2|T_s| = 2 \times 2^{h_T - 1} = 2^{h_T}$, q.e.d.

Fast UNION

- From $|T| \geq 2^{h_T}$ we have $h_T = \log_2 |T| = O(\log n)$
- FIND() needs only $O(\log n)$ time

Fast UNION

- From $|T| \geq 2^{h_T}$ we have $h_T = \log_2 |T| = O(\log n)$
- FIND() needs only $O(\log n)$ time

Lemma (Fast UNION):

- A sequence of $n-1$ unions and f finds can be performed in $O(n + f \log n)$ time.

Remark: For few finds ($f = o(n)$) this is faster than the previous approach

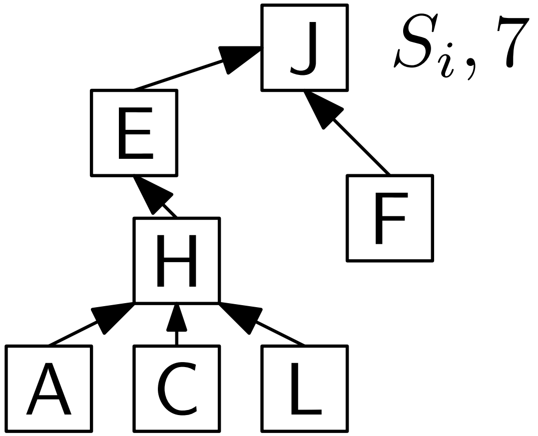
Version III: Almost Linear

Fast UNION and FIND operation

Almost Linear

Idea: shorten path to root

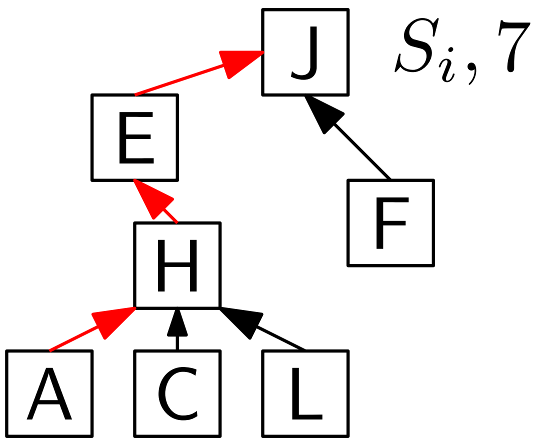
- For each FIND() of the previous approach link the visited nodes directly to the root



Almost Linear

Idea: shorten path to root

- For each FIND() of the previous approach link the visited nodes directly to the root

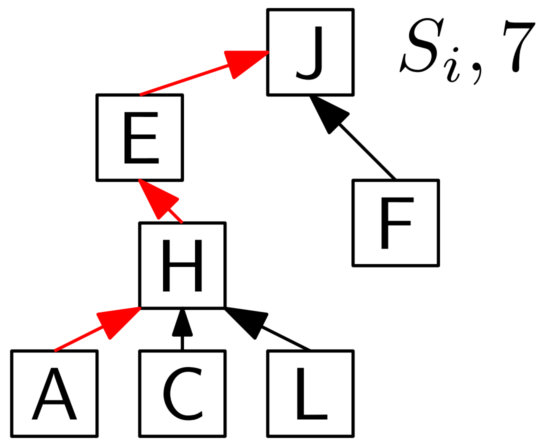


- First search for the root r as usual

Almost Linear

Idea: shorten path to root

- For each FIND() of the previous approach link the visited nodes directly to the root

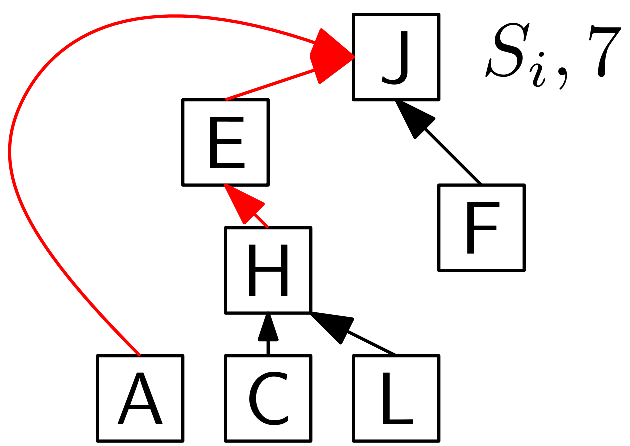


- First search for the root r as usual
- Then search a second time and relink all pointers to r

Almost Linear

Idea: shorten path to root

- For each FIND() of the previous approach link the visited nodes directly to the root

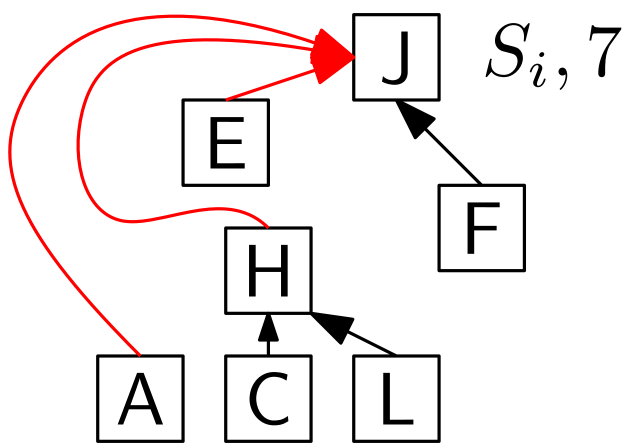


- First search for the root r as usual
- Then search a second time and relink all pointers to r

Almost Linear

Idea: shorten path to root

- For each FIND() of the previous approach link the visited nodes directly to the root

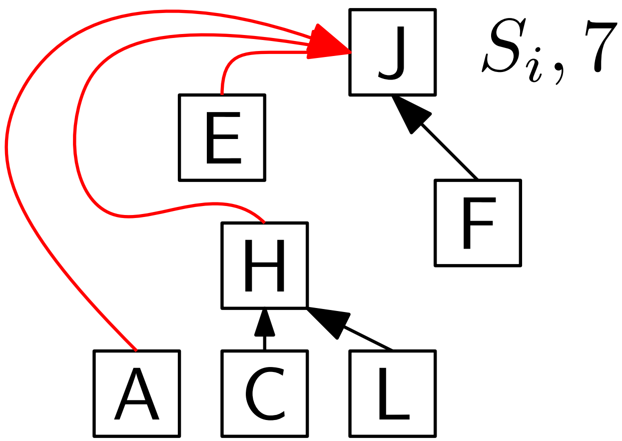


- First search for the root r as usual
- Then search a second time and relink all pointers to r

Almost Linear

Idea: shorten path to root

- For each FIND() of the previous approach link the visited nodes directly to the root

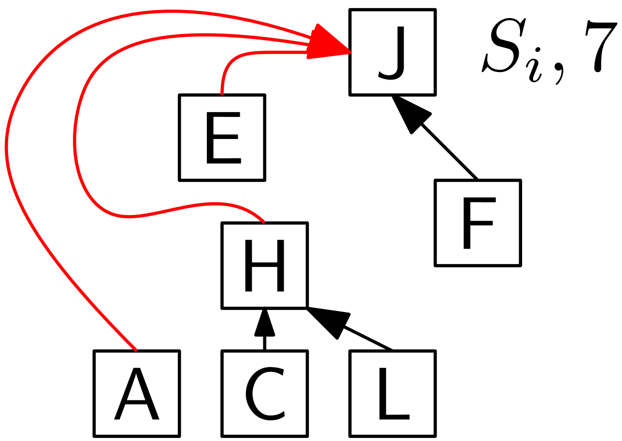


- First search for the root r as usual
- Then search a second time and relink all pointers to r

Almost Linear

Idea: shorten path to root

- For each FIND() of the previous approach link the visited nodes directly to the root



- First search for the root r as usual
- Then search a second time and relink all pointers to r
- Further FIND() operations get potentially much faster

Almost Linear

Lemma (Almost Linear):

- A sequence of $n-1$ unions and $f \geq n$ finds can be performed in $O(f \cdot \alpha(f,n))$ time.

- Lemma without proof
- $\alpha(f,n)$ is the inverse of the Ackermann-function $A(i,n)$

Almost Linear

Lemma (Almost Linear):

- A sequence of $n-1$ unions and $f \geq n$ finds can be performed in $O(f \cdot \alpha(f,n))$ time.

- Lemma without proof
- $\alpha(f,n)$ is the inverse of the Ackermann-function $A(i,n)$

- $A(1,n) = 2^n$ for $n \geq 1$
- $A(i,1) = A(i-1,2)$ for $i \geq 2$
- $A(i,n) = A(i-1, A(i,n-1))$ for $i,n \geq 2$

Example: $A(2,n) = A(1, A(2,n-1)) = 2^{A(2,n-1)} =$
$2^{2^{A(2,n-2)}} = \ldots = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$ (n times power of 2)

Almost Linear

Reasonable input: $\alpha(f, n) < 4$

- $\alpha(f, n)$ grows more than moderate:
- For $f \geq 3$ we have $\alpha(f, n) < \alpha(2, n) = \log_2^* n - 1$

Almost Linear

Reasonable input: $\alpha(f,n) < 4$

- $\alpha(f,n)$ grows more than moderate:
- For $f \geq 3$ we have $\alpha(f,n) < \alpha(2,n) = \log_2^* n - 1$
- $\log_2^* n = \min\{i \in \mathbb{N} | \log_2^{(i)} n \leq 1\}$ (iterative logarithm)
- $\log_2^{(i)} n = \log_2(\log_2(\log_2(\ldots \log_2 n)))$ i times
- Example: $\log_2^*(65536) = 1 + \log_2^*(16) = 2 + \log_2^*(4) = 3 + \log_2^*(2) = 4$
- For $n \leq 2^{65536} \approx 10^{19728}$ we have $\log_2^* n \leq 5$.

Almost Linear

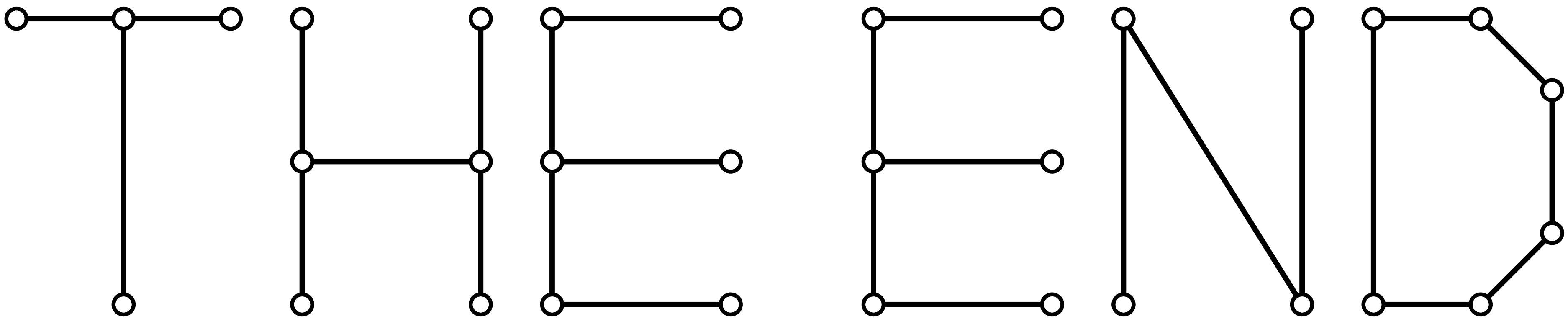
Reasonable input: $\alpha(f, n) < 4$

- $\alpha(f, n)$ grows more than moderate:
- For $f \geq 3$ we have $\alpha(f, n) < \alpha(2, n) = \log_2^* n - 1$

- $\log_2^* n = \min\{i \in \mathbb{N} | \log_2^{(i)} n \leq 1\}$ (iterative logarithm)
- $\log_2^{(i)} n = \log_2(\log_2(\log_2(\ldots \log_2 n)))$ i times
- Example: $\log_2^*(65536) = 1 + \log_2^*(16) = 2 + \log_2^*(4) = 3 + \log_2^*(2) = 4$
- For $n \leq 2^{65536} \approx 10^{19728}$ we have $\log_2^* n \leq 5$.

- So for any reasonable input $\log_2^* n \leq 5$
- Thus $O(f\alpha(f, n))$ is linear in f for all practical problems
- A lower bound of $\Omega(f\alpha(f, n))$ can be shown

UNION - FINAL



Lemma (Fast FIND): A sequence of $n-1$ unions and f finds can be performed in $O(n \log n + f)$ time.

Lemma (Fast UNION): A sequence of $n-1$ unions and f finds can be performed in $O(n + f \log n)$ time.

Lemma (Almost Linear): A sequence of $n-1$ unions and $f \geq n$ finds can be performed in $O(f \cdot \alpha(f, n))$ time.