Design and analysis of algorithms Lecture 20& 21

Edyta Szymańska

edyta@cc.gatech.edu

Disjoint-sets data structure

Set operations:

- MakeSet(x): creates set $\{x\}$
- Find(x): to which set does x belong?
- Union(x, y): merge sets containing x and y.

Disjoint-sets data structure

- An array:
  `Component[s]:=the name of the set containing` s
  Cost:
  MakeSet(x): $O(1)$
  Find(x): $O(1)$
  Union(x, y): $O(n)$

Disjoint-sets data structure

- An array:
  `Component[s]:=the name of the set containing` s
  Cost:

  MakeSet(x): $O(1)$
  Find(x): $O(1)$
  Union(x, y): $O(n)$

- A directed tree (disjoint-sets forest):
  Represent every set by a directed tree, nodes in an arbitrary order, with root as a *representative*

Disjoint-sets data structure

- An array:
  `Component[s]:=the name of the set containing` s
  Cost:
  MakeSet(x): $O(1)$
  Find(x): $O(1)$
  Union(x, y): $O(n)$
- A directed tree (disjoint-sets forest):
  Represent every set by a directed tree, nodes in an arbitrary order, with root as a *representative*

Example:

Disjoint-sets data structure

- A directed tree:
  for all $x:\ \pi(x)$-parent pointer, $rank(x)$-the height of the subtree rooted at x

  MAKESET(x):
  $\pi(x) := x$
  $rank(x) := 0$

  FIND(x):
  while $x \neq \pi(x)$ do $x = \pi(x)$
  return x

  UNION(x, y): ??

Disjoint-sets data structure

- A directed tree:
  for all $x:\ \pi(x)$-parent pointer, $rank(x)$-the height of the subtree rooted at x

  MAKESET(x):
  $\pi(x) := x$
  $rank(x) := 0$ $O(1)$

  FIND(x):
  while $x \neq \pi(x)$ do $x = \pi(x)$
  return x $O(height(Tree))$

  UNION(x, y): ??

Union by rank

Idea: by rank, i.e. *make the root of the shorter tree point to the root of the taller tree*

Union by rank

Idea: by rank, i.e. *make the root of the shorter tree point to the root of the taller tree*

UNION(x, y)

$r_x := Find(x)$

$r_y := Find(y)$

if $r_x = r_y$ then return

if $rank(r_y) > rank(r_x)$ then

$\pi(r_x) = r_y$

else

$\pi(r_y) = r_x$

if $rank(r_x) = rank(r_y)$ then

$rank(r_x) := rank(r_x) + 1$

Union by rank

Idea: by rank, i.e. *make the root of the shorter tree point to the root of the taller tree*

UNION(x, y)

$r_x := Find(x)$

$r_y := Find(y)$

if $r_x = r_y$ then return

if $rank(r_y) > rank(r_x)$ then

$\pi(r_x) = r_y$

else

$\pi(r_y) = r_x$

if $rank(r_x) = rank(r_y)$ then

$rank(r_x) := rank(r_x) + 1$

see Example

Union by rank

Properties:

(1) For any x that is not a root, we have $rank(x) < rank(\pi(x))$

(2) any root node of rank k has at least 2^k nodes in its tree

(3) if there are n elements overall, there can be at most $\frac{n}{2^k}$ nodes of rank k

Union by rank

Properties:

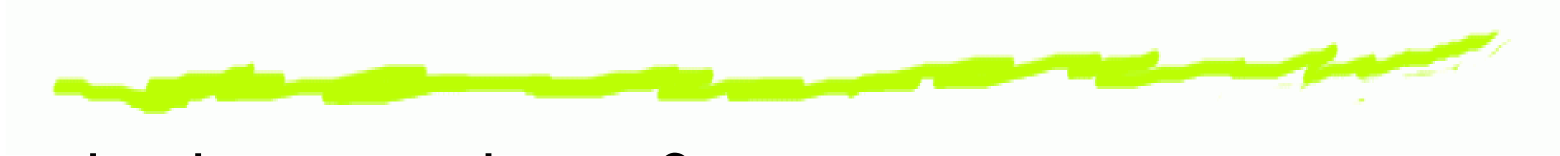
(1) For any x that is not a root, we have $rank(x) < rank(\pi(x))$

(2) any root node of rank k has at least 2^k nodes in its tree

(3) if there are n elements overall, there can be at most $\frac{n}{2^k}$ nodes of rank k

Thus, from (3), the maximum rank is $\log n$ and any disjoint-sets operation takes $O(\log n)$

Path compression



How to make the trees shorter?

Path compression

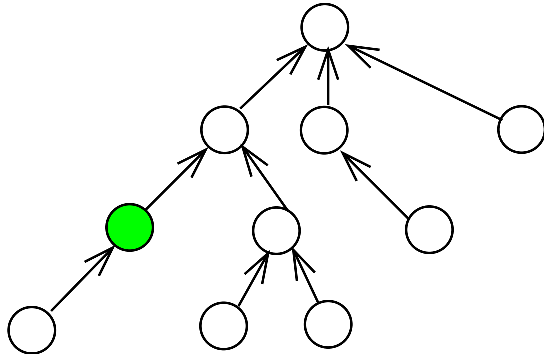
How to make the trees shorter?
Place for improvement: FIND operation
modify it so that all pointers of nodes visited by Find (on a path to a root) will point directly to the root (path compression.)

Path compression

How to make the trees shorter?
Place for improvement: FIND operation
modify it so that all pointers of nodes visited by Find (on a path to a root) will point directly to the root (path compression.)
This doubles the time of Find and is easy to implement:

Path compression

How to make the trees shorter?
Place for improvement: FIND operation
modify it so that all pointers of nodes visited by Find (on a path to a root) will point directly to the root (path compression.)
This doubles the time of Find and is easy to implement:



Path compression

How to make the trees shorter?

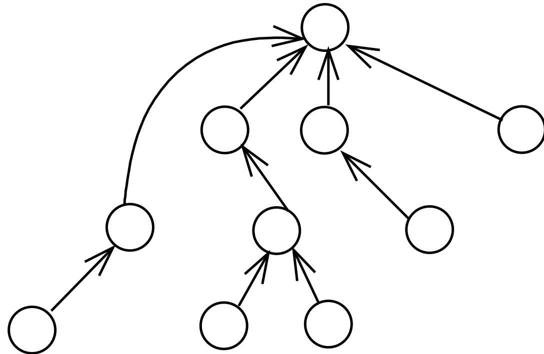
Place for improvement: FIND operation

modify it so that all pointers of nodes visited by Find (on a path to a root) will point directly to the root (path compression.)

This doubles the time of Find and is easy to implement:

Path compression

How to make the trees shorter?
Place for improvement: FIND operation
modify it so that all pointers of nodes visited by Find (on a path to a root) will point directly to the root (path compression.)
This doubles the time of Find and is easy to implement:



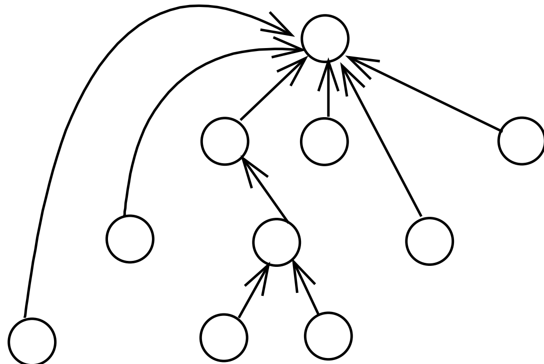
Path compression

How to make the trees shorter?

Place for improvement: FIND operation

modify it so that all pointers of nodes visited by Find (on a path to a root) will point directly to the root (path compression.)

This doubles the time of Find and is easy to implement:



Path compression

How to make the trees shorter?
Place for improvement: FIND operation
modify it so that all pointers of nodes visited by Find (on a path to a root) will point directly to the root (path compression.)
This doubles the time of Find and is easy to implement:

FIND(x):
if $x \neq \pi(x)$ do $\pi(x)$:=FIND$(\pi(x))$
return x

Path compression

How to make the trees shorter?

Place for improvement: FIND operation

modify it so that all pointers of nodes visited by Find (on a path to a root) will point directly to the root (path compression.)

This doubles the time of Find and is easy to implement:

FIND(x):

if $x \neq \pi(x)$ do $\pi(x) :=$FIND$(\pi(x))$

return x

With this modification later FIND operations on the path will be cheaper.

Union by rank analysis

Properties (1)-(3) continue to hold.

(1) For any x that is not a root, we have $rank(x) < rank(\pi(x))$

(2) any root node of rank k has at least 2^k nodes in its tree

(3) if there are n elements overall, there can be at most $\frac{n}{2^k}$ nodes of rank k

Union by rank analysis

Properties (1)-(3) continue to hold. *

(1) For any x that is not a root, we have $rank(x) < rank(\pi(x))$

(2) any root node of rank k has at least 2^k nodes in its tree

(3) if there are n elements overall, there can be at most $\frac{n}{2^k}$ nodes of rank k

Union by rank analysis

Properties (1)-(3) continue to hold.

(1) For any x that is not a root, we have $rank(x) < rank(\pi(x))$

(2) any root node of rank k has at least 2^k nodes in its tree

(3) if there are n elements overall, there can be at most $\frac{n}{2^k}$ nodes of rank k

BUT: $rank(x)$ is no longer storing the same information because we only update it for the root in UNION and it is not changed during path compression in FIND. It is now the rank of x when it was the root for the last time before the path compression.

Union by rank analysis

Properties (1)-(3) continue to hold.

(1) For any x that is not a root, we have $rank(x) < rank(\pi(x))$

(2) any root node of rank k has at least 2^k nodes in its tree

(3) if there are n elements overall, there can be at most $\frac{n}{2^k}$ nodes of rank k

BUT: $rank(x)$ is no longer storing the same information because we only update it for the root in UNION and it is not changed during path compression in FIND. It is now the rank of x when it was the root for the last time before the path compression.

Union by rank analysis

We will prove that any sequence of m UNION and FIND operations on n elements take at most $O((m+n)\log^* n)$ steps, where $\log^* n$ is the number of times one must iterate the log function on n before it will result in a number less than or equal to 1.

Union by rank analysis

We will prove that any sequence of m UNION *and* FIND *operations on* n *elements take at most* $O((m+n)\log^* n)$ *steps, where* $\log^* n$ *is the number of times one must iterate the log function on* n *before it will result in a number less than or equal to 1.*

$\log^* n \leq 5$ for all $n \leq 2^{2^{16}} \approx 10^{19728}$

Union by rank analysis

We will prove that any sequence of m Union *and* Find *operations on n elements take at most $O((m+n)\log^* n)$ steps, where $\log^* n$ is the number of times one must iterate the log function on n before it will result in a number less than or equal to 1.*

$\log^* n \leq 5$ for all $n \leq 2^{2^{16}} \approx 10^{19728}$

To prove it, group the elements according to their ranks in the following way:

$[0],\ [1],\ (1,2],\ (2,3,4],\ (4,\ldots,16],\ (16,\ldots,2^{16}],\ldots,(k,\ldots,2^k]$

Union by rank analysis

We will prove that any sequence of m UNION and FIND operations on n elements take at most $O((m+n)\log^* n)$ steps, where $\log^* n$ is the number of times one must iterate the log function on n before it will result in a number less than or equal to 1.

$\log^* n \leq 5$ for all $n \leq 2^{2^{16}} \approx 10^{19728}$

To prove it, group the elements according to their ranks in the following way:

$[0],\ [1],\ (1,2],\ (2,3,4],\ (4,\ldots,16],\ (16,\ldots,2^{16}],\ldots,(k,\ldots,2^k]$

Notice that the number of groups reveals the mysterious $\log^* n$.

Union by rank analysis

Fact: *no more than* $\frac{n}{2^k}$ *elements have rank in* $(k, 2^k]$.

Union by rank analysis

Fact: *no more than* $\frac{n}{2^k}$ *elements have rank in* $(k, 2^k]$.

Proof: $\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots + \frac{n}{2^{2^k}} \leq \frac{n}{2^k}(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots) \leq \frac{n}{2^k}$.

Union by rank analysis

Fact: *no more than* $\frac{n}{2^k}$ *elements have rank in* $(k, 2^k]$.

Proof: $\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots + \frac{n}{2^{2^k}} \leq \frac{n}{2^k}\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) \leq \frac{n}{2^k}$.

Each operation apart from FIND takes constant time ($O(m)$ in total). We will now calculate the time taken by m FIND operations.

Union by rank analysis

Fact: *no more than* $\frac{n}{2^k}$ *elements have rank in* $(k, 2^k]$.

Proof: $\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots + \frac{n}{2^{2^k}} \leq \frac{n}{2^k}(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots) \leq \frac{n}{2^k}$.

Each operation apart from FIND takes constant time ($O(m)$ in total). We will now calculate the time taken by m FIND operations.

For every x, the cost of FIND(x) is proportional the the number of nodes on the find path from x to the root(before path compression).

Union by rank analysis

We will count up the total cost of FIND using accounting method:

Union by rank analysis

We will count up the total cost of FIND using accounting method:
$\forall x_i, i = 0, 1, \ldots, l$ on the find path, x_i will pay $1\$$ into 1 of 4 accounts:

ROOT	(for root nodes)
CHILD	(direct child of a root)
GROUP	(x_i s.t $\pi(x_i)$ is in a different group than x_i)
PATH	(x_i and $\pi(x_i)$ are in the same group)

Union by rank analysis

We will count up the total cost of FIND using accounting method:

$\forall x_i, i = 0, 1, \ldots, l$ on the find path, x_i will pay $1\$$ into 1 of 4 accounts:

ROOT	(for root nodes)
CHILD	(direct child of a root)
GROUP	(x_i s.t $\pi(x_i)$ is in a different group than x_i)
PATH	(x_i and $\pi(x_i)$ are in the same group)

see Example

Union by rank analysis

During a single FIND we pay $1\$$ into ROOT, $\leq 1\$$ into CHILD and $\leq 1\$$ into GROUP (for each of $\log^* n$ groups).

Union by rank analysis

During a single FIND we pay $1\$$ into ROOT, $\leq 1\$$ into CHILD and $\leq 1\$$ into GROUP (for each of $\log^* n$ groups).
Total (on 3 accounts), after m operations is $\leq 2m + m \log^* n$

Union by rank analysis

Total (on 3 accounts), after m operations is $\leq 2m + m \log^* n$

PATH account ?

Union by rank analysis

Total (on 3 accounts), after m operations is $\leq 2m + m\log^* n$

PATH account ?
If $x \in (k, 2^k]$ it will pay to the PATH account at most 2^k times before its parent jumps into a different group.

Union by rank analysis

Total (on 3 accounts), after m operations is $\leq 2m + m\log^* n$

PATH account ?
If $x \in (k, 2^k]$ it will pay to the PATH account at most 2^k times before its parent jumps into a different group.
The number of nodes in the group is $\leq \frac{n}{2^k}$ and thus they pay $\frac{n}{2^k} \cdot 2^k = O(n)$, which gives $O(n \log^* n)$ in total.

Union by rank analysis

Total (on 3 accounts), after m operations is $\leq 2m + m \log^* n$

PATH account ?
If $x \in (k, 2^k]$ it will pay to the PATH account at most 2^k times before its parent jumps into a different group.
The number of nodes in the group is $\leq \frac{n}{2^k}$ and thus they pay $\frac{n}{2^k} \cdot 2^k = O(n)$, which gives $O(n \log^* n)$ in total.
Thus, the total time is $O(m + n) \log^* n$.