CSE373: Data Structures & Algorithms
Lecture 10: Implementing Union-Find

Kevin Quinn
Fall 2015
The plan

Last lecture:

• What are *disjoint sets*
  – And how are they “the same thing” as *equivalence relations*

• The union-find ADT for disjoint sets

• Applications of union-find

Now:

• **Basic implementation of the ADT with “up trees”**

• Optimizations that make the implementation much faster
Our goal

• Start with an initial partition of \( n \) subsets
  – Often 1-element sets, e.g., \{1\}, \{2\}, \{3\}, \ldots, \{n\}

• May have \( m \) \textit{find} operations and up to \( n-1 \) \textit{union} operations in any order
  – After \( n-1 \) \textit{union} operations, every \textit{find} returns same 1 set

• If total for all these operations is \( O(m+n) \), then amortized \( O(1) \)
  – We will get very, very close to this
  – \( O(1) \) worst-case is impossible for \textit{find} and \textit{union}
    • Trivial for one or the other
**Up-tree data structure**

- Tree with:
  - No limit on branching factor
  - References from children to parent

- Start with *forest* of 1-node trees

- Possible forest after several unions:
  - Will use roots for set names
Find

`find(x)`:
- Assume we have $O(1)$ access to each node
  - Will use an array where index $i$ holds node $i$
- Start at $x$ and follow parent pointers to root
- Return the root

$\text{find}(6) = 7$
**Union**

$\text{union}(x, y)$:
- Assume $x$ and $y$ are roots
  - If they are not, just find the roots of their trees
- Assume distinct trees (else do nothing)
- Change root of one to have parent be the root of the other
  - Notice no limit on branching factor

$\text{union}(1, 7)$
Simple implementation

- If set elements are contiguous numbers (e.g., 1,2,...,n), use an array of length \( n \) called \( \text{up} \):
  - Starting at index 1 on slides
  - Put in array index of parent, with 0 (or -1, etc.) for a root
- Example:
  - Example:

  \[
  \begin{array}{ccccccc}
  1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  \hline
  \text{up} & 0 & 0 & 0 & 0 & 0 & 0
  \end{array}
  \]

- If set elements are not contiguous numbers, could have a separate dictionary to map elements (keys) to numbers (values)
Implement operations

- Worst-case run-time for `union`?
- Worst-case run-time for `find`?
- Worst-case run-time for $m$ finds and $n-1$ unions?

```c
// assumes x in range 1,n
int find(int x) {
    while(up[x] != 0) {
        x = up[x];
    }
    return x;
}

// assumes x,y are roots
void union(int x, int y) {
    // y = find(y)
    // x = find(x)
    up[y] = x;
}
```
Implement operations

```c
// assumes x in range 1,n
int find(int x) {
    while(up[x] != 0) {
        x = up[x];
    }
    return x;
}

// assumes x,y are roots
void union(int x, int y) {
    // y = find(y)
    // x = find(x)
    up[y] = x;
}
```

- Worst-case run-time for `union`? \( O(1) \) (with our assumption…)
- Worst-case run-time for `find`? \( O(n) \)
- Worst-case run-time for \( m \) finds and \( n-1 \) unions? \( O(m *n) \)
The plan

Last lecture:

• What are disjoint sets
  – And how are they “the same thing” as equivalence relations

• The union-find ADT for disjoint sets

• Applications of union-find

Now:

• Basic implementation of the ADT with “up trees”

• Optimizations that make the implementation much faster
Two key optimizations

1. Improve \texttt{union} so it stays $O(1)$ but makes \texttt{find} $O(\log n)$
   - So $m$ finds and $n-1$ unions is $O(m \log n + n)$
   - \textit{Union-by-size}: connect smaller tree to larger tree

2. Improve \texttt{find} so it becomes even faster
   - Make $m$ finds and $n-1$ unions \textbf{almost} $O(m + n)$
   - \textit{Path-compression}: connect directly to root during finds
The bad case to avoid

union(2,1)
union(3,2)
\vdots
union(n,n-1)

find(1) \ n \ steps!!
Weighted union

Weighted union:
- Always point the *smaller* (total # of nodes) tree to the root of the larger tree
Weighted union

Weighted union:
- Always point the *smaller* (total # of nodes) tree to the root of the larger tree
Weighted union

Weighted union:
- Always point the *smaller* (total # of nodes) tree to the root of the larger tree
Array implementation

Keep the weight (number of nodes in a second array)
- Or have one array of objects with two fields
**Nifty trick**

Actually we do not need a second array…

- Instead of storing 0 for a root, store negation of weight
- So up value < 0 means a root
Bad example? Great example…

1 2 3 n

union(2,1)

2 3 ⋮ n

union(3,2)

⋯

⋯

n

union(n,n-1)

2

1 3

2

1 3 ...

n

find(1) constant here
General analysis

• Showing that one worst-case example is now good is not a proof that the worst-case has improved

• So let’s prove:
  – `union` is still $O(1)$ – this is fairly easy to show
  – `find` is now $O(\log n)$

• Claim: If we use weighted-union, an up-tree of height $h$ has at least $2^h$ nodes
  – Proof by induction on $h$…
Exponential number of nodes

P(h)= With weighted-union, up-tree of height $h$ has at least $2^h$ nodes

Proof by induction on $h$...

• Base case: $h = 0$: The up-tree has 1 node and $2^0 = 1$
• Inductive case: Assume P($h$) and show P($h+1$)
  – A height $h+1$ tree $T$ has at least one height $h$ child $T_1$
  – $T_1$ has at least $2^h$ nodes by induction
  – And $T$ has at least as many nodes not in $T_1$ than in $T_1$
    • Else weighted-union would have
      had $T$ point to $T_1$, not $T_1$ point to $T$ (!!!)
  – So total number of nodes is at least $2^h + 2^h = 2^{h+1}$
The key idea

Intuition behind the proof: No one child can have more than half the nodes.

So, as usual, if number of nodes is exponential in height, then height is logarithmic in number of nodes.

So \textbf{find} is \( O(\log n) \)
The new worst case

n/2 Weighted Unions

n/4 Weighted Unions
The new worst case (continued)

After $n/2 + n/4 + \ldots + 1$ Weighted Unions:

Height grows by 1 a total of $\log n$ times
What about union-by-height

We could store the height of each root rather than number of descendants (weight)

• Still guarantees logarithmic worst-case find
  – Proof left as an exercise if interested

• But does not work well with our next optimization
  – Maintaining height becomes inefficient, but maintaining weight still easy
Two key optimizations

1. Improve union so it stays $O(1)$ but makes find $O(\log n)$
   - So $m$ finds and $n-1$ unions is $O(m \log n + n)$
   - Union-by-size: connect smaller tree to larger tree

2. Improve find so it becomes even faster
   - Make $m$ finds and $n-1$ unions almost $O(m + n)$
   - Path-compression: connect directly to root during finds
Path compression

- Simple idea: As part of a find, change each encountered node’s parent to point directly to root
  - Faster future finds for everything on the path (and their descendants)
// performs path compression
find(i)
  // find root
  r = i
  while up[r] > 0
    r = up[r]

  // compress path
  if i == r
    return r

  old_parent = up[i]
  while (old_parent != r)
    up[i] = r
    i = old_parent
    old_parent = up[i]

  return r
So, how fast is it?

A single worst-case `find` could be $O(\log n)$
- But only if we did a lot of worst-case unions beforehand
- And path compression will make future finds faster

Turns out the amortized worst-case bound is much better than $O(\log n)$
- We won’t prove it – see text if curious
- But we will understand it:
  • How it is *almost* $O(1)$
  • Because total for $m$ finds and $n-1$ unions is *almost* $O(m+n)$
A really slow-growing function

\( \log^* (x) \) is the minimum number of times you need to apply “\( \log \) of \( \log \) of \( \log \) of” to go from \( x \) to a number \( \leq 1 \)

For just about every number we care about, \( \log^*(x) \) is 5 (!)

If \( x \leq 2^{65536} \) then \( \log^* x \leq 5 \)

- \( \log^* 2 = 1 \)
- \( \log^* 4 = \log^* 2^2 = 2 \)
- \( \log^* 16 = \log^* 2^{(2^2)} = 3 \quad (\log(\log(\log(16))) = 1) \)
- \( \log^* 65536 = \log^* 2^{((2^2)^2)} = 4 \quad (\log(\log(\log(\log(65536)))) = 1) \)
- \( \log^* 2^{65536} = \ldots \ldots = 5 \)
Wait…. how big?

Just how big is $2^{65536}$

Well $2^{10} = 1024$
$2^{20} = 1048576$
$2^{30} = 1073741824$
$2^{100} = 1.125\times 10^{15}$
$2^{65536} = \ldots$ pretty big

But its still not technically constant
Almost linear

- Turns out total time for \( m \) finds and \( n-1 \) unions is:
  \( O((m+n)^*(\log^* (m+n))) \)
  - Remember, if \( m+n < 2^{65536} \) then \( \log^* (m+n) < 5 \)

- At this point, it feels almost silly to mention it, but even that bound is not tight…
  - “Inverse Ackerman’s function” grows even more slowly than \( \log^* \)
    - Inverse because Ackerman’s function grows really fast
    - Function also appears in combinatorics and geometry
    - For any number you can possibly imagine, it is < 4
  - Can replace \( \log^* \) with “Inverse Ackerman’s” in bound
Theory and terminology

• Because $\log^*$ or Inverse Ackerman’s grows so incredibly slowly
  – For all practical purposes, amortized bound is constant, i.e.,
    total cost is linear
  – We say “near linear” or “effectively linear”

• Need weighted-union and path-compression for this bound
  – Path-compression changes height but not weight, so they
    interact well

• As always, asymptotic analysis is separate from “coding it up”