Application: Connected Components in a Social Network

- Social networking research studies how relationships between various people can influence behavior.
- Given a set, $S$, of $n$ people, we can define a social network for $S$ by creating a set, $E$, of edges or ties between pairs of people that have a certain kind of relationship. For example, in a friendship network, like Facebook, ties would be defined by pairs of friends.
- A **connected component** in a friendship network is a subset, $T$, of people from $S$ that satisfies the following:
  - Every person in $T$ is related through friendship, that is, for any $x$ and $y$ in $T$, either $x$ and $y$ are friends or there is a chain of friendship, such as through a friend of a friend of a friend, that connects $x$ and $y$.
  - No one in $T$ is friends with anyone outside of $T$. 

© 2015 Goodrich and Tamassia
Example

2 Connected components in a friendship network of some of the key figures in the American Revolutionary War.

Union-Find Operations

A **partition** or **union-find** structure is a data structure supporting a collection of disjoint sets subject to the following operations:

- **makeSet**(e): Create a singleton set containing the element e and return the position storing e in this set
- **union**(A,B): Return the set A U B, naming the result “A” or “B”
- **find**(e): Return the set containing the element e
**Connected Components Algorithm**

The output from this algorithm is an identification, for each person $x$ in $S$, of the connected component to which $x$ belongs.

**Algorithm** UFConnectedComponents($S, E$):

- **Input:** A set, $S$, of $n$ people and a set, $E$, of $m$ pairs of people from $S$ defining pairwise relationships
- **Output:** An identification, for each $x$ in $S$, of the connected component containing $x$

```plaintext
for each $x$ in $S$ do
    makeSet($x$)
for each $(x, y)$ in $E$ do
    if find($x$) $\neq$ find($y$) then
        union(find($x$), find($y$))
for each $x$ in $S$ do
    Output “Person $x$ belongs to connected component” find($x$)
```

The running time of this algorithm is $O(t(n, n+m))$, where $t(j, k)$ is the time for $k$ union-find operations starting from $j$ singleton sets.

---

**Another Application: Maze Construction and Percolation**

**Problem:** Construct a good maze.
A Maze Generator

Algorithm MazeGenerator(G, E):
   Input: A grid, G, consisting of n cells and a set, E, of m “walls,” each of
   which divides two cells, x and y, such that the walls in E initially separate
   and isolate all the cells in G
   Output: A subset, R of E, such that removing the edges in R from E creates a
   maze defined on G by the remaining walls
   while R has fewer than n − 1 edges do
      Choose an edge, (x, y), in E uniformly at random from among those pre-
      viously unchosen
      if find(x) ≠ find(y) then
         union(find(x), find(y))
      Add the edge (x, y) to R
   return R

This is actually is related to the science of percolation theory, which is the study of how liquids permeate porous materials.

For instance, a porous material might be modeled as a three-dimensional n x n x n grid of cells. The barriers separating adjacent pairs of cells might then be removed virtually with some probability p and remain with probability 1 − p. Simulating such a system is another application of union-find structures.

List-based Implementation

- Each set is stored in a sequence represented with a linked-list
- Each node should store an object containing the element and a reference to the set name
Analysis of List-based Representation

- When doing a union, always move elements from the smaller set to the larger set
  - Each time an element is moved it goes to a set of size at least double its old set
  - Thus, an element can be moved at most $O(\log n)$ times
- Total time needed to do $n$ unions and $m$ finds is $O(n \log n + m)$.

Tree-based Implementation

- Each element is stored in a node, which contains a pointer to a set name
- A node $v$ whose set pointer points back to $v$ is also a set name
- Each set is a tree, rooted at a node with a self-referencing set pointer
- For example: The sets “1”, “2”, and “5”:
Union-Find Operations

- To do a **union**, simply make the root of one tree point to the root of the other.

- To do a **find**, follow set-name pointers from the starting node until reaching a node whose set-name pointer refers back to itself.

Union-Find Heuristic 1

- **Union by size:**
  - When performing a **union**, make the root of smaller tree point to the root of the larger.
  - Implies $O(n \log n)$ time for performing $n$ union-find operations:
    - Each time we follow a pointer, we are going to a subtree of size at least double the size of the previous subtree.
    - Thus, we will follow at most $O(\log n)$ pointers for any find.
Union-Find Heuristic 2

**Path compression:**
- After performing a find, compress all the pointers on the path just traversed so that they all point to the root.

Implies a fast “almost linear” time for \( n \) union-find operations.

---

Ackermann Function

The version of the Ackermann function we use is based on an indexed function, \( A_i \), which is defined as follows, for integers \( x \geq 0 \) and \( i > 0 \):

\[
A_0(x) = x + 1 \\
A_{i+1}(x) = A_i^x(x)
\]

where \( A_i^x(x) \) denotes the \( k \)-fold composition of the function \( A_i \) with itself. That is,

\[
f^{(0)}(x) = x \\
f^{(k)}(x) = f(f^{(k-1)}(x)).
\]

So, in other words, \( A_{i+1}(x) \) involves making \( x \) applications of the \( A_i \) function on itself, starting with \( x \). This indexed function actually defines a progression of functions, with each function growing much faster than the previous one:

- \( A_0(x) = x + 1 \), which is the increment-by-one function
- \( A_1(x) = 2x \), which is the multiply-by-two function
- \( A_2(x) = x2^x \geq 2^x \), which is the power-of-two function
- \( A_3(x) \geq 2^{2^x} \) (with \( x \) number of 2's), which is the tower-of-twos function
- \( A_4(x) \) is greater than or equal to the tower-of-tower-of-twos function
- and so on.
### Ackermann Function

We then define the **Ackermann function** as

\[
A(x) = A_x(2),
\]

which is an incredibly fast-growing function.

- To get some perspective, note that \(A(3) = 2048\) and \(A(4)\) is greater than or equal to a tower of 2048 twos, which is much larger than the number of subatomic particles in the universe.

Likewise, its inverse, which is pronounced “alpha of n”,

\[
\alpha(n) = \min\{x : A(x) \geq n\},
\]

is an incredibly slow-growing function. Even though \(\alpha(n)\) is indeed growing as \(n\) goes to infinity, for all practical purposes, \(\alpha(n) \leq 4\).

---

### Fast Amortized Time Analysis

- For each node \(v\) in the union tree that is a root
  - define \(n(v)\) to be the size of the subtree rooted at \(v\) (including \(v\))
  - identified a set with the root of its associated tree.
- We update the size field of \(v\) each time a set is unioned into \(v\). Thus, if \(v\) is not a root, then \(n(v)\) is the largest the subtree rooted at \(v\) can be, which occurs just before we union \(v\) into some other node whose size is at least as large as \(v\)’s.
- For any node \(v\), then, define the **rank** of \(v\), which we denote as \(r(v)\), as \(r(v) = \lceil \log n(v) \rceil + 2\):
  - Thus, \(n(v) \geq 2^{r(v)-2}\).
  - Also, since there are at most \(n\) nodes in the tree of \(v\), \(r(v) \leq \lceil \log n \rceil + 2\), for each node \(v\).
Amortized Time Analysis (2)

For each node \( v \) with parent \( w \):
- \( r(v) < r(w) \)

**Proof:** We make \( v \) point to \( w \) only if the size of \( w \) before the union is at least as large as the size of \( v \). Let \( n(w) \) denote the size of \( w \) before the union and let \( n'(w) \) denote the size of \( w \) after the union. Thus, after the union we get

\[
\begin{align*}
r(v) &= \lfloor \log n(v) \rfloor + 2 \\
&< \lfloor \log n(v) + 1 \rfloor + 2 \\
&= \lfloor \log 2n(v) \rfloor + 2 \\
&\leq \lfloor \log (n(v) + n(w)) \rfloor + 2 \\
&= \lfloor \log n'(w) \rfloor + 2 \\
&\leq r(w).
\end{align*}
\]

Thus, ranks are strictly increasing as we follow parent pointers.

Amortized Time Analysis (3)

**Claim:** There are at most \( n/2^{s-2} \) nodes of rank \( s \).

**Proof:**
- Since \( r(v) < r(w) \), for any node \( v \) with parent \( w \), ranks are monotonically increasing as we follow parent pointers up any tree.
- Thus, if \( r(v) = r(w) \) for two nodes \( v \) and \( w \), then the nodes counted in \( n(v) \) must be separate and distinct from the nodes counted in \( n(w) \).
- If a node \( v \) is of rank \( s \), then \( n(v) \geq 2^{s-2} \).
- Therefore, since there are at most \( n \) nodes total, there can be at most \( n/2^{s-2} \) that are of rank \( s \).
Amortized Time Analysis (4)

For the sake of our amortized analysis, let us define a labeling function, \( L(v) \), for each node \( v \), which changes over the course of the execution of the operations in \( \sigma \). In particular, at each step \( t \) in the sequence \( \sigma \), define \( L(v) \) as follows:

\[
L(v) = \text{the largest } i \text{ for which } r(p(v)) \geq A_i(r(v)).
\]

Note that if \( v \) has a parent, then \( L(v) \) is well-defined and is at least 0, since

\[
r(p(v)) \geq r(v) + 1 = A_0(r(v)).
\]

because ranks are strictly increasing as we go up the tree \( U \). Also, for \( n \geq 5 \), the maximum value for \( L(v) \) is \( \alpha(n) - 1 \), since, if \( L(v) = t \), then

\[
n > \lfloor \log n \rfloor + 2 \\
\geq r(p(v)) \\
\geq A_i(r(v)) \\
\geq A_i(2).
\]

Or, put another way,

\[
L(v) < \alpha(n),
\]

for all \( v \) and \( t \).

© 2015 Goodrich and Tamassia

Amortized Time Analysis (5)

Let \( v \) be a node along a path, \( P \), in the union tree. Charge 1 cyber-dollar for following the parent pointer for \( v \) during a find:

- If \( v \) has an ancestor \( w \) in \( P \) such that \( L(v) = L(w) \), at this point in time, then we charge 1 cyber-dollar to \( v \) itself.
- If \( v \) has no such ancestor, then we charge 1 cyber-dollar to this find.

Since there are most \( \alpha(n) \) rank groups, this rule guarantees that any find operation is charged at most \( \alpha(n) \) cyber-dollars.
Amortized Time Analysis (6)

- After we charge a node \( v \) then \( v \) will get a new parent, which is a node higher up in \( v \)'s tree.
- The rank of \( v \)'s new parent will be greater than the rank of \( v \)'s old parent \( w \).
- Any node \( v \) can be charged at most \( r(v) \) cyber-dollars before \( v \) goes to a higher label group.
- Since \( L(v) \) can increase at most \( \alpha(n) - 1 \) times, this means that each vertex is charged at most \( r(n) \alpha(n) \) cyber-dollars.

Amortized Time Analysis (7)

- Combining this fact with the bound on the number of nodes of each rank, this means there are at most
  \[
  s \alpha(n) \frac{n}{2^{s-2}} = n \alpha(n) \frac{s}{2^{s-2}}
  \]
  cyber-dollars charged to all the vertices of rank \( s \).
- Summer over all possible ranks, the total number of cyber-dollars charged to all nodes is at most
  \[
  \sum_{s=0}^{\lfloor \log n \rfloor + 2} n \alpha(n) \frac{s}{2^{s-2}} \leq \sum_{s=0}^{\infty} n \alpha(n) \frac{s}{2^{s-2}}
  = n \alpha(n) \sum_{s=0}^{\infty} \frac{s}{2^{s-2}}
  \leq 8n \alpha(n).
  \]
  so the total time for \( m \) union-find operations, starting with \( n \) singleton sets is \( O((n+m)\alpha(n)) \).