



Disjoint Sets

CptS 223 – Advanced Data Structures

Larry Holder

School of Electrical Engineering and Computer Science
Washington State University



Disjoint Sets

- Data structure for problems requiring equivalence relations
 - I.e., Are two elements in the same equivalence class
- Applications
 - Reachability of components in a graph
- Disjoint sets provide a simple, fast solution
 - Simple: array-based implementation
 - Fast: $O(1)$ per operation average case
- Analysis is challenging



Equivalence Relation

- Relation R on set S maps pairs of elements of S to true or false
 - For all $a, b \in S$, $(a R b) \rightarrow \{\text{true}, \text{false}\}$
- Equivalence relation is a relation R such that the following hold
 - R is reflexive: $(a R a)$ for all $a \in S$
 - R is symmetric: $(a R b) \Leftrightarrow (b R a)$
 - R is transitive: $(a R b)$ and $(b R c) \rightarrow (a R c)$
- Example: Equality over integers



Equivalence Class

- Given set S and equivalence relation R
- Find the subsets S_i of S such that
 - For all $a, b \in S_i$: $(a R b)$
 - For all $a \in S_i, b \in S_j, i \neq j$: not $(a R b)$
- These S_i are the equivalence classes of S for relation R
 - The S_i are “disjoint sets”
- Example: $S = \{1, 2, 3, 4, 3, 3, 2, 1, 3\}$, R is =



Disjoint Sets

- Main operation
 - Determine if a and b are in the same equivalence class
- Approach
 - Put each element of S in a disjoint set of its own
 - If a and b are related, then union the sets containing a and b



Disjoint Sets

- Example

- $S = \{1_a, 2_a, 3_a, 4_a, 3_b, 3_c, 2_b, 1_b, 3_d\}$
- $DS = \{ \{1_a\}, \{2_a\}, \{3_a\}, \{4_a\}, \{3_b\}, \{3_c\}, \{2_b\}, \{1_b\}, \{3_d\} \}$
- $3_a R 3_b ?$, $3_c R 3_d ?$
- $DS = \{ \{1_a\}, \{2_a\}, \{3_a, 3_b\}, \{4_a\}, \{3_c, 3_d\}, \{2_b\}, \{1_b\} \}$
- $3_a R 3_c ?$
- $DS = \{ \{1_a\}, \{2_a\}, \{3_a, 3_b, 3_c, 3_d\}, \{4_a\}, \{2_b\}, \{1_b\} \}$



Disjoint Sets

- Operations
 - Find(a)
 - Returns a representative of the equivalence class containing a
 - Union(S_i, S_j)
 - Creates a new set $S_k = S_i \cup S_j$
 - Associates single representative to all elements of S_k
- Assume each element can be associated with a unique integer 0 to N-1



Disjoint Sets

- Solution #1
 - Maintain an array of size N containing the representative of each element
 - Find is a $O(1)$ lookup
 - Union(a, b)
 - Assuming a in class i and b in class j
 - Scan array, changing all i 's to j 's
 - $O(N)$ per union (how many unions?)
 - Okay if $\Omega(N^2)$ find operations
 - $O(1)$ per union/find operation



Disjoint Sets

- Solution #2a
 - Maintain a linked list for each equivalence class
 - Increases time to find an element
 - Decreases time for unions by not having to search all N elements
 - Just the two lists where the elements are found
 - And then concatenate lists: $O(\text{size of larger list})$
 - Still, $\Theta(N^2)$ performance in worst case



Disjoint Sets

- Solution #2b
 - Maintain a linked list for each equivalence class
 - Also maintain size of each class (list)
 - Union always concatenates the smaller to the larger class (list)
 - Thus, $N-1$ unions cost $O(N \log N)$ (why?)
 - Any sequence of M finds and $N-1$ unions takes time $O(M + N \log N)$



Disjoint Sets

- Performance
 - Can ensure $O(1)$ worst-case time for find operation
 - Or, can ensure $O(1)$ worst-case time for union operation
 - But not both
- Solution #3
 - Fast unions, slow finds
 - But, achieves $O(M+N)$ time for any sequence of M finds and $N-1$ unions



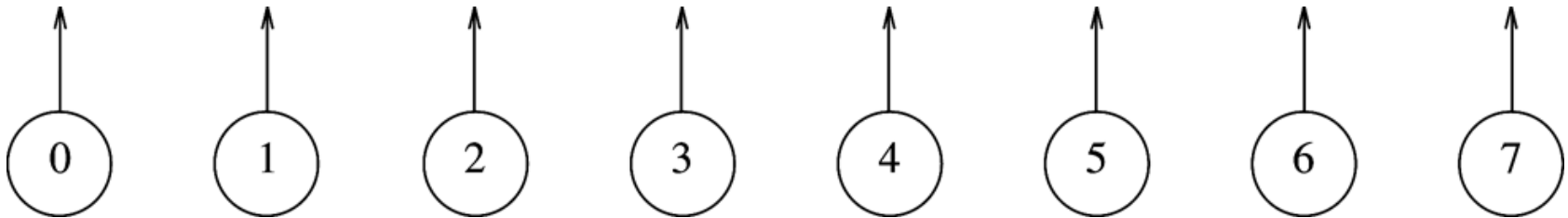
Disjoint Sets

- Solution 3
 - Represent each set as a tree
 - Tree's root is the representative element for the set
 - Disjoint sets are a forest of trees
 - Find(a) returns root element of tree containing a
 - Union(a,b) points root node of tree containing b to root node of tree containing a
 - Implemented as array s , where $s[i]$ = index of parent node in tree (or -1 if root)

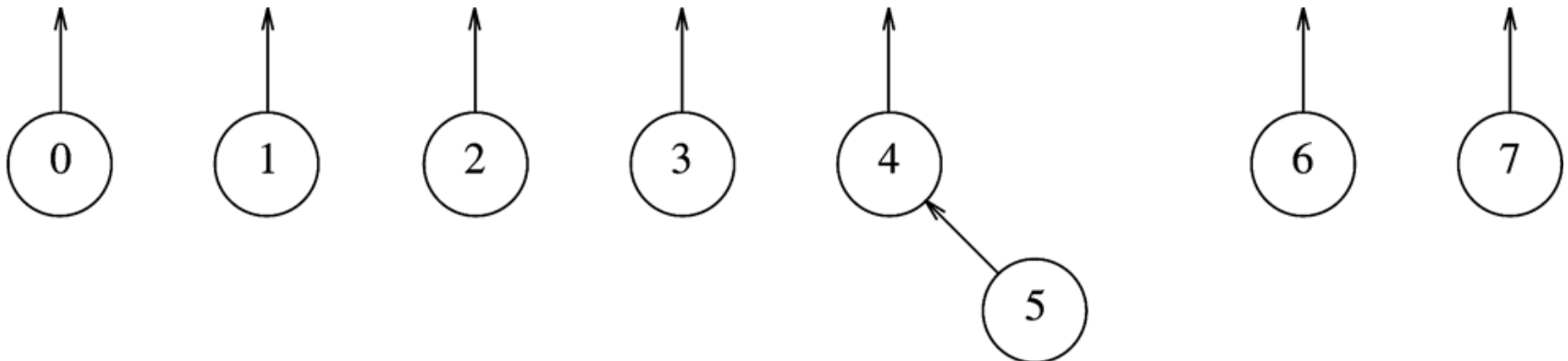


Example

Initial disjoint sets of 8 elements (really an array of size 8 of all -1s):

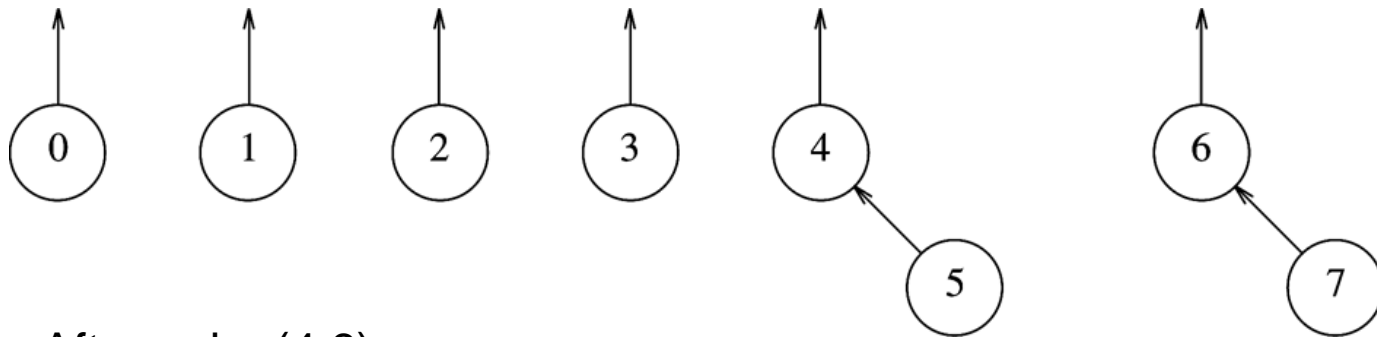


After union(4,5):

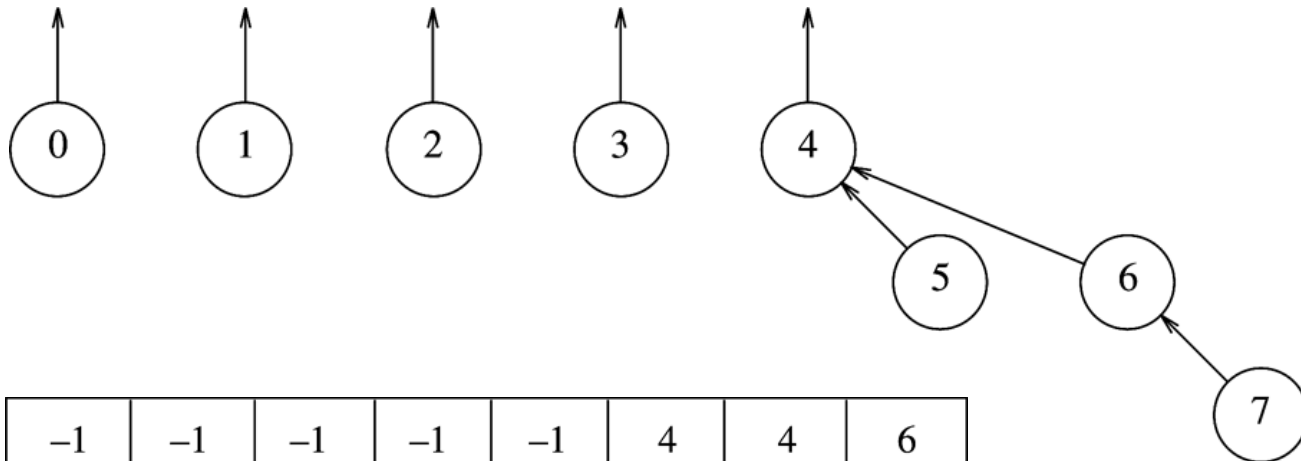


Example (cont.)

After union(6,7):



After union(4,6):



-1	-1	-1	-1	-1	4	4	6
0	1	2	3	4	5	6	7



Implementation

```
1  class DisjSets
2  {
3      public:
4          explicit DisjSets( int numElements );
5
6          int find( int x ) const;
7          int find( int x );
8          void unionSets( int root1, int root2 );
9
10     private:
11         vector<int> s;
12 };
```



Implementation

```
1  /**
2   * Construct the disjoint sets object.
3   * numElements is the initial number of disjoint sets.
4   */
5  DisjSets::DisjSets( int numElements ) : s( numElements )
6  {
7      for( int i = 0; i < s.size( ); i++ )
8          s[ i ] = -1;
9  }
```



Implementation

```
1  /**
2   * Union two disjoint sets.
3   * For simplicity, we assume root1 and root2 are distinct
4   * and represent set names.
5   * root1 is the root of set 1.
6   * root2 is the root of set 2.
7   */
8  void DisjSets::unionSets( int root1, int root2 )
9  {
10     s[ root2 ] = root1;
11 }
```



Implementation

```
1  /**
2   * Perform a find.
3   * Error checks omitted again for simplicity.
4   * Return the set containing x.
5   */
6  int DisjSets::find( int x ) const
7  {
8      if( s[ x ] < 0 )
9          return x;
10     else
11         return find( s[ x ] );
12 }
```



Analysis

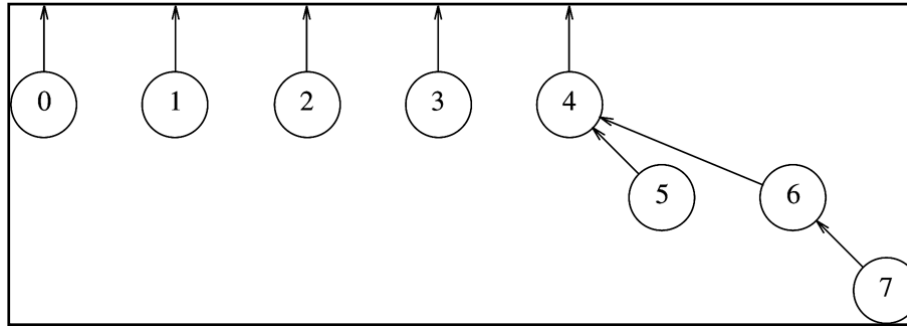
- Find(x)
 - Proportional to depth of tree containing x
 - Deepest tree?
 - Worst-case running time $O(N)$
 - M consecutive find operations $O(MN)$ worst case
- Average case analysis
 - What is the average case?
 - Unions can still cost $O(N^2)$
 - But we can do better...



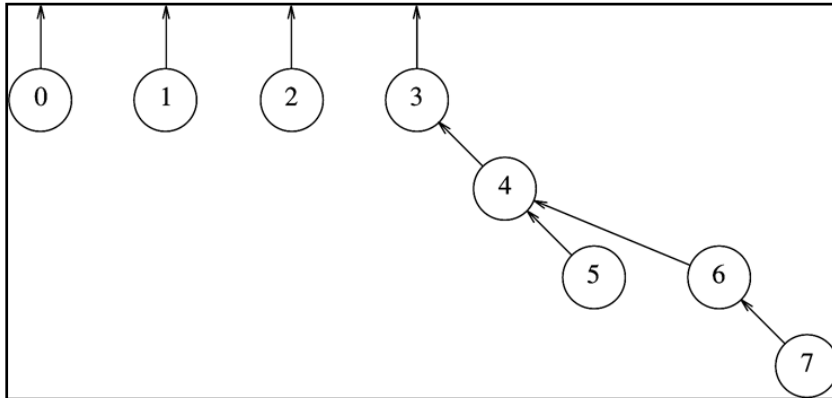
Smart Union

- Union by size
 - Link smaller tree to larger tree
- Maximum node depth is $(\log_2 N)$ (why?)
- Find(x) running time?
- Sequence of M operations requires $O(M)$ time
 - Random unions tend to merge large sets with small sets
 - Thus, only increase depth of smaller set
- Implementation
 - Use (- size) instead of -1 for root entries

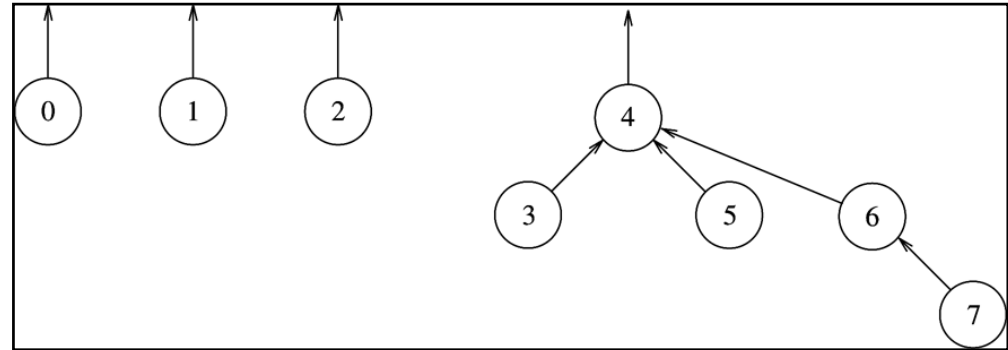
Smart Union Example



Union(3,4)



Smart-Union(3,4)



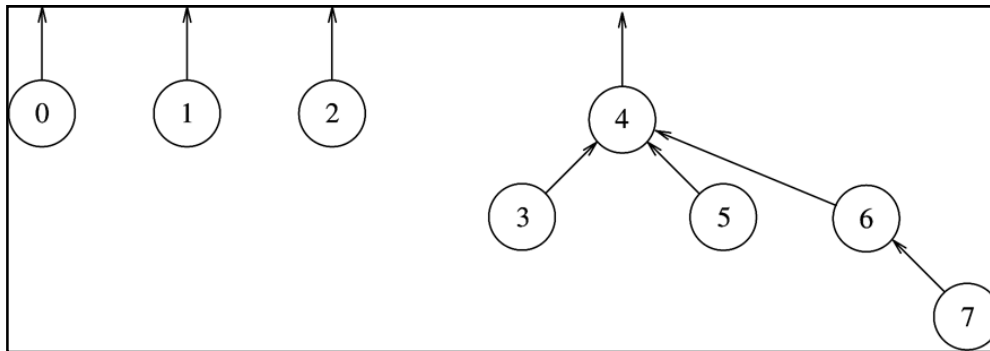
-1	-1	-1	4	-5	4	4	6
0	1	2	3	4	5	6	7



Smart Union by Height

- Keep track of height of each tree, rather than size
- Union: Link smaller-height tree to larger-height tree
- Height only increases when two equal-height trees joined
- Still $O(\log N)$ maximum depth
- Still $O(M)$ time for M operations
- Implementation
 - Store (negative of height) minus 1

Smart Union by Height Example



-1	-1	-1	4	-3	4	4	6
0	1	2	3	4	5	6	7

Smart Union by Height Implementation

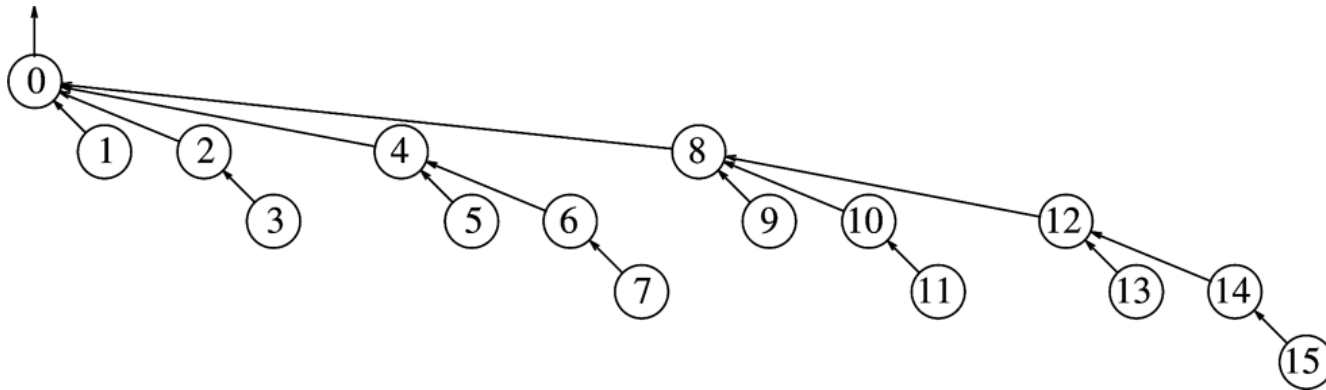
```
1  /**
2   * Union two disjoint sets.
3   * For simplicity, we assume root1 and root2 are distinct
4   * and represent set names.
5   * root1 is the root of set 1.
6   * root2 is the root of set 2.
7   */
8  void DisjSets::unionSets( int root1, int root2 )
9  {
10     if( s[ root2 ] < s[ root1 ] ) // root2 is deeper
11         s[ root1 ] = root2;      // Make root2 new root
12     else
13     {
14         if( s[ root1 ] == s[ root2 ] )
15             s[ root1 ]--;        // Update height if same
16         s[ root2 ] = root1;      // Make root1 new root
17     }
18 }
```



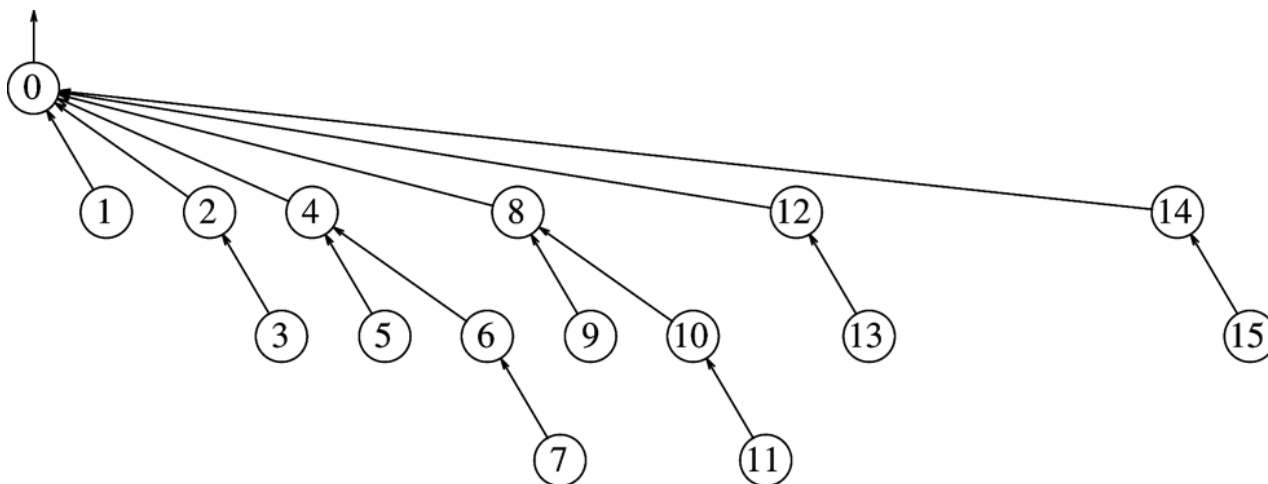
Path Compression

- Smart union achieves $O(M)$ time for M operations (average case)
- But still $O(M \log N)$ in the worst case
- Path compression
 - All nodes accessed during a $\text{Find}(x)$ are linked directly to the root
- Path compression without smart union still $O(M \log N)$ worst case

Path Compression Example



After Find(14):



Path Compression Implementation

```
1  /**
2   * Perform a find with path compression.
3   * Error checks omitted again for simplicity.
4   * Return the set containing x.
5   */
6  int DisjSets::find( int x )
7  {
8      if( s[ x ] < 0 )
9          return x;
10     else
11         return s[ x ] = find( s[ x ] );
12 }
```



Path Compression with Smart Union

- Path compression works as is with union-by-size (tree sizes don't change)
- Path compression with union-by-height requires re-computation of heights
- Solution: Don't recompute heights
 - Heights become (possibly over) estimates of true height
 - Also called "ranks" and this solution is called "union-by-rank"
 - Ranks are modified far less than sizes, so slightly faster in practice
- Path compression does not change average case time, but does reduce worst-case time

Analysis of Union-by-Rank and Path Compression

- Worst case is $\Theta(M\alpha(M,N))$
 - M is number of operations (find, union)
 - N is number of elements in disjoint set
 - $\alpha(M,N)$ is the inverse of Ackermann's function
- In practice, $\alpha(M,N) \leq 4$
- Thus, worst case is $\Theta(M)$ for M operations



Ackermann's Function

$$A(1, j) = 2^j \text{ for } j \geq 1$$

$$A(i, 1) = A(i - 1, 2) \text{ for } i \geq 2$$

$$A(i, j) = A(i - 1, A(i, j - 1)) \text{ for } i, j \geq 2$$

A(i,j)	j=1	j=2	j=3	j=4
i=1	$2^1 = 2$	$2^2 = 4$	$2^3 = 8$	$2^4 = 16$
i=2	$2^2 = 4$	$2^{2^2} = 16$	$2^{16} = 65536$	2^{65536}
i=3	$2^{2^2} = 16$	$2^{16} = 65536$	2^{65536}	$2^{2^{65536}} = \text{BIG}$

Inverse of Ackermann's Function

$$\alpha(M, N) = \min\{i \geq 1 \mid A(i, \lfloor M / N \rfloor) > \log N\}$$

$$\alpha(M, N) = O(\log_2^* N)$$

$\log_2^* N = \log_2 \log_2 \log_2 \cdots \log_2 N$ such that result ≤ 1

$$\log_2^* 65536 = 4$$

$$\log_2^* 2^{65536} = 5 \text{ (note that } 2^{65536} \text{ is a 20,000 digit number)}$$

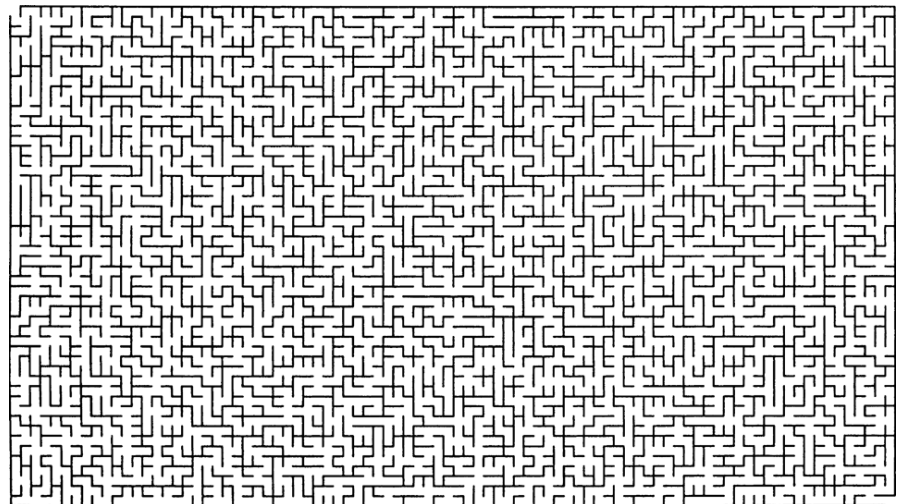
Analysis of Union-by-Rank and Path Compression



- Worst case is $\Theta(M\alpha(M,N))$ for M operations on disjoint set with N elements
 - But, technically not linear in M
- Any sequence of $M = \Omega(N)$ union/find operations takes $O(M \log^*N)$ time

Application: Maze Generation

- Start with walls everywhere
- Randomly choose a wall that separates two disconnected cells
- Continue until start and finish cells connected
- Or, continue until all cells connected
 - More dead ends





Maze Generation Example

Initial state: All walls up, all cells in their own set.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

{0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21}
{22} {23} {24}



Maze Generation Example

Intermediate state:

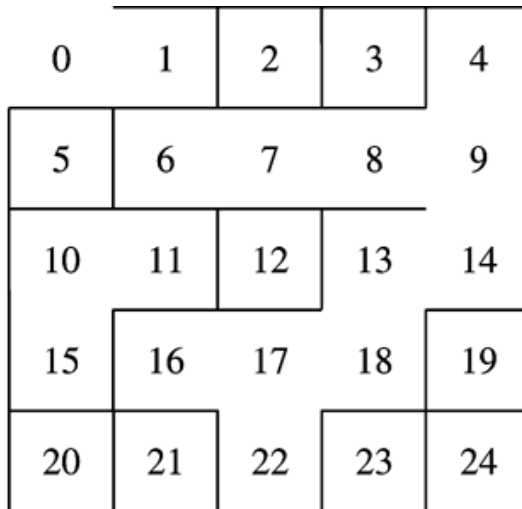
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14} {5} {10, 11, 15} {12} {16, 17, 18, 22} {19} {20} {21} {23} {24}



Maze Generation Example

After joining 13 and 18 from previous intermediate state:

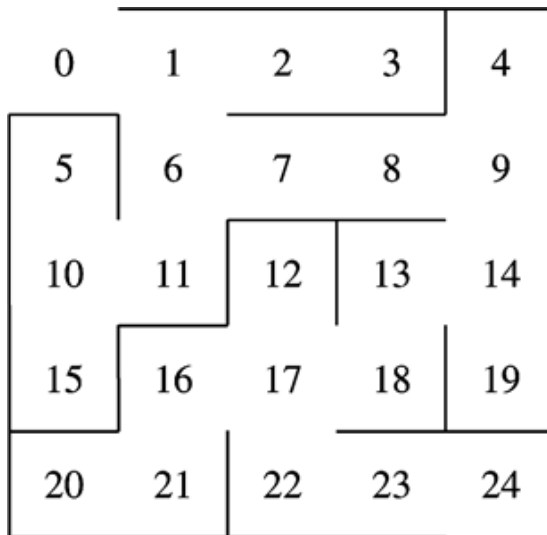


{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22} {5} {10, 11, 15} {12} {19} {20} {21} {23} {24}



Maze Generation Example

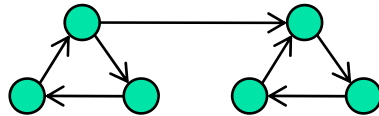
Final state: All cells connected.



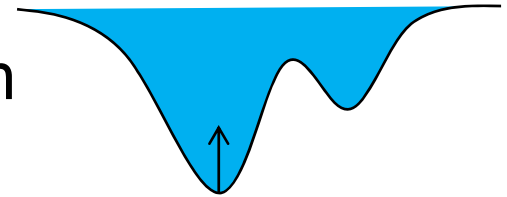
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}

More Applications

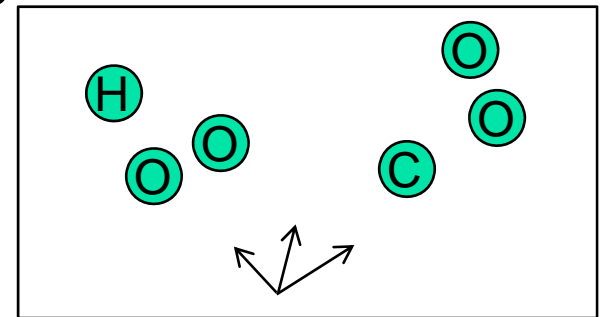
- Finding the connected components of an undirected graph



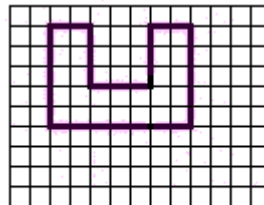
- Computing shorelines of a terrain



- Molecular identification from fragmentation



- Image processing
 - Movie coloring



Coloring movie



Summary

- Disjoint sets data structure provides simple, fast solution to equivalence problems
 - Array-based implementation
 - Average case $O(1)$ time per operation
- Despite simplicity, analysis is challenging
- Numerous applications