Chapter 8
The Disjoint Sets Class

Introduction

- equivalence problem
  - can be solved fairly simply
    - simple data structure
    - each function requires only a few lines of code
    - two operations: union and find
    - can be implemented with simple array
  - outline
    - equivalence relations and the dynamic equivalence problem
    - data structure and smart union algorithms
    - path compression
    - analysis
    - application

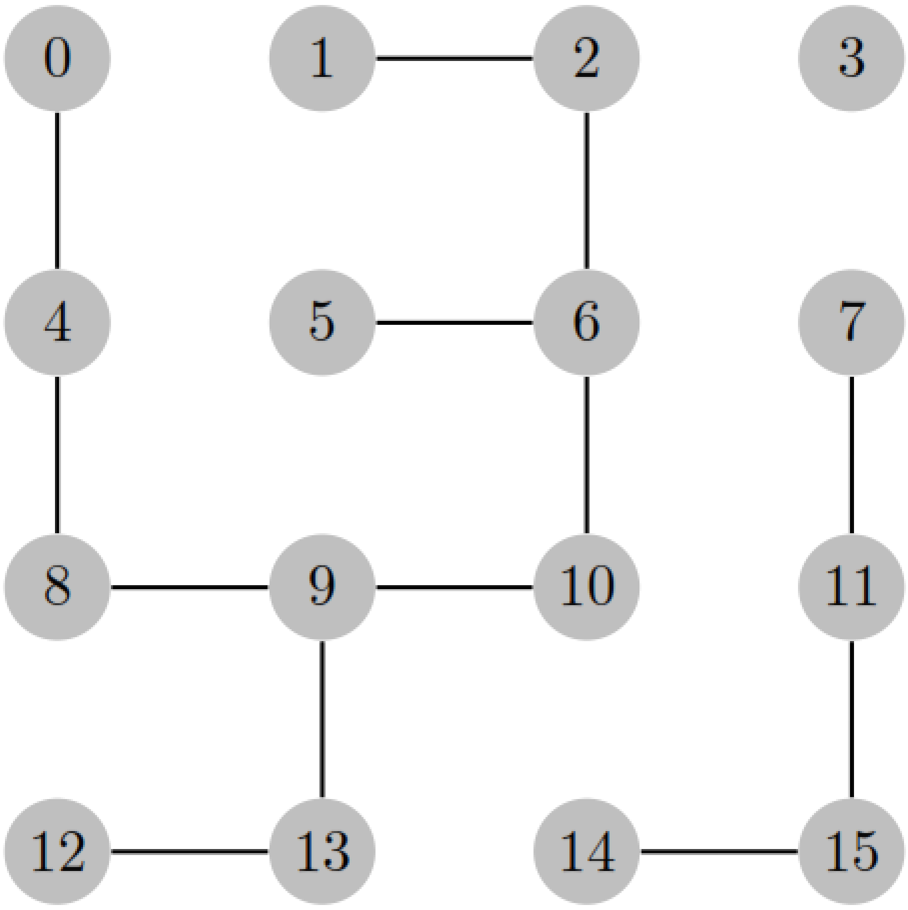
Equivalence Relations

- a relation R on a set S is a subset of $S \times S$
  - i.e., the set of ordered pairs (p, q) with $p, q \in S$
  - p is related to q, denoted pRq, if $(p, q) \in R$

- an equivalence relation is a relation R with these properties:
  - Reflexive: pRp or p is related to p
  - Symmetric: if pRq, then qRp
  - Transitive: if pRq, and qRr, then pRr
  - given an equivalence relation R, the equivalence class of p is $\{q \mid pRq\}$ (the set of q related to p)

Example

- two nodes are <u>equivalent</u> if they are connected by a path



Dynamic Equivalence Problem

- an equivalence relation on a set partitions the set into disjoint equivalence classes
- $p \sim q$ if p and q are in the same equivalence class
  - the difficulty is that the equivalence classes are probably defined indirectly
- in the preceding example, two nodes are in the same equivalence class if and only if they are connected by a path
  - however, the entire graph was specified by a small number of pairwise connections:

0~4, 4~8, 8~9, 1~2, 2~6, 9~13, 11~15, 14~15,
12~13, 7~11, 5~6, 6~10

  - how can we decide if 0 ~ 1?

Dynamic Equivalence Problem

- in the general version of the dynamic equivalence problem, we begin with a collection of disjoint sets $S_1, \ldots, S_N$, each with a single distinct element

- two operations exist on these sets:
  - find(p), which returns the id of the equivalence class containing p
  - union(p,q), which merges the equivalence classes of p and q, with the root of p being the new parent of the root of q

- in the case of building up the connected components of the graph example, given a connection $p \sim q$ we would call union(p,q) which in turn would need to call find(p) and find(q)

- these operations are dynamic:
  - the sets may change because of the union operation, and
  - find must return an answer before the entire equivalence classes have been constructed

Union-Find

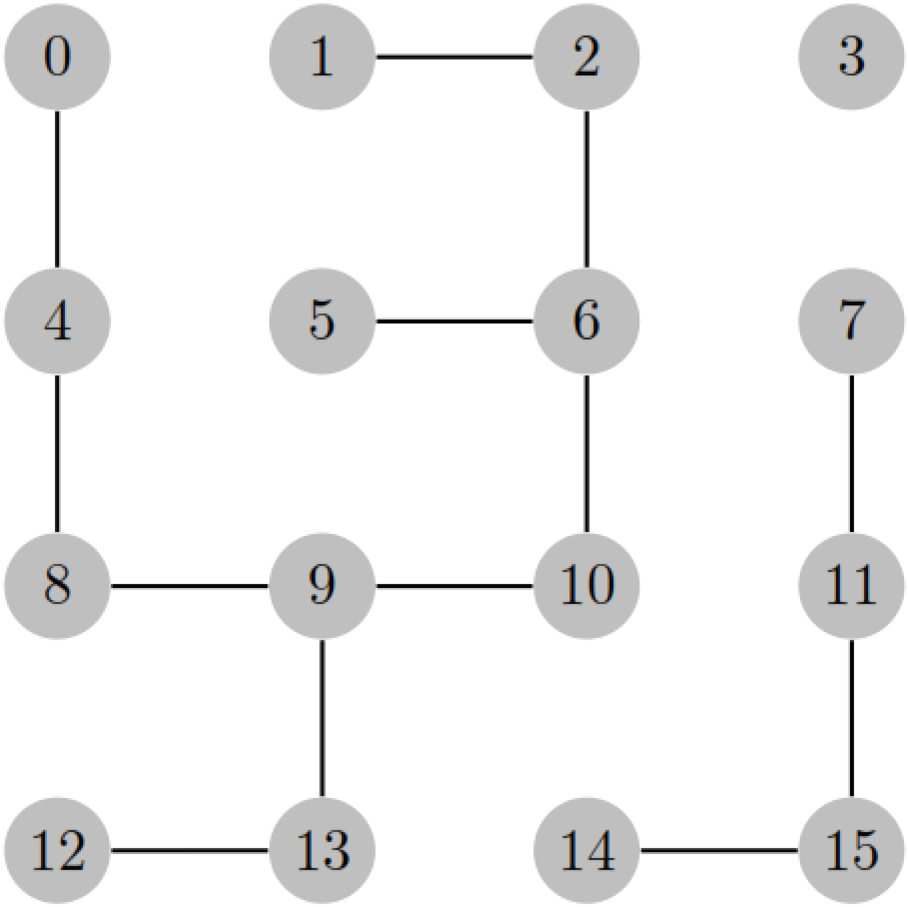
- in a computer network, we know that certain pairs of computers are connected
  - how do we use that information to determine whether we can get traffic from one arbitrary computer to another?
  - in a social network, we know that certain people are friends; how do we use that information to determine whether we are a friend of a friend of a friend?

Union-Find

- denote the items by $0, 1, 2, \ldots, N-1$
- given pairs of items $(p, q), 0 \leq p, q \leq N-1$, which is interpreted as meaning $p \sim q$
- in keeping with the graph example, we will refer to the items as vertices and say that p and q are connected if $p \sim q$
- we will also refer to the equivalence classes as connected components, or just components

Union-Find: Graph Abstraction

- previous example



Union-Find

- we need a data structure that will represent known connections and allow us to answer the following:
  - given arbitrary vertices p and q, can we tell if they are connected?
  - can we determine the number of components?

- Union-find API:

UF(N)	initialize N vertices with 0 to N-1
union(p, q)	add connection between p and q
find(p)	return the component id (0 to N-1) for p
connected(p, q)	true if p and q are in the same component
num_components()	return the number of components

Union-Find

- basic data structure
  - we will use a vertex-indexed array id[ ] to represent the components
  - the value id[p] is the <u>component</u> that p belongs to
  - initially, we do not know that any vertices are connected, so we initialize id[p] = p for all p (i.e., each vertex is initially in its own component)

Union-Find

- invariants
  - in the analysis of algorithms, an invariant is a condition that is guaranteed to be true at specified points in the algorithm
  - we can use invariants and their preservation by an algorithm to prove that the algorithm is correct

Quick-Find

- quick-find maintains the invariant that p and q are connected if

  id[p] = id[q]

- this is called quick-find because the function find() is trivial:

```
function find(p)
  return id[p]
end
```

- there is just a single array reference, so a call to find() is a constant time operation

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
id[i]	0	1	9	9	9	6	6	7	8	9

Quick-Find

```
function union(p,q) {
        p_id = find(p)
        q_id = find(q)

        // if p and q are already in the same component, we're done!
        if (q_id == p_id) return

        // otherwise, re-label q's components as being in p's component
        for i = 0 to N-1 {
                if (id[i] == q_id)   id[i] = p_id
        }
}
```

– worst-case, the number of operations is $\propto N$

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
id[i]	0	1	6	6	6	6	6	7	8	6

union(6,3)

Quick-Find

- it should be clear that quick-find union() preserves the invariant
- if there is only a single component, then we will need at least N-1 calls to union()
- in this situation each call to union() requires work $\propto N$
- this means that in this case, the work is at least $\propto N(N-1) \sim N^2$
- quick-find can be a quadratic-time algorithm!

Quick-Union

- quick-union avoids the quadratic behavior of quick-find
- in quick-union, given a vertex p, the value id[p] is the name of another vertex that is in the same component
  - we call such a connection a link
- to determine which component p lies in, we start at p
  - follow the link from p to id[p]
  - follow the link from there to (id[id[p]]), and so on, until we come to a vertex that has a link to itself
  - we call such a vertex a root
- we use the roots as the identifiers of the components
- recall that initially, id[p] = p, so all vertices start off as roots

Quick-Union: find()

```
function find(p) {
            // follow the links to a root
            if (p != id[p]) {
                  return find(id[p])
            }
            else {
                 // return the root as the component identifier
                 return p
            }
}
```

- the operation of find() will ensure that we eventually arrive at a root

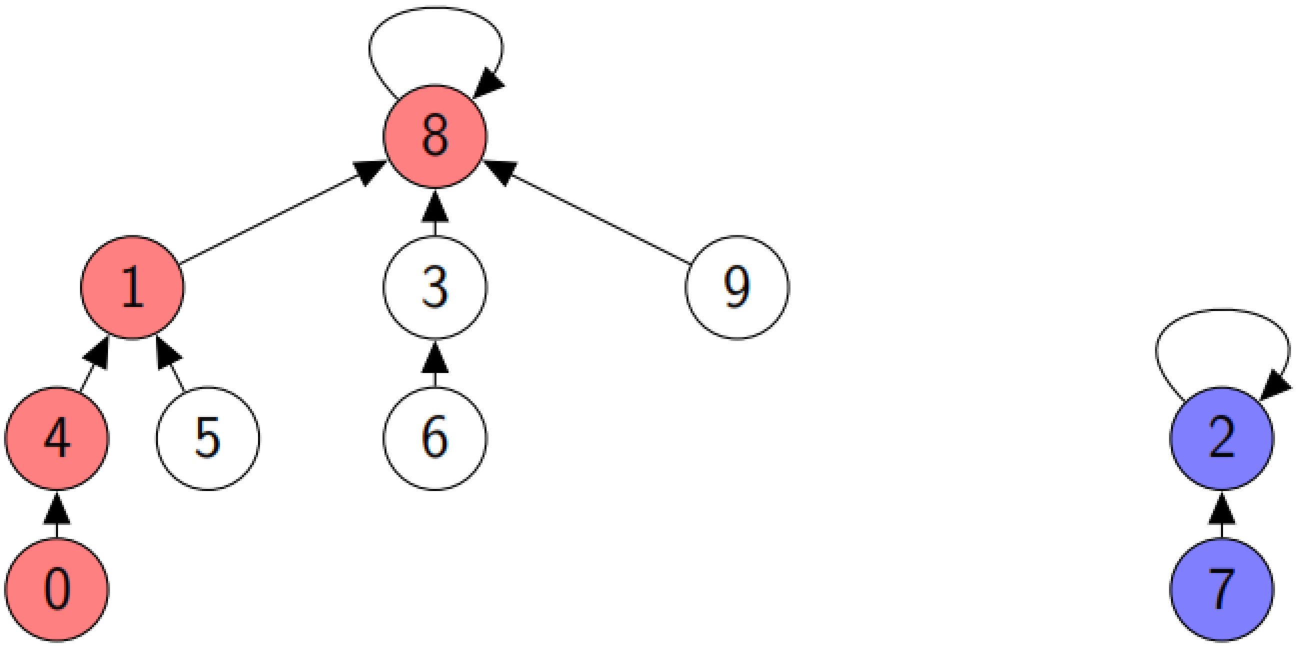
Quick-Union: find()

p	q	0	1	2	3	4	5	6	7	8	9
7	0	4	8	2	8	1	1	3	2	8	8

Read: the root of 7 will become the parent of the root of 0

find(7) = id[7] = id[2] or id[id[7]]

find(0) = id[0] = id[4] = id[1] = id[8] or id[id[id[id[0]]]]



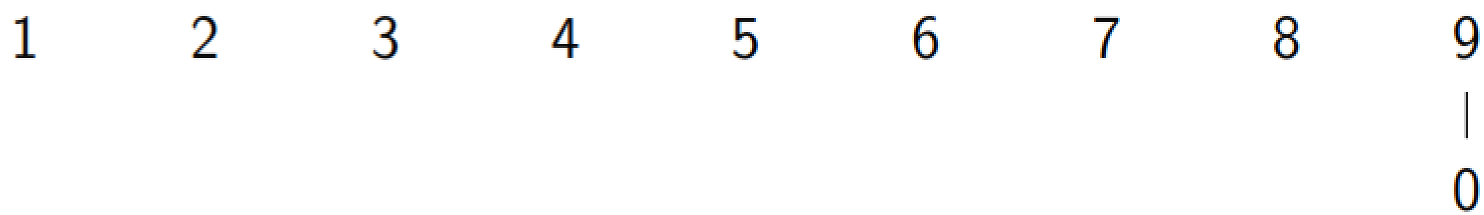
Quick-Union: union()

```
function union(p, q) {
        i = find(p)
        j = find(q)

        if (i == j) return;
        id [j] = i
end
```

Quick-Union

Example

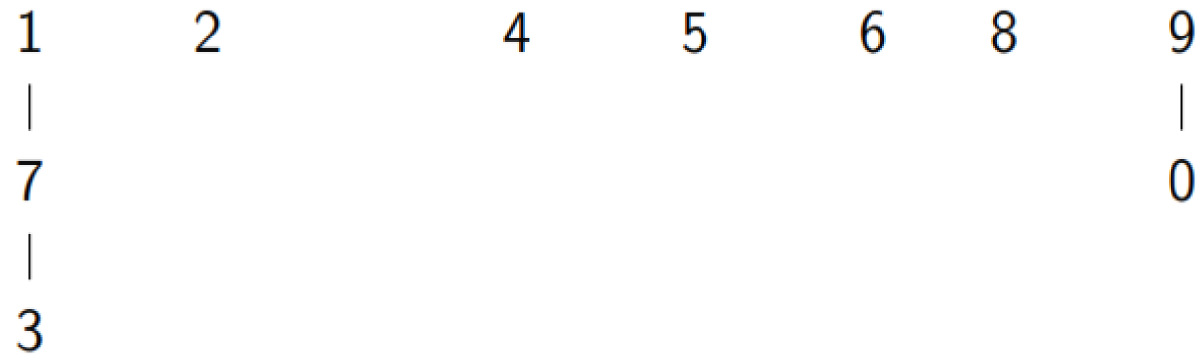
p	q	0	1	2	3	4	5	6	7	8	9
9	0	9	1	2	3	4	5	6	7	8	9



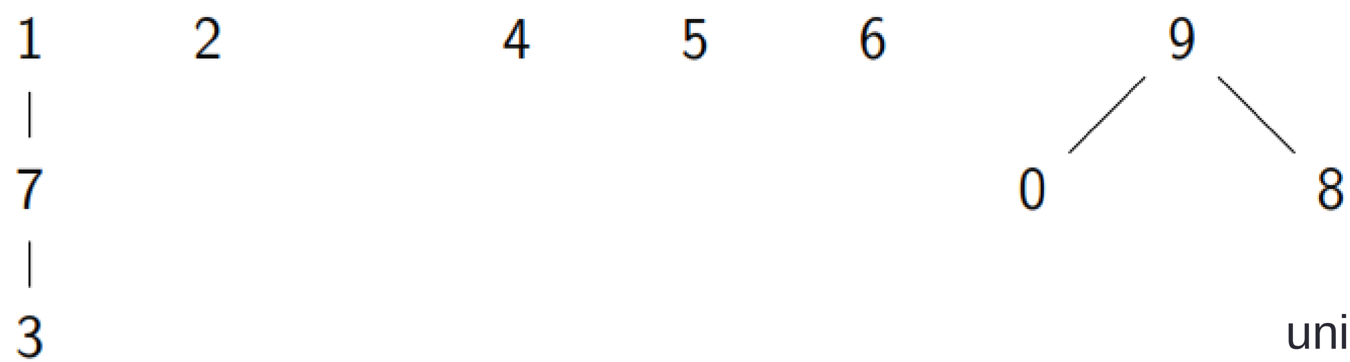
p	q	0	1	2	3	4	5	6	7	8	9
7	3	9	1	2	7	4	5	6	7	8	9

Quick-Union

p	q	0	1	2	3	4	5	6	7	8	9
1	7	9	1	2	7	4	5	6	1	8	9



p	q	0	1	2	3	4	5	6	7	8	9
9	8	9	1	2	7	4	5	6	1	9	9



union(3,8)?

Quick-Union: Complexity

the main computational cost of quick-union is the cost of find():

```
function find(p) {
        // follow the links to a root
        if (p != id[p]) {
            return find(id[p])
        }
        else {
            // return the root as the component identifier
            return p
        }
}
```

– the cost of a call to find() depends on how many links we must follow to find a root, which, in turn, depends on union()

Quick-Union: Complexity

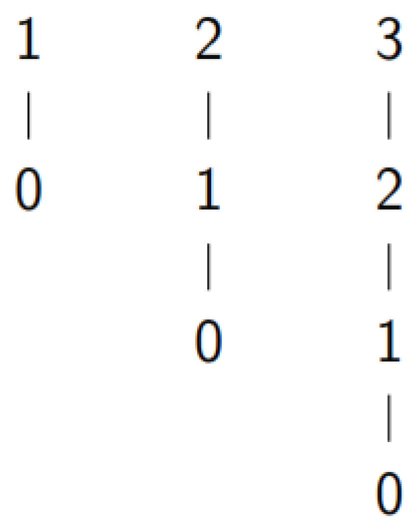
- the number of accesses of id[] used by the call find(p) in quick-union is ∝ to the <u>depth</u> of p in its tree
- the number of accesses used by union() and connected() is ∝ the cost of find()

- so, how tall can the trees be in the <u>worst</u> case?

Quick-Union: Worst-Case Complexity

- suppose there is only a single component, and the connections are specified as follows:

  (1,0), (2,1), . . . , (N-1,N-2)



- in the worst case, the height is $\propto N$, so applying union() to all N nodes is quadratic!

Weighted Quick-Union: union-by-size

- weighted quick-union is more clever: in union(), it connects the smaller tree to the larger to avoid growth in the height of the trees
- the depth of any node in a forest built by weighted quick-union for N vertices is at most lg N.

Weighted Quick-Union: union-by-size

- proof: we will prove that the height of every tree with k nodes in the forest is at most $\lg k$
  - if $k = 1$, such a tree has height 0.
- now assume that the height of a tree of size i is at most $\lg i$ for all $i < k$
- when we combine a tree of size i with a tree of size j, with $i \leq j$, and $i + j = k$, we increase the depth of each node in the smaller tree by 1
- however, they are now in a tree of size $i + j = k$, and

$$1 + \lg i = \lg 2 + \lg i = \lg(2 * i) \leq \lg(i + j) = \lg k$$

as threatened

Path Compression

- ideally, we would like every node in a tree to link to its root, so find() would be $O(1)$ time
- we can almost achieve this using path compression – we set the entries in id[] that we visit along the way to finding the root to point directly to the root

find() with Path Compression

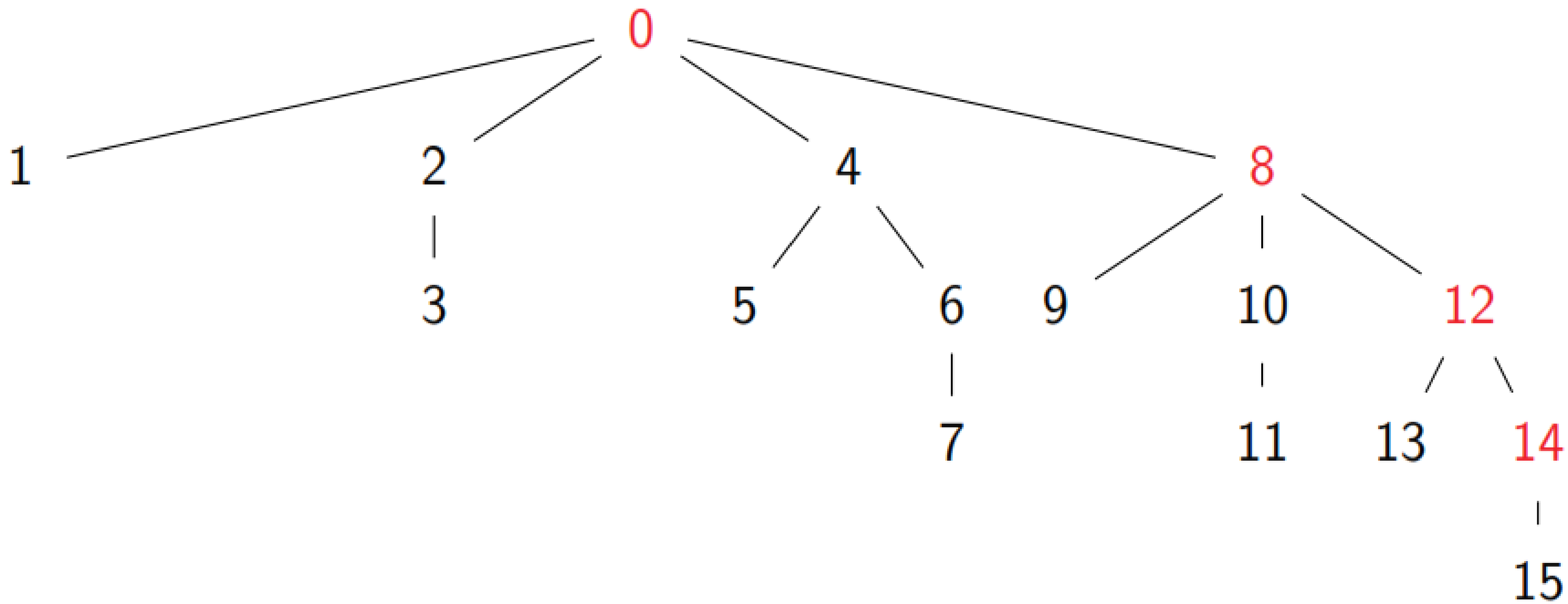
```
function find(p) {
        if (p == id[p])  return p;     // stop at the root…

        // otherwise link visited nodes to the root
        id[p] = find (id(p))
        return id[p]
}
```

the call find(14) visits 14, 12, 8, and 0 (on next slide):

```
        find(14): return find(id[14]) = find(12)
                find(12): return find(id[12]) = find(8)
                        find(8): return find(id[8]) = find(0)
                                find(0): return find(id[0]) = 0
                        find(8): id[8] = 0
                find(12): id[12] = 0
        find(14): id[14] = 0
```

Path Compression

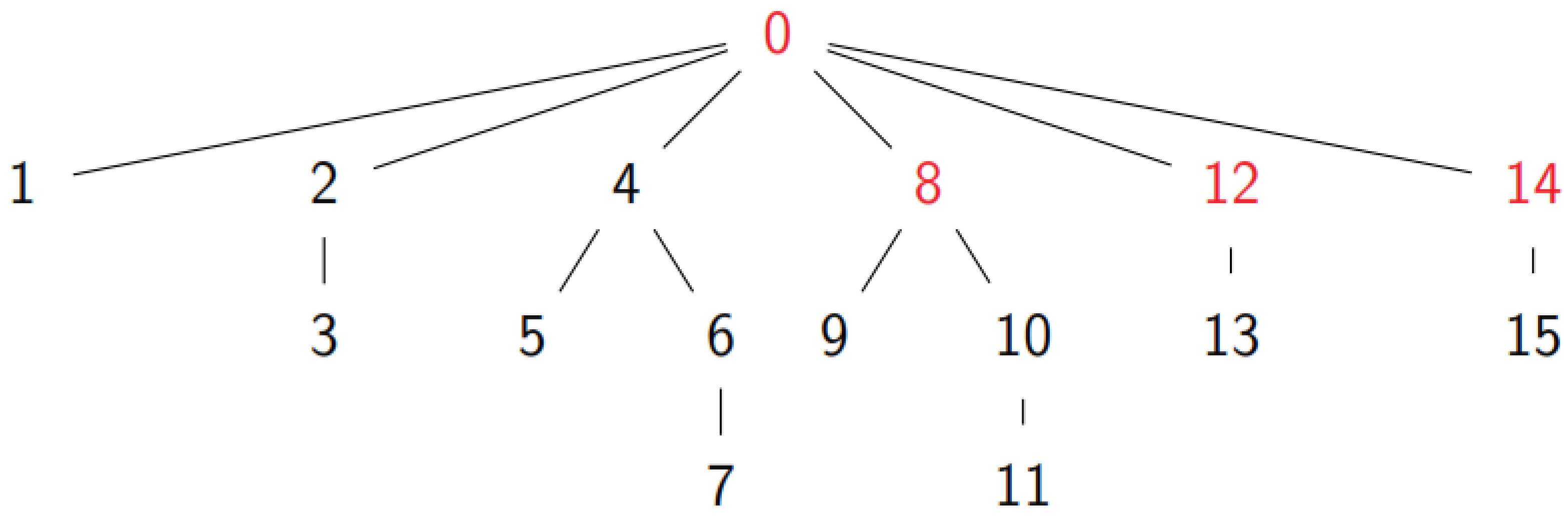
- example



- red components visited by find(14)

Path Compression

– example: effect of path compression



– the call find(14) links every element on the path from 14 to 0 directly to 0

Path Compression

- complexity
  - by itself, weighted quick-union (union by size) yields trees with worst-case height $\lg N$
  - by itself, quick-union with path compression yields trees with worst-case height $\lg N$
  - if used together, union by size + path compression does <u>better</u>: the worst-case complexity of a sequence of M calls to find() (where $M \geq N$) is almost, but not quite $\Theta(M)$
    - proved by Robert Tarjan in 1975
  - more exactly, it is $\Theta(M\ \alpha(N))$, where $\alpha(N)$ is a *very* <u>slow</u> growing function of a type known as an inverse Ackerman function

Inverse Ackerman Function

- our α is one version of the inverse Ackerman function:

$$\alpha(N) = \min\left\{ i \geq 1 \mid \overbrace{\lg\lg\lg\cdots\lg}^{i \text{ times}} N \leq 1 \right\}$$

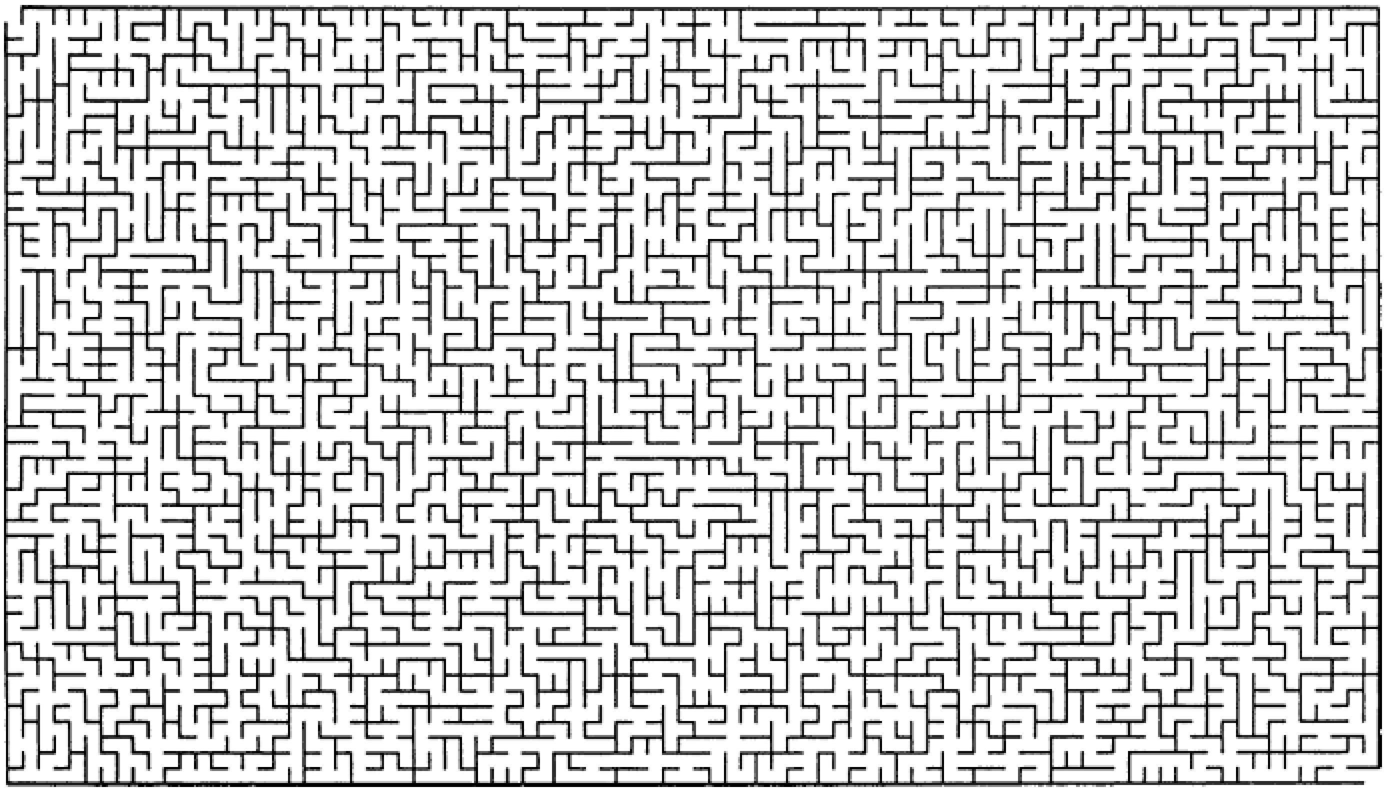
- the iterated logarithm:

$$\begin{array}{ll} \lg 2 = 1 & \alpha(2) = 1 \\ \lg\lg 4 = 1 & \alpha(4) = 2 \\ \lg\lg\lg 16 = 1 & \alpha(16) = 3 \\ \lg\lg\lg\lg 65536 = 1 & \alpha(65536) = 4 \\ \lg\lg\lg\lg\lg 2^{65536} = 1 & \alpha(2^{65536}) = 5 \end{array}$$

- this is a *very* slowly growing function of N!
- for any practical value of N, $\alpha(N) \leq 5$
- termed lg*, lg**, etc.

Union-Find Application

- generation of mazes



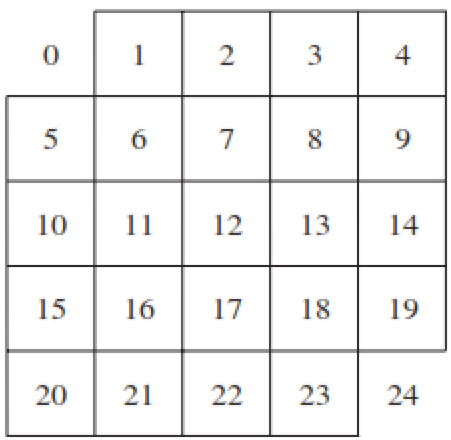
- can view as 80x50 set of cells where top right is connected to bottom left, and cells are separated from neighbors by walls

Union-Find Application

- algorithm
  - start with walls everywhere except <u>entrance</u> and <u>exit</u>
  - choose wall randomly
    - knock it down if cells not already connected
    - repeat until start and end cell <u>connected</u>
      - actually better to continue to knock down walls until every cell is reachable from every other cell (false leads)

Union-Find Application

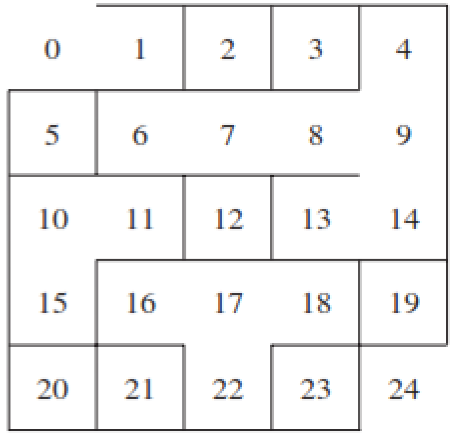
- example
  - 5x5 maze
  - use union-find data structure to show connected cells
    - initially, walls are <u>everywhere</u>, so each cell is its own equivalence class



{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}

Union-Find Application

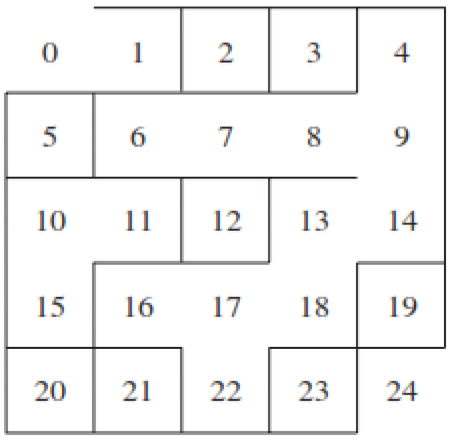
- example (cont.)
  - later stage in algorithm, after some walls have been deleted
  - randomly pick cells 8 and 13
    - no wall removed since they are already connected



{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}

Union-Find Application

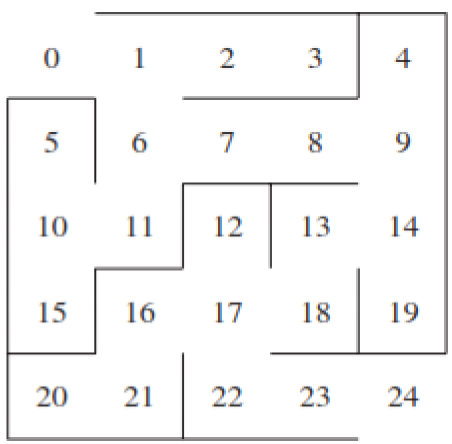
- example (cont.)
  - randomly pick cells 18 and 13
    - two calls to <u>find</u> show they are not connected
    - knock down wall
    - sets containing 18 and 13 combined with <u>union</u>



{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}

Union-Find Application

- example (cont.)
  - eventually, all cells are connected and we are done
    - could have stopped earlier once 0 and 24 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}

Union-Find Application

- analysis
  - running time dominated by union-find costs
  - size N is number of cells
  - number of finds $\propto$ number of cells
    - number of removed walls is one less than number of cells
    - only twice as many walls as cells
  - for N cells, there are two finds per randomly targeted wall, or between $2N$ and $4N$ find operations
  - total running time: $O(N \log^* N)$