CSE 326: Data Structures
Topic #10
The Dynamic (Equivalence) Duo: Union-by-Size & Path Compression

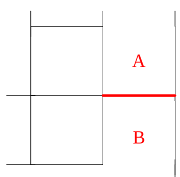
Luke McDowell
Summer Quarter 2003

What's a Good Maze?

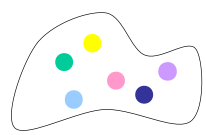
Maze Construction Algorithm

- Given:
  - A collection of rooms **V**
  - Connections between the rooms (initially all closed) **E**
- We want to build a collection of connections to knock down, **E'** ? **E**, such that one unique path connects every two rooms

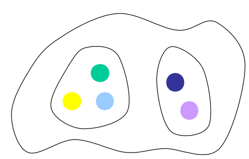
```
While edges remain in E {
  (A, B) = RemoveRandomWall()
  if( A and B have not been
            connected ) {
      Add (A, B) to E'
      Mark A and B as connected
  }
}
```



The Problem, Formally



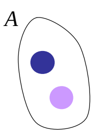
- "If **A** and **B** have not yet been connected"
  - Are two elements in the same set?

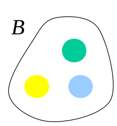


- "Mark **A** and **B** as connected"
  - Form the *union* of two sets

Disjoint Sets ADT

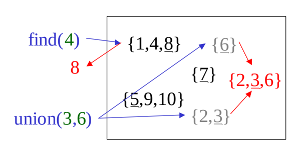
- Find(*x*)
  - Returns set identifier
  - Find(*x*) = Find(*y*) iff *x* and *y* are in the same set
- Union(*A*, *B*)
  - Arguments are set identifiers
  - How do we union the sets containing *x* and *y*?
- MakeNewSet(*item*)
  - Create a new set containing only *item*





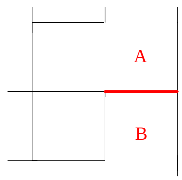
Disjoint Sets Formal Properties

- Equivalence property
  - Every element of a DS belongs to exactly one set
- *Dynamic* equivalence property
  - The set of an element can change after execution of a union



Our Modified Maze Construction Algorithm

```
While edges remain in  E
  (A, B) = RemoveRandomWall()
  if( Find(A) != Find(B) )
     E? = E? U (A, B)
     Union( Find(A), Find(B) )
```

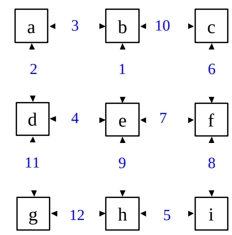


Example

Construct the maze on the right

Initially (the name of each set is underlined):

{<u>a</u>}{<u>b</u>}{<u>c</u>}{<u>d</u>}{<u>e</u>}{<u>f</u>}{<u>g</u>}{<u>h</u>}{<u>i</u>}



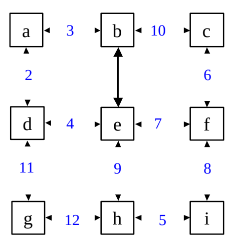
Order of edges in blue

Example, continued

{<u>a</u>}{<u>b</u>}{<u>c</u>}{<u>d</u>}{<u>e</u>}{<u>f</u>}{<u>g</u>}{<u>h</u>}{<u>i</u>}

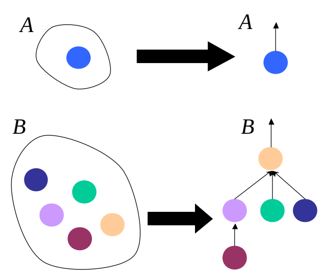
find(b) ? <u>b</u>
find(e) ? <u>e</u>
find(b) ? find(e) so:
  add 1 to **E**?
  union(b, e)

Result:



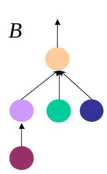
Order of edges in blue

DS ADT Tree Representation



- Maintain a forest of up-trees
- Each set is a tree
- What's the set identifier?

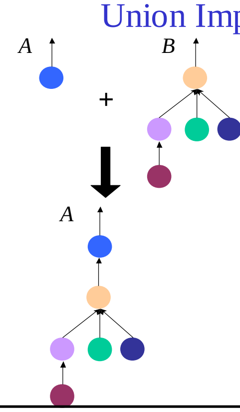
Find Implementation



Find(*x*)
- Walk parents of *x* to the root

Runtime:

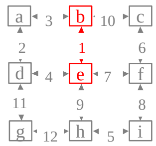
Union Implementation



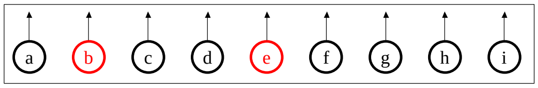
Union(*A*, *B*)
- Join the two trees
- Since *A* and *B* are already the roots of a tree, this is easy!

Runtime:

More of the Example

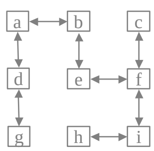


union(b,e)



(extra space)

The Final Maze



Ooh… scary!
Such a hard maze!

Mini-Exercise

Assume union always keeps first argument as the root

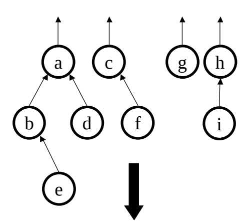
1. Starting with distinct sets a,b,c,d,e,f,g
   - Union(a,c)
   - Union(b,d)
   - Union(a,e)
   - Find(c)
   - Union(e,f)
   - Union(f,a)
   - Union(b,c)
   - Find(c)
2. Must Find(c) always return the same value?
3. Could Union have done a better job?

(extra space)

Nifty storage trick

A forest of up-trees can easily be stored in an array.

Use hashtable to map node names to array indices



	0 (a)	1 (b)	2 (c)	3 (d)	4 (e)	5 (f)	6 (g)	7 (h)	8 (i)
up-index:	-1	0	-1	0	1	2	-1	-1	7

Implementation

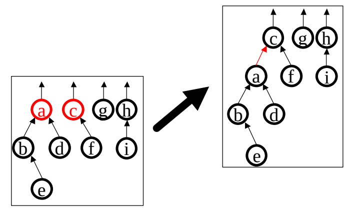
```
int Find(Object x) {
  int xID = hTable[x];

  while(up[xID] != -1) {
    xID = up[xID];
  }

  return xID;
}
```

```
void Union( int x, int y) {
  up[y] = x;
}
```

Improving Union



Could we do a better job on this union?

Union-by-size Code

```
int Union(int x, int y) {
  // If up[x] and up[y] aren't both
  // -1, this algorithm is in trouble

  if (size[x] > size[y]) {
    up[y] = x;
    size[x] += size[y];
  }
  else {
    up[x] = y;
    size[y] += size[x];
  }
}
```

new runtime for Union():

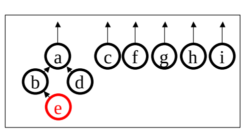
new runtime for Find():

Union-by-Size Find Analysis

- Finds are O(max node depth)
- All nodes start at depth 0
- Depth increases
  - Only when part of smaller tree in a union
  - Only by one (1) level at a time
  - How many times can this happen?

- ? , union runtime =

Improving Find

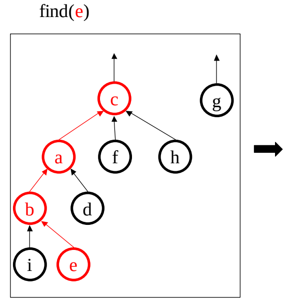


*While we're finding **e**, could we do anything else?*

Wait - what's there to improve?

Path Compression!

find(e)



Exercise

Use union-by-size. Keep the first argument as root if there's a tie.
How many nodes does each Find access?

1. Starting with distinct sets a,b,c,d,e,f,g
   - Union(a,c)
   - Union(b,d)
   - Union(a,e)
   - Union(g,h)
   - Find(c)
   - Union(b,h)
   - Union(e,f)
   - Union(f,a)
   - Union(b,c)
   - Find(c)
   - Find(h)
   - Find(g)
2. Modify the above to also use Path Compression. Does it help?
3. Using union-by-size, what is the worst case depth of any node? Construct a sequence of union operations that produces this for a depth of 5.

(extra space)

Path Compression Code

```
int Find(Object x) {
  // x had better be in
  // the set!
  int xID = hTable[x];
  int i = xID;

  // Get the root for
  // this set
  while(up[xID] != -1) {
    xID = up[xID];
  }

  // Change the parent for
  // all nodes along the path
  while(up[i] != -1) {
     temp = up[i];
     up[i] = xID;
     i = temp;
  }
  return xID;
}
```

(New?) runtime for Find():

Interlude: A Really Slow Function

Ackermann created a <u>really</u> big function A(x, y) with the inverse ? (x, y) which is <u>really</u> small

How fast does ? (x, y) grow?

? (x, y) = 4 for *x* **far** larger than the number of atoms in the universe (2^{300})

? shows up in:

- Computation Geometry (surface complexity)
- Combinatorics of sequences

Complex Complexity of Union-by-Size + Path Compression

Tarjan proved that, with these optimizations, *m* union and find operations on a set of *n* elements have worst case complexity of O(*m*?? (*m*, *n*))

For **all** practical purposes this is amortized constant time: O(*m*?4) for *m* operations!

In some practical cases, one or both optimizations is unnecessary, because trees do not naturally get very deep.

Disjoint Sets ADT Summary

- Also known as Union-Find or Disjoint Set Union/Find
- Simple, efficient implementation
  - With union-by-size and path compression
- Great asymptotic bounds
- Kind of weird at first glance, but lots of applications