## Implementing Union-Find

A set is an unordered collection of things.

One way to implement a set is with a doubly-linked list.

- makeset(x)
  - create a linked list with a single node containing x
  - O(1)
- union(x,y)
  - append y's list to x's list
  - O(1) if you have tail pointers – set x's tail's next to point to y's head
- find(x)
  - how to identify a set? – could use the head node as the representative of the set
  - given a node, it is O(size of list) to find the head of its list – follow prev pointers backwards from node to the head

→ total: O(n × makeset + m log n + m × find + n × union)

= O(n + m log n + nm + n) = O(nm)

- (this is much worse than graph traversal, can we do better?)



## Implementing Union-Find

Can we do better? Find is the slow part…

- what's better than O(n)? → O(1)
  - if every node also had a pointer directly to the head, find(x) could be done in constant time!

New implementation: singly-linked list with tail pointer and each node also pointing directly to the head.

- makeset(x)
  - O(1)

(can actually store head pointers instead of prev pointers since the only reason to back up was to find the head)

- union(x,y)
  - O(1) to append...but O(size of y) to update all head pointers in y
- find(x)
  - O(1)

→ total: O(n × makeset + m log n + m × find + n × union)

= O(n + m log n + m + $n^2$) = O(m log n + $n^2$)

- (somewhat better...)



## Implementing Union-Find

union by rank list implementation

Can we do better? Union is the slow part – what if we updated as few head pointers as possible?

- union(x,y)
  - O(1) to append the smaller list to the larger list...but still O(size of smaller list) to update head pointers in the smaller list
    - can store list sizes so it is possible to find the smaller list in O(1)

→ total: O(n × makeset + m log n + m × find + n × union)

- observation: in the worst case O(size of smaller list) is O(n), but we know something about the series of unions
  - each time we union and the head pointer for a node is updated, the node is going into a set at least twice as big as it come from
  - this can happen at most log n times if there's a total of n elements
- thus n unions with appending the smaller list is O(n log n) instead of O($n^2$)

= O(n + m log n + m + n log n) = O((n+m) log n)

- an improvement for sparse graphs!



## Implementing Union-Find

O((n+m) log n) for Kruskal's algorithm is pretty good – can we do better?

Observation.

- sorting the edges by weight requires O(m log n), which will dominate O(n log n) as long as the graph is connected
  - improving the data structure will result in elapsed time gains, but not change the big-Oh

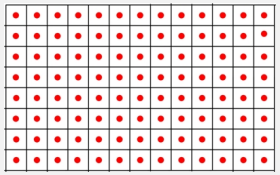
However…

- union-find has applications beyond Kruskal's algorithm
  - greater efficiency in union-find operations may make a difference there

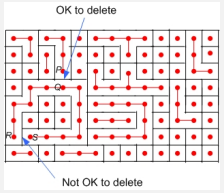
# Maze Creation

In a good maze, every room is reachable from every other and there's only one possible path from start to goal.

How to generate a random maze?



start with all of the walls and every room in a separate set




while there is more than one set left
- choose a random wall
- if the rooms on either side belong to different sets, knock down the wall



# Implementing Union-Find

We improved the implementation to speed up Kruskal's algorithm by making union the slow part instead of find.

We've seen O(n)↔O(1) tradeoffs before...and sometimes could compromise on O(log n) for both.

Will that work here?

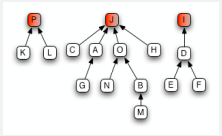
Observation.

- trees are often associated with O(log n) run times
  - the height can be as good as O(log n)



# Implementing Union-Find

Idea:
Represent a set as a directed tree.

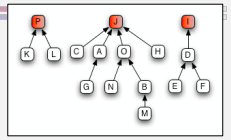



- makeset(x)
  - create a tree with a single node containing x
  - O(1)
- union(x,y)
  - find the roots of x's and y's trees
  - make y's root point to x's root
  - O(find)
- find(x)
  - use the root as the representative element
  - given a node, it is O(height of tree) to find the root of its tree



# Implementing Union-Find

union by rank tree implementation




How tall are the trees?

- could be n – can we do better?
- union(x,y)
  - find the roots of x's and y's trees
  - make the root of the shorter tree point to the root of the taller tree
    - store rank(v) = height of subtree rooted at *v* for each node so the shorter tree can be found in O(1) time
    - only the rank of the taller tree's root may change as a result of the union – O(1) to update
- result is O(log n) height for each tree – so find is O(log n)
  - idea: height of merged tree only increases if the two trees are equally tall – that merge doubles the size of the tree so it can happen at most log n times

→ total: O(n × makeset + m log n + m × find + n × union)
= O(n + m log n + m log n + n log n) = O((n+m) log n)

- (no improvement over union-by-rank lists)

## Implementing Union-Find

O(log n) find(x) isn't bad, but O(1) is still better...

Observation:

- could get O(1) find(x) if each node had a direct pointer to the root

But:

- updating these pointers during union is too expensive

Observation:

- find(x) locates the root for every node between x and the root

It seems a waste to throw that information away!



## Implementing Union-Find

The best of both worlds: (*path compression*)

- union(x,y)
  - find the roots of x's and y's trees
  - make the root of the shorter tree point to the root of the taller tree
- find(x)
  - locate the root
  - update the pointers for every node on the path x→root to point directly to the root



## Implementing Union-Find

Running time?

- find(x) and thus union(x,y) are still O(height of tree)

What's the height of the trees?

- path compression keeps the height of the trees short
- find(x) and union(x,y) are effectively O(1)



## Implementing Union-Find

"Short"?! "Effectively"?!

- based on *amortized time*, not worst case
  - an individual operation may take longer, but if a sequence of *k* operations takes a total of O(k f(n)) time, we can say each operation is O(f(n)) amortized
- total time for *m* find(x) operations is O((m+n) log* n)
  - on average, O(n/m log* n) per find
  - with m > n (typical), this is O(log* n)
    - log* n = the number of successive log operations to bring n down to 1
    - extremely slow growing! (value < 5 for any value of *n* you might encounter, and thus is effectively constant time)

# Union-Find Summary

- union-by-rank list implementation yields O((n+m) log n) for Kruskal's algorithm
  - O(1) makeset(x)
  - O(1) find(x)
  - O(n log n) for a series of *n* union(x,y)
- union-by-rank tree implementation with path compression yields O(m log n) for Kruskal's algorithm
  - O(1) makeset(x)
  - effectively O(1) find(x) and union(x,y)
    - the tree height is a very slow-growing log*
    - *amortized* over a series of operations

Both are an improvement over our initial O(nm) algorithm.