

CSE 100

Find with path compression
and run time for Dijkstra and Prim

Cost of disjoint subsets operations with smarter Union and simple Find

Q: What is the worst case run time of find and union operations in terms of the number of elements N ?

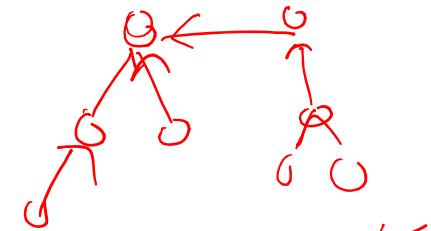
A. find: $O(N)$, union: $O(1)$

B. find: $O(1)$, union: $O(N)$

C. find: $O(\log_2 N)$, union: $O(1)$

D. find: $O(1)$, union: $O(\log_2 N)$

Discuss and revote

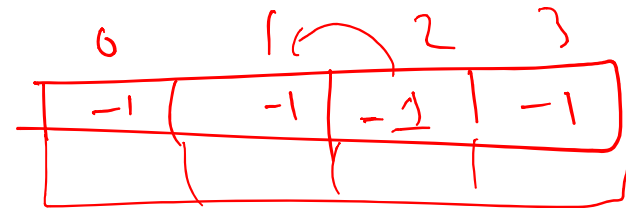


- Therefore, doing $N-1$ union operations (the maximum possible) and M find operations takes time $O(N + M \log_2 N)$ worst case
- With simple unions the complexity was: $O(N + M \times N)$
- This is a big improvement; but we can do still better, by a slight change to the Find operation:
adding path compression (this lecture)

Union-Find Code

- Using the array representation for disjoint subsets, the code for implementing the Disjoint Subset ADT's methods is compact

```
class DisjSets{  
    int *array;  
  
    /**  
     * Construct the disjoint sets object  
     * numElements is the initial number of disjoint sets  
     */  
    DisjSets( int numElements )  
    {  
        array = new int [ numElements ];  
        for( int i = 0; i < numElements; i++ )  
            array[ i ] = -1;  
    }  
}
```



Union-by-height

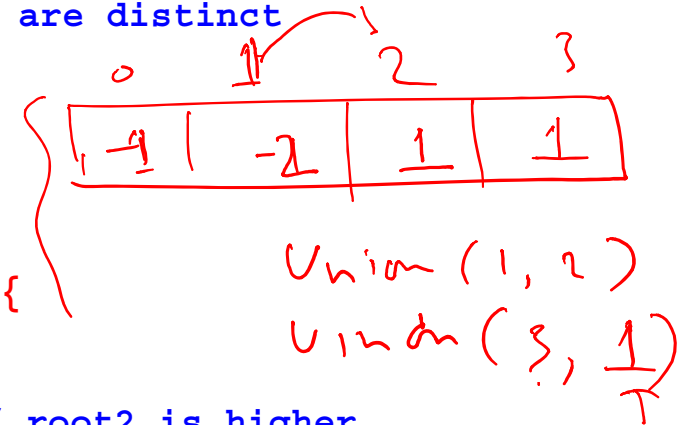
(-1 x height)

```

/**
 * Union two disjoint sets using the height heuristic.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set labels.
 * root1 is the root of set 1
 * root2 is root of set 2
 * returns the root of the union
 */
int union (      int root1, int root2 ) {

    if( array[ root2 ] < array[ root1 ] ) {
        { array[ root1 ] = root2;           // root2 is higher
          return root2;                     // Make root2 new root
        }
    } else {
        if( array[ root1 ] == array[ root2 ] )
        { array[ root1 ]--;                // Update height if same
          array[ root2 ] = root1;           // Make root1 new root
          return root1;
        }
    }
}

```



Simple Find

- In a disjoint subsets structure using parent-pointer trees, the basic Find operation is implemented as:
 - Go to the node corresponding to the item you want to Find the equivalence class for
 - Traverse parent pointers from that node to the root of its tree
 - Return the label of the root
 - This has worst-case time cost $O(\log_2 N)$ 
 - The time cost of doing another Find operation on the same item is the same

Find with Path-compression

- The path-compression Find operation is implemented as:
 - Go to the node corresponding to the item you want to Find the equivalence class for
 - Traverse parent pointers from that node to the root of its tree
 - Return the label of the root
 - But

... as part of the traversal to the root, change the parent pointers of every node visited, to point to the root of the tree (all become children of the root)

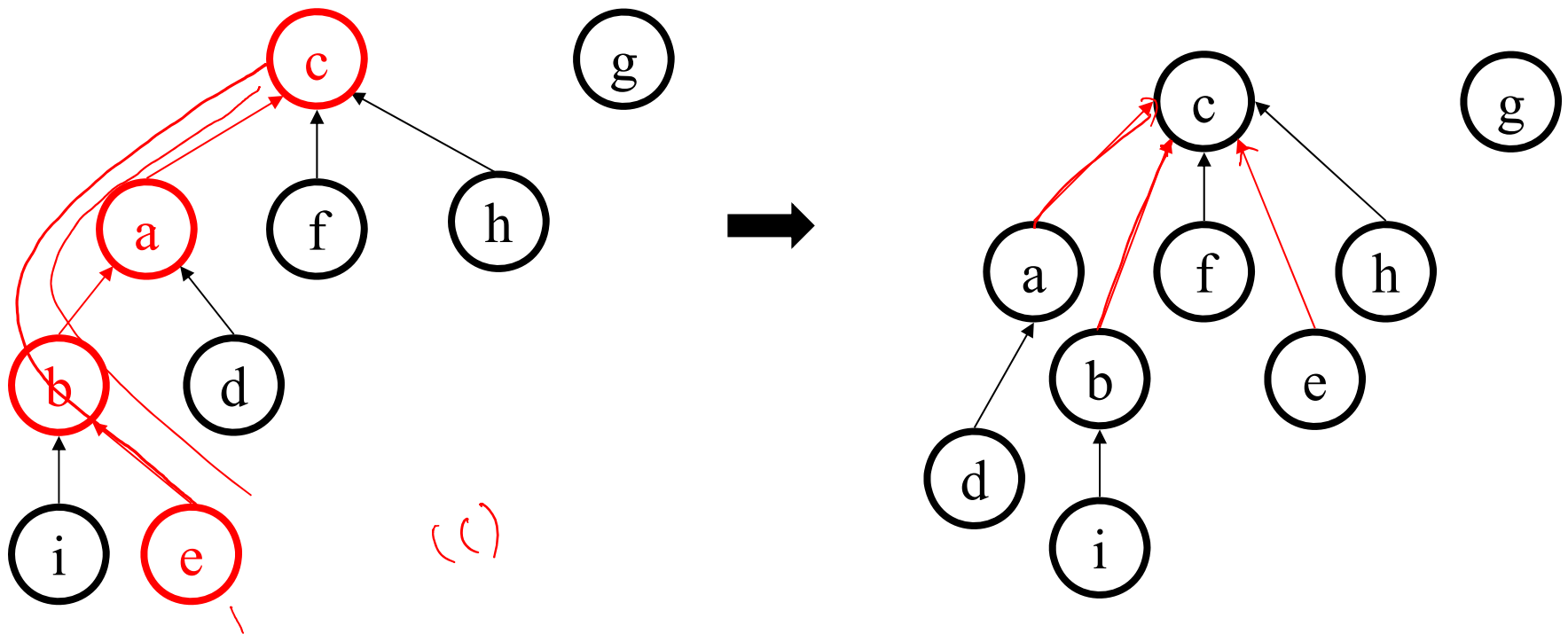
worst-case time cost is still $O(\log N)$

Example of path compression

find(e)

... as part of the traversal to the root, change the parent pointers of every node visited, to point to the root of the tree (all become children of the root)

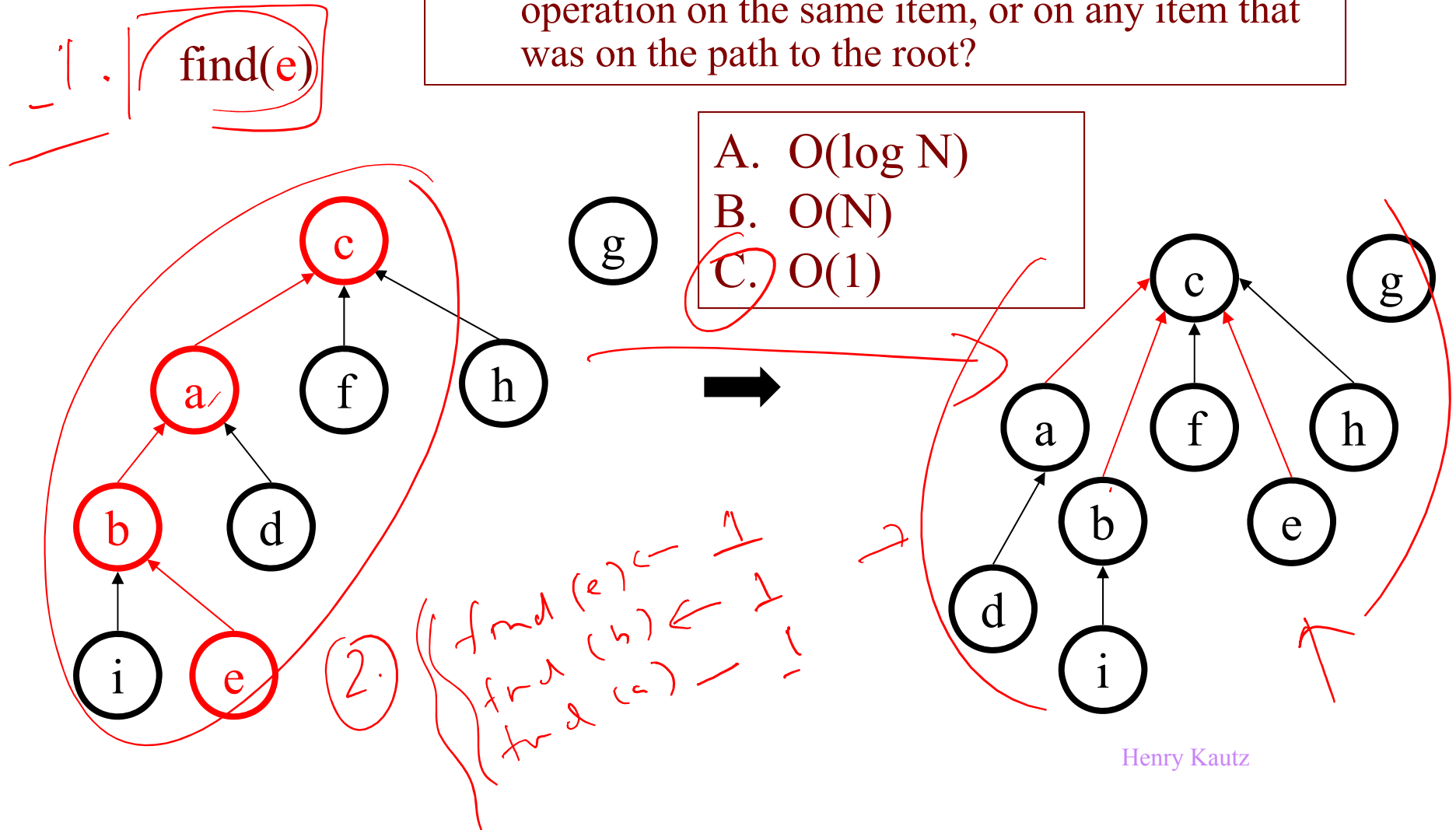
worst-case time cost is still $O(\log N)$



Henry Kautz

Cost of find with path compression

- What is the time cost of doing another Find operation on the same item, or on any item that was on the path to the root?



Disjoint subsets using trees

(Union-by-height and path-compression Find)

- Start with 4 items: 0, 1, 2, 3
- Union(i, j)** makes the root of the shorter tree a child of the root of the taller tree
- We perform path compression ✓
- If an array element contains a negative *int*, then that element represents a tree root, and the value stored there is -1 times (the height of the tree plus 1)

Perform these operations:

Union(2, 3) ←

Union(0, 1)

Find(0) = 0 →

Find(3) = 2

Union(0, 2)

Find(1) = 0

Find(3) = 0

Find(3) = 0

0	1	2	3
-1	-1	-1	-1

-1 -1 -2 2

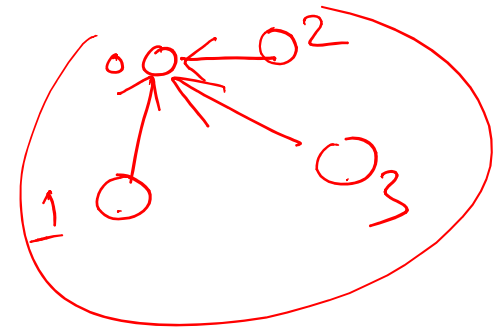
-2 0 -2 2

-2 0 -2 2

-3 0 0 2

-3 0 0 0

-3



Self-adjusting data structures

- Path-compression Find for disjoint subset structures is an example of a self-adjusting structure
- Other examples of self-adjusting data structures are splay trees, self-adjusting lists, skew heaps, etc
- In a self-adjusting structure, a find operation occasionally incurs high cost because it does extra work to modify (adjust) the data structure, with the hope of making subsequent operations much more efficient
- Does this strategy pay off? Amortized cost analysis is the key to the answering that question...


Find with path compression

```
/**  
 * Perform a find with path compression  
 * Error checks omitted again for simplicity  
 * @param x the label of the element being searched for  
 * @return the label of the set containing x  
 */
```

```
int find( int x ) {  
    if( array[ x ] < 0 )  
        return x;  
    else  
        return array[ x ] = ( find( array[ x ] ) );  
}
```

- Note that this path-compression find method does not update the disjoint subset tree heights; so the stored heights (called “ranks”) will overestimate of the true height
- A problem for the cost analysis of the union-by-height method (which now is properly called union-by-rank)

Running Time of Kruskal's algorithm with union find data structure:

1. Sort edges in increasing order of cost

$$|E| \log |E|$$

2. Set of edges in MST, $T = \{\}$

3. For $i = 1$ to $|E|$

$$(u, v) \quad |E|$$

$S1 = \text{find}(u); S2 = \text{find}(v);$

if ($S1 \neq S2$) { //If $T \cup \{e_i = u, v\}$ has no cycles

Add e_i to T

union($S1, S2$)

$$O(1)$$

}

$$\frac{|V| \log |V|}{|E| \log |V|}$$

Amortized cost analysis

- Amortization corresponds to spreading the cost of an infrequent expensive item (car, house) over the use period of the item.
- The amortized cost should be comparable to alternatives such as renting the item or taking a loan.
- Amortized analysis of a data structure considers the average cost over many actions.

Amortized cost analysis results for path compression Find

- It can be shown (Tarjan, 1984) that with Union-by-size or Union-by-height, using path-compression

Find makes any combination of up to $N-1$ Union operations and M Find operations have a worst-case time cost of $O(N + M \log^* N)$

$(\log^* M)$

- This is very good: it is almost constant time per operation, when amortized over the $N-1 + M$ operations!

$$\frac{O(N + M)}{(N-1 + M)} \approx O(1)$$

Amortized cost analysis results for path compression Find

- $\log^* N$ = “log star of N” = smallest k such that $\log^{(k)} n \leq 1$ or # times you can take the log base-2 of N, before we get a number ≤ 1
- Also known as the “single variable inverse Ackerman function”

$$\log^{(k)} n = \underbrace{\log(\log(\log \dots (\log n)))}_{k \text{ logs}}$$

Handwritten examples and calculations:

- $\log^* 2 = 1$
- $\log^* 4 = 2$
- $\log^* 16 = 3$
- $\log^* 65536 = 4$
- $\log^* 2^{65536} = 5$

Other calculations shown:

- $4 = (2)^2$
- $16 = (2)^4 = 2^{2^2} = 2^4$
- $65536 = (2)^{16}$
- $(2^{65536} = \text{a huge number of 20000 digits})$

Handwritten note: $(\log_2 2 = 1)$

- $\log^* N$ grows extremely slowly as a function of N
- It is not constant, but for all practical purposes, $\log^* N$ is never more than 5

$O(1)$

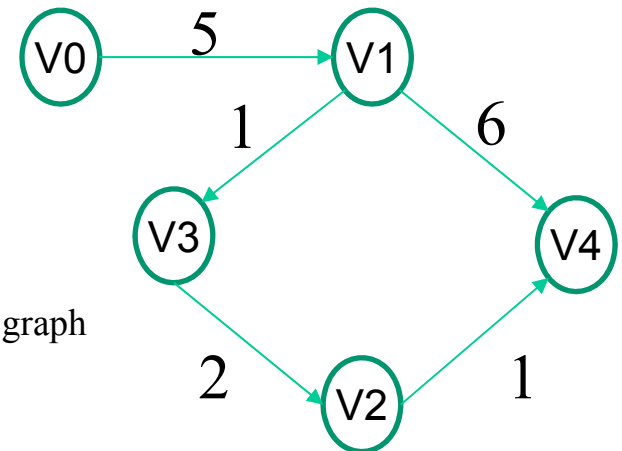
Dijkstra's Algorithm: Run time

- Initialize the graph: Give all vertices a dist of INFINITY, set all “done” flags to false
- Start at s ; give s dist = 0 and set prev field to -1
- Enqueue $(s, 0)$ into a **priority queue**. This queue contains pairs (v, cost) where cost is the best cost path found so far from s to v . It will be ordered by cost, with smallest cost at the head.
- While the priority queue is not empty:
 - Dequeue the pair (v, c) from the head of the queue.
 - If v 's “done” is true, continue
 - Else set v 's “done” to true. We have found the shortest path to v . (It's prev and dist field are already correct).
 - For each of v 's adjacent nodes, w :
 - Calculate the best path cost, c , to w via v by adding the edge cost for (v, w) to v 's “dist”.
 - If c is less than w 's “dist”, replace w 's “dist” c and enqueue (w, c) .

What is the running time of this algorithm in terms of $|V|$ and $|E|$? (More than one might be correct—which is tighter?)

- A. $O(|V|^2)$
- B. $O(|E| + |V|)$
- C. $O(|E| \log(|V|))$
- D. $O(|E| \log(|E|) + |V|)$**
- E. $O(|E| * |V|)$

Assume adjacency list representation for the graph



Prim's MST Algorithm: Run Time

1. Create an empty graph T . Initialize the vertex vector for the graph. Set all “done” fields to false. Pick an arbitrary start vertex s . Set its “done” field to true. Iterate through the adjacency list of s , and put those edges in the priority queue.
2. While the priority queue is not empty:
 - Remove from the priority queue the edge (v, w, cost) with the smallest cost.
 - Is the “done” field of the vertex w marked true?
 - If Yes: this edge connects two vertices already connected in the spanning tree, and we cannot use it. Go to 2.
 - Else accept the edge:
 - Mark the “done” field of vertex w true, and add the edge (v, w) to the spanning tree T .
 - Iterate through w 's adjacency list, putting each edge in the priority queue.

What is the running time of this algorithm in terms of $|V|$ and $|E|$? (More than one might be correct—which is tighter?)

- A. $O(|V|^2)$
- B. $O(|E| + |V|)$
- C. $O(|E| \log(|V|))$
- D. $O(|E| \log(|E|) + |V|)$
- E. $O(|E| * |V|)$