Lecture 14

- An application of disjoint subsets
- Disjoint subset structures and union/find algorithms
- Union-by-size and union-by-height
- Find with path compression
- Amortized cost analysis

Reading: Weiss, Ch. 8

Computing with equivalence classes

- Consider this general problem situation:
  - You are given a set `S` of items
  - You are given some pairs of items in `S` that satisfy some equivalence relation `E()`
  - Given that information, you want to do things like
    - Determine how many equivalence classes there are in `S`, as defined by the pairs of items satisfying `E()` that you have seen so far
    - Given an item in `S`, determine which equivalence class is it in
    - Given two items in `S`, determine whether they are in the same equivalence class
    - Given a new pair of items in `S` that satisfy `E()`, update the system of equivalence classes appropriately
- Problems of that kind come up often in computer applications. We will look at one

Building a random maze

- Suppose you want to construct a nice maze on a NxM grid
- One good way to approach the problem is to see it as a problem of computing with equivalence classes
  - Start with an array of NxM cells, each isolated from its neighbors by 4 ‘walls’.
  - Consider the equivalence relation E(i,j) true iff you can get from cell i to cell j
  - Initially each cell is in its own singleton equivalence class
  - Pick a wall at random. If knocking it down would join two distinct equivalence classes of cells, do so; otherwise leave it standing
  - Continue until all the cells form one equivalence class, then stop
- The result is a connected undirected graph: there is a path between any two cells in the maze. Pick one as entrance, another as exit, and find a path between them!
- Question: Given a NxM maze constructed in this way, how many distinct simple paths are there from the “upper left corner” cell to the “lower right corner” cell?

A Disjoint Subset ADT

- An abstract data type designed for basic computations on equivalence classes is sometimes called a "disjoint subset" structure (since equivalence classes are disjoint subsets of their domain). It is sometimes also called a "union-find" structure because of the names of its principal operations
- Typical Disjoint Subset ADT operations:

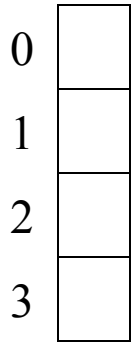
```
Create(int N)
   Create a system of N items, each initially in its own singleton
   equivalence class. These items are labelled with ints 0 up to N.
   The equivalence classes are also labelled with ints 0 up to N.

int Find(int i)
   Return the int label of the equivalence class containing item i.

int Union(int m, int n)
   Perform a set union operation on the equivalence classes with
   labels m and n; return the label of the result.
    (Common variant: perform a set union operation on the
   equivalence classes containing items with labels m and n;
   return the label of the result.)
```

Implementing the disjoint subset structure

- There are many ways to implement a disjoint subset ADT
- For example, you could use linked lists to represent the subsets:
  - Each list node represents an item, and holds that item's integer label
  - Initially, each singleton subset is a linked list with one element; pointers to these lists are kept in an array
  - To do a find operation on an item, you search all linked lists to find the one containing that item's label; return the index of that linked list, as the label of the subset
  - To do a union operation on two subsets, you move all the items from one subset's list into the other subset's list
    - (for `Union(i,j)`, move `j`'s items into `i`'s list)

Disjoint subsets using linked lists

- Start with 4 items: 0, 1, 2, 3



- Perform these operations:

```
Union(2,3)
Union(1,2)
Find(0) =
Find(3) =
Union(0,1)
Find(1) =
Find(0) =
```

Cost of linked-list disjoint subsets operations

- The linked-list implementation is not very efficient:
  - Suppose you start with N items
  - Doing N-1 union operations (the maximum possible) and M find operations takes time $O(N^2 + NM)$ worst case
- This can be made slightly better by being smarter about which list's items to move when doing a union operation...
- But much better performance is available by using a better data structure: trees instead of lists

Disjoint subset using parent-pointer trees

- A better way of implementing a disjoint-subset structure is using trees to represent the subsets
  - Each tree node represents an item, and holds that item's label, and a pointer to that node's parent (nodes can have any number of children... but at most one parent)
  - Initially, each singleton subset is a tree with one node, containing the item's label and a null parent pointer
  - To do a find operation on an item, go to the node for that item, and traverse parent pointers to the root of the tree it's in; return the label in the root node as the label of the subset
  - To do a union operation on two subsets, make the root of one subset's tree point to the root of the other subset's tree
    - (for now, assume `Union(i,j)`, makes `i` the parent)

Disjoint subsets using trees (simple Union and Find)

- Start with 4 items: 0, 1, 2, 3

- Perform these operations:

```
Union(2,3)
Union(1,2)
Find(0) =
Find(3) =
Union(0,1)
Find(1) =
Find(0) =
```

Cost of parent-pointer tree disjoint subsets operations

- If you aren't careful about how you join trees when doing Unions, and don't do anything clever when doing Finds, the tree implementation is almost as bad as the list implementation in the worst case:
  - Suppose you start with N items
  - Doing N-1 union operations (the maximum possible) and M find operations takes time O(N + NM) worst case
- The worst case happens when the trees constructed with the Union operation are essentially linked lists:
  - each Union operation just takes constant time, but each Find can take O(N) worst case
- Find performance can be improved considerably with some changes to the Union operation

Smarter Union operations

- In the Union operation, a tree becomes like a linked list if, when joining two trees, the root of the smaller tree (e.g. a single node) always becomes the root of the new tree

- This can be prevented by making sure that in a Union operation, the *larger* of the two trees' roots becomes the root of the new tree (ties are broken arbitrarily)

- How to measure "larger"? Either of these ways will work:

  - Union-by-size: In the root of each tree is stored the size (number of nodes) in the tree. When doing a union, the root with the larger size becomes the parent (break ties arbitrarily); its sored size is updated to be the sum of its former size and the size of its new child.

  - Union-by-height: In the root of each tree is stored the height of the tree. When doing a union:
    - If one root shows greater height than the other, it becomes the parent. Its stored height doesn't need to be updated.
    - If the roots show equal height, pick either one as the parent. Its stored height should be increased by one.

Disjoint subsets using trees (Union-by-height and simple Find)

- Start with 4 items: 0, 1, 2, 3

- Perform these operations:

```
Union(2,3)
Union(1,2)
Find(0) =
Find(3) =
Union(0,2)
Find(1) =
Find(0) =
```

Cost of disjoint subsets operations with smarter Union

- Either union-by-size or union-by-height will guarantee that the height of any tree is no more than $\log_2 N$, where N is the total number of nodes in all trees
- Each union still takes only O(1) time; but now each find operation takes worst-case O(logN) time
- Therefore, doing N-1 union operations (the maximum possible) and M find operations takes time O(N + M logN) worst case
- This is a big improvement; but we can do still better, by a slight change to the Find operation: adding *path compression*

Path-compression 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 N). And the time cost of doing another Find operation on the same item is the same
- 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 on that path, to point to the root of the tree (they all become children of the root)
  - This also has worst-case time cost O(log N). However, the time cost of doing another Find operation on the same item, or on any item that was on the path to the root, is now O(1) ...!

Disjoint subsets using trees (Union-by-height and path-compression Find)

- Start with 4 items: 0, 1, 2, 3

- Perform these operations:

```
Union(2,3)
Union(0,1)
Find(0) =
Find(3) =
Union(0,2)
Find(1) =
Find(1) =
```

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...

Amortized cost analysis

- In ordinary algorithmic analysis, you look at the time or space cost of doing a single operation in either the best case, average case, or worst case
- But it might be a good strategy in designing an algorithm to do extra work during one operation, to make subsequent operations much faster (path compression Find for disjoint subsets is one example)
- Ordinary worst-case analysis might make this strategy look bad: a single operation in the worst case could take a lot of time
- *Amortized analysis* considers the time or space cost of doing a *sequence* of operations (in either the best case, average case, or worst case)
- Some of these operations can individually be quite expensive; but amortized analysis will show if there is a payoff for that extra work:
  - The total cost of the entire sequence of operations might be less with that extra work, than without!

Amortized cost analysis results for path compression Find

- It can be shown 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 worst-case time cost O(N + M log* N)
- This is very good: it is almost constant time per operation, when amortized over the N-1 + M operations!

- log* N (read: "log star of N", also known as the "single variable inverse Ackerman function") is equal to the number of times you can take the log base-2 of N, before you get a number less than or equal to 1
  - log* 2 = 1
  - log* 4 = 2 (note that $4 = 2^2$)
  - log* 16 = 3 (note that $16 = 2^{2^2}$)
  - log* 65536 = 4 (note that $65536 = 2^{2^{2^2}}$)
  - log* 2^{65536} = 5 (note that $2^{65536} = 2^{2^{2^{2^2}}}$ is a *huge* number)

- 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

Implementing parent-pointer trees using arrays

- A very compact and elegant implementation of disjoint subsets suitable for union-by-height and path-compression find can be done by using arrays
- Create an array A of ints of length N, the number of items
- These items will have labels 0, 1, ..., N-1
- Array element indexed *i* represents the node with label *i*
- 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)
- If an array element contains a nonnegative int, then
  - that element represents a non-root, and the value stored there is the index of its parent

Using an array: example

- Write the forest of trees, showing parent pointers and node labels, represented by this array:

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

Union/Find: C++ code

- Using the array representation for disjoint subsets, the code for implementing the Disjoint Subset ADT's methods is very compact
- (This code is too simple... it works, but it would be good to put in some error-checking)

```
class DisjSets{
  private:
   int * array;

  public:
   /**
    * Construct the disjoint sets object,
    * given the initial number of disjoint sets.
    */
   DisjSets( int numElements )     {
       array = new int [ numElements ];
       for( int i = 0; i < array.length; i++ )
           array[ i ] = -1;
   }
```

Union-by-height

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

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

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 be overestimates of the true heights
- Is this a problem for the cost analysis of the union-by-height method (which now is properly called union-by-rank)? Why or why not?

Next time...

- Simple vs. not-so-simple algorithm cost analysis
- The cost of accessing memory
- B-trees
- B-tree performance analysis
- B-tree find, insert, and delete operations
- B-tree example: a 2-3 tree

Reading: Weiss, Ch. 4 section 7