lecture24: Disjoint Sets

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

22nd July, 2013
Announcements

- mp6 due tonight
- mp7 out soon!
- mt3 tomorrow night (7/23)
- Optional review instead of lab tomorrow
Code Challenge Friday Night! 6pm in the lab
A New Task

Given the set \{C\#, C++, Java, Ruby, Python\}, partition students by their favorite programming language. Each student has an integer ID (we can keep track of this mapping with a Dictionary).
What kind of operations can we perform?

- `find(4)`
  - What does this return?
- `find(3) == find(1)?`
- `if find(2) != find(0), then union(2, 0)`
We’re modeling equivalence relations

What is an equivalence relation again?
A given binary relation \( \sim \) on a set \( S \) is said to be an equivalence relation if and only if it is reflexive, symmetric and transitive.

Equivalently, \( \forall a, b, c \in S \):

- \( a \sim a \) (reflexive)
- \( a \sim b \rightarrow b \sim a \) (symmetric)
- \( a \sim b \land b \sim c \rightarrow a \sim c \) (transitive)

How does this translate to our programming language example?
Another example

How could you model a social network using this framework?
What is the equivalence relation?

What about RGBA Pixels in a PNG?
Partitioning a Social Network
Partitioning Regions in an Image
We need a **union-find ADT**: Disjoint Sets. It supports

1. **add elements**, to initialize the structure: *initially all elements are in their own set*

2. **find**, to return which set an element belongs to (*by returning a representative element from the set*)

3. **union**, to connect two elements together

```cpp
class DisjointSets {
   public:
      void addelements(int num);
      int find(int id);
      void union(int id1, int id2);
   private:
      // ???
};
```
Our first implementation

```java
DisjointSets dsets;
dsets.addelements(8);

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

dsets.union(2, 5);

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

dsets.union(5, 7);
dsets.union(0, 1);
dsets.union(0, 6);

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

- When we create the set, each element is its own set and its own representative.
- When unioning, choose an arbitrary element as the set representative.
- How does find work?
- What if we wanted to do `union(1, 2)`?
- What is the running time of `find` and `union`?
DisjointSets via an array

It seems our array implementation gives great running times for find, but union can be very inefficient: $O(n)$ when combining large sets together.

We should think of a way to get sublinear running times for our DisjointSets operations...
UpTrees: A better DS implementation

- We can imagine our sets as a collection of trees (Uptrees) instead of elements in an array.
- The root of each tree is the representative of the set it is in.
- When unioning two elements, we can just point the root at the set we’re unioning with!
- We don’t even have to use pointers for Uptrees – instead, each element’s field holds the index of its parent (roots can have -1 as their data).
- Let’s draw what this looks like...
Imagining Uptrees

DisjointSets dsets;
dsets.addelements(8);

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

dsets.union(2, 5);

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

dsets.union(5, 7);
dsets.union(0, 1);
dsets.union(0, 6);

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>7</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

- When unioning, we need to deal with the roots to make sure all elements are unioned correctly.
- By default, let's point the root of the first argument to the root of the second.
- Can you draw the uptrees?
Concept Check: Draw the Uptrees

Draw the forest of Uptrees if this is our array.

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

Could you write code that generated this forest? How many disjoint sets are there?
// return the root of the Uptree containing ‘‘id’’
int DisjointSets::find(int id)
{

}

// combine the sets containing id1 and id2
void DisjointSets::union(int id1, int id2)
{

}
The running time of DS operations using Uptrees depends on...

So the worst case running time is still $O(n)$

What does an ideal Uptree look like?

If we have to union two Uptrees together, how can we do it so we don’t increase the height?
Smart Unions

- **Union by height**
  - Keeps the overall height of the tree as short as possible
- **Union by size**
  - Increases the distance to the root for the fewest number of nodes
- Both of these smart union schemes guarantee a $O(\log n)$ uptree height in all the disjoint sets
// combine the sets containing id1 and id2
void DisjointSets::union(int id1, int id2)
{
    int root1 = find(id1);
    int root2 = find(id2);
    int newSize = _set[root1] + _set[root2]; // ??

    // root1 has more elements
    if(_set[root1] < _set[root2])
    {
    
    }

    // root2 has more elements (or same)
    else
    {
    
    }
}
Union By Height

- We think you should be able to write a union-by-height implementation, too
- (In mp7, we’ll assume you write a union-by-size though)
- What other considerations do we have to take if doing union-by-height?
Another Optimization

- After computing the root in `find`, set the current node's parent to be that root!
- This makes our trees much *shorter*, which is great because we already know all the operations on our disjoint sets are $O(h)$.

```cpp
int DisjointSets::find(int id) {
}
```
Union-find with smart unions and path compression
New running times...

\[ \log^* n \equiv \text{number of times you can take the logarithm of } n \text{ before you get to 1} \]

<table>
<thead>
<tr>
<th>(n)</th>
<th>(\log^* n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>65,536</td>
<td>4</td>
</tr>
<tr>
<td>2^{65,536}</td>
<td>5</td>
</tr>
</tbody>
</table>

In any sequence of \(m\) union and find operations on a collection of \(n\) items, the running time is \(O(m \log^* n)\) (effectively constant).
Using DS

- mp7 (how?)
- lab_graphs: Kruskal’s Algorithm
- lots of other applications...