CMSC 341

Disjoint Sets

Textbook Chapter 8
Equivalence Relations

- A relation $R$ is defined on a set $S$ if for every pair of elements $(a, b)$ with $a, b \in S$, $a \, R \, b$ is either true or false. If $a \, R \, b$ is true, we say that “$a$ is related to $b$”.

- An equivalence relation is a relation $R$ that satisfies three properties
  - (Reflexive) $a \, R \, a$ for all $a \in S$
  - (Symmetric) $a \, R \, b$ if and only if $b \, R \, a$
  - (Transitive) $a \, R \, b$ and $b \, R \, c$ implies that $a \, R \, c$
Equivalence Relation Examples

- $=$, but not $\leq$
- Students with the same eye color
- All cities in the same country
- Computers connected in a network
Equivalence Classes

- The equivalence class for an element \( a \in S \) is the subset of \( S \) that contains all the elements that are related to \( a \).

- The subsets that represent the equivalence classes will be “disjoint”

Example
- All students in CMSC 341 who are juniors
Equivalence Relation Application

- Suppose we have an application involving $N$ distinct items. We will not be adding new items, nor deleting any items. Our application requires us to use an equivalence relation to partition the items into a collection of equivalence classes (subsets) such that:
  - each item is in a set,
  - no item is in more than one set.

- Examples
  - Classify UMBC students according to class rank.
  - Classify CMSC 341 students according to GPA.
Disjoint Set Terminology

- We identify a set by choosing a representative element of the set. It doesn’t matter which element we choose, but once chosen, it can’t change.

- There are two operations of interest:
  - find (x) -- determine which set x is in. The return value is the representative element of that set
  - union (x, y) -- make one set out of the sets containing x and y.

- Disjoint set algorithms are sometimes called union-find algorithms.
Disjoint Set Example

Given a set of cities, \( C \), and a set of roads, \( R \), that connect two cities \((x, y)\) determine if it’s possible to travel from any given city to another given city.

```
for (each city in C)
    put each city in its own set
for (each road \((x, y)\) in R)
    if (find( \(x\) ) != find( \(y\) ))
        union(\(x, y\))
```

Now we can determine if it’s possible to travel by road between two cities \(c_1\) and \(c_2\) by testing

\[
\text{find}(c_1) == \text{find}(c_2)
\]
Up-Trees

- A simple data structure for implementing disjoint sets is the *up-tree*.

H, A and W belong to the same set. H is the representative.

X, B, R and F are in the same set. X is the representative.
Operations in Up-Trees

find() is easy. Just follow pointer to representative element. The representative has no parent.

```java
find(x)
{
    if (parent(x)) // not the representative
        return(find(parent(x)));
    else
        return (x); // representative
}
```
Union

- Union is more complicated.

- Make one representative element point to the other, but which way? Does it matter?

- In the example, some elements are now twice as deep as they were before.
Union(H, X)

X points to H.

B, R and F are now deeper.

H points to X.

A and W are now deeper.
A Worse Case for Union

Union can be done in $O(1)$, but may cause find to become $O(n)$.

Consider the result of the following sequence of operations:

- Union (A, B)
- Union (C, A)
- Union (D, C)
- Union (E, D)
Array Representation of Up-tree

- Assume each element is associated with an integer $i = 0 \ldots n-1$. From now on, we deal only with $i$.
- Create an integer array, $s[n]$
- An array entry is the element’s parent
- $s[i] = -1$ signifies that element $i$ is the representative element.
Union/Find with an Array

Now the union algorithm might be:

```java
public void union(int root1, int root2) {
    s[root2] = root1; // attaches root2 to root1
}
```

The find algorithm would be

```java
public int find(int x) {
    if (s[x] < 0)
        return (x);
    else
        return (find(s[x]));
}
```
Improving Performance

- There are two heuristics that improve the performance of union-find.
  - Path compression on find
  - Union by weight
Path Compression

Each time we find( ) an element E, we make all elements on the path from E to the root be immediate children of root by making each element’s parent be the representative.

```java
public int find(int x) {
    if (s[x] < 0)
        return (x);
    s[x] = find(s[x]);  // new code
    return (s[x]);
}
```

When path compression is used, a sequence of m operations takes O(m lg n) time. Amortized time is O(lg n) per operation.
“Union by Weight” Heuristic

Always attach the smaller tree to larger tree.

```java
public void union(int root1, int root2) {
    rep_root1 = find(root1);
    rep_root2 = find(root2);
    if (weight[rep_root1] < weight[rep_root2]) {
        s[rep_root1] = rep_root2;
        weight[rep_root2] += weight[rep_root1];
    } else {
        s[rep_root2] = rep_root1;
        weight[rep_root1] += weight[rep_root2];
    }
}
```
Performance with Union by Weight

- If unions are performed by weight, the depth of any element is never greater than $\lg N$.

  - Intuitive Proof:
    - Initially, every element is at depth zero.
    - An element’s depth only increases as a result of a union operation if it’s in the smaller tree in which case it is placed in a tree that becomes at least twice as large as before (union of two equal size trees).
    - Only $\lg N$ such unions can be performed until all elements are in the same tree.

- Therefore, $\text{find}(\ )$ becomes $O(\lg n)$ when union by weight is used -- even without path compression.
Performance with Both Optimizations

- When both optimizations are performed a sequence of $m$ ($m \geq n$) operations (unions and finds), takes no more than $O(m \lg^* n)$ time.
  - $\lg^* n$ is the iterated (base 2) logarithm of $n$ -- the number of times you take $\lg n$ before $n$ becomes $\leq 1$.

- Union-find is essentially $O(m)$ for a sequence of $m$ operations (amortized $O(1)$).
A Union-Find Application

- A random maze generator can use union-find. Consider a 5x5 maze:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>
Maze Generator

- Initially, 25 cells, each isolated by walls from the others.
- This corresponds to an equivalence relation -- two cells are equivalent if they can be reached from each other (walls been removed so there is a path from one to the other).
Maze Generator (cont.)

- To start, choose an entrance and an exit.

IN →

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

OUT →
Randomly remove walls until the entrance and exit cells are in the same set.

Removing a wall is the same as doing a union operation.

Do not remove a randomly chosen wall if the cells it separates are already in the same set.
MakeMaze

MakeMaze(int size) {
    entrance = 0; exit = size-1;
    while (find(entrance) != find(exit)) {
        cell1 = a randomly chosen cell
        cell2 = a randomly chosen adjacent cell
        if (find(cell1) != find(cell2)
            union(cell1, cell2)
    }
}
Initial State

\[
\begin{array}{|c|c|c|c|c|}
\hline
0 & 1 & 2 & 3 & 4 \\
\hline
5 & 6 & 7 & 8 & 9 \\
\hline
10 & 11 & 12 & 13 & 14 \\
\hline
15 & 16 & 17 & 18 & 19 \\
\hline
20 & 21 & 22 & 23 & 24 \\
\hline
\end{array}
\]

\{0\} \{1\} \{2\} \{3\} \{4\} \{5\} \{6\} \{7\} \{8\} \{9\} \{10\} \{11\} \{12\} \{13\} \{14\} \{15\} \{16\} \{17\} \{18\} \{19\} \{20\} \{21\} \{22\} \{23\} \{24\}
Intermediate State

- Algorithm selects wall between 18 and 13. What happens?

```
0  1  2  3  4
5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```
A Different Intermediate State

- Algorithm selects wall between 8 and 13. What happens?

{0, 1} {2} {3} {4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22} {5} {10, 11, 15} {12} {19} {20} {21} {23} {24}
Final State

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
5 & 6 & 7 & 8 & 9 \\
10 & 11 & 12 & 13 & 14 \\
15 & 16 & 17 & 18 & 19 \\
20 & 21 & 22 & 23 & 24 \\
\end{array}
\]

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24\}