Chapter 24

The Disjoint Set Class
**figure 24.1**

A $50 \times 88$ 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>

{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}

**figure 24.2**

Initial state: All walls are up, and all cells are in their own sets.
At some point in the algorithm, several walls have been knocked down and sets have been merged. At this point, if we randomly select the wall between 8 and 13, this wall is not knocked down because 8 and 13 are already connected.
We randomly select the wall between squares 18 and 13 in Figure 24.3; this wall has been knocked down because 18 and 13 were not already connected, and their sets have been merged.
Figure 24.5
Eventually, 24 walls have been knocked down, and all the elements are in the same set.

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

\{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\}
Figure 24.6
(a) A graph $G$ and
(b) its minimum spanning tree.
Kruskal's algorithm after each edge has been considered. The stages proceed left-to-right, top-to-bottom, as numbered.
The nearest common ancestor for each request in the pair sequence \((x, y), (u, z), (w, x), (z, w)\), and \((w, y)\) is \(A, C, A, B,\) and \(y\), respectively.
**Figure 24.9**
The sets immediately prior to the return from the recursive call to $D$; $D$ is marked as visited and $\text{NCA}(D, v)$ is $v$'s anchor to the current path.
After the recursive call from $D$ returns, we merge the set anchored by $D$ into the set anchored by $C$ and then compute all $\text{NCA}(C, v)$ for nodes $v$ marked prior to completing $C$'s recursive call.
Figure 24.12
A forest and its eight elements, initially in different sets
**Figure 24.13**
The forest after the union of trees with roots 4 and 5.
Figure 24.14

The forest after the union of trees with roots 6 and 7.
figure 24.15
The forest after the union of trees with roots 4 and 6
**Figure 24.16**
The forest formed by union-by-size, with the sizes encoded as negative numbers.
**figure 24.17**

Worst-case tree for $N = 16$
A forest formed by union-by-height, with the height encoded as a negative number.
**Figure 24.19**
Path compression resulting from a \texttt{find(14)} on the tree shown in Figure 24.17.
package weiss.nonstandard;

// DisjointSets class
// CONSTRUCTION: with int representing initial number of sets
// ***************PUBLIC OPERATIONS***************
// void union( root1, root2 ) --> Merge two sets
// int find( x ) --> Return set containing x
// ***************ERRORS***********************
// Error checking or parameters is performed

public class DisjointSets {
    public DisjointSets( int numElements )
    { /* Figure 24.21 */ }

    public void union( int root1, int root2 )
    { /* Figure 24.21 */ }

    public int find( int x )
    { /* Figure 24.21 */ }

    private int [ ] s;

    private void assertIsRoot( int root )
    {
        assertIsItem( root );
        if( s[ root ] >= 0 )
            throw new IllegalArgumentException();
    }

    private void assertIsItem( int x )
    {
        if( x < 0 || x >= s.length )
            throw new IllegalArgumentException();
    }

    figure 24.20
    The disjoint sets class skeleton
/**
 * Construct the disjoint sets object.
 * @param numElements the initial number of disjoint sets.
 */
public DisjointSets( int numElements )
{
    s = new int[ numElements ];
    for( int i = 0; i < s.length; i++ )
        s[ i ] = -1;
}
/**
 * Union two disjoint sets using the height heuristic.
 * root1 and root2 are distinct and represent set names.
 * @param root1 the root of set 1.
 * @param root2 the root of set 2.
 * @throws IllegalArgumentException if root1 or root2
 * are not distinct roots.
 */
public void union( int root1, int root2 )
{
    assertIsRoot( root1 );
    assertIsRoot( root2 );
    if( root1 == root2 )
        throw new IllegalArgumentException();
    if( s[ root2 ] < s[ root1 ] ) // root2 is deeper
        s[ root1 ] = root2;  // Make root2 new root
    else
    {
        if( s[ root1 ] == s[ root2 ] )
            s[ root1 ]--; // Update height if same
        s[ root2 ] = root1;  // Make root1 new root
    }
}
/**
 * Perform a find with path compression.
 * @param x the element being searched for.
 * @return the set containing x.
 * @throws IllegalArgumentException if x is not valid.
 */
public int find( int x )
{
    assertIsItem( x );
    if( s[ x ] < 0 )
        return x;
    else
    {
        return s[ x ] = find( s[ x ] );
    }
}
### Figure 24.22

Possible partitioning of ranks into groups

<table>
<thead>
<tr>
<th>Group</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>3</td>
<td>5 through 9</td>
</tr>
<tr>
<td>4</td>
<td>10 through 16</td>
</tr>
<tr>
<td>i</td>
<td>((i-1)^2) through (i^2)</td>
</tr>
</tbody>
</table>
**Figure 24.23**

Actual partitioning of ranks into groups used in the proof

<table>
<thead>
<tr>
<th>Group</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3, 4</td>
</tr>
<tr>
<td>4</td>
<td>5 through 6</td>
</tr>
<tr>
<td>5</td>
<td>17 through 65,536</td>
</tr>
<tr>
<td>6</td>
<td>65,537 through $2^{65,536}$</td>
</tr>
<tr>
<td>7</td>
<td>Truly huge ranks</td>
</tr>
</tbody>
</table>
A graph $G$ for Exercise 24.3