Chapter 24
The Disjoint Set Class

Introduction

- We are looking to solve the equivalence problem: the disjoint set class.
- It is easy to implement and takes very little code.
- We will look at:
  - Three simple applications of the disjoint set class.
  - A way to implement the set with minimal effort.
  - A method that increases the speed of the class using two simple observations.
  - Analysis of running time.

Equivalence Relations

- A relation $R$ is defined on a set $S$ if for every pair of elements $(a, b)$, $a, b \in S$, $a R b$ 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 satisfied three properties:
  - Reflexive: $a R a$ is true 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 $a R c$.

- Electrical connectivity, where all connections are by metal wires, is an equivalence relation.
  - It is reflexive as a component is connected to itself.
  - It is symmetric since if $a$ is connected to $b$, $b$ is also connected to $a$.
  - It is transitive since if $a$ is connected to $b$, and $b$ is connected to $c$, then $a$ is also connected to $c$.
- An example of a relation in which town $a$ is related to town $b$ if traveling from $a$ to $b$ by road is possible.
  - This relationship is an equivalence relation if the roads are two-way.

Dynamic Equivalence and Two Applications

- For any equivalence relation, denoted $\sim$, the natural problem is to decide for any $a$ and $b$ whether $a \sim b$.
  - If the relation is stored as a two-dimensional array of Boolean variables, equivalence can be tested in constant time.
  - Unfortunately, the relation is usually implicitly, rather than explicitly defined.
- Assume we have the set $\{a_1, a_2, a_3, a_4, a_5\}$
  - We would need a $5 \times 5$ array.
However, if \( a_1 \sim a_2, a_3 \sim a_4, a_1 \sim a_5, a_4 \sim a_2 \) are all related implies that all pairs are related.

- How can we find this quickly.
- The equivalence class of an element \( x \in S \) is the subset of \( S \) that contains all the elements related to \( x \).
  - Note the equivalence classes form a partition of \( S \).
  - To decide whether \( a \sim b \), we need only check whether \( a \) an \( b \) are in the same equivalence class.
- Disjoint sets are sets such that \( S_i \cap S_j = \emptyset \).
  - The two basic disjoint set class operations are:
    - find – return the name of the set (i.e., equivalence class containing a given element.
    - union – adds relations to a set.
  - If we want to add a pair \( (a, b) \) to the list of relations, we:
    - Determine whether \( a \) and \( b \) are related.
      - Done by doing a find on both \( a \) and \( b \) and finding out if they are in the same equivalence class.
      - If they are not, apply union.
  - These operations are dynamic because, during the course of the algorithm execution, the sets can change via the union operation.
- Before we look at the implementation of the union and find operations, we look at some applications.

**Generating Mazes**

- An example of a maze is shown in Figure 1.

![Figure 1 50 x 88 maze](image)

- An algorithm for generating the maze:
  - Start with walls everywhere (except for the entrance and exit).
  - Continually choose a wall randomly
- Knock the wall down if the cells that the wall separates are not already connected to each other (same equivalence class).
- Repeat the process until the starting and ending cells are connected
- The application of this algorithm is demonstrated using a 5×5 maze.
  - Figure 2 shows the initial state.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</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 2** All walls are up, and all cells are their own set

- Figure 3 shows a later stage after a few walls have been knocked down.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
</tbody>
</table>

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

**Figure 3** At a later point

- Suppose the wall between 8 and 13 is considered.
  - It would not be knocked down since 8 and 13 are in the same set.
- Suppose we select the wall between 18 and 13.
  - Using the find operation, we see that they are in different sets.
  - Knock down the wall.
  - The sets containing 18 and 13 are combined using the union operation giving us Figure 4.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
</tbody>
</table>

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

**Figure 4** Combining cell 18 and 13
Figure 5 shows all the necessary walls have been knocked down and everything is in the same set.

![Figure 5 Final maze]

The running time is dominated by the union/find costs.
- If you analyze the problem, you will find that there are \( O(N) \) union operations and \( O(N) \) find operations.

**Minimum Spanning Trees**

- A *spanning tree* of an undirected graph is a tree formed by graph edges that connect all the vertices of the graph.
- A *minimum spanning tree* is a connected subgraph of \( G \) that spans all vertices at minimum cost.
  - The number of edges in the minimum spanning tree is \( |V| - 1 \).
- Figure 6(b) is the minimum spanning tree of the graph in Figure 6(a).
  - In this case, the minimum spanning tree happens to be unique. This is unusual.

![Figure 6 Graph and minimum spanning tree]

- Suppose we need to connect several towns with roads, minimizing the total construction cost, with the provision that we can transfer to another road only at a town.
- Kruskal’s algorithm, used to find the minimum spanning tree, is simple.
  - It continually selects edges in order of smallest weight to add to the tree if it does not cause a cycle.
  - Figure 7 shows the action of Kruskal’s algorithm on the graph shown in Figure 6.
  - Notice in step 6 that edges \((v_1, v_3)\) and \((v_0, v_2)\) are rejected because either would cause a cycle.
How do we determine whether an edge \((u, v)\) should be accepted or rejected?

- Maintain each connected component in the spanning forest as a disjoint set.
- If \(u\) and \(v\) are in the same disjoint set, as determined by two find operations, the edge is rejected because \(u\) and \(v\) are already connected.
- Otherwise, the edge is accepted and a union operation is performed on the two disjoint sets containing \(u\) and \(v\), in effect, combining the connected components.

**Nearest Common Ancestor**

- Problem: Given a tree and a list of pairs of nodes in the tree, find the nearest common ancestor for each pair of nodes.
- Consider the tree in Figure 8 with a pair list containing five requests: \((x, y)\), \((u, z)\), \((w, x)\), \((z, w)\), and \((w, y)\).
The response is $A$, $C$, $A$, $B$, and $y$, respectively.

Figure 8 Nearest common ancestor and pair sequence

- The algorithm works as follows:
  - Perform a postorder tree traversal.
  - When we are about to return from processing a node, examine the pair list to determine whether any ancestor calculations are to be performed.
  - If $u$ is the current node, $(u, v)$ is in the pair list and we have already finished the recursive call to $v$, we have enough information to determine $NCA(u, v)$.
- Consider Figure 9 to understand how this algorithm works.

Figure 9 Sets explored prior to the return from the recursive call to $D$

- All nodes in the areas surrounded by a dotted line have been visited.
- All recursive calls, but the one to $D$, have finished.
- A node is marked after its recursive call has been completed.
- The anchor of a visited (but not necessarily marked) node $v$ is the node on the current path that is closest to $v$.
- $p$’s anchor is $A$, $q$’s anchor is $B$, and $r$ is unanchored because it has yet to be visited.
- The visited nodes form an equivalence class.
  - Two nodes are related if they have the same anchor.
  - Each unvisited node is an equivalence class of its own.
- Suppose the $(D, v)$ is in the pair list. Then we have three cases:
  - $v$ is unmarked, so we have no information to compute $NCA(D, v)$.
    - However, when $v$ is marked, we are able to determine $NCA(v, D)$.
• \( v \) is marked but not in \( D \)'s subtree, so \( \text{NCA}(D, v) \) is \( v \)'s anchor.
• \( v \) is in \( D \)'s subtree, so \( \text{NCA}(D, v) = D \).
  • Note that this is not a special case since \( v \)'s anchor is \( D \).

- How do we determine the anchor of any visited node?
  - After the recursive call returns, we call union.
  - For example, after the recursive call to \( D \) returns, all nodes in \( D \) have their anchor changed from \( D \) to \( C \) by merging the two classes.
  - This is seen in Figure 10.

![Figure 10: After recursive call to D returns](image)

- At any point, we can obtain the anchor for a vertex \( v \) by a call to find.

**The Quick-Find Algorithm**

- Two strategies for implementing the union/find data structure.
  - The first insures that the find instruction can be executed in constant worst-case time.
  - The second insures that the union operation can be executed in constant worse-case time.
  - It has been shown that both cannot be done simultaneously in constant worse-case (or even amortized) time.
- To implement the first case, suppose we maintain the equivalence class as an array: the index is the name of the node and the element is the equivalence class.
  - How long does it take to do find?
    • Constant, just an array lookup.
  - What about a union(a, b)?
    • Takes a scan of the array to change one class to another and this is linear.
    • If we have to do \( N - 1 \) of them, then it is quadratic.
  - The time for union is unacceptable.
- What if we keep all the elements that are in the same equivalence class in a linked list?
  - We save in the union operation but we lose with the find.
- In the next section, we look at a solution in which union is done in constant time, but the find is hard.
The Quick-Union Algorithm

- The find operation does not have to return any specific name.
  - Two finds on two elements return the same answer if and only if they are in the same set.
- Maybe we should use a tree, since each element in the tree has the same root.
- A tree represents each set.
  - A forest is a collection of trees.
  - The trees do not have to be binary.
    - They could be implemented using an array because the only information we need is the parent.
    - \( p[i] \) is the parent of element \( i \).
    - A \( -1 \) is used to indicate the parent is the root.
    - Figure 11 shows a forest and the array that represents it.

![Figure 11 Forest of eight elements](image)

- To perform a union of two sets, we merge the two trees by making the root of one tree a child of the root of the other.
- Consider Figures 12, 13, and 14 that show the forest after union(4, 5), union(6, 7), and union(4, 6) where the convention is adopted that the new root after union(\( x, y \)) is \( x \).

![Figure 12 Forest after the union of trees rooted at 4 and 5](image)
• A `find` operation on element `x` is performed by returning the root of the tree containing `x`.
  - How long does this take?
    - The number of nodes on the path from `x` to the root.

**Smart Union Algorithms**

- A simple improvement to the previous algorithm is to make the smaller tree a subtree of the larger, breaking ties by any method.
  - This is called `union-by-size`.
- Figure 15 shows the `union(3, 4)`.
  - What would have happened if the `union-by-size` had not been used?
    - A deeper forest would have been formed.
    - Would take more time for the `find` operation.
- Figure 16 shows the worst case tree possible after 15 `union` operations.
  - The tree is obtained by unioning trees of the same size.
• We need to maintain the size of each tree. This can be done as part of the array as seen in Figure 15.
• Another implementation is union-by-height in which we keep track of the height of the trees and perform union operations by making a shallower tree a subtree of the deeper tree.
  o This technique can be seen in Figure 17.

![Figure 16 Worst-case tree for N=16](image)

**Path Compression**

• We have a good algorithm, but sometimes find can be costly because the shape of the tree.
• Can we do something clever is decrease the time of the find operation?
• After we do a find on \( x \), changing \( x \)’s parent to the root would make sense.
  o However, we can also change the parents of all nodes on the path from \( x \) to the root on the access path.
  o This is called *path compression*.
  o Notice that this shortens the length of the path from the node accessed to the root.
• Figure 18 shows the effect of path compression after find(14) on the generic worst tree shown in Figure 16.
Path compression is compatible with union-by-size.
However, it is not compatible with union-by-height as there is no efficient way to change
the height of the tree.
  - No problem, we do not recompute the height.
  - Thus, the heights stored for each tree become estimated heights, called \textit{ranks}.
  - The resulting algorithm is called union-by-rank.
  - This algorithm gives an almost linear guarantee of running time for a sequence of
    $M$ operations.