Chapter 4

Union-Find for Disjoint Sets

Equivalence Relation
Relation $R$ on $S$ is a subset of $S \times S$.
- For every pair of elements $a, b$ from a set $S$, $a R b$ is either true or false.
- $a R b$ is true iff $(a, b)$ is in $R$. In this case, we say $a$ is related to $b$.

An equivalence relation satisfies:
1. (Reflexive) $a R a$
2. (Symmetric) $a R b$ iff $b R a$
3. (Transitive) $a R b$ and $b R c$ implies $a R c$

Equivalence Classes
- Given a set of things...
  {grapes, blackberries, plums, apples, oranges, peaches, raspberries, lemons, bananas}
- ...define the equivalence relation
  All citrus fruit is related, all berries, all stone fruits, ...
- ...partition them into related subsets
  {grapes}, {blackberries, raspberries}, {oranges, lemons}, {plums, peaches}, {apples}, {bananas}

Everything in an equivalence class is related to each other.

Determining equivalence classes
- Idea: give every equivalence class a name
  - {oranges, limes, lemons} = "like-ORANGES"
  - {peaches, plums} = "like-PEACHES"
  - Etc.
- To answer if two fruits are related:
  - FIND the class name of one fruit.
  - FIND the class name of the other fruit.
  - Are they the same name?

Building Equivalence Classes
- Start with disjoint, singleton sets:
  - {apples}, {bananas}, {peaches}, ...
- As you gain information about the equivalence relation, take UNION of sets that are now related:
  - {peaches, plums}, {limes, oranges, lemons}, {apples}, {bananas}, ...
- E.g. if peaches R limes, then we get
  - {peaches, plums, limes, oranges, lemons}

Disjoint Union - Find
- Maintain a set of pairwise disjoint sets.
  - (3,5,7), (4,2,8), (9), (1,6)
- Each set has a unique name, using one of its members
  - (3,5,7), (4,2,8), (9), (1,6)
Union

• Union(x,y) – take the union of two sets named x and y
  – {3, 5, 7}, {4, 2, 8}, {1, 6}
    Union(5, 1)
  – {3, 5, 7, 1, 6}, {4, 2, 8}, {9}.

Find

• Find(x) – return the name of the set containing x.
  – {3, 5, 7, 1, 6}, {4, 2, 8}, {9},
  – Find(1) = 5
  – Find(4) = 8

Example

\[
\begin{align*}
S: & \quad \{1, 2, 3, 8, 9, 13, 19\} \\
& \quad \{1\} \\
& \quad \{2\} \\
& \quad \{3\} \\
& \quad \{4\} \\
& \quad \{5\} \\
& \quad \{6\} \\
& \quad \{10\} \\
& \quad \{11, 17\} \\
& \quad \{12\} \\
& \quad \{14, 20, 26, 27\} \\
& \quad \{15, 16, 21\} \\
& \quad \{22, 23, 24, 29, 39, 32\} \\
& \quad \{33, 34, 35, 36\} \\
\end{align*}
\]

\[
\begin{align*}
S: & \quad \{1, 2, 7, 8, 9, 13, 19, 14, 20, 26, 27\} \\
& \quad \{1\} \\
& \quad \{2\} \\
& \quad \{3\} \\
& \quad \{4\} \\
& \quad \{5\} \\
& \quad \{6\} \\
& \quad \{10\} \\
& \quad \{11, 17\} \\
& \quad \{12\} \\
& \quad \{15, 16, 21\} \\
& \quad \{22, 23, 24, 29, 39, 32\} \\
& \quad \{33, 34, 35, 36\} \\
\end{align*}
\]

Implementing Disjoint Sets

• \(n\) elements
  Total Cost of: \(m\) finds, \(\leq n-1\) unions

• Target complexity: \(O(m+n)\) i.e. \(O(1)\) amortized per operation.

• \(O(1)\) worst-case for find as well as union would be great, but it’s simply not true.

• Known result: find and union can be done practically in \(O(1)\) time.

Implementing Disjoint Sets

• Observation: trees let us find many elements given one root...

• Idea: if we reverse the pointers (make them point up from child to parent), we can find a single root from many elements...

• Idea: Use one tree for each equivalence class. The name of the class is the tree root.

Up-Tree for Union/Find

Initial state

Intermediate state

Roots are the names of each set.
Find Operation
• Find(x) follow x to the root and return the root.
• Cost: O(h), h: height of the tree

Union Operation
• Union(i,j) - assuming i and j roots, point i to j.
• Cost: O(1)

Simple Implementation
• Array of indices

Union

void Union(int[] up, int x, int y) {
  //precondition: x and y are roots
  Up[x] = y;
}

Constant Time!

FIND
• Design Find operator
  – Recursive version
  – Iterative version

static int Find(int x) {
  //Pre: Up[0..(siz-1)] is the parent info;
  // x is in the range 0 to size-1
  if (Up[x] == "-" or "-1") return x;
  return Find(Up[x]);
}

A Bad Case

Find(1)  n steps!!
m finds: O(mn)
Now this doesn’t look good 😒

Can we do better? Yes!

1. Improve union so that find only takes $O(\log n)$
   - Union-by-size
   - Union-by-rank (height)
   - The cost of $m$ finds is $\Theta(m \log n)$

2. Improve find so that it becomes even better!
   - Path compression
   - Reduces complexity to almost $O(1)$ per operation

---

Union by Weight/Rank

- Union by weight
  - Always point the smaller tree to the root of the larger tree
- Union by rank (height)
  - Always point the shorter tree to the root of the higher tree

---

Elegant Array Implementation

- Union by Weight
- Union by Rank

---

Union by Weight

```c
void W_Union(int i,j){
    //Pre: i and j are roots/
    int wi = weight[i];
    int wj = weight[j];
    if (wi < wj) {
        Up[i] = j;
        weight[j] = wi + wj;
    } else {
        Up[j] = i;
        weight[i] = wi + wj;
    }
}
```

Computing time?

---

Union by Rank

```c
void R_Union(int i,j){
    //Pre: i and j are roots/
    int ri = rank[i];
    int rj = rank[j];
    if (ri < rj) {
        Up[i] = j;
    } else if (ri > rj) {
        Up[j] = i;
    } else { // ri == rj
        Up[j]++;
    }
}
```

Computing time?

---

Example Again

- Find(1) constant time
Analysis of Union by Weight/Rank

- With union by weight/rank an up-tree of height $h$ has weight at least $2^h$.
- Proof by induction on height
  - Basis: $h = 0$. The up-tree has one node, $2^0 = 1$
  - Inductive step: Assume true for all $h' < h$.

A tree $T$ of height $h$ must have a child $T_2$ of height $h-1$

$$W(T_1) + W(T_2) \geq 2^{h-1} + 2^{h-1} = 2^h$$

Analysis of Union by Weight/Rank

- Let $T$ be an up-tree of weight $n$ formed by union by weight/rank. Let $h$ be its height.
- $n \geq 2^h$ (just proved)
- $\log n \geq h$

- Find($x$) in tree $T$ takes $O(\log n)$ time.
- Can we do better?

Worst Case for Union by Weight/Rank

- n/2 Weighted Unions
- n/4 Weighted Unions

Example of Worst Cast (cont’)

- After $n - 1 = n/2 + n/4 + \ldots + 1$ Unions

A binomial tree

If there are $n = 2^h$ nodes then the longest path from leaf to root has length $k = \log_2(n)$.

Path Compression

- On a Find operation point all the nodes on the search path directly to the root.

Self-Adjustment Works

- PC-Find($x$)
Exercise: Draw the result of Find(e)

Path Compression Find

```c
int PC_Find(int i) {
    int r = i;
    while (Up[r] != -1) //find root
        r = Up[r];
    if (i != r) { //compress path//
        int k = Up[i];
        while (k != r) {
            Up[i] = r;
            i = k;
            k = Up[k];
        }
    }
    return r;
}
```

Function Definition

Ackermann’s function was defined in 1920s by German mathematician and logician Wilhelm Ackermann (1896–1962).

\[ A(m,n), \quad m,n \in \mathbb{N} \text{ such that,} \]

- \( A(0,n) = n + 1, \quad n \geq 0; \)
- \( A(m,0) = A(m-1, 1), \quad m > 0; \)
- \( A(m,n) = A(m-1, A(m, n-1)), \quad m, n > 0; \)

Example - 1

\[
\begin{align*}
A(1, 2) &= A(0, A(1, 1)) \\
        &= A(0, A(0, A(1, 0))) \\
        &= A(0, A(0, A(0, 1))) \\
        &= A(0, A(0, 2)) \\
        &= A(0, 3) \\
        &= 4
\end{align*}
\]

Simple addition and subtraction!!

Equivalent Definition

\[
\begin{align*}
A(0, n) &= n + 1 \\
A(1, n) &= 2 \times (n + 3) - 3 \\
A(2, n) &= 2 \times (n + 3) - 3 \\
A(3, n) &= 2^{2^{2^{2^{\ldots^2}}}} - 3 \\
A(4, n) &= 2^{2^{2^{2^{\ldots^2}}}} - 3 \\
&\quad (n + 3 \text{ terms})
\end{align*}
\]

Terms of the form \(2^{2^{2^{\ldots^2}}}\) are known as power towers.

It is a well defined total function that grows so fast.

Inverse of Ackermann’s Function

\[
\alpha(m, n) = \min \{ i \geq 1 : A( i, \lfloor m / n \rfloor ) > \{ n \} \}
\]

\(\alpha(x, y)\) is a really slowly growing function.

How slow does \(\alpha(x, y)\) grow?

\(\alpha(x, y) = 4\) for \(x\) far larger than the number of atoms in the universe (\(2^{300}\))

\(\alpha\) shows up in:

- Computation Geometry (surface complexity)
- Combinatorics of sequences
Disjoint Union / Find with Union by Weight/Rank and Path Compression

- Worst case time complexity for a W-Union/R-Union is O(1) and for a PC-Find is O(log n).
- Time complexity for m ≥ n operations on n elements is O(m α(m, n))
  - α(m, n) ≤ 4 for all reasonable n. Essentially constant time per operation!

Amortized Complexity

- For disjoint union / find with union by weight/rank and path compression.
  - amortized time per operation is essentially a constant.
  - worst case time for a PC-Find is O(log n).
- An individual operation can be costly, but over time the average cost per operation is not.

Cute Application

- Build a random maze by erasing edges.

Cute Application

- Pick Start and End

Desired Properties

- None of the boundary is deleted
- Every cell is reachable from every other cell.
- There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.
A Cycle, not allowed

A Good Solution

A Hidden Tree

Number the Cells
We have disjoint sets \( S = \{ \{1\}, \{2\}, \{3\}, \{4\}, \ldots, \{36\} \} \) each cell is a singleton set.
We have all possible edges \( E = \{(1,2), (1,7), (2,8), (2,3), \ldots \} \) 60 edges total, representing the neighborhood relation.

Number the Cells

<table>
<thead>
<tr>
<th>Start</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

We have disjoint sets \( S = \{ \{1\}, \{2\}, \{3\}, \{4\}, \ldots, \{36\} \} \) each cell is a singleton set.
We have all possible edges \( E = \{(1,2), (1,7), (2,8), (2,3), \ldots \} \) 60 edges total, representing the neighborhood relation.

Basic Algorithm

- \( S \) = set of sets of connected cells
- Initially \( S = \{ \{1\}, \{2\}, \ldots, \{n^2\} \} \)
- \( E \) = set of edges, representing the neighborhood of each cell.

Basic Algorithm

```
Alg. CreateMaze (S, E) {
    while (|S| > 1) {
        pick a random, unused edge \((x,y)\) from \( E \);
        u = Find(x);
        v = Find(y);
        if \((u \neq v)\) \{ Union(u,v); remove \((x, y)\) from \( E \) \}
        else mark \((x, y)\) as “used”;
    }
    return \( E \);
} // All remaining members of \( E \) form the maze.
```

Example Step
Pick \((8,14)\)

Example Step

```
Start

\| | 1 | 2 | 3 | 4 | 5 | 6 |
---|---|---|---|---|---|---|
7 | 8 | 9 | 10 | 11 | 12 |
13 | 14 | 15 | 16 | 17 | 18 |
19 | 20 | 21 | 22 | 23 | 24 |
25 | 26 | 27 | 28 | 29 | 30 |
31 | 32 | 33 | 34 | 35 | 36 |
\| \|\|\|\|\|\|

\| S \|\|\|\|\|\|\|
---|---|---|---|---|---|---|
\{1,2,7,8,9,13,19\} | \{3\} | \{4\} | \{5\} | \{6\} | \{10\} | \{11,17\} |
\{12\} | \{14,20,26,27\} | \{15,16,21\} | \{\} | \{22,23,24,29,30,32\} | \{33,34,35,36\} |
```
Example

$S = \{1,2,7,8,9,13,19,14,20,26,27\}$

Find(8) = 7
Find(14) = 20

Union(7,20)

S
\{1,2,7,8,9,13,19,14,20,26,27\}
\{3\}
\{4\}
\{5\}
\{10\}
\{11,17\}
\{12\}
\{14,20,26,27\}
\{15,16,21\}
\{22,23,24,29,39,32\}
\{33,34,35,36\}

Example

S
\{1,2,7,8,9,13,19,14,20,26,27\}
\{3\}
\{4\}
\{5\}
\{10\}
\{11,17\}
\{12\}
\{14,20,26,27\}
\{15,16,21\}
\{22,23,24,29,39,32\}
\{33,34,35,36\}

Pick (19,20)

Start
1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
31 32 33 34 35 36

Example at the End

S
\{1,2,3,4,5,6,\ldots,36\}

Start
1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
31 32 33 34 35 36