**Data Structures for Disjoint Sets**

Advanced Data Structures and Algorithms  
(CLRS, Chapter 21)  
James Worrell  
Oxford University Computing Laboratory, UK  

HT 2010

---

**Disjoint-set data structures**

- Also known as *union find*
- Maintain collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of disjoint sets that change over time
- Each set is identified by a *representative*:
  - It doesn’t matter which member of a set is the representative, but we require that if we ask for the representative twice without modifying the set, we get the same answer both times
- Simple data structure, but a classic application of aggregate analysis
**Operations**

- **MAKE-SET**(x): make a new set $S_i = \{x\}$ and add $S_i$ to $S$
- **UNION**(x, y): if $x \in S_i$ and $y \in S_j$, then
  
  $S \leftarrow S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$

  - Representative of $S_i \cup S_j$ is typically the representative of $S_i$ or $S_j$
- **FIND**(x): returns the representative of the set containing $x$
- Analyse complexity of sequence of $m$ MAKE-SET, FIND and UNION operations, $n$ of which are MAKE-SET operations
- Complexity is analysed in terms of $n$ and $m$

---

**Application 1: Kruskal’s Algorithm**

MST-KRUSKAL($G, w$)

1. $A \leftarrow \emptyset$
2. for each vertex $v \in V[G]$
3.     do MAKE-SET($v$)
4. sort the edges of $E$ into nondecreasing order by weight $w$
5. for each edge $(u, v) \in E$, taken in nondecreasing order by weight
6.     do if FIND($u$) $\neq$ FIND($v$)
7.         then $A \leftarrow A \cup \{(u, v)\}$
8.             UNION($u, v$)
9. return $A$
Disjoint-set forests

- Represent each set as a rooted tree
  - Each node has a pointer to its parent (root points to itself)
  - Root node represents a given tree
- The family of disjoint sets is therefore a forest
- Operations:
  - \texttt{MAKE-SET}(x): make a tree with single node \textit{x}
  - \texttt{FIND}(x): find the root of \textit{x}'s tree
  - \texttt{UNION}(x, y): join the roots of \textit{x} and \textit{y}'s respective trees
Union-by-rank and path compression

Assume each node $x$ has an attribute $\text{rank}(x)$. For now think of the rank of a node as its height (more about this later)

**Union by rank:** When linking two trees, make root with smaller rank a child of the root with larger rank

Define the *find path* corresponding to node $x$ to be the path of nodes linking $x$ to the root of $x$’s tree

**Path compression:** When executing $\text{FIND}(x)$, make all nodes on the find path of $x$ direct children of the root

(Due to path compression means the rank of a node is only an upper bound on its height.)

The above two heuristics are very simple, but their analysis is subtle!

---

**Operations**

- Make a one-node tree:

  $$\text{MAKE-SET}(x)$$

  $p[x] \leftarrow x$

  $\text{rank}[x] \leftarrow 0$

- $\text{FIND}$ works in *two passes*—the first pass traces a find path to the root and in the second pass, as the recursive calls unwind, all nodes in the find path are updated to point directly to the root

  $$\text{FIND}(x)$$

  if $x \neq p[x]$

  then $p[x] \leftarrow \text{FIND}(p[x])$

  return $p[x]$
Operations

- We can define union in terms of a procedure \( \text{LINK}(x, y) \), which joins the trees rooted at \( x \) and \( y \)—taking \( O(1) \) time

\[
\text{LINK}(x, y) \\
\text{if } \text{rank}[x] > \text{rank}[y] \\
\quad \text{then } p[y] \leftarrow x \\
\text{else } p[x] \leftarrow y \\
\quad \text{if } \text{rank}[x] = \text{rank}[y] \\
\quad \quad \text{then } \text{rank}[y] \leftarrow \text{rank}[y] + 1
\]

- Find the roots and link them

\[
\text{UNION}(x, y) \\
\text{LINK}(\text{Find}(x), \text{Find}(y))
\]

Properties

Assuming only union-by-rank the following three properties hold:

Property 1. For any non-root node \( x \), \( \text{rank}(x) < \text{rank}(p(x)) \).

Property 2. The sub-tree rooted at a rank-\( k \) node has at least \( 2^k \) nodes.

Since a node of rank \( k \) is created by linking two nodes of rank \( k - 1 \), Property 2 holds by induction on \( k \). Properties 1 and 2 entail

Property 3. The number of nodes of rank \( k \) is at most \( n/2^k \).

If we additionally assume path compression, then Property 3 still holds since removing path compression from a sequence of \( \text{LINK} \) and \( \text{Find} \) operations does not affect the rank of any nodes.\(^a\)

\(^a\)However Property 2, which was used to establish Property 3, no longer holds!
Tetration

- Define $2 \uparrow n$ by $2 \uparrow 0 = 1$ and $2 \uparrow (n + 1) = 2^{2 \uparrow n}$

$$
\begin{align*}
2 \uparrow 0 &= 1 \\
2 \uparrow 1 &= 2^1 = 2 \\
2 \uparrow 2 &= 2^2 = 4 \\
2 \uparrow 3 &= 2^4 = 16 \\
2 \uparrow 4 &= 2^{16} = 65536 \\
2 \uparrow 5 &= 2^{65536} \gg 10^{80}
\end{align*}
$$

- Note that $10^{80}$ is the estimated number of atoms in the observable universe!

- Define $\lg^* n = \max\{k : 2 \uparrow k \leq n\}$ for $n \geq 1$. For all practical purposes we can assume that $\lg^* n \leq 5$

Complexity

**Theorem.** A sequence of $m$ UNION and FIND operations starting with $n$ singletons takes time $O(m \lg^* n)$ in the worst case

*To all intents and purposes the complexity is linear in $m$.*

**Proof.** Since $\text{UNION}(x, y) = \text{LINK}(\text{FIND}(x), \text{FIND}(y))$, it suffices to prove that a sequence $m$ LINK and FIND operations on $n$ singletons takes time $O(m \lg^* n)$.

Now LINK is a constant-time operation so we only have to bound the complexity of the FIND operations.

As a first step, note that the complexity of FIND($x$) is proportional to the length of the find path from $x$ to the root of $x$’s tree.
Partition node ranks into blocks $B_0, B_1, B_2, \ldots$, where

$$B_k = \begin{cases} 
\{1\} & \text{if } k = 0 \\
\{r : 2 \uparrow (k - 1) < r \leq 2 \uparrow k\} & \text{if } k > 0.
\end{cases}$$

<table>
<thead>
<tr>
<th>$k$</th>
<th>$B_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${1}$</td>
</tr>
<tr>
<td>1</td>
<td>${2}$</td>
</tr>
<tr>
<td>2</td>
<td>${3, 4}$</td>
</tr>
<tr>
<td>3</td>
<td>${5, 6, \ldots, 16}$</td>
</tr>
<tr>
<td>4</td>
<td>${17, 65536}$</td>
</tr>
<tr>
<td>5</td>
<td>${65537, 2^{65536}}$</td>
</tr>
</tbody>
</table>

**Key property:** there are exponentially fewer nodes in each successive block.

Let $N_k$ denote the number of nodes whose final rank lies in block $B_k$.

**Lemma.** $N_k \leq \frac{n}{2 \uparrow k}$.

**Proof.** We use Property 3 above:

$$N_k \leq \sum_{j \in B_k} \frac{n}{2^j} < \sum_{j > 2 \uparrow (k-1)} \frac{n}{2^j} = \frac{n}{2 \uparrow k}$$
• Execution of \textsc{Find-Set}(x) traversing find path \(x_0, x_1, \ldots, x_{l-1}\) is charged \(l\) units

• Divide charges into:
  – **Block charges**: \(p(x_i)\) is the root, or \(\text{rank}(x_i)\) and \(\text{rank}(x_{i+1})\) are in different blocks
  – **Path charges**: every node in find path not given a block charge

\begin{center}
\includegraphics[width=0.8\textwidth]{diagram.png}
\end{center}

**Completing the Proof**

• **Block charges**:
  – There are at most \(\lg^* n\) blocks. Therefore at most \(\lg^* n + 1\) block charges per \textsc{Find} operation
  – At most \(m(\lg^* n + 1) = O(m \lg^* n)\) block charges in total

• **Path charges**:
  – Each time a node is assessed a path charge it gets a new parent of strictly higher rank
  – Therefore at most \(|B_k| \leq 2 \uparrow k\) path charges on a given \(B_k\)-node
  – Summing over all \(B_k\)-nodes, the total number of path charges across all \(B_k\)-nodes is at most \(N_k \cdot 2 \uparrow k \leq n\)
  – Summing over all blocks, the total number of path charges across all nodes is at most \(n \lg^* n\)
A deterministic finite automaton \( A = (\Sigma, S, \delta, F) \) consists of:

- An alphabet \( \Sigma \)
- Set of states \( S \)
- Set of accepting states \( F \subseteq S \)
- A transition function \( \delta : S \times \Sigma \rightarrow S \)

---

\( A \) consists of:

- An alphabet \( \Sigma \)
- Set of states \( S \)
- Set of accepting states \( F \subseteq S \)
- A transition function \( \delta : S \times \Sigma \rightarrow S \)

---

Application 2: Equivalence of DFA

- Given a DFA \( A = (\Sigma, S, \delta, F) \) and a state \( s_0 \in S \), let \( L(A_{s_0}) \) denote the set of all words accepted by \( A \) from initial state \( s_0 \).
- For example, the automaton on the previous slide accepts the language \( 0^*1^+0^+1(0 + 1)^* \) from initial state \( s_0 \).
- The language equivalence problem asks, given DFA \( A \) and \( B \), and initial states \( s_0 \) for \( A \) and \( t_0 \) for \( B \), does \( L(A_{s_0}) = L(B_{s_0}) \)?
- Equivalently we can ask, given a single DFA \( A = (\Sigma, S, \delta, F) \) and two states \( s_0, t_0 \in S \), does \( L(A_{s_0}) = L(A_{t_0}) \)?
Application 2: Equivalence of DFA

- Given $A = (\Sigma, S, \delta, F)$ and $s_0, t_0 \in S$ does $L(A_{s_0}) = L(A_{t_0})$?

- Extend $\delta$ to a function $\delta : S \times \Sigma^* \to S$ by
  $$\delta(s, \varepsilon) = s \text{ and } \delta(s, x.a) = \delta(\delta(s, x), a)$$
  for all $x \in \Sigma^*$ and $a \in \Sigma$.

- Consider the relation $\text{Reach} \overset{\text{def}}{=} \{(\delta(s_0, x), \delta(t_0, x)) : x \in \Sigma^*\}$ and observe that $L(A_{s_0}) = L(A_{t_0})$ iff for all $(s, t) \in \text{Reach}$ it holds that $s \in F$ iff $t \in F$.

- Idea for algorithm: compute the equivalence closure of $\text{Reach}$, denoted $\text{Reach}^a$

  This is the smallest equivalence relation on $S$ that contains Reach.

---

Equivalence of DFA

DFA-EQUIV$(A, s_0, t_0)$

1. for $s \in S$ do \text{MAKE-SET}(s)
2. Waiting $\leftarrow \{(s_0, t_0)\}$
3. while Waiting $\neq \emptyset$
   4. do choose and remove $(s, t)$ from Waiting
   5. if $(s \in F)$ xor $(t \in F)$ then return \text{FALSE}
   6. if $\text{FIND}(s) \neq \text{FIND}(t)$ then \text{UNION}(s, t)
   7. for $a \in \Sigma$ do add $(\delta(s, a), \delta(t, a))$ to Waiting
4. return \text{TRUE}
### Complexity and Correctness

**Complexity.** The running time of DFA-EQUIV is \(O(\lg^* (|S|)|\Sigma||S|)\) (See Exercise Sheet 3.)

**Correctness.** Let \(\Pi = \{(s, t) \in S \times S : \text{FIND}(s) = \text{FIND}(t)\}\). Then the following loop invariant holds in DFA-EQUIV:

(i) \((s_0, t_0) \in \Pi \cup \text{Waiting}\)

(ii) \(\text{Waiting} \subseteq \text{Reach}\)

(iii) \((s, t) \in \Pi \Rightarrow s \in F \text{ iff } t \in F\)

(iv) \((s, t) \in \Pi \Rightarrow (\delta(s, a), \delta(t, a)) \in \Pi \cup \text{Waiting} \text{ for all } a \in \Sigma\)

The correctness of DFA-EQUIV can be established with reference to the above invariant (See Exercise Sheet 3.)

### Application 3: Least Common Ancestors

The least common ancestor of two nodes \(u\) and \(v\) in a rooted tree \(T\) is the node \(w\) of least height that is an ancestor of both \(u\) and \(v\) (see the picture below).

In the off-line\(^4\) least common ancestors problem we are given a rooted tree \(T\) and a set \(P\) of pairs of nodes in \(T\), and we wish to determine the least common ancestor of each pair in \(T\).

Applications include:

- Biology
- Compilers and static analysis
- VLSI testing

\(^4\)This problem is said to be off-line because the whole set \(P\) is given in advance of any output being required.
Off-line LCA

Idea for algorithm: adapt depth-first search.

- Recall that in dfs all nodes start white, open nodes are coloured grey and closed nodes are coloured black. Grey nodes reside in a stack $S$.
- For each grey node $w$ maintain a set $D_w$ consisting of $w$ and all black nodes whose least grey ancestor is $w$.
- When we finish a node $u$, for all pairs $(u, v) \in P$ such that $v$ is black and $v \in D_w$ for some grey node $w$, $w$ is the least common ancestor of $u$ and $v$.

Least Common Ancestors

$b = \text{lca}(a, c)$
Initial call is to LCA(root[T]):

LCA(u)
1  MAKE-SET(u) ▷ create a new set when we push u on dfs stack
2  colour[u] ← grey ; ancestor[u] ← u
3  for each child v of u
do
4      LCA(v)
5    UNION(u, v) ▷ union u and v’s sets when we pop v from dfs stack
6    ancestor[FIND(u)] ← u ▷ representative of u’s set has pointer to u
7  colour(u) ← black
8  for each node v such that \{u, v\} ∈ P
do
9      if colour[v] = BLACK
10         then print ‘the lca of’ u ‘and’ v ‘is’ ancestor[FIND(v)]