COMP251: Disjoint sets

Jérôme Waldispühl
School of Computer Science
McGill University

Based on slides from M. Langer (McGill)
Problem

Let $G=(V,E)$ be undirected graph, and $A, B \in V$ two nodes of $G$.

Question: Is there a path between $A$ and $B$?

But we are not interested in knowing the path between $A$ and $B$.

Is there a faster way to solve this problem (faster than DFS or BFS)?
Connected components

Connected component: Set of nodes connected by a path.

Question: Given 2 nodes A & B, are they in the same component?
Partition

Generalization: Set of object partitioned into disjoint subsets.

\[ S = S_1 \cup S_2 \cup \ldots \cup S_n \]

\[ S_i \neq \emptyset \; \forall \; i \in \{1, \ldots, n\} \]

\[ S_i \cap S_j = \emptyset \; \text{iff} \; i \neq j \]
Motivations

• Data structure used to manage sets and perform classical operations (union, find, intersection...)

• Used in many algorithms (e.g. Kruskal, Floyd-Marshall)
Map or function \( f: S \rightarrow \{ (a,f(a)) \} \)

Relation \( R \subseteq \{ (a,b) : a,b \in S \} \)

Any Boolean matrix defines a relation
Equivalence relation

i is equivalent to j if they belong to the same set.

(more constrained than a general relation)
Equivalence relation

- Reflexivity \( \forall a \in S, (a,a) \in R \)
- Symmetry \( \forall a, b \in S, (a,b) \in R \implies (b,a) \in R \)
- Transitivity \( \forall a, b, c \in S, (a,b) \in R \text{ and } (b,c) \in R \implies (a,c) \in R \)

Example:

For any undirected graph, the connections define an equivalence relation on vertices.
- For all \( u \in V \), there is a path of length 0 from \( u \) to \( u \).
- For all \( u,v \in V \), there is a path from \( u \) to \( v \), iff there is a path from \( v \) to \( u \).
- For all \( u,v,w \in V \), if there is a path from \( u \) to \( v \) and a path from \( v \) to \( w \), then there is a path from \( u \) to \( w \).
Disjoint set ADT

Each set in the partition as a representative member.

- `find(i)` returns the representative of the set that contains `i`.
- `sameset(i,j)` returns the boolean value `find(i)==find(j)`.
- `union(i,j)` merges the sets containing `i` and `j`.
Union of disjoint sets

union(i,j) merges the sets containing i and j.

- Does nothing if i and j are already in the same set.
- Otherwise, we merge the set and need a policy to decide who will be the representative of the new merged set.
Let $\text{Rep}[i] \in \{1, 2, \ldots, n\}$ be the representative of the set containing $i$. 

<table>
<thead>
<tr>
<th>Rep[]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
(Quick find) & union

- `find(i) { return rep[i]; }`
- `union(i,j)` merges the sets containing `i` and `j`.

Example: `union(2,6)`
(Quick find) & union

union(i,j) {
    if rep[i] != rep[j] {
        prevrepi = rep[i];
        for (k=1; k<=n; k++) {
            if rep[k] == prevrepi {
                rep[k] = rep[j];
            }
        }
    }
}

• store value of rep[i] because it may change during the execution of the algorithm.
• O(n) running time... slow.
Tree representation & forests

- Represent the disjoint sets by a forest of rooted trees.
- Roots are the representative (i.e. find(i) == findroot(i)).
- Each node points to its parent.

Note: The tree structure does not necessarily represent the relationship between the stored objects.
Array representation

<table>
<thead>
<tr>
<th>p[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

- Non-root nodes hold index of their parent.
- Root nodes store their own value.
Find & Union

find(i) {
    if p[i] == i {
        return i;
    } else {
        return find(p[i]);
    }
}

union(i,j) {
    if find(i) != find(j) {
        p[find(i)] = find(j);
    }
}

Remark: Arbitrarily merge the set on i into the set of j.
Root of the tree of 11 becomes the parent of the root of the tree of 9.
Worst case

union(1,2)
union(1,3)
union(1,4)
...
union(1,n)

Then, find(1) is O(n)...
Union by size

Heuristic to control the height to the trees after merging.

**Idea:** Merge tree with smaller number of nodes into the tree with the largest number of nodes (In practice, we can also use rank which is an upper bound on the height of nodes).
**Claim:** The depth of any node is at most log n.

**Proof:**

- If union causes the depth of a node to increase, then this node must belong to the smallest tree.

- Thus, the size of the (merged) tree containing this node will at least double.

- But we can double the size of a tree at most log n times.
Union by height

**Idea:** Merge tree with smaller height into tree with larger height.

**Claim:** The height of trees obtained by union-by-height is at most \( \log n \).

**Corollary:** An union-by-height tree of height \( h \) has at least \( n_h \geq 2^h \) nodes.

**Proof (Corollary):**
- Base case: a tree of height 0 has one node.
- Induction: (hypothesis) \( n_h \geq 2^h \). Show \( n_{h+1} \geq 2^{h+1} \).
Running time

Quick Union

- Quick find
  - Union by size: $O(\log n)$
  - Union by height: $O(\log n)$

<table>
<thead>
<tr>
<th>find(i)</th>
<th>union(i,j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Quick union makes 2 calls to find.

Note: These are worst case complexities.
Path compression

- Find path = nodes visited during the execution of find() on the trip to the root.
- Make all nodes on the find path direct children of root.
Path compression

```cpp
find(i) {
    if p[i] == i {
        return i;
    } else {
        p[i] = find(p[i]);
        return p[i];
    }
}
```
Running time

- Use union by size and path compression.
- Worst case running time is $O(\log n)$.
- However, we can show that $m$ union or find operations take $O(m \alpha(n))$.

What is $\alpha(n)$?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\alpha(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4 - 7</td>
<td>2</td>
</tr>
<tr>
<td>8 - 2047</td>
<td>3</td>
</tr>
<tr>
<td>$2048 - A_4(1)$</td>
<td>4</td>
</tr>
</tbody>
</table>

Where $A_4(1) \gg 10^{80}!!$