COMP 3170 - Analysis of Algorithms & Data Structures

Shahin Kamali

Disjoin Sets and Union-Find Structures

CLRS 21.121.4

University of Manitoba
Disjoint Sets

Disjoint set is an abstract data type for maintaining a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint, non-empty sets.

- **Disjoint:** there is no common element between any two sets (if $a$ is in $S_i$ it cannot be in $S_j$ where $i \neq j$).
- **Dynamic:** sets can be modified by make-set and union operations
- Each set is identified by a representative element of the set.

$k = 4; \quad S_a = \{a, b, m, n\}, S_c = \{c, g, h\}, S_e = \{d, e, f\}, S_q = \{q\}$
Disjoint Sets Operations

- **makeSet(x):**
  - Create a new set \{x\} whose only element is x.
  - By property 1 above, x cannot be an element of any other set.
  - By default, x is the representative of the new set.

E.g., **makeSet({p})**

\[ k = 4; \quad S_a = \{a, b, m, n\}, \quad S_c = \{c, g, h\}, \quad S_e = \{d, e, f\}, \quad S_q = \{q\} \]

\[ S_p = \{p\} \]
Disjoint Sets Operations

- **find(x)** (also called Find-Set(x)):
  - Return the representative element of the set containing x.

  E.g., \( \text{find}(b) \rightarrow a \)
  E.g., \( \text{find}(c) \rightarrow c \)

\[ k = 4; \quad S_a = \{a, b, m, n\}, \quad S_c = \{c, g, h\}, \quad S_e = \{d, e, f\}, \quad S_q = \{q\}, \]
Disjoint Sets Operations

- **union**($x, y$):
  - Unite the sets containing $x$ and $y$.
  - Suppose set $S_x$ contains $x$ and set $S_y$ contains $y$.
  - $S \leftarrow S \cup \{S_x \cup S_y\} - S_x - S_y$
  - Assign a representative for $x \cup y$.
  - union($x, y$) is equivalent to union(find($x$), find($y$)).

E.g., Union($b, d$) $\rightarrow$ merge $S_a$ and $S_e$.

$k = 4; \quad S_a = \{a, b, m, n\}, S_c = \{c, g, h\}, S_e = \{d, e, f\}, S_q = \{q\}$,

$\rightarrow \quad S_c = \{c, g, h\}, S_q = \{q\}, S_a = \{a, b, m, n, d, e, f\}$
Disjoint Sets Operations

- **makeSet(x):**
  - Create a new set \{x\} whose only element is x.
  - By default, x is the representative of the new set.

- **find(x)** (also called Find-Set(x):)
  - Return the representative element of the set containing x.

- **union(x, y):**
  - Unite the sets containing x and y.
  - Assign a representative for \(x \cup y\).
  - \(union(x, y)\) is equivalent to \(union(find(x), find(y))\).
Applications of Disjoint Sets

- Many applications in designing algorithms
- E.g., Kruskal’s minimum spanning tree
Kruskam’s MST algorithm

- Sort edges by their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - $e$ does not form a cycle iff its endpoints are in different components
Disjoint Sets Review

- **Disjoint set** is an abstract data type for maintaining a set of disjoint sets
  - make-set(x): create a new set with a single item x (which is not in any of the existing sets).
  - find(x): returns the representative item of the set that includes x.
  - union(x,y): removes the sets in which x and y belong to and adds a new set which is the union of deleted sets

- Disjoint sets have many applications in design of algorithms (e.g., Kruskal’s MST algorithm)
Data Structures for Disjoint Sets

- Linked lists for disjoint sets:
  - Each set is stored as a linked-list.
  - In a ‘set object’, store head/tail pointers to the first/last elements.
  - Each node stores a set pointer to the set object.
  - The representative element is the first element in the list.

\[ S_1 = \{ x, p \} \]

\[ S_2 = \{ a, h, c \} \]
Linked lists for disjoint sets

- **makeSet(x):**
  - Create a list containing one node.
  - takes $O(1)$
  - $O(1)$ time

```plaintext
makeSet(q)
```

![Diagram showing the process of makeSet(x) and makeSet(q) with two sets $S_1$ and $S_2$. $S_1$ starts with one node and $S_2$ starts with three nodes. After the processes, $S_1$ contains only one node, and $S_2$ is empty.]

$S_1 = \{x, p\}$

$S_2 = \{a, h, c\}$

$S_1 = \{q\}$
Linked lists for disjoint sets

- **find(x):**
  - follow the set-pointer to find the set object and get the representative element.
  - We assume we’re given a reference to \( x \).
  - It takes \( O(1) \) time

\[ \text{find}(h) \rightarrow a \]
Linked lists for disjoint sets

- **union(x,y):**
  - Append y’s list to the end of x’s list.
  - find(x) becomes the representative of the new set.
  - Use head pointer from x’s list and tail pointer from y’s list.
  - Requires updating the **set pointer** for each node in y’s list, i.e., \( \Theta(n) \) time per operation in the worst case (when y has size \( \Theta(n) \)).
  - What is the **amortized cost** of performing \( n - 1 \) union operations?

**union(p,h)**
Review of Amortized Analysis

- Amortized analysis considers the average cost per operation for a sequence of $m$ operations.
- In our previous examples, there is only one possible sequence of $m$ operations. E.g., $m$ increments and $m$ insertions to a dynamic array.
- In many data structures, there are many different sequences of operations.
  - We often consider the **worst-case amortized time**, i.e., the average cost of an operation for the worst-case sequence.
  - Sometimes people look at expected amortized time which considers the average cost for a random sequence (we do not talk about it in this course).
Linked lists for disjoint sets

- What is the amortized cost of performing $n - 1$ union operations?
- The following example is a worst-case sequence which provides a lower bound.
  - `makeSet(x_i)` for $i \in \{1, 2 \ldots, n\}$
  - `union(x_i, x_1)` for $i \in \{2, \ldots n\}$, that is:
    - `union(x_2, x_1)`: update 1 set-pointers
    - `union(x_3, x_1)`: update 2 set-pointers
    - \ldots
    - `union(x_i, x_1)`: at this point $x_1$ has $i$ items $\rightarrow$ update $i$ set-pointers
    - \ldots
    - `union(x_n, x_i)`: updated $n - 1$ set-pointers
  - Total set-pointer updates: $1 + 2 + 3 + \ldots + n - 1 \in \Omega(n^2)$.
    - Amortized number of updates is $\Omega(n)$.
    - This is a worst-case amortized time, e.g., for a sequence of $m$ operations formed by $m$ make-sets, the amortized cost is constant.
  - **If we simply append the second list to the end of the first list, the (worst-case) amortized time for union is $\Theta(n)$**.
Linked lists & Union by Weight

- What if we append the smallest list to the end of the larger list?
- In the set object, in addition to head and tail pointers, maintain a **weight** field which indicates the number of items in that list (set).
  - Make-set and find are as before, i.e., they take constant time per operation
  - For union, we compare the weights and append the smaller list to the end of the larger list

![Diagram of set operations]

\[
S_1 = \{x, p\} \\
S_2 = \{a, h, c\} \\
S_3 = \{x, p, a, h, c\}
\]
Consider a single node \( u \) of the list. We count the number of times the set-pointer is updated for that node.

Each time the pointer of \( u \) is updated, that means that the set of \( u \) is merged with a larger set.

- The weight of the set of \( u \) is at least doubled after the merge.

If there are \( n \) items in all sets, the weight of each set is at most \( n \).

- Each update for set-pointer of \( u \) doubles the weight of its list, and this weight cannot be more than \( n \).
- Hence, there are at most \( \lfloor \log n \rfloor \) set-pointer updates per item, i.e., a total of \( O(n \log n) \) set-pointer updates.
There are at most \(\lceil \log n \rceil\) set-pointer updates per item, i.e., a total of \(O(n \log n)\) set-pointer updates.

Ignoring the cost of set-pointer updates, the cost of union for other pointer updates and also find and make-set are constants \(\rightarrow \Theta(m)\) cost for \(m\) operation

Union by Weight has a cost of \(O(n \log n + m)\) for a sequence of \(m\) operations on a universe of size \(n\)

- The amortized cost per operation is \(O(1 + n \log n/m) = O(\log n)\)
- Note that \(m \geq n\) since we need \(m\) operations to make a universe of size \(n\).

Union by weight (appending smaller list to the end of larger one) improves the amortized time complexity from \(\Theta(n)\) to \(O(\log n)\).

- In your next assignment, you will see this bound is tight, i.e., the amortized cost is \(\Theta(\log n)\).
Review of Linked lists & Union by Weight

- Each set is represented by a linked list
  - Each node has a set-pointer to the set object, which makes find(x) run in constant time
- For union(x,y), we append one list to the end of another
  - This requires updating all set pointers of the appended list
- If we append the smaller list to the end of the larger list, each operation takes amortized time of $\Theta(\log n)$ in the worst case.

**Theorem**

Union-by-weight for linked list results in amortized cost of $\Theta(\log n)$ per operation for a disjoint set.
Disjoint Set Forests

- A data structure for disjoint sets which is based on trees instead of lists.
  - Each set is stored as a rooted tree
  - Each node points to its parent
  - The root points to itself
  - The representative element is the root

\[ S_1 = \{x, p\} \quad S_2 = \{a, h, c, f\} \]
Disjoint Set Forests

- **MakeSet(x)** takes $O(1)$ time:
  - Create a new tree containing one node $x$
  - $\text{parent}(x) \rightarrow x$

- **Find(x):**
  - Follow parent pointers to the root and return it.
    - $y \leftarrow x$
    - while $y \neq \text{parent}(y)$
      - $y \leftarrow \text{parent}(y)$
      - return $y$
  - time proportional to the tree’s height

$$S_1 = \{x, p\} \quad S_2 = \{a, h, c, f\}$$

![Diagram showing disjoint set forests with sets $S_1$ and $S_2$]
Disjoint Set Forests

- **Union(x,y) (first approach):**
  - Set root of y’s tree to point to the root of x’s tree.
    - \( \text{root}_x \leftarrow \text{find}(x) \)
    - \( \text{root}_y \leftarrow \text{find}(y) \)
    - \( \text{parent} \left( \text{root}_y \right) \leftarrow \text{root}_x \).
  - Time is proportional to tree’s height

- Tree’s height can be \( \Theta(n) \) for a universe of size \( n \)
  - In the worst case, each operation takes \( \Theta(n) \).
Amortized cost of first approach

- What is the amortized cost when performing $m$ operations?
  - If we simply make the second tree point to the first one, it can \( \Theta(n) \) in the worst case:
  - consider the following worst-case sequence of operations:
    - make-set\((x_i)\) for \( i \in \{1, \ldots, n\} \)
    - union\((x_i, x_1)\) for \( i \in \{2, \ldots, n\} \).
  - after the \( i \)'th union, set of \( x_1 \) is a tree of height \( i \).
  - the total time for the \( 2n - 1 \) operations is \( \sum_{i=1}^{n-1} i = n(n - 1)/2 \), i.e., the amortized cost is \( \Theta(n) \).
  - after forming this bad tree, the worst-case sequence of operations continues with \( m - 2n + 1 \) find\((x)\) operation where \( x \) is the only leaf of the tree.

Observation

*Having the second tree point to the first one for union results in the worst-case trees of height \( n \) and amortized time of \( \Theta(n) \) for each operation.*
Reducing the Height of Trees

- Two strategies for bounding tree heights:
  - union by rank
  - path compression
Union by Rank

- Always attach the shorter tree to the root of the taller one
  - Similar to union-by-weight on lists
- Maintain the **rank** as an upper bound for the height of each tree.
  - The rank increased when both trees have the same rank

\[
\text{root}_x \leftarrow \text{find}(x); \quad \text{root}_y \leftarrow \text{find}(y)
\]

\[
\text{if rank(root}_x) > \text{rank(root}_y) \quad \text{parent(root}_y) \leftarrow \text{root}_x
\]

\[
\text{else}
\]

\[
\text{parent(root}_x) \leftarrow \text{root}_y
\]

\[
\text{if rank(root}_x) = \text{rank(root}_y) \quad \text{rank(root}_y) \leftarrow \text{rank(root}_y) + 1
\]

\[
S_1 = \{x, p\} \quad S_2 = \{a, h, c, f\}
\]

\[
\{x, p, a, h, c, f\}
\]

- attach shorter tree to root of taller one
- maintain rank as upper bound for height
- rank increases when both trees have the same rank
- find(x); find(y)
- if rank(root_x) > rank(root_y), parent(root_y) <- root_x
- else parent(root_x) <- root_y
- if rank(root_x) = rank(root_y), rank(root_y) <- rank(root_y) + 1
- S_1 = {x, p}, S_2 = {a, h, c, f}
- attach shorter tree to root of taller one
- maintain rank as upper bound for height
- rank increases when both trees have the same rank
- find(x); find(y)
- if rank(root_x) > rank(root_y), parent(root_y) <- root_x
- else parent(root_x) <- root_y
- if rank(root_x) = rank(root_y), rank(root_y) <- rank(root_y) + 1
- S_1 = {x, p}, S_2 = {a, h, c, f}
Union by Rank

- If \( \text{rank}(x) = h \), the tree rooted at \( x \) has at least \( 2^h \) nodes.
  - use induction; for the base, we know when \( h = 0 \), the tree contains \( 1 = 2^0 \) nodes.
  - choose any \( h > 0 \) and consider the union operation in which the rank is increased from \( h - 1 \) to \( h \).
  - at the time of union, both trees had rank \( h - 1 \)
  - by induction hypothesis, they each included at least \( 2^{h-1} \) nodes.
  - then the resulting tree has at least \( 2 \cdot 2^{h-1} = 2^h \) nodes.
  - The number of nodes is at least \( 2^h \) since after the union, the number of nodes can be increased further.

- Since the number of nodes is at least \( 2^h \), the height of the trees is \( O(\log n) \)
  - Union, find operations when we use union by rank is \( O(\log n) \).
Path Compression

- A simple, effective add on to union by rank
  - Find(x) involves finding a path from x to the root of its tree
  - For each node on the path, updated its pointer to point directly to the root:
    
    ```
    if x ≠ parent(x)
        parent(x) ← find(parent(x))
    return parent(x)
    ```

- For each visited node, the additional work is updating one pointer.
  - Time complexity remains the same asymptotically, i.e., $O(\log n)$.

- For any $y$ that used to lie on the path from $x$ to the root, any subsequent call to find($y$) takes $O(1)$ time
  - the amortized time is significantly improved.
Disjoint set data structure

- Maintain a set of disjoint forests
  - Apply union-by rank after union operation (attach the tree with smaller rank to the one with higher rank)
  - Apply path compression after find operation (update the pointer of any node on the Find path to point to the root)
    - Note that the height might change after path compression; hence we use term rank as an upper bound for height
- The amortized time for performing any operation is $O(\alpha(n))$ where $\alpha(n)$ is a very, very, very slow growing function of $n$ similar to inverse Ackermann function.
  - For any practical reason, $\alpha(n) \leq 4$.
  - In practice (not in theory) you can support disjoint operations in constant time.
Disjoint set data structure Review

- Maintain a set of disjoint forests
  - Apply union-by rank after union operation (attach the tree with smaller rank to the one with higher rank)
  - Apply path compression after find operation (update the pointer of any node on the Find path to point to the root)
- The amortized time for performing any operation is $O(\alpha(n))$ where $\alpha(n)$ is a very, very, very slow growing function of $n$ similar to inverse Ackermann function.
$\alpha(n)$ Description

Let $f^{(i)}(n)$ denote $f(n)$ iteratively applied $i$ times to the initial value of $n$.

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases}$$

- E.g., if $f(n) = 2n$, then
  
  $f^{(0)}(n) = n = 2^0 n,$
  $f^{(1)}(n) = f(f^{(0)}(n)) = 2(n) = 2^1 n,$
  $f^{(2)}(n) = f(f^{(1)}(n)) = 2(2^1 n) = 2^2 n,$
  ...
  
  $f^{(i)}(n) = f(f^{(i-1)}(n)) = 2(2^{i-1} n) = 2^i n,$

- E.g., if $f(n) = 2^n$, then

  $f^{(0)}(n) = n$
  $f^{(1)}(n) = f(f^{(0)}(n)) = f(n) = 2^n$
  $f^{(2)}(n) = f(f^{(1)}(n)) = f(2^n) = 2^{2^n}$
  ...
  
  $f^{i}(n) = f(f^{(i-1)}(n)) = 2^{2^{\cdot\cdot\cdot^{2^{i-1}n}}} \} i \text{ times}$
For any \( k \geq 0 \) and \( j \geq 1 \), let

\[
A_k(j) = \begin{cases} 
  j + 1 & \text{if } k = 0 \\
  A_{k-1}^{(j+1)}(j) & \text{if } k > 0 
\end{cases}
\]

Function \( A_k(j) \) is strictly increasing in both \( j \) and \( k \)

- For \( j > 0 \), \( A_1(j) = 2j + 1 \).
- For \( j > 0 \), \( A_2(j) = 2^{j+1}(j + 1) - 1 \).
- \( A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2^{11} - 1 = 2047 \)
- \( A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047) >> A_2(2047) = 2^{2048}(2048) - 1 > 2^{2048} >> 10^{80} \)
- \( A_4(1) \) is by far larger than the number of atoms in the universe.
\(\alpha(n)\) Description (cntd.)

- \(\alpha(n)\) is the inverse of \(A_k(n)\): \(\alpha(n) = \min\{k|A_k(1) \geq n\}\)
  - \(\alpha(n)\) is the lowest value of \(k\) for which \(A_k(1)\) is at least \(n\)
    
    \[
    \alpha(n) = \begin{cases} 
      0 & \text{for } 0 \leq n \leq 2 \\
      1 & \text{for } n = 3 \\
      2 & \text{for } 4 \leq n \leq 7 \\
      3 & \text{for } 8 \leq n \leq 2047 \\
      4 & \text{for } 2048 \leq n \leq A_4(1) 
    \end{cases}
    \]

- For any practical purpose, \(\alpha(n) \leq 4\).
- Theoretically, however, \(\alpha(n) \in \omega(1)\), i.e., for every constant \(c\), there is a very huge \(n\) such that \(\alpha(n) \geq c\).

Recall that the worst-case amortized time for performing an operation (make-set, union, find) is \(\alpha(n)\).

- This bound is tight, i.e., we cannot do better than \(\alpha(n)\).
- \(\alpha(n)\) is the smallest super-constant function that appears in algorithm analysis (there are smaller ones like \(\alpha(\alpha(n))\) which don't appear in analysis of algorithms).