Lecture 14

Instructor: Subrahmanyam Kalyanasundaram

3rd October 2019

Plan

- Union-Find Data Structure
- Disjoint Forests
- Path Compression

Kruskal's Algorithm Running Time

Algorithm 1 Kruskal's algorithm

```
A = ∅
for each vertex v ∈ V do
    MAKE-SET(v)
end for
Sort the edges in E into nondecreasing order by weight w
for each edge (u, v) ∈ E taken in nondecreasing order by weight do
    if FIND-SET(u) ≠ FIND-SET(v) then
        A = A ∪ {(u, v)}
        UNION(u, v)
    end if
end for
Return A
```

Kruskal's Algorithm Running Time

- $|V|$ Make-Set(v) operations
- At most $2|E|$ Find-Set(v) operations
- $|V| - 1$ Union(u, v) operations

Kruskal's Algorithm Running Time

- $|V|$ $\textsc{Make-Set}(v)$ operations
- At most $2|E|$ $\textsc{Find-Set}(v)$ operations
- $|V| - 1$ $\textsc{Union}(u, v)$ operations

- And a sort the edges – takes $O(|E| \log |E|)$ time
- Running time: $O(|E| \log |E| + |V| + |E| \cdot T_F + |V| \cdot T_U)$

Disjoint Set

Maintain a collection $\mathcal{F} = \{S_1, S_2, \ldots, S_k\}$ of disjoint sets.
One element from each set serves as a 'representative' for that set.

Disjoint Set supports the following procedures:

- $\textsc{MakeSet}(x)$ – Creates a singleton set with element x.
- $\textsc{Union}(x, y)$ – Performs union on sets containing x and y.
- $\textsc{FindSet}(x)$ – Find the set containing x.

MakeSet

MakeSet(x)

Creates a singleton set containing x.

We assume that x is not an element of any other set in $\mathcal{F}$.
We assign x as the representative for the set just created.

Union

$\textsc{Union}(x, y)$

Performs union on sets containing x and y.

Let $S, T \in \mathcal{F}$ such that $x \in S$ and $y \in T$.
Create a new set $U = S \cup T$.
Choose and assign a representative for U.
Remove S and T from $\mathcal{F}$.

FindSet

FindSet(x)
Find the set containing x.

Let $S \in \mathcal{F}$ such that $x \in S$. (Note: exactly one set contains x.)
Return a pointer to the representative element of S.

Implementation

Disjoint Set using linked lists:

Implementation

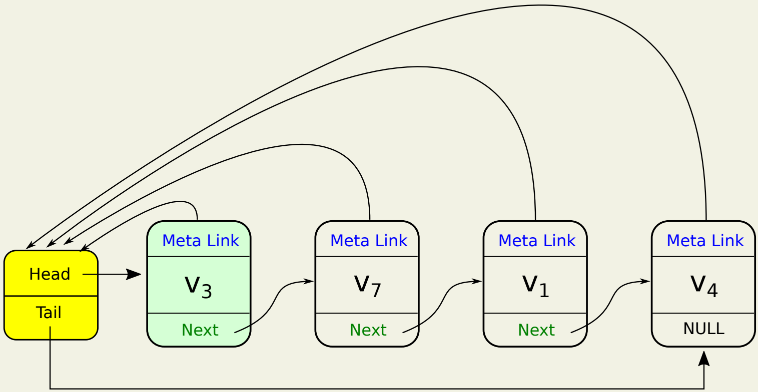
Disjoint Set using linked lists:

- For each set S, maintain:
  - a node with metadata
  - a linked list L_S with the objects in the set.
- The "Metadata Node" stores head and tail pointers to the linked list.
- Each node in the linked list consists of:
  - The value of the element.
  - A pointer to the next element.
  - A pointer to the Metadata Node.

The head of L_S is the representative of S.

Implementation

Linked list for set $\{v_1, v_3, v_4, v_7\}$.



Implementation

$\textsc{MakeSet}(x)$

Creates a singleton set with element x

- Create a new node for metadata
- Create a linked list containing just x.
- Node x is the head and tail of the list.
- Representative for this set is x itself.

Implementation

FindSet(x)

Find the set containing node x.

- Return a pointer to the representative.

Implementation

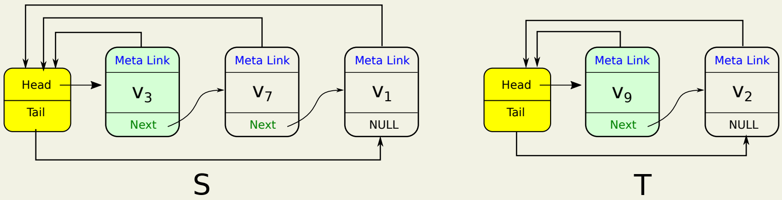
UNION(X,Y)

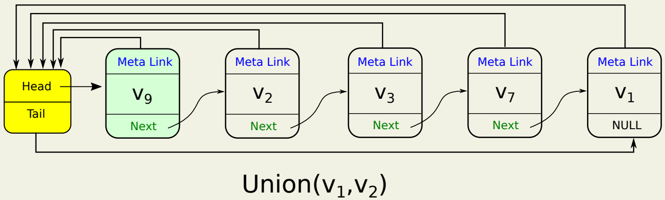
Union of sets containing x and y.

- Append linked list of set S containing x to set T containing y.
- Representative of new set is same as representative of T.
- Update meta pointers of nodes in S to the correct metadata node.
- Update tail pointer in metadata node of T.

Implementation

Union of sets $S = \{v_1, v_3, v_7\}$ and $T = \{v_2, v_9\}$.





Analysis

Running time under Linked List implementation:

- $\textsc{MakeSet}(x)$ – $O(1)$
- $\textsc{FindSet}(x)$ – $O(1)$
- $\textsc{Union}(x, y)$ – ?

Analysis

$\textsc{Union}(x, y)$ –

- $S \leftarrow \textsc{FindSet}(\text{x})$ and $T \leftarrow \textsc{FindSet}(\text{y})$ – $O(1)$ time.
- Appending linked list of S to tail end of T – $O(1)$ time.
- Updating the new metadata (tail) – $O(1)$ time.
- Updating the backward pointers of nodes in S takes $O(n)$ time.

We can show a case where after $O(n)$ operations, time taken would be $O(n^2)$.

Recap: List Implementation

Disjoint Set using linked lists:

- For each set S, maintain:
  - a node with metadata
  - a linked list L_S with the objects in the set.
- The "Metadata Node" stores:
  - Head and tail pointers to the linked list.
- Each node in the linked list consists of:
  - The value of the element.
  - A pointer to the next element.
  - A pointer to the Metadata Node.

The head of L_S is the representative of S.

List Implementation - Union by Rank heuristic

Disjoint Set using linked lists, union by rank:

- For each set S, maintain:
  - a node with metadata
  - a linked list L_S with the objects in the set.
- The "Metadata Node" stores:
  - Head and tail pointers to the linked list.
  - Size of the set.
- Each node in the linked list consists of:
  - The value of the element.
  - A pointer to the next element.
  - A pointer to the Metadata Node.

The head of L_S is the representative of S.

List Implementation - Union by Rank heuristic

UNION(X,Y)

Union of sets containing x and y.

Let $x \in S$ and $y \in T$.

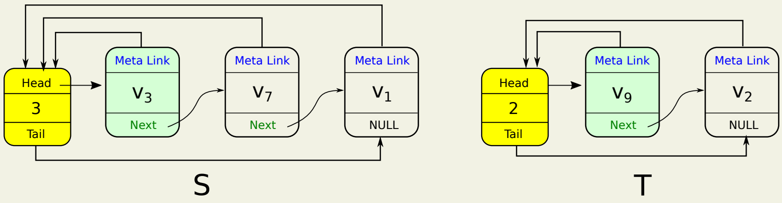
If $|S| \leq |T|$,

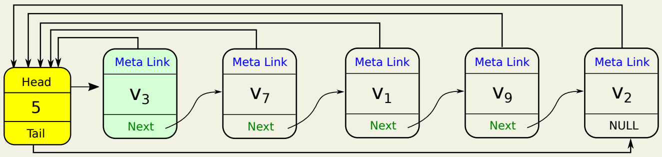
- Append list of S to tail end of list of T.
- Representative of new set is same as that of T.
- Update meta pointers of nodes in S
- Update tail pointer in metadata node of T.
- Update size of set in the metadata node.

Else, do the opposite.

Implementation - Union by Rank heuristic

Union of sets $S = \{v_1, v_3, v_7\}$ and $T = \{v_2, v_9\}$.





Union(v_1,v_2)

Analysis - Union by Rank heuristic

Theorem

A sequence of m operations in total, n of which are MakeSet takes $O(m + n \log n)$ time.

Analysis - Union by Rank heuristic

Observation 1

Updating the meta pointers takes the most time.

Observation 2

The meta pointer of a node x is updated only when union happens with a bigger set.

Proof strategy

- Fix an element x.
- Count number of times the meta pointer on node x is updated.

Analysis - Union by Rank heuristic

Observation 2 (informal)

If x lived inside a set of size s,
and a union operation updated its meta pointer,
then x now lives inside a set of size at least $2s$.

- Initially, x starts off as a singleton set.
- After k many updates to its meta pointer, it lives inside a set of size at least 2^k.
- Total number of elements is n. So $2^k \leq n$.
- This means $k \leq \log n$

Hence, for each element the meta pointer can be updated at most $k \leq \log n$ many times.
Worst case total number of updates to meta pointer across all n elements is $n \log n$.

□

Implementation - disjoint forests

The disjoint forest implementation:

- Each set S is implemented as a rooted tree.

A node corresponding to an $x \in S$ contains:

- The value (or pointer to) x.
- A pointer to its parent.

Implementation - disjoint forests

The disjoint forest implementation:

- Each set S is implemented as a rooted tree.

A node corresponding to an $x \in S$ contains:

- The value (or pointer to) x.
- A pointer to its parent.

Note:

- There are no pointers to children nodes!
- There is no dedicated metadata node for each set.
- Convention: Parent of root will be itself.
- Root node is also the representative.

Implementation - disjoint forests

Example picture on whiteboard

MakeSet(x)

MakeSet(x) involves creating a new tree with a single node for x.

MakeSet(x)

Whiteboard

FindSet(x)

FindSet(x):

- Start at node x.
- Follow the parent pointer starting from x.
- Return the root.

$\textsc{Union}(x, y)$

$\textsc{Union}(x, y)$:

- $r_x \leftarrow \textsc{FindSet}(x)$
- $r_y \leftarrow \textsc{FindSet}(y)$.
- $\mathrm{parent}(r_x) \leftarrow r_y$.

Whiteboard

Analysis – disjoint forests

Worst case running times under disjoint forest implementation:

- $\textsc{MakeSet}(x)$ –

Analysis – disjoint forests

Worst case running times under disjoint forest implementation:

- MakeSet(x) $-O(1)$
- FindSet(x) –

Analysis – disjoint forests

Worst case running times under disjoint forest implementation:

- $\textsc{MakeSet}(x)$ – $O(1)$
- $\textsc{FindSet}(x)$ – $O(n)$
- $\textsc{Union}(x, y)$ –

Analysis – disjoint forests

Worst case running times under disjoint forest implementation:

- $\textsc{MakeSet}(x)$ $-O(1)$
- $\textsc{FindSet}(x)$ $-O(n)$
- $\textsc{Union}(x, y)$ $-O(n)$

where n is the number of elements handled.

Disjoint Forests – "Union by Size" heuristic

Let's try to imitate what we did in the case of lists.
Let $n(x)$ denote the number of points in the set containing x.
$\textsc{Union}(x, y)$:

- If $n(x) \leq n(y)$, then
  - $r_x \leftarrow \textsc{FindSet}(x)$
  - $r_y \leftarrow \textsc{FindSet}(y)$.
  - $\text{parent}(r_x) \leftarrow r_y$.
- Else, do the opposite.

Disjoint Forests – “Union by Size” heuristic

Theorem

Suppose a tree contains n nodes. Then the height of a tree is at most $\log n$.

Proof

Key Induction Step: Suppose two trees are merged with n_1 and n_2 nodes. The number of nodes in the merged tree is $n = n_1 + n_2$.

Disjoint Forests – “Union by Size” heuristic

Theorem

Suppose a tree contains n nodes. Then the height of a tree is at most $\log n$.

Proof

Key Induction Step: Suppose two trees are merged with n_1 and n_2 nodes. The number of nodes in the merged tree is $n = n_1 + n_2$.
Let h_1, h_2 be the heights of the original trees and h be that of the merged tree. WLOG, we have $n_1 \leq n_2$.

Disjoint Forests – “Union by Size” heuristic

Theorem

Suppose a tree contains n nodes. Then the height of a tree is at most $\log n$.

Proof

Key Induction Step: Suppose two trees are merged with n_1 and n_2 nodes. The number of nodes in the merged tree is $n = n_1 + n_2$.

Let h_1, h_2 be the heights of the original trees and h be that of the merged tree. WLOG, we have $n_1 \leq n_2$.

- $h = \max(h_2, h_1 + 1)$
- By induction, $h_2 \leq \log n_2 \leq \log(n_1 + n_2)$
- Also, $h_1 + 1 \leq (\log n_1) + 1 = \log(2n_1) \leq \log(n_1 + n_2)$.

So $h \leq \log(n_1 + n_2)$.

Disjoint Forests – Union by Rank heuristic

To use the Rank heuristic:

- Each node will contain a “rank” (or height).
- Every node starts with a rank of 0.
- Update rank only when Union is called.

Disjoint Forests – Union by Rank heuristic

Union(x, y):
Let x, y with representatives r_x and r_y respectively.

If $\text{rank}(r_x) \leq \text{rank}(r_y)$, then:

- $\text{parent}(r_x) \leftarrow (r_y)$.

Else:

- $\text{parent}(r_y) \leftarrow (r_x)$.

If $\text{rank}(r_x) = \text{rank}(r_y)$, then:

- Increment $\text{rank}(r_y)$.

Analysis – Union by Rank

Theorem

A tree of height h has at least 2^h nodes.

Proof

Key Induction Step: Suppose two trees to be merged have height h_1 and h_2. Let the number of nodes in the two trees be n_1 and n_2 respectively.

Analysis – Union by Rank

Theorem

A tree of height h has at least 2^h nodes.

Proof

Key Induction Step: Suppose two trees to be merged have height h_1 and h_2. Let the number of nodes in the two trees be n_1 and n_2 respectively.
WLOG, let $h_1 \leq h_2$. The height of the merged tree is $h = \max(h_2, h_1 + 1)$.

Analysis – Union by Rank

Theorem

A tree of height h has at least 2^h nodes.

Proof

Key Induction Step: Suppose two trees to be merged have height h_1 and h_2. Let the number of nodes in the two trees be n_1 and n_2 respectively.
WLOG, let $h_1 \leq h_2$. The height of the merged tree is $h = \max(h_2, h_1 + 1)$.

- $h = \max(h_2, h_1 + 1)$
- The number of nodes in the merged tree is $n = n_1 + n_2$
- $n \geq 2^{h_1} + 2^{h_2} \geq 2^{h_2}$.
- $n \geq 2^{h_1} + 2^{h_2} \geq 2^{h_1} + 2^{h_1} = 2^{h_1+1}$.

So $n \geq \max(2^{h_2}, 2^{h_1+1}) = 2^{\max(h_2, h_1+1)} = 2^h$.

Implementation – Union by Rank

- Maintain the rank (or size) of each node
- Union can be done in $O(1)$ time
- FindSet requires $O(\log n)$ time

Disjoint Forests – Path Compression heuristic

When FindSet(x) is called:

- Follow the parent pointer from x to root.
- Change the parent pointer of every node on this path to directly point to root.

Note: We can do Union by Size or Rank here.

Disjoint Forests – Path Compression heuristic

Whiteboard

Analysis – Union by Rank and Path Compression

Theorem

A sequence of m operations in total, n of which are MakeSet takes $O(m\alpha(n))$ time where $\alpha(n)$ is the inverse Ackermann

Ackermann function

The Ackermann function $A(m, n)$ is defined as:

- $n + 1$ if $m = 0$
- $A(m - 1, 1)$ if $m > 0$ and $n = 0$
- $A(m - 1, A(m, n - 1))$ if $m > 0$ and $n > 0$

Ackermann function

The Ackermann function $A(m, n)$ is defined as:

- $n + 1$ if $m = 0$
- $A(m - 1, 1)$ if $m > 0$ and $n = 0$
- $A(m - 1, A(m, n - 1))$ if $m > 0$ and $n > 0$

Example values:

- $A(0, 0) = 1$
- $A(1, 1) = 3$
- $A(2, 2) = 7$
- $A(3, 3) = 61$
- $A(4, 4) = 2^{2^{65536}} - 3$

Ackermann function

The Ackermann function $A(m, n)$ is defined as:

- $n + 1$ if $m = 0$
- $A(m - 1, 1)$ if $m > 0$ and $n = 0$
- $A(m - 1, A(m, n - 1))$ if $m > 0$ and $n > 0$

Example values:

- $A(0, 0) = 1$
- $A(1, 1) = 3$
- $A(2, 2) = 7$
- $A(3, 3) = 61$
- $A(4, 4) = 2^{2^{65536}} - 3$

The inverse Ackermann $\alpha(n)$ is the smallest k for which $n < A(k, k)$.