Implementing Kruskal’s Algorithm: Union-Find

Algorithm Design & Analysis

Fall 2018
Outline

Geeks and repetitive tasks

- Time spent
- Runs script
- Gets annoyed
- Does it manually
- Makes fun of geek's complicated method

Geek vs. non-geek

- Does it manually
- Loses
- Wins

Task size
Moderate Greed: Prim’s Algorithm

Here $T$ is at tree at all times—the cheapest tree on the subgraph of $G$ that is spans.
Here $T$ is at tree at all times—the cheapest tree on the subgraph of $G$ that is spans.
Prim maintains a subtree $T = (V', E')$ of $G$ and adds the cheapest cut edge of $E(V', V - V')$ (in $G$) to $T$. 

**Algorithm 2**

```plaintext
procedure Prim(G, c()) // G = (V, E) is connected
Select some $v \in V$; $V' \leftarrow \{v\}$; $T \leftarrow (V', \emptyset)$ // The eventual MCST
while $|E(T)| < |V| - 1$
  Select cheapest edge $e \in E(V', V - V')$
  Add $e$ to $T$ // This adds a new vertex to $T$
end procedure
```
Moderate Greed: Prim’s Algorithm

Here $T$ is at tree at all times—the cheapest tree on the subgraph of $G$ that is spans.

Prim maintains a subtree $T = (V', E')$ of $G$ and adds the cheapest cut edge of $E(V', V - V')$ (in $G$) to $T$.

**Algorithm 3 Prim’s Algorithm**

```plaintext
procedure PRIM(G, c()) // $G = (V, E)$ is connected
    Select some $v \in V$; $V' \leftarrow \{v\}; \ T \leftarrow (V', \emptyset)$ // The eventual MCST
    while $|E(T)| < |V| - 1$ do
        Select cheapest edge $e \in E(V', V - V')$
        Add $e$ to $T$ // This adds a new vertex to $T$
    end procedure
```
**Moderate Greed: Prim’s Algorithm**

**Algorithm 4** Prim’s Algorithm: More Detail

\[
\text{procedure } \text{PRIM}(G, c()) \quad // \quad G = (V, E) \text{ is connected}
\]

Select some \( v \in V; \ V' \leftarrow \{v\} \)

\( T \leftarrow (V', \emptyset) \quad // \quad \text{The eventual MCST} \)

\( C \leftarrow E(V', V - V') \)

// \( C \) will always contain \( E(V', V - V') \)

\[ \text{while } |E(T)| < |V| - 1 \text{ do} \]

Select cheapest edge \( e \in C \)

\[ \text{if } e \in E(V', V - V') \text{ then} \]

// Let \( e = \{u, v\} \), where \( u \in V' \) and \( v \in V - V' \)

Add \( v \) to \( T \); Add \( e \) to \( T \)

\[ \text{for neighbors } w \text{ of } v \text{ do} \]

if \( w \in V - V' \) then add \( \{v, w\} \) to \( C \)

end procedure
The Implementation

Notes

Theorem

Prim's algorithm uses $O(|V| + |E|)$ space and runs in $O((|V| + |E|) \log |E|)$ time.

Proof:
The loop executes at most $|E|$ times and each iteration performs a constant number of operations other than addition to (or removal from) $C$, a heap of size $O(|E|)$.

But the total number of heap operations is $O(|E|)$ and each takes $O(\log |E|)$ time.

Note:

Since $|E| \in O(|V|^2)$, $\log |E| \in O(\log |V|)$.

So Prim runs in time $O((|V| + |E|) \log |V|)$ time.
The Implementation

Notes

- $C$ is stored as a priority queue with edge weight as priority
The Implementation

Notes

- $C$ is stored as a priority queue with edge weight as priority
- $E(V', V - V') \subseteq C$; $C$ may also contain edges of $G$ between vertices of $V'$

Theorem

Prim's algorithm uses $O(|V| + |E|)$ space and runs in $O((|V| + |E|) \log |E|)$ time.

Proof:
The loop executes at most $|E|$ times and each iteration performs a constant number of operations other than addition to (or removal from) $C$, a heap of size $O(|E|)$.

But the total number of heap operations is $O(|E|)$ and each takes $O(\log |E|)$ time.

Note:
Since $|E| \in O(|V|^2)$, $\log |E| \in O(\log |V|)$.

So Prim runs in time $O((|V| + |E|) \log |V|)$ time.
The Implementation

Notes

- \( C \) is stored as a priority queue with edge weight as priority
- \( E(V', V - V') \subseteq C \); \( C \) may also contain edges of \( G \) between vertices of \( V' \)
- If cheapest edge in \( C \) is not in \( E(V', V - V') \), ignore it

Theorem

Prim's algorithm uses \( O(|V| + |E|) \) space and runs in \( O((|V| + |E|) \log |E|) \) time.

Proof:
The loop executes at most \( |E| \) times and each iteration performs a constant number of operations other than addition to (or removal from) \( C \), a heap of size \( O(|E|) \). But the total number of heap operations is \( O(|E|) \) and each takes \( O(\log |E|) \) time.

Note:
Since \( |E| \in O(|V|^2) \), \( \log |E| \in O(\log |V|) \).

So Prim runs in time \( O((|V| + |E|) \log |V|) \) time.
The Implementation

Notes

- $C$ is stored as a priority queue with edge weight as priority
- $E(V', V - V') \subseteq C$; $C$ may also contain edges of $G$ between vertices of $V'$
- If cheapest edge in $C$ is not in $E(V', V - V')$, ignore it

Theorem

Prim’s algorithm uses $O(|V| + |E|)$ space and runs in $O((|V| + |E|) \log |E|)$ time
The Implementation

Notes

- $C$ is stored as a priority queue with edge weight as priority
- $E(V', V - V') \subseteq C$; $C$ may also contain edges of $G$ between vertices of $V'$
- If cheapest edge in $C$ is not in $E(V', V - V')$, ignore it

Theorem

Prim’s algorithm uses $O(|V| + |E|)$ space and runs in $O((|V| + |E|) \log |E|)$ time

Proof: The loop executes at most $|E|$ times and each iteration performs a constant number of operations other than addition to (or removal from) $C$, a heap of size $O(|E|)$. 
The Implementation

Notes

• C is stored as a priority queue with edge weight as priority
• $E(V', V - V') \subseteq C$; C may also contain edges of $G$ between vertices of $V'$
• If cheapest edge in $C$ is not in $E(V', V - V')$, ignore it

Theorem

Prim's algorithm uses $O(|V| + |E|)$ space and runs in $O((|V| + |E|) \log |E|)$ time

Proof: The loop executes at most $|E|$ times and each iteration performs a constant number of operations other than addition to (or removal from) $C$, a heap of size $O(|E|)$. But the total number of heap operations is $O(|E|)$ and each takes time $O(\log |E|)$. 
The Implementation

Notes

• $C$ is stored as a priority queue with edge weight as priority
• $E(V', V - V') \subseteq C$; $C$ may also contain edges of $G$ between vertices of $V'$
• If cheapest edge in $C$ is not in $E(V', V - V')$, ignore it

Theorem

*Prim’s algorithm uses $O(|V| + |E|)$ space and runs in $O((|V| + |E|) \log |E|)$ time*

Proof: The loop executes at most $|E|$ times and each iteration performs a constant number of operations other than addition to (or removal from) $C$, a heap of size $O(|E|)$.

But the total number of heap operations is $O(|E|)$ and each takes time $O(\log |E|)$.

Note: Since $|E| \in O(|V|^2)$, $\log |E| \in O(\log |V|)$
The Implementation

Notes

• $C$ is stored as a priority queue with edge weight as priority
• $E(V', V - V') \subseteq C$; $C$ may also contain edges of $G$ between vertices of $V'$
• If cheapest edge in $C$ is not in $E(V', V - V')$, ignore it

Theorem

Prim’s algorithm uses $O(|V| + |E|)$ space and runs in $O((|V| + |E|) \log |E|)$ time

Proof: The loop executes at most $|E|$ times and each iteration performs a constant number of operations other than addition to (or removal from) $C$, a heap of size $O(|E|)$. But the total number of heap operations is $O(|E|)$ and each takes time $O(\log |E|)$.

Note: Since $|E| \in O(|V|^2)$, $\log |E| \in O(\log |V|)$
So Prim runs in time $O((|V| + |E|) \log |V|)$ time.
Extreme greed: Kruskal’s Algorithm

Keep adding cheapest edge that doesn’t create a cycle.

**Algorithm 5 Kruskal’s Algorithm**

```
procedure KRUSKAL(G, c()) // G = (V, E) is connected
    T ← (V, {}) // The eventual MCST
    F ← E
    while |E(T)| < |V| − 1 do
        Remove cheapest edge e ∈ F from F
        if T + {e} does not contain a cycle then
            Add e to T
    end procedure
```
The Implementation

To implement Kruskal’s Algorithm, we want efficient methods to

• Find the cheapest edge in $F$
• Determine whether $T + \{e\}$ contains a cycle
• Add an edge to $T$

To find the cheapest edge in $F$, we can store $F$ as a min-heap.
To create the heap, we store $F$ as an array and heapify it.
How should we heapify it: bottom-up or top-down?
The Implementation

To implement Kruskal’s Algorithm, we want efficient methods to

- Find the cheapest edge in $F$
The Implementation

To implement Kruskal’s Algorithm, we want efficient methods to

• Find the cheapest edge in $F$
• Determine whether $T + \{e\}$ contains a cycle
To implement Kruskal’s Algorithm, we want efficient methods to

- Find the cheapest edge in $F$
- Determine whether $T + \{e\}$ contains a cycle
- Add an edge to $T$
The Implementation

To implement Kruskal’s Algorithm, we want efficient methods to

- Find the cheapest edge in $F$
- Determine whether $T + \{e\}$ contains a cycle
- Add an edge to $T$

To find the cheapest edge in $F$, we can store $F$ as a min-heap.
To implement Kruskal’s Algorithm, we want efficient methods to

- Find the cheapest edge in $F$
- Determine whether $T + \{e\}$ contains a cycle
- Add an edge to $T$

To find the cheapest edge in $F$, we can store $F$ as a min-heap.

To create the heap, we store $F$ as an array and heapify it.
To implement Kruskal’s Algorithm, we want efficient methods to

- Find the cheapest edge in $F$
- Determine whether $T + \{e\}$ contains a cycle
- Add an edge to $T$

To find the cheapest edge in $F$, we can store $F$ as a min-heap.

To create the heap, we store $F$ as an array and heapify it.

How should we heapify it: bottom-up or top-down?
**Bottom-Up Heapify**

How to turn an array into a heap quickly

---

**Algorithm 6** Bottom-up Heapify

```plaintext
procedure BUHEAPIFY(H[])
    // Ignore H[0]; n is largest value with H[n] not empty
    // H[n/2 + 1..] have no children with data
    for i ← n/2 down to i ← 1 do
        HeapifyDown(H,i)
end procedure
```

---

**Correctness:**

- Before the loop executes, `H[n/2 + 1..n]` has the heap property.
- By induction, just before we call `HeapifyDown(H,i)`, `H[i+1..n]` satisfies the heap property.
- So, after the call, `H[i..n]` satisfies the heap property.
**Bottom-Up Heapify**

How to turn an array into a heap quickly

---

### Algorithm 7 Bottom-up Heapify

```plaintext
procedure BUHEAPIFY(H[])
    // Ignore H[0]; n is largest value with H[n] not empty
    // H[n/2 + 1..] have no children with data
    for i ← n/2 down to i ← 1 do
        HeapifyDown(H,i)
    end procedure
```

---

**Correctness:**

Before the loop executes, H[n/2..n] has the heap property. By induction, just before we call HeapifyDown(H, i), H[i+1..n] satisfies the heap property. So, after the call, H[i..n] satisfies the heap property.
Bottom-Up Heapify

How to turn an array into a heap quickly

Algorithm 8 Bottom-up Heapify

procedure \text{BUHeapify}(H[])
// Ignore $H[0]$; $n$ is largest value with $H[n]$ not empty
// $H[n/2 + 1..]$ have no children with data
for $i \leftarrow n/2$ down to $i \leftarrow 1$ do
  HeapifyDown(H,i)
end procedure

Correctness:

- Before the loop executes, $H[n/2 + 1..n]$ has the heap property.
**Bottom-Up Heapify**

How to turn an array into a heap quickly

---

**Algorithm 9 Bottom-up Heapify**

```plaintext
procedure BUHEAPIFY(H[])
    // Ignore H[0]; n is largest value with H[n] not empty
    // H[n/2 + 1..] have no children with data
    for i ← n/2 down to i ← 1 do
        HeapifyDown(H,i)
end procedure
```

---

**Correctness:**

- Before the loop executes, $H[n/2 + 1..n]$ has the heap property.
- By induction, just before we call `HeapifyDown(H, i)`, $H[i + 1...n]$ satisfies the heap property.
**Bottom-Up Heapify**

How to turn an array into a heap quickly

---

**Algorithm 10 Bottom-up Heapify**

```plaintext
procedure BUHEAPIFY(H[])
    // Ignore H[0]; n is largest value with H[n] not empty
    // H[n/2 + 1..] have no children with data
    for i ← n/2 down to i ← 1 do
        HeapifyDown(H,i)
end procedure
```

---

**Correctness:**

- Before the loop executes, $H[n/2 + 1..n]$ has the heap property.
- By induction, just before we call `HeapifyDown(H, i)`, $H[i + 1..n]$ satisfies the heap property.
- So, after the call, $H[i..n]$ satisfies the heap property.
Time Complexity of Bottom-Up Heapify

A binary tree of height \( h \) has at most \( n = 2^h + 1 \) nodes:

\[
\sum_{i=0}^{h} \text{# of elements at depth } i = 2^h + 1 - 1
\]

If the heap has height \( h \), it has \( 2^h \leq n \leq 2^h + 1 - 1 \) nodes:

- There are \( 2^h - 1 \) nodes of depth at most \( h - 1 \)
- There are at most \( 2^h \) additional nodes at depth \( h \), for a total of at most \( 2^h + 2^h - 1 = 2^h + 1 \) nodes

A node at depth \( d \) will be swapped at most \( h - d \) times, so the total number of swaps is

\[
\sum_{d=0}^{h} \left( \frac{h - d}{2^d} \right) \leq 2n - \lfloor \log n \rfloor - 2 \in O(n)
\]
Time Complexity of Bottom-Up Heapify

• A binary tree of height $h$ has at most $n = 2^{h+1} - 1$ nodes:

\[
\sum_{i=0}^{h} 2^i = 2^{h+1} - 1
\]
**Time Complexity of Bottom-Up Heapify**

- A binary tree of height $h$ has at most $n = 2^{h+1} - 1$ nodes:
  
  $$n = \sum_{i=0}^{h} \# \text{ of elements at depth } i = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$
Time Complexity of Bottom-Up Heapify

- A binary tree of height $h$ has at most $n = 2^{h+1} - 1$ nodes:

$$n = \sum_{i=0}^{h} \text{# of elements at depth } i = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

- If the heap has height $h$, it has $2^h \leq n \leq 2^{h+1} - 1$ nodes:
Time Complexity of Bottom-Up Heapify

- A binary tree of height $h$ has at most $n = 2^{h+1} - 1$ nodes:

$$n = \sum_{i=0}^{h} \text{# of elements at depth } i = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

- If the heap has height $h$, it has $2^h \leq n \leq 2^{h+1} - 1$ nodes:
  - There are $2^h - 1$ nodes of depth at most $h - 1$
Time Complexity of Bottom-Up Heapify

- A binary tree of height $h$ has at most $n = 2^{h+1} - 1$ nodes:

  \[ n = \sum_{i=0}^{h} \text{# of elements at depth } i = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \]

- If the heap has height $h$, it has $2^h \leq n \leq 2^{h+1} - 1$ nodes:
  - There are $2^h - 1$ nodes of depth at most $h - 1$
  - There are at most $2^h$ additional nodes at depth $h$, for a total of at most $2^h + 2^h - 1 = 2^{h+1} - 1$ nodes

- A node at depth $d$ will be swapped at most $h - d$ times, so total number of swaps is
**Time Complexity of Bottom-Up Heapify**

- A binary tree of height \( h \) has at most \( n = 2^{h+1} - 1 \) nodes:

\[
n = \sum_{i=0}^{h} \# \text{ of elements at depth } i = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1
\]

- If the heap has height \( h \), it has \( 2^h \leq n \leq 2^{h+1} - 1 \) nodes:
  - There are \( 2^h - 1 \) nodes of depth at most \( h - 1 \)
  - There are at most \( 2^h \) additional nodes at depth \( h \), for a total of at most \( 2^h + 2^h - 1 = 2^{h+1} - 1 \) nodes

- A node at depth \( d \) will be swapped at most \( h - d \) times, so total number of swaps is

\[
\sum_{d=0}^{d=h} (\# \text{ of nodes at depth } d) \times (\# \text{ of swaps}) \leq \sum_{d=0}^{d=h} (h - d)2^d
\]
Time Complexity of Bottom-Up Heapify

- A binary tree of height $h$ has at most $n = 2^{h+1} - 1$ nodes:

$$n = \sum_{i=0}^{h} \text{# of elements at depth } i = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

- If the heap has height $h$, it has $2^h \leq n \leq 2^{h+1} - 1$ nodes:
  - There are $2^h - 1$ nodes of depth at most $h - 1$
  - There are at most $2^h$ additional nodes at depth $h$, for a total of at most $2^h + 2^h - 1 = 2^{h+1} - 1$ nodes

- A node at depth $d$ will be swapped at most $h - d$ times, so total number of swaps is

$$\sum_{d=0}^{d=h} \text{(# of nodes at depth } d) \times \text{(# of swaps)} \leq \sum_{d=0}^{d=h} (h - d)2^d$$

$$\sum_{d=0}^{d=h}(h - d)2^d = 2^{h+1} - h - 2 \leq 2n - \lfloor \log n \rfloor - 2 \in O(n)$$
Union-Find: Cycle Checking and Tree Merging

**Cycle Checking:** Given the next edge $e \in F$, does $T + \{e\}$ contain a cycle?

**Idea:**
Union-Find: Cycle Checking and Tree Merging

**Cycle Checking:** Given the next edge \( e \in F \), does \( T + \{ e \} \) contain a cycle?

**Idea:**

- Maintain a partition of \( V \) based on components of \( T \)
**Union-Find: Cycle Checking and Tree Merging**

**Cycle Checking:** Given the next edge $e \in F$, does $T + \{e\}$ contain a cycle?

**Idea:**
- Maintain a partition of $V$ based on components of $T$
- Note: $T$ is a forest and each component of $T$ is a tree
**Union-Find: Cycle Checking and Tree Merging**

**Cycle Checking:** Given the next edge \( e \in F \), does \( T + \{e\} \) contain a cycle?

**Idea:**
- Maintain a partition of \( V \) based on components of \( T \)
- Note: \( T \) is a forest and each component of \( T \) is a tree
- Denote by \( V_u \) the set in the partition containing vertex \( u \).
Cycle Checking: Given the next edge $e \in F$, does $T + \{e\}$ contain a cycle?

Idea:

- Maintain a partition of $V$ based on components of $T$
- Note: $T$ is a forest and each component of $T$ is a tree
- Denote by $V_u$ the set in the partition containing vertex $u$.
- Recall: For $u, w \in V$, either $V_u = V_w$ or $V_u \cap V_w = \emptyset$ (Equivalence classes)
Union-Find: Cycle Checking and Tree Merging

Cycle Checking: Given the next edge \( e \in F \), does \( T + \{e\} \) contain a cycle?

Idea:

- Maintain a partition of \( V \) based on components of \( T \)
- Note: \( T \) is a forest and each component of \( T \) is a tree
- Denote by \( V_u \) the set in the partition containing vertex \( u \).
- Recall: For \( u, w \in V \), either \( V_u = V_w \) or \( V_u \cap V_w = \emptyset \)
  (Equivalence classes)
- Adding an edge \( e = \{u, w\} \) from \( F \) to \( T \) creates a cycle iff \( V_u = V_w \)
Union-Find: Cycle Checking and Tree Merging

Cycle Checking: Given the next edge $e \in F$, does $T + \{e\}$ contain a cycle?

Idea:

- Maintain a partition of $V$ based on components of $T$
- Note: $T$ is a forest and each component of $T$ is a tree
- Denote by $V_u$ the set in the partition containing vertex $u$.
- Recall: For $u, w \in V$, either $V_u = V_w$ or $V_u \cap V_w = \emptyset$ (Equivalence classes)
- Adding an edge $e = \{u, w\}$ from $F$ to $T$ creates a cycle iff $V_u = V_w$
- So we need to be able to determine whether $V_u = V_w$
**Union-Find: Cycle Checking and Tree Merging**

**Cycle Checking:** Given the next edge $e \in F$, does $T + \{e\}$ contain a cycle?

**Idea:**

- Maintain a partition of $V$ based on components of $T$
- Note: $T$ is a forest and each component of $T$ is a tree
- Denote by $V_u$ the set in the partition containing vertex $u$.
- Recall: For $u, w \in V$, either $V_u = V_w$ or $V_u \cap V_w = \emptyset$ (Equivalence classes)
- Adding an edge $e = \{u, w\}$ from $F$ to $T$ creates a cycle iff $V_u = V_w$
- So we need to be able to determine whether $V_u = V_w$
- And we need to be able to merge $V_u$ with $V_w$ to "add" $e$ to $T"
A First Union-Find Structure

Union-Find Data Structure

• Manages a dynamic partition of a set $S$
• Provides the following methods:
  - MakeUnionFind(): Initialize the structure
  - Find($x$): Return name of set containing $x$
  - Union($X$, $Y$): Replace sets $X$ and $Y$ of partition with $Z = X \cup Y$.

Kruskal's Algorithm can then use Find for cycle checking and Union to update the structure after adding an edge to $T$. 
Union-Find Data Structure

- Manages a dynamic partition of a set $S$
A First Union-Find Structure

Union-Find Data Structure

- Manages a dynamic partition of a set $S$
- Provides the following methods

Kruskal’s Algorithm can then use Find for cycle checking and Union to update the structure after adding an edge to $T$. 
A First Union-Find Structure

Union-Find Data Structure

- Manages a dynamic partition of a set $S$
- Provides the following methods

  $\text{MakeUnionFind}()$: Initialize the structure

  $\text{Find}(x)$: Return name of set containing $x$

  $\text{Union}(X, Y)$: Replace sets $X$ and $Y$ of partition with $Z = X \cup Y$.

Kruskal's Algorithm can then use $\text{Find}$ for cycle checking and $\text{Union}$ to update the structure after adding an edge to $T$. 
A First Union-Find Structure

Union-Find Data Structure

• Manages a dynamic partition of a set $S$
• Provides the following methods

  $\text{MakeUnionFind}()$: Initialize the structure
  $\text{Find}(x)$: Return name of set containing $x$

Kruskal's Algorithm can then use $\text{Find}$ for cycle checking and $\text{Union}$ to update the structure after adding an edge to $T$. 
A First Union-Find Structure

Union-Find Data Structure

- Manages a dynamic partition of a set $S$
- Provides the following methods

  - `MakeUnionFind()`: Initialize the structure
  - `Find(x)`: Return name of set containing $x$
  - `Union(X, Y)`: Replace sets $X$ and $Y$ of partition with $Z = X \cup Y$.

Kruskal's Algorithm can then use `Find` for cycle checking and `Union` to update the structure after adding an edge to $T$. 
A First Union-Find Structure

Union-Find Data Structure

- Manages a dynamic partition of a set $S$
- Provides the following methods

  - $\text{MakeUnionFind}()$: Initialize the structure
  - $\text{Find}(x)$: Return name of set containing $x$
  - $\text{Union}(X, Y)$: Replace sets $X$ and $Y$ of partition with $Z = X \cup Y$.

Kruskal’s Algorithm can then use Find for cycle checking and Union to update the structure after adding an edge to $T$. 
Extreme greed: Kruskal’s Algorithm

Keep adding cheapest edge that doesn’t create a cycle.

Algorithm 11 Kruskal’s Algorithm

\begin{algorithm}
\begin{algorithmic}
\Procedure{Kruskal}{G, c() \text{ // } G = (V, E) \text{ is connected}}
\State $T \gets (V, \emptyset) \text{ // The eventual MCST}$
\State $F \gets E$
\State $MakeUnionFind(V)$
\While{$|E(T)| < |V| - 1$}
\State Remove cheapest edge $e = \{u, v\} \in F$ from $F$
\State $uName = \text{Find}(u); \ vName = \text{Find}(v)$
\If{$uName \neq vName$}
\State Add $e$ to $T$
\State $\text{Union}(uName, vName)$
\EndIf
\EndWhile
\EndProcedure
\end{algorithmic}
\end{algorithm}
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- $MakeUnionFind()$ creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- $\text{MakeUnionFind()}$ creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $UFSets[1 \ldots n]$ to hold the names: $UFSets[v] = v$: $O(n)$ time

Let's try an example...


First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- $MakeUnionFind()$ creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $UFSets[1 \ldots n]$ to hold the names: $UFSets[v] = v$: $O(n)$ time

- $Find(v)$ works by looking up the name of the set containing $v$ in the array $UFSets[1 \ldots n]$: $O(1)$ time
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- **MakeUnionFind()** creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $UFSets[1 \ldots n]$ to hold the names: $UFSets[v] = v$: $O(n)$ time

- **Find(v)** works by looking up the name of the set containing $v$ in the array $UFSets[1 \ldots n]$: $O(1)$ time

- **Union(X, Y)**: $X \cup Y$ gets the name of whichever set $X$ or $Y$ is larger (ties are broken arbitrarily)
**First Union-Find Implementation**

Let $S = \{1, \ldots, n\}$ be our set of items

- **MakeUnionFind()** creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $UFSets[1 \ldots n]$ to hold the names: $UFSets[v] = v$: $O(n)$ time

- **Find(v)** works by looking up the name of the set containing $v$ in the array $UFSets[1 \ldots n]$: $O(1)$ time

- **Union(X, Y)**: $X \cup Y$ gets the name of whichever set $X$ or $Y$ is larger (ties are broken arbitrarily)

- **Union(X, Y)** changes the names of each of the elements in the smaller set to the name of the larger set: $O(n)$ time
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- *MakeUnionFind()* creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $UFSets[1 \ldots n]$ to hold the names: $UFSets[v] = v: O(n)$ time

- *Find*(v) works by looking up the name of the set containing $v$ in the array $UFSets[1 \ldots n]: O(1)$ time

- *Union*(X, Y): $X \cup Y$ gets the name of whichever set $X$ or $Y$ is larger (ties are broken arbitrarily)

- *Union*(X, Y) changes the names of each of the elements in the smaller set to the name of the larger set: $O(n)$ time
  - Doing this changes fewer names...
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- $\text{MakeUnionFind}()$ creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $\text{UFSets}[1 \ldots n]$ to hold the names: $\text{UFSets}[v] = v$: $O(n)$ time

- $\text{Find}(v)$ works by looking up the name of the set containing $v$ in the array $\text{UFSets}[1 \ldots n]$: $O(1)$ time

- $\text{Union}(X, Y)$: $X \cup Y$ gets the name of whichever set $X$ or $Y$ is larger (ties are broken arbitrarily)

- $\text{Union}(X, Y)$ changes the names of each of the elements in the smaller set to the name of the larger set: $O(n)$ time
  - Doing this changes fewer names
  - Keeping linked lists of the elements of each set makes it easy to find the elements whose names need changing
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- $\text{MakeUnionFind}()$ creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $UFSets[1 \ldots n]$ to hold the names: $UFSets[v] = v$: $O(n)$ time

- $\text{Find}(v)$ works by looking up the name of the set containing $v$ in the array $UFSets[1 \ldots n]$: $O(1)$ time

- $\text{Union}(X, Y)$: $X \cup Y$ gets the name of whichever set $X$ or $Y$ is larger (ties are broken arbitrarily)

- $\text{Union}(X, Y)$ changes the names of each of the elements in the smaller set to the name of the larger set: $O(n)$ time
  - Doing this changes fewer names
  - Keeping linked lists of the elements of each set makes it easy to find the elements whose names need changing

Let’s try an example...
First Union-Find Implementation

Algorithm 12 Union()

procedure UNION(i, j) // i and j are set names
    // Assume |Si| \leq |Sj|; if not, swap i and j
    for x \in Si do
        UFSets[x] = j
    end procedure
Lemma Any initial sequence of $k$ Union’s takes total time $O(k \log k)$
A Surprising Fact

Lemma Any initial sequence of $k$ Union’s takes total time $O(k \log k)$

Proof Claim: The number of elements involved in the unions is at most $2k$
Lemma Any initial sequence of $k$ Union's takes total time $O(k \log k)$

Proof Claim: The number of elements involved in the unions is at most $2k$

- The first time an item is involved, it is a singleton set and so a parameter of the call to Union
A Surprising Fact

Lemma Any initial sequence of $k$ Union’s takes total time $O(k \log k)$

Proof Claim: The number of elements involved in the unions is at most $2k$

- The first time an item is involved, it is a singleton set and so a parameter of the call to Union
- $k$ calls to Union involves at most $2k$ distinct parameters
A Surprising Fact

**Lemma** Any initial sequence of $k$ Union’s takes total time $O(k \log k)$

**Proof** Claim: The number of elements involved in the unions is at most $2k$

- The first time an item is involved, it is a singleton set and so a parameter of the call to Union
- $k$ calls to Union involves at most $2k$ distinct parameters

Note that
**Lemma** Any initial sequence of $k$ Union's takes total time $O(k \log k)$

**Proof** Claim: The number of elements involved in the unions is at most $2k$

- The first time an item is involved, it is a singleton set and so a parameter of the call to Union
- $k$ calls to Union involves at most $2k$ distinct parameters

Note that

- After its first Union, an element is now in a set of size greater than 1—but the union of all of those sets can have size no greater than $2k$. 
A Surprising Fact: Continued

- **Union** only renames the vertices of the smaller set

- So every time an element is renamed, we have doubled the size of the set it is in.

- So if a vertex had its name changed \( d \) times, its set now has size at least \( 2^d \).

- But \( 2^d \leq 2^k \) so \( d \leq \log(2^k) \).

- Thus each element was renamed at most \( \log(2^k) \) times and at most \( 2^k \) elements were renamed.

- So total amount of renaming work is \( 2^k \cdot \log(2^k) = O(k \log k) \).
• *Union* only renames the vertices of the smaller set
• So every time an element is renamed, we have doubled the size of the set it is in.

Thus each element was renamed at most \( \log(2^k) \) times and at most \( 2^k \) elements were renamed.

So total amount of renaming work is \( 2^k \times \log(2^k) = O(k \log k) \).
A Surprising Fact: Continued

• *Union* only renames the vertices of the smaller set
• So every time an element is renamed, we have doubled the size of the set it is in.
• So if a vertex had its name changed $d$ times, its set now has size at least $2^d$. 

...
• *Union* only renames the vertices of the smaller set
• So every time an element is renamed, we have doubled the size of the set it is in.
• So if a vertex had its name changed $d$ times, its set now has size at least $2^d$.
• But $2^d \leq 2k$ so $d \leq \log(2k)$. 

*A Surprising Fact: Continued*
A Surprising Fact: Continued

- *Union* only renames the vertices of the smaller set
- So every time an element is renamed, we have doubled the size of the set it is in.
- So if a vertex had its name changed $d$ times, its set now has size at least $2^d$.
- But $2^d \leq 2k$ so $d \leq \log(2k)$.
- Thus each element was renamed at most $\log(2k)$ times and at most $2k$ elements were renamed
A Surprising Fact: Continued

- *Union* only renames the vertices of the smaller set
- So every time an element is renamed, we have doubled the size of the set it is in.
- So if a vertex had its name changed $d$ times, its set now has size at least $2^d$.
- But $2^d \leq 2k$ so $d \leq \log(2k)$.
- Thus each element was renamed at most $\log(2k)$ times and at most $2k$ elements were renamed.
- So total amount of renaming work is $2k \times \log(2k) = O(k \log k)$
Our First Union-Find Theorem

Theorem

Union-Find can be implemented so that MakeUnionFind takes $O(n)$ time, Find takes $O(1)$ time and any initial sequence of $k$ Unions takes $O(k \log k)$ time.

Corollary

Kruskal’s Algorithm can be implemented to run in $O(m \log m)$ time. [Repeatedly deleting from the heap is the bottleneck.]
Our First Union-Find Theorem

Theorem

Union-Find can be implemented so that MakeUnionFind takes $O(n)$ time, Find takes $O(1)$ time and any initial sequence of $k$ Unions takes $O(k \log k)$ time.

Theorem

Restated Union-Find can be implemented so that MakeUnionFind takes $O(n)$ time, Find takes $O(1)$ time and any initial sequence of $k$ Unions and Finds takes $O(k \log k)$ time.
Our First Union-Find Theorem

Theorem
Union-Find can be implemented so that MakeUnionFind takes \( O(n) \) time, Find takes \( O(1) \) time and any initial sequence of \( k \) Unions takes \( O(k \log k) \) time.

Theorem
Restated Union-Find can be implemented so that MakeUnionFind takes \( O(n) \) time, Find takes \( O(1) \) time and any initial sequence of \( k \) Unions and Finds takes \( O(k \log k) \) time.

Corollary
Kruskal’s Algorithm can be implemented to run in \( O(m \log m) \) time. [Repeatedly deleting from the heap is the bottleneck.]
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If $v \in X$, then $\text{UFSets}[v]$ needn't be $w$.

- $\text{UFSets}[]$ encodes a tree for each set $X$ in our partition
- The root of the tree for $X$ is name of $X$
- So $X$ has the name $x$ for some particular $x \in X$
- We'll only call $\text{Union}(x, y)$ when $x$ and $y$ are vertices that name sets
- $\text{Union}(x, y)$ creates a set named by larger of the two sets
- Thus, if the set named $x$ is larger, $\text{UFSets}[y] \leftarrow x$. (Set $y$ now points to set $x$)
- So, $\text{Union}$ now takes $O(1)$ time!
- Lists of vertices in each set no longer needed
An Improvement?

Can we rename fewer vertices during a Union?
Idea: Relax the naming strategy: If $v \in X_w$, then $UFSets[v]$ needn’t be $w$. 

- $UFSets[]$ encodes a tree for each set $X$ in our partition
- The root of the tree for $X$ is name of $X$
- So $X$ has the name $x$ for some particular $x \in X$
- We’ll only call $Union(x, y)$ when $x$ and $y$ are vertices that name sets
- $Union(x, y)$ creates a set named by larger of the two sets
- Thus, if the set named $x$ is larger, $UFSets[y] \leftarrow x$. (Set $y$ now points to set $x$)
- So, $Union$ now takes $O(1)$ time!
- Lists of vertices in each set no longer needed
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If $v \in X_w$, then $UFSets[v]$ needn’t be $w$.

- $UFSets[]$ encodes a tree for each set $X$ in our partition
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If $v \in X_w$, then $UFSets[v]$ needn’t be $w$.

- $UFSets[]$ encodes a tree for each set $X$ in our partition
- The root of the tree for $X$ is name of $X$
An Improvement?

Can we rename fewer vertices during a Union?
Idea: Relax the naming strategy: If \( v \in X_w \), then \( UFSets[v] \) needn’t be \( w \).

- \( UFSets[] \) encodes a tree for each set \( X \) in our partition
- The root of the tree for \( X \) is name of \( X \)
- So \( X \) has the name \( x \) for some particular \( x \in X \).
An Improvement?

Can we rename fewer vertices during a Union?
Idea: Relax the naming strategy: If \( v \in X_w \), then \( UFSets[v] \) needn’t be \( w \).

- \( UFSets[] \) encodes a tree for each set \( X \) in our partition
- The root of the tree for \( X \) is name of \( X \)
- So \( X \) has the name \( x \) for some particular \( x \in X \).
- **We’ll only call** \( Union(x, y) \) **when** \( x \) **and** \( y \) **are vertices that name sets**
An Improvement?

Can we rename fewer vertices during a Union?
Idea: Relax the naming strategy: If \( v \in X_w \), then \( UFSets[v] \) needn’t be \( w \).

• \( UFSets[] \) encodes a tree for each set \( X \) in our partition

• The root of the tree for \( X \) is name of \( X \)

• So \( X \) has the name \( x \) for some particular \( x \in X \).

• **We’ll only call** \( Union(x, y) \) **when** \( x \) **and** \( y \) **are vertices that name sets**

• \( Union(x, y) \) creates a set named by larger of the two sets
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If \( v \in X_w \), then \( UFSets[v] \) needn’t be \( w \).

- \( UFSets[] \) encodes a tree for each set \( X \) in our partition
- The root of the tree for \( X \) is name of \( X \)
- So \( X \) has the name \( x \) for some particular \( x \in X \).
- **We’ll only call** \( Union(x, y) \) **when** \( x \) **and** \( y \) **are vertices that name sets**
- \( Union(x, y) \) creates a set named by larger of the two sets
- Thus, if the set named \( x \) is larger, \( UFSets[y] \leftarrow x \). (Set \( y \) now points to set \( x \))
Can we rename fewer vertices during a Union?
Idea: Relax the naming strategy: If \( v \in X_w \), then \( UFSets[v] \) needn’t be \( w \).

- \( UFSets[] \) encodes a tree for each set \( X \) in our partition
- The root of the tree for \( X \) is name of \( X \)
- So \( X \) has the name \( x \) for some particular \( x \in X \).
- **We’ll only call** \( Union(x, y) \) **when** \( x \) **and** \( y \) **are vertices that name sets**
- \( Union(x, y) \) creates a set named by larger of the two sets
- Thus, if the set named \( x \) is larger, \( UFSets[y] \leftarrow x \). (Set \( y \) now points to set \( x \))
- So, \( Union \) now takes \( O(1) \) time!
An Improvement?

Can we rename fewer vertices during a Union?
Idea: Relax the naming strategy: If $v \in X_w$, then $UFSets[v]$ needn’t be $w$.

- $UFSets[]$ encodes a tree for each set $X$ in our partition
- The root of the tree for $X$ is name of $X$
- So $X$ has the name $x$ for some particular $x \in X$.
- **We'll only call $Union(x, y)$ when $x$ and $y$ are vertices that name sets**
- $Union(x, y)$ creates a set named by larger of the two sets
- Thus, if the set named $x$ is larger, $UFSets[y] \leftarrow x$. (Set $y$ now points to set $x$)
- So, Union now takes $O(1)$ time!
- Lists of vertices in each set no longer needed
An Improvement?

But how long does Find take now?

Observation

We can refer to the sets of the partition as trees.

• An element of $x \in S$ is the root of a tree iff $\text{UFSets}[x] = x$.

• The tree resulting from a $\text{Union}(X, Y)$ has height at most 1 greater than the heights of $X$ and $Y$.

• Thus the height of any tree of size $K$ is at most $\log K$.

• This is easy to prove by induction.

• Thus, for $x \in X$, $\text{Find}(x)$ now uses $\text{UFSets}$ array to find root of tree containing $x$ (that is, find $s \in X$ with $\text{UFSets}[s] = s$).

• Since tree has height at most $\log |X| \leq \log n$, Find takes at most $O(\log n)$ time.

Theorem

Version 2

$\text{Union-Find}$ can be implemented so that $\text{MakeUnionFind}$ takes $O(n)$ time, $\text{Union}$ takes $O(1)$ time, $\text{Find}$ takes $O(\log n)$ time, and any initial sequence of $k$ Unions and Finds takes $O(k \log k)$ time.
An Improvement?

But how long does Find take now?

Observation We can refer to the sets of the partition as trees.
An Improvement?

But how long does Find take now?

Observation We can refer to the sets of the partition as trees.

- An element of $x \in S$ is the root of a tree iff $UFSets[x] = x$. 

Theorem

Version 2

Union-Find can be implemented so that MakeUnionFind takes $O(n)$ time, Union takes $O(1)$ time, Find takes $O(\log n)$ time and any initial sequence of $k$ Unions and Finds takes $O(k \log k)$ time.
An Improvement?

But how long does Find take now?

**Observation** We can refer to the sets of the partition as trees.

- An element of \( x \in S \) is the root of a tree iff \( UFSets[x] = x \).
- The tree resulting from a \( Union(X, Y) \) has height at most 1 greater than the heights of \( X \) and \( Y \).
An Improvement?

But how long does Find take now?

**Observation** We can refer to the sets of the partition as trees.

- An element of \( x \in S \) is the root of a tree iff \( UFSets[x] = x \).
- The tree resulting from a \( Union(X, Y) \) has height at most 1 greater than the heights of \( X \) and \( Y \).
- Thus the height of any tree of size \( K \) is at most \( \log K \).
An Improvement?

But how long does Find take now?

**Observation** We can refer to the sets of the partition as trees.

- An element of $x \in S$ is the root of a tree iff $\text{UFSets}[x] = x$.
- The tree resulting from a $\text{Union}(X, Y)$ has height at most 1 greater than the heights of $X$ and $Y$.
- Thus the height of any tree of size $K$ is at most $\log K$
  - This is easy to prove by induction
**An Improvement?**

But how long does Find take now?

**Observation** We can refer to the sets of the partition as trees.

- An element of $x \in S$ is the root of a tree iff $UFSets[x] = x$.
- The tree resulting from a $Union(X, Y)$ has height at most 1 greater than the heights of $X$ and $Y$.
- Thus the height of any tree of size $K$ is at most $\log K$
  - This is easy to prove by induction
- Thus, for $x \in X$, $Find(x)$ now uses $UFSets$ array to find root of tree containing $x$ (that is, find $s \in X$ with $UFSets[s] = s$)
An Improvement?

But how long does Find take now?

Observation We can refer to the sets of the partition as trees.

- An element of $x \in S$ is the root of a tree iff $UFSets[x] = x$.
- The tree resulting from a $Union(X, Y)$ has height at most 1 greater than the heights of $X$ and $Y$.
- Thus the height of any tree of size $K$ is at most $\log K$
  - This is easy to prove by induction
- Thus, for $x \in X$, $Find(x)$ now uses UFSets array to find root of tree containing $x$ (that is, find $s \in X$ with $UFSets[s] = s$)
- Since tree has height at most $\log |X| \leq \log n$, Find takes at most $O(\log n)$ time
An Improvement?

But how long does Find take now?

**Observation** We can refer to the sets of the partition as trees.

- An element of $x \in S$ is the root of a tree iff $UFSets[x] = x$.
- The tree resulting from a $Union(X, Y)$ has height at most 1 greater than the heights of $X$ and $Y$.
- Thus the height of any tree of size $K$ is at most $\log K$
  - This is easy to prove by induction
- Thus, for $x \in X$, $Find(x)$ now uses UFSets array to find root of tree containing $x$ (that is, find $s \in X$ with $UFSets[s] = s$)
- Since tree has height at most $\log |X| \leq \log n$, Find takes at most $O(\log n)$ time

**Theorem**

**Version 2** Union-Find can be implemented so that $MakeUnionFind$ takes $O(n)$ time, $Union$ takes $O(1)$ time, $Find$ takes $O(\log n)$ time and any initial sequence of $k$ Unions and Finds takes $O(k \log k)$ time.
But wait! There’s more...

- Maybe using sizes of trees to determine how $Union$ merges isn’t optimal
But wait! There’s more...

- Maybe using sizes of trees to determine how *Union* merges isn’t optimal
- We could use tree height instead: merge shallower tree into deeper tree

Theorem: Using path compression, any initial sequence of \( m \) Union and Find operations on \( n \) items after a MakeUnionFind can be carried out in \( O(n + m \log^* n) \) time.
But wait! There’s more...

- Maybe using sizes of trees to determine how \textit{Union} merges isn’t optimal
- We could use tree height instead: merge shallower tree into deeper tree
- Maintains fact that height is at most $\log(size)$
But wait! There’s more...

• Maybe using sizes of trees to determine how Union merges isn’t optimal
• We could use tree height instead: merge shallower tree into deeper tree
• Maintains fact that height is at most log(size)
• Even better: Every time we do a Find(x), redirect all nodes traversed in tree to root
But wait! There’s more...

- Maybe using sizes of trees to determine how Union merges isn’t optimal
- We could use tree height instead: merge shallower tree into deeper tree
- Maintains fact that height is at most $\log(size)$
- Even better: Every time we do a $Find(x)$, redirect all nodes traversed in tree to root
- This is called path compression
But wait! There’s more...

- Maybe using sizes of trees to determine how Union merges isn’t optimal
- We could use tree height instead: merge shallower tree into deeper tree
- Maintains fact that height is at most log(size)
- Even better: Every time we do a Find(x), redirect all nodes traversed in tree to root
- This is called path compression

**Theorem**

Using path compression, any initial sequence of m Union and Find operations on n items after a MakeUnionFind can be carried out in $O(n + m \log^* n)$ time.
But What is This $\log^* n$ Function?

Definition

For any base $b > 1$, $\log^* (n)$ is the number of times $\log_b$ must be repeatedly applied to $n$ before the result is less than 1. Precisely:

$\log^* (n) = \begin{cases} 
0 & \text{if } n \leq 1 \\
1 + \log^* (\log n) & \text{if } n > 1 
\end{cases}$

$\log^* n$ grows very slowly....

$1, 2, 4, 8, 16, 32, 64, 128,$...
But What is This $\log^* n$ Function?

**Definition**

For any base $b > 1$, $\log^*_b(n)$ is the number of times $\log_b$ must be repeatedly applied to $n$ before the result is less than 1. Precisely:

$$\log^*(n) = \begin{cases} 
0 & \text{if } n \leq 1 \\
1 + \log^*(\log n) & \text{if } n > 1
\end{cases}$$
**But What is This \( \log^* n \) Function?**

**Definition**
For any base \( b > 1 \), \( \log_b^*(n) \) is the number of times \( \log_b \) must be repeatedly applied to \( n \) before the result is less than 1. Precisely:

\[
\log^*(n) = \begin{cases} 
0 & \text{if } n \leq 1 \\
1 + \log^*(\log n) & \text{if } n > 1 
\end{cases}
\]

\( \log^* n \) grows very slowly....
But What is This $\log^* n$ Function?

**Definition**

For any base $b > 1$, $\log_b^*(n)$ is the number of times $\log_b$ must be repeatedly applied to $n$ before the result is less than 1. Precisely:

$$\log^*(n) = \begin{cases} 
0 & \text{if } n \leq 1 \\
1 + \log^*(\log n) & \text{if } n > 1
\end{cases}$$

$\log^* n$ grows very slowly....

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>2</th>
<th>$4 = 2^2$</th>
<th>16 = $2^4$</th>
<th>65,536 = $2^{16}$</th>
<th>$2^{65,536}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log^*(n)$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>