Implementing Kruskal’s Algorithm: Union-Find

Algorithm Design & Analysis

Spring 2019
Outline

Many MCST Algorithms
  The Problem
  Kruskal’s Algorithm
  Moderate Greed: Prim’s Algorithm
  Reverse Greed: Reverse-Delete Algorithm

Implementing Kruskal’s Algorithm
  Priority Queue Implementation
  Union-Find Implementations
Real Life Asymptotic Analysis
Outline

Many MCST Algorithms

The Problem
Kruskal’s Algorithm
Moderate Greed: Prim’s Algorithm
Reverse Greed: Reverse-Delete Algorithm

Implementing Kruskal’s Algorithm
Priority Queue Implementation
Union-Find Implementations
Minimum-Cost Spanning Trees

Figure: A Graph $G$ with Positive Edge-Weights
Minimum-Cost Spanning Trees

Figure: A Min-Cost Spanning Tree for $G$
Outline

Many MCST Algorithms
- The Problem
- Kruskal’s Algorithm
- Moderate Greed: Prim’s Algorithm
- Reverse Greed: Reverse-Delete Algorithm

Implementing Kruskal’s Algorithm
- Priority Queue Implementation
- Union-Find Implementations
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 2 Kruskal’s Algorithm
Many MCST Algorithms

Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 3 Kruskal’s Algorithm
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 4 Kruskal’s Algorithm

procedure \texttt{Kruskal}(G, c()) // G = (V, E) is connected
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 5 Kruskal’s Algorithm

```
procedure KRUSCAL(G, c()) // G = (V, E) is connected
    T ← (V, ∅) // The eventual MCST
```
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

**Algorithm 6** Kruskal’s Algorithm

```
procedure KRUSKAL(G, c()) // G = (V, E) is connected
    T ← (V, ∅) // The eventual MCST
    F ← E
```

Implementing Kruskal’s Algorithm
Many MCST Algorithms

Implementing Kruskal’s Algorithm

Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 7 Kruskal’s Algorithm

procedure $\text{KRUSKAL}(G, c())$ // $G = (V, E)$ is connected

$T \leftarrow (V, \emptyset)$ // The eventual MCST

$F \leftarrow E$

while $|E(T)| < |V| - 1$ do
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

**Algorithm 8** Kruskal’s Algorithm

```plaintext
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
```
Many MCST Algorithms

Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 9 Kruskal’s Algorithm

procedure \textsc{Kruskal}(G, c()) \-comment{G = (V, E) is connected}
    \begin{align*}
    T & \leftarrow (V, \emptyset) \-comment{The eventual MCST} \\
    F & \leftarrow E \\
    \textbf{while} & \ |E(T)| < |V| - 1 \ \textbf{do} \\
    & \text{Remove cheapest edge } e \in F \text{ from } F \\
    & \textbf{if } T + \{e\} \text{ does not contain a cycle \textbf{then}}
    \end{align*}
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 10 Kruskal’s Algorithm

procedure KRUSKAL($G, c()$)  // $G = (V, E)$ is connected
    $T \leftarrow (V, \emptyset)$  // The eventual MCST
    $F \leftarrow E$
    while $|E(T)| < |V| - 1$ do
        Remove cheapest edge $e \in F$ from $F$
        if $T + \{e\}$ does not contain a cycle then
            Add $e$ to $T$
    end procedure
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 11 Kruskal’s Algorithm

procedure $\text{Kruskal}(G, c())$ // $G = (V, E)$ is connected
$T \leftarrow (V, \emptyset)$ // The eventual MCST
$F \leftarrow E$

while $|E(T)| < |V| - 1$ do
    Remove cheapest edge $e \in F$ from $F$
    if $T + \{e\}$ does not contain a cycle then
        Add $e$ to $T$
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 12 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
Maximum Greed: Kruskal’s Algorithm

Idea: Add cheapest remaining edge that don’t create a cycle

Algorithm 13 Kruskal’s Algorithm

procedure \textsc{Kruskal}(G, c()) \-comment{G = (V, E) is connected}
\hspace{1em} T \leftarrow (V, \emptyset) \-comment{The eventual MCST}
T \leftarrow E
\hspace{1em} \textbf{while} |E(T)| < |V| - 1 \hspace{1em} \textbf{do}
\hspace{2em} \text{Remove cheapest edge } e \in F \text{ from } F
\hspace{2em} \textbf{if } T + \{e\} \text{ does not contain a cycle \textbf{then}}
\hspace{3em} \text{Add } e \text{ to } T
\hspace{1em} \textbf{end procedure}
Outline

Many MCST Algorithms
The Problem
Kruskal’s Algorithm
Moderate Greed: Prim’s Algorithm
Reverse Greed: Reverse-Delete Algorithm

Implementing Kruskal’s Algorithm
Priority Queue Implementation
Union-Find Implementations
Moderate Greed: Prim’s Algorithm

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

Implementing Kruskal’s Algorithm

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 15

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

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 16 Prim’s Algorithm
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 17 Prim’s Algorithm

procedure PRIM$(G, c())$ // $G = (V, E)$ is connected
Many MCST Algorithms

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 18 Prim’s Algorithm

\begin{verbatim}
procedure PRIM(G, c()) // G = (V, E) is connected
  Select some $v \in V$
\end{verbatim}
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 19** 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
```
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 20 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
```
Moderate Greed: Prim’s Algorithm

Here $T$ is a 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 21 Prim’s Algorithm

```
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 $e \in E(V', V - V')$
```
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 22 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 \( T \)
```
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 23 Prim’s Algorithm

```
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 24** Prim’s Algorithm: More Detail
Moderate Greed: Prim’s Algorithm

Algorithm 25 Prim’s Algorithm: More Detail

procedure \textsc{Prim}(G, c()) \hfill G = (V, E) is connected
Moderate Greed: Prim’s Algorithm

Algorithm 26 Prim’s Algorithm: More Detail

procedure PRIM(G, c()) // G = (V, E) is connected
Select some v ∈ V; V' ← {v}
**Moderate Greed: Prim’s Algorithm**

**Algorithm 27** Prim’s Algorithm: More Detail

**procedure** PRIM($G, c()$) // $G = (V, E)$ is connected

Select some $v \in V$; $V' \leftarrow \{v\}$

$T \leftarrow (V', \emptyset)$ // The eventual MCST
Many MCST Algorithms

Implementing Kruskal’s Algorithm

Moderate Greed: Prim’s Algorithm

Algorithm 28 Prim’s Algorithm: More Detail

procedure PRIM(G, c()) // G = (V, E) is connected
Select some v ∈ V; V′ ← {v}
T ← (V′, ∅) // The eventual MCST
C ← E(V′, V − V′)
Moderate Greed: Prim’s Algorithm

Algorithm 29 Prim’s Algorithm: More Detail

procedure PRIM(G, c()) // G = (V, E) is connected
    Select some \( v \in V \); \( V' \leftarrow \{v\} \)
    \( T \leftarrow (V', \emptyset) \) // The eventual MCST
    \( C \leftarrow E(V', V - V') \)
    // \( C \) will always contain \( E(V', V - V') \)

Many MCST Algorithms

Implementing Kruskal’s Algorithm
Algorithm 30 Prim’s Algorithm: More Detail

procedure \text{PRIM}(G, c()) // \( G = (V, E) \) is connected 
\begin{align*}
&\text{Select some } v \in V; \ V' \leftarrow \{v\} \\
&T \leftarrow (V', \emptyset) // \text{The eventual MCST} \\
&C \leftarrow E(V', V - V') \\
&\text{// } C \text{ will always contain } E(V', V - V') \\
\text{while } |E(T)| < |V| - 1 \text{ do}
\end{align*}
**Moderate Greed: Prim’s Algorithm**

**Algorithm 31** Prim’s Algorithm: More Detail

```plaintext
procedure PRIM(G, c()) // G = (V, E) is connected
    Select some v ∈ V; V' ← {v}
    T ← (V', ∅) // The eventual MCST
    C ← E(V', V − V')
    // C will always contain E(V', V − V')
    while |E(T)| < |V| − 1 do
        Select cheapest edge e ∈ C
```

Algorithm 32 Prim’s Algorithm: More Detail

procedure $\text{PRIM}(G, c())$ // $G = (V, E)$ is connected
    Select some $v \in V$; $V' \leftarrow \{v\}$
    $T \leftarrow (V', \emptyset)$ // The eventual MCST
    $C \leftarrow E(V', V - V')$
    // $C$ will always contain $E(V', V - V')$
    while $|E(T)| < |V| - 1$ do
        Select cheapest edge $e \in C$
        if $e \in E(V', V - V')$ then

Moderate Greed: Prim’s Algorithm

Algorithm 33 Prim’s Algorithm: More Detail

procedure \texttt{PRIM}(G, c()) // $G = (V, E)$ is connected
    Select some $v \in V$; $V' \leftarrow \{v\}$
    $T \leftarrow (V', \emptyset)$ // The eventual MCST
    $C \leftarrow E(V', V - V')$
    // $C$ will always contain $E(V', V - V')$
while $|E(T)| < |V| - 1$ do
    Select cheapest edge $e \in C$
    if $e \in E(V', V - V')$ then
        // Let $e = \{u, v\}$, where $u \in V'$ and $v \in V - V'$

Moderate Greed: Prim’s Algorithm

Algorithm 34 Prim’s Algorithm: More Detail

procedure \textsc{Prim}(G, c()) // $G = (V, E)$ is connected

Select some $v \in V$; $V' \leftarrow \{v\}$

$T \leftarrow (V', \emptyset)$ // The eventual MCST

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

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

while $|E(T)| < |V| - 1$ do

Select cheapest edge $e \in C$

if $e \in E(V', V - V')$ then

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

Add $v$ to $T$; Add $e$ to $T$
Algorithm 35 Prim’s Algorithm: More Detail

procedure PRIM(G, c()) // G = (V, E) is connected
    Select some $v \in V$; $V' \leftarrow \{v\}$
    $T \leftarrow (V', \emptyset)$ // The eventual MCST
    $C \leftarrow E(V', V - V')$
    // $C$ will always contain $E(V', V - V')$
    while $|E(T)| < |V| - 1$ do
        Select cheapest edge $e \in C$
        if $e \in E(V', V - V')$ then
            // Let $e = \{u, v\}$, where $u \in V'$ and $v \in V - V'$
            Add $v$ to $T$; Add $e$ to $T$
        for neighbors $w$ of $v$ do
Moderate Greed: Prim’s Algorithm

Algorithm 36 Prim’s Algorithm: More Detail

procedure PRIM(G, c()) // G = (V, E) is connected
    Select some v ∈ V; V’ ← {v}
    T ← (V’, ∅) // The eventual MCST
    C ← E(V’, V − V’)
    // C will always contain E(V’, V − V’)
    while |E(T)| < |V| − 1 do
        Select cheapest edge e ∈ C
        if e ∈ E(V’, V − V’) then
            // Let e = {u, v}, where u ∈ V’ and v ∈ V − V’
            Add v to T; Add e to T
            for neighbors w of v do
                if w ∈ V − V’ then add {v, w} to C
```
Moderate Greed: Prim’s Algorithm

Algorithm 37 Prim’s Algorithm: More Detail

\begin{verbatim}
procedure PRIM(G, c()) // G = (V, E) is connected
    Select some v ∈ V; V' ← {v}
    T ← (V', ∅) // The eventual MCST
    C ← E(V', V − V')
    // C will always contain E(V', V − V')
    while |E(T)| < |V| − 1 do
        Select cheapest edge e ∈ C
        if e ∈ E(V', V − V') then
            // Let e = {u, v}, where u ∈ V' and v ∈ V − V'
            Add v to T; Add e to T
            for neighbors w of v do
                if w ∈ V − V' then add {v, w} to C
        end procedure
\end{verbatim}
Moderate Greed: Prim’s Algorithm

**Algorithm 38 Prim’s Algorithm: More Detail**

```plaintext
procedure PRIM(G, c()) // G = (V, E) is connected
  Select some v ∈ V; V' ← {v}
  T ← (V', ∅) // The eventual MCST
  C ← E(V', V - V')
  // C will always contain E(V', V - V')
  while |E(T)| < |V| - 1 do
    Select cheapest edge e ∈ C
    if e ∈ E(V', V - V') then
      // Let e = {u, v}, where u ∈ V' and v ∈ V - V'
      Add v to T; Add e to T
      for neighbors w of v do
        if w ∈ V - V' then add {v, w} to C
    end if
  end while
end procedure
```
Moderate Greed: Prim’s Algorithm

Algorithm 39 Prim’s Algorithm: More Detail

procedure \textsc{Prim}(G, c()) // $G = (V, E)$ is connected
  Select some $v \in V$; $V' \leftarrow \{v\}$
  $T \leftarrow (V', \emptyset)$ // The eventual MCST
  $C \leftarrow E(V', V - V')$
  // $C$ will always contain $E(V', V - V')$
  while $|E(T)| < |V| - 1$ do
    Select cheapest edge $e \in C$
    if $e \in E(V', V - V')$ then
      // Let $e = \{u, v\}$, where $u \in V'$ and $v \in V - V'$
      Add $v$ to $T$; Add $e$ to $T$
      for neighbors $w$ of $v$ do
        if $w \in V - V'$ then add $\{v, w\}$ to $C$
Moderate Greed: Prim’s Algorithm

**Algorithm 40 Prim’s Algorithm: More Detail**

```plaintext
procedure PRIM(G, c()) // G = (V, E) is connected
    Select some v ∈ V; V′ ← {v}
    T ← (V′, ∅) // The eventual MCST
    C ← E(V′, V − V′)
    // C will always contain E(V′, V − V′)
    while |E(T)| < |V| − 1 do
        Select cheapest edge e ∈ C
        if e ∈ E(V′, V − V′) then
            // Let e = {u, v}, where u ∈ V′ and v ∈ V − V'
            Add v to T; Add e to T
            for neighbors w of v do
                if w ∈ V − V′ then add {v, w} to C
        end if
    end while
end procedure
```
The Implementation

Notes
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 $V'$
Many MCST Algorithms

Implementing Kruskal’s Algorithm

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:

Loop runs 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 total number of heap ops is $O(|E|)$; each takes time $O(\log |E|)$.

Note:

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

So Prim runs in time $O(|V| + |E| \log |V|)$ ($O(|E| \log |V|)$ if $G$ is connected)
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 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: Loop runs 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: Loop runs 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 total number of heap ops is $O(|E|)$; 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:** Loop runs 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 total number of heap ops is $O(|E|)$; each takes time $O(\log |E|)$.

**Note:** Since $|E| \in O(|V|^2)$, $\log |E| \in O(\log |V|)$
Many MCST Algorithms

Implementing Kruskal’s Algorithm

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: Loop runs 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 total number of heap ops is $O(|E|)$; each takes time $O(\log |E|)$.

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

So Prim runs in time $O(|V| + |E| \log |V|)$ ($O(|E| \log |V|)$ if $G$ is connected)
Outline

Many MCST Algorithms
  The Problem
  Kruskal’s Algorithm
  Moderate Greed: Prim’s Algorithm
  Reverse Greed: Reverse-Delete Algorithm

Implementing Kruskal’s Algorithm
  Priority Queue Implementation
  Union-Find Implementations
Many MCST Algorithms

Implementing Kruskal’s Algorithm

Reverse-Delete Algorithm

We can also construct an MCST by throwing away all of the most expensive edges.
Reverse-Delete Algorithm

We can also construct an MCST by throwing away all of the most expensive edges.

**Algorithm 42 Reverse-Delete Algorithm**
Reverse-Delete Algorithm

We can also construct an MCST by throwing away all of the most expensive edges.

Algorithm 43 Reverse-Delete Algorithm

procedure \textsc{ReverseDelete}(G, c()) // G = (V, E) is connected
Reverse-Delete Algorithm

We can also construct an MCST by throwing away all of the most expensive edges.

Algorithm 44 Reverse-Delete Algorithm

```
procedure ReverseDelete(G, c()) // G = (V, E) is connected
    while |E(G)| > |V| − 1 do
        Select most expensive edge e ∈ G that does not disconnect G
        Remove e from G
    end procedure
```

When might you ever want to use this algorithm?
Reverse-Delete Algorithm

We can also construct an MCST by throwing away all of the most expensive edges.

**Algorithm 45** Reverse-Delete Algorithm

procedure `REVERSE_DELETE(G, c())` // \( G = (V, E) \) is connected

\[ \text{while } |E(G)| > |V| - 1 \text{ do} \]

Select most expensive edge \( e \in G \) that does not disconnect \( G \)
Many MCST Algorithms

Implementing Kruskal’s Algorithm

Reverse-Delete Algorithm

We can also construct an MCST by throwing away all of the most expensive edges.

Algorithm 46 Reverse-Delete Algorithm

procedure REVERSEDELETE(G, c())  // G = (V, E) is connected
    while |E(G)| > |V| − 1 do
        Select most expensive edge e ∈ G that does not disconnect G
        Remove e from G
Reverse-Delete Algorithm

We can also construct an MCST by throwing away all of the most expensive edges.

Algorithm 47 Reverse-Delete Algorithm

procedure REVERSEDELETE \(G, c())\)  
// \(G = (V, E)\) is connected

while \(|E(G)| > |V| - 1\) do

Select most expensive edge \(e \in G\) that does not disconnect \(G\)

Remove \(e\) from \(G\)

end procedure
Many MCST Algorithms

Implementing Kruskal’s Algorithm

We can also construct an MCST by throwing away all of the most expensive edges.

**Algorithm 48** Reverse-Delete Algorithm

```plaintext
procedure REVERSEDELETE(G, c()) // G = (V, E) is connected
    while |E(G)| > |V| − 1 do
        Select most expensive edge e ∈ G that does not disconnect G
        Remove e from G
    end procedure
```

When might you ever want to use this algorithm?
Outline

Many MCST Algorithms
  The Problem
  Kruskal’s Algorithm
  Moderate Greed: Prim’s Algorithm
  Reverse Greed: Reverse-Delete Algorithm

Implementing Kruskal’s Algorithm
  Priority Queue Implementation
  Union-Find Implementations
Extreme greed: Kruskal’s Algorithm

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

Algorithm 49

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
Extreme greed: Kruskal’s Algorithm

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

Algorithm 50 Kruskal’s Algorithm
Extreme greed: Kruskal’s Algorithm

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

Algorithm 51 Kruskal’s Algorithm

procedure KRUSKAL(G, c()) // G = (V, E) is connected
Extreme greed: Kruskal’s Algorithm

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

Algorithm 52 Kruskal’s Algorithm

procedure \textsc{Kruskal}(G, c()) // G = (V, E) is connected
    \hspace{1em} T \leftarrow (V, \emptyset) // The eventual MCST
Extreme greed: Kruskal’s Algorithm

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

Algorithm 53 Kruskal’s Algorithm

procedure KRUSKAL(G, c()) // G = (V, E) is connected
T ← (V, ∅) // The eventual MCST
F ← E
Extreme greed: Kruskal’s Algorithm

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

Algorithm 54 Kruskal’s Algorithm

procedure \textsc{Kruskal}(G, c()) // \( G = (V, E) \) is connected
\[ T \leftarrow (V, \emptyset) \] // The eventual MCST
\[ F \leftarrow E \]
while \(|E(T)| < |V| - 1\) do
Extreme greed: Kruskal’s Algorithm

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

**Algorithm 55** Kruskal’s Algorithm

```plaintext
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
```

Implementing Kruskal’s Algorithm
Extreme greed: Kruskal’s Algorithm

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

**Algorithm 56** Kruskal’s Algorithm

```plaintext
procedure \text{Kruskal}(G, c()) // G = (V, E) is connected
    T ← (V, ∅) // The eventual MCST
    F ← E
    \text{while} |E(T)| < |V| - 1 \text{ do}
        Remove cheapest edge e ∈ F from F
        if T + \{e\} does not contain a cycle then
```
**Extreme greed: Kruskal’s Algorithm**

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

**Algorithm 57** 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
```
**Extreme greed: Kruskal’s Algorithm**

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

---

**Algorithm 58 Kruskal’s Algorithm**

```plaintext
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
```
Extreme greed: Kruskal’s Algorithm

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

Algorithm 59 Kruskal’s Algorithm

procedure \textsc{Kruskal}(G, c()) // G = (V, E) is connected
\begin{align*}
T & \leftarrow (V, \emptyset) // The eventual MCST \\
F & \leftarrow E \\
\text{while } |E(T)| < |V| - 1 \text{ do} \\
& \quad \text{Remove cheapest edge } e \in F \text{ from } F \\
& \quad \text{if } T + \{e\} \text{ does not contain a cycle then} \\
& \quad \quad \text{Add } e \text{ to } T \\
\end{align*}
end procedure
The Implementation

To implement Kruskal’s Algorithm, we want efficient methods to
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
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 $T$
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.
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.
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 60** 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 61** 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 62 Bottom-up Heapify

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.
Bottom-Up Heapify
How to turn an array into a heap quickly

Algorithm 63 Bottom-up Heapify

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 \( \text{HeapifyDown}(H, i) \), \( H[i + 1 \ldots n] \) satisfies the heap property.
**Bottom-Up Heapify**

How to turn an array into a heap quickly

**Algorithm 64 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
Time Complexity of Bottom-Up Heapify

- A binary tree of height $h$ has at most $n = 2^{h+1} - 1$ nodes:
**Time Complexity of Bottom-Up Heapify**

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

  $$n = \sum_{i=0}^{h} \# \text{ of elements at depth } i = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$
**Time Complexity of Bottom-Up Heapify**

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

- 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
  \[
  \sum_{d=0}^{d=h} (\# \text{ of nodes at depth } d) \times (\# \text{ of swaps}) \leq \sum_{d=0}^{d=h} (h - d)2^d
  \]
  \[
  \sum_{d=0}^{d=h} (h - d)2^d = 2^{h+1} - h - 2 \leq 2n - \lfloor \log n \rfloor - 2 \in O(n)
  \]
Outline

Many MCST Algorithms

The Problem
Kruskal’s Algorithm
Moderate Greed: Prim’s Algorithm
Reverse Greed: Reverse-Delete Algorithm

Implementing Kruskal’s Algorithm

Priority Queue Implementation
Union-Find Implementations
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 \( T \).
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 $u$. 
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)
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
A First Union-Find Structure

**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
A First Union-Find Structure

Union-Find Data Structure

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

  \[ \text{MakeUnionFind}(): \text{Initialize the structure} \]
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 $x$
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 $x$
  
  $\text{Union}(X, Y)$: 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 65

procedure Kruskal (G, c) // $G = (V, E)$ is connected

T ← (V, ∅) // The eventual MCST

F ← E

UF ← MakeUnionFind (V)

while |E(T)| < |V| − 1 do

Remove cheapest edge $e = \{u, v\} \in F$ from $F$

uName = UF.Find(u);
vName = UF.Find(v);

if uName ≠ vName then

Add $e$ to $T$

UF.Union(uName, vName)

end procedure
Extreme greed: Kruskal’s Algorithm

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

Algorithm 66 Kruskal’s Algorithm
Extreme greed: Kruskal’s Algorithm

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

Algorithm 67 Kruskal’s Algorithm

procedure \textsc{Kruskal}(G, c()) // $G = (V, E)$ is connected
Extreme greed: Kruskal’s Algorithm

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

Algorithm 68 Kruskal’s Algorithm

\[\text{procedure } \text{KRUSKAL}(G, c()) // G = (V, E) is connected}\]
\[T \leftarrow (V, \emptyset) // \text{The eventual MCST}\]
Extreme greed: Kruskal’s Algorithm

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

Algorithm 69 Kruskal’s Algorithm

procedure KRUSKAL(G, c()) // G = (V, E) is connected
    T ← (V, ∅) // The eventual MCST
    F ← E

Extreme greed: Kruskal’s Algorithm

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

Algorithm 70 Kruskal’s Algorithm

procedure \textsc{Kruskal}(G, c()) \tcp*{G = (V, E) is connected}
\quad T \leftarrow (V, \emptyset) \tcp*{The eventual MCST}
\quad F \leftarrow E
\quad UF \leftarrow \text{MakeUnionFind}(V)
Extreme greed: Kruskal’s Algorithm

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

Algorithm 71 Kruskal’s Algorithm

procedure Kruskal\((G, c())\) // \(G = (V, E)\) is connected
\[T \leftarrow (V, \emptyset)\] // The eventual MCST
\[F \leftarrow E\]
\[UF \leftarrow \text{MakeUnionFind}(V)\]
while |\(E(T)\)| < |\(V\)| − 1 do
Extreme greed: Kruskal’s Algorithm

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

Algorithm 72 Kruskal’s Algorithm

procedure KRUSKAL(G, c()) // G = (V, E) is connected
\[ T \leftarrow (V, \emptyset) \] // The eventual MCST
\[ F \leftarrow E \]
\[ UF \leftarrow \text{MakeUnionFind}(V) \]
while \( |E(T)| < |V| - 1 \) do
\[ \text{Remove cheapest edge } e = \{u, v\} \in F \text{ from } F \]
Extreme greed: Kruskal’s Algorithm

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

Algorithm 73 Kruskal’s Algorithm

procedure \textsc{Kruskal}(G, c()) // $G = (V, E)$ is connected
\hspace*{1em} $T \leftarrow (V, \emptyset)$ // The eventual MCST
\hspace*{1em} $F \leftarrow E$
\hspace*{1em} $UF \leftarrow \text{MakeUnionFind}(V)$
\hspace*{1em} while $|E(T)| < |V| - 1$ do
\hspace*{2em} Remove cheapest edge $e = \{u, v\} \in F$ from $F$
\hspace*{2em} $uName = UF.Find(u)$; $vName = UF.Find(v)$
Extreme greed: Kruskal’s Algorithm

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

Algorithm 74 Kruskal’s Algorithm

procedure KRUSKAL(G, c()) // G = (V, E) is connected
    T ← (V, ∅) // The eventual MCST
    F ← E
    UF ← MakeUnionFind(V)
while |E(T)| < |V| − 1 do
    Remove cheapest edge e = {u, v} ∈ F from F
    uName = UF.Find(u); vName = UF.Find(v)
    if uName ≠ vName then
Extreme greed: Kruskal’s Algorithm

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

Algorithm 75 Kruskal’s Algorithm

procedure Kruskal(G, c()) // G = (V, E) is connected
    T ← (V, ∅) // The eventual MCST
    F ← E
    UF ← MakeUnionFind(V)
while |E(T)| < |V| − 1 do
    Remove cheapest edge e = {u, v} ∈ F from F
    uName = UF.Find(u); vName = UF.Find(v)
    if uName ≠ vName then
        Add e to T
Extreme greed: Kruskal’s Algorithm

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

Algorithm 76 Kruskal’s Algorithm

procedure $\text{Kruskal}(G, c())$ // $G = (V, E)$ is connected
    $T \leftarrow (V, \emptyset)$ // The eventual MCST
    $F \leftarrow E$
    $UF \leftarrow \text{MakeUnionFind}(V)$
while $|E(T)| < |V| - 1$ do
    Remove cheapest edge $e = \{u, v\} \in F$ from $F$
    $uName = UF.\text{Find}(u)$; $vName = UF.\text{Find}(v)$
    if $uName \neq vName$ then
        Add $e$ to $T$
        $UF.\text{Union}(uName, vName)$
Extreme greed: Kruskal’s Algorithm

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

Algorithm 77 Kruskal’s Algorithm

procedure \textsc{Kruskal}(G, c()) // $G = (V, E)$ is connected
\begin{align*}
T &\leftarrow (V, \emptyset) // \text{The eventual MCST} \\
F &\leftarrow E \\
UF &\leftarrow \text{MakeUnionFind}(V) \\
\text{while } |E(T)| < |V| - 1 \text{ do} \\
\quad \text{Remove cheapest edge } e = \{u, v\} \in F \text{ from } F \\
\quad uName &= UF.\text{Find}(u); \ vName = UF.\text{Find}(v) \\
\quad \text{if } uName \neq vName \text{ then} \\
\quad \quad \text{Add } e \text{ to } T \\
\quad \quad UF.\text{Union}(uName, vName)
\end{align*}
Extreme greed: Kruskal’s Algorithm

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

Algorithm 78 Kruskal’s Algorithm

procedure $\text{Kruskal}(G, c())$ // $G = (V, E)$ is connected
$T \leftarrow (V, \emptyset)$ // The eventual MCST
$F \leftarrow E$
$UF \leftarrow \text{MakeUnionFind}(V)$
while $|E(T)| < |V| - 1$ do
    Remove cheapest edge $e = \{u, v\} \in F$ from $F$
    $uName = UF.\text{Find}(u)$; $vName = UF.\text{Find}(v)$
    if $uName \neq vName$ then
        Add $e$ to $T$
        $UF.\text{Union}(uName, vName)$
end procedure
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

- $\text{MakeUnionFind}()$ 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

- *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
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

- \textit{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

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

- \textit{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

- \( \text{MakeUnionFind}() \) 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
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
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 the 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

- \textit{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

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

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

- \textit{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 79 Union()

procedure UNION(i, j) // i and j are set names
  // Assume |S_i| ≤ |S_j|; if not, swap i and j
  for x ∈ S_i do
    UFSets[x] = j
  end procedure
A Surprising Fact

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 $2k$
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
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
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

- 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
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.
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 \(2^d\).
**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 $d \leq \log(2k)$. 
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.
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.]
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.]
To be continued in our next class....