# Minimum Spanning Trees

CS 2860: Algorithms and Complexity

Magnus Wahlström and Gregory Gutin

February 24, 2018

# Minimum Spanning Trees

# Minimum spanning trees

- Input: Graph $G = (V, E)$, edge weights/costs $w(e)$
- Spanning tree: A tree $T = (V, F)$ with edges from $E$ that connects every vertex of $G$
- Minimum spanning tree: A spanning tree $T$ of minimum total cost

$$\sum_{e \in E(T)} w(e)$$

# Minimum spanning trees

- Input: Graph $G = (V, E)$, edge weights/costs $w(e)$
- Spanning tree: A tree $T = (V, F)$ with edges from $E$ that connects every vertex of $G$
- Minimum spanning tree: A spanning tree $T$ of minimum total cost

$$\sum_{e \in E(T)} w(e)$$

# Applications

- Historical example: Electrifying Moravia
  - Region of present Czech republic
  - Build power lines to connect villages to electric grid
  - Different connections have different costs to build (long/short distance, good/bad terrain)
  - Borůvka's algorithm, 1926
- Computer networks, road networks, ...

# Applications

- Historical example: Electrifying Moravia
  - Region of present Czech republic
  - Build power lines to connect villages to electric grid
  - Different connections have different costs to build (long/short distance, good/bad terrain)
  - Borůvka's algorithm, 1926
- Computer networks, road networks, …

# Applications

- Historical example: Electrifying Moravia
  - Region of present Czech republic
  - Build power lines to connect villages to electric grid
  - Different connections have different costs to build (long/short distance, good/bad terrain)
  - Borůvka's algorithm, 1926
- Computer networks, road networks, ...

# Properties of trees

## Definition

A tree is a connected graph with no cycles

We could also have chosen any of the following (equivalent statements):

- A tree is a connected graph with $n - 1$ edges
- A tree is a graph with $n - 1$ edges and no cycles
- A tree is a graph with a unique path between any two vertices
- A tree is a connected graph such that removing any edge leaves a disconnected graph

# Properties of trees

**Definition**

A tree is a connected graph with no cycles

We could also have chosen any of the following (equivalent statements):

- A tree is a connected graph with $n - 1$ edges
- A tree is a graph with $n - 1$ edges and no cycles
- A tree is a graph with a unique path between any two vertices
- A tree is a connected graph such that removing any edge leaves a disconnected graph

# Properties of trees

## Definition

A tree is a connected graph with no cycles

We could also have chosen any of the following (equivalent statements):

- A tree is a connected graph with $n - 1$ edges
- A tree is a graph with $n - 1$ edges and no cycles
- A tree is a graph with a unique path between any two vertices
- A tree is a connected graph such that removing any edge leaves a disconnected graph

# Properties of trees

## Definition

A tree is a connected graph with no cycles

We could also have chosen any of the following (equivalent statements):

- A tree is a connected graph with $n - 1$ edges
- A tree is a graph with $n - 1$ edges and no cycles
- A tree is a graph with a unique path between any two vertices
- A tree is a connected graph such that removing any edge leaves a disconnected graph

# Properties of trees

**Definition**

A tree is a connected graph with no cycles

We could also have chosen any of the following (equivalent statements):

- A tree is a connected graph with $n - 1$ edges
- A tree is a graph with $n - 1$ edges and no cycles
- A tree is a graph with a unique path between any two vertices
- A tree is a connected graph such that removing any edge leaves a disconnected graph

# Properties of Spanning Trees

## Two properties of spanning trees

Let $T$ be a spanning tree in a graph $G = (V, E)$.

- Cycle property:
  1. Adding any edge $uv$ to $T$ creates a cycle $C$
  2. Removing any edge $u'v'$ from $C$ creates a new spanning tree
- Cut property:
  1. Removing any edge $uv$ from $T$ disconnects $T$ into two parts
  2. Adding any edge $u'v'$ between these parts creates a new spanning tree

Proof (both cases): The new tree is a cycle-free graph with $n - 1$ edges

# Properties of Spanning Trees

## Two properties of spanning trees

Let $T$ be a spanning tree in a graph $G = (V, E)$.

- Cycle property:
  1. Adding any edge $uv$ to $T$ creates a cycle $C$
  2. Removing any edge $u'v'$ from $C$ creates a new spanning tree
- Cut property:
  1. Removing any edge $uv$ from $T$ disconnects $T$ into two parts
  2. Adding any edge $u'v'$ between these parts creates a new spanning tree

Proof (both cases): The new tree is a cycle-free graph with $n-1$ edges

# Properties of Spanning Trees

## Two properties of spanning trees

Let $T$ be a spanning tree in a graph $G = (V, E)$.

- Cycle property:
  1. Adding any edge $uv$ to $T$ creates a cycle $C$
  2. Removing any edge $u'v'$ from $C$ creates a new spanning tree
- Cut property:
  1. Removing any edge $uv$ from $T$ disconnects $T$ into two parts
  2. Adding any edge $u'v'$ between these parts creates a new spanning tree

Proof (both cases): The new tree is a cycle-free graph with $n - 1$ edges

# Min-cost spanning trees

## Min-cost spanning trees: Cycle rule

Let $G = (V, E)$ be a graph with edge weights, $T$ a spanning tree in $G$. Then $T$ is min-cost if and only if the following holds:

- Let $uv \in E$ be any edge not used in $T$
- Let $C$ be the cycle created by adding $uv$ to $T$
- Then $uv$ is the most expensive edge of $C$
  (or tied for most expensive, if there are ties)

Why?

- Suppose not. Let $uv$ be an edge not used in $T$, that is not the most expensive in its cycle $C$ in $T + uv$
- Create a tree $T'$ by adding $uv$, then deleting the most expensive edge $u'v'$ from $C$.
- Then $T'$ is a cheaper spanning tree than $T$, a contradiction.

# Min-cost spanning trees

## Min-cost spanning trees: Cycle rule

Let $G = (V, E)$ be a graph with edge weights, $T$ a spanning tree in $G$. Then $T$ is min-cost if and only if the following holds:

- Let $uv \in E$ be any edge not used in $T$
- Let $C$ be the cycle created by adding $uv$ to $T$
- Then $uv$ is the most expensive edge of $C$
  (or tied for most expensive, if there are ties)

Why?

- Suppose not. Let $uv$ be an edge not used in $T$, that is not the most expensive in its cycle $C$ in $T + uv$
- Create a tree $T'$ by adding $uv$, then deleting the most expensive edge $u'v'$ from $C$.
- Then $T'$ is a cheaper spanning tree than $T$, a contradiction.

# Min-cost spanning trees

## Min-cost spanning trees: Cycle rule

Let $G = (V, E)$ be a graph with edge weights, $T$ a spanning tree in $G$. Then $T$ is min-cost if and only if the following holds:

- Let $uv \in E$ be any edge not used in $T$
- Let $C$ be the cycle created by adding $uv$ to $T$
- Then $uv$ is the most expensive edge of $C$
  (or tied for most expensive, if there are ties)

Why?

- Suppose not. Let $uv$ be an edge not used in $T$, that is not the most expensive in its cycle $C$ in $T + uv$
- Create a tree $T'$ by adding $uv$, then deleting the most expensive edge $u'v'$ from $C$.
- Then $T'$ is a cheaper spanning tree than $T$, a contradiction.

# Min-cost spanning trees

## Min-cost spanning trees: Cycle rule

Let $G = (V, E)$ be a graph with edge weights, $T$ a spanning tree in $G$. Then $T$ is min-cost if and only if the following holds:

- Let $uv \in E$ be any edge not used in $T$
- Let $C$ be the cycle created by adding $uv$ to $T$
- Then $uv$ is the most expensive edge of $C$
  (or tied for most expensive, if there are ties)

Why?

- Suppose not. Let $uv$ be an edge not used in $T$, that is not the most expensive in its cycle $C$ in $T + uv$
- Create a tree $T'$ by adding $uv$, then deleting the most expensive edge $u'v'$ from $C$.
- Then $T'$ is a cheaper spanning tree than $T$, a contradiction.

# Min-cost spanning trees

## Min-cost spanning trees: Cycle rule

Let $G = (V, E)$ be a graph with edge weights, $T$ a spanning tree in $G$. Then $T$ is min-cost if and only if the following holds:

- Let $uv \in E$ be any edge not used in $T$
- Let $C$ be the cycle created by adding $uv$ to $T$
- Then $uv$ is the most expensive edge of $C$
  (or tied for most expensive, if there are ties)

Why? [1]

- Suppose not. Let $uv$ be an edge not used in $T$, that is not the most expensive in its cycle $C$ in $T + uv$
- Create a tree $T'$ by adding $uv$, then deleting the most expensive edge $u'v'$ from $C$.
- Then $T'$ is a cheaper spanning tree than $T$, a contradiction.

[1] This was necessity. To show sufficiency is a little bit harder.

# Min-cost spanning trees: Option II

## Min-cost spanning trees: Cut rule

Let $G = (V, E)$ be a graph with edge weights, $T$ a spanning tree in $G$. Then $T$ is min-cost if and only if the following holds:

- For any split of $V$ into two parts $V = A \cup B$,
  $T$ contains a cheapest edge between $A$ and $B$.

Proof: Same as cycle rule (if the statement were false, you could create a cheaper tree from $T$ as $T - uv + u'v'$).

# Min-cost spanning trees: Option II

## Min-cost spanning trees: Cut rule

Let $G = (V, E)$ be a graph with edge weights, $T$ a spanning tree in $G$. Then $T$ is min-cost if and only if the following holds:

- For any split of $V$ into two parts $V = A \cup B$,
  $T$ contains a cheapest edge between $A$ and $B$.

Proof: Same as cycle rule (if the statement were false, you could create a cheaper tree from $T$ as $T - uv + u'v'$).

# Minimum spanning tree: algorithms

- Prim's algorithm (the one you know):
  1. Begin with tree $T$ of single vertex $v$, no edges
  2. Repeat until $T$ is a spanning tree:
     2.1 Extend $T$ by the cheapest edge $uv$ where $u$ in $T$, $v$ not in $T$
- Kruskal's algorithm
  1. Begin with $T = \emptyset$
  2. Sort all edges of $E$: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$
  3. For every edge $e$ in this order:
     3.1 If adding $e$ to $T$ does not create a cycle,
     add $e$ to $T$.

# Minimum spanning tree: algorithms

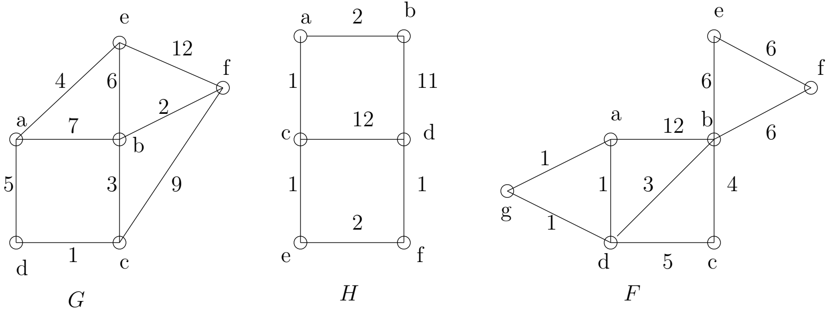
- Prim's algorithm (the one you know):
    1. Begin with tree $T$ of single vertex $v$, no edges
    2. Repeat until $T$ is a spanning tree:
        2.1 Extend $T$ by the cheapest edge $uv$ where $u$ in $T$, $v$ not in $T$
- Kruskal's algorithm
    1. Begin with $T = \emptyset$
    2. Sort all edges of $E$: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$
    3. For every edge $e$ in this order:
        3.1 If adding $e$ to $T$ does not create a cycle, add $e$ to $T$.

# Minimum spanning tree: algorithms

- Prim's algorithm (the one you know):
  1. Begin with tree $T$ of single vertex $v$, no edges
  2. Repeat until $T$ is a spanning tree:
     2.1 Extend $T$ by the cheapest edge $uv$ where $u$ in $T$, $v$ not in $T$
- Kruskal's algorithm
  1. Begin with $T = \emptyset$
  2. Sort all edges of $E$: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$
  3. For every edge $e$ in this order:
     3.1 If adding $e$ to $T$ does not create a cycle,
     add $e$ to $T$.
- Issues:
  - Algorithm workings
  - Correctness
  - Efficient implementation

# Using the algorithms

(1) Run Prim's algorithm (start from vertex *a*) and Kruskal's algorithm on graph *G* below. (2) Using Kruskal's algorithm, determine how many minimum spanning trees (MSTs) graph *F* has. (3) How many MSTs are in the graph $H'$ obtained from *H* by replacing cost 11 with 1 and 12 with 2?



# Using the algorithms: Solutions

1. Graph $G$ has a minimum spanning tree (MST) with the edges of costs 1, 2, 3, 4, 5.
2. Since in any MSP of $F$ only 2 edges of the 3 edges of cost 1 can be used and the same for edges of cost 6, $F$ has $3 \times 3 = 9$ MSTs (edges of cost 3 and 4 must be in each MST).
3. In $H$ each MST must have edges of cost 1, but only one of 3 edges of cost 2. Hence, 3 MSTs.

# Using the algorithms: Solutions

1. Graph $G$ has a minimum spanning tree (MST) with the edges of costs 1, 2, 3, 4, 5.
2. Since in any MSP of $F$ only 2 edges of the 3 edges of cost 1 can be used and the same for edges of cost 6, $F$ has $3 \times 3 = 9$ MSTs (edges of cost 3 and 4 must be in each MST).
3. In $H$ each MST must have edges of cost 1, but only one of 3 edges of cost 2. Hence, 3 MSTs.

# Using the algorithms: Solutions

1. Graph $G$ has a minimum spanning tree (MST) with the edges of costs 1, 2, 3, 4, 5.
2. Since in any MSP of $F$ only 2 edges of the 3 edges of cost 1 can be used and the same for edges of cost 6, $F$ has $3 \times 3 = 9$ MSTs (edges of cost 3 and 4 must be in each MST).
3. In $H$ each MST must have edges of cost 1, but only one of 3 edges of cost 2. Hence, 3 MSTs.

# Spanning trees: Prim's algorithm

# Prim's algorithm, outline

Input: Graph $G = (V, E)$, edge weights $w$

- Start with a tree $T$ of a single vertex $v$, no edges
- Repeat until complete:
  - Extend $T$ with the cheapest edge $e$ adding a new vertex to $T$

- Discover/Extend central loop (Priority Queue)

# Prim's algorithm, outline

Input: Graph $G = (V, E)$, edge weights $w$

- Start with a tree $T$ of a single vertex $v$, no edges
- Repeat until complete:
  - Extend $T$ with the cheapest edge $e$ adding a new vertex to $T$

- Discover/Extend central loop (Priority Queue)

# First implementation

- Data structures:
  1. Vertices marked/unmarked, initially all unmarked
  2. Priority Queue pq of edges, initially empty
  3. The tree T
- Use a "visit" subroutine process(v):
  1. Mark $v$ as visited
  2. For every unmarked neighbour $u$ of $v$:
     2.1 Add edge $uv$ to pq with weight $w(uv)$
- Code, main loop:
  1. Select initial vertex $v$, process(v)
  2. Until pq is empty:
     2.1 e=pq.deleteMin() // pull cheapest edge $e$ from pq
     2.2 If $e$ has an unmarked endpoint $u$:
     add $e$ to $T$, process(u)

# First implementation

- Data structures:
  1. Vertices marked/unmarked, initially all unmarked
  2. Priority Queue pq of edges, initially empty
  3. The tree T
- Use a "visit" subroutine process(v):
  1. Mark $v$ as visited
  2. For every unmarked neighbour $u$ of $v$:
     - 2.1 Add edge $uv$ to pq with weight $w(uv)$
- Code, main loop:
  1. Select initial vertex $v$, process(v)
  2. Until pq is empty:
     - 2.1 e=pq.deleteMin() // pull cheapest edge $e$ from pq
     - 2.2 If $e$ has an unmarked endpoint $u$:
       add $e$ to $T$, process(u)

# First implementation

- Data structures:
  1. Vertices marked/unmarked, initially all unmarked
  2. Priority Queue pq of edges, initially empty
  3. The tree T
- Use a "visit" subroutine process(v):
  1. Mark $v$ as visited
  2. For every unmarked neighbour $u$ of $v$:
     - 2.1 Add edge $uv$ to pq with weight $w(uv)$
- Code, main loop:
  1. Select initial vertex $v$, process(v)
  2. Until pq is empty:
     - 2.1 e=pq.deleteMin() // pull cheapest edge $e$ from pq
     - 2.2 If $e$ has an unmarked endpoint $u$:
       add $e$ to $T$, process(u)

# First version, running time

- Our code performs:
  1. $\mathcal{O}(|V|)$ executions of "visit all neighbours of a vertex", takes $\mathcal{O}(|E|)$ time in total
  2. $\mathcal{O}(|E|)$ insertions into pq at $\mathcal{O}(\log |E|)$ per op
  3. $\mathcal{O}(|E|)$ deleteMin() calls to pq at $\mathcal{O}(\log |E|)$ per op
- Total time[2] $\mathcal{O}(|E| \log |E|)$
- Correctness proof (skipped):
  - Can show that the tree constructed follows cut rule

[2] We assume that the graph is connected, otherwise it has no spanning tree – so $|V| \leq |E| - 1$

# First version, running time

- Our code performs:
  1. $\mathcal{O}(|V|)$ executions of "visit all neighbours of a vertex", takes $\mathcal{O}(|E|)$ time in total
  2. $\mathcal{O}(|E|)$ insertions into pq at $\mathcal{O}(\log |E|)$ per op
  3. $\mathcal{O}(|E|)$ deleteMin() calls to pq at $\mathcal{O}(\log |E|)$ per op
- Total time[2] $\mathcal{O}(|E| \log |E|)$
- Correctness proof (skipped):
  - Can show that the tree constructed follows cut rule

[2]We assume that the graph is connected, otherwise it has no spanning tree – so $|V| \leq |E| - 1$

# Advanced implementation

- The code adds edges to pq, but the algorithm needs to discover vertices
- This leads to a minor inefficiency (in space and time)
- Using an advanced priority queue (with decreaseKey method) we can write the code using a priority queue of vertices for space $\mathcal{O}(|V|)$, time $\mathcal{O}(|E| + |V| \log |V|)$ (theoretical)
- Details at the end of slide set (optional, not examined, i.e. marked ★)

# Advanced implementation

- The code adds edges to pq, but the algorithm needs to discover vertices
- This leads to a minor inefficiency (in space and time)
- Using an advanced priority queue (with decreaseKey method) we can write the code using a priority queue of vertices for space $\mathcal{O}(|V|)$, time $\mathcal{O}(|E| + |V| \log |V|)$ (theoretical)
- Details at the end of slide set (optional, not examined, i.e. marked ★)

# Advanced implementation

- The code adds edges to pq, but the algorithm needs to discover vertices
- This leads to a minor inefficiency (in space and time)
- Using an advanced priority queue (with decreaseKey method) we can write the code using a priority queue of vertices for space $\mathcal{O}(|V|)$, time $\mathcal{O}(|E| + |V| \log |V|)$ (theoretical)
- Details at the end of slide set (optional, not examined, i.e. marked $\bigstar$)

# Kruskal's algorithm

# Kruskal's algorithm

The pseudocode looks even simpler than Prim:

1. Begin with $T = \emptyset$
2. Sort all edges of $E$: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$
3. For every edge $e$ in this order:
   3.1 If adding $e$ to $T$ does not create a cycle, add $e$ to $T$.

Note: During the loop, $T$ has several connected components – it's a forest, not a tree

# Kruskal's algorithm

The pseudocode looks even simpler than Prim:

1. Begin with $T = \emptyset$
2. Sort all edges of $E$: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$
3. For every edge $e$ in this order:
    3.1 If adding $e$ to $T$ does not create a cycle,
    add $e$ to $T$.

Note: During the loop, $T$ has several connected components – it's a forest, not a tree

# Kruskal: Correctness

**Correctness**

Kruskal's algorithm creates a minimum spanning tree

- It clearly creates a spanning tree
- Min-cost via the cycle rule:
  - Let $e$ be any edge not used in $T$
  - Since $e$ was not added to $T$, at the point where it was considered by the algorithm, it would have created a cycle $C$
  - Since the edges are considered in sorted order, $e$ was a most-expensive edge of $C$
- Every edge $e$ from $G$ not in $T$ satisfies the cycle rule $\Rightarrow$ $T$ is a minimum spanning tree

# Kruskal: Correctness

**Correctness**

Kruskal's algorithm creates a minimum spanning tree

- It clearly creates a spanning tree
- Min-cost via the cycle rule:
  - Let $e$ be any edge not used in $T$
  - Since $e$ was not added to $T$, at the point where it was considered by the algorithm, it would have created a cycle $C$
  - Since the edges are considered in sorted order, $e$ was a most-expensive edge of $C$
- Every edge $e$ from $G$ not in $T$ satisfies the cycle rule $\Rightarrow$ $T$ is a minimum spanning tree

# Kruskal: Correctness

**Correctness**

Kruskal's algorithm creates a minimum spanning tree

- It clearly creates a spanning tree
- Min-cost via the cycle rule:
  - Let $e$ be any edge not used in $T$
  - Since $e$ was not added to $T$, at the point where it was considered by the algorithm, it would have created a cycle $C$
  - Since the edges are considered in sorted order, $e$ was a most-expensive edge of $C$
- Every edge $e$ from $G$ not in $T$ satisfies the cycle rule $\Rightarrow$ $T$ is a minimum spanning tree

# Kruskal: Correctness

**Correctness**

Kruskal's algorithm creates a minimum spanning tree

- It clearly creates a spanning tree
- Min-cost via the cycle rule:
  - Let $e$ be any edge not used in $T$
  - Since $e$ was not added to $T$, at the point where it was considered by the algorithm, it would have created a cycle $C$
  - Since the edges are considered in sorted order, $e$ was a most-expensive edge of $C$
- Every edge $e$ from $G$ not in $T$ satisfies the cycle rule $\Rightarrow$ $T$ is a minimum spanning tree

# Kruskal: Correctness

**Correctness**

Kruskal's algorithm creates a minimum spanning tree

- It clearly creates a spanning tree
- Min-cost via the cycle rule:
  - Let $e$ be any edge not used in $T$
  - Since $e$ was not added to $T$, at the point where it was considered by the algorithm, it would have created a cycle $C$
  - Since the edges are considered in sorted order, $e$ was a most-expensive edge of $C$
- Every edge $e$ from $G$ not in $T$ satisfies the cycle rule $\Rightarrow$ $T$ is a minimum spanning tree

# Kruskal: Correctness

**Correctness**

Kruskal's algorithm creates a minimum spanning tree

- It clearly creates a spanning tree
- Min-cost via the cycle rule:
  - Let $e$ be any edge not used in $T$
  - Since $e$ was not added to $T$, at the point where it was considered by the algorithm, it would have created a cycle $C$
  - Since the edges are considered in sorted order, $e$ was a most-expensive edge of $C$
- Every edge $e$ from $G$ not in $T$ satisfies the cycle rule $\Rightarrow$ $T$ is a minimum spanning tree

# Implementation

1. Sort the edges
   - No problem
2. For every edge $uv$ in sorted order:
   2.1 Decide if it completes a cycle
   2.2 If it does not, add it to $T$

Completes a cycle has two implementations:

- A slow one:
  - Search through the graph formed by $T$, starting from $u$, to see if we find $v$ (e.g., DFS from $u$ in $T$)
  - Up to $\mathcal{O}(n)$ time for a single check
- A fast one: Using a data structure called Union Find

# Implementation

1. Sort the edges
   - No problem
2. For every edge $uv$ in sorted order:
   2.1 Decide if it completes a cycle
   2.2 If it does not, add it to $T$

Completes a cycle has two implementations:

- A slow one:
  - Search through the graph formed by $T$, starting from $u$, to see if we find $v$ (e.g., DFS from $u$ in $T$)
  - Up to $\mathcal{O}(n)$ time for a single check
- A fast one: Using a data structure called Union Find

# Implementation

1. Sort the edges
   - No problem
2. For every edge $uv$ in sorted order:
   2.1 Decide if it completes a cycle
   2.2 If it does not, add it to $T$

Completes a cycle has two implementations:

- A slow one:
  - Search through the graph formed by $T$, starting from $u$, to see if we find $v$ (e.g., DFS from $u$ in $T$)
  - Up to $\mathcal{O}(n)$ time for a single check
- A fast one: Using a data structure called Union Find

# Fast implementation

We need a data structure for connected components:

- Find(v) – return a name for the connected component of $T$ that contains $v$
- Union(u,v) – "merge" the components of $u$ and $v$ into one (by adding $uv$ to $T$)

Efficient implementation using this:

1. Sort the edges $E$ (e.g., merge sort) at $\mathcal{O}(|E| \log |E|)$ time
2. Initiate your Union-Find structure to "empty"
3. For every edge $uv$ in sorted order:
   3.1 If Find(u) == Find(v): ignore edge
   3.2 Otherwise call Union(u,v) and add to $T$

With a good Union-Find, the time is dominated by $\mathcal{O}(|E| \log |E|)$.

# Fast implementation

We need a data structure for connected components:

- Find(v) – return a name for the connected component of $T$ that contains $v$
- Union(u,v) – "merge" the components of $u$ and $v$ into one (by adding $uv$ to $T$)

Efficient implementation using this:

1. Sort the edges $E$ (e.g., merge sort) at $\mathcal{O}(|E| \log |E|)$ time
2. Initiate your Union-Find structure to "empty"
3. For every edge $uv$ in sorted order:
   3.1 If Find(u) == Find(v): ignore edge
   3.2 Otherwise call Union(u,v) and add to $T$

With a good Union-Find, the time is dominated by $\mathcal{O}(|E| \log |E|)$.

# Fast implementation

We need a data structure for connected components:

- Find(v) – return a name for the connected component of $T$ that contains $v$
- Union(u,v) – "merge" the components of $u$ and $v$ into one (by adding $uv$ to $T$)

Efficient implementation using this:

1. Sort the edges $E$ (e.g., merge sort) at $\mathcal{O}(|E| \log |E|)$ time
2. Initiate your Union-Find structure to "empty"
3. For every edge $uv$ in sorted order:
    3.1 If Find(u) == Find(v): ignore edge
    3.2 Otherwise call Union(u,v) and add to $T$

With a good Union-Find, the time is dominated by $\mathcal{O}(|E| \log |E|)$.

# $\star$ Prim: advanced implementation

# "Advanced PQ" interface

- In addition to the usual min-Priority Queue operations:
  - insert(item, weight): add item with value weight
  - minItem(), deleteMin(): Return min-weight item
- We will need a Priority Queue that also supports the following:
  1. contains(item): membership test
  2. currentValue(item): return current item priority/weight
  3. decreaseValue(item, newWeight): decrease weight of item to newWeight
- Implementations:
  1. Heap+Hash table with $\mathcal{O}(\log n)$ time per decreaseKey
  2. Fibonacci heaps: Advanced structure, not yet practical
     - Impractical: Pointer jumping, complex code
     - But average time $\mathcal{O}(1)$ for decreaseKey

# "Advanced PQ" interface

- In addition to the usual min-Priority Queue operations:
  - insert(item, weight): add item with value weight
  - minItem(), deleteMin(): Return min-weight item
- We will need a Priority Queue that also supports the following:
  1. contains(item): membership test
  2. currentValue(item): return current item priority/weight
  3. decreaseValue(item, newWeight): decrease weight of item to newWeight
- Implementations:
  1. Heap+Hash table with $\mathcal{O}(\log n)$ time per decreaseKey
  2. Fibonacci heaps: Advanced structure, not yet practical
     - Impractical: Pointer jumping, complex code
     - But average time $\mathcal{O}(1)$ for decreaseKey

# "Advanced PQ" interface

- In addition to the usual min-Priority Queue operations:
  - insert(item, weight): add item with value weight
  - minItem(), deleteMin(): Return min-weight item
- We will need a Priority Queue that also supports the following:
  1. contains(item): membership test
  2. currentValue(item): return current item priority/weight
  3. decreaseValue(item, newWeight): decrease weight of item to newWeight
- Implementations:
  1. Heap+Hash table with $\mathcal{O}(\log n)$ time per decreaseKey
  2. Fibonacci heaps: Advanced structure, not yet practical
     - Impractical: Pointer jumping, complex code
     - But average time $\mathcal{O}(1)$ for decreaseKey

# "Advanced PQ" interface

- In addition to the usual min-Priority Queue operations:
  - insert(item, weight): add item with value weight
  - minItem(), deleteMin(): Return min-weight item
- We will need a Priority Queue that also supports the following:
  1. contains(item): membership test
  2. currentValue(item): return current item priority/weight
  3. decreaseValue(item, newWeight): decrease weight of item to newWeight
- Implementations:
  1. Heap+Hash table with $\mathcal{O}(\log n)$ time per decreaseKey
  2. Fibonacci heaps: Advanced structure, not yet practical
     - Impractical: Pointer jumping, complex code
     - But average time $\mathcal{O}(1)$ for decreaseKey

# “Advanced PQ” interface

- In addition to the usual min-Priority Queue operations:
  - insert(item, weight): add item with value weight
  - minItem(), deleteMin(): Return min-weight item
- We will need a Priority Queue that also supports the following:
  1. contains(item): membership test
  2. currentValue(item): return current item priority/weight
  3. decreaseValue(item, newWeight): decrease weight of item to newWeight
- Implementations:
  1. Heap+Hash table with $\mathcal{O}(\log n)$ time per decreaseKey
  2. Fibonacci heaps: Advanced structure, not yet practical
     - Impractical: Pointer jumping, complex code
     - But average time $\mathcal{O}(1)$ for decreaseKey

# Advanced Prim implementation: Outline

- Data structures:
  1. Min-Priority Queue pq containing vertices
  2. edgeTo[v] array codes edges, will encode tree
  3. Vertices marked/unmarked as before
- Principles:
  - pq contains candidate vertex to add to tree
  - Priority (weight) of vertex v in pq is cheapest known edge to v from tree
  - When we pull v from pq ("discover" step), we update neighbours u of v if a cheaper edge to u was found
  - If a cheaper edge was found, we call decreaseKey to *v* instead of a full insert

# Advanced Prim implementation: Outline

- Data structures:
  1. Min-Priority Queue pq containing vertices
  2. edgeTo[v] array codes edges, will encode tree
  3. Vertices marked/unmarked as before
- Principles:
  - pq contains candidate vertex to add to tree
  - Priority (weight) of vertex v in pq is cheapest known edge to v from tree
  - When we pull v from pq ("discover" step), we update neighbours u of v if a cheaper edge to u was found
  - If a cheaper edge was found, we call decreaseKey to *v* instead of a full insert

# Advanced Prim implementation: Outline

- Data structures:
  1. Min-Priority Queue pq containing vertices
  2. edgeTo[v] array codes edges, will encode tree
  3. Vertices marked/unmarked as before
- Principles:
  - pq contains candidate vertex to add to tree
  - Priority (weight) of vertex v in pq is cheapest known edge to v from tree
  - When we pull v from pq ("discover" step), we update neighbours u of v if a cheaper edge to u was found
  - If a cheaper edge was found, we call decreaseKey to $v$ instead of a full insert

# Full code

- Data structures:
  1. Min-Priority Queue pq containing vertices
  2. edgeTo[v] array containing edges, will encode tree
- Main loop:
  1. Select initial vertex $v$, call pq.insert(v,0)
  2. Until pq is empty:
     - 2.1 u=pq.deleteMin(); mark u
     - 2.2 For every neighbour $v$ of $u$:
       Call Update(v, w(uv))
- Support code: Update(Vertex v, Edge uv):
  1. If $v$ marked: return, do nothing
  2. If $v$ not in pq:
     - 2.1 edgeTo[v] = uv
     - 2.2 pq.insert(v, w(uv))
  3. If w(uv)< pq.currentValue(v):
     - 3.1 edgeTo[v] = uv
     - 3.2 pq.decreaseValue(v, w(uv))

# Full code

- Data structures:
  1. Min-Priority Queue pq containing vertices
  2. edgeTo[v] array containing edges, will encode tree
- Main loop:
  1. Select initial vertex $v$, call pq.insert(v,0)
  2. Until pq is empty:
     - 2.1 u=pq.deleteMin(); mark u
     - 2.2 For every neighbour $v$ of $u$:

       Call Update(v, w(uv))
- Support code: Update(Vertex v, Edge uv):
  1. If $v$ marked: return, do nothing
  2. If $v$ not in pq:
     - 2.1 edgeTo[v] = uv
     - 2.2 pq.insert(v, w(uv))
  3. If w(uv)< pq.currentValue(v):
     - 3.1 edgeTo[v] = uv
     - 3.2 pq.decreaseValue(v, w(uv))

# Full code

- Data structures:
  1. Min-Priority Queue pq containing vertices
  2. edgeTo[v] array containing edges, will encode tree
- Main loop:
  1. Select initial vertex $v$, call pq.insert(v,0)
  2. Until pq is empty:
     - 2.1 u=pq.deleteMin(); mark u
     - 2.2 For every neighbour $v$ of $u$:
       Call Update(v, w(uv))
- Support code: Update(Vertex v, Edge uv):
  1. If $v$ marked: return, do nothing
  2. If $v$ not in pq:
     - 2.1 edgeTo[v] = uv
     - 2.2 pq.insert(v, w(uv))
  3. If w(uv)< pq.currentValue(v):
     - 3.1 edgeTo[v] = uv
     - 3.2 pq.decreaseValue(v, w(uv))

# Advanced version: running time

- Our new implementation performs:
  1. $\mathcal{O}(|V|)$ insertions into pq
  2. $\mathcal{O}(|E|)$ update, decreaseKey calls
  3. $\mathcal{O}(|V|)$ deletions from pq
- With our familiar implementation:
  1. All operations take $\mathcal{O}(\log|V|)$ time
  2. Worst-case $\mathcal{O}(|E|\log|V|) = \mathcal{O}(|E|\log|E|)$ as before
  3. New best case: Zero decreaseKey calls, $\mathcal{O}(|E| + |V|\log|V|)$ time (maybe even just $\mathcal{O}(|E|)$?)
- With Fibonacci heaps (theoretical result):
  1. Insert/delete amounts to $\mathcal{O}(|V|\log|V|)$ time
  2. $\mathcal{O}(|E|)$ times decreaseKey takes $\mathcal{O}(|E|)$ total time
  3. In total worst-case time $\mathcal{O}(|E| + |V|\log|V|)$

# Advanced version: running time

- Our new implementation performs:
  1. $\mathcal{O}(|V|)$ insertions into pq
  2. $\mathcal{O}(|E|)$ update, decreaseKey calls
  3. $\mathcal{O}(|V|)$ deletions from pq
- With our familiar implementation:
  1. All operations take $\mathcal{O}(\log |V|)$ time
  2. Worst-case $\mathcal{O}(|E| \log |V|) = \mathcal{O}(|E| \log |E|)$ as before
  3. New best case: Zero decreaseKey calls,
     $\mathcal{O}(|E| + |V| \log |V|)$ time (maybe even just $\mathcal{O}(|E|)$?)
- With Fibonacci heaps (theoretical result):
  1. Insert/delete amounts to $\mathcal{O}(|V| \log |V|)$ time
  2. $\mathcal{O}(|E|)$ times decreaseKey takes $\mathcal{O}(|E|)$ total time
  3. In total worst-case time $\mathcal{O}(|E| + |V| \log |V|)$

# Advanced version: running time

- Our new implementation performs:
  1. $\mathcal{O}(|V|)$ insertions into pq
  2. $\mathcal{O}(|E|)$ update, decreaseKey calls
  3. $\mathcal{O}(|V|)$ deletions from pq
- With our familiar implementation:
  1. All operations take $\mathcal{O}(\log |V|)$ time
  2. Worst-case $\mathcal{O}(|E| \log |V|) = \mathcal{O}(|E| \log |E|)$ as before
  3. New best case: Zero decreaseKey calls,
     $\mathcal{O}(|E| + |V| \log |V|)$ time (maybe even just $\mathcal{O}(|E|)$?)
- With Fibonacci heaps (theoretical result):
  1. Insert/delete amounts to $\mathcal{O}(|V| \log |V|)$ time
  2. $\mathcal{O}(|E|)$ times decreaseKey takes $\mathcal{O}(|E|)$ total time
  3. In total worst-case time $\mathcal{O}(|E| + |V| \log |V|)$