Weighted Graphs

Viet Tung Hoang

The slides are loosely based on those of Dr Kevin Wayne, Princeton University and Prof. Mary Wootters, Stanford University
Agenda

1. Dijkstra’s algorithm

2. Minimum spanning Tree

3. Prim

4. Kruskal
Shortest Paths from CS Department to Airport

17 min
7.0 miles

17 min
6.5 miles
Shortest Path Problem

Find the shortest path from A to H
Shortest Path Problem

Find the shortest path from A to H

Run BFS: A → E → H
Shortest Path Problem

Find the shortest path from A to H

Run BFS: A → E → H

Doesn’t work if the edges are labeled by distance
Shortest Path Problem

Weighted graph

For now, weights are non-negative

$w(u, v)$: weight of edge between $u$ and $v$
Shortest Path Problem

Weighted graph

For now, weights are non-negative

$w(u, v) : \text{ weight of edge between } u \text{ and } v$

If I care about the weights, I should go $A \rightarrow B \rightarrow G \rightarrow H$
Shortest Path Problem

Given a weighted graph, find a shortest path from a source $s$ to a target $t$

For simplicity, we’ll focus on undirected graphs, but directed graphs are similar
A sub-path of a shortest path is also a shortest path

Say \textit{this} is a shortest path from \textit{s} to \textit{t}.

\textbf{Claim:} \textit{this} is a shortest path from \textit{s} to \textit{x}.
Warm-up

A sub-path of a shortest path is also a shortest path

Suppose not, this one is a shorter path from $s$ to $x$

cost = 30

cost = 40

cost = 100
Warm-up

A sub-path of a shortest path is also a shortest path

Then, that gives an **even shorter path** from $s$ to $t$

Contradiction

---

cost = 100

cost = 90
Single-Source Shortest Path Problem

Find the shortest paths from A to every other node

<table>
<thead>
<tr>
<th>Destination</th>
<th>Cost</th>
<th>How to get there</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>A → B</td>
</tr>
<tr>
<td>C</td>
<td>17</td>
<td>A → C</td>
</tr>
<tr>
<td>D</td>
<td>15</td>
<td>A → D</td>
</tr>
<tr>
<td>E</td>
<td>23</td>
<td>A → B → E</td>
</tr>
</tbody>
</table>

(Not necessarily stored as a table – how this information is represented will depend on the application)
An Application: Network Routing
Dijkstra’s Algorithm

Find the shortest path from A to everywhere else
Dijkstra
Intuition
A node is done when it’s not on the ground anymore.
Dijkstra

Intuition
Dijkstra
Intuition
Dijkstra

Intuition
Dijkstra
Intuition
Dijkstra Intuition

This creates a tree!

The shortest paths are the lengths along this tree
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- $x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$

Pick the \textbf{not-sure} node $u$ with the smallest estimate
Dijkstra: Implementation

**How far is a node from A?**

- I’m not sure yet
- I’m sure
- Current node

\[ x = d[v] \] is my best *over-estimate* for \( \text{dist}(A, v) \)

Pick the **not-sure** node \( u \) with the smallest estimate
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

$x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$

Update the (outgoing) neighbors $v$ of $u$:

$$d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$$
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure

Current node

$x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$

Update the (outgoing) neighbors $v$ of $u$:

$$d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$$
Dijkstra: Implementation

How far is a node from A?

I’m not sure yet

I’m sure

\( x = d[v] \) is my best over-estimate for \( \text{dist}(A, v) \)

predecessor of \( X \) is \( Y \)

If \( v \) is updated, set \( \text{pred}[v] \leftarrow u \)
Dijkstra: Implementation

How far is a node from A?

I’m not sure yet

I’m sure

$x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$

predecessor of $X$ is $Y$

Mark $u$ as sure
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- $x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$
- Predecessor of $X$ is $Y$

Pick the not-sure node $u$ with the smallest estimate
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

$x = d[u]$ is my best over-estimate
for $\text{dist}(A, v)$

predecessor of $X$ is $Y$

Pick the not-sure node $u$ with the smallest estimate
**Dijkstra: Implementation**

**How far is a node from A?**

- I’m not sure yet
- I’m sure

$x = d[v]$ is my best **over-estimate** for $\text{dist}(A, v)$

predecessor of $X$ is $Y$

Update the (outgoing) neighbors $v$ of $u$:

$$d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$$
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

\[ x = d[v] \text{ is my best over-estimate for } \text{dist}(A, v) \]

predecessor of \( X \) is \( Y \)

Update the (outgoing) neighbors \( v \) of \( u \):

\[ d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\} \]
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

\( x = d[v] \) is my best over-estimate for \( \text{dist}(A, v) \)

predecessor of \( X \) is \( Y \)

If \( v \) is updated, set \( \text{pred}[v] \leftarrow u \)
Dijkstra: Implementation

How far is a node from A?

I’m not sure yet

I’m sure

Current node

\( x = d[v] \) is my best over-estimate for \( \text{dist}(A, v) \)

predecessor of \( X \) is \( Y \)

Mark \( u \) as sure
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

\[ x = d[v] \text{ is my best over-estimate for } \text{dist}(A, v) \]

\( \xrightarrow{Y} X \) predecessor of \( X \) is \( Y \)

Pick the **not-sure** node \( u \) with the smallest estimate
Dijkstra: Implementation

How far is a node from A?

I’m not sure yet
I’m sure

$x = d[v]$ is my best **over-estimate** for $\text{dist}(A, v)$

predecessor of $X$ is $Y$

Pick the **not-sure** node $u$ with the smallest estimate
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

\[ x = d[v] \] is my best over-estimate for \( \text{dist}(A, v) \)

predecessor of \( X \) is \( Y \)

Update the (outgoing) neighbors \( v \) of \( u \):

\[ d[v] \leftarrow \min \{ d[v], d[u] + w(u, v) \} \]
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node
- $x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$

predecessor of $X$ is $Y$

Update the (outgoing) neighbors $v$ of $u$:

$$d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$$
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

\[ x = d[v] \text{ is my best over-estimate} \]
for \( \text{dist}(A, v) \)

predecessor of \( X \) is \( Y \)

If \( v \) is updated, set \( \text{pred}[v] \leftarrow u \)
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure

\[ x = d[v] \] is my best over-estimate for \( \text{dist}(A, v) \)

- predecessor of \( X \) is \( Y \)

Mark \( u \) as sure
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

\[ x = d[v] \] is my best over-estimate for \( \text{dist}(A, v) \)

predecessor of \( X \) is \( Y \)

Pick the not-sure node \( u \) with the smallest estimate
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure

$x = d[v]$ is my best **over-estimate** for $\text{dist}(A, v)$

Predecessor of $X$ is $Y$

Pick the **not-sure** node $u$ with the smallest estimate
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

$x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$

predecessor of $X$ is $Y$

Update the (outgoing) neighbors $v$ of $u$:

$$d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$$
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

\[ x = d[v] \] is my best over-estimate for \( \text{dist}(A, v) \)

Y ← X predecessor of X is Y

Update the (outgoing) neighbors \( v \) of \( u \):

\[ d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\} \]

If \( v \) is updated, set \( \text{pred}[v] \leftarrow u \)
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- $x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$
- Predecessor of $X$ is $Y$

Mark $u$ as sure
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

$x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$

Predecessor of $X$ is $Y$

Pick the not-sure node $u$ with the smallest estimate
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node
- $x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$
- Predecessor of $X$ is $Y$

Pick the not-sure node $u$ with the smallest estimate
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

$x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$

Predecessor of $X$ is $Y$

Update the (outgoing) neighbors $v$ of $u$:

$$d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$$

If $v$ is updated, set $\text{pred}[v] \leftarrow u$
Dijkstra: Implementation

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node $x = d[v]$ is my best over-estimate for $\text{dist}(A, v)$
- Predecessor of $X$ is $Y$

Mark $u$ as sure

Terminate when every node is sure
Dijkstra In a Nut Shell

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

\[ x = d[v] \text{ is my best over-estimate for } \text{dist}(A, v) \]

predecessor of \( X \) is \( Y \)

Grow a tree (formed by sure-nodes and their predecessor links)
Dijkstra In a Nut Shell

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

\[ x = d[v] \] is my best over-estimate for \( \text{dist}(A, v) \)

\( Y \rightarrow X \) predecessor of \( X \) is \( Y \)

**Greedy choice:** Add the closest node (by the estimates) to the tree
Dijkstra In a Nut Shell

How far is a node from A?

- I’m not sure yet
- I’m sure
- Current node

\[ x = d[v] \] is my best over-estimate for \( \text{dist}(A, v) \)

predecessor of \( X \) is \( Y \)

Update (outgoing) neighbors of the newly added node
Dijkstra: Implementation

We need a data structure (a priority queue) to support:

- **Insert:** Store every node in the queue with their initial estimate
- **DeleteMin:** Extract the node with smallest estimate
- **UpdateKey:** Update the estimates of (outgoing) neighbors
Dijkstra’s Algorithm: Implementation

```
procedure Dijkstra(V, E, w, s)
Create an empty priority queue Q

\[ d[s] \leftarrow 0 \] and \[ d[v] \leftarrow \infty \] for every \( v \neq s \) // Initialize distance

for \( v \in V \) do Insert(Q, v, d[v]) // Put all nodes in the queue

while (NotEmpty(Q)) // Process until every node is marked sure

  // Choose the unsure node of smallest estimate
  \( u \leftarrow \text{DeleteMin}(Q) \)

  for \((u, v) \in E \) incident to \( u \) do // Look at (outgoing) neighbors
    if \( d[v] > d[u] + w(u, v) \) then // Update if new estimate is better
      \[ d[v] \leftarrow d[u] + w(u, v) \]; \( u \leftarrow \text{UpdateKey}(Q, v, d[v]) \); pred[v] \( \leftarrow u \)
```
Timing Analysis

$n$ nodes and $m$ edges

Insert: $O(n)$ calls
Put every node in the queue once

DeleteMin: $O(n)$ calls
Each node is marked sure exactly once

UpdateKey: $O(m)$ calls
Each node $v$ updates at most $\text{deg}(v)$ neighbors

$$\sum_{v \in V} \text{deg}(v) = 2m$$

- Array implementation is optimal for dense graphs
- Binary heap much faster for sparse graphs

<table>
<thead>
<tr>
<th>Priority Queue Implementation</th>
<th>Insert</th>
<th>DeleteMin</th>
<th>UpdateKey</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered Array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Binary heap</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
<td>$O(m \log(n))$</td>
</tr>
</tbody>
</table>
In Practice

Shortest paths on a graph with $n$ nodes and about $5n$ edges

BFS is much faster but doesn’t work on weighted graphs.
Dijkstra Is Used In Practice

- e.g., OSPF (Open Shortest Path First), a routing protocol for IP networks, uses Dijkstra

- **Drawback**: If weights change, need to re-run the whole thing.
Dijkstra Restriction

Need **non-negative** weights

Find **shortest** paths from A

Incorrect shortest path from A to B!
When Negative Weights Are Needed?

Say edges are labeled with **profits**

Find **longest** paths from A to maximize **profits**

Find **shortest** paths from A, with negative weights
Exercise: Graphs With Node Weights

Here nodes (instead of edges) are labeled with weights.
Exercise: Graphs With Node Weights

This path has cost 33
Exercise: Graphs With Node Weights

How to find shortest paths from A to every other nodes?
Exercise: Nearest Airports

Goal: For each city, find its nearest airport

City with airport

City without airport
Exercise: Nearest Airports

Goal: For each city, find its nearest airport

- City with airport
- City without airport

nearest airport of $X$ is $Y$
Goal: Minimize the shortest path from source to destination if we have the ability to set the weight of one edge to 0
Exercise: Shortest Path With Edge Skipping

Goal: Minimize the shortest path from source to destination if we have the ability to set the weight of one edge to 0

If we pick (C, E) then the cost of the shortest path is now 3
Exercise: Shortest Path With Edge Skipping

If we instead pick (B, E) then the cost of the shortest path is now 1

**Goal:** Minimize the shortest path from source to destination if we have the ability to set the weight of one edge to 0
1. Dijkstra’s algorithm

2. Minimum spanning Tree

3. Prim

4. Kruskal
Minimum Spanning Tree

Say we have a weighted, undirected graph

Recall: a tree is a connected graph without cycle

A spanning tree is a tree that connects all nodes.
Minimum Spanning Tree

Say we have a weighted, undirected graph

The cost of a spanning tree is the sum of the weights on the edges.

A spanning tree is a tree that connects all nodes.

This is a spanning tree

It has cost 66
Minimum Spanning Tree

Say we have a weighted, undirected graph

A spanning tree is a tree that connects all nodes.
Minimum Spanning Tree

Say we have a weighted, undirected graph

A minimum spanning tree is a spanning tree of minimal cost.
Minimum Spanning Tree

Say we have a weighted, undirected graph

A minimum spanning tree is a spanning tree of minimal cost.

This is a minimum spanning tree

It has cost 37
Why MST?

- Network design:
  Connect cities with roads/electricity/phones

- Clustering

- Useful tool for other algorithms
How To Find An MST?

Red rules with cycles

Blue rules with cutsets

Red-Blue Principle

Prim’s algorithm

Kruskal’s algorithm

Boruvka’s algorithm

What we’ll learn in this class
Terminology: Cuts in Graph

A cut is a partition of the nodes into 2 parts.

This is the cut “\{A,B,D,E\} and \{C,I,H,G,F\}”
The cutset of a cut is the set of edges crossing the two parts of the cut.

Terminology: Cutset

This is the cutset \{ (A,H), (B,H), (B,C), (D,C), (D,F), (E,F) \}
The Red-Blue Principle

Initially all edges are uncolored

Repeatedly apply red and blue rules to color the edges
Upon termination, blue edges form an MST
The Red-Blue Principle

**Red rule:** - Pick a cycle of no red edges.
  - Select an uncolored edge of max weight and color it red

**Intuition:** MST has no cycle
Some edge in the cycle must not be in MST

**Greedy choice:** Pick the most expensive uncolored edge
The Red-Blue Principle

**Red rule:** - Pick a cycle of no red edges.

- Select an uncolored edge of max weight and color it red
The Red-Blue Principle

**Blue rule:** - Pick a cutset of no blue edges.
- Select an uncolored edge of min weight and color it blue

**Intuition:** MST is connected  Some edge in the cutset must be in MST

**Greedy choice:** Pick the cheapest uncolored edge
The Red-Blue Principle

**Blue rule:** - Pick a cutset of no blue edges.
   - Select an uncolored edge of min weight and color it blue
The Red-Blue Principle

**Blue rule:**
- Pick a cutset of no blue edges.
- Select an uncolored edge of min weight and color it blue
The Red-Blue Principle

**Blue rule:** - Pick a cutset of no blue edges.  
- Select an uncolored edge of min weight and color it blue
The Red-Blue Principle

**Red rule:** - Pick a cycle of no red edges.

  - Select an uncolored edge of max weight and color it red
The Red-Blue Principle

**Red rule:**
- Pick a cycle of no red edges.
- Select an uncolored edge of max weight and color it red
The Red-Blue Principle

Eventually all the edges are colored
The Red-Blue Principle

How to design an algorithm based on this principle?

Blue edges form an MST
Agenda

1. Dijkstra’s algorithm
2. Minimum spanning Tree
3. Prim
4. Kruskal
5. Clustering
Prim’s Algorithm

**Idea:**
- Start growing a tree
- Greedily add the shortest edge we can to grow the tree
Prim’s Algorithm

**Idea:**
- Start growing a tree
- Greedily add the shortest edge we can to grow the tree
Prim’s Algorithm

**Idea:**
- Start growing a tree
- Greedily add the shortest edge we can to grow the tree
Prim’s Algorithm

**Idea:** - Start growing a tree
-Greedily add the shortest edge we can to grow the tree
Prim’s Algorithm

**Idea:**
- Start growing a tree
- Greedly add the shortest edge we can to grow the tree
Prim’s Algorithm

Idea: - Start growing a tree
    - Greedily add the shortest edge we can to grow the tree
Prim’s Algorithm

Idea: - Start growing a tree
-Greedily add the shortest edge we can to grow the tree
Prim’s Algorithm

**Idea:** - Start growing a tree

-Greedily add the shortest edge we can to grow the tree
Prim’s Algorithm

**Idea:** - Start growing a tree
- Greedily add the shortest edge we can to grow the tree
Why Does It Work?
Prim As a Special Case of Red-Blue Principle

- The tree that we grow creates a cut:

“{Nodes inside tree} and {Nodes outside tree}”

- Prim applies the blue rule to this cutset
Why Does It Work?
Prim As a Special Case of Red-Blue Principle

- The tree that we grow creates a cut:

  "{Nodes inside tree} and {Nodes outside tree}"

- Prim applies the blue rule to this cutset
Why Does It Work?
Prim As a Special Case of Red-Blue Principle

- The tree that we grow creates a cut:
  
  “{Nodes inside tree} and {Nodes outside tree}”

- Prim applies the blue rule to this cutset
Why Does It Work?

Prim As a Special Case of Red-Blue Principle

- The tree that we grow creates a cut:
  
  "\{Nodes inside tree\} and \{Nodes outside tree\}"

- Prim applies the blue rule to this cutset
Why Does It Work?
Prim As a Special Case of Red-Blue Principle

- The tree that we grow creates a cut:
  
  “{Nodes inside tree} and {Nodes outside tree}”

- Prim applies the blue rule to this cutset
Why Does It Work?
Prim As a Special Case of Red-Blue Principle

- The tree that we grow creates a cut:
  
  "\{Nodes inside tree\} and \{Nodes outside tree\}"

- Prim applies the blue rule to this cutset
Why Does It Work?
Prim As a Special Case of Red-Blue Principle

- The tree that we grow creates a cut:

  "{Nodes inside tree} and {Nodes outside tree}"

- Prim applies the blue rule to this cutset
Prim’s Algorithm: Implementation

- Each node keeps:
  - the **distance** from itself to the **growing tree**
  - **how to get there.**

I’m 7 away. C is the closest.

I can’t get to the tree in one edge.
Prim’s Algorithm: Implementation

• Each node keeps:
  – the distance from itself to the growing tree
  – how to get there.

• Choose the closest node, add it.

I’m 7 away. C is the closest.

I can’t get to the tree in one edge.
Prim’s Algorithm: Implementation

• Each node keeps:
  – the **distance** from itself to the **growing tree**
  – **how to get there**.

• Choose the closest node, add it.
Prim’s Algorithm: Implementation

• Each node keeps:
  – the **distance** from itself to the **growing tree**
  – how to get there.

• Choose the closest node, add it.
• Update stored info.
Prim’s Algorithm: Implementation

Every node has a value and a parent

- $x$: Can’t reach $x$ yet
- $x$: $x$ is “active”
- $x$: Can reach $x$

$d[x]$: distance of $x$ from the growing tree

$a \rightarrow b$: Predecessor of $b$ is $a$
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- \( x \): Can’t reach \( x \) yet
- \( x \): \( x \) is “active”
- \( x \): Can reach \( x \)

\( d[x] \): distance of \( x \) from the growing tree

\( a \rightarrow b \): predecessor of \( b \) is \( a \)
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- $x$: Can’t reach $x$ yet
- $x$: $x$ is “active”
- $x$: Can reach $x$

$d[x]$: distance of $x$ from the growing tree

$a \xrightarrow{} b$: predecessor of $b$ is $a$
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- $x$: Can’t reach $x$ yet
- $x$: $x$ is “active”
- $x$: Can reach $x$

$d[x]$: distance of $x$ from the growing tree

Predecessor of $b$ is $a$
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- $x$: Can’t reach $x$ yet
- $x$: $x$ is “active”
- $x$: Can reach $x$

$d[x]$: distance of $x$ from the growing tree

$a \leftarrow b$: predecessor of $b$ is $a$
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- $x$: Can’t reach $x$ yet
- $x$: $x$ is “active”
- $x$: Can reach $x$

$d[x]$: distance of $x$ from the growing tree

$a$ ← $b$: predecessor of $b$ is $a$
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- $x$: Can’t reach $x$ yet
- $x$: $x$ is “active”
- $x$: Can reach $x$

$d[x]$: distance of $x$ from the growing tree

predecessor of $b$ is $a$
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- $x$: Can’t reach $x$ yet
- $x$: $x$ is “active”
- $x$: Can reach $x$

$d[x]$: distance of $x$ from the growing tree

$a \leftarrow b$: predecessor of $b$ is $a$
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- $x$: Can’t reach $x$ yet
- $x$: $x$ is “active”
- $x$: Can reach $x$

$d[x]$: distance of $x$ from the growing tree

$a \rightarrow b$: predecessor of $b$ is $a$
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- **x** Can’t reach x yet
- **x** x is “active”
- **x** Can reach x

- **d[x]** distance of x from the growing tree
- **a** ← **b** predecessor of b is a
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor.

- Can’t reach $x$ yet
- $x$ is “active”
- Can reach $x$

**$d[x]$**: distance of $x$ from the growing tree

**$a$** ← **$b$**: predecessor of $b$ is $a$
Prim’s Algorithm: Implementation
Every node has an estimated distance and a predecessor

- $x$: Can’t reach $x$ yet
- $x$: $x$ is “active”
- $x$: Can reach $x$

- $d[x]$: distance of $x$ from the growing tree
- $a \rightarrow b$: predecessor of $b$ is $a$
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- $x$ Can’t reach $x$ yet
- $x$ is “active”
- Can reach $x$

$d[x]$ distance of $x$ from the growing tree

$a \leftarrow b$ predecessor of $b$ is $a$
**Prim’s Algorithm: Implementation**

Every node has an estimated distance and a predecessor.

- $d[x]$ distance of $x$ from the growing tree
- $a \rightarrow b$ predecessor of $b$ is $a$
- $x$ Can’t reach $x$ yet
- $x$ is “active”
- $x$ Can reach $x$
Prim’s Algorithm: Implementation

Every node has an estimated distance and a predecessor

- $d[x]$ distance of $x$ from the growing tree
- Can’t reach $x$ yet
- $x$ is “active”
- Can reach $x$
- Predecessor of $b$ is $a$
Prim’s Algorithm: Implementation
Every node has an estimated distance and a predecessor

- $x$: Can’t reach $x$ yet
- $x$: $x$ is “active”
- $x$: Can reach $x$

$d[x]$: distance of $x$ from the growing tree

- $a \leftarrow b$: predecessor of $b$ is $a$
Prim’s Algorithm: Implementation

Very similar to Dijkstra’s algorithm!

In updating a neighbor $v$ of the current node $u$, the potential new distance is:

- **Prim**: $w(u, v)$
- **Dijkstra**: $d[u] + w(u, v)$

To see the difference, consider:

Prim’s algorithm can be implemented in $O(m \log(n))$ time, like Dijkstra’s $n$ nodes and $m$ edges.
Prim’s Algorithm: Implementation

**procedure** Prim(\(V, E, w\))

Create an empty priority queue \(Q\) and pick an arbitrary node \(s\)

\(T \leftarrow \emptyset\) \hspace{1em} // Initialize an empty tree

\(d[s] \leftarrow 0\) and \(d[v] \leftarrow \infty\) for every \(v \neq s\) \hspace{1em} // Initialize distance

**for** \(v \in V\) **do** Insert\((Q, v, d[v])\) \hspace{1em} // Put all nodes in the queue

**while** (NotEmpty\((Q)\)) \hspace{1em} // Process until every node is added to the tree

\hspace{2em} // Choose the closest node and add it to the tree

\(u \leftarrow \text{DeleteMin}(Q); T \leftarrow T \cup \{(u, \text{pred}[u])\}\)

**for** \(\{u, v\} \in E\) incident to \(u\) **do** \hspace{1em} // Look at neighbors

\hspace{2em} **if** \(d[v] > w(u, v)\) **then** \hspace{1em} // Update if new distance is better

\hspace{4em} \(d[v] \leftarrow w(u, v); u \leftarrow \text{UpdateKey}(Q, v, d[v]); \text{pred}[v] \leftarrow u\)

**return** \(T\)
Agenda

1. Dijkstra’s algorithm

2. Minimum spanning Tree

3. Prim

4. Kruskal
**Kruskal’s Algorithm**

**Idea:** - Greedily add the cheapest edge that *won’t cause a cycle*
Kruskal’s Algorithm

**Idea:** - Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

**Idea:** - Greedily add the cheapest edge that **won’t cause a cycle**
Kruskal’s Algorithm

Idea: - Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

Idea: -Greedily add the cheapest edge that won’t cause a cycle
**Kruskal’s Algorithm**

**Idea:** - Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

Idea: -Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

Idea: - Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

Idea: -Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

Idea: - Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

Idea: -Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

**Idea:** - Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

**Idea:** - Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

**Idea:** - Greedily add the cheapest edge that won’t cause a cycle
Kruskal’s Algorithm

Idea: -Greedily add the cheapest edge that won’t cause a cycle
Another View of Kruskal’s Algorithm

At each step we actually maintain a forest
Another View of Kruskal’s Algorithm

At each step we actually maintain a forest

Diagram showing a network with nodes A, B, C, D, E, F, G, and H, with edges labeled with weights such as 1, 2, 4, 7, 8, 9, 10, and 11.
Another View of Kruskal’s Algorithm

When we add an edge, we merge two trees
Another View of Kruskal’s Algorithm

When we add an edge, we merge two trees
Another View of Kruskal’s Algorithm

When we add an edge, we merge two trees
We never add an edge that creates a cycle
Why Does It Work?
Kruskal As a Special Case of Red-Blue Principle

**Case 1:** The endpoints of the edge stay in different trees

Apply **blue rule** to the cutset created by one of the two trees

![Graph with nodes A, B, C, D, E, F, G, H, I and edges labeled with numbers 1 to 14. The graph is a network of connected nodes with varying edge weights.]
Why Does It Work?
Kruskal As a Special Case of Red-Blue Principle

Case 1: The endpoints of the edge stay in different trees

Apply blue rule to the cutset created by one of the two trees
Why Does It Work?

Kruskal As a Special Case of Red-Blue Principle

**Case 2:** The endpoints of the edge stay in the same tree

- Apply **red rule** to the cycle created by adding the edge to the tree
Why Does It Work?
Kruskal As a Special Case of Red-Blue Principle

**Case 2**: The endpoints of the edge stay in the same tree

Apply red rule to the cycle created by adding the edge to the tree
Kruskal’s Algorithm: Implementation Based on Union-Find Data Structure

- Used for storing collections of sets

\begin{align*}
\text{MakeSet}(u) & : \text{Create a set } \{u\} \\
\text{Find}(u) & : \text{Return the set that } u \text{ is in} \\
\text{Union}(u,v) & : \text{Merge the sets that } u \text{ and } v \text{ are in}
\end{align*}

\begin{align*}
\text{MakeSet}(x) & \\
\text{Union}(x,y) & \\
\text{MakeSet}(y) &
\end{align*}
Kruskal’s Algorithm: Implementation
Based on Union-Find Data Structure

- Used for storing collections of sets

\[
\text{MakeSet}(u) : \text{Create a set } \{u\}
\]

\[
\text{Find}(u) : \text{Return the set that } u \text{ is in}
\]

\[
\text{Union}(u, v) : \text{Merge the sets that } u \text{ and } v \text{ are in}
\]

\[
\text{Find}(x)
\]

\[
\text{Union}(x, y)
\]

Can implement MakeSet, Find, and Union in essentially constant time
Kruskal’s Algorithm: Implementation

procedure Kruskal(V, E, w)
Sort edges by weight so that \( w(e_1) \leq \cdots \leq w(e_m) \)

\[ T \leftarrow \emptyset \] // Initialize an empty tree

for \( v \in V \) do MakeSet(v) // Put each node in its own tree in the forest

for \( i \leftarrow 1 \) to \( m \) do // Go through the edges in sorted order

\( (u, v) \leftarrow e_i \)

if \( \text{Find}(u) \neq \text{Find}(v) \) then // If \( u \) and \( v \) are not in the same tree

\[ T \leftarrow T \cup \{e_i\} \]

Union\( (u, v) \) // merge \( u \)'s tree with \( v \)'s tree

return \( T \)
Once More ...

To start, every vertex is in its own tree.
Once More ...

Then start merging
Once More ...

Then start merging
Once More ...

Then start merging
Once More ...

Then start merging
Once More ...

Then start merging
Once More ...

Then start merging
Once More ...

Then start merging
Once More ...

Then start merging

Stop when we have a big tree
Running Time

$n$ nodes and $m$ edges

- Sorting $m$ edges takes $O(m \log(n))$ time

MakeSet: $O(n)$ calls

Put each node in its own set

Find: $O(m)$ calls

For each edge, find its end points

Union: $O(n)$ calls

We’ll never add more than $n - 1$ edges to the tree
So we’ll never call Union more than $n - 1$ times

Running time is $O(m \log(n))$
Exercise: Maximum Spanning Tree

Question: Find the maximum spanning tree
Exercise: Maximum Spanning Tree

Question: Find the **maximum** spanning tree
Bottleneck Spanning Tree:
Cost is measured as the weight of the **heaviest** edge of the tree
Exercise: Minimum Bottleneck Spanning Tree

This has cost 14
Exercise: Minimum Bottleneck Spanning Tree

**Question 1:** Given a value $V$, determine if the graph has a bottleneck spanning tree of cost at most $V$

**Question 2:** Find a Minimum Bottleneck Spanning Tree
Exercise: Water Supply

Cost to dig a well at house B

Cost to build a pipe between house B and C

A house gets water by either having a well there, or by being connected to a house of well via pipes
Exercise: Water Supply

This is a way to build pipes/wells, so that everyone has water.
It has cost 25.
Exercise: Water Supply

Here’s a better way to build pipes/wells, and everybody still gets water

It has cost 15
Exercise: Water Supply

Question: Build wells/pipes so that everybody has water, but minimize the cost
A few edges must be included in our spanning tree

**Question:** Find a spanning tree subject that contains the required edges and has as small cost as possible