# Discrete Mathematical Models

**Lecture 25**

Kane Townsend

Semester 2, 2024

# D2: Weighted Graphs

# D2. Weighted Graphs

| | |
|---|---|
| Text Reference (Epp) | 3ed: Chapter 11 |
| | 4ed: Chapter 10 |
| | 5ed: Chapter 10 |

Some of the work is not covered in Epp.
but is based on some examples from:
Kolman, Busby & Ross *Discrete Mathematical Structures*
Johnsonbaugh *Discrete Mathematics*

## Weighted graphs

- In some applications of graphs there is a non-negative number associated with each edge, known as the **weight** of the edge.

# Weighted graphs

- In some applications of graphs there is a non-negative number associated with each edge, known as the **weight** of the edge.
- So a **weighted graph** is a graph $G$ together with a **weight function** $\text{weight} : E(G) \to \mathbb{Q}_+$.

# Weighted graphs

- In some applications of graphs there is a non-negative number associated with each edge, known as the **weight** of the edge.
- So a **weighted graph** is a graph $G$ together with a **weight function** $\text{weight} : E(G) \to \mathbb{Q}_+$.
- Examples include:
  - **Airline networks:** Vertices are airports used; edges are services or potential services between airports; weights are air miles or cost of flights.

# Weighted graphs

- In some applications of graphs there is a non-negative number associated with each edge, known as the **weight** of the edge.
- So a **weighted graph** is a graph $G$ together with a **weight function** $\text{weight} : E(G) \to \mathbb{Q}_+$.
- Examples include:
  - **Airline networks:** Vertices are airports used; edges are services or potential services between airports; weights are air miles or cost of flights.
  - **Fibre telecommunications:** Vertices are primary routing stations; there are edges between every pair of vertices (a complete graph); weights are the costs of laying fibre between stations.

# Weighted graphs

- In some applications of graphs there is a non-negative number associated with each edge, known as the **weight** of the edge.
- So a **weighted graph** is a graph $G$ together with a **weight function** $\text{weight} : E(G) \to \mathbb{Q}_+$.
- Examples include:
  - **Airline networks:** Vertices are airports used; edges are services or potential services between airports; weights are air miles or cost of flights.
  - **Fibre telecommunications:** Vertices are primary routing stations; there are edges between every pair of vertices (a complete graph); weights are the costs of laying fibre between stations.
  - **The internet:** Vertices are internet nodes; edges are all direct connections between nodes; weights are times (in milliseconds) for a packet to travel across a connection.

# Problems on weighted graphs

We define the **(total) weight** of a subgraph $S$ of a weighted graph $G$ to be the sum of the weights of all its edges.

# Problems on weighted graphs

We define the **(total) weight** of a subgraph $S$ of a weighted graph $G$ to be the sum of the weights of all its edges.

There are several well-studied optimisation problems relating to subgraphs of weighted graphs. We will look at just three:

# Problems on weighted graphs

We define the **(total) weight** of a subgraph $S$ of a weighted graph $G$ to be the sum of the weights of all its edges.

There are several well-studied optimisation problems relating to subgraphs of weighted graphs. We will look at just three:

**Minimal spanning tree**:

Find a spanning tree of least possible total weight.

# Problems on weighted graphs

We define the **(total) weight** of a subgraph $S$ of a weighted graph $G$ to be the sum of the weights of all its edges.

There are several well-studied optimisation problems relating to subgraphs of weighted graphs. We will look at just three:

**Minimal spanning tree**:

Find a spanning tree of least possible total weight.

**Travelling salesman problem**:

Find a Hamilton circuit of least possible total weight.

# Problems on weighted graphs

We define the **(total) weight** of a subgraph $S$ of a weighted graph $G$ to be the sum of the weights of all its edges.

There are several well-studied optimisation problems relating to subgraphs of weighted graphs. We will look at just three:

**Minimal spanning tree**:

Find a spanning tree of least possible total weight.

**Travelling salesman problem**:

Find a Hamilton circuit of least possible total weight.

**Shortest path**:

Find a path between two given vertices that has least possible total weight.

# Problems on weighted graphs

We define the **(total) weight** of a subgraph $S$ of a weighted graph $G$ to be the sum of the weights of all its edges.

There are several well-studied optimisation problems relating to subgraphs of weighted graphs. We will look at just three:

**Minimal spanning tree**:

Find a spanning tree of least possible total weight.

**Travelling salesman problem**:

Find a Hamilton circuit of least possible total weight.

**Shortest path**:

Find a path between two given vertices that has least possible total weight.

We will also look at a different kind of problem on a weighted **directed** graph: Maximal Flow. Details later.

# Minimal Spanning Tree

# Kruskal's algorithm for minimal spanning tree

Kruskal's algorithm is an obvious modification to the spanning-tree-finding algorithm we discussed previously. We simply choose the 'cheapest' possible edge at each step (proof: Theorem 10.6.2 5th Ed. of Epp):

# Kruskal's algorithm for minimal spanning tree

Kruskal's algorithm is an obvious modification to the spanning-tree-finding algorithm we discussed previously. We simply choose the 'cheapest' possible edge at each step (proof: Theorem 10.6.2 5th Ed. of Epp):

**Input:** Weighted connected graph $G$ with $n$ vertices.

**Output:** Minimal spanning tree $T$ for $G$ (w.r.t. total weight).
Total weight $W$ of this tree.

# Kruskal's algorithm for minimal spanning tree

Kruskal's algorithm is an obvious modification to the spanning-tree-finding algorithm we discussed previously. We simply choose the 'cheapest' possible edge at each step (proof: Theorem 10.6.2 5th Ed. of Epp):

**Input:** Weighted connected graph $G$ with $n$ vertices.
**Output:** Minimal spanning tree $T$ for $G$ (w.r.t. total weight).
Total weight $W$ of this tree.

**Method:**

1. Initialise $T$ to have all the vertices of $G$ but no edges.
   Initialise $W$ to 0.

# Kruskal's algorithm for minimal spanning tree

Kruskal's algorithm is an obvious modification to the spanning-tree-finding algorithm we discussed previously. We simply choose the 'cheapest' possible edge at each step (proof: Theorem 10.6.2 5th Ed. of Epp):

**Input:** Weighted connected graph $G$ with $n$ vertices.

**Output:** Minimal spanning tree $T$ for $G$ (w.r.t. total weight).
Total weight $W$ of this tree.

**Method:**

1. Initialise $T$ to have all the vertices of $G$ but no edges. Initialise $W$ to 0.
2. From the edges of currently in $G$ pick one, $e$, of least weight and remove it from $G$.

# Kruskal's algorithm for minimal spanning tree

Kruskal's algorithm is an obvious modification to the spanning-tree-finding algorithm we discussed previously. We simply choose the 'cheapest' possible edge at each step (proof: Theorem 10.6.2 5th Ed. of Epp):

**Input:** Weighted connected graph $G$ with $n$ vertices.
**Output:** Minimal spanning tree $T$ for $G$ (w.r.t. total weight).
Total weight $W$ of this tree.

**Method:**

1. Initialise $T$ to have all the vertices of $G$ but no edges. Initialise $W$ to 0.
2. From the edges of currently in $G$ pick one, $e$, of least weight and remove it from $G$.
3. If adding $e$ to $T$ does not create a circuit in $T$, add $e$ to $T$ and add weight($e$) to $W$.

# Kruskal's algorithm for minimal spanning tree

Kruskal's algorithm is an obvious modification to the spanning-tree-finding algorithm we discussed previously. We simply choose the 'cheapest' possible edge at each step (proof: Theorem 10.6.2 5th Ed. of Epp):

**Input:** Weighted connected graph $G$ with $n$ vertices.

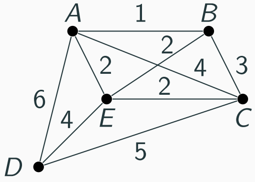
**Output:** Minimal spanning tree $T$ for $G$ (w.r.t. total weight). Total weight $W$ of this tree.

**Method:**

1. Initialise $T$ to have all the vertices of $G$ but no edges. Initialise $W$ to 0.
2. From the edges of currently in $G$ pick one, $e$, of least weight and remove it from $G$.
3. If adding $e$ to $T$ does not create a circuit in $T$, add $e$ to $T$ and add weight($e$) to $W$.
4. Repeat steps 2 and 3 until $T$ has $n - 1$ edges.

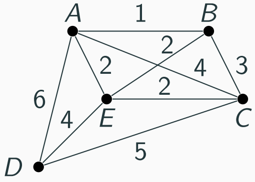
# Example: Applying Kruskal's algorithm

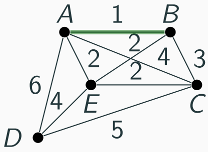
Find a minimal spanning tree for this weighted graph:



# Example: Applying Kruskal's algorithm

Find a minimal spanning tree for this weighted graph:

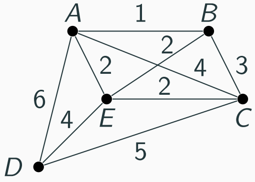


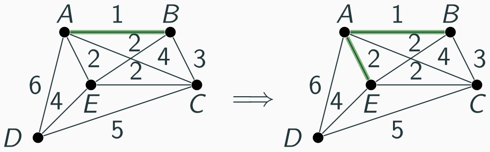




$W = 1$

# Example: Applying Kruskal's algorithm

Find a minimal spanning tree for this weighted graph:

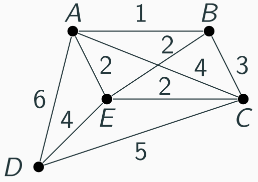


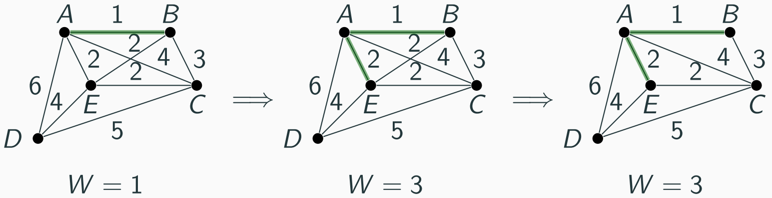




$W = 1$ $W = 3$

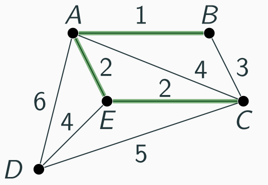
# Example: Applying Kruskal's algorithm

Find a minimal spanning tree for this weighted graph:



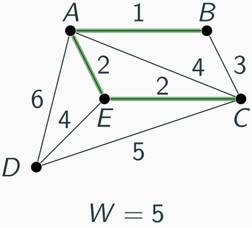



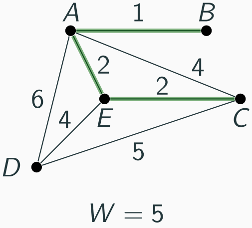
# Example: Applying Kruskal's algorithm (cont.)

$\Longrightarrow$ 


$W = 5$

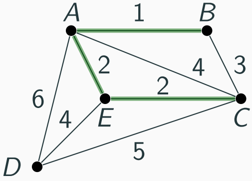
# Example: Applying Kruskal's algorithm (cont.)

$\Longrightarrow$ 


$\Longrightarrow$ 

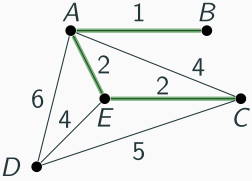
# Example: Applying Kruskal's algorithm (cont.)

$\Longrightarrow$



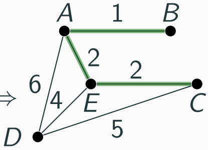

$W = 5$

$\Longrightarrow$




$W = 5$

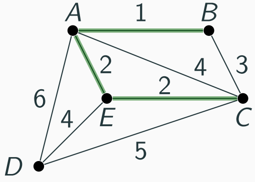
$\Longrightarrow$




$W = 5$

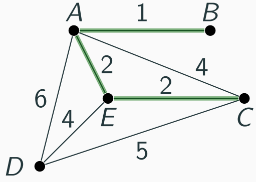
# Example: Applying Kruskal's algorithm (cont.)

$\Longrightarrow$



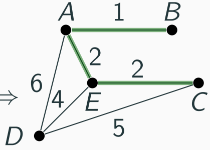

$W = 5$

$\Longrightarrow$



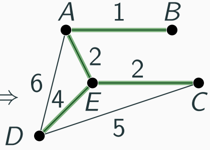

$W = 5$

$\Longrightarrow$



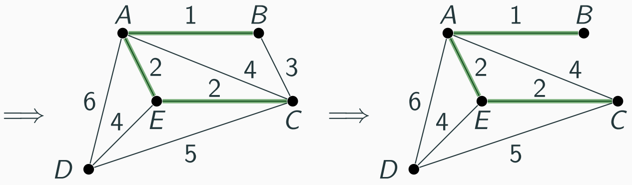

$W = 5$

$\Longrightarrow$

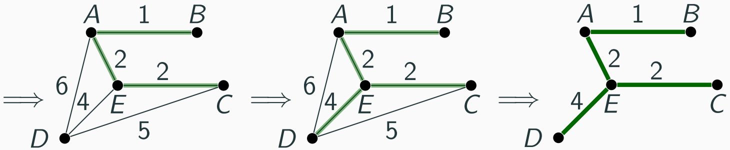



$W = 9$

# Example: Applying Kruskal's algorithm (cont.)




$W = 5$ $\qquad$ $W = 5$




$W = 5$ $\qquad$ $W = 9$ $\qquad$ $W = 9$

(The generated spanning tree)

# 'Greedy' algorithms

- A 'greedy' algorithm is an algorithm for an optimisation problem that attempts to find a globally optimal solution by finding a locally optimal solution at each step.

# ‘Greedy’ algorithms

- A ‘greedy’ algorithm is an algorithm for an optimisation problem that attempts to find a globally optimal solution by finding a locally optimal solution at each step.
- Kruskal’s algorithm, that we examined for solving the minimal spanning tree problem, is an example of a greedy algorithm because:

# ‘Greedy’ algorithms

- A ‘greedy’ algorithm is an algorithm for an optimisation problem that attempts to find a globally optimal solution by finding a locally optimal solution at each step.

- Kruskal’s algorithm, that we examined for solving the minimal spanning tree problem, is an example of a greedy algorithm because:

- Kruskal’s algorithm attempts to find a spanning tree of **least** possible *total* weight by, at each step, adding an edge of **least** possible (*individual*) weight (from amongst all unused edges that would not create a circuit).

# 'Greedy' algorithms

- A 'greedy' algorithm is an algorithm for an optimisation problem that attempts to find a globally optimal solution by finding a locally optimal solution at each step.

- Kruskal's algorithm, that we examined for solving the minimal spanning tree problem, is an example of a greedy algorithm because:

- Kruskal's algorithm attempts to find a spanning tree of **least** possible *total* weight by, at each step, adding an edge of **least** possible (*individual*) weight (from amongst all unused edges that would not create a circuit).

- Kruskal's algorithm always succeeds! (Non-obvious theorem omitted)

  That is, it always finds a minimal spanning tree, given any weighted connected (finite) graph.

# ‘Nearest neighbour’ algorithm

# The ‘Travelling salesman’ problem

- The salesman needs to visit $n$ towns on a shortest possible circuit.

# The ‘Travelling salesman’ problem

- The salesman needs to visit $n$ towns on a shortest possible circuit.
- Given: a table of distances between every pair of towns.

# The ‘Travelling salesman’ problem

- The salesman needs to visit $n$ towns on a shortest possible circuit.
- Given: a table of distances between every pair of towns.
- **Model:** Graph $K_n$ with towns as vertices and edges weighted by the the inter-town distances.

  Find a Hamilton circuit of minimum possible total weight.

# The 'Nearest Neighbour' algorithm

**Input:** Weighted complete graph $G$ with $n$ vertices.

**Output:** Hamilton circuit for $G$ as a list $L$ of vertices.
Total weight $W$ of this circuit.

# The 'Nearest Neighbour' algorithm

**Input:** Weighted complete graph $G$ with $n$ vertices.

**Output:** Hamilton circuit for $G$ as a list $L$ of vertices.
Total weight $W$ of this circuit.

**Method:** 1. Initialise $L$ to the empty list, $W$ to 0 and index $i$ to 1

# The ‘Nearest Neighbour’ algorithm

**Input:** Weighted complete graph $G$ with $n$ vertices.

**Output:** Hamilton circuit for $G$ as a list $L$ of vertices.
Total weight $W$ of this circuit.

**Method:**
1. Initialise $L$ to the empty list, $W$ to 0 and index $i$ to 1
2. Choose any vertex and append it to $L$ as $L(1)$.

# The ‘Nearest Neighbour’ algorithm

**Input:** Weighted complete graph $G$ with $n$ vertices.

**Output:** Hamilton circuit for $G$ as a list $L$ of vertices.
Total weight $W$ of this circuit.

**Method:** 1. Initialise $L$ to the empty list, $W$ to 0 and index $i$ to 1
2. Choose any vertex and append it to $L$ as $L(1)$.
3. From all vertices in $G$ but not in $L$, choose a vertex $v$ such that weight$(L(i), v)$ is as small as possible. ($v$ is a ‘nearest neighbour’ to $L(i)$).

# The ‘Nearest Neighbour’ algorithm

**Input:** Weighted complete graph $G$ with $n$ vertices.

**Output:** Hamilton circuit for $G$ as a list $L$ of vertices.
Total weight $W$ of this circuit.

**Method:** 1. Initialise $L$ to the empty list, $W$ to 0 and index $i$ to 1

2. Choose any vertex and append it to $L$ as $L(1)$.

3. From all vertices in $G$ but not in $L$, choose a vertex $v$ such that weight$(L(i), v)$ is as small as possible. ($v$ is a ‘nearest neighbour’ to $L(i)$).

4. Add weight$(L(i), v)$ to $W$. Increment $i$ by 1. Append $v$ to $L$ as $L(i)$.

# The ‘Nearest Neighbour’ algorithm

**Input:** Weighted complete graph $G$ with $n$ vertices.

**Output:** Hamilton circuit for $G$ as a list $L$ of vertices.
Total weight $W$ of this circuit.

**Method:** 1. Initialise $L$ to the empty list, $W$ to 0 and index $i$ to 1

2. Choose any vertex and append it to $L$ as $L(1)$.

3. From all vertices in $G$ but not in $L$, choose a vertex $v$ such that weight$(L(i), v)$ is as small as possible. ($v$ is a ‘nearest neighbour’ to $L(i)$).

4. Add weight$(L(i), v)$ to $W$. Increment $i$ by 1. Append $v$ to $L$ as $L(i)$.

5. Repeat steps 3 and 4 until $i = n$.

# The 'Nearest Neighbour' algorithm

**Input:** Weighted complete graph $G$ with $n$ vertices.

**Output:** Hamilton circuit for $G$ as a list $L$ of vertices.
Total weight $W$ of this circuit.

**Method:** 1. Initialise $L$ to the empty list, $W$ to 0 and index $i$ to 1

2. Choose any vertex and append it to $L$ as $L(1)$.

3. From all vertices in $G$ but not in $L$, choose a vertex $v$ such that weight$(L(i), v)$ is as small as possible. ($v$ is a 'nearest neighbour' to $L(i)$).

4. Add weight$(L(i), v)$ to $W$. Increment $i$ by 1. Append $v$ to $L$ as $L(i)$.

5. Repeat steps 3 and 4 until $i = n$.

6. Add weight$(L(n), L(1))$ to $W$. Append $L(1)$ to $L$ as $L(n+1)$.

# Greedy algorithms don't always succeed

- The Nearest Neighbour algorithm is another example of a greedy algorithm because at each step it looks for the 'best way out', ignoring any possible disadvantages later on.

# Greedy algorithms don't always succeed

- The Nearest Neighbour algorithm is another example of a greedy algorithm because at each step it looks for the 'best way out', ignoring any possible disadvantages later on.
- But the Nearest Neighbour algorithm often fails to find the shortest Hamilton circuit.

# Greedy algorithms don't always succeed

- The Nearest Neighbour algorithm is another example of a greedy algorithm because at each step it looks for the 'best way out', ignoring any possible disadvantages later on.
- But the Nearest Neighbour algorithm often fails to find the shortest Hamilton circuit.
- Greed doesn't always pay !!

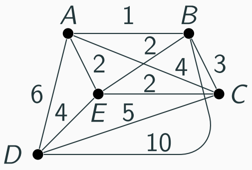
# Greedy algorithms don't always succeed

- The Nearest Neighbour algorithm is another example of a greedy algorithm because at each step it looks for the 'best way out', ignoring any possible disadvantages later on.
- But the Nearest Neighbour algorithm often fails to find the shortest Hamilton circuit.
- Greed doesn't always pay !!
- In fact, no efficient successful algorithm for the travelling salesman problem is known at this time. Finding one, or proving that none exists, is a major outstanding problem in mathematics.

# Example: Applying the Nearest Neighbour algorithm

Find a minimal Hamilton circuit for this weighted graph:

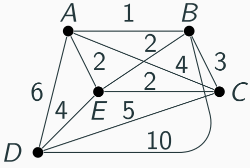
**Note:** This graph is as for the minimal spanning tree example but with the addition of an edge $BD$ to make it complete.



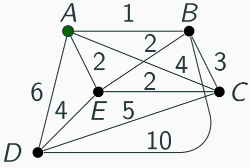
# Example: Applying the Nearest Neighbour algorithm

Find a minimal Hamilton circuit for this weighted graph:

**Note:** This graph is as for the minimal spanning tree example but with the addition of an edge $BD$ to make it complete.




Let's start at $A$ (we can start anywhere):

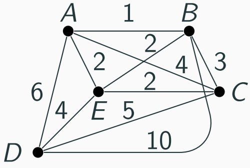



$L(1) = A, \ W = 0$

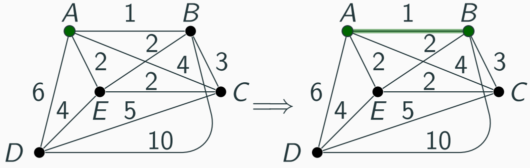
# Example: Applying the Nearest Neighbour algorithm

Find a minimal Hamilton circuit for this weighted graph:

**Note:** This graph is as for the minimal spanning tree example but with the addition of an edge $BD$ to make it complete.




Let's start at $A$ (we can start anywhere):



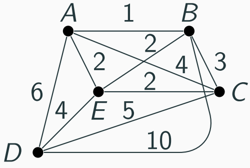

$L(1) = A,\ W = 0$

$L(2) = B,\ W = 1$

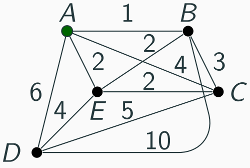
# Example: Applying the Nearest Neighbour algorithm

Find a minimal Hamilton circuit for this weighted graph:

**Note:** This graph is as for the minimal spanning tree example but with the addition of an edge $BD$ to make it complete.

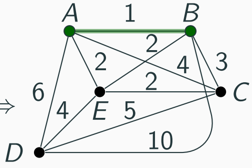



Let's start at $A$ (we can start anywhere):



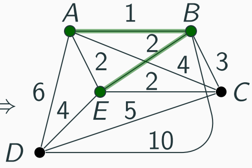

$L(1) = A, \ W = 0$

$\Longrightarrow$




$L(2) = B, \ W = 1$

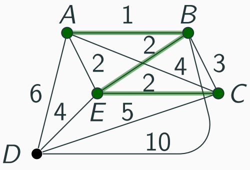
$\Longrightarrow$




$L(3) = E, \ W = 3$

# Example: Applying the Nearest Neighbour algorithm (cont.)

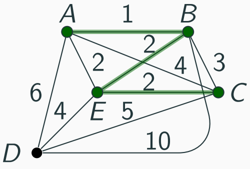
$\Longrightarrow$




$L(4) = C, \; W = 5$

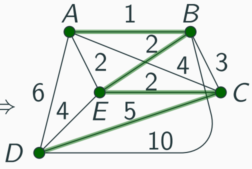
## Example: Applying the Nearest Neighbour algorithm (cont.)

$\Longrightarrow$




$L(4) = C, \ W = 5$

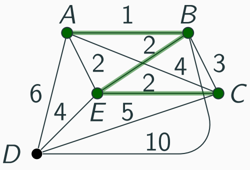
$\Longrightarrow$




$L(5) = D, \ W = 10$

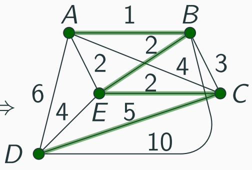
# Example: Applying the Nearest Neighbour algorithm (cont.)

$\Longrightarrow$



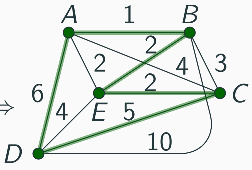

$L(4) = C, \; W = 5$

$\Longrightarrow$



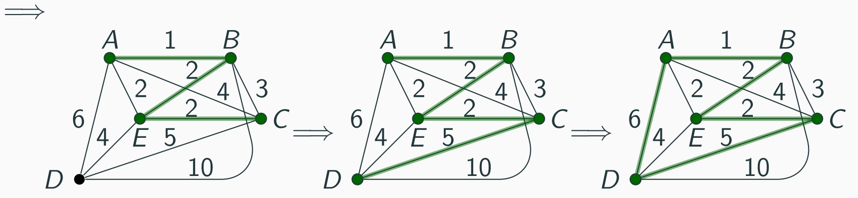

$L(5) = D, \; W = 10$

$\Longrightarrow$




$L(6) = A, \; W = 16$

# Example: Applying the Nearest Neighbour algorithm (cont.)

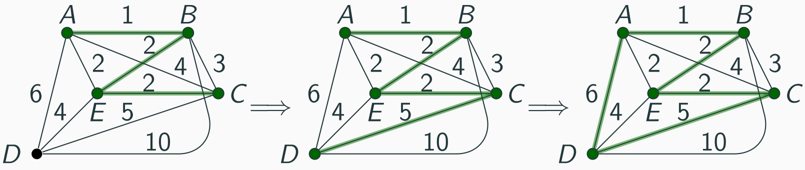



$L(4) = C,\ W = 5$ $\quad$ $L(5) = D,\ W = 10$ $\quad$ $L(6) = A,\ W = 16$

So Nearest Neighbour has generated a circuit of weight 16.

# Example: Applying the Nearest Neighbour algorithm (cont.)

$\Longrightarrow$



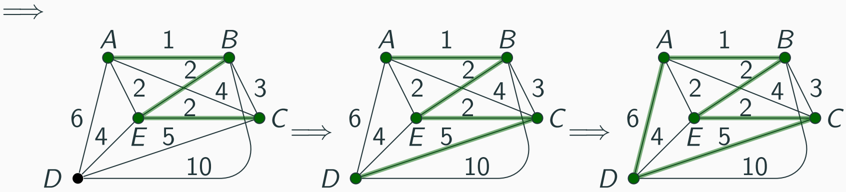

$L(4) = C,\ W = 5$ $\qquad$ $L(5) = D,\ W = 10$ $\qquad$ $L(6) = A,\ W = 16$

So Nearest Neighbour has generated a circuit of weight 16.

However this is NOT minimal.

The weight of a minimal circuit is in fact 15.

# Example: Applying the Nearest Neighbour algorithm (cont.)



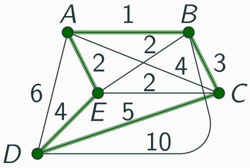

$L(4) = C, \; W = 5$ $\quad$ $L(5) = D, \; W = 10$ $\quad$ $L(6) = A, \; W = 16$

So Nearest Neighbour has generated a circuit of weight 16.

However this is NOT minimal.

The weight of a minimal circuit is in fact 15.

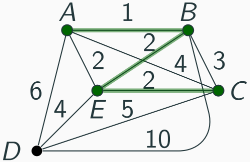
Here is a circuit of minimal weight:




$W = 15$

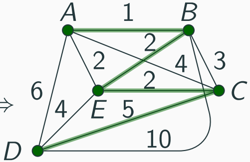
# Example: Applying the Nearest Neighbour algorithm (cont.)

$\Longrightarrow$



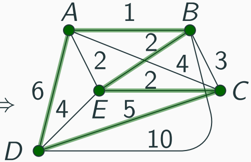

$L(4) = C,\ W = 5$

$\Longrightarrow$




$L(5) = D,\ W = 10$

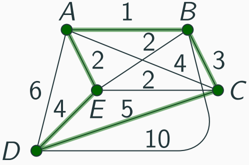
$\Longrightarrow$




$L(6) = A,\ W = 16$

So Nearest Neighbour has generated a circuit of weight 16.
However this is NOT minimal.
The weight of a minimal circuit is in fact 15.

Here is a circuit of minimal weight:



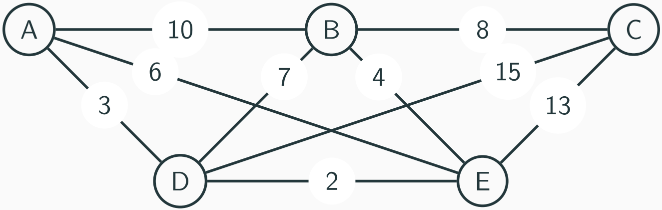

$W = 15$

Note that Nearest Neighbour may generate this circuit if we start at $D$ instead of $A$. Then $L(2) = E$ and it just depends on the choice for $L(3)$.

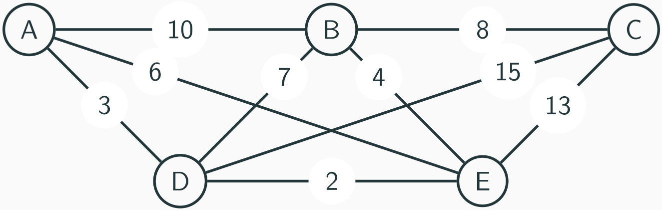
# Shortest Path

# Shortest Path — Introduction

Consider this weighted graph:



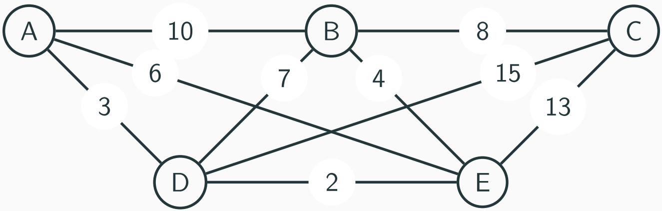
Consider this weighted graph:




**Problem:** Find a shortest path from A to C .

# Shortest Path — Introduction

Consider this weighted graph:

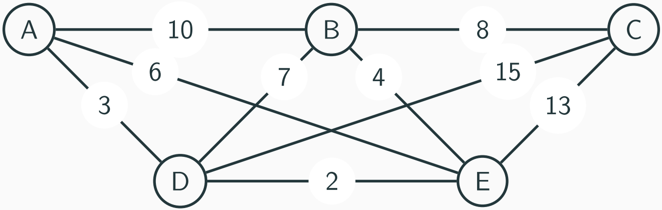



**Problem:** Find a shortest path from A to C .

That is, from the many paths from from A to C find one whose total weight is as small as possible.

# Shortest Path — Introduction

Consider this weighted graph:



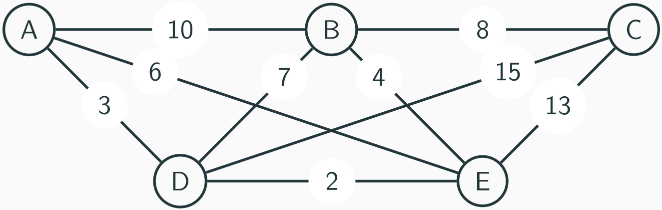

**Problem:** Find a shortest path from A to C .

That is, from the many paths from from A to C find one whose total weight is as small as possible.

For a small graph like this you can soon find a shortest path just by trying many alternatives (there are 10 or so simple A→C paths here).

# Shortest Path — Introduction

Consider this weighted graph:




**Problem:** Find a shortest path from A to C .

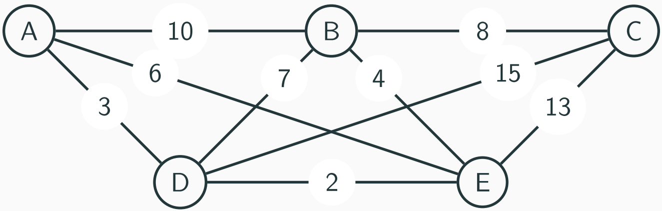
That is, from the many paths from from A to C find one whose total weight is as small as possible.

For a small graph like this you can soon find a shortest path just by trying many alternatives (there are 10 or so simple A→C paths here).

Can you spot a shortest path?

# Shortest Path — Introduction

Consider this weighted graph:




**Problem:** Find a shortest path from A to C .

That is, from the many paths from from A to C find one whose total weight is as small as possible.

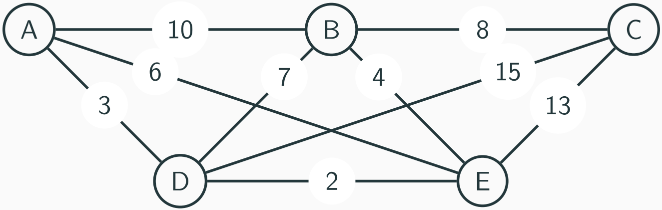
For a small graph like this you can soon find a shortest path just by trying many alternatives (there are 10 or so simple A→C paths here).

Can you spot a shortest path?

For large graphs this approach is not practical.

# Shortest Path — Introduction

Consider this weighted graph:




**Problem:** Find a shortest path from A to C .

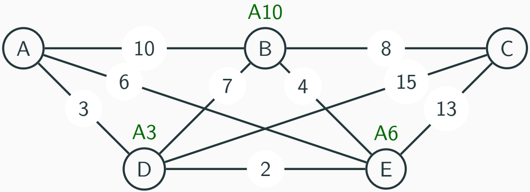
That is, from the many paths from from A to C find one whose total weight is as small as possible.

For a small graph like this you can soon find a shortest path just by trying many alternatives (there are 10 or so simple A→C paths here).

Can you spot a shortest path?

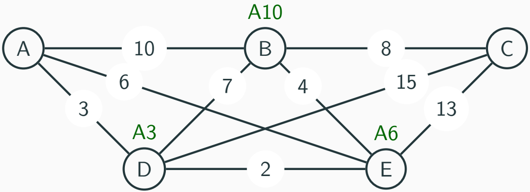
For large graphs this approach is not practical. We need an *algorithm*.

# Dijkstra's Algorithm




**Edsger Dijkstra 1930 - 2002**

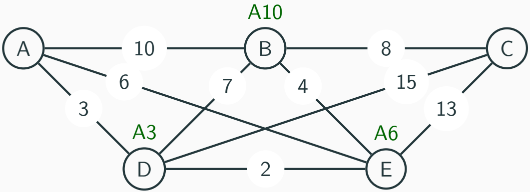
# Dijkstra's Algorithm




**Edsger Dijkstra 1930 - 2002**

Did you notice the special layout of the graph?

# Dijkstra's Algorithm

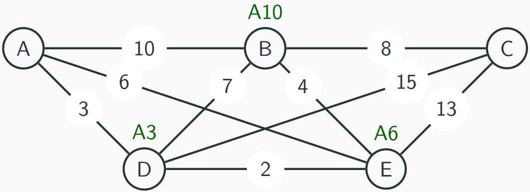



**Edsger Dijkstra 1930 - 2002**

Did you notice the special layout of the graph?

As will be explained, when using Dijkstra's algorithm (for finding a shortest path) markers and labels are inserted above vertices (green text in the above example).

# Dijkstra's Algorithm

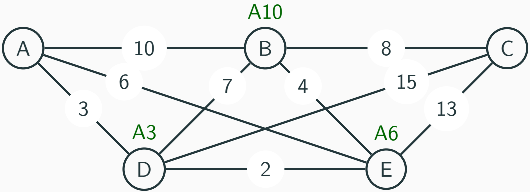



**Edsger Dijkstra 1930 - 2002**

Did you notice the special layout of the graph?

As will be explained, when using Dijkstra's algorithm (for finding a shortest path) markers and labels are inserted above vertices (green text in the above example). The special layout helps to avoid these annotations getting mixed up with the main graph information.

# Dijkstra's Algorithm



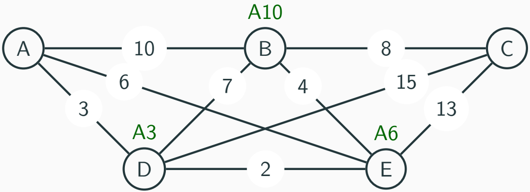

**Edsger Dijkstra 1930 - 2002**

Did you notice the special layout of the graph?

As will be explained, when using Dijkstra's algorithm (for finding a shortest path) markers and labels are inserted above vertices (green text in the above example). The special layout helps to avoid these annotations getting mixed up with the main graph information.

Dijkstra's algorithm has some similarities to Kruskal's algorithm for finding a minimum spanning tree but is a little more complicated.

# Dijkstra's Algorithm




**Edsger Dijkstra 1930 - 2002**

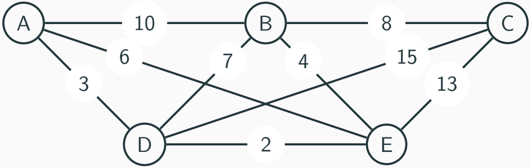
Did you notice the special layout of the graph?

As will be explained, when using Dijkstra's algorithm (for finding a shortest path) markers and labels are inserted above vertices (green text in the above example). The special layout helps to avoid these annotations getting mixed up with the main graph information.

Dijkstra's algorithm has some similarities to Kruskal's algorithm for finding a minimum spanning tree but is a little more complicated.
We launch straight in to demonstrating the algorithm on the above example, then we describe the algorithm.

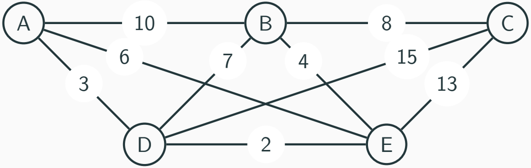
# Example 1 — Slide 1

We seek a minimal weight path from A to C in the graph below.



# Example 1 — Slide 1

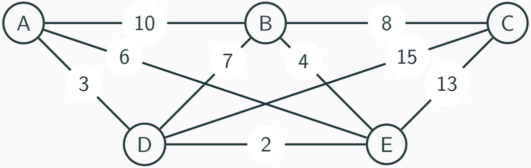
We seek a minimal weight path from A to C in the graph below.




This will yield the minimal 'distance', via graph edges, of C from A.

# Example 1 — Slide 1

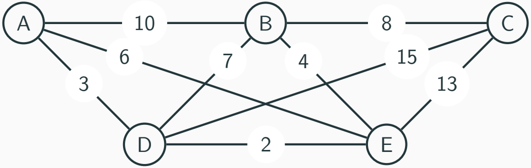
We seek a minimal weight path from A to C in the graph below.




This will yield the minimal 'distance', via graph edges, of C from A. The algorithm proceeds by progressively finding the minimal distance of many, possibly all, vertices from A.

# Example 1 — Slide 1

We seek a minimal weight path from A to C in the graph below.

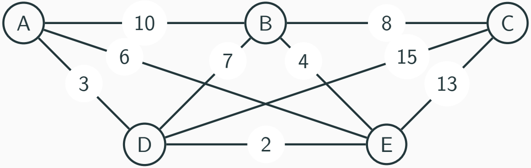



This will yield the minimal 'distance', via graph edges, of C from A. The algorithm proceeds by progressively finding the minimal distance of many, possibly all, vertices from A.

Label each vertex adjacent to A with its 'direct' distance from A:

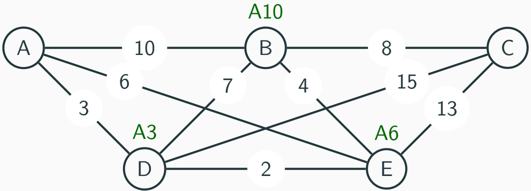
# Example 1 — Slide 1

We seek a minimal weight path from A to C in the graph below.




This will yield the minimal 'distance', via graph edges, of C from A. The algorithm proceeds by progressively finding the minimal distance of many, possibly all, vertices from A.

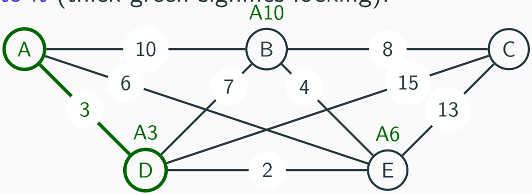
Label each vertex adjacent to A with its 'direct' distance from A:



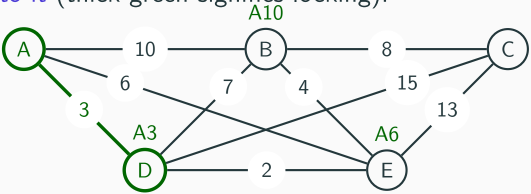
## Example 1 — Slide 2

Vertex D carries the smallest of these distances. ‘Lock it in’ and also the edge leading to it

Vertex D carries the smallest of these distances. ‘Lock it in’ and also the edge leading to it (thick green signifies locking):



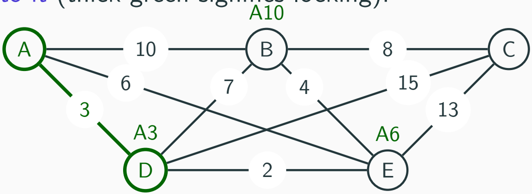
Vertex D carries the smallest of these distances. 'Lock it in' and also the edge leading to it (thick green signifies locking):




We call vertex D the current vertex $c$.

# Example 1 — Slide 2

Vertex D carries the smallest of these distances. ‘Lock it in’ and also the edge leading to it (thick green signifies locking):

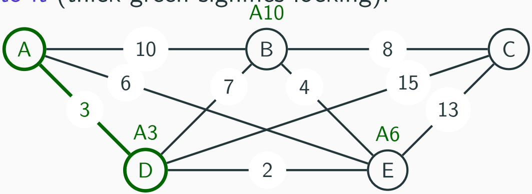



We call vertex D the current vertex $c$.

Mark and label any un-locked vertex $v$ adjacent to the current vertex $c$.

# Example 1 — Slide 2

Vertex D carries the smallest of these distances. 'Lock it in' and also the edge leading to it (thick green signifies locking):

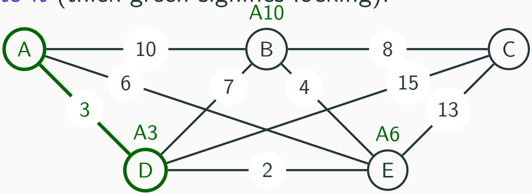



We call vertex D the current vertex $c$.

Mark and label any un-locked vertex $v$ adjacent to the current vertex $c$.

There are always three possibilities, and all three occur here:

Vertex D carries the smallest of these distances. 'Lock it in' and also the edge leading to it (thick green signifies locking):




We call vertex D the current vertex $c$.

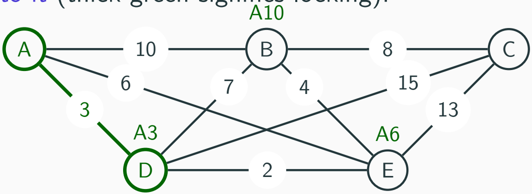
Mark and label any un-locked vertex $v$ adjacent to the current vertex $c$.

There are always three possibilities, and all three occur here:

**Possibility 1:** $v$ **is unmarked.** *e.g.* Vertex C .

# Example 1 — Slide 2

Vertex D carries the smallest of these distances. ‘Lock it in’ and also the edge leading to it (thick green signifies locking):




We call vertex D the current vertex $c$.

Mark and label any un-locked vertex $v$ adjacent to the current vertex $c$.

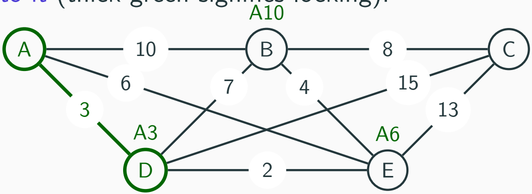
There are always three possibilities, and all three occur here:

**Possibility 1:** $v$ **is unmarked.** *e.g.* Vertex C .

Mark $v$ with $c$ and label with its distance from A via $c$.

## Example 1 — Slide 2

Vertex D carries the smallest of these distances. 'Lock it in' and also the edge leading to it (thick green signifies locking):




We call vertex D the current vertex $c$.

Mark and label any un-locked vertex $v$ adjacent to the current vertex $c$.

There are always three possibilities, and all three occur here:

**Possibility 1:** $v$ **is unmarked.** *e.g.* Vertex C .

Mark $v$ with $c$ and label with its distance from A via $c$.

So we write D18 above C.

**Possibility 2:** $v$ **needs updating.** *e.g.* Vertex E .

**Possibility 2:** $v$ **needs updating.** *e.g.* Vertex E .

The distance to $v$ via $c$ is less than the distance currently shown. Remark with $c$ and relabel with the shorter distance.

**Possibility 2:** $v$ **needs updating.** *e.g.* Vertex E .

The distance to $v$ via $c$ is less than the distance currently shown. Remark with $c$ and relabel with the shorter distance.

So we replace A6 above E with D5

**Possibility 3:** $v$ **needs no updating.** *e.g.* Vertex B.

**Possibility 2:** $v$ **needs updating.** *e.g.* Vertex E .

The distance to $v$ via $c$ is less than the distance currently shown. Remark with $c$ and relabel with the shorter distance.

So we replace A6 above E with D5

**Possibility 3:** $v$ **needs no updating.** *e.g.* Vertex B.

The distance to $v$ via $c$ is no less than the distance currently shown.

**Possibility 2:** $v$ **needs updating.** *e.g.* Vertex E .

The distance to $v$ via $c$ is less than the distance currently shown. Remark with $c$ and relabel with the shorter distance.

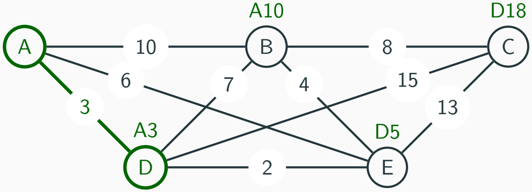
So we replace A6 above E with D5

**Possibility 3:** $v$ **needs no updating.** *e.g.* Vertex B.

The distance to $v$ via $c$ is no less than the distance currently shown.

So we leave the A10 above B as it is.

The annotated graph now looks like this:



# Example 1 — Slide 4

We now have three so-called ‘fringe’ vertices, B, C and E.

# Example 1 — Slide 4

We now have three so-called ‘fringe’ vertices, B, C and E.

Fringe vertices are those that have been annotated (marked and labelled) but not yet locked in.

# Example 1 — Slide 4

We now have three so-called 'fringe' vertices, B, C and E.

Fringe vertices are those that have been annotated (marked and labelled) but not yet locked in.

Now locate and lock in a fringe vertex $v$ with the lowest label value.

That's vertex E in our example since $5 < 10$ and $5 < 18$.

# Example 1 — Slide 4

We now have three so-called ‘fringe’ vertices, B, C and E.

Fringe vertices are those that have been annotated (marked and labelled) but not yet locked in.

Now locate and lock in a fringe vertex $v$ with the lowest label value.

That’s vertex E in our example since $5 < 10$ and $5 < 18$.

Also lock in its marked lead-in edge. That’s edge DE for us.

# Example 1 — Slide 4

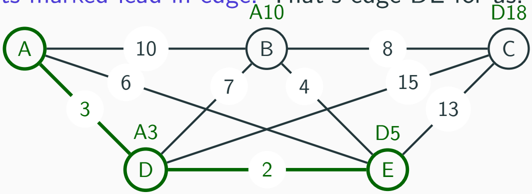
We now have three so-called 'fringe' vertices, B, C and E.

Fringe vertices are those that have been annotated (marked and labelled) but not yet locked in.

Now locate and lock in a fringe vertex $v$ with the lowest label value.

That's vertex E in our example since $5 < 10$ and $5 < 18$.

Also lock in its marked lead-in edge. That's edge DE for us.



# Example 1 — Slide 4

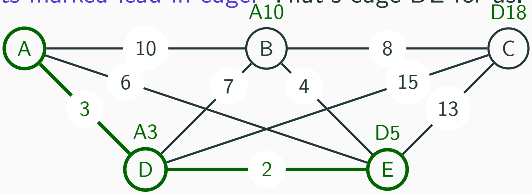
We now have three so-called 'fringe' vertices, B, C and E.

Fringe vertices are those that have been annotated (marked and labelled) but not yet locked in.

Now locate and lock in a fringe vertex $v$ with the lowest label value.

That's vertex E in our example since $5 < 10$ and $5 < 18$.

Also lock in its marked lead-in edge. That's edge DE for us.




The latest locked-in vertex E becomes the new current vertex.

# Example 1 — Slide 4

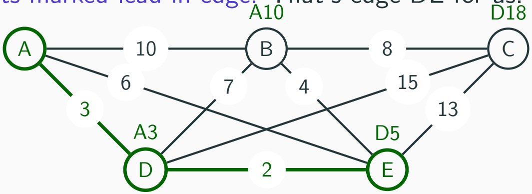
We now have three so-called 'fringe' vertices, B, C and E.

Fringe vertices are those that have been annotated (marked and labelled) but not yet locked in.

Now locate and lock in a fringe vertex $v$ with the lowest label value.

That's vertex E in our example since $5 < 10$ and $5 < 18$.

Also lock in its marked lead-in edge. That's edge DE for us.




The latest locked-in vertex E becomes the new current vertex.

We now repeat the process applied to the previous current vertex D.

Vertex E is adjacent to two un-locked vertices, B and C:

# Example 1 — Slide 5

Vertex E is adjacent to two un-locked vertices, B and C:

> Vertex B needs updating, since $5 + 4 = 9 < 10$.

# Example 1 — Slide 5

Vertex E is adjacent to two un-locked vertices, B and C:

> Vertex B needs updating, since $5 + 4 = 9 < 10$. So it is re-marked with E and re-labelled 9; *i.e.* A10 is replaced by E9.

# Example 1 — Slide 5

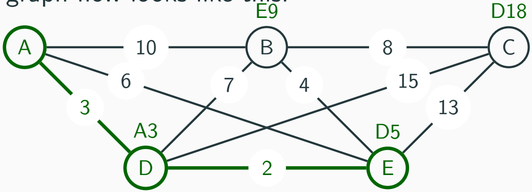
Vertex E is adjacent to two un-locked vertices, B and C:

> Vertex B needs updating, since $5 + 4 = 9 < 10$. So it is re-marked with E and re-labelled 9; *i.e.* A10 is replaced by E9.
>
> Vertex C does not need updating since $5 + 13 = 18$, and C is already labelled 18.

Vertex E is adjacent to two un-locked vertices, B and C:

> Vertex B needs updating, since $5 + 4 = 9 < 10$. So it is re-marked with E and re-labelled 9; *i.e.* A10 is replaced by E9.
>
> Vertex C does not need updating since $5 + 13 = 18$, and C is already labelled 18.

The updated graph now looks like this:

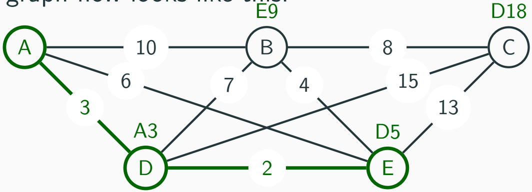


# Example 1 — Slide 5

Vertex E is adjacent to two un-locked vertices, B and C:

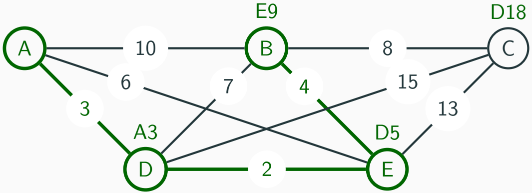
> Vertex B needs updating, since $5 + 4 = 9 < 10$. So it is re-marked with E and re-labelled 9; *i.e.* A10 is replaced by E9.
>
> Vertex C does not need updating since $5 + 13 = 18$, and C is already labelled 18.

The updated graph now looks like this:

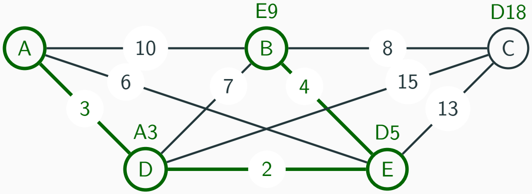



The lowest fringe value is now 9 on B, so we lock in B and its lead-in edge EB (next slide).

# Example 1 — Slide 6

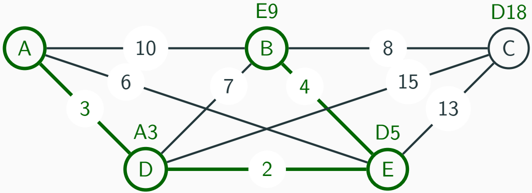


# Example 1 — Slide 6




The new current vertex is the just locked-in B.

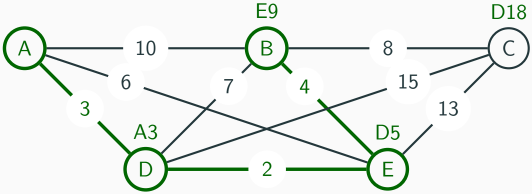
# Example 1 — Slide 6




The new current vertex is the just locked-in B.

There is only one vertex adjacent to B that has not already been locked in, namely C.

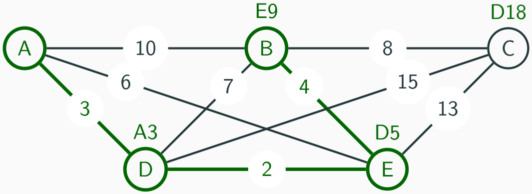
# Example 1 — Slide 6




The new current vertex is the just locked-in B.

There is only one vertex adjacent to B that has not already been locked in, namely C. This needs updating since $9 + 8 = 17 < 18$.

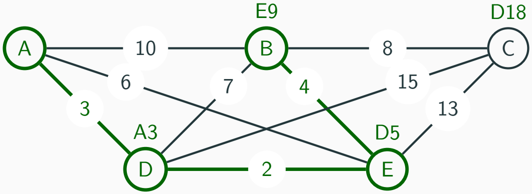
# Example 1 — Slide 6




The new current vertex is the just locked-in B.

There is only one vertex adjacent to B that has not already been locked in, namely C. This needs updating since $9 + 8 = 17 < 18$.
So D18 is replaced by B17.

# Example 1 — Slide 6

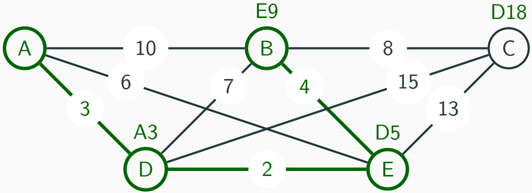



The new current vertex is the just locked-in B.

There is only one vertex adjacent to B that has not already been locked in, namely C. This needs updating since $9 + 8 = 17 < 18$.
So D18 is replaced by B17.

Since vertex C is the only vertex in the fringe it has the lowest label by default. So C and its lead-in edge BC are locked in.

# Example 1 — Slide 6



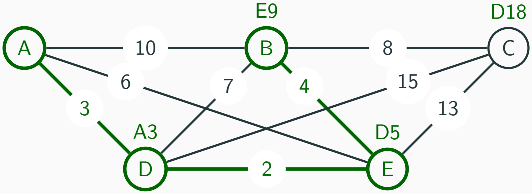

The new current vertex is the just locked-in B.

There is only one vertex adjacent to B that has not already been locked in, namely C. This needs updating since $9 + 8 = 17 < 18$.
So D18 is replaced by B17.

Since vertex C is the only vertex in the fringe it has the lowest label by default. So C and its lead-in edge BC are locked in.

We have now locked in the minimal distance 17 into our 'target' vertex C, so we can stop.

# Example 1 — Slide 6



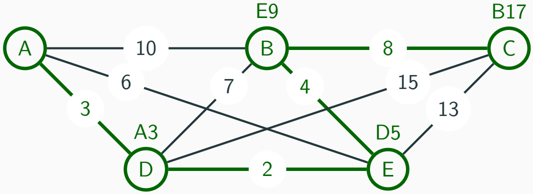

The new current vertex is the just locked-in B.

There is only one vertex adjacent to B that has not already been locked in, namely C. This needs updating since $9 + 8 = 17 < 18$.
So D18 is replaced by B17.

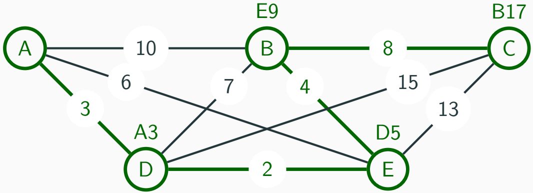
Since vertex C is the only vertex in the fringe it has the lowest label by default. So C and its lead-in edge BC are locked in.

We have now locked in the minimal distance 17 into our 'target' vertex C, so we can stop. The final diagram is .....

# Example 1 — Slide 7; Results and Comments



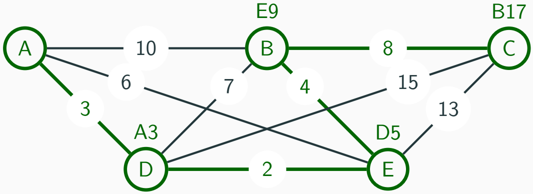
# Example 1 — Slide 7; Results and Comments




Some Observations:

- Besides the shortest path from A to C, the solution provides the shortest path to all the vertices along that path. For this example that happens to be the entire vertex set.

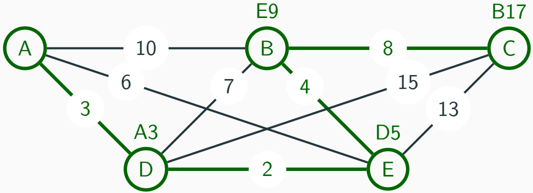
# Example 1 — Slide 7; Results and Comments




Some Observations:

- Besides the shortest path from A to C, the solution provides the shortest path to all the vertices along that path. For this example that happens to be the entire vertex set.
- Had there remained un-locked vertices, we could have continued the process until there were none.

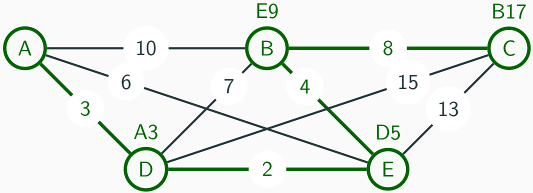
# Example 1 — Slide 7; Results and Comments




Some Observations:

- Besides the shortest path from A to C, the solution provides the shortest path to all the vertices along that path. For this example that happens to be the entire vertex set.
- Had there remained un-locked vertices, we could have continued the process until there were none.
- Since no vertex is locked twice, the locked edges form a tree. The required shortest path is the unique path on that tree from A to C.

# Example 1 — Slide 7; Results and Comments




Some Observations:

- Besides the shortest path from A to C, the solution provides the shortest path to all the vertices along that path. For this example that happens to be the entire vertex set.
- Had there remained un-locked vertices, we could have continued the process until there were none.
- Since no vertex is locked twice, the locked edges form a tree. The required shortest path is the unique path on that tree from A to C.
- With all vertices locked, the solution provides a spanning tree for the graph.

# Dijkstra's Algorithm — A Formal Description

**Input:** (a) Connected simple graph $G$. Vertices A, Z from $G$.

# Dijkstra's Algorithm — A Formal Description

**Input:** (a) Connected simple graph $G$. Vertices A, Z from $G$.

(b) Distance function $\mathsf{dist}\colon E(G) \to \mathbb{Q}^+$.

# Dijkstra's Algorithm — A Formal Description

**Input:** (a) Connected simple graph $G$. Vertices A, Z from $G$.

(b) Distance function $\mathsf{dist}\colon E(G) \to \mathbb{Q}^+$.

**Output:** (a) Tree $T$ containing A and Z as vertices.

$T$ is a subgraph of $G$.

# Dijkstra's Algorithm — A Formal Description

**Input:** (a) Connected simple graph $G$. Vertices A, Z from $G$.

(b) Distance function $\mathsf{dist}\colon E(G) \to \mathbb{Q}^+$.

**Output:** (a) Tree $T$ containing A and Z as vertices.

$T$ is a subgraph of $G$.

The unique path A$\to$Z in $T$ has minimal total distance of all paths A$\to$Z in $G$.

# Dijkstra's Algorithm — A Formal Description

**Input:** (a) Connected simple graph $G$. Vertices A, Z from $G$.
(b) Distance function $\mathrm{dist}\colon E(G) \to \mathbb{Q}^+$.

**Output:** (a) Tree $T$ containing A and Z as vertices.
$T$ is a subgraph of $G$.
The unique path A$\to$Z in $T$ has minimal total distance of all paths A$\to$Z in $G$.
(b) 'Labelling' $L\colon V(T) \to \mathbb{Q}_+$; $L(v) = \mathrm{min.dist}(\mathrm{A} \to v)$.

# Dijkstra's Algorithm — A Formal Description

**Input:** (a) Connected simple graph $G$. Vertices A, Z from $G$.
(b) Distance function $\mathsf{dist}\colon E(G) \to \mathbb{Q}^+$.

**Output:** (a) Tree $T$ containing A and Z as vertices.
$T$ is a subgraph of $G$.
The unique path A$\to$Z in $T$ has minimal total distance of all paths A$\to$Z in $G$.
(b) 'Labelling' $L\colon V(T) \to \mathbb{Q}_+$; $L(v) = \mathsf{min.dist}(\mathrm{A} \to v)$.

**Method:** 1. Initialize the tree $T$: Set $V(T) = \{\mathrm{A}\}$, $E(T) = \emptyset$.

# Dijkstra's Algorithm — A Formal Description

**Input:** (a) Connected simple graph $G$. Vertices A, Z from $G$.
(b) Distance function $\mathsf{dist}\colon E(G) \to \mathbb{Q}^+$.

**Output:** (a) Tree $T$ containing A and Z as vertices.
$T$ is a subgraph of $G$.
The unique path A$\to$Z in $T$ has minimal total distance of all paths A$\to$Z in $G$.
(b) 'Labelling' $L\colon V(T) \to \mathbb{Q}_+$; $L(v) = \mathsf{min.dist}(\mathsf{A} \to v)$.

**Method:**
1. Initialize the tree $T$: Set $V(T) = \{\mathsf{A}\}$, $E(T) = \emptyset$.
2. Initialize a 'Marking' function $M\colon V(G) \to V(G) \cup \{\mathsf{blank}\}$:
   Set $M(v) = \mathsf{blank}$ for all $v \in G$.

# Dijkstra's Algorithm — A Formal Description

**Input:** (a) Connected simple graph $G$. Vertices A, Z from $G$.

(b) Distance function $\mathsf{dist}\colon E(G) \to \mathbb{Q}^{+}$.

**Output:** (a) Tree $T$ containing A and Z as vertices.

$T$ is a subgraph of $G$.

The unique path A$\to$Z in $T$ has minimal total distance of all paths A$\to$Z in $G$.

(b) 'Labelling' $L\colon V(T) \to \mathbb{Q}_{+}$; $L(v) = \mathsf{min.dist}(\mathsf{A} \to v)$.

**Method:**

1. Initialize the tree $T$: Set $V(T) = \{\mathsf{A}\}$, $E(T) = \emptyset$.
2. Initialize a 'Marking' function $M\colon V(G) \to V(G) \cup \{\mathsf{blank}\}$:
   Set $M(v) = \mathsf{blank}$ for all $v \in G$.
3. Set $L(\mathsf{A}) = 0$. Set 'current vertex' $c$ to A.

# Dijkstra's Algorithm — A Formal Description

**Input:** (a) Connected simple graph $G$. Vertices A, Z from $G$.

(b) Distance function $\mathsf{dist}: E(G) \to \mathbb{Q}^+$.

**Output:** (a) Tree $T$ containing A and Z as vertices.

$T$ is a subgraph of $G$.

The unique path A$\to$Z in $T$ has minimal total distance of all paths A$\to$Z in $G$.

(b) 'Labelling' $L: V(T) \to \mathbb{Q}_+$; $L(v) = \mathsf{min.dist}(\mathsf{A} \to v)$.

**Method:**

1. Initialize the tree $T$: Set $V(T) = \{\mathsf{A}\}$, $E(T) = \emptyset$.
2. Initialize a 'Marking' function $M: V(G) \to V(G) \cup \{\mathsf{blank}\}$:
   Set $M(v) = \mathsf{blank}$ for all $v \in G$.
3. Set $L(\mathsf{A}) = 0$. Set 'current vertex' $c$ to A.

While $c \neq \mathsf{Z}$:

4. For each vertex $v$ adjacent to $c$ but not in $T$:
   If $v$ is unmarked (*i.e.* $M(v) = \mathsf{blank}$)
   or if $L(v) > L(c) + \mathsf{dist}(\{c, v\})$
   set $M(v) = c$, $L(v) = L(c) + \mathsf{dist}(\{c, v\})$.

5. From all marked $v \in G \setminus T$ (*i.e.* $M(v) \neq \textsf{blank}$ and $v \notin T$)
   (such $v$ are said to be 'on the fringe')
   select one, say $w$, with minimal $L(v)$.

# Dijkstra's Algorithm — A Formal Description (cont.)

5. From all marked $v \in G \setminus T$ (*i.e.* $M(v) \neq$ blank and $v \notin T$)
   (such $v$ are said to be 'on the fringe')
   select one, say $w$, with minimal $L(v)$.
6. Insert vertex $w$ and edge $\{M(w), w\}$ into the tree $T$.
   (I call this "locking in" $w$ and its lead-in edge.)

# Dijkstra's Algorithm — A Formal Description (cont.)

5. From all marked $v \in G \setminus T$ (*i.e.* $M(v) \neq \textsf{blank}$ and $v \notin T$)
   (such $v$ are said to be 'on the fringe')
   select one, say $w$, with minimal $L(v)$.
6. Insert vertex $w$ and edge $\{M(w), w\}$ into the tree $T$.
   (I call this "locking in" $w$ and its lead-in edge.)
7. Update $c$ to $w$. (*i.e.* make $w$ the new current vertex.)

# Dijkstra's Algorithm — A Formal Description (cont.)

5. From all marked $v \in G \setminus T$ (*i.e.* $M(v) \neq \text{blank}$ and $v \notin T$) (such $v$ are said to be 'on the fringe') select one, say $w$, with minimal $L(v)$.
6. Insert vertex $w$ and edge $\{M(w), w\}$ into the tree $T$. (I call this "locking in" $w$ and its lead-in edge.)
7. Update $c$ to $w$. (*i.e.* make $w$ the new current vertex.)

End of While Loop

# Dijkstra's Algorithm — A Formal Description (cont.)

5. From all marked $v \in G \setminus T$ (*i.e.* $M(v) \neq \text{blank}$ and $v \notin T$)
   (such $v$ are said to be 'on the fringe')
   select one, say $w$, with minimal $L(v)$.
6. Insert vertex $w$ and edge $\{M(w), w\}$ into the tree $T$.
   (I call this "locking in" $w$ and its lead-in edge.)
7. Update $c$ to $w$. (*i.e.* make $w$ the new current vertex.)

End of While Loop

End of Method

# Dijkstra's Algorithm — A Formal Description (cont.)

5. From all marked $v \in G \backslash T$ (*i.e.* $M(v) \neq$ blank and $v \notin T$) (such $v$ are said to be 'on the fringe') select one, say $w$, with minimal $L(v)$.
6. Insert vertex $w$ and edge $\{M(w), w\}$ into the tree $T$. (I call this "locking in" $w$ and its lead-in edge.)
7. Update $c$ to $w$. (*i.e.* make $w$ the new current vertex.)

End of While Loop

End of Method

This completes the formal description of Dijkstra's shortest path algorithm. Make some example weighted graphs and apply the algorithm for practice!