# MATH 322 – Graph Theory
# Fall Term 2021

## Notes for Lecture 14

Tuesday, October 26

# Recall: (Instances of) The Connector Problem

*from the Balakrishnan-Ranganathan book*

Problem 1. Various cities in a country are to be linked by means of roads. Given the various possibilities of connecting the cities and the costs involved, what is the most economical way of laying roads so that in the resulting road network any two cities are connected by a chain of roads?

Similar problems involve designing railroad networks, or water-line transports.

Problem 2. A layout for a housing colony in a city is to be prepared. Various locations of the colony are to be linked by roads. Given the various possibilities of linking the locations and their costs, what is the minimum-cost layout so that any two locations are connected by a chain of roads?

Problem 3. A layout for the electrical wiring of a building is to be prepared. Given the costs of the various possibilities, what is the minimum-cost layout?

# Recall: (Instances of) The Connector Problem

*from the Balakrishnan-Ranganathan book*

Problem 1. Various cities in a country are to be linked by means of roads. Given the various possibilities of connecting the cities and the costs involved, what is the most economical way of laying roads so that in the resulting road network any two cities are connected by a chain of roads?

Similar problems involve designing railroad networks, or water-line transports.

Problem 2. A layout for a housing colony in a city is to be prepared. Various locations of the colony are to be linked by roads. Given the various possibilities of linking the locations and their costs, what is the minimum-cost layout so that any two locations are connected by a chain of roads?

Problem 3. A layout for the electrical wiring of a building is to be prepared. Given the costs of the various possibilities, what is the minimum-cost layout?

**Solution here: find, in each case,**
**a minimum weight spanning tree.**

# Recall: Kruskal's algorithm

*One algorithm for finding minimum weight spanning trees of a given weighted graph is:*

## Kruskal's algorithm

Suppose you are given a weighted connected graph $G$, and you are looking for a minimum weight spanning tree of it.

# Recall: Kruskal's algorithm

*One algorithm for finding minimum weight spanning trees of a given weighted graph is:*

## Kruskal's algorithm

Suppose you are given a weighted connected graph $G$, and you are looking for a minimum weight spanning tree of it. Then:

1. Begin by finding an edge of minimum weight, and mark it.

# Recall: Kruskal's algorithm

*One algorithm for finding minimum weight spanning trees of a given weighted graph is:*

## Kruskal's algorithm

Suppose you are given a weighted connected graph $G$, and you are looking for a minimum weight spanning tree of it. Then:

1. Begin by finding an edge of minimum weight, and mark it.
2. Out of all the edges that remain unmarked **and which do not form a cycle with any of the already marked edges**, pick an edge of minimum weight and mark it.

# Recall: Kruskal's algorithm

*One algorithm for finding minimum weight spanning trees of a given weighted graph is:*

## Kruskal's algorithm

Suppose you are given a weighted connected graph $G$, and you are looking for a minimum weight spanning tree of it. Then:

1. Begin by finding an edge of minimum weight, and mark it.
2. Out of all the edges that remain unmarked **and which do not form a cycle with any of the already marked edges**, pick an edge of minimum weight and mark it.
3. If the set of the already marked edges gives a spanning tree of $G$, then terminate the process. Otherwise, return to Step 2.

# Recall: Kruskal's algorithm

*One algorithm for finding minimum weight spanning trees of a given weighted graph is:*

## Kruskal's algorithm

Suppose you are given a weighted connected graph $G$, and you are looking for a minimum weight spanning tree of it. Then:

1. Begin by finding an edge of minimum weight, and mark it.
2. Out of all the edges that remain unmarked **and which do not form a cycle with any of the already marked edges**, pick an edge of minimum weight and mark it.
3. If the set of the already marked edges gives a spanning tree of $G$, then terminate the process. Otherwise, return to Step 2.

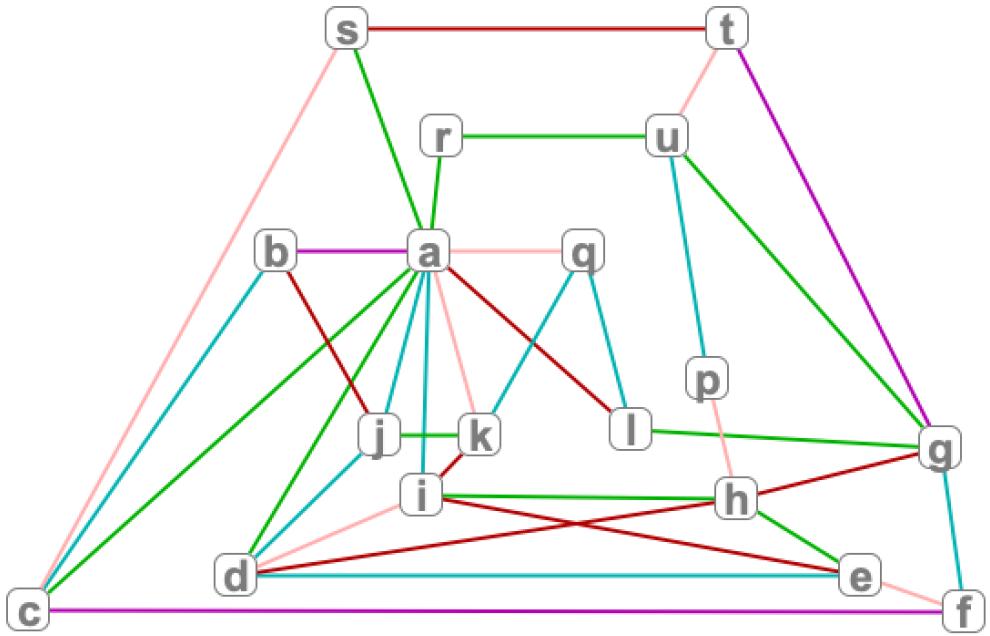
## Theorem 2 of Lecture 13

For every weighted connected graph $G$, Kruskal's algorithm gives a minimum weight spanning tree.

# Past Exam Problem

Solve the connector problem for the given graph, that is, find a minimum weight spanning tree.

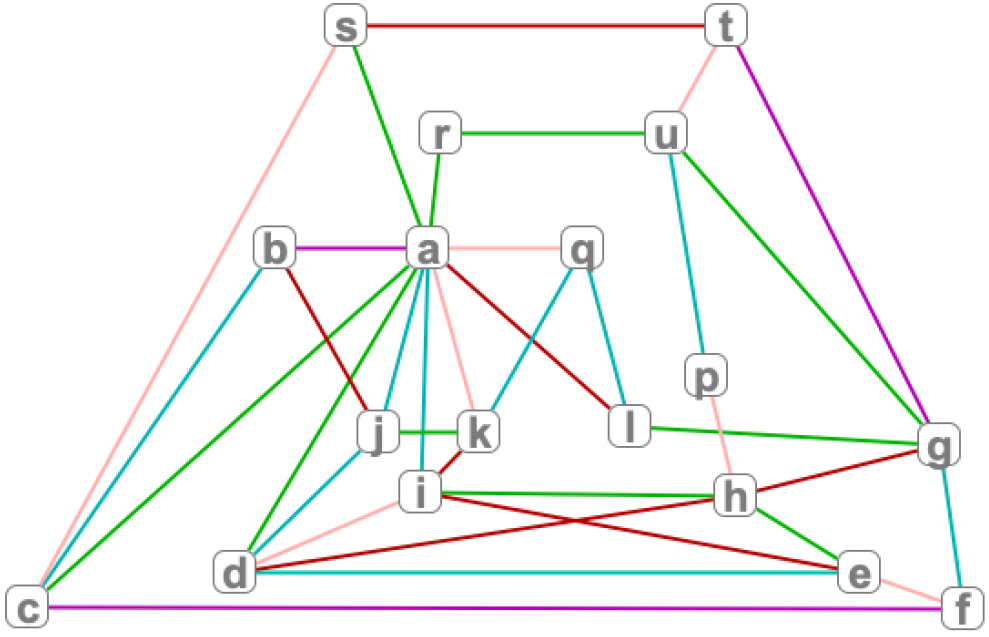
Also, find the total weight of such a tree. Show all your work.




Here we assume that

- edges $ak$, $aq$, $cs$, $di$, $ef$, $hp$ and $tu$ have weight 2;
- edges $ai$, $aj$, $bc$, $de$, $dj$, $fg$, $kq$, $lq$ and $pu$ have weight 3;
- edges $ac$, $ad$, $ar$, $as$, $eh$, $gl$, $gu$, $hi$, $jk$ and $ru$ have weight 4;
- edges $al$, $bj$, $dh$, $ei$, $gh$, $ik$ and $st$ have weight 5;
- and edges $ab$, $cf$ and $gt$ have weight 6.

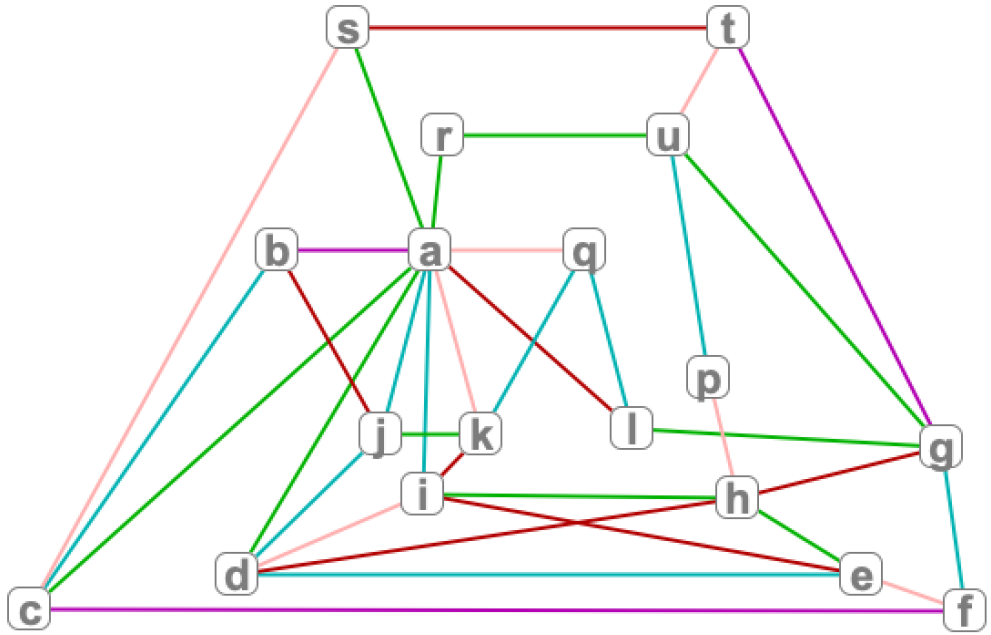
# One suggested solution




In the suggested solution given, the spanning tree had

seven edges with weight 2, seven edges with weight 3,
and three edges with weight 4.

# One suggested solution




In the suggested solution given, the spanning tree had

seven edges with weight 2, seven edges with weight 3,
and three edges with weight 4.

In fact, we can now see that any other minimum weight spanning tree of the given graph will have seven edges with weight 2, seven edges with weight 3, and three edges with weight 4, even if clearly some of these edges will have to be different between different such trees.

# Proof (of strong version) of Theorem 2

**Theorem 2 of Lecture 13 (in more detailed form)**

For every weighted connected graph $G$ on $n$ vertices, Kruskal's algorithm gives a minimum weight spanning tree *(in fact, it can give all minimum weight spanning trees of $G$).*

# Proof (of strong version) of Theorem 2

## Theorem 2 of Lecture 13 (in more detailed form)

For every weighted connected graph $G$ on $n$ vertices, Kruskal's algorithm gives a minimum weight spanning tree *(in fact, it can give all minimum weight spanning trees of $G$).*

Moreover, if $T_0$ is a minimum weight spanning tree of $G$, and

$$w(e_1) \leqslant w(e_2) \leqslant \cdots \leqslant w(e_{n-1})$$

are the weights of its edges, arranged in increasing order, then any other minimum weight spanning tree of $G$ (which we can find via Kruskal's algorithm) will give rise to the same sequence of weights.

# Proof (of strong version) of Theorem 2

Let $T_1, T_2, \ldots, T_s$ be all the spanning trees of $G$ that different applications of Kruskal's algorithm give us. Consider one of these, say tree $T_{i_0}$.

# Proof (of strong version) of Theorem 2

Let $T_1, T_2, \ldots, T_s$ be all the spanning trees of $G$ that different applications of Kruskal's algorithm give us. Consider one of these, say tree $T_{i_0}$. Write $w_1 < w_2 < \cdots < w_r$ for the different weights the edges of $T_{i_0}$ can have, and

$$w(e_{1,1})\,(= w_1) = w(e_{1,2}) = \cdots = w(e_{1,j_1}) < w(e_{2,1})\,(= w_2) = w(e_{2,2}) = \cdots = w(e_{2,j_2})$$
$$< \cdots\cdots < w(e_{r,1})\,(= w_r) = w(e_{r,2}) = \cdots = w(e_{r,j_r})$$

for its full sequence of weights of its edges (note that here we work under the most general assumption, that some of the edges of $T_{i_0}$ may have equal weights).

# Proof (of strong version) of Theorem 2

Let $T_1, T_2, \ldots, T_s$ be all the spanning trees of $G$ that different applications of Kruskal's algorithm give us. Consider one of these, say tree $T_{i_0}$. Write $w_1 < w_2 < \cdots < w_r$ for the different weights the edges of $T_{i_0}$ can have, and

$$w(e_{1,1})\,(= w_1) = w(e_{1,2}) = \cdots = w(e_{1,j_1}) < w(e_{2,1})\,(= w_2) = w(e_{2,2}) = \cdots = w(e_{2,j_2})$$
$$< \cdots\cdots < w(e_{r,1})\,(= w_r) = w(e_{r,2}) = \cdots = w(e_{r,j_r})$$

for its full sequence of weights of its edges (note that here we work under the most general assumption, that some of the edges of $T_{i_0}$ may have equal weights).

Assume towards a contradiction

- that either $T_{i_0}$ is not a minimum weight spanning tree of $G$,
- or that there exist minimum weight spanning trees $K$ of $G$ which are NOT among the trees $T_1, T_2, \ldots, T_s$ that Kruskal's algorithm gives us,
- or finally that there exist minimum weight spanning trees $K'$ of $G$ which have a different (increasing) sequence of weights of edges from $T_{i_0}$.

# Proof (of strong version) of Theorem 2

Let $T_1, T_2, \ldots, T_s$ be all the spanning trees of $G$ that different applications of Kruskal's algorithm give us. Consider one of these, say tree $T_{i_0}$. Write $w_1 < w_2 < \cdots < w_r$ for the different weights the edges of $T_{i_0}$ can have, and

$$w(e_{1,1})\,(= w_1) = w(e_{1,2}) = \cdots = w(e_{1,j_1}) < w(e_{2,1})\,(= w_2) = w(e_{2,2}) = \cdots = w(e_{2,j_2})$$
$$< \cdots\cdots < w(e_{r,1})\,(= w_r) = w(e_{r,2}) = \cdots = w(e_{r,j_r})$$

for its full sequence of weights of its edges (note that here we work under the most general assumption, that some of the edges of $T_{i_0}$ may have equal weights).

Assume towards a contradiction

- that either $T_{i_0}$ is not a minimum weight spanning tree of $G$,
- or that there exist minimum weight spanning trees $K$ of $G$ which are NOT among the trees $T_1, T_2, \ldots, T_s$ that Kruskal's algorithm gives us,
- or finally that there exist minimum weight spanning trees $K'$ of $G$ which have a different (increasing) sequence of weights of edges from $T_{i_0}$.

In all three cases, we can find (some 'special') minimum weight spanning trees of $G$ whose sequence of edges (arranged so that the corresponding weights are in increasing order) is different from that of $T_{i_0}$ (and in the 1st and 3rd cases, even the sequence of weights must necessarily be different; in the 1st case, this is because we assume that the total weight of $T_{i_0}$ is NOT minimum possible).

# Proof (of strong version) of Theorem 2

Let $T_1, T_2, \ldots, T_s$ be all the spanning trees of $G$ that different applications of Kruskal's algorithm give us. Consider one of these, say tree $T_{i_0}$. Write $w_1 < w_2 < \cdots < w_r$ for the different weights the edges of $T_{i_0}$ can have, and

$$w(e_{1,1})\,(= w_1) = w(e_{1,2}) = \cdots = w(e_{1,j_1}) < w(e_{2,1})\,(= w_2) = w(e_{2,2}) = \cdots = w(e_{2,j_2}) < \cdots\cdots < w(e_{r,1})\,(= w_r) = w(e_{r,2}) = \cdots = w(e_{r,j_r})$$

for its full sequence of weights of its edges (note that here we work under the most general assumption, that some of the edges of $T_{i_0}$ may have equal weights).

Assume towards a contradiction

- that either $T_{i_0}$ is not a minimum weight spanning tree of $G$,
- or that there exist minimum weight spanning trees $K$ of $G$ which are NOT among the trees $T_1, T_2, \ldots, T_s$ that Kruskal's algorithm gives us,
- or finally that there exist minimum weight spanning trees $K'$ of $G$ which have a different (increasing) sequence of weights of edges from $T_{i_0}$.

In all three cases, we can find (some 'special') minimum weight spanning trees of $G$ whose sequence of edges (arranged so that the corresponding weights are in increasing order) is different from that of $T_{i_0}$ (and in the 1st and 3rd cases, even the sequence of weights must necessarily be different; in the 1st case, this is because we assume that the total weight of $T_{i_0}$ is NOT minimum possible). Out of those 'special' trees, pick a tree $K_0$ whose sequence of edges coincides initially with the sequence of edges of $T_{i_0}$ **for the longest time possible,** that is,

- the first $k$ edges $\ell_1, \ell_2, \ldots, \ell_k$ of $K_0$ are the same as the first $k$ edges $e_1, e_2, \ldots, e_k$ of $T_{i_0}$, while $\ell_{k+1}$ is different from $e_{k+1}$,
- and for any other of the 'special' trees, at the very least the $(k+1)$-th edge also differs from edge $e_{k+1}$ of $T_{i_0}$.

# Proof (of strong version) of Theorem 2

**Given that $K_0$ and $T_{i_0}$ are different, we must have $k < n - 1$.**

# Proof (of strong version) of Theorem 2

**Given that $K_0$ and $T_{i_0}$ are different, we must have $k < n - 1$.**

Note also the following: given that we can reorder edges which have equal weights in the sequence for $K_0$, assume that we have optimised as much as possible here, and hence that

- either $w(\ell_{k+1}) < w(e_{k+1})$,
- or, in the case that $w(\ell_{k+1}) \geqslant w(e_{k+1})$, edge $e_{k+1}$ does NOT appear among the edges of $K_0$ somewhere later in the sequence (because in such a case we could just swap the positions of edges $\ell_{k+1}$ and $e_{k+1}$ without altering the sequence of weights, but then the sequences for $K_0$ and $T_{i_0}$ would coincide for even longer).

# Proof (of strong version) of Theorem 2

**Given that $K_0$ and $T_{i_0}$ are different, we must have $k < n-1$.**

Note also the following: given that we can reorder edges which have equal weights in the sequence for $K_0$, assume that we have optimised as much as possible here, and hence that

- either $w(\ell_{k+1}) < w(e_{k+1})$,
- or, in the case that $w(\ell_{k+1}) \geqslant w(e_{k+1})$, edge $e_{k+1}$ does NOT appear among the edges of $K_0$ somewhere later in the sequence (because in such a case we could just swap the positions of edges $\ell_{k+1}$ and $e_{k+1}$ without altering the sequence of weights, but then the sequences for $K_0$ and $T_{i_0}$ would coincide for even longer).

We now consider the 3 possibilities we have:

# Proof (of strong version) of Theorem 2

**Given that $K_0$ and $T_{i_0}$ are different, we must have $k < n-1$.**

Note also the following: given that we can reorder edges which have equal weights in the sequence for $K_0$, assume that we have optimised as much as possible here, and hence that

- either $w(\ell_{k+1}) < w(e_{k+1})$,
- or, in the case that $w(\ell_{k+1}) \geqslant w(e_{k+1})$, edge $e_{k+1}$ does NOT appear among the edges of $K_0$ somewhere later in the sequence (because in such a case we could just swap the positions of edges $\ell_{k+1}$ and $e_{k+1}$ without altering the sequence of weights, but then the sequences for $K_0$ and $T_{i_0}$ would coincide for even longer).

We now consider the 3 possibilities we have:

Case 1: $w(\ell_{k+1}) < w(e_{k+1})$.

# Proof (of strong version) of Theorem 2

**Given that $K_0$ and $T_{i_0}$ are different, we must have $k < n-1$.**

Note also the following: given that we can reorder edges which have equal weights in the sequence for $K_0$, assume that we have optimised as much as possible here, and hence that

- either $w(\ell_{k+1}) < w(e_{k+1})$,
- or, in the case that $w(\ell_{k+1}) \geqslant w(e_{k+1})$, edge $e_{k+1}$ does NOT appear among the edges of $K_0$ somewhere later in the sequence (because in such a case we could just swap the positions of edges $\ell_{k+1}$ and $e_{k+1}$ without altering the sequence of weights, but then the sequences for $K_0$ and $T_{i_0}$ would coincide for even longer).

We now consider the 3 possibilities we have:

Case 1: $w(\ell_{k+1}) < w(e_{k+1})$. Then edge $\ell_{k+1}$ would be among the edges of $G$ which do not form a cycle with any of the already selected (for $T_{i_0}$) edges

$$\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_k = e_k,$$

and it would also have weight strictly smaller than edge $e_{k+1}$

# Proof (of strong version) of Theorem 2

**Given that $K_0$ and $T_{i_0}$ are different, we must have $k < n-1$.**

Note also the following: given that we can reorder edges which have equal weights in the sequence for $K_0$, assume that we have optimised as much as possible here, and hence that

- either $w(\ell_{k+1}) < w(e_{k+1})$,
- or, in the case that $w(\ell_{k+1}) \geqslant w(e_{k+1})$, edge $e_{k+1}$ does NOT appear among the edges of $K_0$ somewhere later in the sequence (because in such a case we could just swap the positions of edges $\ell_{k+1}$ and $e_{k+1}$ without altering the sequence of weights, but then the sequences for $K_0$ and $T_{i_0}$ would coincide for even longer).

We now consider the 3 possibilities we have:

Case 1: $w(\ell_{k+1}) < w(e_{k+1})$. Then edge $\ell_{k+1}$ would be among the edges of $G$ which do not form a cycle with any of the already selected (for $T_{i_0}$) edges

$$\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_k = e_k,$$

and it would also have weight strictly smaller than edge $e_{k+1}$ $\rightsquigarrow$ Kruskal's algorithm would never select edge $e_{k+1}$ at this stage (nor any of the edges of $T_{i_0}$ coming after $e_{k+1}$ since these would also have larger weights), so in other words Kruskal's algorithm would never produce tree $T_{i_0}$ (contradicting our assumptions).

# Proof (of strong version) of Theorem 2

Case 2: $w(\ell_{k+1}) > w(e_{k+1})$.

# Proof (of strong version) of Theorem 2

Case 2: $w(\ell_{k+1}) > w(e_{k+1})$. Then definitely $e_{k+1}$ cannot be among the edges of $K_0$ (because it's not among the first $k$ edges, and it cannot come after edge $\ell_{k+1}$ given that we have ordered the sequence so that the weights are in increasing order).

# Proof (of strong version) of Theorem 2

Case 2: $w(\ell_{k+1}) > w(e_{k+1})$. Then definitely $e_{k+1}$ cannot be among the edges of $K_0$ (because it's not among the first $k$ edges, and it cannot come after edge $\ell_{k+1}$ given that we have ordered the sequence so that the weights are in increasing order).

But then, given that $K_0$ is a tree NOT containing edge $e_{k+1}$, by adding it we create precisely one cycle, say cycle $C_0$.

# Proof (of strong version) of Theorem 2

Case 2: $w(\ell_{k+1}) > w(e_{k+1})$. Then definitely $e_{k+1}$ cannot be among the edges of $K_0$ (because it's not among the first $k$ edges, and it cannot come after edge $\ell_{k+1}$ given that we have ordered the sequence so that the weights are in increasing order).

But then, given that $K_0$ is a tree NOT containing edge $e_{k+1}$, by adding it we create precisely one cycle, say cycle $C_0$. Moreover, this cycle must contain some edge different from the edges

$$\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_k = e_k$$

(because $e_{k+1}$ is an edge of $T_{i_0}$, as are the edges $e_1, e_2, \ldots, e_k$, so $e_{k+1}$ does NOT form a cycle with just those edges).

# Proof (of strong version) of Theorem 2

Case 2: $w(\ell_{k+1}) > w(e_{k+1})$. Then definitely $e_{k+1}$ cannot be among the edges of $K_0$ (because it's not among the first $k$ edges, and it cannot come after edge $\ell_{k+1}$ given that we have ordered the sequence so that the weights are in increasing order).

But then, given that $K_0$ is a tree NOT containing edge $e_{k+1}$, by adding it we create precisely one cycle, say cycle $C_0$. Moreover, this cycle must contain some edge different from the edges

$$\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_k = e_k$$

(because $e_{k+1}$ is an edge of $T_{i_0}$, as are the edges $e_1, e_2, \ldots, e_k$, so $e_{k+1}$ does NOT form a cycle with just those edges). Thus, say that $C_0$ contains the edge $\ell_t$ with $t \geqslant k+1$; this also implies that

$$w(\ell_t) \geqslant w(\ell_{k+1}) > w(e_{k+1}).$$

# Proof (of strong version) of Theorem 2

Case 2: $w(\ell_{k+1}) > w(e_{k+1})$. Then definitely $e_{k+1}$ cannot be among the edges of $K_0$ (because it's not among the first $k$ edges, and it cannot come after edge $\ell_{k+1}$ given that we have ordered the sequence so that the weights are in increasing order).

But then, given that $K_0$ is a tree NOT containing edge $e_{k+1}$, by adding it we create precisely one cycle, say cycle $C_0$. Moreover, this cycle must contain some edge different from the edges

$$\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_k = e_k$$

(because $e_{k+1}$ is an edge of $T_{i_0}$, as are the edges $e_1, e_2, \ldots, e_k$, so $e_{k+1}$ does NOT form a cycle with just those edges). Thus, say that $C_0$ contains the edge $\ell_t$ with $t \geqslant k+1$; this also implies that

$$w(\ell_t) \geqslant w(\ell_{k+1}) > w(e_{k+1}).$$

$\rightsquigarrow$ $K_0 + e_{k+1} - \ell_t$ is a spanning tree of $G$ again, and at the same time it has smaller total weight than $K_0$ (contradicting our assumption that $K_0$ is a minimum weight spanning tree of $G$).

# Proof (of strong version) of Theorem 2

Case 2: $w(\ell_{k+1}) > w(e_{k+1})$. Then definitely $e_{k+1}$ cannot be among the edges of $K_0$ (because it's not among the first $k$ edges, and it cannot come after edge $\ell_{k+1}$ given that we have ordered the sequence so that the weights are in increasing order).

But then, given that $K_0$ is a tree NOT containing edge $e_{k+1}$, by adding it we create precisely one cycle, say cycle $C_0$. Moreover, this cycle must contain some edge different from the edges

$$\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_k = e_k$$

(because $e_{k+1}$ is an edge of $T_{i_0}$, as are the edges $e_1, e_2, \ldots, e_k$, so $e_{k+1}$ does NOT form a cycle with just those edges). Thus, say that $C_0$ contains the edge $\ell_t$ with $t \geqslant k+1$; this also implies that

$$w(\ell_t) \geqslant w(\ell_{k+1}) > w(e_{k+1}).$$

$\rightsquigarrow$ $K_0 + e_{k+1} - \ell_t$ is a spanning tree of $G$ again, and at the same time it has smaller total weight than $K_0$ (contradicting our assumption that $K_0$ is a minimum weight spanning tree of $G$).

Case 3: $w(\ell_{k+1}) = w(e_{k+1})$.

# Proof (of strong version) of Theorem 2

Case 2: $w(\ell_{k+1}) > w(e_{k+1})$. Then definitely $e_{k+1}$ cannot be among the edges of $K_0$ (because it's not among the first $k$ edges, and it cannot come after edge $\ell_{k+1}$ given that we have ordered the sequence so that the weights are in increasing order).

But then, given that $K_0$ is a tree NOT containing edge $e_{k+1}$, by adding it we create precisely one cycle, say cycle $C_0$. Moreover, this cycle must contain some edge different from the edges

$$\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_k = e_k$$

(because $e_{k+1}$ is an edge of $T_{i_0}$, as are the edges $e_1, e_2, \ldots, e_k$, so $e_{k+1}$ does NOT form a cycle with just those edges). Thus, say that $C_0$ contains the edge $\ell_t$ with $t \geqslant k+1$; this also implies that

$$w(\ell_t) \geqslant w(\ell_{k+1}) > w(e_{k+1}).$$

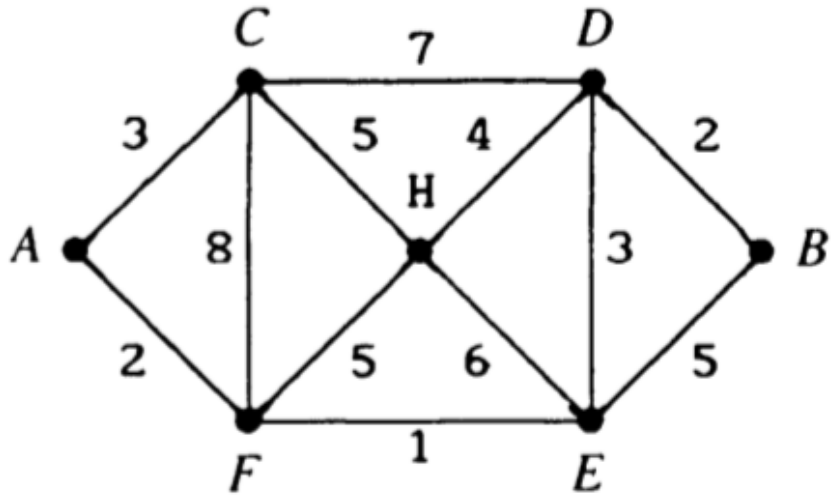
$\rightsquigarrow$ $K_0 + e_{k+1} - \ell_t$ is a spanning tree of $G$ again, and at the same time it has smaller total weight than $K_0$ (contradicting our assumption that $K_0$ is a minimum weight spanning tree of $G$).

Case 3: $w(\ell_{k+1}) = w(e_{k+1})$. We can treat it (almost) analogously to Case 2; how?

Other similar problems which have motivated interesting questions in Graph Theory (which in turn has led to further understanding of the applications too)

# The shortest path problem

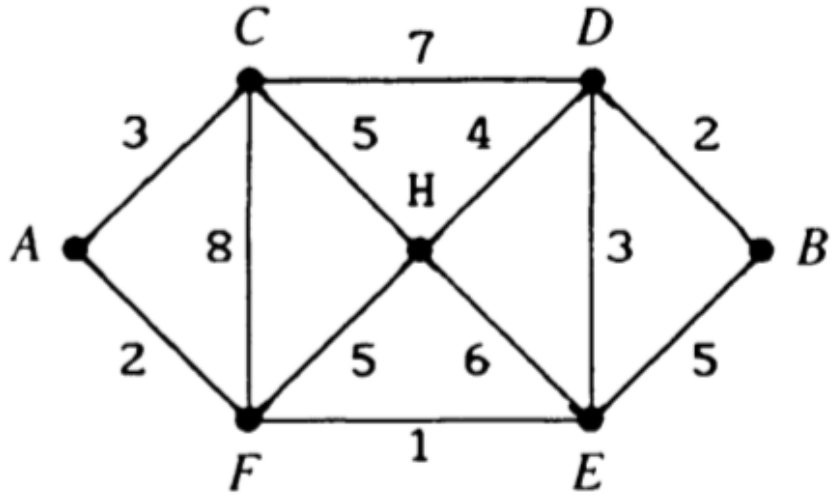
Let $G_0$ be a weighted connected graph, and let $A, B$ be two different vertices of $G_0$.




from the Balakrishnan-Ranganathan book

# The shortest path problem

Let $G_0$ be a weighted connected graph, and let $A, B$ be two different vertices of $G_0$.




from the Balakrishnan-Ranganathan book

Since $G_0$ is connected, we know that there are paths in $G_0$ taking us from $A$ to $B$.

Question. How do we find such a path which also has minimum weight?

# The travelling salesman problem

Let $G_0$ be a weighted connected graph whose vertices represent different cities that a salesman wants to visit, which are connected by, say, roads and highways, or by train routes, or by airline routes, represented by the edges of the graph *(with each edge weight capturing the cost or distance of travel from one city - endvertex to the other city - endvertex which this edge joins).*

# The travelling salesman problem

Let $G_0$ be a weighted connected graph whose vertices represent different cities that a salesman wants to visit, which are connected by, say, roads and highways, or by train routes, or by airline routes, represented by the edges of the graph *(with each edge weight capturing the cost or distance of travel from one city - endvertex to the other city - endvertex which this edge joins).*

Question 1. What is the most cost-efficient (or time-efficient) way for the salesman to visit all the cities and finally return to the city which he is supposed to start from?

# The travelling salesman problem

Let $G_0$ be a weighted connected graph whose vertices represent different cities that a salesman wants to visit, which are connected by, say, roads and highways, or by train routes, or by airline routes, represented by the edges of the graph *(with each edge weight capturing the cost or distance of travel from one city - endvertex to the other city - endvertex which this edge joins).*

Question 1. What is the most cost-efficient (or time-efficient) way for the salesman to visit all the cities and finally return to the city which he is supposed to start from?

Question 2. Is there a way for the salesman to visit all the cities but not pass by any city more than once (except perhaps to return to the city where he starts from at the end of his trip)?

# An efficient scenic route...

The city of Königsberg, Prussia was set on the Pregel River, and included two large islands that were connected to each other and the mainland by seven bridges.

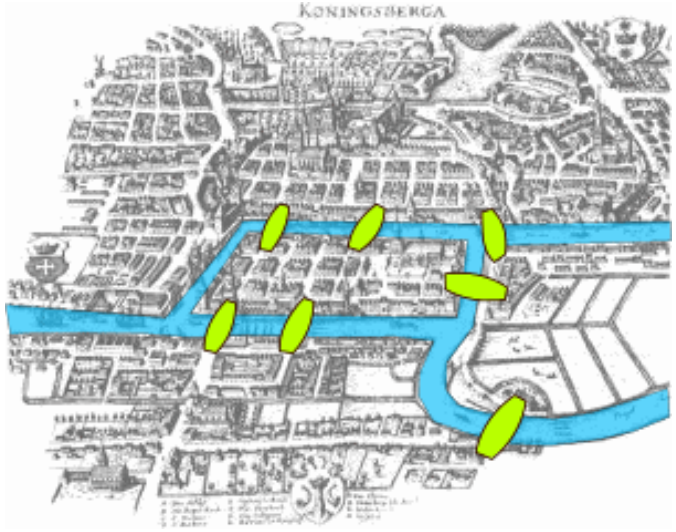



**Image from Wikipedia:** Map of the city in Leonhard Euler's time showing the actual layout of the seven bridges, and highlighting the river Pregel and the bridges.

# An efficient scenic route...

The city of Königsberg, Prussia was set on the Pregel River, and included two large islands that were connected to each other and the mainland by seven bridges.

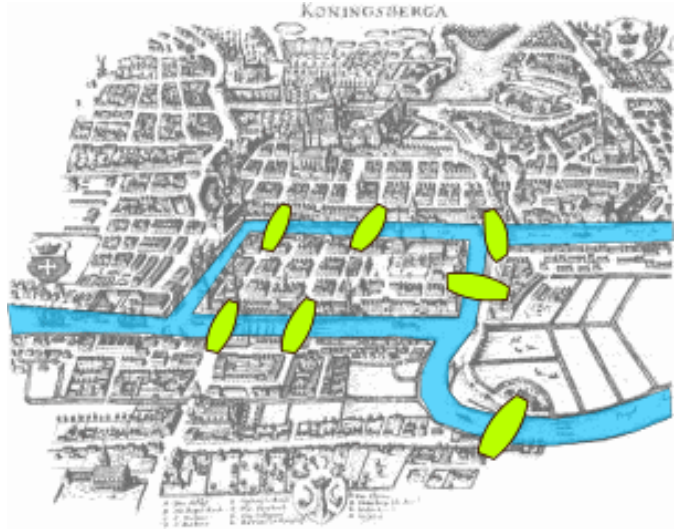



**Image from Wikipedia:** Map of the city in Leonhard Euler's time showing the actual layout of the seven bridges, and highlighting the river Pregel and the bridges.

People spent time trying to discover a way in which they could cross each bridge exactly once before returning to the point / place in the city that they started from.

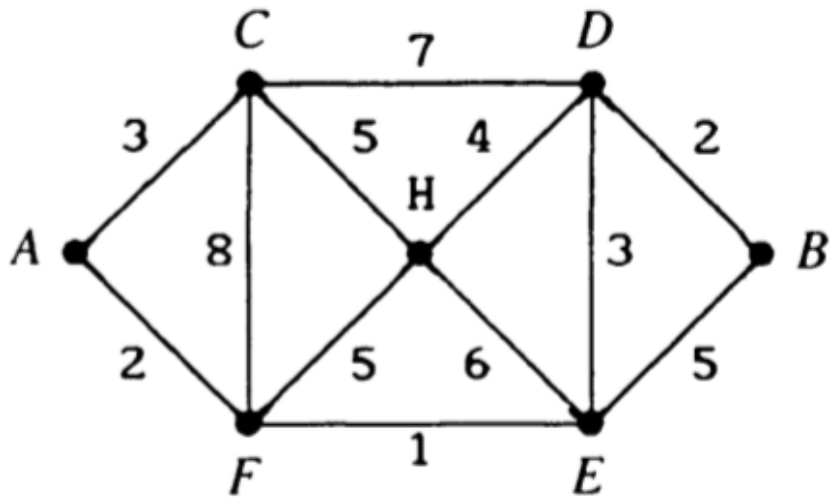
# An efficient scenic route...

This problem is now known as the *Königsberg Bridges Problem*, or equivalently the *Euler-Königsberg Bridges Problem*, since it was Leonhard Euler who determined in 1735 that finding such a way is not possible.

This result of his is regarded as essentially giving rise to Graph Theory as an area of Mathematics.

# We start with:
# The Shortest Path Problem

Recall that we have a weighted connected graph $G_0$ of order $n$ (here $n = 7$), and want to find a minimum weight path connecting, say, vertices $A$ and $B$ below.



# Weight matrix of a weighted graph

In the process of solving the shortest path problem, we need to consider the *weight matrix* $W_{G_0}$ of $G_0$.

Assume that the vertex set of $G_0$ is $\{v_1, v_2, \ldots, v_n\}$; then $W_{G_0} = \left(w_{i,j}\right)_{i,j}$ is an $n \times n$ matrix satisfying

$$w_{i,j} = \begin{cases} \infty & \text{if } i = j, \text{ or the vertices } v_i \text{ and } v_j \text{ are non-adjacent} \\ \text{weight of the edge } \{v_i,\ v_j\} & \text{if } v_i \text{ and } v_j \text{ are adjacent vertices} \end{cases}$$

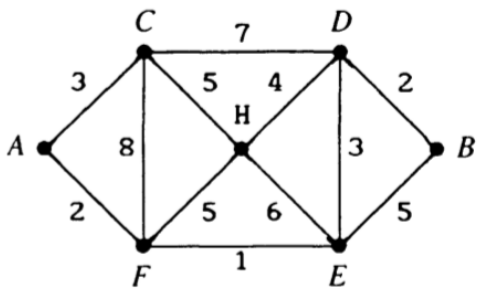
# Weight matrix of a weighted graph

In the process of solving the shortest path problem, we need to consider the *weight matrix* $W_{G_0}$ of $G_0$.

Assume that the vertex set of $G_0$ is $\{v_1, v_2, \ldots, v_n\}$; then $W_{G_0} = \left(w_{i,j}\right)_{i,j}$ is an $n \times n$ matrix satisfying

$$w_{i,j} = \begin{cases} \infty & \text{if } i = j, \text{ or the vertices } v_i \text{ and } v_j \text{ are non-adjacent} \\ \text{weight of the edge } \{v_i,\ v_j\} & \text{if } v_i \text{ and } v_j \text{ are adjacent vertices} \end{cases}$$

For the example above, we have



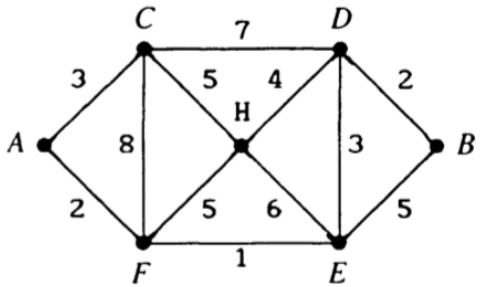
# Weight matrix of a weighted graph

In the process of solving the shortest path problem, we need to consider the *weight matrix* $W_{G_0}$ of $G_0$.

Assume that the vertex set of $G_0$ is $\{v_1, v_2, \ldots, v_n\}$; then $W_{G_0} = \left(w_{i,j}\right)_{i,j}$ is an $n \times n$ matrix satisfying

$$w_{i,j} = \begin{cases} \infty & \text{if } i = j, \text{ or the vertices } v_i \text{ and } v_j \text{ are non-adjacent} \\ \text{weight of the edge } \{v_i,\ v_j\} & \text{if } v_i \text{ and } v_j \text{ are adjacent vertices} \end{cases}$$

For the example above, we have




$$W_{G_0} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ H \end{array} \begin{array}{c} \begin{array}{ccccccc} A & B & C & D & E & F & H \end{array} \\ \begin{pmatrix} \infty & \infty & 3 & \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 2 & 5 & \infty & \infty \\ 3 & \infty & \infty & 7 & \infty & 8 & 5 \\ \infty & 2 & 7 & \infty & 3 & \infty & 4 \\ \infty & 5 & \infty & 3 & \infty & 1 & 6 \\ 2 & \infty & 8 & \infty & 1 & \infty & 5 \\ \infty & \infty & 5 & 4 & 6 & 5 & \infty \end{pmatrix} \end{array}.$$

*An algorithm that returns the 'shortest possible distance' a path connecting any two fixed vertices (say, vertices A and B here) covers:*

## Dijkstra's algorithm

— The algorithm proceeds as follows: at each stage it assigns weights to the vertices (which are based in some sense on the weights of the edges).

— Some of these weights are called temporary *(in a sense, we're still testing out what the value for those vertices should be)*, while the rest have become permanent at some point *(and the algorithm cannot alter the latter anymore)*.

*An algorithm that returns the 'shortest possible distance' a path connecting any two fixed vertices (say, vertices A and B here) covers:*

## Dijkstra's algorithm

— The algorithm proceeds as follows: at each stage it assigns weights to the vertices (which are based in some sense on the weights of the edges).

— Some of these weights are called temporary *(in a sense, we're still testing out what the value for those vertices should be)*, while the rest have become permanent at some point *(and the algorithm cannot alter the latter anymore)*.

— **At each stage a new vertex is allotted a permanent weight,** with vertex $A$ being the first one to be allotted a permanent weight.

*An algorithm that returns the 'shortest possible distance' a path connecting any two fixed vertices (say, vertices A and B here) covers:*

## Dijkstra's algorithm

— The algorithm proceeds as follows: at each stage it assigns weights to the vertices (which are based in some sense on the weights of the edges).

— Some of these weights are called temporary *(in a sense, we're still testing out what the value for those vertices should be)*, while the rest have become permanent at some point *(and the algorithm cannot alter the latter anymore)*.

— **At each stage a new vertex is allotted a permanent weight,** with vertex $A$ being the first one to be allotted a permanent weight.

— The algorithm can be terminated as soon as vertex $B$ (the vertex that we want our path to end at) is allotted a permanent weight.

**Also, the permanent weight allotted to $B$ is precisely the shortest distance covered by a path from $A$ to $B$.** *(more about this shortly)*

## Dijkstra's algorithm

In more detail now,

- at the first stage, we assign permanent weight 0 to vertex $A$ and temporary weight $\infty$ to every other vertex of $G_0$.

## Dijkstra's algorithm

In more detail now,

- at the first stage, we assign permanent weight 0 to vertex $A$ and temporary weight $\infty$ to every other vertex of $G_0$.
- At Stage $r$, let $v_{j_1}, v_{j_2}, \ldots, v_{j_{n-r+1}}$ be the vertices which still have temporary weight, while $v_{i_1}, v_{i_2}, \ldots, v_{i_{r-1}}$ are the vertices of $G_0$ which have already been allotted a permanent weight.

## Dijkstra's algorithm

In more detail now,

- at the first stage, we assign permanent weight 0 to vertex $A$ and temporary weight $\infty$ to every other vertex of $G_0$.
- At Stage $r$, let $v_{j_1}, v_{j_2}, \ldots, v_{j_{n-r+1}}$ be the vertices which still have temporary weight, while $v_{i_1}, v_{i_2}, \ldots, v_{i_{r-1}}$ are the vertices of $G_0$ which have already been allotted a permanent weight.
  - For each $j_s$, $1 \leqslant s \leqslant n-r+1$, and each $i_t$, $1 \leqslant t \leqslant r-1$, set

    $$a_{j_s,i_t} = \min\{\text{current weight of } v_{j_s},\ (\text{permanent weight of } v_{i_t}) + w_{j_s,i_t}\}.$$

    Furthermore, set $w_{j_s} = \min\{a_{j_s,i_t} : 1 \leqslant t \leqslant r-1\}$. This is the new temporary weight of the (arbitrary) vertex $v_{j_s} \in \{v_{j_1}, v_{j_2}, \ldots, v_{j_{n-r+1}}\}$.

## Dijkstra's algorithm

In more detail now,

- at the first stage, we assign permanent weight 0 to vertex $A$ and temporary weight $\infty$ to every other vertex of $G_0$.
- At Stage $r$, let $v_{j_1}, v_{j_2}, \ldots, v_{j_{n-r+1}}$ be the vertices which still have temporary weight, while $v_{i_1}, v_{i_2}, \ldots, v_{i_{r-1}}$ are the vertices of $G_0$ which have already been allotted a permanent weight.
  - For each $j_s$, $1 \leqslant s \leqslant n-r+1$, and each $i_t$, $1 \leqslant t \leqslant r-1$, set

    $$a_{j_s,i_t} = \min\{\text{current weight of } v_{j_s},\ (\text{permanent weight of } v_{i_t}) + w_{j_s,i_t}\}.$$

    Furthermore, set $w_{j_s} = \min\{a_{j_s,i_t} : 1 \leqslant t \leqslant r-1\}$. This is the new temporary weight of the (arbitrary) vertex $v_{j_s} \in \{v_{j_1}, v_{j_2}, \ldots, v_{j_{n-r+1}}\}$.
  - Find $s_0 \in \{1, 2, \ldots, n-r+1\}$ such that

    $$w_{j_{s_0}} = \min\{w_{j_s} : 1 \leqslant s \leqslant n-r+1\}$$

    (there might be two or more indices that work here, that is, the vertex that has minimum temporary weight at this point might not be unique; in such a case, just pick one index that works).

    Then vertex $v_{j_{s_0}}$ is the vertex that is allotted permanent weight at Stage $r$ (with this weight being $w_{j_{s_0}}$).

# Applying Dijkstra's algorithm to examples

*From the Balakrishnan-Ranganathan book:*

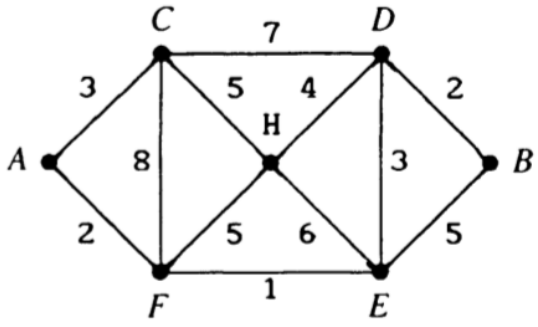
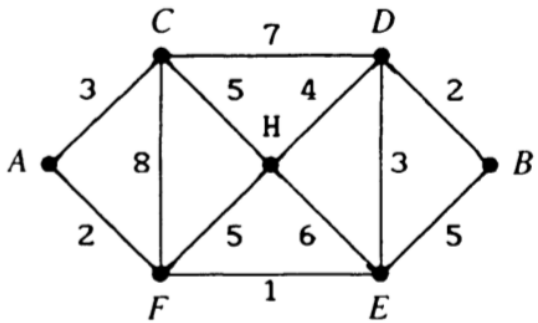



TABLE 10.5. *Steps of algorithm for shortest path from A to B*

| | *A* | *B* | *C* | *D* | *E* | *F* | *H* |
|---|---|---|---|---|---|---|---|
| Iteration 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Iteration 1 | 0 | ∞ | 3 | ∞ | ∞ | 2 | ∞ |
| Iteration 2 | 0 | ∞ | 3 | ∞ | 3 | 2 | 7 |
| Iteration 3 | 0 | ∞ | 3 | 10 | 3 | 2 | 7 |
| Iteration 4 | 0 | 8 | 3 | 6 | 3 | 2 | 7 |
| Iteration 5 | 0 | 8 | 3 | 6 | 3 | 2 | 7 |
| Iteration 6 | 0 | 8 | 3 | 6 | 3 | 2 | 7 |

# Applying Dijkstra's algorithm to examples

*From the Balakrishnan-Ranganathan book:*



# Understanding the output of the algorithm

- Suppose that $v_0$ is the vertex we start with, namely the vertex which is assigned permanent weight 0 (in the previous example, this is vertex $A$).

  For each vertex $v$ which is allotted a permanent weight before the algorithm is terminated, **this permanent weight is precisely the shortest distance that can be covered by a path from the initial vertex $v_0$ to $v$.**

# Understanding the output of the algorithm

- Suppose that $v_0$ is the vertex we start with, namely the vertex which is assigned permanent weight 0 (in the previous example, this is vertex $A$).

  For each vertex $v$ which is allotted a permanent weight before the algorithm is terminated, **this permanent weight is precisely the shortest distance that can be covered by a path from the initial vertex $v_0$ to $v$.**

- In the case of the previous example, in which we were 'unlucky' (or 'lucky' from another point of view) and ended up needing to find permanent weights for all the vertices of the graph, we now know the shortest distance between vertex $A$ and any other vertex of the graph! *For instance,*

  - *the shortest possible distance covered by a path from $A$ to $D$ is 6,*

# Understanding the output of the algorithm

- Suppose that $v_0$ is the vertex we start with, namely the vertex which is assigned permanent weight 0 (in the previous example, this is vertex $A$).

  For each vertex $v$ which is allotted a permanent weight before the algorithm is terminated, **this permanent weight is precisely the shortest distance that can be covered by a path from the initial vertex $v_0$ to $v$.**

- In the case of the previous example, in which we were 'unlucky' (or 'lucky' from another point of view) and ended up needing to find permanent weights for all the vertices of the graph, we now know the shortest distance between vertex $A$ and any other vertex of the graph! *For instance,*
  - *the shortest possible distance covered by a path from $A$ to $D$ is 6,*
  - *while the shortest possible distance covered by a path from $A$ to $H$ is 7.*

# Understanding the output of the algorithm (cont.)

- **Important Question.** Once we have found the shortest distance from vertex $A$ to, say, vertex $B$, how can we also find a path of shortest total distance from $A$ to $B$ (or in other words, a minimum weight path)?

# Understanding the output of the algorithm (cont.)

- **Important Question.** Once we have found the shortest distance from vertex $A$ to, say, vertex $B$, how can we also find a path of shortest total distance from $A$ to $B$ (or in other words, a minimum weight path)?

  — We start moving 'backwards': 'standing at' vertex $B$, we look at its neighbours, which are the only vertices that could come right after $B$ in the path *(or right before $B$, depending on which direction we traverse the path in)*.

# Understanding the output of the algorithm (cont.)

- **Important Question.** Once we have found the shortest distance from vertex $A$ to, say, vertex $B$, how can we also find a path of shortest total distance from $A$ to $B$ (or in other words, a minimum weight path)?

  — We start moving 'backwards': 'standing at' vertex $B$, we look at its neighbours, which are the only vertices that could come right after $B$ in the path *(or right before $B$, depending on which direction we traverse the path in)*.

**Crucial Observation here**

No matter what graph we are working with, at least some of the neighbours of vertex $B$ will have been allotted a permanent weight **before $B$ is allotted one**, and thus before the algorithm is terminated *(try to explain why this is so)*.

Let's say $N_1, N_2, \ldots, N_s$ are the neighbours of $B$ that have been allotted a permanent weight, and suppose that $w_B, w_{N_1}, w_{N_2}, \ldots, w_{N_s}$ are these permanent weights.

# Understanding the output of the algorithm (cont.)

- **Important Question.** Once we have found the shortest distance from vertex $A$ to, say, vertex $B$, how can we also find a path of shortest total distance from $A$ to $B$ (or in other words, a minimum weight path)?

  — We start moving 'backwards': 'standing at' vertex $B$, we look at its neighbours, which are the only vertices that could come right after $B$ in the path *(or right before $B$, depending on which direction we traverse the path in)*.

**Crucial Observation here**

No matter what graph we are working with, at least some of the neighbours of vertex $B$ will have been allotted a permanent weight **before $B$ is allotted one**, and thus before the algorithm is terminated *(try to explain why this is so)*.

Let's say $N_1, N_2, \ldots, N_s$ are the neighbours of $B$ that have been allotted a permanent weight, and suppose that $w_B, w_{N_1}, w_{N_2}, \ldots, w_{N_s}$ are these permanent weights.

Note that, for each $i \in \{1, 2, \ldots, s\}$, we will have

$$w_B - w_{N_i} \leqslant w_{B,N_i} = \text{weight of the edge } \{B, N_i\}.$$

# Understanding the output of the algorithm (cont.)

- **Important Question.** Once we have found the shortest distance from vertex $A$ to, say, vertex $B$, how can we also find a path of shortest total distance from $A$ to $B$ (or in other words, a minimum weight path)?

  — We start moving 'backwards': 'standing at' vertex $B$, we look at its neighbours, which are the only vertices that could come right after $B$ in the path *(or right before $B$, depending on which direction we traverse the path in)*.

**Crucial Observation here**

No matter what graph we are working with, at least some of the neighbours of vertex $B$ will have been allotted a permanent weight **before $B$ is allotted one**, and thus before the algorithm is terminated *(try to explain why this is so)*.

Let's say $N_1, N_2, \ldots, N_s$ are the neighbours of $B$ that have been allotted a permanent weight, and suppose that $w_B, w_{N_1}, w_{N_2}, \ldots, w_{N_s}$ are these permanent weights.

Note that, for each $i \in \{1, 2, \ldots, s\}$, we will have

$$w_B - w_{N_i} \leqslant w_{B,N_i} = \text{weight of the edge } \{B, N_i\}.$$

If for some $i_0$ we have $w_B - w_{N_{i_0}} = w_{B,N_{i_0}}$, then the path of shortest total distance that we are trying to find can be chosen to move from $B$ to $N_{i_0}$.

# Understanding the output of the algorithm (cont.)

- **Important Question.** Once we have found the shortest distance from vertex $A$ to, say, vertex $B$, how can we also find a path of shortest total distance from $A$ to $B$ (or in other words, a minimum weight path)?

# Understanding the output of the algorithm (cont.)

- **Important Question.** Once we have found the shortest distance from vertex $A$ to, say, vertex $B$, how can we also find a path of shortest total distance from $A$ to $B$ (or in other words, a minimum weight path)?

— 'Standing at' vertex $N_{i_0}$, we repeat the above process for $N_{i_0}$ now: we look at all neighbours of $N_{i_0}$, except for vertex $B$, which have been allotted a permanent weight; let's write $M_1, M_2, \ldots, M_t$ for these neighbours of $N_{i_0}$.

# Understanding the output of the algorithm (cont.)

- **Important Question.** Once we have found the shortest distance from vertex $A$ to, say, vertex $B$, how can we also find a path of shortest total distance from $A$ to $B$ (or in other words, a minimum weight path)?

— 'Standing at' vertex $N_{i_0}$, we repeat the above process for $N_{i_0}$ now: we look at all neighbours of $N_{i_0}$, except for vertex $B$, which have been allotted a permanent weight; let's write $M_1, M_2, \ldots, M_t$ for these neighbours of $N_{i_0}$.

For any $j_0 \in \{1, 2, \ldots, t\}$ for which we have

$$w_{N_{i_0}} - w_{M_{j_0}} = w_{N_{i_0}, M_{j_0}} = \text{weight of the edge } \{N_{i_0}, M_{j_0}\},$$

we can choose the path of shortest total distance
that we are trying to find to move next to vertex $M_{j_0}$.

# Understanding the output of the algorithm (cont.)

- **Important Question.** Once we have found the shortest distance from vertex $A$ to, say, vertex $B$, how can we also find a path of shortest total distance from $A$ to $B$ (or in other words, a minimum weight path)?

— 'Standing at' vertex $N_{i_0}$, we repeat the above process for $N_{i_0}$ now: we look at all neighbours of $N_{i_0}$, except for vertex $B$, which have been allotted a permanent weight; let's write $M_1, M_2, \ldots, M_t$ for these neighbours of $N_{i_0}$.

For any $j_0 \in \{1, 2, \ldots, t\}$ for which we have

$$w_{N_{i_0}} - w_{M_{j_0}} = w_{N_{i_0}, M_{j_0}} = \text{weight of the edge } \{N_{i_0}, M_{j_0}\},$$

we can choose the path of shortest total distance that we are trying to find to move next to vertex $M_{j_0}$.

— We continue like this until we reach vertex $A$, and thus find a full $A-B$ path of shortest total distance as we wanted *(we are guaranteed to reach vertex $A$ after finitely many steps, because there are only finitely many vertices with permanent weights to consider).*

# Finding minimum weight paths from $A$ to $B$

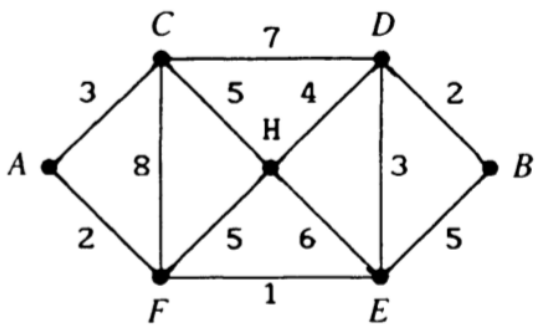
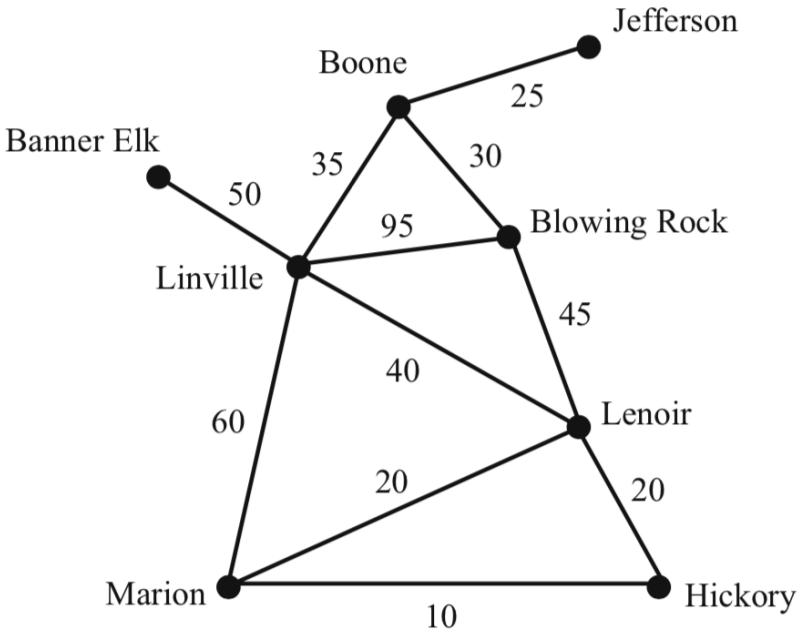


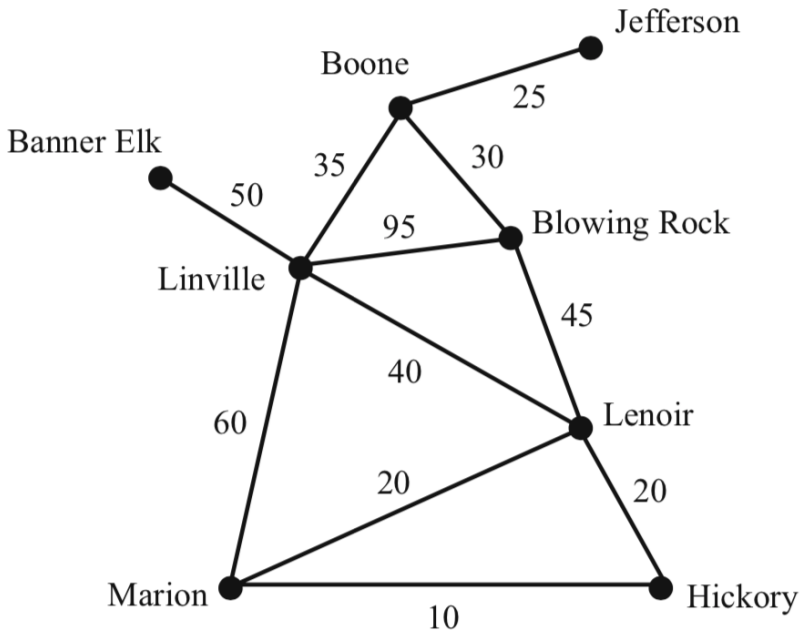

TABLE 10.5. *Steps of algorithm for shortest path from A to B*

| | *A* | *B* | *C* | *D* | *E* | *F* | *H* |
|---|---|---|---|---|---|---|---|
| Iteration 0 | [0] | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Iteration 1 | [0] | ∞ | [3] | ∞ | ∞ | [2] | ∞ |
| Iteration 2 | [0] | ∞ | [3] | ∞ | [3] | [2] | [7] |
| Iteration 3 | [0] | ∞ | [3] | [10] | [3] | [2] | [7] |
| Iteration 4 | [0] | [8] | [3] | [6] | [3] | [2] | [7] |
| Iteration 5 | [0] | [8] | [3] | [6] | [3] | [2] | [7] |
| Iteration 6 | [0] | [8] | [3] | [6] | [3] | [2] | [7] |

# Applying Dijkstra's algorithm to examples (cont.)




Question. What is the shortest possible distance a path connecting the cities of Marion and Boone can cover? What about a path connecting the cities of Marion and Jefferson? Can you find minimum weight paths too?


Jefferson
Boone
25
Banner Elk
35
30
50
95
Blowing Rock
Linville
45
40
60
Lenoir
20
20
Marion
Hickory
10