Part A Graph Theory

Marc Lackenby

Trinity Term 2022

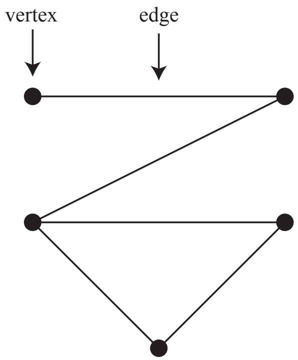
Graphs

Graph theory is the mathematical theory of networks.

Graphs

Graph theory is the mathematical theory of networks.

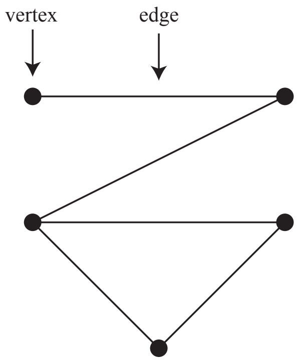
A graph has 'nodes' called vertices.



Graphs

Graph theory is the mathematical theory of networks.

A graph has ‘nodes’ called vertices.
These are connected by ‘lines’ called edges.

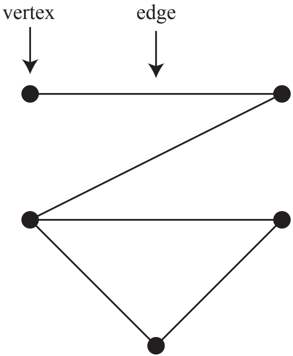


Graphs

Graph theory is the mathematical theory of networks.

A graph has ‘nodes’ called vertices.
These are connected by ‘lines’ called edges.

We will give a formal definition shortly.



Colouring maps

A famous result in graph theory is the Four Colour Theorem.

Colouring maps

A famous result in graph theory is the Four Colour Theorem.

This answers a question first posed by Francis Guthrie in 1852:

Colouring maps

A famous result in graph theory is the Four Colour Theorem.

This answers a question first posed by Francis Guthrie in 1852:

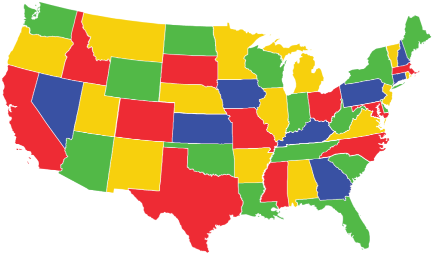
Is it possible to colour the countries using
only four different colours, so that any two countries
sharing a border receive different colours?

Colouring maps

A famous result in graph theory is the Four Colour Theorem.

This answers a question first posed by Francis Guthrie in 1852:

Is it possible to colour the countries using
only four different colours, so that any two countries
sharing a border receive different colours?

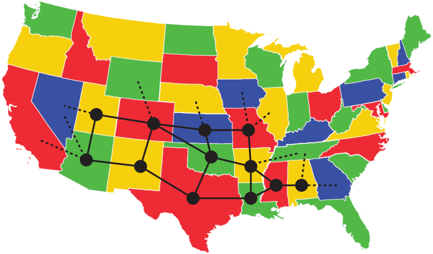


Colouring maps

A famous result in graph theory is the Four Colour Theorem.

This answers a question first posed by Francis Guthrie in 1852:

Is it possible to colour the countries using
only four different colours, so that any two countries
sharing a border receive different colours?

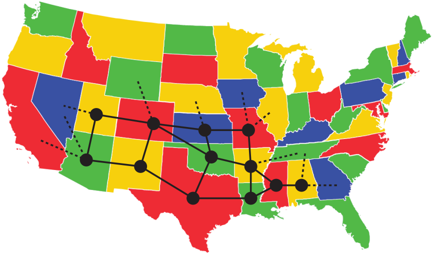


Colouring maps

A famous result in graph theory is the Four Colour Theorem.

This answers a question first posed by Francis Guthrie in 1852:

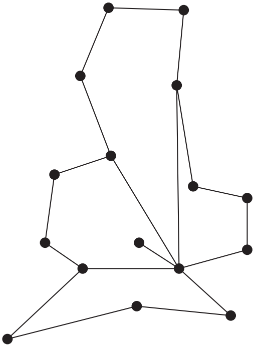
Is it possible to colour the countries using
only four different colours, so that any two countries
sharing a border receive different colours?



This was proved by Appel and Haken in 1976, using a controversial computer-assisted proof.

Connectedness

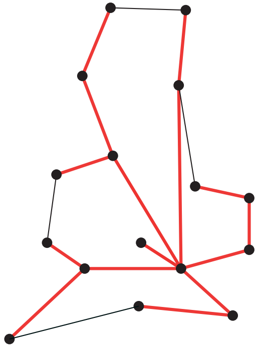
The government wants to build a new high speed rail network that links all of the major cities in the country.



Connectedness

The government wants to build a new high speed rail network that links all of the major cities in the country.

It wants to decide which existing rail lines to upgrade.

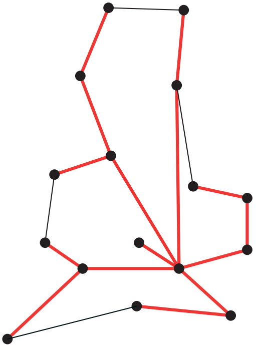


Connectedness

The government wants to build a new high speed rail network that links all of the major cities in the country.

It wants to decide which existing rail lines to upgrade.

The government's main priority is not to minimise journey times, but rather to minimise the cost subject to making a connected network.

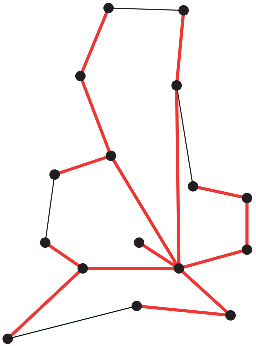


Connectedness

The government wants to build a new high speed rail network that links all of the major cities in the country.

It wants to decide which existing rail lines to upgrade.

The government's main priority is not to minimise journey times, but rather to minimise the cost subject to making a connected network.



Let us make some definitions and formulate this problem mathematically.

Definitions

A graph $G = (V(G), E(G))$ consists of two sets:

$V(G)$ (the vertex set) and
$E(G)$ (the edge set),

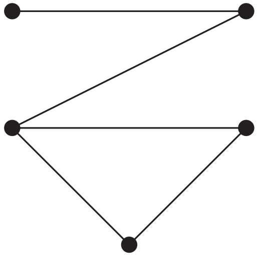
where each element of $E(G)$ consists of a pair of elements of $V(G)$.

Definitions

A graph $G = (V(G), E(G))$ consists of two sets:

$V(G)$ (the vertex set) and
$E(G)$ (the edge set),

where each element of $E(G)$ consists of a pair of elements of $V(G)$.

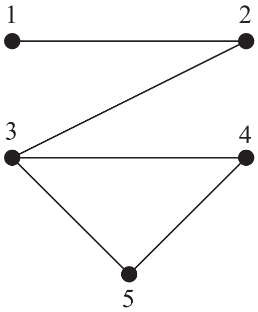


Definitions

A graph $G = (V(G), E(G))$ consists of two sets:

$V(G)$ (the vertex set) and
$E(G)$ (the edge set),

where each element of $E(G)$ consists of a pair of elements of $V(G)$.

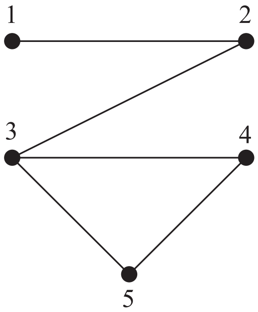


Definitions

A graph $G = (V(G), E(G))$ consists of two sets:

$V(G)$ (the vertex set) and
$E(G)$ (the edge set),

where each element of $E(G)$ consists of a pair of elements of $V(G)$.



$V(G) = \{1, 2, 3, 4, 5\}$

$E(G) = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{3, 5\}\}$

Definitions

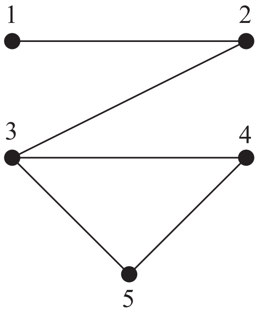
A graph $G = (V(G), E(G))$ consists of two sets:

$V(G)$ (the vertex set) and
$E(G)$ (the edge set),

where each element of $E(G)$ consists of a pair of elements of $V(G)$.

We will always assume without further comment that

$|V(G)|$ is finite.



$V(G) = \{1, 2, 3, 4, 5\}$

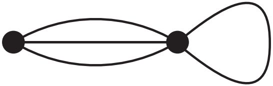
$E(G) = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{3, 5\}\}$

Definitions

We use the term ‘graph’ where some would say ‘simple graph’, using ‘graph’ for a more general structure which allows several ‘parallel’ edges between a given pair of vertices and ‘loop’ edges that join a vertex to itself.

Definitions

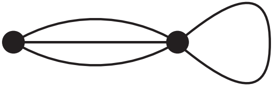
We use the term ‘graph’ where some would say ‘simple graph’, using ‘graph’ for a more general structure which allows several ‘parallel’ edges between a given pair of vertices and ‘loop’ edges that join a vertex to itself.



non-simple

Definitions

We use the term 'graph' where some would say 'simple graph', using 'graph' for a more general structure which allows several 'parallel' edges between a given pair of vertices and 'loop' edges that join a vertex to itself.



non-simple

We write $uv = \{u, v\} = vu$ for the (unordered) pair representing an edge between u and v.

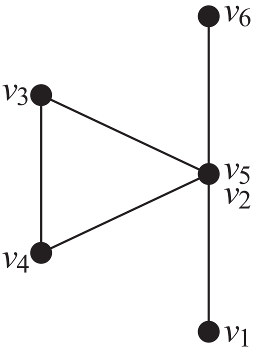
Connectedness

Walks, paths and cycles

Let G be a graph. A *walk* in G is a sequence W of vertices $v_1, \ldots, v_t$ such that $v_i v_{i+1} \in E(G)$ for all $1 \leq i < t$.

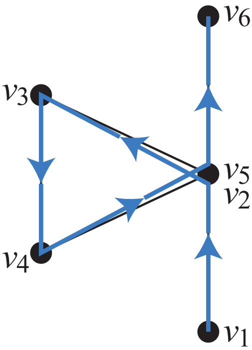
Walks, paths and cycles

Let G be a graph. A *walk* in G is a sequence W of vertices $v_1, \ldots, v_t$ such that $v_i v_{i+1} \in E(G)$ for all $1 \leq i < t$.



Walks, paths and cycles

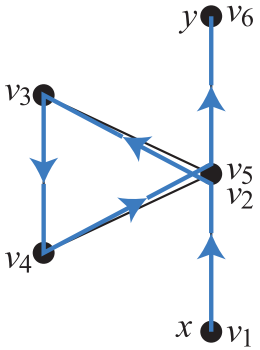
Let G be a graph. A *walk* in G is a sequence W of vertices $v_1, \ldots, v_t$ such that $v_i v_{i+1} \in E(G)$ for all $1 \leq i < t$.



Walks, paths and cycles

Let G be a graph. A *walk* in G is a sequence W of vertices $v_1, \ldots, v_t$ such that $v_i v_{i+1} \in E(G)$ for all $1 \leq i < t$.

If we want to specify the start and end then we call W an *xy-walk* with $x = v_1$ and $y = v_t$.



Walks, paths and cycles

Let G be a graph. A *walk* in G is a sequence W of vertices $v_1, \ldots, v_t$ such that $v_i v_{i+1} \in E(G)$ for all $1 \leq i < t$.

If we want to specify the start and end then we call W an *xy-walk* with $x = v_1$ and $y = v_t$.

If the vertices in W are distinct we call it a *path*, or if we want to specify the ends an *xy-path*.

Walks, paths and cycles

Let G be a graph. A *walk* in G is a sequence W of vertices $v_1, \ldots, v_t$ such that $v_i v_{i+1} \in E(G)$ for all $1 \leq i < t$.

If we want to specify the start and end then we call W an *xy-walk* with $x = v_1$ and $y = v_t$.

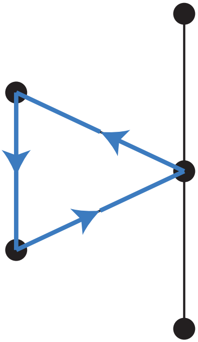
If the vertices in W are distinct we call it a *path*, or if we want to specify the ends an *xy-path*.

If $x = y$ we call W a *closed walk*.

Walks, paths and cycles

Let G be a graph. A *walk* in G is a sequence W of vertices $v_1, \ldots, v_t$ such that $v_i v_{i+1} \in E(G)$ for all $1 \leq i < t$.

If we want to specify the start and end then we call W an *xy-walk* with $x = v_1$ and $y = v_t$.



If the vertices in W are distinct we call it a *path*, or if we want to specify the ends an *xy-path*.

If $x = y$ we call W a *closed walk*.

If $x = y$ but the vertices are otherwise distinct and W has at least 3 vertices then we call W a *cycle*.

Walks, paths and cycles

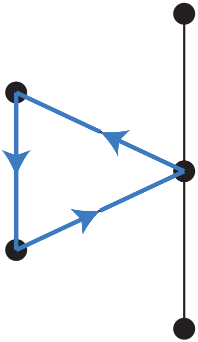
Let G be a graph. A *walk* in G is a sequence W of vertices $v_1, \ldots, v_t$ such that $v_i v_{i+1} \in E(G)$ for all $1 \leq i < t$.

If we want to specify the start and end then we call W an *xy-walk* with $x = v_1$ and $y = v_t$.

If the vertices in W are distinct we call it a *path*, or if we want to specify the ends an *xy-path*.

If $x = y$ we call W a *closed walk*.

If $x = y$ but the vertices are otherwise distinct and W has at least 3 vertices then we call W a *cycle*.



Walks, paths and cycles

Let G be a graph. A *walk* in G is a sequence W of vertices $v_1, \ldots, v_t$ such that $v_i v_{i+1} \in E(G)$ for all $1 \leq i < t$.

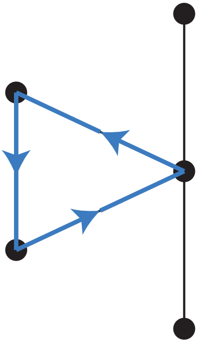
If we want to specify the start and end then we call W an *xy-walk* with $x = v_1$ and $y = v_t$.

If the vertices in W are distinct we call it a *path*, or if we want to specify the ends an *xy-path*.

If $x = y$ we call W a *closed walk*.

If $x = y$ but the vertices are otherwise distinct and W has at least 3 vertices then we call W a *cycle*.

We also regard paths and cycles as subgraphs of G.



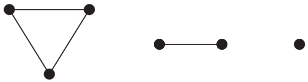
Connectedness and components

We say that G is *connected* if for any x, y in $V(G)$ there is an xy-walk in G.

Connectedness and components

We say that G is *connected* if for any x, y in $V(G)$ there is an xy-walk in G.

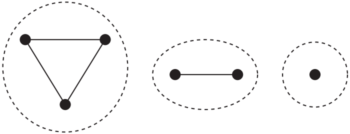
We say that two vertices x and y of a graph G *lie in the same component* if they are joined by an xy-walk.



Connectedness and components

We say that G is *connected* if for any x, y in $V(G)$ there is an xy-walk in G.

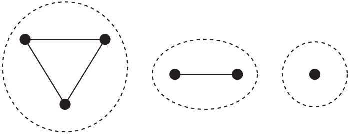
We say that two vertices x and y of a graph G *lie in the same component* if they are joined by an xy-walk.



Connectedness and components

We say that G is *connected* if for any x, y in $V(G)$ there is an xy-walk in G.

We say that two vertices x and y of a graph G *lie in the same component* if they are joined by an xy-walk.



Clearly this forms an equivalence relation and the partition of $V(G)$ into equivalence classes expresses G as a union of disjoint connected graphs called its *components*.

The high-speed network question

Let G be a connected graph.

The high-speed network question

Let G be a connected graph.

Suppose that for each edge $e \in E(G)$ we are given a 'cost' $c(e) > 0$.

The high-speed network question

Let G be a connected graph.

Suppose that for each edge $e \in E(G)$ we are given a 'cost' $c(e) > 0$.

For any $S \subseteq E(G)$ we call

$$c(S) = \sum_{e \in S} c(e)$$

the cost of S.

The high-speed network question

Let G be a connected graph.

Suppose that for each edge $e \in E(G)$ we are given a 'cost' $c(e) > 0$.

For any $S \subseteq E(G)$ we call

$$c(S) = \sum_{e \in S} c(e)$$

the cost of S.

Our task:

Find $S \subseteq E(G)$ with minimum possible $c(S)$ such that $(V(G), S)$ is a connected graph.

An inefficient algorithm

A silly way of solving this task would be to list all $S \subseteq E(G)$, check each one to see whether $(V(G), S)$ is a connected graph, compute $c(S)$ for each, and take the best one.

An inefficient algorithm

A silly way of solving this task would be to list all $S \subseteq E(G)$, check each one to see whether $(V(G), S)$ is a connected graph, compute $c(S)$ for each, and take the best one.

This is silly because there are $2^{|E(G)|}$ subsets of $E(G)$, so we could never check them all in practice unless G is very small.

An inefficient algorithm

A silly way of solving this task would be to list all $S \subseteq E(G)$, check each one to see whether $(V(G), S)$ is a connected graph, compute $c(S)$ for each, and take the best one.

This is silly because there are $2^{|E(G)|}$ subsets of $E(G)$, so we could never check them all in practice unless G is very small.

We are interested in 'efficient algorithms'. We will not define this concept precisely in this course, but it will be exemplified by the algorithms that we present.

Minimally connected graphs

What can we say about the possible $S \subseteq E(G)$ that solves our task?

Minimally connected graphs

What can we say about the possible $S \subseteq E(G)$ that solves our task?

One obvious property is that $(V(G), S)$ is 'minimally connected', i.e. $(V(G), S)$ is connected but $(V(G), S \setminus \{e\})$ is not connected for any $e \in S$ (otherwise we contradict minimality of $c(S)$).

Minimally connected graphs

What can we say about the possible $S \subseteq E(G)$ that solves our task?

One obvious property is that $(V(G), S)$ is 'minimally connected', i.e. $(V(G), S)$ is connected but $(V(G), S \setminus \{e\})$ is not connected for any $e \in S$ (otherwise we contradict minimality of $c(S)$).

This motivates the next section.

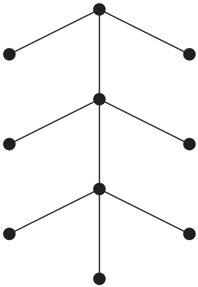
Trees

Trees

A *tree* is a minimally connected graph.

Trees

A *tree* is a minimally connected graph.



Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Lemma 1. Any tree is acyclic.

Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Lemma 1. Any tree is acyclic.

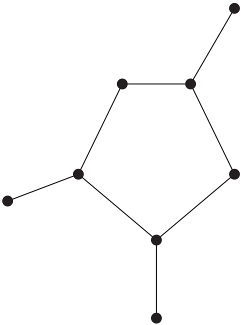
Proof. Let G be a tree, i.e. G is minimally connected.

Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Lemma 1. Any tree is acyclic.

Proof. Let G be a tree, i.e. G is minimally connected.
Suppose for a contradiction that G contains a cycle C.

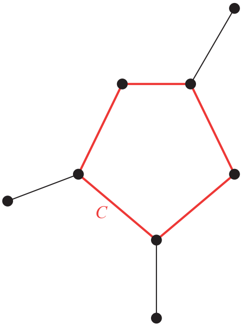


Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Lemma 1. Any tree is acyclic.

Proof. Let G be a tree, i.e. G is minimally connected.
Suppose for a contradiction that G contains a cycle C.

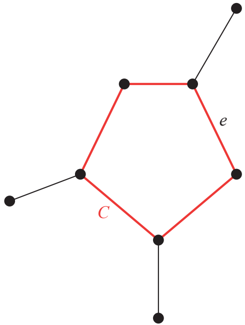


Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Lemma 1. Any tree is acyclic.

Proof. Let G be a tree, i.e. G is minimally connected.
Suppose for a contradiction that G contains a cycle C. Let $e \in E(C)$.

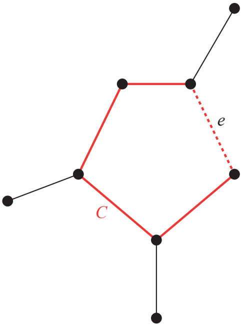


Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Lemma 1. Any tree is acyclic.

Proof. Let G be a tree, i.e. G is minimally connected.
Suppose for a contradiction that G contains a cycle C. Let $e \in E(C)$.
We will obtain our contradiction by showing that $G - e := (V(G), E(G) \setminus \{e\})$ is connected.

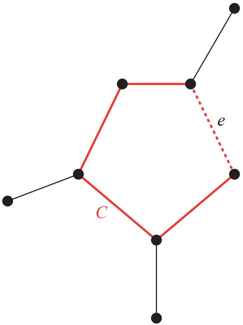


Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Lemma 1. Any tree is acyclic.

Proof. Let G be a tree, i.e. G is minimally connected.
Suppose for a contradiction that G contains a cycle C. Let $e \in E(C)$.
We will obtain our contradiction by showing that $G - e := (V(G), E(G) \setminus \{e\})$ is connected.
Let P be the path obtained by deleting e from C.

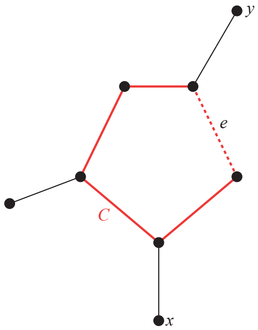


Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Lemma 1. Any tree is acyclic.

Proof. Let G be a tree, i.e. G is minimally connected.
Suppose for a contradiction that G contains a cycle C. Let $e \in E(C)$.
We will obtain our contradiction by showing that $G - e := (V(G), E(G) \setminus \{e\})$ is connected.
Let P be the path obtained by deleting e from C.
Consider any x, y in $V(G)$.

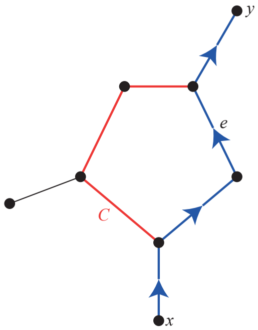


Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Lemma 1. Any tree is acyclic.

Proof. Let G be a tree, i.e. G is minimally connected.
Suppose for a contradiction that G contains a cycle C. Let $e \in E(C)$.
We will obtain our contradiction by showing that $G - e := (V(G), E(G) \setminus \{e\})$ is connected.
Let P be the path obtained by deleting e from C.
Consider any x, y in $V(G)$. As G is connected, there is an xy-walk W in G.

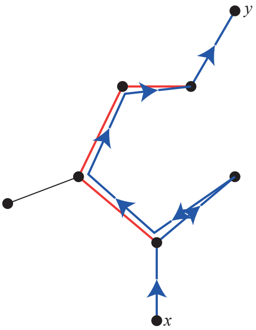


Acyclic graphs

If a graph G has no cycle we call it *acyclic*.

Lemma 1. Any tree is acyclic.

Proof. Let G be a tree, i.e. G is minimally connected.
Suppose for a contradiction that G contains a cycle C. Let $e \in E(C)$.
We will obtain our contradiction by showing that $G - e := (V(G), E(G) \setminus \{e\})$ is connected.
Let P be the path obtained by deleting e from C.
Consider any x, y in $V(G)$. As G is connected, there is an xy-walk W in G.
Replacing any use of e in W by P gives an xy-walk in $G - e$. Thus $G - e$ is connected, contradiction. □



A characterisation of trees

There are many equivalent characterisations of trees, any of which could be taken as the definition. Here is one:

A characterisation of trees

There are many equivalent characterisations of trees, any of which could be taken as the definition. Here is one:

Lemma 2. G is a tree if and only if G is connected and acyclic.

A characterisation of trees

There are many equivalent characterisations of trees, any of which could be taken as the definition. Here is one:

Lemma 2. G is a tree if and only if G is connected and acyclic.

Proof. ($\Rightarrow$) If G is a tree then G is connected by definition and acyclic by Lemma 1.

A characterisation of trees

There are many equivalent characterisations of trees, any of which could be taken as the definition. Here is one:

Lemma 2. G is a tree if and only if G is connected and acyclic.

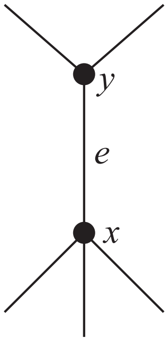
Proof. ($\Rightarrow$) If G is a tree then G is connected by definition and acyclic by Lemma 1.

($\Leftarrow$) Conversely, let G be connected and acyclic.

A characterisation of trees

There are many equivalent characterisations of trees, any of which could be taken as the definition. Here is one:

Lemma 2. G is a tree if and only if G is connected and acyclic.



Proof. ($\Rightarrow$) If G is a tree then G is connected by definition and acyclic by Lemma 1.

($\Leftarrow$) Conversely, let G be connected and acyclic. Suppose for a contradiction that $G - e$ is connected for some $e = xy \in E(G)$.

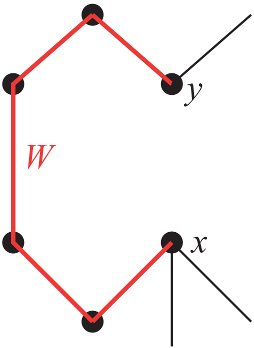
A characterisation of trees

There are many equivalent characterisations of trees, any of which could be taken as the definition. Here is one:

Lemma 2. G is a tree if and only if G is connected and acyclic.

Proof. ($\Rightarrow$) If G is a tree then G is connected by definition and acyclic by Lemma 1.

($\Leftarrow$) Conversely, let G be connected and acyclic. Suppose for a contradiction that $G - e$ is connected for some $e = xy \in E(G)$. Let W be a shortest xy-walk in $G - e$. Then W must be a path, i.e. have no repeated vertices, otherwise we would find a shorter walk by deleting a segment of W between two visits to the same vertex.



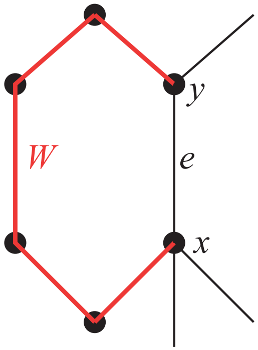
A characterisation of trees

There are many equivalent characterisations of trees, any of which could be taken as the definition. Here is one:

Lemma 2. G is a tree if and only if G is connected and acyclic.

Proof. ($\Rightarrow$) If G is a tree then G is connected by definition and acyclic by Lemma 1.

($\Leftarrow$) Conversely, let G be connected and acyclic. Suppose for a contradiction that $G - e$ is connected for some $e = xy \in E(G)$. Let W be a shortest xy-walk in $G - e$. Then W must be a path, i.e. have no repeated vertices, otherwise we would find a shorter walk by deleting a segment of W between two visits to the same vertex. Combining W with xy gives a cycle, contradiction. □



Paths in trees

The fact that a shortest walk between two points is a path is often useful. More generally, considering an extremal (shortest, longest, minimal, maximal, ...) object is often a useful proof technique. Another example:

Paths in trees

The fact that a shortest walk between two points is a path is often useful. More generally, considering an extremal (shortest, longest, minimal, maximal, . . .) object is often a useful proof technique. Another example:

Lemma 3. Any two vertices in a tree are joined by a unique path.

Paths in trees

The fact that a shortest walk between two points is a path is often useful. More generally, considering an extremal (shortest, longest, minimal, maximal, ...) object is often a useful proof technique. Another example:

Lemma 3. Any two vertices in a tree are joined by a unique path.

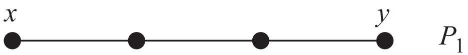
Proof. Suppose for a contradiction that this fails for some tree G.

Paths in trees

The fact that a shortest walk between two points is a path is often useful. More generally, considering an extremal (shortest, longest, minimal, maximal, ...) object is often a useful proof technique. Another example:

Lemma 3. Any two vertices in a tree are joined by a unique path.

Proof. Suppose for a contradiction that this fails for some tree G. Choose x, y in $V(G)$ so that there are distinct xy-paths P_1, P_2, and P_1 is as short as possible over all such choices of x and y.

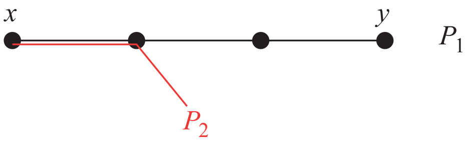


Paths in trees

The fact that a shortest walk between two points is a path is often useful. More generally, considering an extremal (shortest, longest, minimal, maximal, ...) object is often a useful proof technique. Another example:

Lemma 3. Any two vertices in a tree are joined by a unique path.

Proof. Suppose for a contradiction that this fails for some tree G. Choose x, y in $V(G)$ so that there are distinct xy-paths P_1, P_2, and P_1 is as short as possible over all such choices of x and y. Then P_1 and P_2 only intersect in x and y.

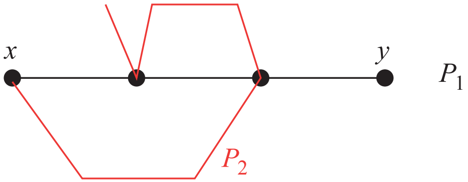


Paths in trees

The fact that a shortest walk between two points is a path is often useful. More generally, considering an extremal (shortest, longest, minimal, maximal, . . .) object is often a useful proof technique. Another example:

Lemma 3. Any two vertices in a tree are joined by a unique path.

Proof. Suppose for a contradiction that this fails for some tree G. Choose x, y in $V(G)$ so that there are distinct xy-paths P_1, P_2, and P_1 is as short as possible over all such choices of x and y. Then P_1 and P_2 only intersect in x and y.

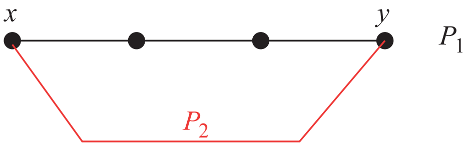


Paths in trees

The fact that a shortest walk between two points is a path is often useful. More generally, considering an extremal (shortest, longest, minimal, maximal, ...) object is often a useful proof technique. Another example:

Lemma 3. Any two vertices in a tree are joined by a unique path.

Proof. Suppose for a contradiction that this fails for some tree G. Choose x, y in $V(G)$ so that there are distinct xy-paths P_1, P_2, and P_1 is as short as possible over all such choices of x and y. Then P_1 and P_2 only intersect in x and y. So their union is a cycle, contradicting Lemma 2. □



Adjacency and degree

Let G be a graph.

Adjacency and degree

Let G be a graph.

If $uv \in E(G)$ we say that u and v are *neighbours*. We also say that u and v are *adjacent*.

Adjacency and degree

Let G be a graph.

If $uv \in E(G)$ we say that u and v are *neighbours*. We also say that u and v are *adjacent*.

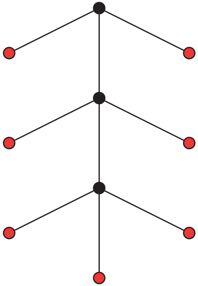
The degree $d(v)$ of v is the number of neighbours of v in G.

Leaves

A *leaf* is a vertex of degree one, i.e. with a unique neighbour.

Leaves

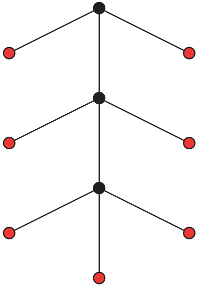
A *leaf* is a vertex of degree one, i.e. with a unique neighbour.



Leaves

A *leaf* is a vertex of degree one, i.e. with a unique neighbour.

Lemma 4. Any tree with at least two vertices has at least two leaves.

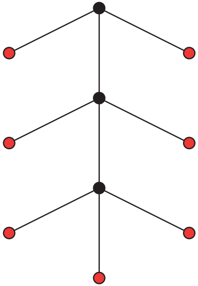


Leaves

A *leaf* is a vertex of degree one, i.e. with a unique neighbour.

Lemma 4. Any tree with at least two vertices has at least two leaves.

Proof. Consider any tree G.

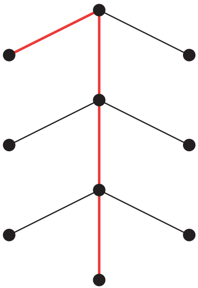


Leaves

A *leaf* is a vertex of degree one, i.e. with a unique neighbour.

Lemma 4. Any tree with at least two vertices has at least two leaves.

Proof. Consider any tree G. Let P be a longest path in G.

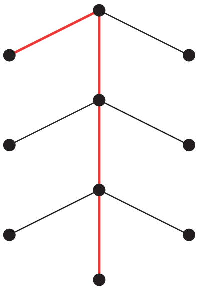


Leaves

A *leaf* is a vertex of degree one, i.e. with a unique neighbour.

Lemma 4. Any tree with at least two vertices has at least two leaves.

Proof. Consider any tree G. Let P be a longest path in G. The two ends of P must be leaves.

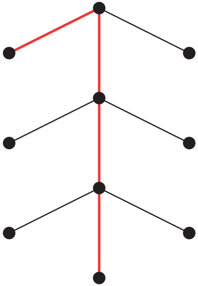


Leaves

A *leaf* is a vertex of degree one, i.e. with a unique neighbour.

Lemma 4. Any tree with at least two vertices has at least two leaves.

Proof. Consider any tree G. Let P be a longest path in G. The two ends of P must be leaves. Indeed, an end cannot have a neighbour in $V(G) \setminus V(P)$, or we could make P longer, and cannot have any neighbour in $V(P)$ other than the next in the sequence of P, or we would have a cycle. □



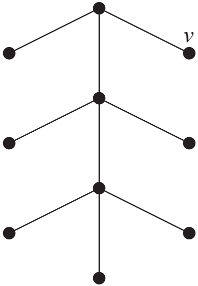
Removing a leaf from a tree

Given $v \in V(G)$, let $G - v$ be the graph with
$V(G - v) = V(G) \setminus \{v\}$ and
$E(G - v) = \{xy \in E(G) : v \notin \{x, y\}\}$.

Removing a leaf from a tree

Given $v \in V(G)$, let $G - v$ be the graph with $V(G - v) = V(G) \setminus \{v\}$ and $E(G - v) = \{xy \in E(G) : v \notin \{x, y\}\}$.

Lemma 5. If G is a tree and v is a leaf of G then $G - v$ is a tree.

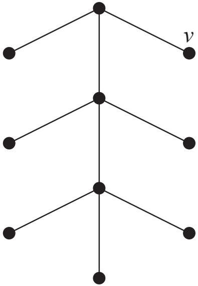


Removing a leaf from a tree

Given $v \in V(G)$, let $G - v$ be the graph with
$V(G - v) = V(G) \setminus \{v\}$ and
$E(G - v) = \{xy \in E(G) : v \notin \{x, y\}\}$.

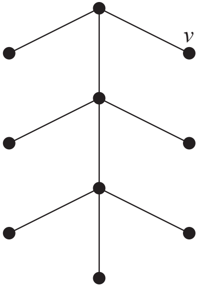
Lemma 5. If G is a tree and v is a leaf of G then $G - v$ is a tree.

Proof. By Lemma 2 it suffices to show that $G - v$ is connected and acyclic.



Removing a leaf from a tree

Given $v \in V(G)$, let $G - v$ be the graph with $V(G - v) = V(G) \setminus \{v\}$ and $E(G - v) = \{xy \in E(G) : v \notin \{x, y\}\}$.

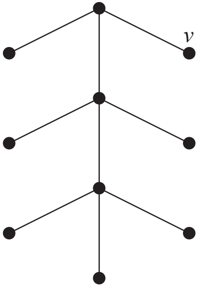


Lemma 5. If G is a tree and v is a leaf of G then $G - v$ is a tree.

Proof. By Lemma 2 it suffices to show that $G - v$ is connected and acyclic. Acyclicity is immediate from Lemma 2.

Removing a leaf from a tree

Given $v \in V(G)$, let $G - v$ be the graph with $V(G - v) = V(G) \setminus \{v\}$ and $E(G - v) = \{xy \in E(G) : v \notin \{x, y\}\}$.



Lemma 5. If G is a tree and v is a leaf of G then $G - v$ is a tree.

Proof. By Lemma 2 it suffices to show that $G - v$ is connected and acyclic. Acyclicity is immediate from Lemma 2. Connectivity follows by noting for any x, y in $V(G) \setminus \{v\}$ that the unique xy-path in G is contained in $G - v$. $\square$

The number of edges in a tree

Lemma 6. Any tree on n vertices has $n - 1$ edges.

The number of edges in a tree

Lemma 6. Any tree on n vertices has $n - 1$ edges.

Proof. By induction on the number of vertices.

The number of edges in a tree

Lemma 6. Any tree on n vertices has $n - 1$ edges.

Proof. By induction on the number of vertices. A tree with 1 vertex has 0 edges.

The number of edges in a tree

Lemma 6. Any tree on n vertices has $n-1$ edges.

Proof. By induction on the number of vertices. A tree with 1 vertex has 0 edges. Let G be a tree on $n > 1$ vertices.

The number of edges in a tree

Lemma 6. Any tree on n vertices has $n - 1$ edges.

Proof. By induction on the number of vertices. A tree with 1 vertex has 0 edges. Let G be a tree on $n > 1$ vertices. By Lemma 4, G has a leaf v.

The number of edges in a tree

Lemma 6. Any tree on n vertices has $n-1$ edges.

Proof. By induction on the number of vertices. A tree with 1 vertex has 0 edges. Let G be a tree on $n > 1$ vertices. By Lemma 4, G has a leaf v. By Lemma 5, $G - v$ is a tree.

The number of edges in a tree

Lemma 6. Any tree on n vertices has $n-1$ edges.

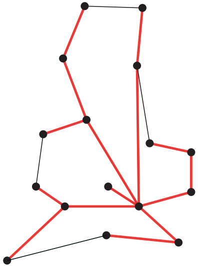
Proof. By induction on the number of vertices. A tree with 1 vertex has 0 edges. Let G be a tree on $n > 1$ vertices. By Lemma 4, G has a leaf v. By Lemma 5, $G - v$ is a tree. By the induction hypothesis, $G - v$ has $n - 2$ edges. Replacing v gives $n - 1$ edges in G. □

Spanning trees

Any connected graph G contains a minimally connected subgraph (i.e. a tree) with the same vertex set, which we call a *spanning tree* of G.

Spanning trees

Any connected graph G contains a minimally connected subgraph (i.e. a tree) with the same vertex set, which we call a *spanning tree* of G.



Another characterisation of trees

Lemma 7. A graph G is a tree on n vertices if and only if G is connected and has $n - 1$ edges.

Another characterisation of trees

Lemma 7. A graph G is a tree on n vertices if and only if G is connected and has $n - 1$ edges.

Proof. If G is a tree then G is connected by definition and has $n - 1$ edges by Lemma 6.

Another characterisation of trees

Lemma 7. A graph G is a tree on n vertices if and only if G is connected and has $n-1$ edges.

Proof. If G is a tree then G is connected by definition and has $n-1$ edges by Lemma 6.

Conversely, suppose that G is connected and has $n-1$ edges. Let H be a spanning tree of G. Then H has $n-1$ edges by Lemma 6, so $H = G$, so G is a tree. $\square$

Minimum Cost Spanning Trees

Minimum cost spanning trees

Recall our high-speed rail network problem:

Minimum cost spanning trees

Recall our high-speed rail network problem:

Let G be a connected graph.

Minimum cost spanning trees

Recall our high-speed rail network problem:

Let G be a connected graph.

Suppose that for each edge $e \in E(G)$ we are given a 'cost' $c(e) > 0$.

Minimum cost spanning trees

Recall our high-speed rail network problem:

Let G be a connected graph.

Suppose that for each edge $e \in E(G)$ we are given a 'cost' $c(e) > 0$.

For any $S \subseteq E(G)$ we call

$$c(S) = \sum_{e \in S} c(e)$$

the cost of S.

Minimum cost spanning trees

Recall our high-speed rail network problem:

Let G be a connected graph.

Suppose that for each edge $e \in E(G)$ we are given a 'cost' $c(e) > 0$.

For any $S \subseteq E(G)$ we call

$$c(S) = \sum_{e \in S} c(e)$$

the cost of S.

Recall that a spanning tree for G is a tree $T = (V(G), S)$ where $S \subseteq E(G)$.

Minimum cost spanning trees

Recall our high-speed rail network problem:

Let G be a connected graph.

Suppose that for each edge $e \in E(G)$ we are given a 'cost' $c(e) > 0$.

For any $S \subseteq E(G)$ we call

$$c(S) = \sum_{e \in S} c(e)$$

the cost of S.

Recall that a spanning tree for G is a tree $T = (V(G), S)$ where $S \subseteq E(G)$.

A spanning tree T for G has minimum cost if any other spanning tree T' satisfies $c(T') \geq c(T)$.

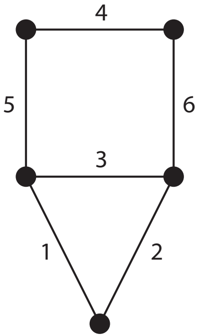
Kruskal's algorithm

How can we find a minimum cost spanning tree efficiently?

Kruskal's algorithm

How can we find a minimum cost spanning tree efficiently?

Kruskal's Algorithm.

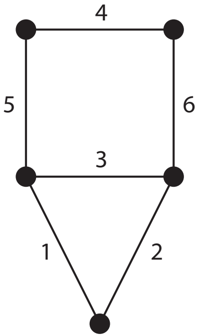


Kruskal's algorithm

How can we find a minimum cost spanning tree efficiently?

Kruskal's Algorithm.

At step $i \geq 0$, we will keep track of a subset $A_i \subseteq E(G)$. This will have the property that $(V(G), A_i)$ is acyclic.



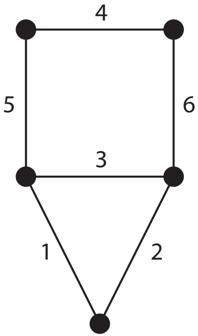
Kruskal's algorithm

How can we find a minimum cost spanning tree efficiently?

Kruskal's Algorithm.

At step $i \geq 0$, we will keep track of a subset $A_i \subseteq E(G)$. This will have the property that $(V(G), A_i)$ is acyclic.

Start with $A_0 = \emptyset$.



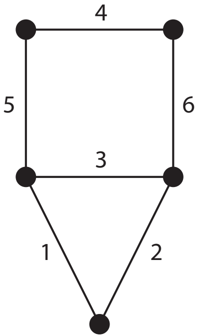
Kruskal's algorithm

How can we find a minimum cost spanning tree efficiently?

Kruskal's Algorithm.

At step $i \geq 0$, we will keep track of a subset $A_i \subseteq E(G)$. This will have the property that $(V(G), A_i)$ is acyclic.

Start with $A_0 = \emptyset$.
At step $i \geq 0$, is there an edge $e \in E(G) \setminus A_i$ such that $(V(G), A_i \cup \{e\})$ is acyclic? If no, then output $A = A_i$ and stop. If yes, then set $A_{i+1} = A_i \cup \{e\}$ for one such e such that $c(e)$ is minimal, and proceed to step $i + 1$.



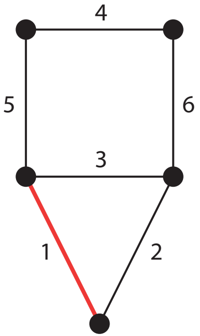
Kruskal's algorithm

How can we find a minimum cost spanning tree efficiently?

Kruskal's Algorithm.

At step $i \geq 0$, we will keep track of a subset $A_i \subseteq E(G)$. This will have the property that $(V(G), A_i)$ is acyclic.

Start with $A_0 = \emptyset$.
At step $i \geq 0$, is there an edge $e \in E(G) \setminus A_i$ such that $(V(G), A_i \cup \{e\})$ is acyclic? If no, then output $A = A_i$ and stop. If yes, then set $A_{i+1} = A_i \cup \{e\}$ for one such e such that $c(e)$ is minimal, and proceed to step $i + 1$.



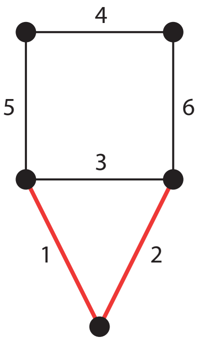
Kruskal's algorithm

How can we find a minimum cost spanning tree efficiently?

Kruskal's Algorithm.

At step $i \geq 0$, we will keep track of a subset $A_i \subseteq E(G)$. This will have the property that $(V(G), A_i)$ is acyclic.

Start with $A_0 = \emptyset$.
At step $i \geq 0$, is there an edge $e \in E(G) \setminus A_i$ such that $(V(G), A_i \cup \{e\})$ is acyclic? If no, then output $A = A_i$ and stop. If yes, then set $A_{i+1} = A_i \cup \{e\}$ for one such e such that $c(e)$ is minimal, and proceed to step $i + 1$.



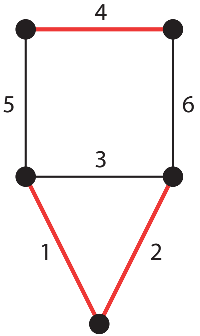
Kruskal's algorithm

How can we find a minimum cost spanning tree efficiently?

Kruskal's Algorithm.

At step $i \geq 0$, we will keep track of a subset $A_i \subseteq E(G)$. This will have the property that $(V(G), A_i)$ is acyclic.

Start with $A_0 = \emptyset$.
At step $i \geq 0$, is there an edge $e \in E(G) \setminus A_i$ such that $(V(G), A_i \cup \{e\})$ is acyclic? If no, then output $A = A_i$ and stop. If yes, then set $A_{i+1} = A_i \cup \{e\}$ for one such e such that $c(e)$ is minimal, and proceed to step $i + 1$.



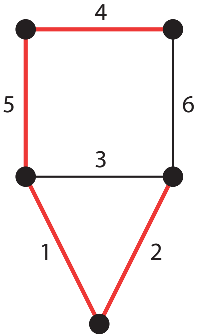
Kruskal's algorithm

How can we find a minimum cost spanning tree efficiently?

Kruskal's Algorithm.

At step $i \geq 0$, we will keep track of a subset $A_i \subseteq E(G)$. This will have the property that $(V(G), A_i)$ is acyclic.

Start with $A_0 = \emptyset$.
At step $i \geq 0$, is there an edge $e \in E(G) \setminus A_i$ such that $(V(G), A_i \cup \{e\})$ is acyclic? If no, then output $A = A_i$ and stop. If yes, then set $A_{i+1} = A_i \cup \{e\}$ for one such e such that $c(e)$ is minimal, and proceed to step $i + 1$.



Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Proof.

Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Proof. $(V(G), A)$ is a spanning tree of G.

Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

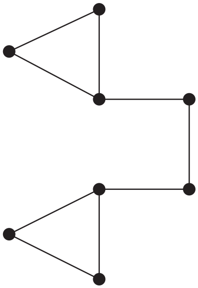
Proof. $(V(G), A)$ is a spanning tree of G.

By construction, it is is acyclic.

Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Proof. $(V(G), A)$ is a spanning tree of G.



By construction, it is is acyclic.

Suppose, for a contradiction, that $(V(G), A)$ is not connected.

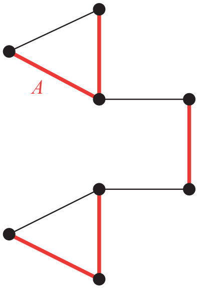
Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Proof. $(V(G), A)$ is a spanning tree of G.

By construction, it is is acyclic.

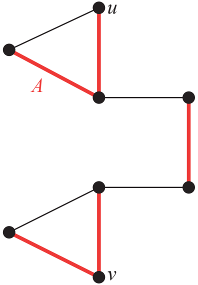
Suppose, for a contradiction, that $(V(G), A)$ is not connected.



Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Proof. $(V(G), A)$ is a spanning tree of G.



By construction, it is is acyclic.

Suppose, for a contradiction, that $(V(G), A)$ is not connected. Let u, v lie in different components of $(V(G), A)$.

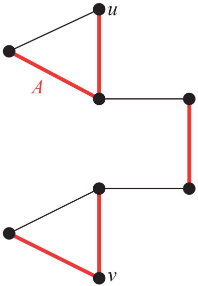
Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Proof. $(V(G), A)$ is a spanning tree of G.

By construction, it is is acyclic.

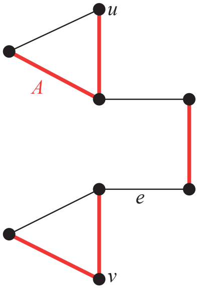
Suppose, for a contradiction, that $(V(G), A)$ is not connected. Let u, v lie in different components of $(V(G), A)$.
As G is connected, there is a uv-walk.



Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Proof. $(V(G), A)$ is a spanning tree of G.



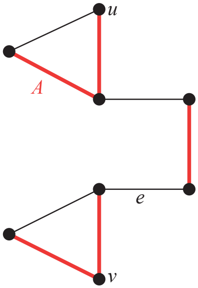
By construction, it is is acyclic.

Suppose, for a contradiction, that $(V(G), A)$ is not connected. Let u, v lie in different components of $(V(G), A)$.
As G is connected, there is a uv-walk. This must contain an edge e of G whose endpoints are in different components of $(V(G), A)$.

Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Proof. $(V(G), A)$ is a spanning tree of G.



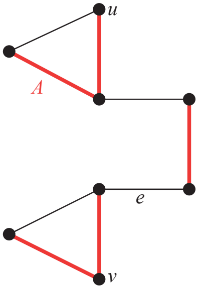
By construction, it is is acyclic.

Suppose, for a contradiction, that $(V(G), A)$ is not connected. Let u, v lie in different components of $(V(G), A)$.
As G is connected, there is a uv-walk. This must contain an edge e of G whose endpoints are in different components of $(V(G), A)$. So $A \cup \{e\}$ is acyclic.

Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Proof. $(V(G), A)$ is a spanning tree of G.



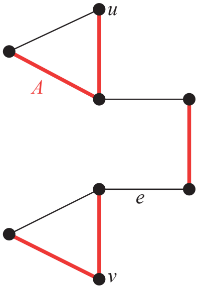
By construction, it is is acyclic.

Suppose, for a contradiction, that $(V(G), A)$ is not connected. Let u, v lie in different components of $(V(G), A)$.
As G is connected, there is a uv-walk. This must contain an edge e of G whose endpoints are in different components of $(V(G), A)$. So $A \cup \{e\}$ is acyclic. So, the algorithm should not have terminated when it did. Instead, it should have added e to A_i.

Kruskal's algorithm

Theorem 9. $(V(G), A)$ is a minimum cost spanning tree of G.

Proof. $(V(G), A)$ is a spanning tree of G.



By construction, it is is acyclic.

Suppose, for a contradiction, that $(V(G), A)$ is not connected. Let u, v lie in different components of $(V(G), A)$.
As G is connected, there is a uv-walk. This must contain an edge e of G whose endpoints are in different components of $(V(G), A)$. So $A \cup \{e\}$ is acyclic. So, the algorithm should not have terminated when it did. Instead, it should have added e to A_i.
This contradiction shows that $(V(G), A)$ is a spanning tree of G.

Kruskal's algorithm

$(V(G), A)$ has minimum cost.

Kruskal's algorithm

$(V(G), A)$ has minimum cost.

Let $\mathcal{M}$ be the set of $B \subset E(G)$ such that $(V(G), B)$ is a MCST.

Kruskal's algorithm

$(V(G), A)$ has minimum cost.

Let $\mathcal{M}$ be the set of $B \subset E(G)$ such that $(V(G), B)$ is a MCST.

We will prove by induction on i that

$$(*) \qquad \text{there is a } B \in \mathcal{M} \text{ with } A_i \subseteq B.$$

Kruskal's algorithm

$(V(G), A)$ has minimum cost.

Let $\mathcal{M}$ be the set of $B \subset E(G)$ such that $(V(G), B)$ is a MCST.

We will prove by induction on i that

$$(*) \qquad \text{there is a } B \in \mathcal{M} \text{ with } A_i \subseteq B.$$

Note that $(*)$ will suffice to prove the theorem, as when we apply it to $A_i = A$ we will have $A \subseteq B$ for some $B \in \mathcal{M}$ and so $|A| = |B|$ by Lemma 6, and so $A = B \in \mathcal{M}$.

Kruskal's algorithm

$$(*) \qquad \text{there is a } B \in \mathcal{M} \text{ with } A_i \subseteq B.$$

Kruskal's algorithm

$$(*) \qquad \text{there is a } B \in \mathcal{M} \text{ with } A_i \subseteq B.$$

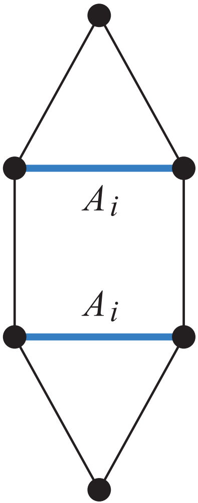
Base case $i = 0$ of (*). We have $A_0 = \emptyset$, so any $B \in \mathcal{M}$ satisfies (*).

Kruskal's algorithm

$$(*) \quad \text{there is a } B \in \mathcal{M} \text{ with } A_i \subseteq B.$$

Base case $i = 0$ of (*). We have $A_0 = \emptyset$, so any $B \in \mathcal{M}$ satisfies (*).

Induction step. Suppose for some $i \geq 0$ we have $A_i \subseteq B \in \mathcal{M}$. We can suppose $A_i \neq A$, otherwise the proof is complete.

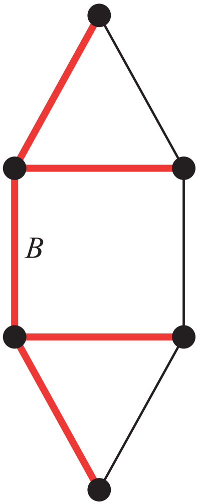


Kruskal's algorithm

$$(*) \quad \text{there is a } B \in \mathcal{M} \text{ with } A_i \subseteq B.$$

Base case $i = 0$ of (*). We have $A_0 = \emptyset$, so any $B \in \mathcal{M}$ satisfies (*).

Induction step. Suppose for some $i \geq 0$ we have $A_i \subseteq B \in \mathcal{M}$. We can suppose $A_i \neq A$, otherwise the proof is complete.

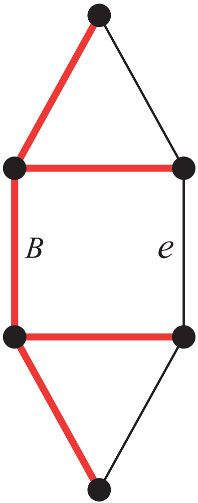


Kruskal's algorithm

$$(*) \quad \text{there is a } B \in \mathcal{M} \text{ with } A_i \subseteq B.$$

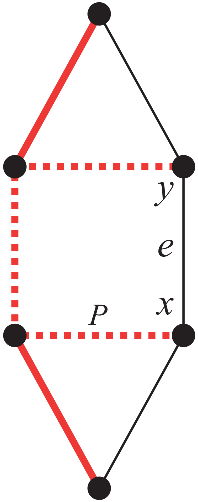
Base case $i = 0$ of (*). We have $A_0 = \emptyset$, so any $B \in \mathcal{M}$ satisfies (*).

Induction step. Suppose for some $i \geq 0$ we have $A_i \subseteq B \in \mathcal{M}$. We can suppose $A_i \neq A$, otherwise the proof is complete.
Consider $A_{i+1} = A_i \cup \{e\}$ given by the algorithm. We need to find $B' \in \mathcal{M}$ with $A_{i+1} \subseteq B'$. We can assume $e \notin B$, otherwise we could take $B' = B$.



Kruskal's algorithm

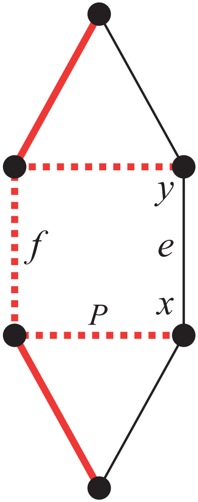
Let $e = xy$ and let P be the unique xy-path in the spanning tree $(V(G), B)$.



Kruskal's algorithm

Let $e = xy$ and let P be the unique xy-path in the spanning tree $(V(G), B)$.

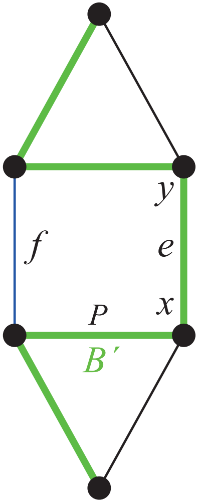
Then $C = P \cup \{e\}$ is a cycle. As A_{i+1} is acyclic, we can choose $f \in C \setminus A_{i+1}$.



Kruskal's algorithm

Let $e = xy$ and let P be the unique xy-path in the spanning tree $(V(G), B)$.

Then $C = P \cup \{e\}$ is a cycle. As A_{i+1} is acyclic, we can choose $f \in C \setminus A_{i+1}$.
Let $B' = (B \setminus \{f\}) \cup \{e\}$.



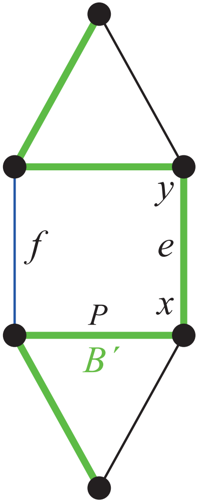
Kruskal's algorithm

Let $e = xy$ and let P be the unique xy-path in the spanning tree $(V(G), B)$.

Then $C = P \cup \{e\}$ is a cycle. As A_{i+1} is acyclic, we can choose $f \in C \setminus A_{i+1}$. Let $B' = (B \setminus \{f\}) \cup \{e\}$.

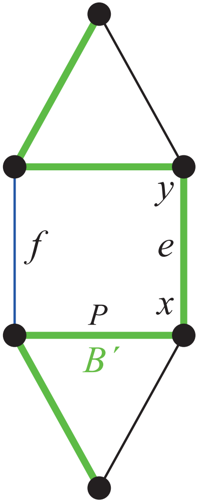
To finish the proof we need to show that

1. $A_{i+1} \subseteq B'$,
2. $(V(G), B')$ is a spanning tree, and
3. $c(B') \leq c(B)$.



Kruskal's algorithm

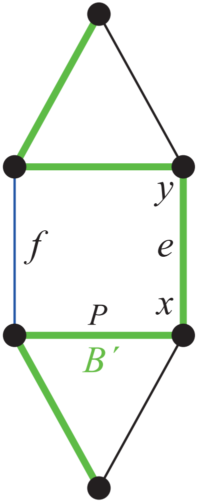
$A_{i+1} \subseteq B'$:



Kruskal's algorithm

$A_{i+1} \subseteq B'$:

Note that $A_{i+1} = A_i \cup \{e\} \subseteq B'$, as $A_i \subseteq B$ and $f \notin A_{i+1}$.

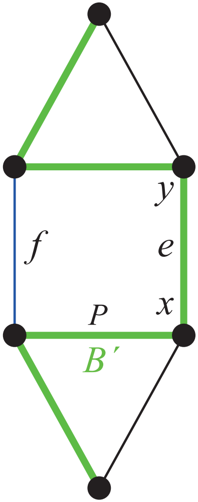


Kruskal's algorithm

$A_{i+1} \subseteq B'$:

Note that $A_{i+1} = A_i \cup \{e\} \subseteq B'$, as $A_i \subseteq B$ and $f \notin A_{i+1}$.

$(V(G), B')$ is a spanning tree:



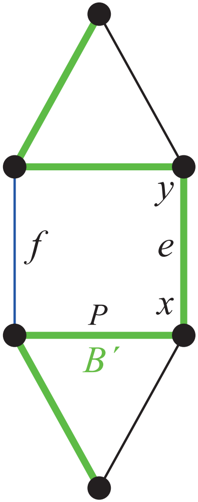
Kruskal's algorithm

$A_{i+1} \subseteq B'$:

Note that $A_{i+1} = A_i \cup \{e\} \subseteq B'$, as $A_i \subseteq B$ and $f \notin A_{i+1}$.

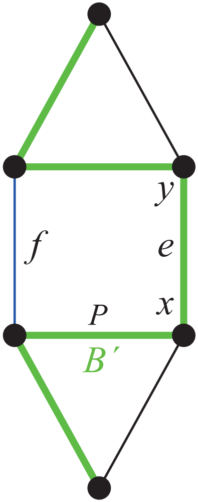
$(V(G), B')$ is a spanning tree:

Note that B' is connected, for the following reason. Any two vertices in $V(G)$ are joined by a path in B. Replace each occurence of f in this path by $C \setminus \{f\}$. Also B' has $|V(G)| - 1$ edges. So it is a spanning tree by Lemma 7.



Kruskal's algorithm

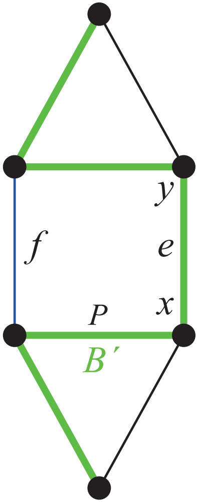
$c(B') \leq c(B)$:



Kruskal's algorithm

$c(B') \leq c(B)$:

Note that $A_i \cup \{f\} \subseteq B$, so $A_i \cup \{f\}$ is acyclic.

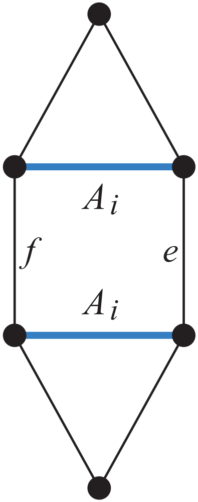


Kruskal's algorithm

$c(B') \leq c(B)$:

Note that $A_i \cup \{f\} \subseteq B$, so $A_i \cup \{f\}$ is acyclic.

Now e was chosen so that $c(e)$ is minimal among all edges e such that $A_i \cup \{e\}$ is acyclic. Hence, $c(e) \leq c(f)$.



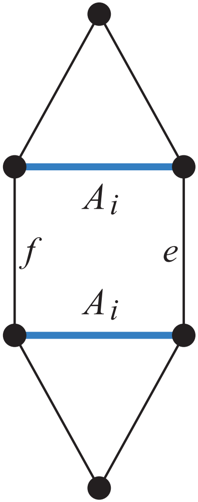
Kruskal's algorithm

$c(B') \leq c(B)$:

Note that $A_i \cup \{f\} \subseteq B$, so $A_i \cup \{f\}$ is acyclic.

Now e was chosen so that $c(e)$ is minimal among all edges e such that $A_i \cup \{e\}$ is acyclic. Hence, $c(e) \leq c(f)$.

So $c(B') = c(B) - c(f) + c(e) \leq c(B)$.



Kruskal's algorithm

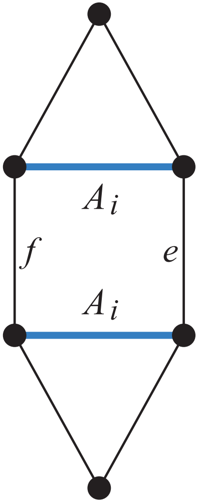
$c(B') \leq c(B)$:

Note that $A_i \cup \{f\} \subseteq B$, so $A_i \cup \{f\}$ is acyclic.

Now e was chosen so that $c(e)$ is minimal among all edges e such that $A_i \cup \{e\}$ is acyclic. Hence, $c(e) \leq c(f)$.

So $c(B') = c(B) - c(f) + c(e) \leq c(B)$.

This finishes the proof of the inductive step of $(*)$, and so of the theorem. □



The number of steps of Kruskal's algorithm

How fast is this algorithm?

The number of steps of Kruskal's algorithm

How fast is this algorithm?

To make this question mathematically precise would take us far afield (we would need to define a model of computation). In this course, we will take the intuitive approach of estimating the number of 'steps' taken by an algorithm, where a 'step' should be a 'simple' operation.

The number of steps of Kruskal's algorithm

In each iteration we add an edge, so there will be $|V(G)| - 1$ iterations.

The number of steps of Kruskal's algorithm

In each iteration we add an edge, so there will be $|V(G)| - 1$ iterations.

If at each stage of the algorithm, we naively find the next edge by checking every edge then there will be $|E(G)|$ steps in each iteration, giving about $|V(G)||E(G)|$ steps in total.

The number of steps of Kruskal's algorithm

In each iteration we add an edge, so there will be $|V(G)| - 1$ iterations.

If at each stage of the algorithm, we naively find the next edge by checking every edge then there will be $|E(G)|$ steps in each iteration, giving about $|V(G)||E(G)|$ steps in total.

We say that the running time is $O(|V(G)||E(G)|)$, where the 'big O' notation means that there is a constant C so that for any graph G the running time is at most $C|V(G)||E(G)|$.

The number of steps of Kruskal's algorithm

In each iteration we add an edge, so there will be $|V(G)| - 1$ iterations.

If at each stage of the algorithm, we naively find the next edge by checking every edge then there will be $|E(G)|$ steps in each iteration, giving about $|V(G)||E(G)|$ steps in total.

We say that the running time is $O(|V(G)||E(G)|)$, where the 'big O' notation means that there is a constant C so that for any graph G the running time is at most $C|V(G)||E(G)|$.

Here 'running time' could be measured in any units, say milliseconds on your favourite computer, as changing the units or using a different computer will just replace C by a different constant.

The number of steps of Kruskal's algorithm

A smarter implementation is to start by making a list of all edges ordered by cost, cheapest first. Then at each step we go through the list from the start, discarding edges that make a cycle until we find the first edge which can be added.

The number of steps of Kruskal's algorithm

A smarter implementation is to start by making a list of all edges ordered by cost, cheapest first. Then at each step we go through the list from the start, discarding edges that make a cycle until we find the first edge which can be added.

This gives a running time that is 'roughly comparable' with the number of edges, which is essentially best possible.

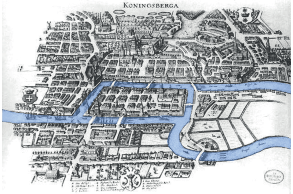
Euler tours

The bridges of Königsberg

The town of Königsberg is divided into 4 districts by the river Pregel.

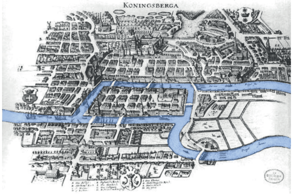
The bridges of Königsberg

The town of Königsberg is divided into 4 districts by the river Pregel.



The bridges of Königsberg

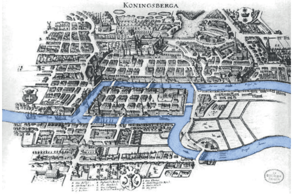
The town of Königsberg is divided into 4 districts by the river Pregel.



In the 18th century, the river was spanned by 7 bridges.

The bridges of Königsberg

The town of Königsberg is divided into 4 districts by the river Pregel.



In the 18th century, the river was spanned by 7 bridges.

Is it possible to take a walk that crosses every bridge exactly once?

The bridges of Königsberg

The town of Königsberg is divided into 4 districts by the river Pregel.

In the 18th century, the river was spanned by 7 bridges.

Is it possible to take a walk that crosses every bridge exactly once?

The bridges of Königsberg

The town of Königsberg is divided into 4 districts by the river Pregel.

In the 18th century, the river was spanned by 7 bridges.

Is it possible to take a walk that crosses every bridge exactly once?

Let W be a walk in a graph G. We call W an *Euler trail* if every edge of G appears exactly once in W.

Euler tours

Euler tours

The problem was solved by Leonard Euler in 1766.

Euler tours

The problem was solved by Leonard Euler in 1766.

Let W be an Euler trail. We call W an *Euler tour* if it is closed, i.e. it starts and ends at the same vertex.

Euler tours

The problem was solved by Leonard Euler in 1766.

Let W be an Euler trail. We call W an *Euler tour* if it is closed, i.e. it starts and ends at the same vertex.

Here we will only solve the problem of finding an Euler tour; the solution of the Euler trail problem can be deduced (see exercise sheet 1).

Eulerian graphs

What can we say about a graph G with an Euler tour W?

Eulerian graphs

What can we say about a graph G with an Euler tour W?

Clearly, G must be connected after we delete all *isolated* vertices (i.e. vertices of degree zero).

Eulerian graphs

What can we say about a graph G with an Euler tour W?

Clearly, G must be connected after we delete all *isolated* vertices (i.e. vertices of degree zero).

Next we note that each visit of W to a vertex v uses two edges at v (one to arrive and one to leave). This is also true of the start and end vertex of W if we consider them to be a single visit.

Eulerian graphs

What can we say about a graph G with an Euler tour W?

Clearly, G must be connected after we delete all *isolated* vertices (i.e. vertices of degree zero).

Next we note that each visit of W to a vertex v uses two edges at v (one to arrive and one to leave). This is also true of the start and end vertex of W if we consider them to be a single visit.

As every edge is used exactly once, we deduce that every vertex has even degree; we call a graph with this property *Eulerian*.

Eulerian graphs

What can we say about a graph G with an Euler tour W?

Clearly, G must be connected after we delete all *isolated* vertices (i.e. vertices of degree zero).

Next we note that each visit of W to a vertex v uses two edges at v (one to arrive and one to leave). This is also true of the start and end vertex of W if we consider them to be a single visit.

As every edge is used exactly once, we deduce that every vertex has even degree; we call a graph with this property *Eulerian*.

is not Eulerian.

Euler's theorem

These necessary conditions are also sufficient:

Euler's theorem

These necessary conditions are also sufficient:

Theorem 9. (Euler) Let G be a connected Eulerian graph. Then G has an Euler tour.

Euler's theorem

These necessary conditions are also sufficient:

Theorem 9. (Euler) Let G be a connected Eulerian graph. Then G has an Euler tour.

In fact, we will show that we can find an Euler tour efficiently, using the following algorithm.

Fleury's Algorithm.

Start at any vertex of G.

Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again).

Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

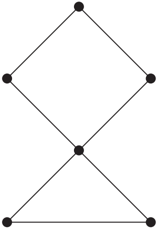
We will show that when G is Eulerian, this produces an Euler tour.

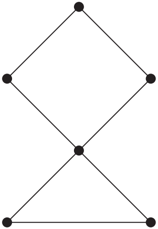
Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.



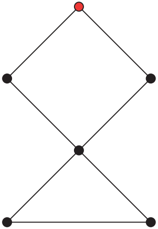


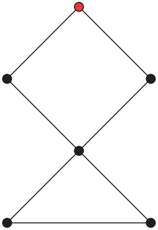
Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.



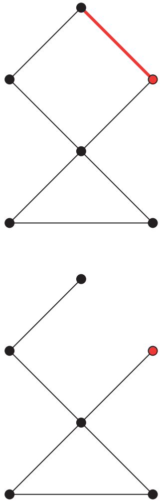


Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.

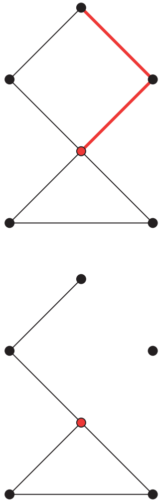


Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.

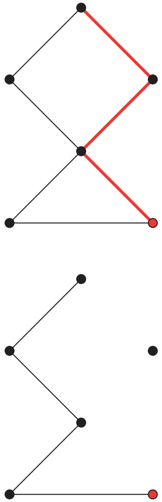


Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.

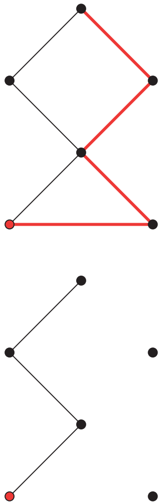


Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.

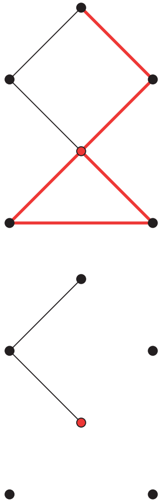


Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.

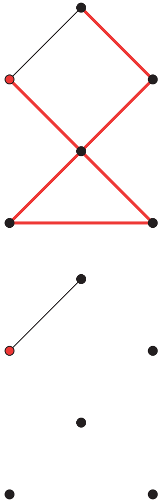


Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.

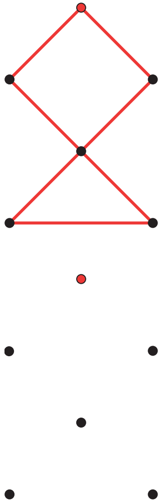


Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.

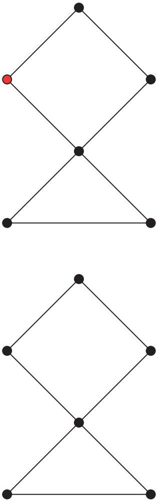


Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.

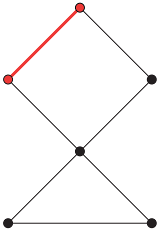


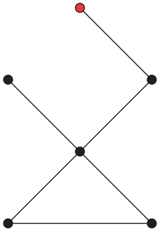
Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.



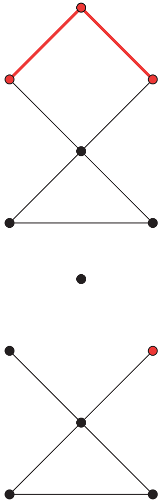


Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.

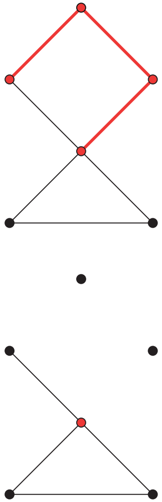


Fleury's Algorithm.

Start at any vertex of G. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

1. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
2. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We will show that when G is Eulerian, this produces an Euler tour.



Vertices of odd degree

We require a useful lemma.

Vertices of odd degree

We require a useful lemma.

Lemma 10. In any graph, there are an even number of vertices with odd degree.

Vertices of odd degree

We require a useful lemma.

Lemma 10. In any graph, there are an even number of vertices with odd degree.

Proof. Since every edge has two endpoints,

$$\sum_{v \in V(G)} d(v) = 2|E(G|.$$

Vertices of odd degree

We require a useful lemma.

Lemma 10. In any graph, there are an even number of vertices with odd degree.

Proof. Since every edge has two endpoints,

$$\sum_{v \in V(G)} d(v) = 2|E(G)|.$$

Therefore, in the sum, there must be an even number of occurrences of $d(v)$ for which $d(v)$ is odd. $\square$

Fleury's algorithm produces an Euler tour

Note first that at each stage of the algorithm, either there are two vertices of odd degree (the initial vertex u and the current one) or there are no vertices of odd degree.

Fleury's algorithm produces an Euler tour

Note first that at each stage of the algorithm, either there are two vertices of odd degree (the initial vertex u and the current one) or there are no vertices of odd degree.

Suppose for a contradiction Fleury's Algorithm fails. Say it stops at some vertex v and can go no further. Let H be the component of the current graph containing v.

Fleury's algorithm produces an Euler tour

Note first that at each stage of the algorithm, either there are two vertices of odd degree (the initial vertex u and the current one) or there are no vertices of odd degree.

Suppose for a contradiction Fleury's Algorithm fails. Say it stops at some vertex v and can go no further. Let H be the component of the current graph containing v.

The degree of v in H must be positive, as otherwise in the previous step, we ran along an edge to a leaf violating (2).

Fleury's algorithm produces an Euler tour

Note first that at each stage of the algorithm, either there are two vertices of odd degree (the initial vertex u and the current one) or there are no vertices of odd degree.

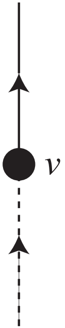
Suppose for a contradiction Fleury's Algorithm fails. Say it stops at some vertex v and can go no further. Let H be the component of the current graph containing v.

The degree of v in H must be positive, as otherwise in the previous step, we ran along an edge to a leaf violating (2).

Fleury's algorithm produces an Euler tour

Note first that at each stage of the algorithm, either there are two vertices of odd degree (the initial vertex u and the current one) or there are no vertices of odd degree.

Suppose for a contradiction Fleury's Algorithm fails. Say it stops at some vertex v and can go no further. Let H be the component of the current graph containing v.

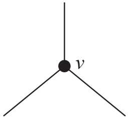


The degree of v in H must be positive, as otherwise in the previous step, we ran along an edge to a leaf violating (2).

If the degree of v in H is one, then we can continue the walk.

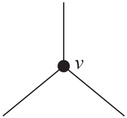
Fleury's algorithm produces an Euler tour

So there are at least two edges of H containing v.



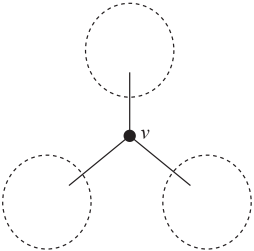
Fleury's algorithm produces an Euler tour

So there are at least two edges of H containing v.
Since the algorithm cannot continue, the graph $H - e$ is disconnected for each edge e containing v.



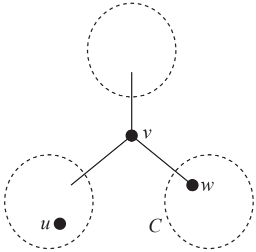
Fleury's algorithm produces an Euler tour

So there are at least two edges of H containing v.
Since the algorithm cannot continue, the graph $H - e$ is disconnected for each edge e containing v.
Hence, the edges e incident to v all have endpoints in distinct components of $H - v$.



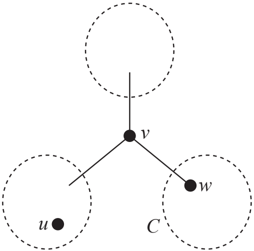
Fleury's algorithm produces an Euler tour

So there are at least two edges of H containing v.
Since the algorithm cannot continue, the graph $H - e$ is disconnected for each edge e containing v.
Hence, the edges e incident to v all have endpoints in distinct components of $H - v$.
So, we can choose one edge vw, such that the component C of $G - vw$ which contains w does not contain the first vertex u of the walk.



Fleury's algorithm produces an Euler tour

So there are at least two edges of H containing v.
Since the algorithm cannot continue, the graph $H - e$ is disconnected for each edge e containing v.
Hence, the edges e incident to v all have endpoints in distinct components of $H - v$.
So, we can choose one edge vw, such that the component C of $G - vw$ which contains w does not contain the first vertex u of the walk.
But then w is the only vertex of odd degree in C, which is impossible by Lemma 10. □



Hamiltonian cycles

Hamiltonian cycles

One can ask the following about a connected graph G:

Hamiltonian cycles

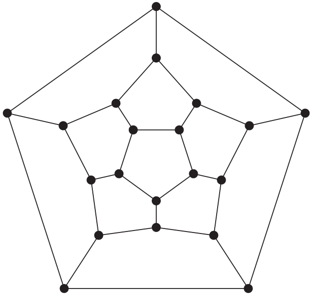
One can ask the following about a connected graph G:

Does there exists a closed walk
that visits every vertex exactly once?

Hamiltonian cycles

One can ask the following about a connected graph G:

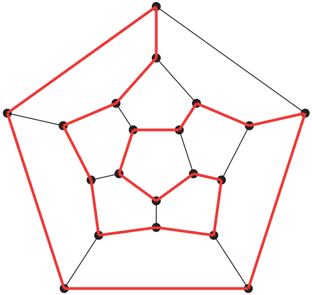
Does there exists a closed walk
that visits every vertex exactly once?



Hamiltonian cycles

One can ask the following about a connected graph G:

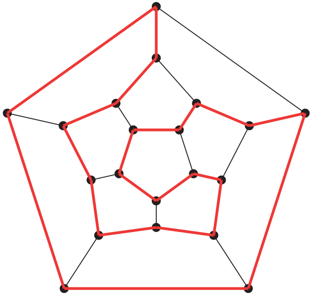
Does there exists a closed walk
that visits every vertex exactly once?



Hamiltonian cycles

One can ask the following about a connected graph G:

Does there exists a closed walk
that visits every vertex exactly once?

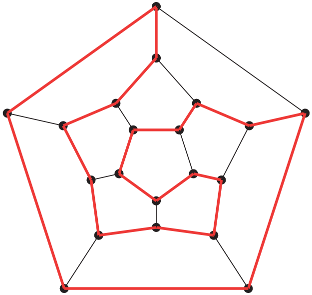


In fact, such a walk is a cycle (provided G has more than two vertices) and is known as a *Hamiltonian cycle*.

Hamiltonian cycles

One can ask the following about a connected graph G:

Does there exists a closed walk
that visits every vertex exactly once?



In fact, such a walk is a cycle (provided G has more than two vertices) and is known as a *Hamiltonian cycle*.

When a graph G contains such a cycle, it is *Hamiltonian*.

Hamiltonian cycles

Unlike the case of Eulerian tours, it turns out that there is, almost certainly, no efficient algorithm to determine whether a general graph G is Hamiltonian.

Hamiltonian cycles

Unlike the case of Eulerian tours, it turns out that there is, almost certainly, no efficient algorithm to determine whether a general graph G is Hamiltonian.

By 'efficient', we mean that the algorithm gives the answer after polynomially many 'steps', as a function of $|V(G)|$ and $|E(G)|$.

Hamiltonian cycles

Unlike the case of Eulerian tours, it turns out that there is, almost certainly, no efficient algorithm to determine whether a general graph G is Hamiltonian.

By 'efficient', we mean that the algorithm gives the answer after polynomially many 'steps', as a function of $|V(G)|$ and $|E(G)|$.

But what do we mean by 'almost certainly'?

Hamiltonian cycles

Unlike the case of Eulerian tours, it turns out that there is, almost certainly, no efficient algorithm to determine whether a general graph G is Hamiltonian.

By 'efficient', we mean that the algorithm gives the answer after polynomially many 'steps', as a function of $|V(G)|$ and $|E(G)|$.

But what do we mean by 'almost certainly'?

Currently, mathematicians do not have a proof that there is no efficient algorithm to determine whether a general graph G is Hamiltonian.

Hamiltonian cycles

Unlike the case of Eulerian tours, it turns out that there is, almost certainly, no efficient algorithm to determine whether a general graph G is Hamiltonian.

By 'efficient', we mean that the algorithm gives the answer after polynomially many 'steps', as a function of $|V(G)|$ and $|E(G)|$.

But what do we mean by 'almost certainly'?

Currently, mathematicians do not have a proof that there is no efficient algorithm to determine whether a general graph G is Hamiltonian.

However, we do no know that there is no efficient algorithm if we assume the famous conjecture $P \neq NP$.

Hamiltonian cycles

Unlike the case of Eulerian tours, it turns out that there is, almost certainly, no efficient algorithm to determine whether a general graph G is Hamiltonian.

By 'efficient', we mean that the algorithm gives the answer after polynomially many 'steps', as a function of $|V(G)|$ and $|E(G)|$.

But what do we mean by 'almost certainly'?

Currently, mathematicians do not have a proof that there is no efficient algorithm to determine whether a general graph G is Hamiltonian.

However, we do no know that there is no efficient algorithm if we assume the famous conjecture $P \neq NP$.

But to discuss this conjecture would take us too far afield.

A sufficient condition

We will therefore content ourselves with a sufficient condition for a graph to be Hamiltonian.

A sufficient condition

We will therefore content ourselves with a sufficient condition for a graph to be Hamiltonian.

Theorem 11. Let G be a connected graph with n vertices. Suppose that for every pair of non-adjacent vertices x and y,

$$d(x) + d(y) \geq n.$$

Then G is Hamiltonian.

A sufficient condition

We will therefore content ourselves with a sufficient condition for a graph to be Hamiltonian.

Theorem 11. Let G be a connected graph with n vertices. Suppose that for every pair of non-adjacent vertices x and y,

$$d(x) + d(y) \geq n.$$

Then G is Hamiltonian.

Corollary 12. If G is connected with n vertices and for every vertex v, $d(v) \geq n/2$, then G is Hamiltonian.

The length of paths and cycles

We first note if G is Hamiltonian and has n vertices, then the length of the longest cycle is n and the length of the longest path is $n-1$.

The length of paths and cycles

We first note if G is Hamiltonian and has n vertices, then the length of the longest cycle is n and the length of the longest path is $n - 1$. (The *length* of a path is its number of edges.)

The length of paths and cycles

We first note if G is Hamiltonian and has n vertices, then the length of the longest cycle is n and the length of the longest path is $n - 1$. (The *length* of a path is its number of edges.)

Lemma 13. If G is connected and non-Hamiltonian, then the length of the longest path is least the length of the longest cycle.

The length of paths and cycles

We first note if G is Hamiltonian and has n vertices, then the length of the longest cycle is n and the length of the longest path is $n - 1$. (The *length* of a path is its number of edges.)

Lemma 13. If G is connected and non-Hamiltonian, then the length of the longest path is least the length of the longest cycle.

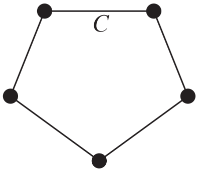
Proof.

The length of paths and cycles

We first note if G is Hamiltonian and has n vertices, then the length of the longest cycle is n and the length of the longest path is $n - 1$. (The *length* of a path is its number of edges.)

Lemma 13. If G is connected and non-Hamiltonian, then the length of the longest path is least the length of the longest cycle.

Proof. Let C be a longest cycle, with length ℓ.

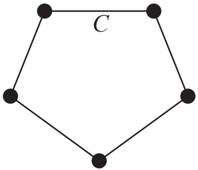


The length of paths and cycles

We first note if G is Hamiltonian and has n vertices, then the length of the longest cycle is n and the length of the longest path is $n-1$. (The *length* of a path is its number of edges.)

Lemma 13. If G is connected and non-Hamiltonian, then the length of the longest path is least the length of the longest cycle.

Proof. Let C be a longest cycle, with length ℓ. Since G is non-Hamiltonian, there is some vertex not in C.

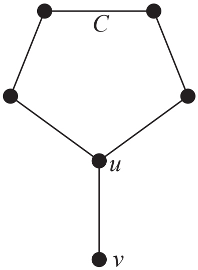


The length of paths and cycles

We first note if G is Hamiltonian and has n vertices, then the length of the longest cycle is n and the length of the longest path is $n-1$. (The *length* of a path is its number of edges.)

Lemma 13. If G is connected and non-Hamiltonian, then the length of the longest path is least the length of the longest cycle.

Proof. Let C be a longest cycle, with length ℓ. Since G is non-Hamiltonian, there is some vertex not in C. Since G is connected, there is therefore some edge uv with one endpoint u in C and one endpoint v not in C.

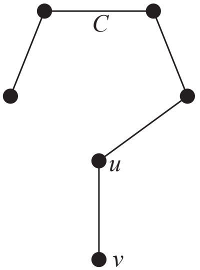


The length of paths and cycles

We first note if G is Hamiltonian and has n vertices, then the length of the longest cycle is n and the length of the longest path is $n - 1$. (The *length* of a path is its number of edges.)

Lemma 13. If G is connected and non-Hamiltonian, then the length of the longest path is least the length of the longest cycle.

Proof. Let C be a longest cycle, with length ℓ. Since G is non-Hamiltonian, there is some vertex not in C. Since G is connected, there is therefore some edge uv with one endpoint u in C and one endpoint v not in C. Removing an edge of C incident to u and adding uv gives a path of length ℓ. □



Proof of Theorem.

Theorem 11. Let G be a connected graph with n vertices. Suppose that for every pair of non-adjacent vertices x and y,

$$d(x) + d(y) \geq n.$$

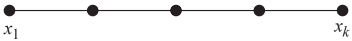
Then G is Hamiltonian.

Proof of Theorem.

Proof. Suppose that G is not Hamiltonian.

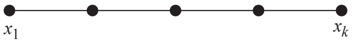
Proof of Theorem.

Proof. Suppose that G is not Hamiltonian.
Let $P = x_1 \cdots x_k$ be a longest path. It has length $k-1$.



Proof of Theorem.

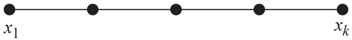
Proof. Suppose that G is not Hamiltonian.
Let $P = x_1 \cdots x_k$ be a longest path. It has length $k-1$.



So by Lemma 13, G does not have a cycle of length k.

Proof of Theorem.

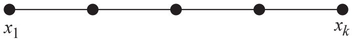
Proof. Suppose that G is not Hamiltonian.
Let $P = x_1 \cdots x_k$ be a longest path. It has length $k-1$.



So by Lemma 13, G does not have a cycle of length k. So x_1 and x_k are not adjacent.

Proof of Theorem.

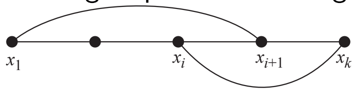
Proof. Suppose that G is not Hamiltonian.
Let $P = x_1 \cdots x_k$ be a longest path. It has length $k-1$.



So by Lemma 13, G does not have a cycle of length k. So x_1 and x_k are not adjacent. Hence, by our assumption, $d(x_1) + d(x_k) \geq n$.

Proof of Theorem.

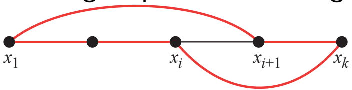
Proof. Suppose that G is not Hamiltonian.
Let $P = x_1 \cdots x_k$ be a longest path. It has length $k-1$.



So by Lemma 13, G does not have a cycle of length k. So x_1 and x_k are not adjacent. Hence, by our assumption, $d(x_1) + d(x_k) \geq n$. There is no integer i such that x_1 is adjacent to x_{i+1} and x_k is adjacent x_i.

Proof of Theorem.

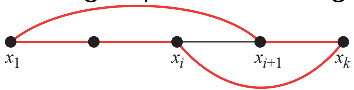
Proof. Suppose that G is not Hamiltonian.
Let $P = x_1 \cdots x_k$ be a longest path. It has length $k - 1$.



So by Lemma 13, G does not have a cycle of length k. So x_1 and x_k are not adjacent. Hence, by our assumption, $d(x_1) + d(x_k) \geq n$. There is no integer i such that x_1 is adjacent to x_{i+1} and x_k is adjacent x_i. Otherwise, $x_1 \cdots x_i x_k x_{k-1} \cdots x_{i+1} x_1$ would be a cycle of length k.

Proof of Theorem.

Proof. Suppose that G is not Hamiltonian.
Let $P = x_1 \cdots x_k$ be a longest path. It has length $k-1$.



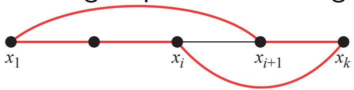
So by Lemma 13, G does not have a cycle of length k. So x_1 and x_k are not adjacent. Hence, by our assumption, $d(x_1) + d(x_k) \geq n$. There is no integer i such that x_1 is adjacent to x_{i+1} and x_k is adjacent x_i. Otherwise, $x_1 \cdots x_i x_k x_{k-1} \cdots x_{i+1} x_1$ would be a cycle of length k. So the sets

$$A = \{i : x_1 x_{i+1} \in E(G)\}, \qquad B = \{i : x_i x_k \in E(G)\}$$

are disjoint subsets of $\{1, \cdots, k-1\}$.

Proof of Theorem.

Proof. Suppose that G is not Hamiltonian.
Let $P = x_1 \cdots x_k$ be a longest path. It has length $k-1$.



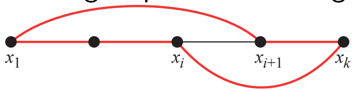
So by Lemma 13, G does not have a cycle of length k. So x_1 and x_k are not adjacent. Hence, by our assumption, $d(x_1) + d(x_k) \geq n$. There is no integer i such that x_1 is adjacent to x_{i+1} and x_k is adjacent x_i. Otherwise, $x_1 \cdots x_i x_k x_{k-1} \cdots x_{i+1} x_1$ would be a cycle of length k. So the sets

$$A = \{i : x_1 x_{i+1} \in E(G)\}, \qquad B = \{i : x_i x_k \in E(G)\}$$

are disjoint subsets of $\{1, \cdots, k-1\}$.
Every neighbour of x_1 lies in P, and similarly every neighbour of x_k lies in P, as P is a longest path.

Proof of Theorem.

Proof. Suppose that G is not Hamiltonian.
Let $P = x_1 \cdots x_k$ be a longest path. It has length $k-1$.



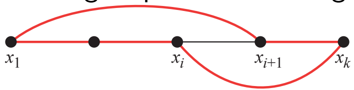
So by Lemma 13, G does not have a cycle of length k. So x_1 and x_k are not adjacent. Hence, by our assumption, $d(x_1) + d(x_k) \geq n$. There is no integer i such that x_1 is adjacent to x_{i+1} and x_k is adjacent x_i. Otherwise, $x_1 \cdots x_i x_k x_{k-1} \cdots x_{i+1} x_1$ would be a cycle of length k. So the sets

$$A = \{i : x_1 x_{i+1} \in E(G)\}, \qquad B = \{i : x_i x_k \in E(G)\}$$

are disjoint subsets of $\{1, \cdots, k-1\}$.
Every neighbour of x_1 lies in P, and similarly every neighbour of x_k lies in P, as P is a longest path. So, A has size $d(x_1)$, and B has size $d(x_k)$.

Proof of Theorem.

Proof. Suppose that G is not Hamiltonian.
Let $P = x_1 \cdots x_k$ be a longest path. It has length $k-1$.



So by Lemma 13, G does not have a cycle of length k. So x_1 and x_k are not adjacent. Hence, by our assumption, $d(x_1) + d(x_k) \geq n$. There is no integer i such that x_1 is adjacent to x_{i+1} and x_k is adjacent x_i. Otherwise, $x_1 \cdots x_i x_k x_{k-1} \cdots x_{i+1} x_1$ would be a cycle of length k. So the sets

$$A = \{i : x_1 x_{i+1} \in E(G)\}, \qquad B = \{i : x_i x_k \in E(G)\}$$

are disjoint subsets of $\{1, \cdots, k-1\}$.
Every neighbour of x_1 lies in P, and similarly every neighbour of x_k lies in P, as P is a longest path. So, A has size $d(x_1)$, and B has size $d(x_k)$. Since A and B are disjoint, $d(x_1) + d(x_k) \leq k - 1 < n$, which is a contradiction. Hence, G must be Hamiltonian. □