Part A Graph Theory

Marc Lackenby

Trinity Term 2022

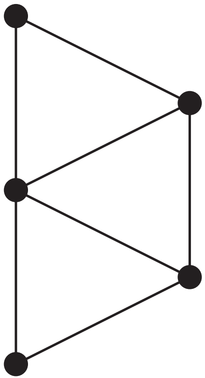
Shortest paths

Shortest Paths

Let G be a connected graph.

Shortest Paths

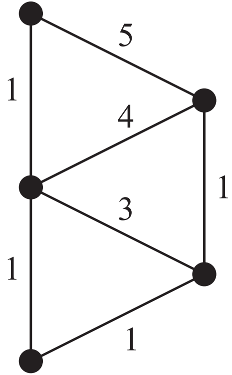
Let G be a connected graph.



Shortest Paths

Let G be a connected graph.

Let $\ell(e) > 0$ for $e \in E(G)$ be the 'length' of the edge e.

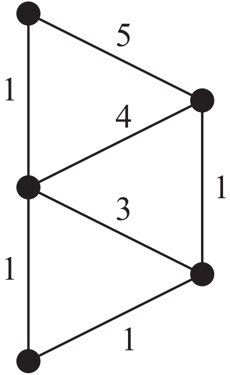


Shortest Paths

Let G be a connected graph.

Let $\ell(e) > 0$ for $e \in E(G)$ be the 'length' of the edge e.

The *ℓ-length* of a path P is $\ell(P) = \sum_{e \in E(P)} \ell(e)$.



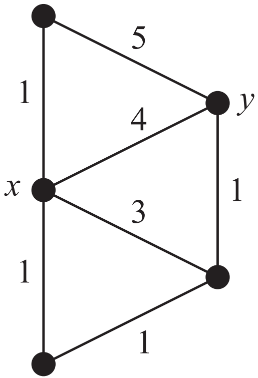
Shortest Paths

Let G be a connected graph.

Let $\ell(e) > 0$ for $e \in E(G)$ be the 'length' of the edge e.

The *ℓ-length* of a path P is $\ell(P) = \sum_{e \in E(P)} \ell(e)$.

Given x and y in $V(G)$, an *ℓ-shortest xy-path* is an xy-path P that minimises $\ell(P)$.



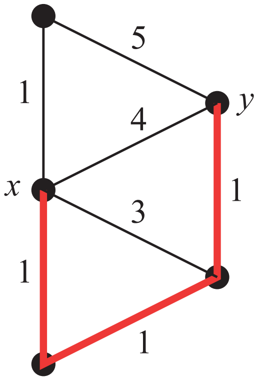
Shortest Paths

Let G be a connected graph.

Let $\ell(e) > 0$ for $e \in E(G)$ be the 'length' of the edge e.

The *ℓ-length* of a path P is $\ell(P) = \sum_{e \in E(P)} \ell(e)$.

Given x and y in $V(G)$, an *ℓ-shortest xy-path* is an xy-path P that minimises $\ell(P)$.



Dijkstra's Algorithm

For vertices x and y, this finds an ℓ-shortest xy-path.

Dijkstra's Algorithm

For vertices x and y, this finds an ℓ-shortest xy-path.

The idea of the algorithm is to maintain a 'tentative distance from x' called $D(v)$ for each $v \in V(G)$.

Dijkstra's Algorithm

For vertices x and y, this finds an ℓ-shortest xy-path.

The idea of the algorithm is to maintain a 'tentative distance from x' called $D(v)$ for each $v \in V(G)$.

At each step of the algorithm we finalise $D(u)$ for some vertex u.

Dijkstra's Algorithm

For vertices x and y, this finds an ℓ-shortest xy-path.

The idea of the algorithm is to maintain a 'tentative distance from x' called $D(v)$ for each $v \in V(G)$.

At each step of the algorithm we finalise $D(u)$ for some vertex u.

At the end of the algorithm all $D(u)$ will be equal to the correct value,

Dijkstra's Algorithm

For vertices x and y, this finds an ℓ-shortest xy-path.

The idea of the algorithm is to maintain a 'tentative distance from x' called $D(v)$ for each $v \in V(G)$.

At each step of the algorithm we finalise $D(u)$ for some vertex u.

At the end of the algorithm all $D(u)$ will be equal to the correct value, i.e. $D(u) = \ell(P_u^*)$ for some ℓ-shortest xu-path P_u^*.

Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]

Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0$,

Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

Repeat the following step:

Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U,

Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.

Dijkstra's Algorithm

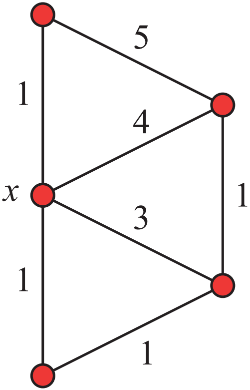
Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.

Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

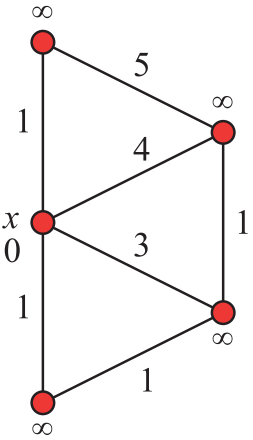
Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.



Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

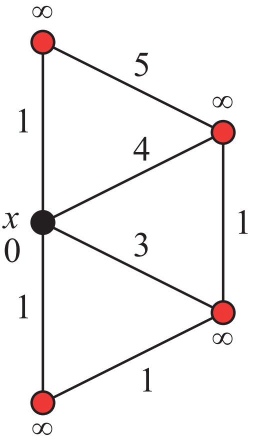
Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.



Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

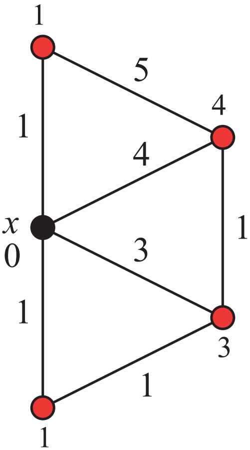
Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.



Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

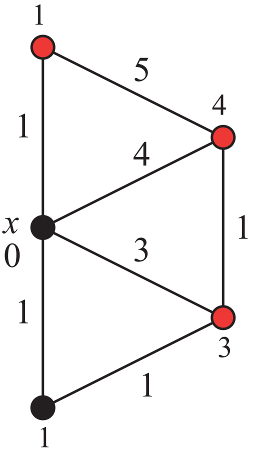
Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.



Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

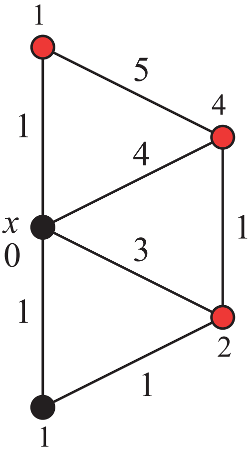
Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.



Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

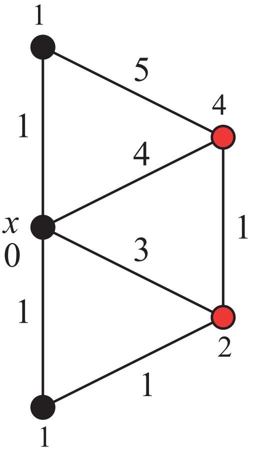
Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.



Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

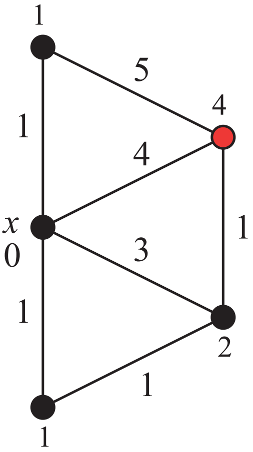
Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.



Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

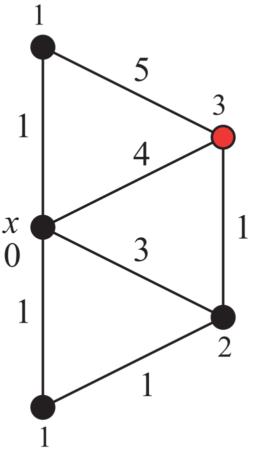
Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.



Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

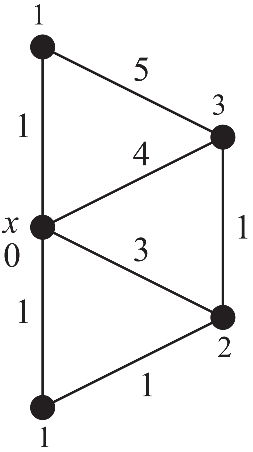
Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.



Dijkstra's Algorithm

Start by letting $U = V(G)$,
[U is the set of vertices v for which $D(v)$ has not yet been finalised]
$D(x) = 0, \quad D(v) = \infty$ for all $v \neq x$.

Repeat the following step:
If $U = \emptyset$ stop. Otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.



Shortest paths rooted trees

Dijkstra's Algorithm can be used to do more:

Shortest paths rooted trees

Dijkstra's Algorithm can be used to do more:

For any $x \in V(G)$ we can construct a spanning tree T such that for any $y \in V(G)$, the unique xy-path in T is an ℓ-shortest xy-path.

Shortest paths rooted trees

Dijkstra's Algorithm can be used to do more:

For any $x \in V(G)$ we can construct a spanning tree T such that for any $y \in V(G)$, the unique xy-path in T is an ℓ-shortest xy-path.

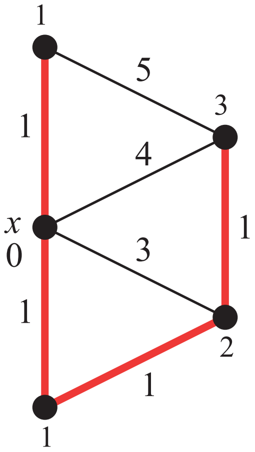
We call T an *ℓ-shortest paths tree rooted at x*.

Shortest paths rooted trees

Dijkstra's Algorithm can be used to do more:

For any $x \in V(G)$ we can construct a spanning tree T such that for any $y \in V(G)$, the unique xy-path in T is an ℓ-shortest xy-path.

We call T an *ℓ-shortest paths tree rooted at x*.



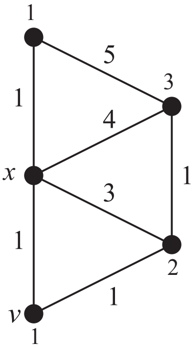
Shortest paths rooted trees

We now describe how to obtain T.

Shortest paths rooted trees

We now describe how to obtain T.

For any vertex $v \neq x$, the *parent* of v is the last vertex u such that we replaced $D(v)$ by $D(u) + \ell(uv)$ during the algorithm.

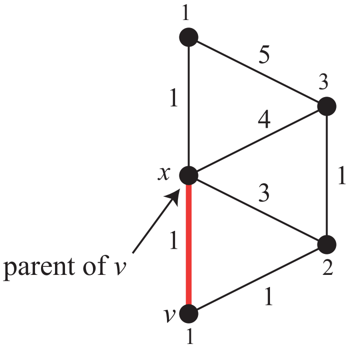


Shortest paths rooted trees

We now describe how to obtain T.

For any vertex $v \neq x$, the *parent* of v is the last vertex u such that we replaced $D(v)$ by $D(u) + \ell(uv)$ during the algorithm.

We obtain T by drawing an edge from each vertex $v \neq x$ to the parent of v.

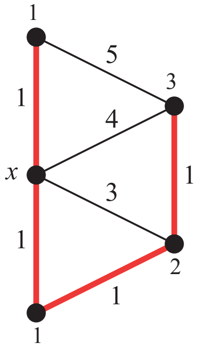


Shortest paths rooted trees

We now describe how to obtain T.

For any vertex $v \neq x$, the *parent* of v is the last vertex u such that we replaced $D(v)$ by $D(u) + \ell(uv)$ during the algorithm.

We obtain T by drawing an edge from each vertex $v \neq x$ to the parent of v.



Start of the proof

Lemma 14. T is a tree,

Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

Proof.

Start of the proof

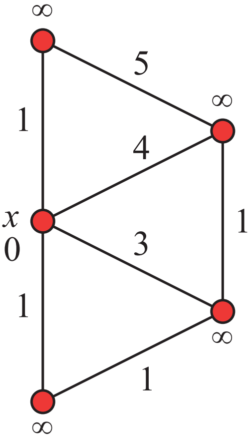
Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$.

Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

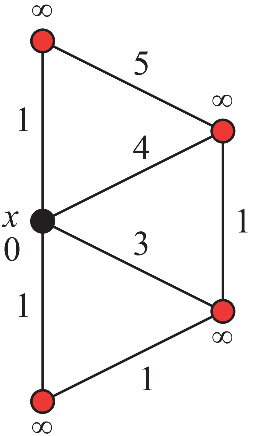
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

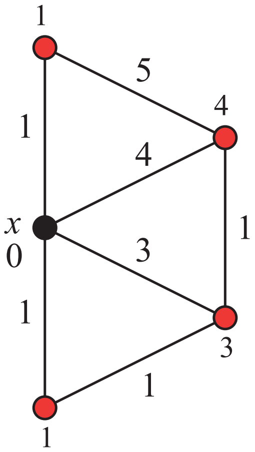
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

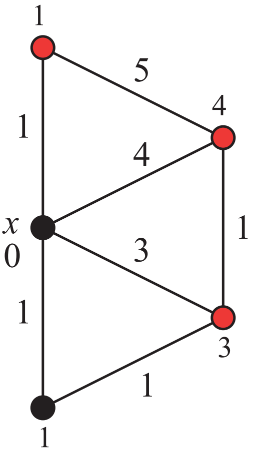
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

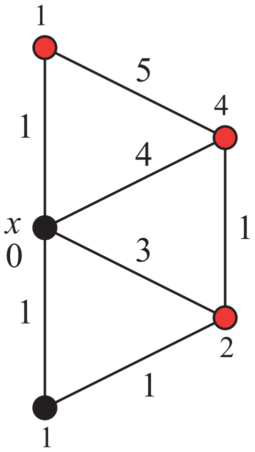
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

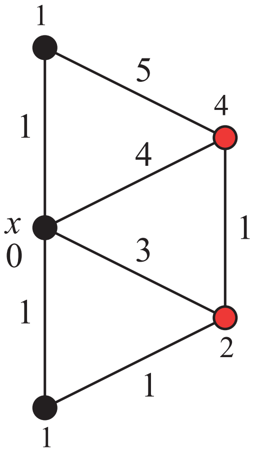
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

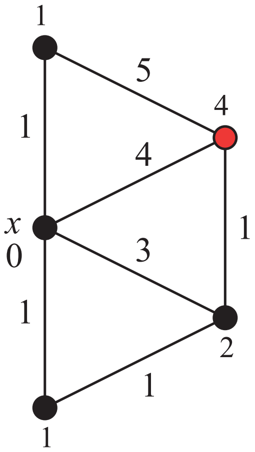
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

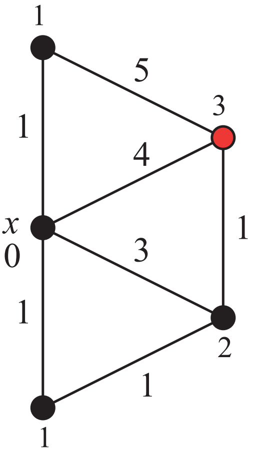
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

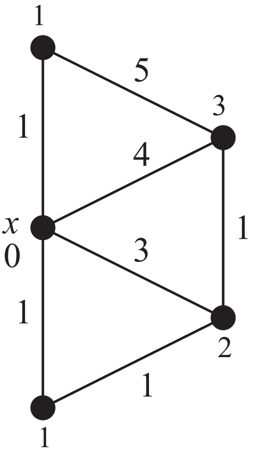
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

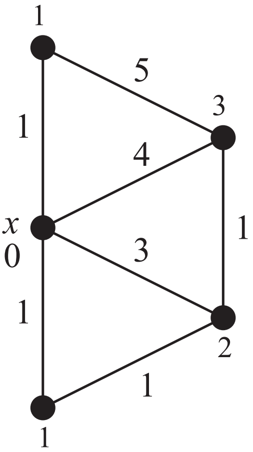
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

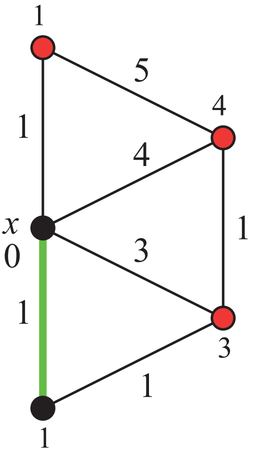
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$. Let T_C be obtained by drawing an edge from each $v \in C \setminus \{x\}$ to its parent. So $V(T_C) = C$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

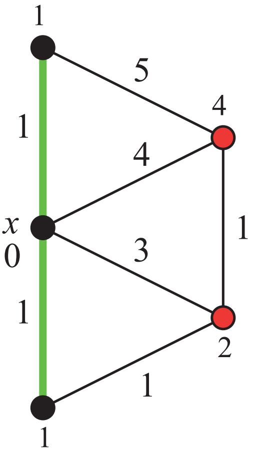
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$. Let T_C be obtained by drawing an edge from each $v \in C \setminus \{x\}$ to its parent. So $V(T_C) = C$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

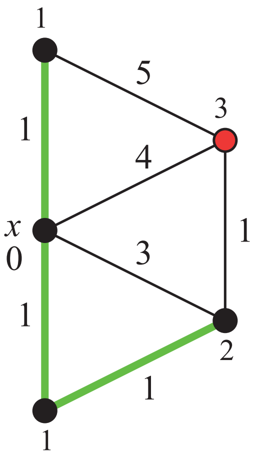
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$. Let T_C be obtained by drawing an edge from each $v \in C \setminus \{x\}$ to its parent. So $V(T_C) = C$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

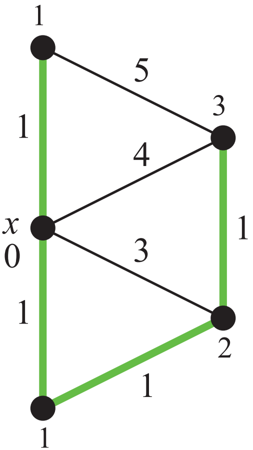
Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$. Let T_C be obtained by drawing an edge from each $v \in C \setminus \{x\}$ to its parent. So $V(T_C) = C$.



Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$. Let T_C be obtained by drawing an edge from each $v \in C \setminus \{x\}$ to its parent. So $V(T_C) = C$.

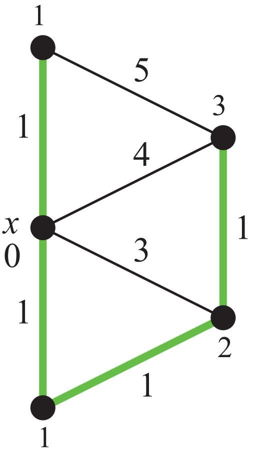


Start of the proof

Lemma 14. T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.

Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$. Let T_C be obtained by drawing an edge from each $v \in C \setminus \{x\}$ to its parent. So $V(T_C) = C$.

We show by induction on $|C|$ that T_C is a tree and for each $u \in V(T_C)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T_C.



Proof

We show by induction on $|C|$ that T_C is a tree and for each $u \in V(T_C)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T_C.

Proof

We show by induction on $|C|$ that T_C is a tree and for each $u \in V(T_C)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T_C.

Base case:

Proof

We show by induction on $|C|$ that T_C is a tree and for each $u \in V(T_C)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T_C.

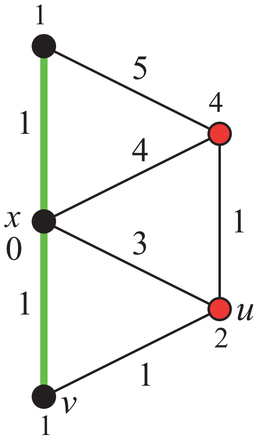
Base case: we start with $V(T_C) = \{x\}$ and no edges, which is a tree, with $D(x) = 0 = \ell(P_x)$.

Proof

We show by induction on $|C|$ that T_C is a tree and for each $u \in V(T_C)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T_C.

Base case: we start with $V(T_C) = \{x\}$ and no edges, which is a tree, with $D(x) = 0 = \ell(P_x)$.

Induction step:

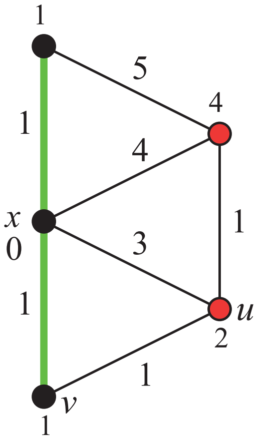


Proof

We show by induction on $|C|$ that T_C is a tree and for each $u \in V(T_C)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T_C.

Base case: we start with $V(T_C) = \{x\}$ and no edges, which is a tree, with $D(x) = 0 = \ell(P_x)$.

Induction step: When we delete u from U, we add u to C, and add an edge from u to the parent v of u, i.e. we add a leaf to T_C, and so obtain another tree.



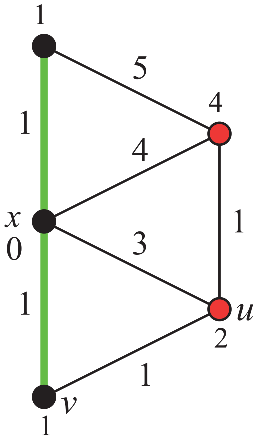
Proof

We show by induction on $|C|$ that T_C is a tree and for each $u \in V(T_C)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T_C.

Base case: we start with $V(T_C) = \{x\}$ and no edges, which is a tree, with $D(x) = 0 = \ell(P_x)$.

Induction step: When we delete u from U, we add u to C, and add an edge from u to the parent v of u, i.e. we add a leaf to T_C, and so obtain another tree.

By definition of parent and induction we have $D(u) = D(v) + \ell(vu) = \ell(P_v) + \ell(vu) = \ell(P_u)$. □



Completion of the proof

Theorem 15. T is an ℓ-shortest paths tree rooted at x.

Completion of the proof

Theorem 15. T is an ℓ-shortest paths tree rooted at x.

Proof. For each $u \in V(G)$ let $D^*(u) = \ell(P_u^*)$ for some ℓ-shortest xu-path P_u^*.

Completion of the proof

Theorem 15. T is an ℓ-shortest paths tree rooted at x.

Proof. For each $u \in V(G)$ let $D^*(u) = \ell(P_u^*)$ for some ℓ-shortest xu-path P_u^*.

We show by induction that in each step of the algorithm, when u is deleted we have $D(u) = D^*(u)$.

Completion of the proof

Theorem 15. T is an ℓ-shortest paths tree rooted at x.

Proof. For each $u \in V(G)$ let $D^*(u) = \ell(P^*_u)$ for some ℓ-shortest xu-path P^*_u.

We show by induction that in each step of the algorithm, when u is deleted we have $D(u) = D^*(u)$.

Base case.

Completion of the proof

Theorem 15. T is an ℓ-shortest paths tree rooted at x.

Proof. For each $u \in V(G)$ let $D^*(u) = \ell(P_u^*)$ for some ℓ-shortest xu-path P_u^*.

We show by induction that in each step of the algorithm, when u is deleted we have $D(u) = D^*(u)$.

Base case. We have $u = x$ and $D(u) = D^*(u) = 0$.

Completion of the proof

Theorem 15. T is an ℓ-shortest paths tree rooted at x.

Proof. For each $u \in V(G)$ let $D^*(u) = \ell(P_u^*)$ for some ℓ-shortest xu-path P_u^*.

We show by induction that in each step of the algorithm, when u is deleted we have $D(u) = D^*(u)$.

Base case. We have $u = x$ and $D(u) = D^*(u) = 0$.

Induction step. Consider the step where we delete some u from U, and suppose for contradiction that $D(u) > D^*(u)$.

Completion of the proof

Theorem 15. T is an ℓ-shortest paths tree rooted at x.

Proof. For each $u \in V(G)$ let $D^*(u) = \ell(P_u^*)$ for some ℓ-shortest xu-path P_u^*.

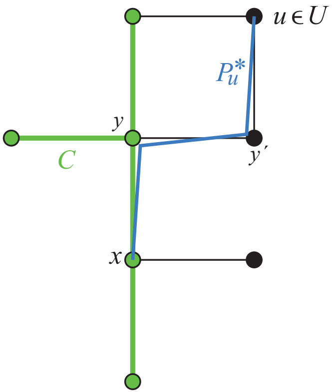
We show by induction that in each step of the algorithm, when u is deleted we have $D(u) = D^*(u)$.

Base case. We have $u = x$ and $D(u) = D^*(u) = 0$.

Induction step. Consider the step where we delete some u from U, and suppose for contradiction that $D(u) > D^*(u)$.
Let $C = V(G) \setminus U$. By induction, for every vertex v in T_C, $D^*(v) = D(v)$.

Completion of the proof

Let yy' be the first edge of P_u^* with $y \notin U$ and $y' \in U$.

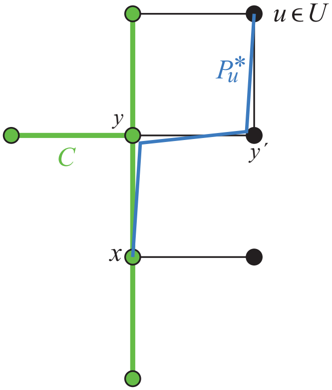


Completion of the proof

Let yy' be the first edge of P_u^* with $y \notin U$ and $y' \in U$.

By induction hypothesis $D(y) = D^*(y)$. Now

$$\begin{aligned} D(y') &\leq D(y) + \ell(yy') \\ &= D^*(y) + \ell(yy') \\ &= \ell(P_y^*) + \ell(yy') \\ &\leq \ell(P_u^*) = D^*(u) < D(u). \end{aligned}$$

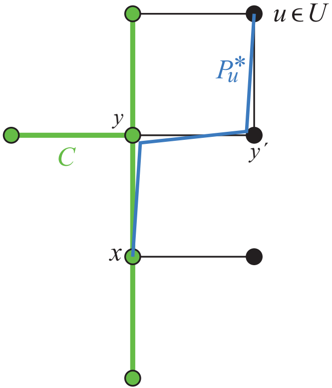


Completion of the proof

Let yy' be the first edge of P_u^* with $y \notin U$ and $y' \in U$.

By induction hypothesis $D(y) = D^*(y)$. Now

$$\begin{aligned} D(y') &\leq D(y) + \ell(yy') \\ &= D^*(y) + \ell(yy') \\ &= \ell(P_y^*) + \ell(yy') \\ &\leq \ell(P_u^*) = D^*(u) < D(u). \end{aligned}$$



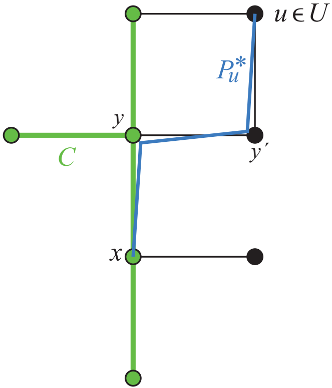
The first inequality uses the update rule for y and y': when y was removed from U, $D(y')$ was replaced by $D(y) + \ell(yy')$ if that was smaller, and so after this, $D(y') \leq D(y) + \ell(yy')$.

Completion of the proof

Let yy' be the first edge of P_u^* with $y \notin U$ and $y' \in U$.

By induction hypothesis $D(y) = D^*(y)$. Now

$$\begin{aligned} D(y') &\leq D(y) + \ell(yy') \\ &= D^*(y) + \ell(yy') \\ &= \ell(P_y^*) + \ell(yy') \\ &\leq \ell(P_u^*) = D^*(u) < D(u). \end{aligned}$$



The first inequality uses the update rule for y and y': when y was removed from U, $D(y')$ was replaced by $D(y) + \ell(yy')$ if that was smaller, and so after this, $D(y') \leq D(y) + \ell(yy')$.

However, $y' \in U$ with $D(y') < D(u)$ contradicts the choice of u in the algorithm. So $D(u) = D^*(u)$. $\square$

Running time

The running time of this implementation of Dijkstra's Algorithm is $O(|V(G)||E(G)|)$.

Running time

The running time of this implementation of Dijkstra's Algorithm is $O(|V(G)||E(G)|)$.

A better implementation (which we omit) gives a running time of $O(|E(G)| + |V(G)| \log |V(G)|)$.

Matchings

The marriage problem

The Marriage Problem:

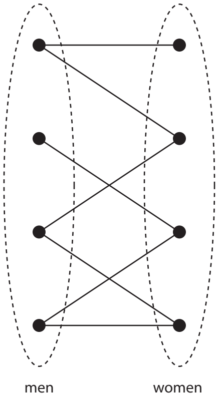
The marriage problem

The Marriage Problem:

Given n men and n women, under what conditions is it possible to pair each man with a woman such that every pair know each other?

The marriage problem

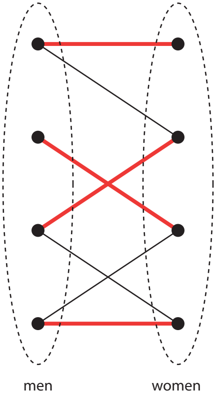
The Marriage Problem:
Given n men and n women, under what conditions is it possible to pair each man with a woman such that every pair know each other?



The marriage problem

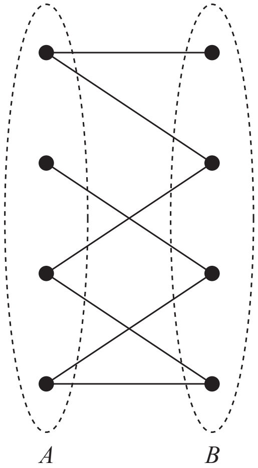
The Marriage Problem:

Given n men and n women, under what conditions is it possible to pair each man with a woman such that every pair know each other?



Definitions

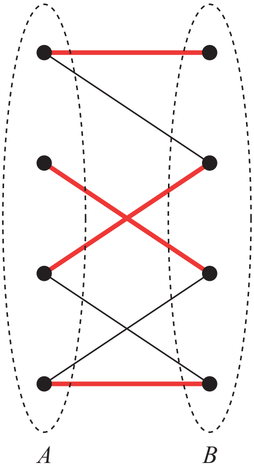
A graph G is *bipartite* if we can partition $V(G)$ into two sets A and B so that every edge of G crosses between A and B.



Definitions

A graph G is *bipartite* if we can partition $V(G)$ into two sets A and B so that every edge of G crosses between A and B.

We say $M \subseteq E(G)$ is a *matching* if the edges in M are pairwise disjoint.

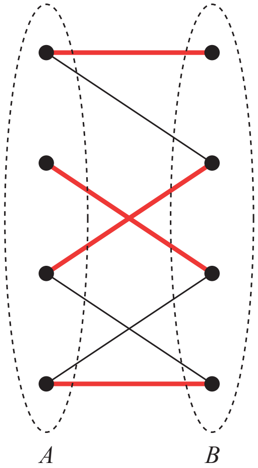


Definitions

A graph G is *bipartite* if we can partition $V(G)$ into two sets A and B so that every edge of G crosses between A and B.

We say $M \subseteq E(G)$ is a *matching* if the edges in M are pairwise disjoint.

We say M is *perfect* if every vertex belongs to some edge of M.

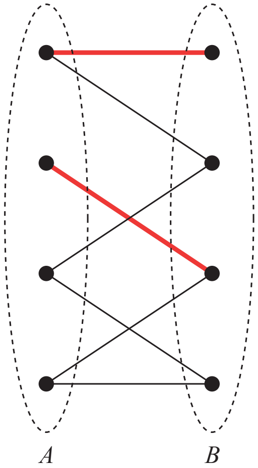


Definitions

A graph G is *bipartite* if we can partition $V(G)$ into two sets A and B so that every edge of G crosses between A and B.

We say $M \subseteq E(G)$ is a *matching* if the edges in M are pairwise disjoint.

We say M is *perfect* if every vertex belongs to some edge of M.



Maximal size matchings

How can we produce a matching of maximal size?

Maximal size matchings

How can we produce a matching of maximal size?

The greedy algorithm does not work.

Maximal size matchings

How can we produce a matching of maximal size?

The greedy algorithm does not work.

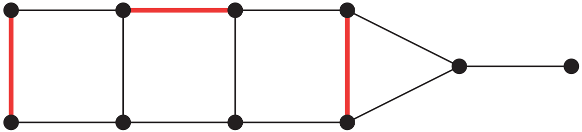
Alternating and augmenting paths

Let G be a graph.
Let M be matching in G.
Let P be a path in G.

Alternating and augmenting paths

Let G be a graph.
Let M be matching in G.
Let P be a path in G.

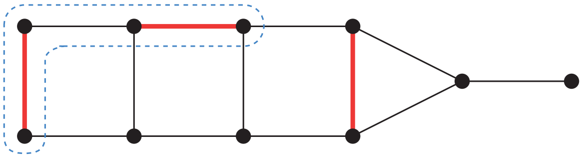
We say P is *M-alternating* if every other edge of P is in M.



Alternating and augmenting paths

Let G be a graph.
Let M be matching in G.
Let P be a path in G.

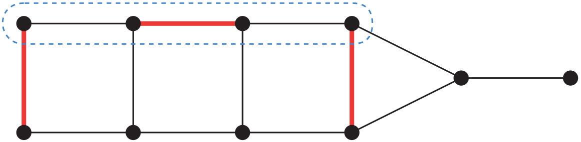
We say P is *M-alternating* if every other edge of P is in M.



Alternating and augmenting paths

Let G be a graph.
Let M be matching in G.
Let P be a path in G.

We say P is *M-alternating* if every other edge of P is in M.

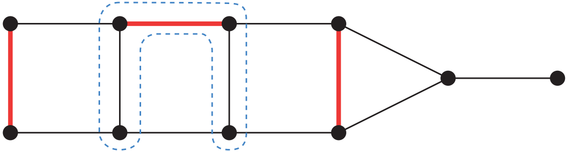


Alternating and augmenting paths

Let G be a graph.
Let M be matching in G.
Let P be a path in G.

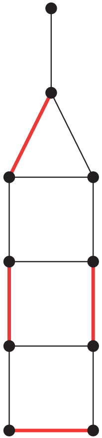
We say P is *M-alternating* if every other edge of P is in M.

We say P is *M-augmenting* if P is M-alternating and its end vertices are not in any edge of M.



Maximal size matchings

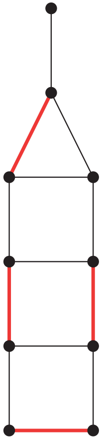
Lemma 16. Let M be a matching in G. Then M is not of maximum size if and only if there is an M-augmenting path in G.



Maximal size matchings

Lemma 16. Let M be a matching in G. Then M is not of maximum size if and only if there is an M-augmenting path in G.

Proof. If there is an M-augmenting path P in G then we can find a larger matching by 'flipping' P: replace M by $M \setminus (M \cap E(P)) \cup (E(P) \setminus M)$.

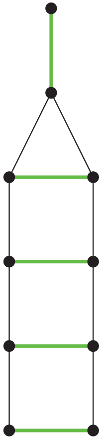


Maximal size matchings

Lemma 16. Let M be a matching in G. Then M is not of maximum size if and only if there is an M-augmenting path in G.

Proof. If there is an M-augmenting path P in G then we can find a larger matching by 'flipping' P: replace M by $M \setminus (M \cap E(P)) \cup (E(P) \setminus M)$.

Conversely, suppose that M^* is a matching in G with $|M^*| > |M|$.

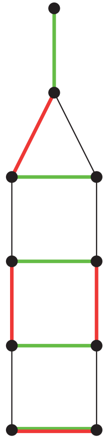


Maximal size matchings

Lemma 16. Let M be a matching in G. Then M is not of maximum size if and only if there is an M-augmenting path in G.

Proof. If there is an M-augmenting path P in G then we can find a larger matching by 'flipping' P: replace M by $M \setminus (M \cap E(P)) \cup (E(P) \setminus M)$.

Conversely, suppose that M^* is a matching in G with $|M^*| > |M|$.
Let $H = M \cup M^*$.

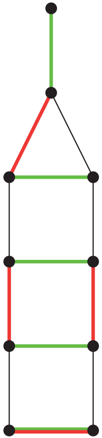


Maximal size matchings

Lemma 16. Let M be a matching in G. Then M is not of maximum size if and only if there is an M-augmenting path in G.

Proof. If there is an M-augmenting path P in G then we can find a larger matching by 'flipping' P: replace M by $M \setminus (M \cap E(P)) \cup (E(P) \setminus M)$.

Conversely, suppose that M^* is a matching in G with $|M^*| > |M|$.
Let $H = M \cup M^*$.
Every vertex has degree at most 2 in H, so each component of H is an edge, path or cycle, the edge components consist of $M \cap M^*$, and the edges in path and cycle components alternate between M and M^*.

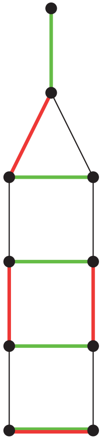


Maximal size matchings

Lemma 16. Let M be a matching in G. Then M is not of maximum size if and only if there is an M-augmenting path in G.

Proof. If there is an M-augmenting path P in G then we can find a larger matching by 'flipping' P: replace M by $M \setminus (M \cap E(P)) \cup (E(P) \setminus M)$.

Conversely, suppose that M^* is a matching in G with $|M^*| > |M|$.
Let $H = M \cup M^*$.
Every vertex has degree at most 2 in H, so each component of H is an edge, path or cycle, the edge components consist of $M \cap M^*$, and the edges in path and cycle components alternate between M and M^*.
As $|M^*| > |M|$ we can find a path component with more edges of M^* than M: this is an M-augmenting path in G. $\square$



Finding a maximal size matching

Lemma 16 reduces the algorithmic question of finding a maximum matching in G to the following: given a matching M in G, find an M-augmenting path or show that there is none.

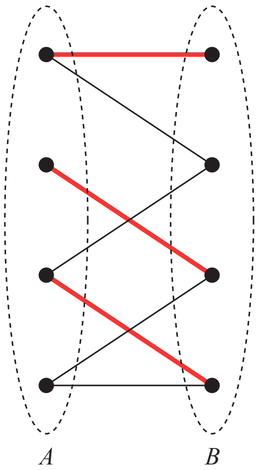
Finding a maximal size matching

Lemma 16 reduces the algorithmic question of finding a maximum matching in G to the following: given a matching M in G, find an M-augmenting path or show that there is none.

We'll focus on the case of bipartite graphs.

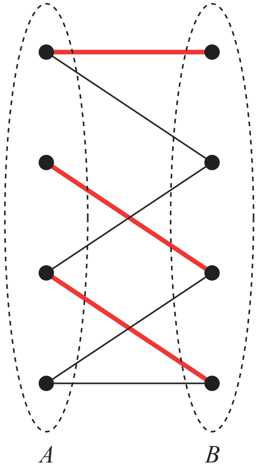
Finding augmenting paths in bipartite graphs

Now suppose that G is bipartite, with parts A and B.



Finding augmenting paths in bipartite graphs

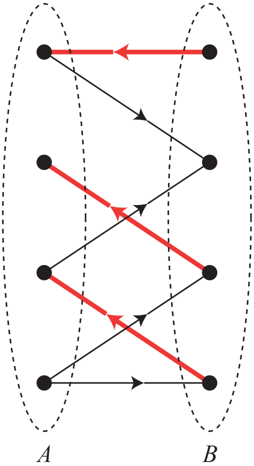
Now suppose that G is bipartite, with parts A and B.
Let M be a matching.



Finding augmenting paths in bipartite graphs

Now suppose that G is bipartite, with parts A and B.
Let M be a matching.

We put directions on $E(G)$, so that all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

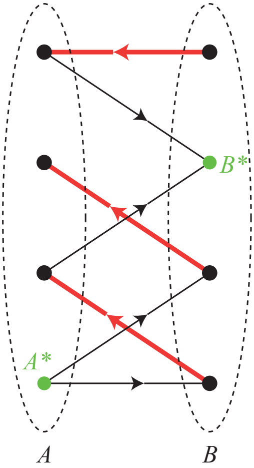


Finding augmenting paths in bipartite graphs

Now suppose that G is bipartite, with parts A and B.
Let M be a matching.

We put directions on $E(G)$, so that all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

Let A^* and B^* be the vertices in A and B that are 'uncovered', i.e. not in any edge of M.



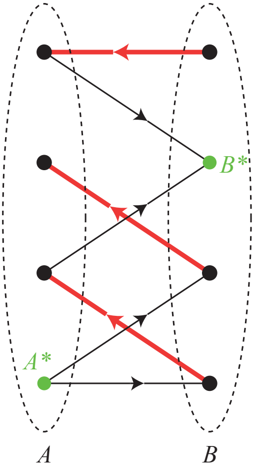
Finding augmenting paths in bipartite graphs

Now suppose that G is bipartite, with parts A and B.
Let M be a matching.

We put directions on $E(G)$, so that all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

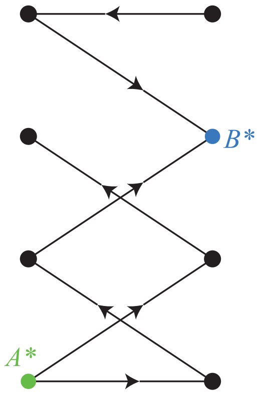
Let A^* and B^* be the vertices in A and B that are 'uncovered', i.e. not in any edge of M.

Then an M-augmenting path is equivalent to a directed path from A^* to B^*, i.e. a path that respects directions of edges.



Finding a directed path

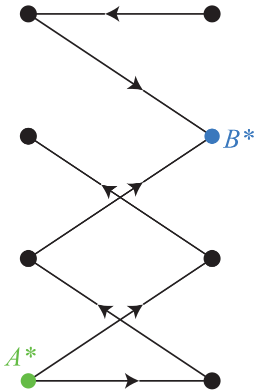
Is there a directed path from A^* to B^*?



Finding a directed path

Is there a directed path from A^* to B^*?

More generally, suppose that we have a directed graph with subsets A^* and B^* of $V(G)$. Is there a directed path from A^* to B^*?

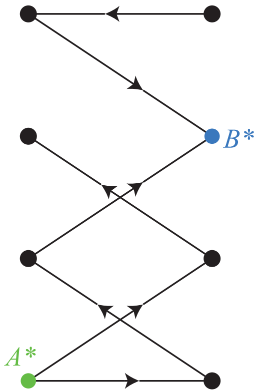


Finding a directed path

Is there a directed path from A^* to B^*?

More generally, suppose that we have a directed graph with subsets A^* and B^* of $V(G)$. Is there a directed path from A^* to B^*?

Start with $R = A^*$.

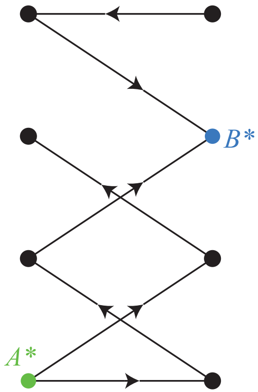


Finding a directed path

Is there a directed path from A^* to B^*?

More generally, suppose that we have a directed graph with subsets A^* and B^* of $V(G)$. Is there a directed path from A^* to B^*?

Start with $R = A^*$.
Search Algorithm. Repeat the following step: if there is any edge directed from some $x \in R$ to some $y \notin R$ then add y to R, otherwise stop.



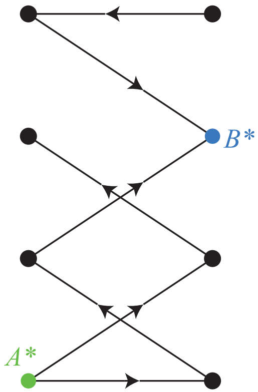
Finding a directed path

Is there a directed path from A^* to B^*?

More generally, suppose that we have a directed graph with subsets A^* and B^* of $V(G)$. Is there a directed path from A^* to B^*?

Start with $R = A^*$.
Search Algorithm. Repeat the following step: if there is any edge directed from some $x \in R$ to some $y \notin R$ then add y to R, otherwise stop.

There is a directed path from A^* to B^* if and only if the final R intersects B^*.



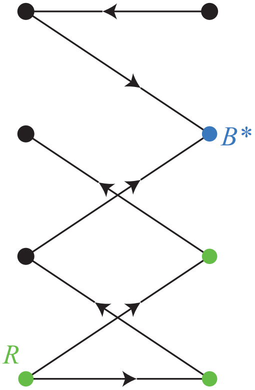
Finding a directed path

Is there a directed path from A^* to B^*?

More generally, suppose that we have a directed graph with subsets A^* and B^* of $V(G)$. Is there a directed path from A^* to B^*?

Start with $R = A^*$.
Search Algorithm. Repeat the following step: if there is any edge directed from some $x \in R$ to some $y \notin R$ then add y to R, otherwise stop.

There is a directed path from A^* to B^* if and only if the final R intersects B^*.



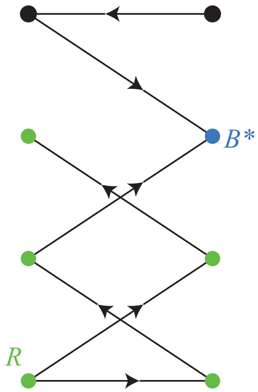
Finding a directed path

Is there a directed path from A^* to B^*?

More generally, suppose that we have a directed graph with subsets A^* and B^* of $V(G)$. Is there a directed path from A^* to B^*?

Start with $R = A^*$.
Search Algorithm. Repeat the following step: if there is any edge directed from some $x \in R$ to some $y \notin R$ then add y to R, otherwise stop.

There is a directed path from A^* to B^* if and only if the final R intersects B^*.



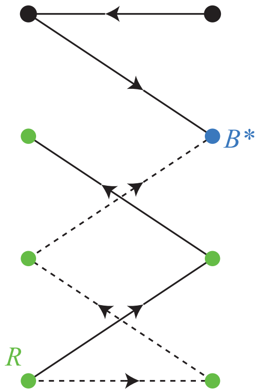
Finding a directed path

Is there a directed path from A^* to B^*?

More generally, suppose that we have a directed graph with subsets A^* and B^* of $V(G)$. Is there a directed path from A^* to B^*?

Start with $R = A^*$.
Search Algorithm. Repeat the following step: if there is any edge directed from some $x \in R$ to some $y \notin R$ then add y to R, otherwise stop.

There is a directed path from A^* to B^* if and only if the final R intersects B^*.



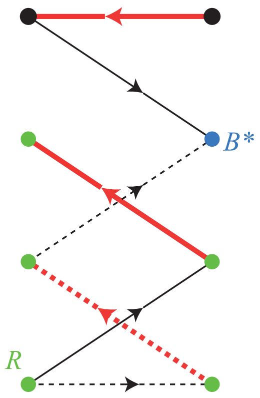
Finding a directed path

Is there a directed path from A^* to B^*?

More generally, suppose that we have a directed graph with subsets A^* and B^* of $V(G)$. Is there a directed path from A^* to B^*?

Start with $R = A^*$.
Search Algorithm. Repeat the following step: if there is any edge directed from some $x \in R$ to some $y \notin R$ then add y to R, otherwise stop.

There is a directed path from A^* to B^* if and only if the final R intersects B^*.



The Hungarian algorithm

This finds a matching of maximum size in a bipartite graph G.

The Hungarian algorithm

This finds a matching of maximum size in a bipartite graph G.

Start with $M = \emptyset$.

The Hungarian algorithm

This finds a matching of maximum size in a bipartite graph G.

Start with $M = \emptyset$.

Orient the edges of G: all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

The Hungarian algorithm

This finds a matching of maximum size in a bipartite graph G.

Start with $M = \emptyset$.

Orient the edges of G: all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

Let A^* and B^* be the vertices in A and B that are 'uncovered', i.e. not in any edge of M.

The Hungarian algorithm

This finds a matching of maximum size in a bipartite graph G.

Start with $M = \emptyset$.

Orient the edges of G: all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

Let A^* and B^* be the vertices in A and B that are 'uncovered', i.e. not in any edge of M.

Use the search algorithm to find a directed path from A^* to B^*.

The Hungarian algorithm

This finds a matching of maximum size in a bipartite graph G.

Start with $M = \emptyset$.

Orient the edges of G: all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

Let A^* and B^* be the vertices in A and B that are 'uncovered', i.e. not in any edge of M.

Use the search algorithm to find a directed path from A^* to B^*.

If there is no such path, stop. If there is, then it is M-augmenting and so we flip the path to increase the size of M.

The Hungarian algorithm

This finds a matching of maximum size in a bipartite graph G.

Start with $M = \emptyset$.

Orient the edges of G: all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

Let A^* and B^* be the vertices in A and B that are 'uncovered', i.e. not in any edge of M.

Use the search algorithm to find a directed path from A^* to B^*.

If there is no such path, stop. If there is, then it is M-augmenting and so we flip the path to increase the size of M.

Repeat.

The Hungarian algorithm

This finds a matching of maximum size in a bipartite graph G.

Start with $M = \emptyset$.

Orient the edges of G: all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

Let A^* and B^* be the vertices in A and B that are 'uncovered', i.e. not in any edge of M.

Use the search algorithm to find a directed path from A^* to B^*.

If there is no such path, stop. If there is, then it is M-augmenting and so we flip the path to increase the size of M.

Repeat.

The running time of the search algorithm is $O(|V(G)||E(G)|)$,

The Hungarian algorithm

This finds a matching of maximum size in a bipartite graph G.

Start with $M = \emptyset$.

Orient the edges of G: all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

Let A^* and B^* be the vertices in A and B that are 'uncovered', i.e. not in any edge of M.

Use the search algorithm to find a directed path from A^* to B^*.

If there is no such path, stop. If there is, then it is M-augmenting and so we flip the path to increase the size of M.

Repeat.

The running time of the search algorithm is $O(|V(G)||E(G)|)$, and there are at most $|V(G)|/2$ iterations of increasing the matching.

The Hungarian algorithm

This finds a matching of maximum size in a bipartite graph G.

Start with $M = \emptyset$.

Orient the edges of G: all edges in M are one-way from B to A, and all edges not in M are one-way from A to B.

Let A^* and B^* be the vertices in A and B that are 'uncovered', i.e. not in any edge of M.

Use the search algorithm to find a directed path from A^* to B^*.

If there is no such path, stop. If there is, then it is M-augmenting and so we flip the path to increase the size of M.

Repeat.

The running time of the search algorithm is $O(|V(G)||E(G)|)$, and there are at most $|V(G)|/2$ iterations of increasing the matching.

So the algorithm has running time $O(|V(G)|^2|E(G)|)$.

Matchings and covers

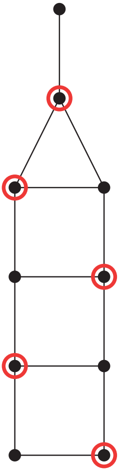
Covers

A *cover* for a graph G is a subset C of the vertices such that every edge contains at least one vertex of C.

Covers

A *cover* for a graph G is a subset C of the vertices such that every edge contains at least one vertex of C.

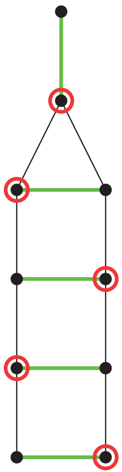
If M is any matching and C is any cover, then $|M| \leq |C|$.



Covers

A *cover* for a graph G is a subset C of the vertices such that every edge contains at least one vertex of C.

If M is any matching and C is any cover, then $|M| \leq |C|$.

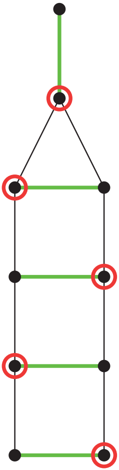


Covers

A *cover* for a graph G is a subset C of the vertices such that every edge contains at least one vertex of C.

If M is any matching and C is any cover, then $|M| \leq |C|$.

To see this, define an injective map $f : M \to C$, where $f(e)$ is any vertex of $e \cap C$.



Matchings and covers

If M is any matching and C is any cover, then $|M| \leq |C|$.

Matchings and covers

If M is any matching and C is any cover, then $|M| \leq |C|$.

Maximum matching / minimum cover:

Matchings and covers

If M is any matching and C is any cover, then $|M| \leq |C|$.

Maximum matching / minimum cover:

Suppose that we had found a matching M and a cover C such that $|M| = |C|$.

Matchings and covers

If M is any matching and C is any cover, then $|M| \leq |C|$.

Maximum matching / minimum cover:

Suppose that we had found a matching M and a cover C such that $|M| = |C|$.
Then we would know that M was a maximal size matching and C was a minimal size cover.

Matchings and covers

If M is any matching and C is any cover, then $|M| \leq |C|$.

Maximum matching / minimum cover:

Suppose that we had found a matching M and a cover C such that $|M| = |C|$.
Then we would know that M was a maximal size matching and C was a minimal size cover.

This is an example of 'weak duality'.

Matchings and covers

If M is any matching and C is any cover, then $|M| \leq |C|$.

Maximum matching / minimum cover:

Suppose that we had found a matching M and a cover C such that $|M| = |C|$.
Then we would know that M was a maximal size matching and C was a minimal size cover.

This is an example of 'weak duality'.

This suggests the question of whether equality holds.

Matchings and covers

If M is any matching and C is any cover, then $|M| \leq |C|$.

Maximum matching / minimum cover:

Suppose that we had found a matching M and a cover C such that $|M| = |C|$.
Then we would know that M was a maximal size matching and C was a minimal size cover.

This is an example of 'weak duality'.

This suggests the question of whether equality holds. The answer to the question is 'no' in general:

Matchings and covers

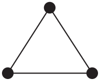
If M is any matching and C is any cover, then $|M| \leq |C|$.

Maximum matching / minimum cover:

Suppose that we had found a matching M and a cover C such that $|M| = |C|$.
Then we would know that M was a maximal size matching and C was a minimal size cover.

This is an example of 'weak duality'.

This suggests the question of whether equality holds. The answer to the question is 'no' in general:



Matchings and covers

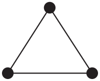
If M is any matching and C is any cover, then $|M| \leq |C|$.

Maximum matching / minimum cover:

Suppose that we had found a matching M and a cover C such that $|M| = |C|$.
Then we would know that M was a maximal size matching and C was a minimal size cover.

This is an example of 'weak duality'.

This suggests the question of whether equality holds. The answer to the question is 'no' in general:



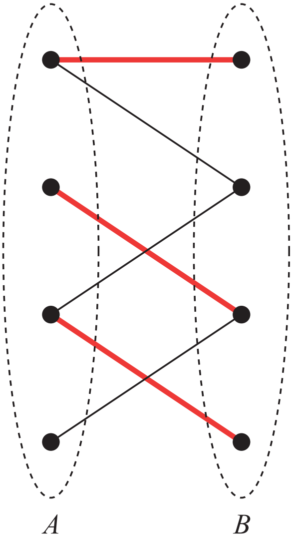
The maximum matching has size 1 but the minimum cover has size 2.

König's Theorem

König's Theorem. In any bipartite graph, the size of a maximum matching equals the size of a minimum cover.

Proof

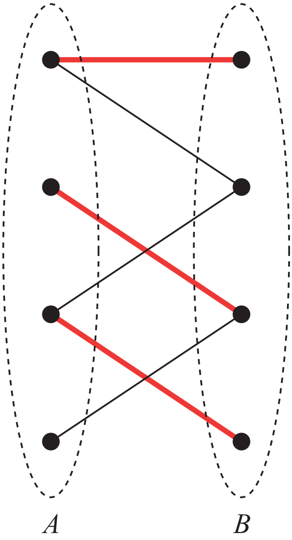
Let G be a bipartite graph with parts A and B. Let M be a maximum matching in G.



Proof

Let G be a bipartite graph with parts A and B. Let M be a maximum matching in G.

It suffices to find a cover C with $|C| = |M|$.

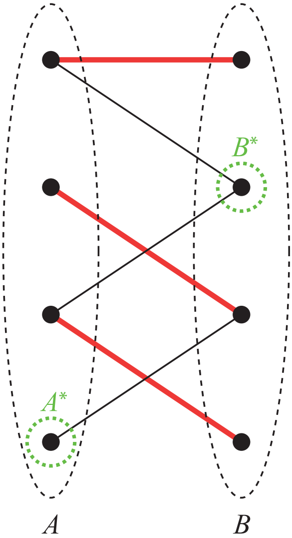


Proof

Let G be a bipartite graph with parts A and B. Let M be a maximum matching in G.

It suffices to find a cover C with $|C| = |M|$.

Recall that we write A^* and B^* for the uncovered vertices in A and B.



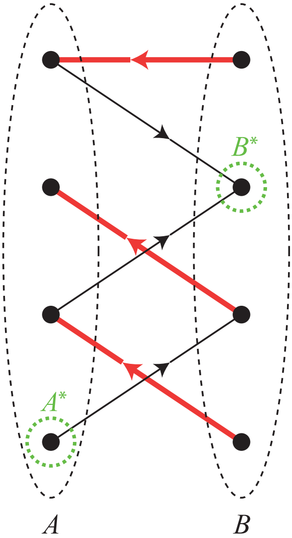
Proof

Let G be a bipartite graph with parts A and B. Let M be a maximum matching in G.

It suffices to find a cover C with $|C| = |M|$.

Recall that we write A^* and B^* for the uncovered vertices in A and B.

Consider the search algorithm for an M-augmenting path in G.



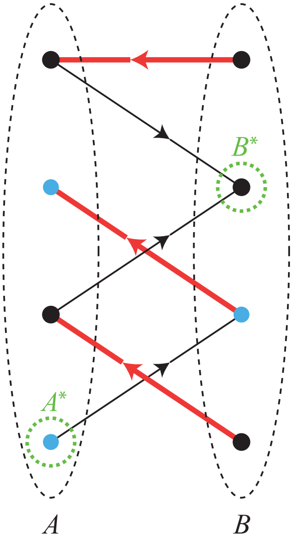
Proof

Let G be a bipartite graph with parts A and B. Let M be a maximum matching in G.

It suffices to find a cover C with $|C| = |M|$.

Recall that we write A^* and B^* for the uncovered vertices in A and B.

Consider the search algorithm for an M-augmenting path in G. The algorithm terminates with some set R that consists of all vertices reachable by M-alternating paths starting in A^*.



Proof

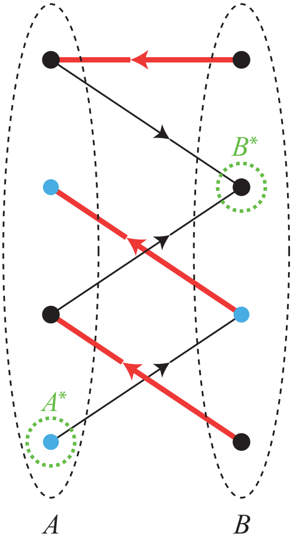
Let G be a bipartite graph with parts A and B. Let M be a maximum matching in G.

It suffices to find a cover C with $|C| = |M|$.

Recall that we write A^* and B^* for the uncovered vertices in A and B.

Consider the search algorithm for an M-augmenting path in G. The algorithm terminates with some set R that consists of all vertices reachable by M-alternating paths starting in A^*.

As M is maximum there is no M-augmenting path, so $R \cap B^* = \emptyset$.



Proof

Let G be a bipartite graph with parts A and B. Let M be a maximum matching in G.

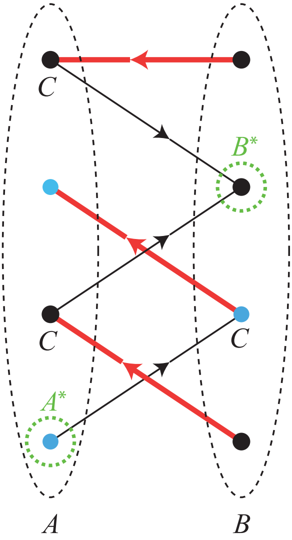
It suffices to find a cover C with $|C| = |M|$.

Recall that we write A^* and B^* for the uncovered vertices in A and B.

Consider the search algorithm for an M-augmenting path in G. The algorithm terminates with some set R that consists of all vertices reachable by M-alternating paths starting in A^*.

As M is maximum there is no M-augmenting path, so $R \cap B^* = \emptyset$.

Let $C = (A \setminus R) \cup (B \cap R)$.



Proof

Let G be a bipartite graph with parts A and B. Let M be a maximum matching in G.

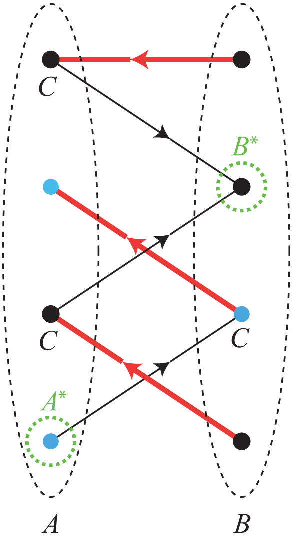
It suffices to find a cover C with $|C| = |M|$.

Recall that we write A^* and B^* for the uncovered vertices in A and B.

Consider the search algorithm for an M-augmenting path in G. The algorithm terminates with some set R that consists of all vertices reachable by M-alternating paths starting in A^*.

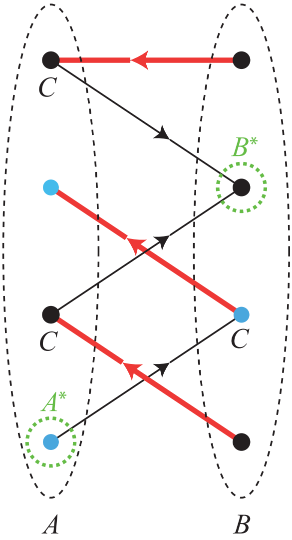
As M is maximum there is no M-augmenting path, so $R \cap B^* = \emptyset$.

Let $C = (A \setminus R) \cup (B \cap R)$.
We claim that C is a cover with $|C| = |M|$.



Proof

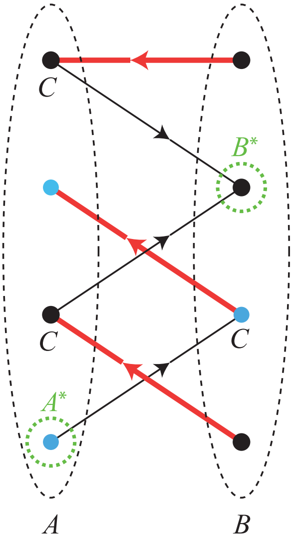
$$C = (A \setminus R) \cup (B \cap R).$$



Proof

$C = (A \setminus R) \cup (B \cap R)$.

We start by showing that C is a cover.

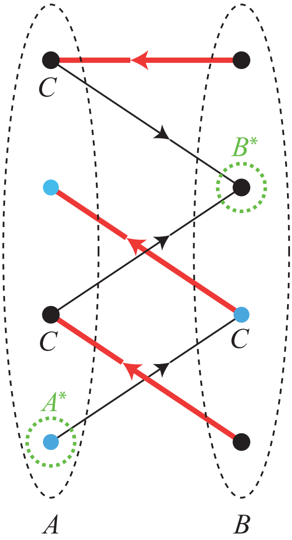


Proof

$C = (A \setminus R) \cup (B \cap R)$.

We start by showing that C is a cover.

Suppose not. Then there is $ab \in E(G)$ with $a \in A \cap R$ and $b \in B \setminus R$.



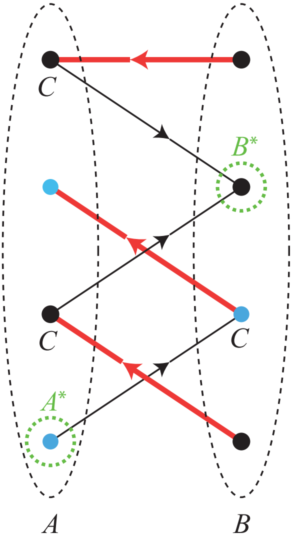
Proof

$C = (A \setminus R) \cup (B \cap R)$.

We start by showing that C is a cover.

Suppose not. Then there is $ab \in E(G)$ with $a \in A \cap R$ and $b \in B \setminus R$.

However, this contradicts the definition of R, as b must be reachable from A^*: if $ab \in M$ we must reach a via b or if $ab \notin M$ we can reach b via a.



Proof

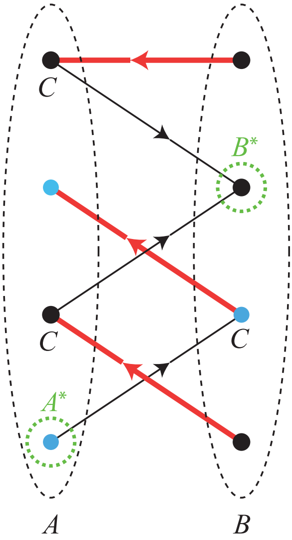
$C = (A \setminus R) \cup (B \cap R)$.

We start by showing that C is a cover.

Suppose not. Then there is $ab \in E(G)$ with $a \in A \cap R$ and $b \in B \setminus R$.

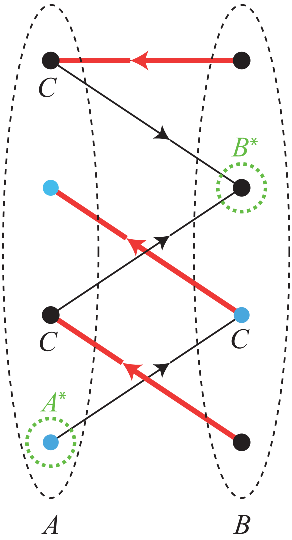
However, this contradicts the definition of R, as b must be reachable from A^*: if $ab \in M$ we must reach a via b or if $ab \notin M$ we can reach b via a.

Thus C is a cover.



Proof

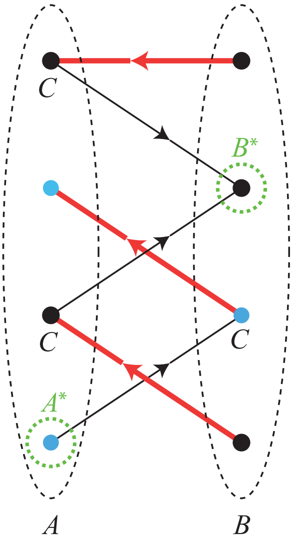
$$C = (A \setminus R) \cup (B \cap R).$$



Proof

$$C = (A \setminus R) \cup (B \cap R).$$

It remains to show $|C| = |M|$.

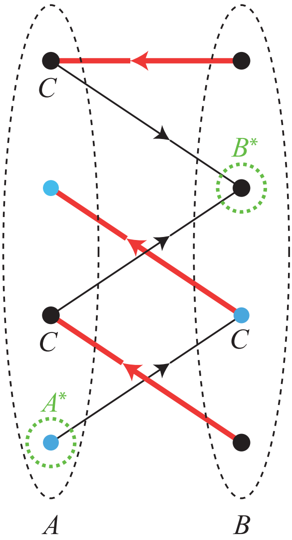


Proof

$C = (A \setminus R) \cup (B \cap R)$.

It remains to show $|C| = |M|$.

It suffices to show that every vertex in C is covered by some edge of M, and that no edge of M covers two vertices of C.



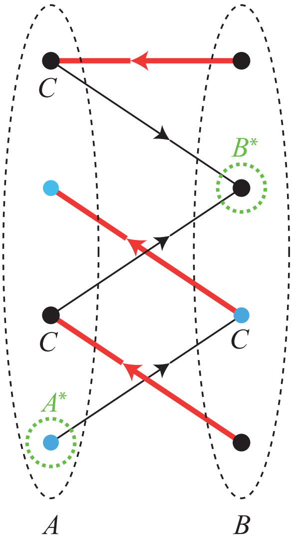
Proof

$C = (A \setminus R) \cup (B \cap R)$.

It remains to show $|C| = |M|$.

It suffices to show that every vertex in C is covered by some edge of M, and that no edge of M covers two vertices of C.

(This will show $|C| \leq |M|$, and we noted previously that $|M| \leq |C|$ is immediate from the definitions.)



Proof

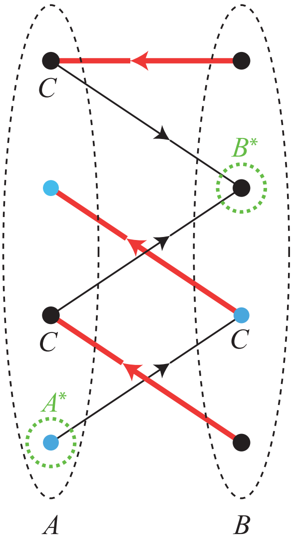
$C = (A \setminus R) \cup (B \cap R)$.

It remains to show $|C| = |M|$.

It suffices to show that every vertex in C is covered by some edge of M, and that no edge of M covers two vertices of C.

(This will show $|C| \leq |M|$, and we noted previously that $|M| \leq |C|$ is immediate from the definitions.)

Firstly, any $a \in A \setminus R$ is covered by M as $A^* \subseteq R$.



Proof

$C = (A \setminus R) \cup (B \cap R)$.

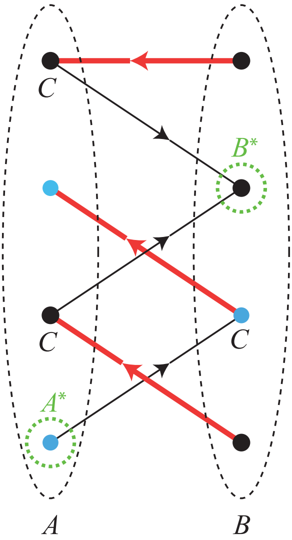
It remains to show $|C| = |M|$.

It suffices to show that every vertex in C is covered by some edge of M, and that no edge of M covers two vertices of C.

(This will show $|C| \leq |M|$, and we noted previously that $|M| \leq |C|$ is immediate from the definitions.)

Firstly, any $a \in A \setminus R$ is covered by M as $A^* \subseteq R$.

Secondly, any $b \in B \cap R$ is covered by M, or $b \in B^* \cap R = \emptyset$ gives a contradiction.



Proof

$C = (A \setminus R) \cup (B \cap R)$.

It remains to show $|C| = |M|$.

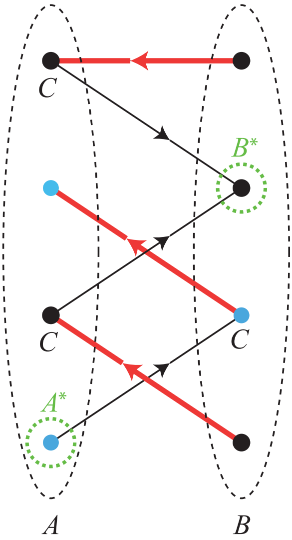
It suffices to show that every vertex in C is covered by some edge of M, and that no edge of M covers two vertices of C.

(This will show $|C| \leq |M|$, and we noted previously that $|M| \leq |C|$ is immediate from the definitions.)

Firstly, any $a \in A \setminus R$ is covered by M as $A^* \subseteq R$.

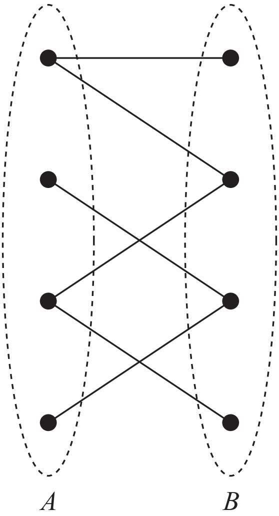
Secondly, any $b \in B \cap R$ is covered by M, or $b \in B^* \cap R = \emptyset$ gives a contradiction.

Finally, if $ab \in M$ with $a \in A \setminus R$, $b \in B \cap R$ then we can reach a via b, contradicting $a \notin R$. Thus $|C| = |M|$. □



The marriage problem

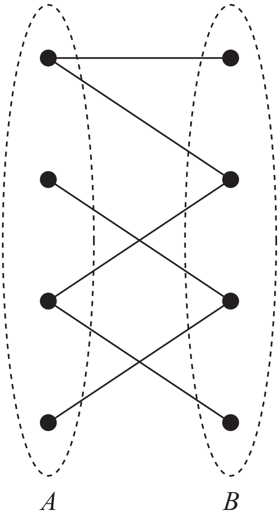
Let G be a bipartite graph with parts A and B.



The marriage problem

Let G be a bipartite graph with parts A and B.

We consider the more general question of whether there is a matching that covers every vertex in A; if $|B| = |A|$ then this will be perfect.



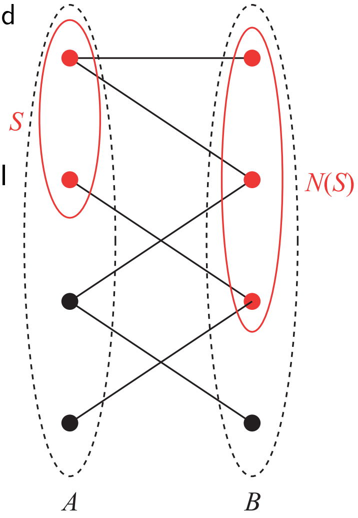
The marriage problem

Let G be a bipartite graph with parts A and B.

We consider the more general question of whether there is a matching that covers every vertex in A; if $|B| = |A|$ then this will be perfect.

For $S \subseteq A$ the *neighbourhood* of S is

$$N(S) = \bigcup_{a \in S} \{b : ab \in E(G)\}.$$



The marriage problem

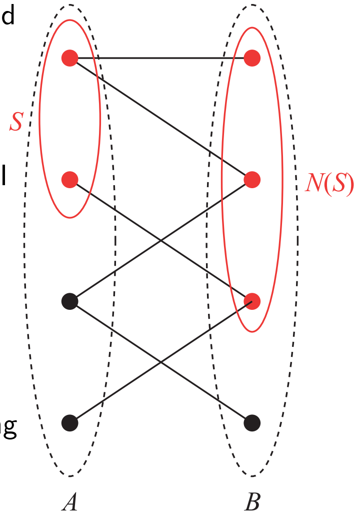
Let G be a bipartite graph with parts A and B.

We consider the more general question of whether there is a matching that covers every vertex in A; if $|B| = |A|$ then this will be perfect.

For $S \subseteq A$ the *neighbourhood* of S is

$$N(S) = \bigcup_{a \in S} \{b : ab \in E(G)\}.$$

Note that if G has a matching M covering A then each $a \in S$ has a 'match' a' with $aa' \in M$, and the matches are distinct, so $|N(S)| \geq |S|$.



The marriage problem

Let G be a bipartite graph with parts A and B.

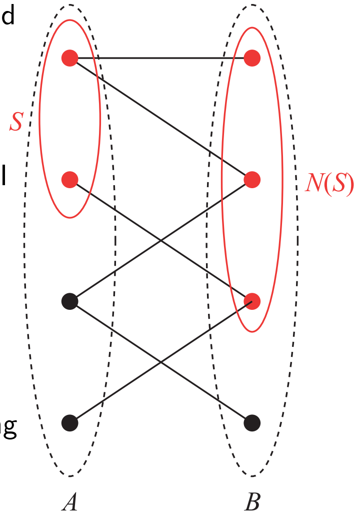
We consider the more general question of whether there is a matching that covers every vertex in A; if $|B| = |A|$ then this will be perfect.

For $S \subseteq A$ the *neighbourhood* of S is

$$N(S) = \bigcup_{a \in S} \{b : ab \in E(G)\}.$$

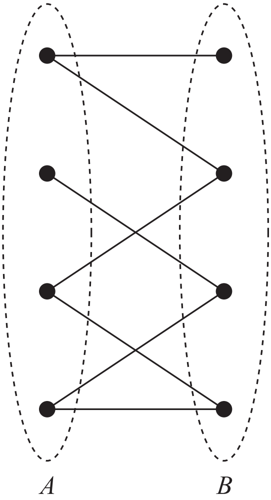
Note that if G has a matching M covering A then each $a \in S$ has a 'match' a' with $aa' \in M$, and the matches are distinct, so $|N(S)| \geq |S|$.

This gives a necessary condition for G to have a matching; it is also sufficient ...



The marriage problem

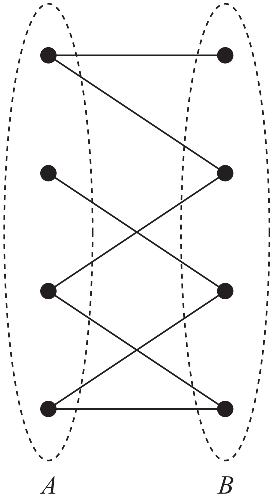
Hall's Theorem. Let G be a bipartite graph with parts A and B. Then G has a matching covering A if and only if every $S \subseteq A$ has $|N(S)| \geq |S|$.



The marriage problem

Hall's Theorem. Let G be a bipartite graph with parts A and B. Then G has a matching covering A if and only if every $S \subseteq A$ has $|N(S)| \geq |S|$.

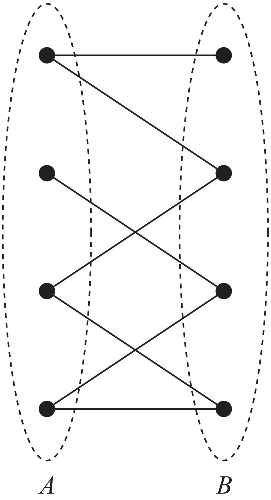
Proof.



The marriage problem

Hall's Theorem. Let G be a bipartite graph with parts A and B. Then G has a matching covering A if and only if every $S \subseteq A$ has $|N(S)| \geq |S|$.

Proof. We have already remarked that the condition is necessary.

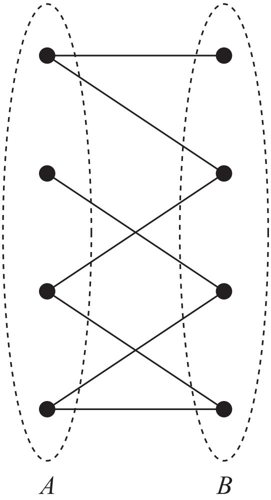


The marriage problem

Hall's Theorem. Let G be a bipartite graph with parts A and B. Then G has a matching covering A if and only if every $S \subseteq A$ has $|N(S)| \geq |S|$.

Proof. We have already remarked that the condition is necessary.

Conversely, suppose that every $S \subseteq A$ has $|N(S)| \geq |S|$.



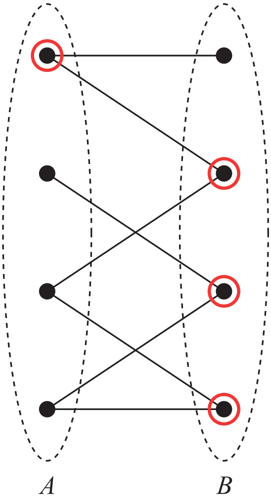
The marriage problem

Hall's Theorem. Let G be a bipartite graph with parts A and B. Then G has a matching covering A if and only if every $S \subseteq A$ has $|N(S)| \geq |S|$.

Proof. We have already remarked that the condition is necessary.

Conversely, suppose that every $S \subseteq A$ has $|N(S)| \geq |S|$.

Let C be any cover of G. By König's Theorem, it suffices to show $|C| \geq |A|$.



The marriage problem

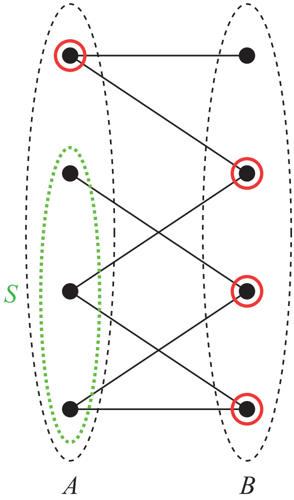
Hall's Theorem. Let G be a bipartite graph with parts A and B. Then G has a matching covering A if and only if every $S \subseteq A$ has $|N(S)| \geq |S|$.

Proof. We have already remarked that the condition is necessary.

Conversely, suppose that every $S \subseteq A$ has $|N(S)| \geq |S|$.

Let C be any cover of G. By König's Theorem, it suffices to show $|C| \geq |A|$.

To see this, let $S = A \setminus C$.



The marriage problem

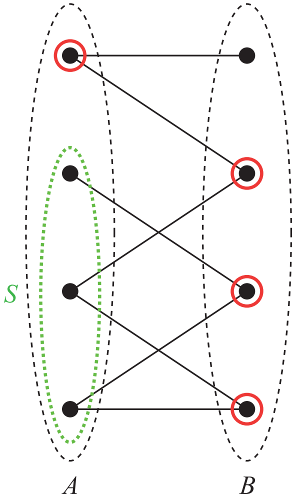
<u>Hall's Theorem.</u> Let G be a bipartite graph with parts A and B. Then G has a matching covering A if and only if every $S \subseteq A$ has $|N(S)| \geq |S|$.

<u>Proof.</u> We have already remarked that the condition is necessary.

Conversely, suppose that every $S \subseteq A$ has $|N(S)| \geq |S|$.

Let C be any cover of G. By König's Theorem, it suffices to show $|C| \geq |A|$.

To see this, let $S = A \setminus C$. Note that by definition of 'cover' we have $N(S) \subseteq B \cap C$.



The marriage problem

Hall's Theorem. Let G be a bipartite graph with parts A and B. Then G has a matching covering A if and only if every $S \subseteq A$ has $|N(S)| \geq |S|$.

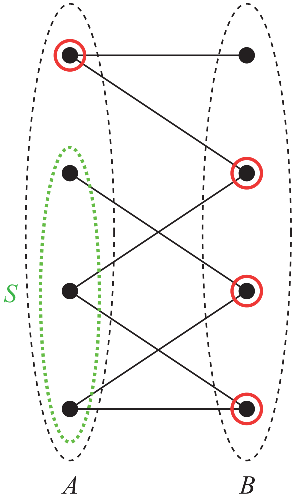
Proof. We have already remarked that the condition is necessary.

Conversely, suppose that every $S \subseteq A$ has $|N(S)| \geq |S|$.

Let C be any cover of G. By König's Theorem, it suffices to show $|C| \geq |A|$.

To see this, let $S = A \setminus C$. Note that by definition of 'cover' we have $N(S) \subseteq B \cap C$.

Then $|C| = |A \cap C| + |B \cap C| \geq |A| - |S| + |N(S)| \geq |A|$.



The Chinese Postman Problem

The Chinese Postman Problem

A postman collects a sack of letters from the sorting office, walks along every street to deliver them, and returns to the office. How can (s)he find the shortest route?

The Chinese Postman Problem

A postman collects a sack of letters from the sorting office, walks along every street to deliver them, and returns to the office. How can (s)he find the shortest route?

Let G be a connected graph. Let W be a closed walk in G.

The Chinese Postman Problem

A postman collects a sack of letters from the sorting office, walks along every street to deliver them, and returns to the office. How can (s)he find the shortest route?

Let G be a connected graph. Let W be a closed walk in G.
We call W a *postman walk* in G if it uses every edge of G at least once.

The Chinese Postman Problem

A postman collects a sack of letters from the sorting office, walks along every street to deliver them, and returns to the office. How can (s)he find the shortest route?

Let G be a connected graph. Let W be a closed walk in G.
We call W a *postman walk* in G if it uses every edge of G at least once.

For each $e \in E(G)$ let $c(e) > 0$ be the length of e. The length of W is $c(W) = \sum_{e \in W} c(e)$.

The Chinese Postman Problem

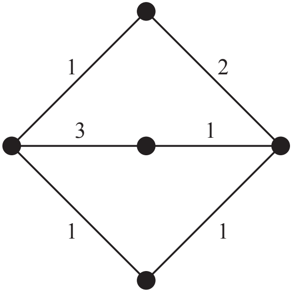
A postman collects a sack of letters from the sorting office, walks along every street to deliver them, and returns to the office. How can (s)he find the shortest route?

Let G be a connected graph. Let W be a closed walk in G.
We call W a *postman walk* in G if it uses every edge of G at least once.

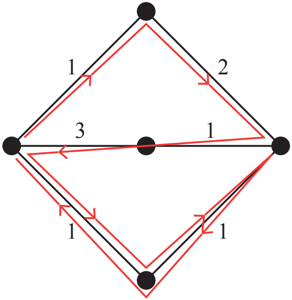
For each $e \in E(G)$ let $c(e) > 0$ be the length of e. The length of W is $c(W) = \sum_{e \in W} c(e)$.

We want to find a shortest postman walk.

Extensions

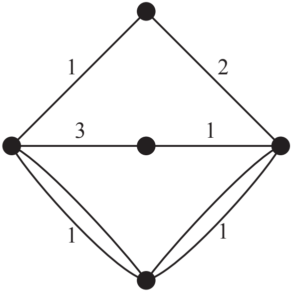


Extensions



Extensions

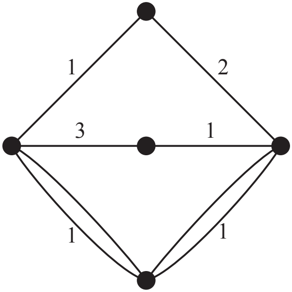
We can interpret a postman walk W as an Euler Tour in an *extension* of G, in which we introduce parallel edges, so that the number of parallel edges joining vertices x and y is the number of times that xy is used in W.



Extensions

We can interpret a postman walk W as an Euler Tour in an *extension* of G, in which we introduce parallel edges, so that the number of parallel edges joining vertices x and y is the number of times that xy is used in W.

Thus an equivalent reformulation of the Chinese Postman Problem is to find a *minimum weight Eulerian extension* G^* of G, i.e. G^* is obtained from G by copying some edges, so that all degrees in G^* are even, and $c(G^*)$ is as small as possible.



Edmonds' algorithm

We will describe an algorithm due to Edmonds.

Edmonds' algorithm

We will describe an algorithm due to Edmonds.

We assume that we have access to an algorithm for finding a minimum weight perfect matching in a weighted graph.

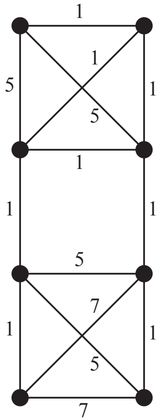
Edmonds' algorithm

We will describe an algorithm due to Edmonds.

We assume that we have access to an algorithm for finding a minimum weight perfect matching in a weighted graph.

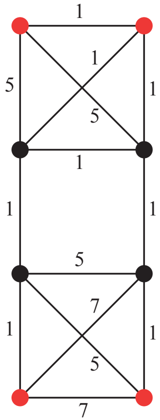
(An algorithm for this problem was also found by Edmonds, but it is beyond the scope of this course).

Edmonds' algorithm



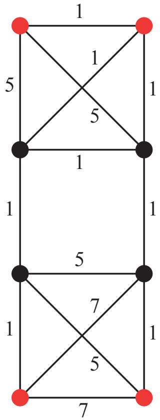
Edmonds' algorithm

1. Let X be the set of vertices with odd degree in G.



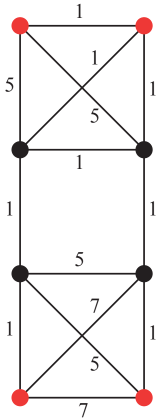
Edmonds' algorithm

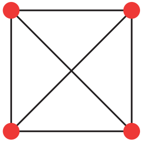
1. Let X be the set of vertices with odd degree in G.
   For each $x \in X$ find a c-shortest paths tree T_x rooted at x.



Edmonds' algorithm

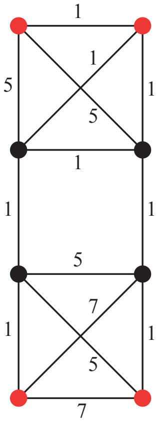
1. Let X be the set of vertices with odd degree in G.
   For each $x \in X$ find a c-shortest paths tree T_x rooted at x.

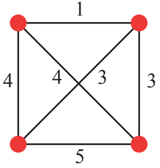




Edmonds' algorithm

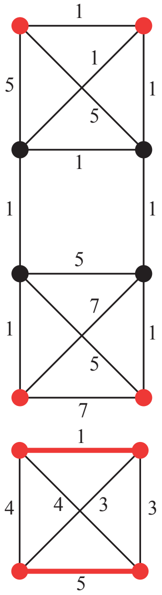
1. Let X be the set of vertices with odd degree in G.
   For each $x \in X$ find a c-shortest paths tree T_x rooted at x.
   Define a weight function w on pairs in X: let $w(xy) = c(P_{xy})$, where P_{xy} is the unique xy-path in T_x.





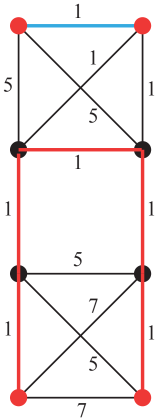
Edmonds' algorithm

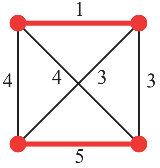
1. Let X be the set of vertices with odd degree in G.
   For each $x \in X$ find a c-shortest paths tree T_x rooted at x.
   Define a weight function w on pairs in X: let $w(xy) = c(P_{xy})$, where P_{xy} is the unique xy-path in T_x.
2. Find a perfect matching M on X with minimum w-weight.



Edmonds' algorithm

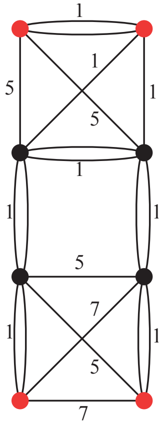
1. Let X be the set of vertices with odd degree in G.
   For each $x \in X$ find a c-shortest paths tree T_x rooted at x.
   Define a weight function w on pairs in X: let $w(xy) = c(P_{xy})$, where P_{xy} is the unique xy-path in T_x.
2. Find a perfect matching M on X with minimum w-weight.
   Let G^* be the Eulerian extension of G obtained by copying all edges of P_{xy} for all $xy \in M$.

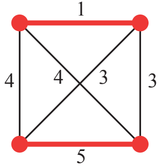




Edmonds' algorithm

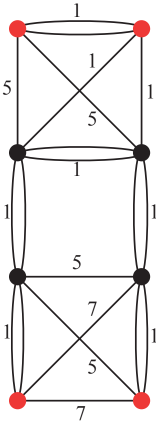
1. Let X be the set of vertices with odd degree in G.
   For each $x \in X$ find a c-shortest paths tree T_x rooted at x.
   Define a weight function w on pairs in X: let $w(xy) = c(P_{xy})$, where P_{xy} is the unique xy-path in T_x.
2. Find a perfect matching M on X with minimum w-weight.
   Let G^* be the Eulerian extension of G obtained by copying all edges of P_{xy} for all $xy \in M$.

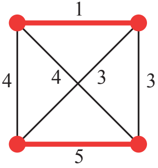




Edmonds' algorithm

1. Let X be the set of vertices with odd degree in G.
   For each $x \in X$ find a c-shortest paths tree T_x rooted at x.
   Define a weight function w on pairs in X: let $w(xy) = c(P_{xy})$, where P_{xy} is the unique xy-path in T_x.
2. Find a perfect matching M on X with minimum w-weight.
   Let G^* be the Eulerian extension of G obtained by copying all edges of P_{xy} for all $xy \in M$.
3. Find an Euler Tour W in G^*. Interpret W as a postman walk in G.





Edmonds' algorithm

Note that the perfect matching step makes sense as $|X|$ is even, by Lemma 10.

Edmonds' algorithm

Note that the perfect matching step makes sense as $|X|$ is even, by Lemma 10.

Lemma 19. Let H be a graph in which not all degrees are even. Then there is a path in H such that both ends have odd degree.

Edmonds' algorithm

Note that the perfect matching step makes sense as $|X|$ is even, by Lemma 10.

Lemma 19. Let H be a graph in which not all degrees are even. Then there is a path in H such that both ends have odd degree.

Proof.

Edmonds' algorithm

Note that the perfect matching step makes sense as $|X|$ is even, by Lemma 10.

Lemma 19. Let H be a graph in which not all degrees are even. Then there is a path in H such that both ends have odd degree.

Proof.
Pick a component of H containing a vertex of odd degree.

Edmonds' algorithm

Note that the perfect matching step makes sense as $|X|$ is even, by Lemma 10.

Lemma 19. Let H be a graph in which not all degrees are even. Then there is a path in H such that both ends have odd degree.

Proof.
Pick a component of H containing a vertex of odd degree.
By Lemma 10, there is another vertex of odd degree in H.

Edmonds' algorithm

Note that the perfect matching step makes sense as $|X|$ is even, by Lemma 10.

Lemma 19. Let H be a graph in which not all degrees are even. Then there is a path in H such that both ends have odd degree.

Proof.
Pick a component of H containing a vertex of odd degree.
By Lemma 10, there is another vertex of odd degree in H.
Pick a path joining these two vertices. □

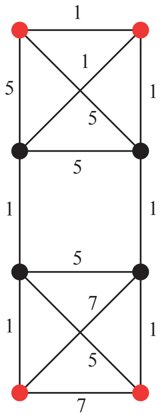
Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

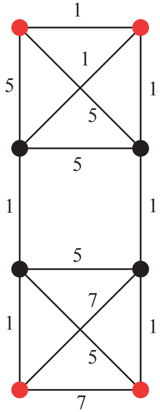
Proof.



Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

Proof.
Let W^* be a minimum length postman walk. It suffices to show that the algorithm finds a postman walk that is no longer than W^*.

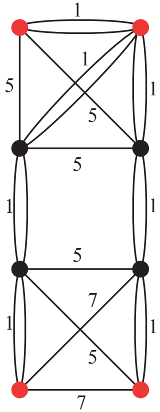


Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

Proof.
Let W^* be a minimum length postman walk. It suffices to show that the algorithm finds a postman walk that is no longer than W^*.

Let G^* be the Eulerian extension of G defined by W^*.

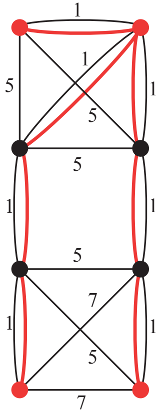


Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

Proof.
Let W^* be a minimum length postman walk. It suffices to show that the algorithm finds a postman walk that is no longer than W^*.

Let G^* be the Eulerian extension of G defined by W^*. Let H be the graph of copied edges: $E(H) = E(G^*) \setminus E(G)$.

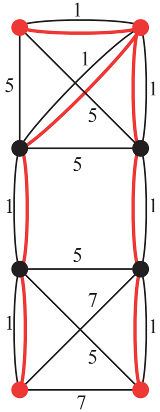


Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

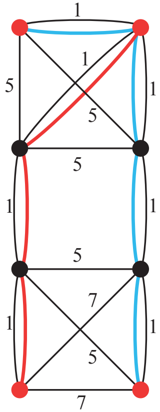
Proof.
Let W^* be a minimum length postman walk. It suffices to show that the algorithm finds a postman walk that is no longer than W^*.

Let G^* be the Eulerian extension of G defined by W^*. Let H be the graph of copied edges: $E(H) = E(G^*) \setminus E(G)$. Note that the set of vertices with odd degree in H is X (i.e. the same set as for G).



Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.



Proof.

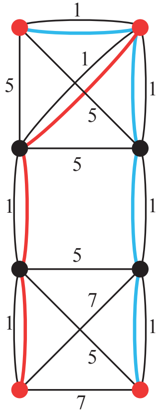
Let W^* be a minimum length postman walk. It suffices to show that the algorithm finds a postman walk that is no longer than W^*.

Let G^* be the Eulerian extension of G defined by W^*. Let H be the graph of copied edges: $E(H) = E(G^*) \setminus E(G)$. Note that the set of vertices with odd degree in H is X (i.e. the same set as for G).

We construct a set of paths in H by repeating the following procedure: if the current graph has any vertices of odd degree, apply Lemma 19 to find a path P such that both ends have odd degree, delete the edges of P and repeat.

Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

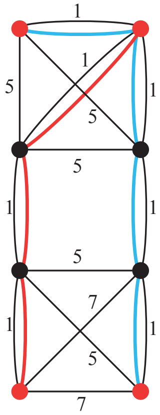


Proof.
Let W^* be a minimum length postman walk. It suffices to show that the algorithm finds a postman walk that is no longer than W^*.

Let G^* be the Eulerian extension of G defined by W^*. Let H be the graph of copied edges: $E(H) = E(G^*) \setminus E(G)$. Note that the set of vertices with odd degree in H is X (i.e. the same set as for G).
We construct a set of paths in H by repeating the following procedure: if the current graph has any vertices of odd degree, apply Lemma 19 to find a path P such that both ends have odd degree, delete the edges of P and repeat.
This procedure pairs up the vertices in X so that each pair is connected by a path in H.

Edmonds' algorithm works

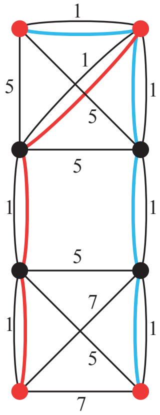
Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.



Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

Let $H' \subseteq H$ be the graph formed by the union of these paths.

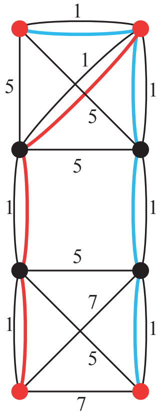


Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

Let $H' \subseteq H$ be the graph formed by the union of these paths.

Let G' be the Eulerian extension of G defined by copying the edges of H'.



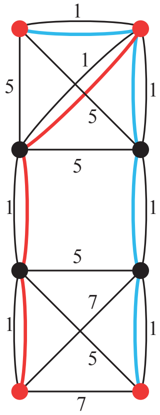
Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

Let $H' \subseteq H$ be the graph formed by the union of these paths.

Let G' be the Eulerian extension of G defined by copying the edges of H'.

Let W' be an Euler tour in G', interpreted as a postman walk in G. Then $c(W') \leq c(W^*)$.



Edmonds' algorithm works

Theorem 20. Edmonds' Algorithm finds a minimum length postman walk.

Let $H' \subseteq H$ be the graph formed by the union of these paths.

Let G' be the Eulerian extension of G defined by copying the edges of H'.

Let W' be an Euler tour in G', interpreted as a postman walk in G. Then $c(W') \leq c(W^*)$. By definition of the algorithm it finds a postman walk that is no longer than W'. $\square$