Recall

A **rogue couple** for a matching S, is a pair b and g, where b and g prefer each other to their partners.

Recall

A **rogue couple** for a matching *S*, is a pair *b* and *g*, where *b* and *g* prefer each other to their partners.

A matching is **stable** if no rogue couple.

Recall

A **rogue couple** for a matching S, is a pair b and g, where b and g prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs?

Recall

A **rogue couple** for a matching S, is a pair b and g, where b and g prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Recall

A **rogue couple** for a matching S, is a pair b and g, where b and g prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is b-optimal** if b's partner is its best partner in any stable pairing.

Recall

A **rogue couple** for a matching S, is a pair b and g, where b and g prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is b-optimal** if b's partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is b-optimal for **all** jobs b.

Recall

A **rogue couple** for a matching S, is a pair b and g, where b and g prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is b-optimal** if b's partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is b-optimal for **all** jobs b.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Recall

A **rogue couple** for a matching *S*, is a pair *b* and *g*, where *b* and *g* prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is *b*-optimal** if *b*'s partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is *b*-optimal for **all** jobs *b*.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

Recall

A **rogue couple** for a matching *S*, is a pair *b* and *g*, where *b* and *g* prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is *b*-optimal** if *b*'s partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is *b*-optimal for **all** jobs *b*.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True?

Recall

A **rogue couple** for a matching *S*, is a pair *b* and *g*, where *b* and *g* prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is *b*-optimal** if *b*'s partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is *b*-optimal for **all** jobs *b*.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True? False?

Recall

A **rogue couple** for a matching S, is a pair b and g, where b and g prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is b-optimal** if b's partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is b-optimal for **all** jobs b.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True? False? False!

Recall

A **rogue couple** for a matching *S*, is a pair *b* and *g*, where *b* and *g* prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is *b*-optimal** if *b*'s partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is *b*-optimal for **all** jobs *b*.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True? False? False!

Subtlety here: Best partner in any stable matching.

Recall

A **rogue couple** for a matching *S*, is a pair *b* and *g*, where *b* and *g* prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is *b*-optimal** if *b*'s partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is *b*-optimal for **all** jobs *b*.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True? False? False!

Subtlety here: Best partner in any stable matching.
As well as you can be in a globally stable solution!

Recall

A **rogue couple** for a matching *S*, is a pair *b* and *g*, where *b* and *g* prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is *b*-optimal** if *b*'s partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is *b*-optimal for **all** jobs *b*.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True? False? False!

Subtlety here: Best partner in any stable matching.
As well as you can be in a globally stable solution!

Question: Is there a job or candidate optimal matching?

Recall

A **rogue couple** for a matching *S*, is a pair *b* and *g*, where *b* and *g* prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is *b*-optimal** if *b*'s partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is *b*-optimal for **all** jobs *b*.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True? False? False!

Subtlety here: Best partner in any stable matching.
As well as you can be in a globally stable solution!

Question: Is there a job or candidate optimal matching?
Is it possible:

Recall

A **rogue couple** for a matching S, is a pair b and g, where b and g prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is b-optimal** if b's partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is b-optimal for **all** jobs b.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True? False? False!

Subtlety here: Best partner in any stable matching.
As well as you can be in a globally stable solution!

Question: Is there a job or candidate optimal matching?
Is it possible:
b-optimal pairing different from the b'-optimal matching!

Recall

A **rogue couple** for a matching *S*, is a pair *b* and *g*, where *b* and *g* prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is *b*-optimal** if *b*'s partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is *b*-optimal for **all** jobs *b*.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True? False? False!

Subtlety here: Best partner in any stable matching.
As well as you can be in a globally stable solution!

Question: Is there a job or candidate optimal matching?
Is it possible:
b-optimal pairing different from the b'-optimal matching!
Yes?

Recall

A **rogue couple** for a matching S, is a pair b and g, where b and g prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is b-optimal** if b's partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is b-optimal for **all** jobs b.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True? False? False!

Subtlety here: Best partner in any stable matching.
As well as you can be in a globally stable solution!

Question: Is there a job or candidate optimal matching?
Is it possible:
b-optimal pairing different from the b'-optimal matching!
Yes? No?

Recall

A **rogue couple** for a matching S, is a pair b and g, where b and g prefer each other to their partners.

A matching is **stable** if no rogue couple.

Is the Job-Proposes better for jobs? for candidates?

Definition: A **matching is b-optimal** if b's partner
is its best partner in any stable pairing.

Definition: A **matching is job optimal** if it is b-optimal for **all** jobs b.

..and so on for job pessimal, candidate optimal, candidate pessimal.

Claim: The optimal partner for a job must be first in its preference list.

True? False? False!

Subtlety here: Best partner in any stable matching.
As well as you can be in a globally stable solution!

Question: Is there a job or candidate optimal matching?
Is it possible:
b-optimal pairing different from the b'-optimal matching!
Yes? No?

Job Propose and Candidate Reject is optimal!

For jobs?

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not:

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.
So S is not a stable pairing.

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.
So S is not a stable pairing. Contradiction.

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.
So S is not a stable pairing. Contradiction. $\square$

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.
So S is not a stable pairing. Contradiction. □

Notes:

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.
So S is not a stable pairing. Contradiction. □

Notes: S - stable.

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.
So S is not a stable pairing. Contradiction. $\square$

Notes: S - stable. $(b^*, g^*) \in S$.

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.
So S is not a stable pairing. Contradiction. $\square$

Notes: S - stable. $(b^*, g^*) \in S$. But (b^*, g)

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.
So S is not a stable pairing. Contradiction. $\square$

Notes: S - stable. $(b^*, g^*) \in S$. But (b^*, g) is rogue couple!

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.
So S is not a stable pairing. Contradiction. $\square$

Notes: S - stable. $(b^*, g^*) \in S$. But (b^*, g) is rogue couple!

Used Well-Ordering principle...

Job Propose and Candidate Reject is optimal!

For jobs? For candidates?

Theorem: Job Propose and Reject produces a job-optimal pairing.

Proof:
Assume not: a job b is not paired with optimal candidate, g. Let b be first job that is rejected
by its optimal candidate g

Not optimal for $b \implies$
Another a stable pairing S where b and g are paired.

b^* - knocks b off of g's string $\implies$ g prefers b^* to b (partner in S)

By choice of b, b^* likes g at least as much as optimal candidate.

$\implies$ b^* prefers g to its partner g^* in S.

Rogue couple for S.
So S is not a stable pairing. Contradiction. $\square$

Notes: S - stable. $(b^*, g^*) \in S$. But (b^*, g) is rogue couple!

Used Well-Ordering principle...Induction.

Poll

What did proof use?

Poll

What did proof use?

(A) Algorithm.

Poll

What did proof use?

(A) Algorithm.
(B) Well ordering principle.

Poll

What did proof use?

(A) Algorithm.
(B) Well ordering principle.
(C) *First* job b, rejected by optimal candidate g

Poll

What did proof use?

(A) Algorithm.
(B) Well ordering principle.
(C) *First* job b, rejected by optimal candidate g
Job $b*$ was by optimal candidate.

Poll

What did proof use?

(A) Algorithm.
(B) Well ordering principle.
(C) *First* job b, rejected by optimal candidate g
    Job $b*$ was by optimal candidate.
      likes g a lot.

Poll

What did proof use?

(A) Algorithm.
(B) Well ordering principle.
(C) *First* job b, rejected by optimal candidate g
    Job $b*$ was by optimal candidate.
        likes g a lot.
(D) Contradiction.

Poll

What did proof use?

(A) Algorithm.
(B) Well ordering principle.
(C) *First* job b, rejected by optimal candidate g
    Job $b*$ was by optimal candidate.
        likes g a lot.
(D) Contradiction.
(E) Definition of optimal.

Poll

What did proof use?

(A) Algorithm.
(B) Well ordering principle.
(C) *First* job b, rejected by optimal candidate g
Job $b*$ was by optimal candidate.
likes g a lot.
(D) Contradiction.
(E) Definition of optimal.
There exists a better stable S.

Poll

What did proof use?

(A) Algorithm.
(B) Well ordering principle.
(C) *First* job b, rejected by optimal candidate g
Job $b*$ was by optimal candidate.
likes g a lot.
(D) Contradiction.
(E) Definition of optimal.
There exists a better stable S.
(F) S is not stable.

Poll

What did proof use?

(A) Algorithm.
(B) Well ordering principle.
(C) *First* job b, rejected by optimal candidate g
    Job $b*$ was by optimal candidate.
        likes g a lot.
(D) Contradiction.
(E) Definition of optimal.
  There exists a better stable S.
(F) S is not stable.

How about for candidates?

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

g prefers b to b^*.

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

g prefers b to b^*.

T is job optimal, so b prefers g to its partner in S.

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

g prefers b to b^*.

T is job optimal, so b prefers g to its partner in S.

(g, b) is Rogue couple for S

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

g prefers b to b^*.

T is job optimal, so b prefers g to its partner in S.

(g, b) is Rogue couple for S

S is not stable.

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

g prefers b to b^*.

T is job optimal, so b prefers g to its partner in S.

(g, b) is Rogue couple for S

S is not stable.

Contradiction.

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

g prefers b to b^*.

T is job optimal, so b prefers g to its partner in S.

(g, b) is Rogue couple for S

S is not stable.

Contradiction. □

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

g prefers b to b^*.

T is job optimal, so b prefers g to its partner in S.

(g, b) is Rogue couple for S

S is not stable.

Contradiction. □

Notes:

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

g prefers b to b^*.

T is job optimal, so b prefers g to its partner in S.

(g, b) is Rogue couple for S

S is not stable.

Contradiction. $\square$

Notes: Not really induction.

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

g prefers b to b^*.

T is job optimal, so b prefers g to its partner in S.

(g, b) is Rogue couple for S

S is not stable.

Contradiction. $\square$

Notes: Not really induction.
Structural statement: Job optimality

How about for candidates?

Theorem: Job Propose and Reject produces candidate-pessimal pairing.

T – pairing produced by JPR.

S – worse stable pairing for candidate g.

In T, (g, b) is pair.

In S, (g, b^*) is pair.

g prefers b to b^*.

T is job optimal, so b prefers g to its partner in S.

(g, b) is Rogue couple for S

S is not stable.

Contradiction. □

Notes: Not really induction.
Structural statement: Job optimality $\implies$ Candidate pessimality.

Quick Questions.

How does one make it better for candidates?

Quick Questions.

How does one make it better for candidates?

Propose and Reject - stable matching algorithm. One side proposes.

Quick Questions.

How does one make it better for candidates?

Propose and Reject - stable matching algorithm. One side proposes.

Jobs Propose $\implies$ job optimal.

Quick Questions.

How does one make it better for candidates?

Propose and Reject - stable matching algorithm. One side proposes.

Jobs Propose $\implies$ job optimal.

Candidates propose.

Quick Questions.

How does one make it better for candidates?

Propose and Reject - stable matching algorithm. One side proposes.

Jobs Propose $\implies$ job optimal.

Candidates propose. $\implies$ optimal for candidates.

Residency Matching..

Residency Matching..

The method was used to match residents to hospitals.

Residency Matching..

The method was used to match residents to hospitals.

Hospital optimal....

Residency Matching..

The method was used to match residents to hospitals.

Hospital optimal....

..until 1990's...

Residency Matching..

The method was used to match residents to hospitals.

Hospital optimal....

..until 1990's...Resident optimal.

Residency Matching..

The method was used to match residents to hospitals.

Hospital optimal....

..until 1990's...Resident optimal.

Another variation: couples.

Residency Matching..

The method was used to match residents to hospitals.

Hospital optimal....

..until 1990's...Resident optimal.

Another variation: couples.

Takeaways.

Analysis of cool algorithm with interesting goal: stability.

Takeaways.

Analysis of cool algorithm with interesting goal: stability.

“Economic”: different utilities.

Takeaways.

Analysis of cool algorithm with interesting goal: stability.

“Economic”: different utilities.

Definition of optimality: best utility in stable world.

Takeaways.

Analysis of cool algorithm with interesting goal: stability.

“Economic”: different utilities.

Definition of optimality: best utility in stable world.

Action gives better results for individuals but gives instability.

Takeaways.

Analysis of cool algorithm with interesting goal: stability.

“Economic”: different utilities.

Definition of optimality: best utility in stable world.

Action gives better results for individuals but gives instability.

Induction over steps of algorithm.

Takeaways.

Analysis of cool algorithm with interesting goal: stability.

“Economic”: different utilities.

Definition of optimality: best utility in stable world.

Action gives better results for individuals but gives instability.

Induction over steps of algorithm.

Proofs carefully use definition:

Takeaways.

Analysis of cool algorithm with interesting goal: stability.

"Economic": different utilities.

Definition of optimality: best utility in stable world.

Action gives better results for individuals but gives instability.

Induction over steps of algorithm.

Proofs carefully use definition:
Stability:
Improvement Lemma plus every day the job gets to choose.
Optimality proof:
Job Optimality:
contradiction of the existence of a better *stable* pairing.
that is, no rogue couple by improvement, job choice, and well ordering principle. Candidate Pessimality:
contradiction plus cuz job optimality implies better pairing.

Takeaways.

Analysis of cool algorithm with interesting goal: stability.

"Economic": different utilities.

Definition of optimality: best utility in stable world.

Action gives better results for individuals but gives instability.

Induction over steps of algorithm.

Proofs carefully use definition:
Stability:
Improvement Lemma plus every day the job gets to choose.
Optimality proof:
Job Optimality:
contradiction of the existence of a better *stable* pairing.
that is, no rogue couple by improvement, job choice, and well ordering principle.
Candidate Pessimality:
contradiction plus cuz job optimality implies better pairing.

Life Lesson: ask,

Takeaways.

Analysis of cool algorithm with interesting goal: stability.

"Economic": different utilities.

Definition of optimality: best utility in stable world.

Action gives better results for individuals but gives instability.

Induction over steps of algorithm.

Proofs carefully use definition:
Stability:
Improvement Lemma plus every day the job gets to choose.
Optimality proof:
Job Optimality:
contradiction of the existence of a better *stable* pairing.
that is, no rogue couple by improvement, job choice, and well ordering principle. Candidate Pessimality:
contradiction plus cuz job optimality implies better pairing.

Life Lesson: ask, you will do better

Takeaways.

Analysis of cool algorithm with interesting goal: stability.

"Economic": different utilities.

Definition of optimality: best utility in stable world.

Action gives better results for individuals but gives instability.

Induction over steps of algorithm.

Proofs carefully use definition:
Stability:
Improvement Lemma plus every day the job gets to choose.
Optimality proof:
Job Optimality:
contradiction of the existence of a better *stable* pairing.
that is, no rogue couple by improvement, job choice, and well ordering principle. Candidate Pessimality:
contradiction plus cuz job optimality implies better pairing.

Life Lesson: ask, you will do better even if rejection is hard.

Lecture 5: Graphs.

Graphs!

Lecture 5: Graphs.

Graphs!
 Definitions: model.

Lecture 5: Graphs.

Graphs!
- Definitions: model.
- Fact!

Lecture 5: Graphs.

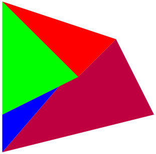
Graphs!
- Definitions: model.
- Fact!

Lecture 5: Graphs.

Graphs!

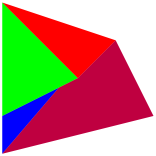
- Definitions: model.
- Fact!

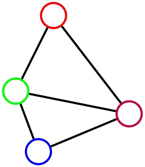
Map Coloring.



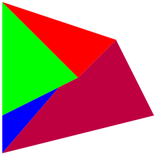
Map Coloring.

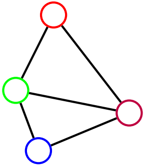
Map Coloring.





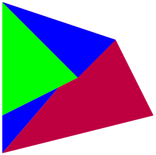
Map Coloring.

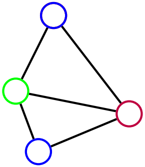




Fewer Colors?

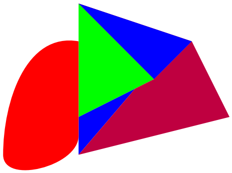
Map Coloring.

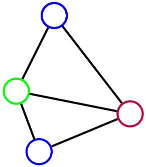




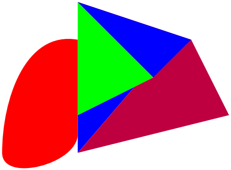
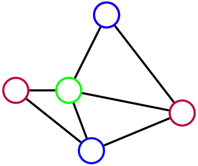
Yes! Three colors.

Map Coloring.

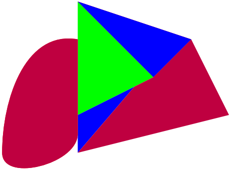


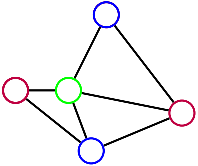


Map Coloring.

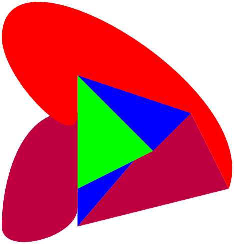
 

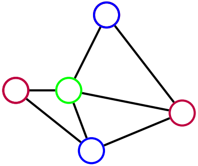
Map Coloring.



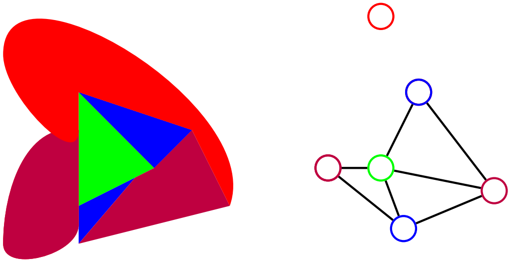


Map Coloring.

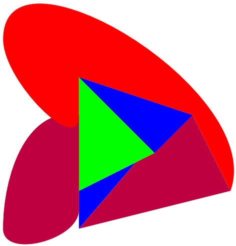


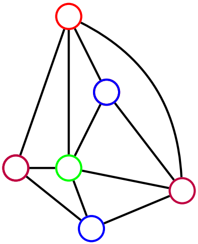


Map Coloring.

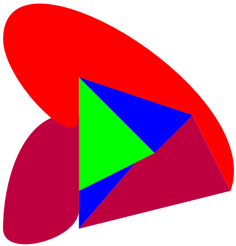


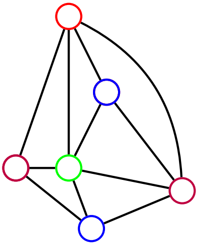
Map Coloring.





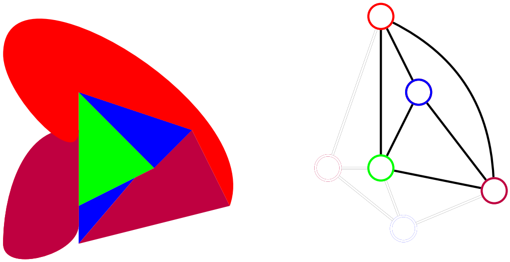
Map Coloring.



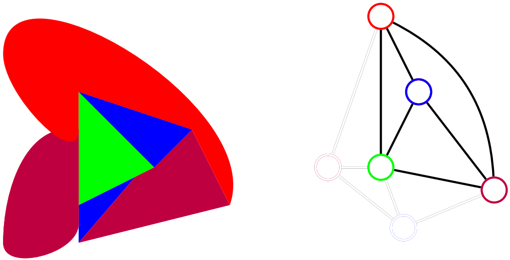


Fewer Colors?

Map Coloring.

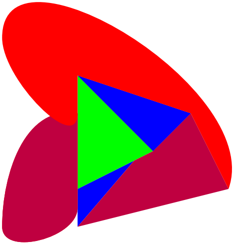


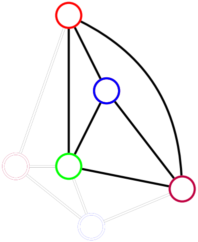
Map Coloring.



Four colors required!

Map Coloring.

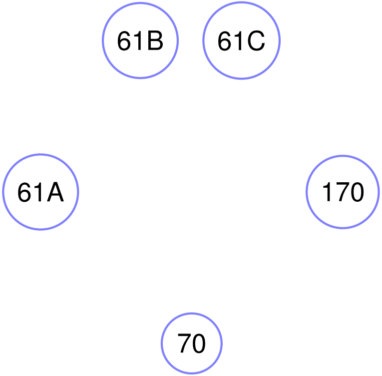




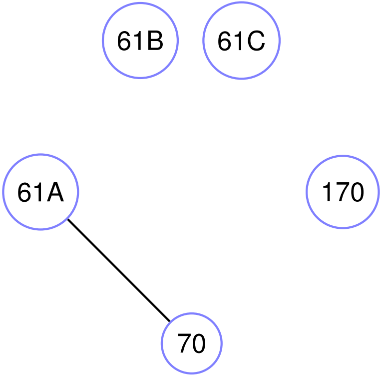
Four colors required!

Theorem: Four colors enough for maps on the plane.

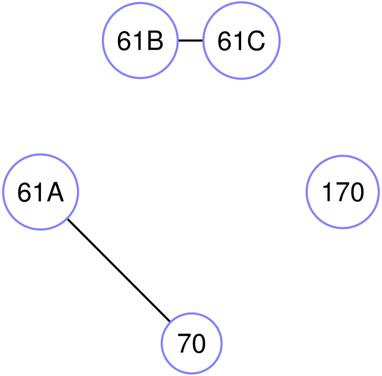
Scheduling: coloring.



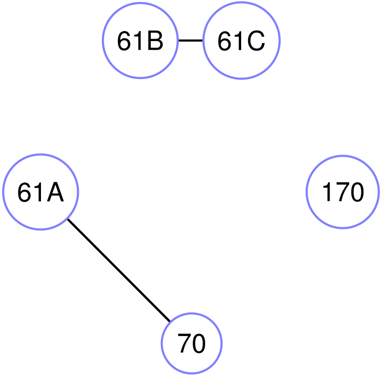
Scheduling: coloring.



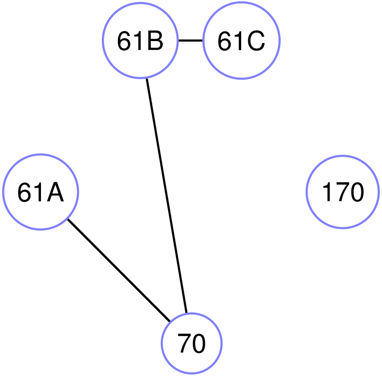
Scheduling: coloring.



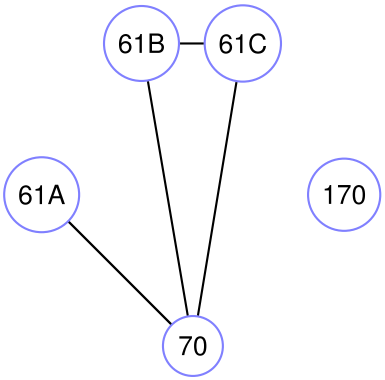
Scheduling: coloring.



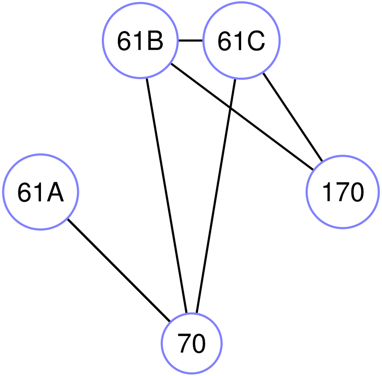
Scheduling: coloring.



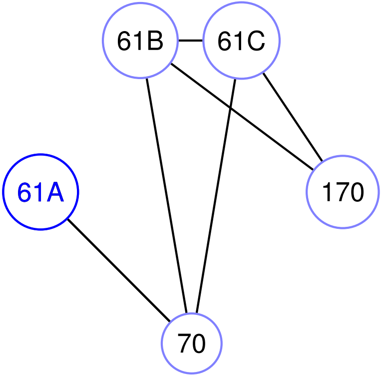
Scheduling: coloring.



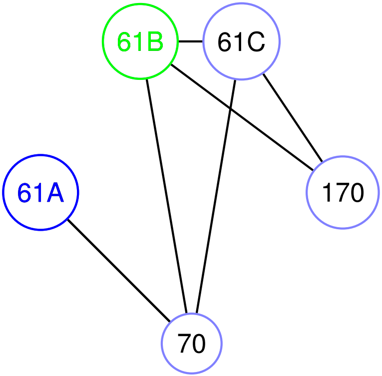
Scheduling: coloring.



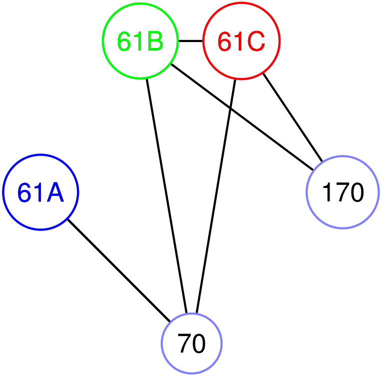
Scheduling: coloring.



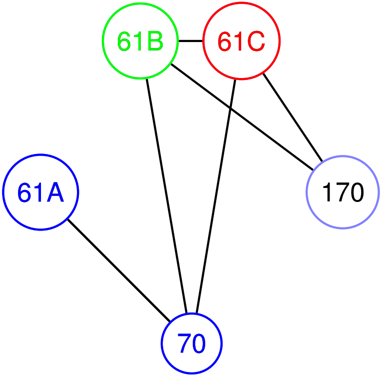
Scheduling: coloring.



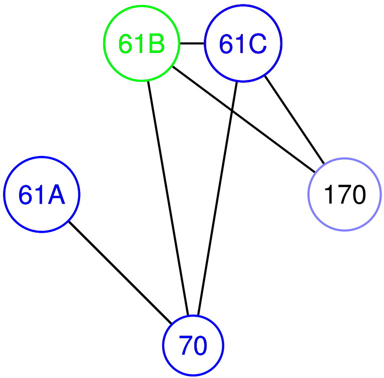
Scheduling: coloring.



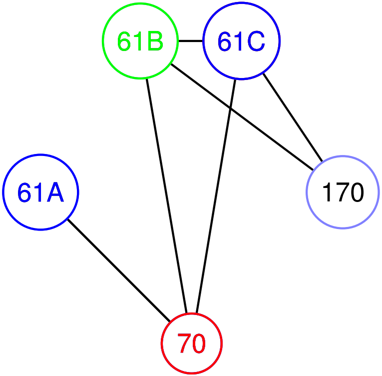
Scheduling: coloring.



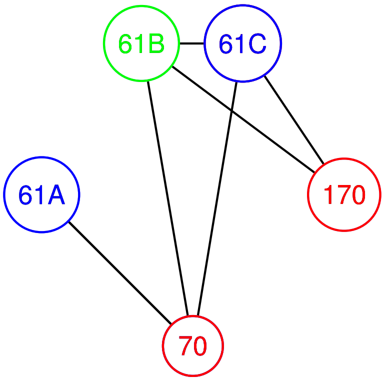
Scheduling: coloring.



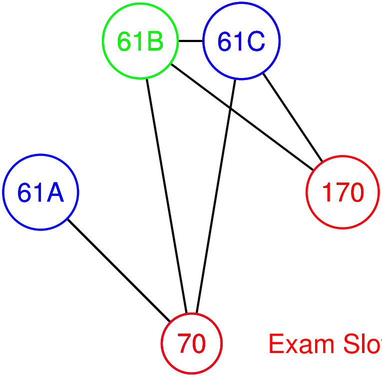
Scheduling: coloring.



Scheduling: coloring.



Scheduling: coloring.

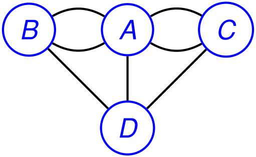


Exam Slot 1.

Exam Slot 2.

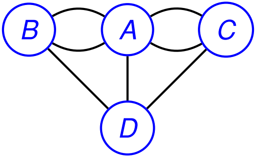
Exam Slot 3.

Graphs: formally.



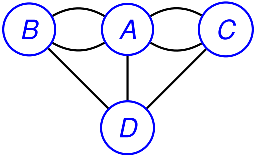
Graph:

Graphs: formally.



Graph: $G = (V, E)$.

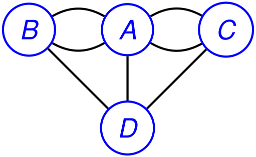
Graphs: formally.



Graph: $G = (V, E)$.

V - set of vertices.

Graphs: formally.

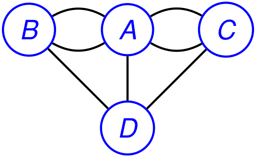


Graph: $G = (V, E)$.

V - set of vertices.

$\{A, B, C, D\}$

Graphs: formally.



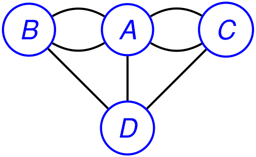
Graph: $G = (V, E)$.

V - set of vertices.

$\{A, B, C, D\}$

$E \subseteq V \times V$ -

Graphs: formally.



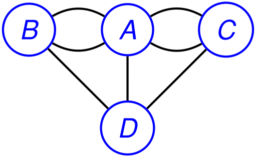
Graph: $G = (V, E)$.

V - set of vertices.

$\{A, B, C, D\}$

$E \subseteq V \times V$ - set of edges.

Graphs: formally.



Graph: $G = (V, E)$.

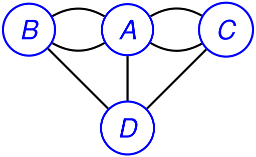
V - set of vertices.

$\{A, B, C, D\}$

$E \subseteq V \times V$ - set of edges.

$\{\{A, B\}$

Graphs: formally.



Graph: $G = (V, E)$.

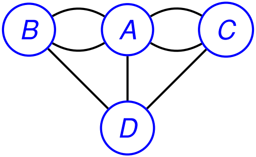
V - set of vertices.

$\{A, B, C, D\}$

$E \subseteq V \times V$ - set of edges.

$\{\{A, B\}, \{A, B\}$

Graphs: formally.



Graph: $G = (V, E)$.

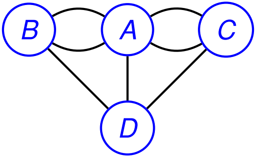
V - set of vertices.

$\{A, B, C, D\}$

$E \subseteq V \times V$ - set of edges.

$\{\{A, B\}, \{A, B\}, \{A, C\},$

Graphs: formally.



Graph: $G = (V, E)$.

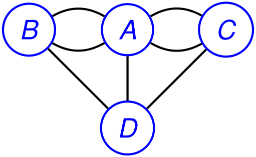
V - set of vertices.

$\{A, B, C, D\}$

$E \subseteq V \times V$ - set of edges.

$\{\{A, B\}, \{A, B\}, \{A, C\}, \{A, C\}, \{B, D\}, \{A, D\}, \{C, D\}\}$.

Graphs: formally.



Graph: $G = (V, E)$.

V - set of vertices.

$\{A, B, C, D\}$

$E \subseteq V \times V$ - set of edges.

$\{\{A, B\}, \{A, B\}, \{A, C\}, \{A, C\}, \{B, D\}, \{A, D\}, \{C, D\}\}$.

For CS 70, usually simple graphs.

Graphs: formally.

Graph: $G = (V, E)$.

V - set of vertices.

$\{A, B, C, D\}$

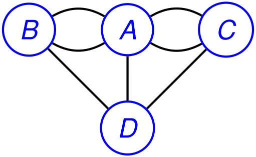
$E \subseteq V \times V$ - set of edges.

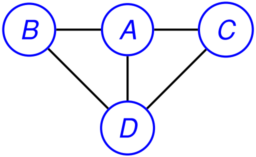
$\{\{A, B\}, \{A, B\}, \{A, C\}, \{A, C\}, \{B, D\}, \{A, D\}, \{C, D\}\}$.

For CS 70, usually simple graphs.

No parallel edges.

Graphs: formally.





Graph: $G = (V, E)$.

V - set of vertices.

$\{A, B, C, D\}$

$E \subseteq V \times V$ - set of edges.

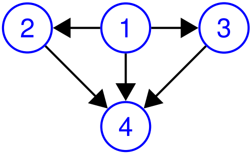
$\{\{A, B\}, \{A, B\}, \{A, C\}, \{A, C\}, \{B, D\}, \{A, D\}, \{C, D\}\}$.

For CS 70, usually simple graphs.

No parallel edges.

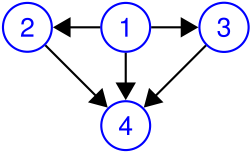
Multigraph above.

Directed Graphs



$G = (V, E)$.

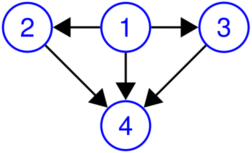
Directed Graphs



$G = (V, E)$.

V - set of vertices.

Directed Graphs

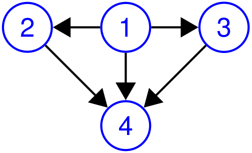


$G = (V, E)$.

V - set of vertices.

$\{1, 2, 3, 4\}$

Directed Graphs



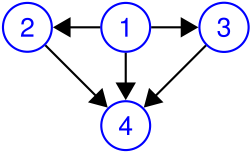
$G = (V, E)$.

V - set of vertices.

$\{1, 2, 3, 4\}$

E ordered pairs of vertices.

Directed Graphs



$G = (V, E)$.

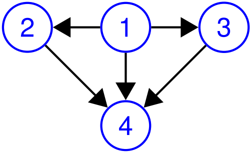
V - set of vertices.

$\{1, 2, 3, 4\}$

E ordered pairs of vertices.

$\{(1, 2),$

Directed Graphs



$G = (V, E)$.

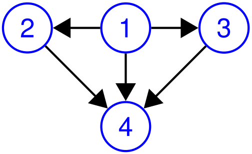
V - set of vertices.

$\{1, 2, 3, 4\}$

E ordered pairs of vertices.

$\{(1, 2), (1, 3),$

Directed Graphs



$G = (V, E)$.

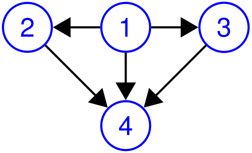
V - set of vertices.

$\{1, 2, 3, 4\}$

E ordered pairs of vertices.

$\{(1, 2), (1, 3), (1, 4),$

Directed Graphs



$G = (V, E)$.

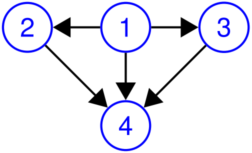
V - set of vertices.

$\{1, 2, 3, 4\}$

E ordered pairs of vertices.

$\{(1,2), (1,3), (1,4), (2,4), (3,4)\}$

Directed Graphs



$G = (V, E)$.

V - set of vertices.

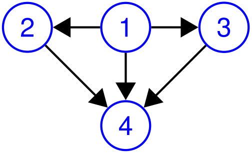
$\{1, 2, 3, 4\}$

E ordered pairs of vertices.

$\{(1,2),(1,3),(1,4),(2,4),(3,4)\}$

One way streets.

Directed Graphs



$G = (V, E)$.

V - set of vertices.

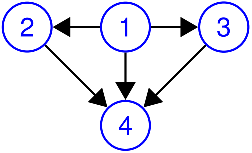
$\{1, 2, 3, 4\}$

E ordered pairs of vertices.

$\{(1,2), (1,3), (1,4), (2,4), (3,4)\}$

One way streets.
Tournament:

Directed Graphs



$G = (V, E)$.

V - set of vertices.

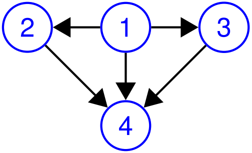
$\{1, 2, 3, 4\}$

E ordered pairs of vertices.

$\{(1,2), (1,3), (1,4), (2,4), (3,4)\}$

One way streets.
Tournament: 1 beats 2,

Directed Graphs



$G = (V, E)$.

V - set of vertices.

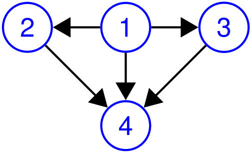
$\{1, 2, 3, 4\}$

E ordered pairs of vertices.

$\{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence:

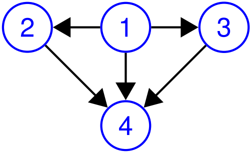
Directed Graphs



$G = (V, E)$.
V - set of vertices.
$\{1, 2, 3, 4\}$
E ordered pairs of vertices.
$\{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence: 1 is before 2,

Directed Graphs



$G = (V, E)$.

V - set of vertices.

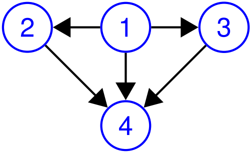
$\{1, 2, 3, 4\}$

E ordered pairs of vertices.

$\{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence: 1 is before 2, ..

Directed Graphs



$G = (V, E)$.

V - set of vertices.

$\{1, 2, 3, 4\}$

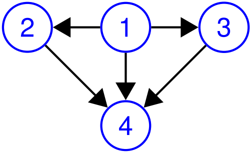
E ordered pairs of vertices.

$\{(1,2),(1,3),(1,4),(2,4),(3,4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence: 1 is before 2, ..

Social Network:

Directed Graphs

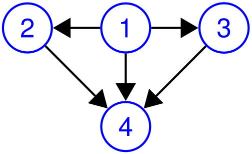


$G = (V, E)$.
V - set of vertices.
$\{1, 2, 3, 4\}$
E ordered pairs of vertices.
$\{(1,2), (1,3), (1,4), (2,4), (3,4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence: 1 is before 2, ..

Social Network: Directed?

Directed Graphs

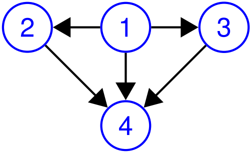


$G = (V, E)$.
V - set of vertices.
$\{1, 2, 3, 4\}$
E ordered pairs of vertices.
$\{(1,2),(1,3),(1,4),(2,4),(3,4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence: 1 is before 2, ..

Social Network: Directed? Undirected?

Directed Graphs

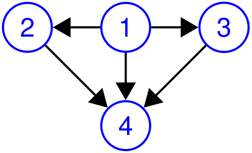


$G = (V, E)$.
V - set of vertices.
$\{1, 2, 3, 4\}$
E ordered pairs of vertices.
$\{(1,2),(1,3),(1,4),(2,4),(3,4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence: 1 is before 2, ..

Social Network: Directed? Undirected?
Friends.

Directed Graphs

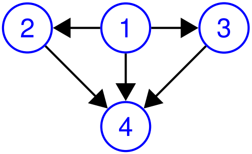


$G = (V, E)$.
V - set of vertices.
$\{1, 2, 3, 4\}$
E ordered pairs of vertices.
$\{(1,2), (1,3), (1,4), (2,4), (3,4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence: 1 is before 2, ..

Social Network: Directed? Undirected?
Friends. Undirected.

Directed Graphs

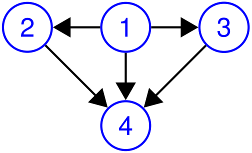


$G = (V, E)$.
V - set of vertices.
$\{1, 2, 3, 4\}$
E ordered pairs of vertices.
$\{(1,2),(1,3),(1,4),(2,4),(3,4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence: 1 is before 2, ..

Social Network: Directed? Undirected?
Friends. Undirected.
Likes.

Directed Graphs

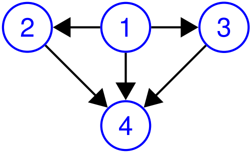


$G = (V, E)$.
V - set of vertices.
$\{1, 2, 3, 4\}$
E ordered pairs of vertices.
$\{(1,2),(1,3),(1,4),(2,4),(3,4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence: 1 is before 2, ..

Social Network: Directed? Undirected?
Friends. Undirected.
Likes. Directed.

Directed Graphs



$G = (V, E)$.
V - set of vertices.
$\{1, 2, 3, 4\}$
E ordered pairs of vertices.
$\{(1,2), (1,3), (1,4), (2,4), (3,4)\}$

One way streets.
Tournament: 1 beats 2, ...
Precedence: 1 is before 2, ..

Social Network: Directed? Undirected?
Friends. Undirected.
Likes. Directed.

Graph Concepts and Definitions.

Graph: $G = (V, E)$

Graph Concepts and Definitions.

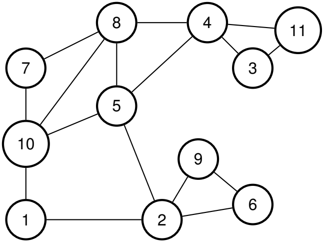
Graph: $G = (V, E)$

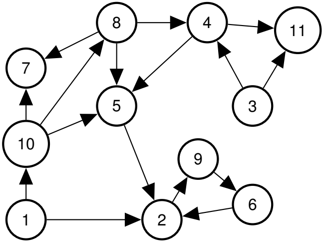
neighbors, adjacent, degree, incident, in-degree, out-degree

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree



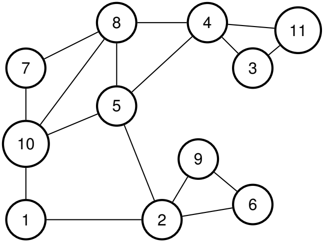


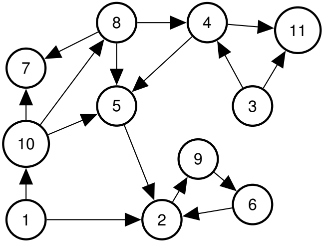
Neighbors of 10?

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree



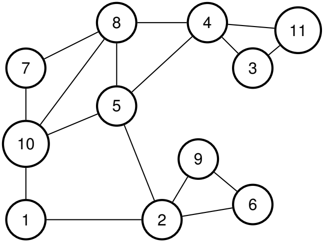


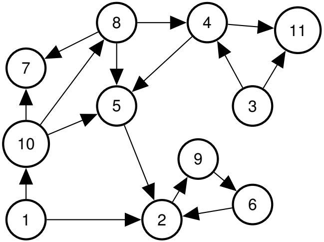
Neighbors of 10? 1,

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree



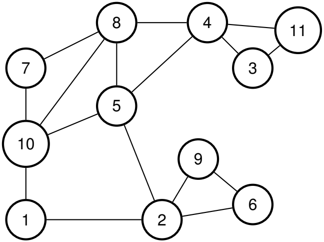


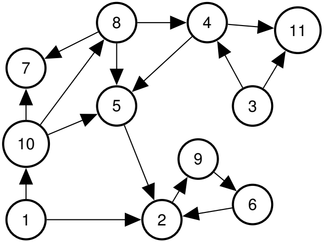
Neighbors of 10? 1,5,

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree



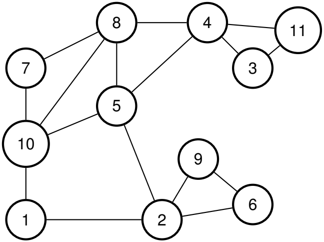


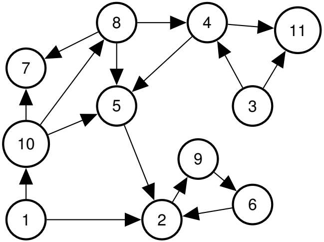
Neighbors of 10? 1,5,7,

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree



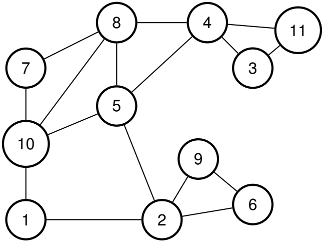


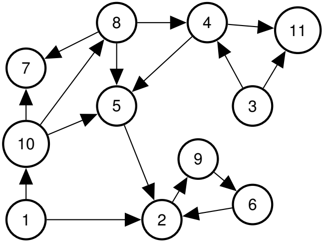
Neighbors of 10? 1,5,7, 8.

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





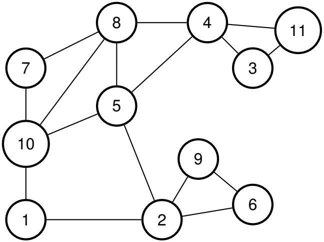
Neighbors of 10? 1,5,7, 8.

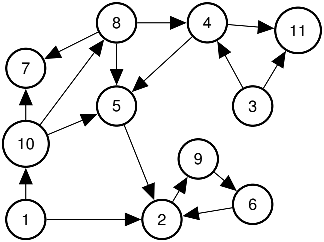
u is neighbor of v if $\{u, v\} \in E$.

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Neighbors of 10? 1,5,7, 8.

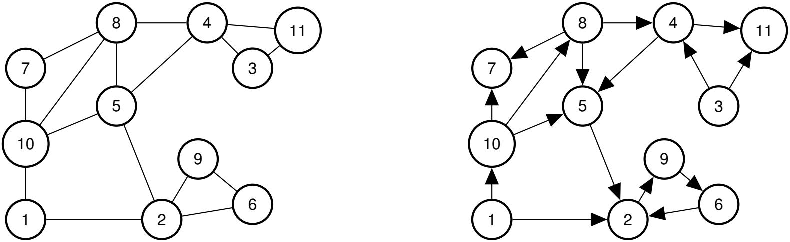
u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree



Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

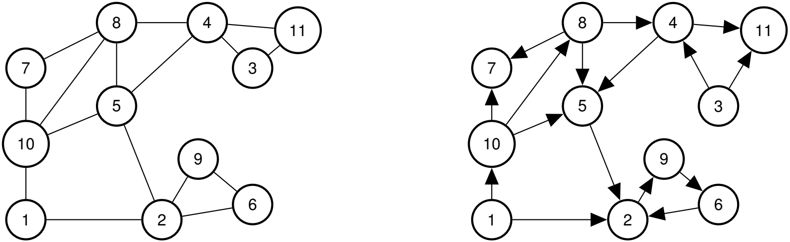
Edge $\{u, v\}$ is incident to u and v.

Degree of vertex 1?

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree



Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

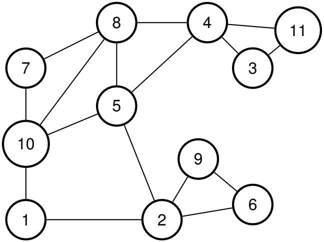
Edge $\{u, v\}$ is incident to u and v.

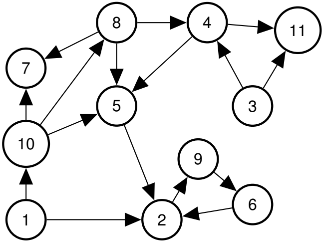
Degree of vertex 1? 2

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

Edge $\{u, v\}$ is incident to u and v.

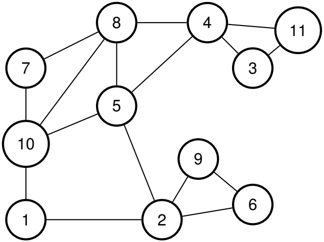
Degree of vertex 1? 2

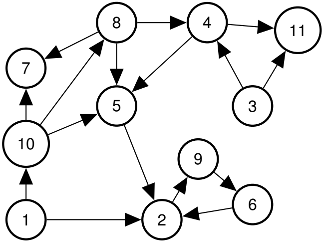
Degree of vertex u is number of incident edges.

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

Edge $\{u, v\}$ is incident to u and v.

Degree of vertex 1? 2

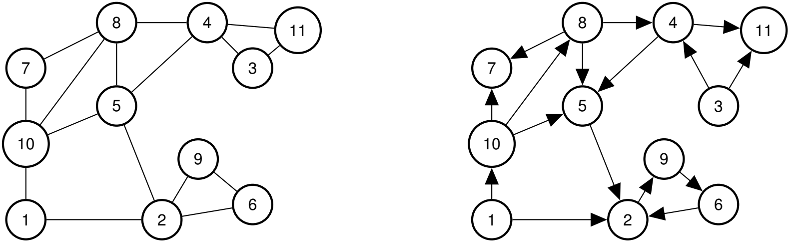
Degree of vertex u is number of incident edges.

Equals number of neighbors in simple graph.

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree



Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

Edge $\{u, v\}$ is incident to u and v.

Degree of vertex 1? 2

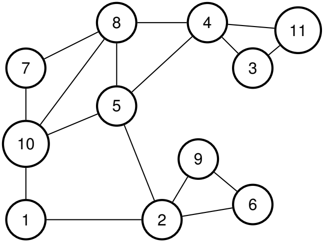
Degree of vertex u is number of incident edges.

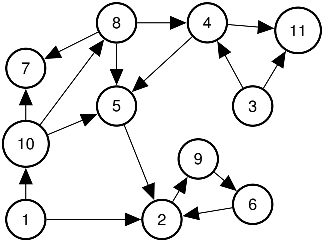
Equals number of neighbors in simple graph.

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

Edge $\{u, v\}$ is incident to u and v.

Degree of vertex 1? 2

Degree of vertex u is number of incident edges.

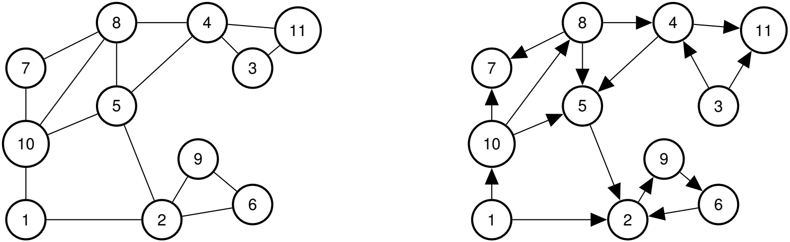
Equals number of neighbors in simple graph.

Directed graph?

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree



Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

Edge $\{u, v\}$ is incident to u and v.

Degree of vertex 1? 2

Degree of vertex u is number of incident edges.

Equals number of neighbors in simple graph.

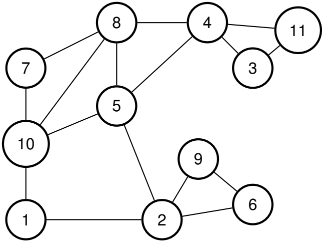
Directed graph?

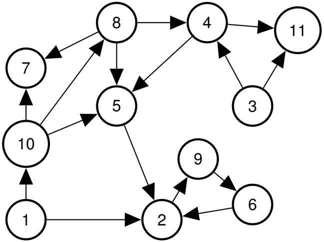
In-degree of 10?

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

Edge $\{u, v\}$ is incident to u and v.

Degree of vertex 1? 2

Degree of vertex u is number of incident edges.

Equals number of neighbors in simple graph.

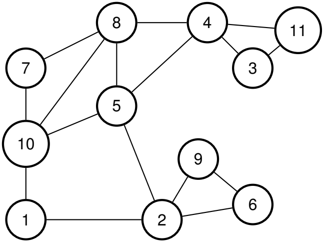
Directed graph?

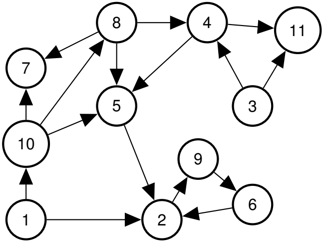
In-degree of 10? 1

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

Edge $\{u, v\}$ is incident to u and v.

Degree of vertex 1? 2

Degree of vertex u is number of incident edges.

Equals number of neighbors in simple graph.

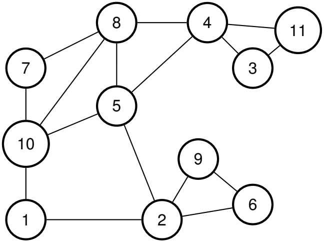
Directed graph?

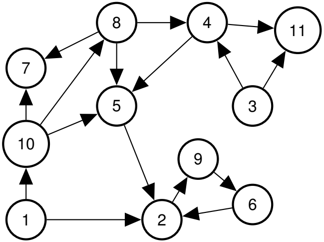
In-degree of 10? 1 Out-degree of 10?

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

Edge $\{u, v\}$ is incident to u and v.

Degree of vertex 1? 2

Degree of vertex u is number of incident edges.

Equals number of neighbors in simple graph.

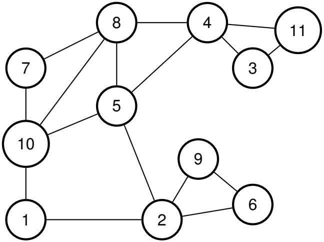
Directed graph?

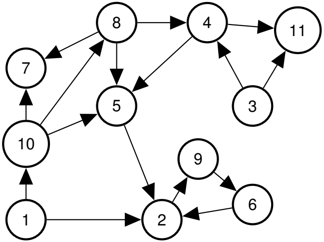
In-degree of 10? 1 Out-degree of 10? 3

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Neighbors of 10? 1,5,7, 8.

u is neighbor of v if $\{u, v\} \in E$.

Edge $\{10, 5\}$ is incident to vertex 10 and vertex 5.

Edge $\{u, v\}$ is incident to u and v.

Degree of vertex 1? 2

Degree of vertex u is number of incident edges.

Equals number of neighbors in simple graph.

Directed graph?

In-degree of 10? 1   Out-degree of 10? 3

Graph Concepts and Definitions.

Graph: $G = (V, E)$

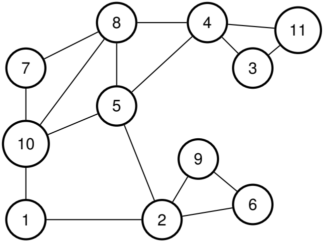
Graph Concepts and Definitions.

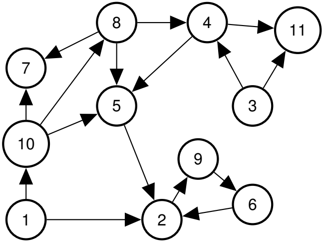
Graph: $G = (V, E)$
neighbors, adjacent, degree, incident, in-degree, out-degree

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





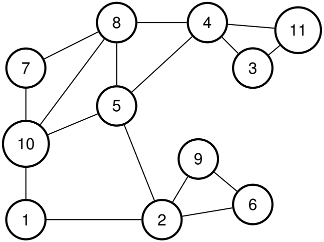
Edge $(8, 5)$ **is incident to:**

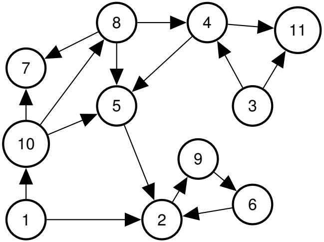
(A) Vertex 8.
(B) Vertex 5.
(C) Edge $(8, 5)$.
(D) Edge $(8, 4)$.
(E) Vertex 10.

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Edge $(8, 5)$ **is incident to:**

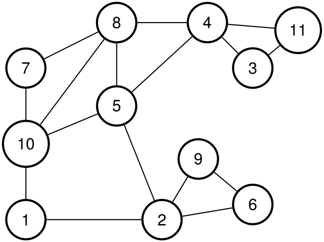
(A) Vertex 8.
(B) Vertex 5.
(C) Edge $(8, 5)$.
(D) Edge $(8, 4)$.
(E) Vertex 10.

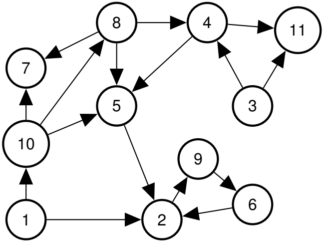
(A) and (B) are true.

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Edge $(8, 5)$ **is incident to:**

(A) Vertex 8.
(B) Vertex 5.
(C) Edge $(8, 5)$.
(D) Edge $(8, 4)$.
(E) Vertex 10.

(A) and (B) are true.

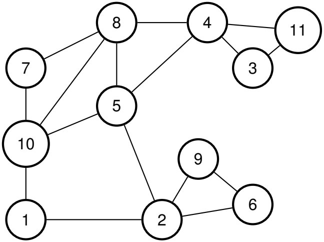
The degree of a vertex is:

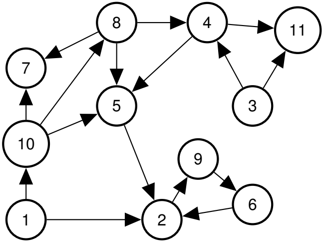
(A) The number of edges incident to it.
(B) The number of neighbors of v.
(C) Is the number of vertices in its connected component.

Graph Concepts and Definitions.

Graph: $G = (V, E)$

neighbors, adjacent, degree, incident, in-degree, out-degree





Edge $(8,5)$ is incident to:

(A) Vertex 8.
(B) Vertex 5.
(C) Edge $(8,5)$.
(D) Edge $(8,4)$.
(E) Vertex 10.

(A) and (B) are true.

The degree of a vertex is:

(A) The number of edges incident to it.
(B) The number of neighbors of v.
(C) Is the number of vertices in its connected component.

(A) and (B) are true.

Sum of degrees?

The sum of the vertex degrees is equal to

Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.

Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.

Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

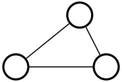
Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.

Not (A)!



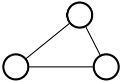
Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.

Not (A)! Triangle.

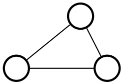


Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.



Not (A)! Triangle.
Not (B)!

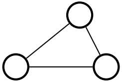
Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.

Not (A)! Triangle.
Not (B)! Triangle.



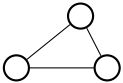
Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.

Not (A)! Triangle.
Not (B)! Triangle.

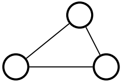


Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.



Not (A)! Triangle.
Not (B)! Triangle.

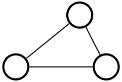
What?

Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.



Not (A)! Triangle.
Not (B)! Triangle.

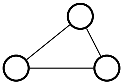
What? For triangle number of edges is 3, the sum of degrees is 6.

Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.



Not (A)! Triangle.
Not (B)! Triangle.

What? For triangle number of edges is 3, the sum of degrees is 6.

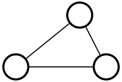
Could sum always be...

Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.



Not (A)! Triangle.
Not (B)! Triangle.

What? For triangle number of edges is 3, the sum of degrees is 6.

Could sum always be...

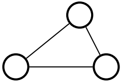
(A) $2|E|$? ..

Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.



Not (A)! Triangle.
Not (B)! Triangle.

What? For triangle number of edges is 3, the sum of degrees is 6.

Could sum always be...

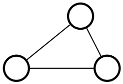
(A) $2|E|$? ..
(B) $2|V|$?

Sum of degrees?

The sum of the vertex degrees is equal to

(A) the total number of vertices, $|V|$.
(B) the total number of edges, $|E|$.
(C) What?

(A) and (B) are false. (C) is a fine response to a poll with no correct answers.



Not (A)! Triangle.
Not (B)! Triangle.

What? For triangle number of edges is 3, the sum of degrees is 6.

Could sum always be...

(A) $2|E|$? ..
(B) $2|V|$?
(A) is true.

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute?

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u,v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Total Incidences?

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Total Incidences? $|E|$ edges, 2 each. $\rightarrow 2|E|$

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Total Incidences? $|E|$ edges, 2 each. $\rightarrow 2|E|$

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Total Incidences? $|E|$ edges, 2 each. $\rightarrow 2|E|$

What is degree v?

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Total Incidences? $|E|$ edges, 2 each. $\rightarrow 2|E|$

What is degree v? Incidences corresponding to v!

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Total Incidences? $|E|$ edges, 2 each. $\rightarrow 2|E|$

What is degree v? Incidences corresponding to v!

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Total Incidences? $|E|$ edges, 2 each. $\rightarrow 2|E|$

What is degree v? Incidences corresponding to v!

Total Incidences?

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Total Incidences? $|E|$ edges, 2 each. $\rightarrow 2|E|$

What is degree v? Incidences corresponding to v!

Total Incidences? The sum over vertices of degrees!

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Total Incidences? $|E|$ edges, 2 each. $\rightarrow 2|E|$

What is degree v? Incidences corresponding to v!

Total Incidences? The sum over vertices of degrees!

Quick Proof of an Equality.

The sum of the vertex degrees is equal to ??

Recall:

edge, (u, v), is incident to endpoints, u and v.

degree of u number of edges incident to u

Let's count incidences in two ways.

How many incidences does each edge contribute? 2.

Total Incidences? $|E|$ edges, 2 each. $\rightarrow 2|E|$

What is degree v? Incidences corresponding to v!

Total Incidences? The sum over vertices of degrees!

Thm: Sum of vertex degress is $2|E|$.

Poll: Proof of “handshake” lemma.

What’s true?

(A) The number of edge-vertex incidences for an edge e is 2.
(B) The total number of edge-vertex incidences is $|V|$.
(C) The total number of edge-vertex incidences is $2|E|$.
(D) The number of edge-vertex incidences for a vertex v is its degree.
(E) The sum of degrees is $2|E|$.
(F) The total number of edge-vertex incidences is the sum of the degrees.

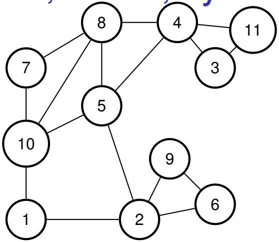
Poll: Proof of “handshake” lemma.

What’s true?

(A) The number of edge-vertex incidences for an edge e is 2.
(B) The total number of edge-vertex incidences is $|V|$.
(C) The total number of edge-vertex incidences is $2|E|$.
(D) The number of edge-vertex incidences for a vertex v is its degree.
(E) The sum of degrees is $2|E|$.
(F) The total number of edge-vertex incidences is the sum of the degrees.

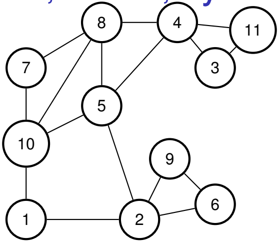
(A),(C), (D), (E), and (F).

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

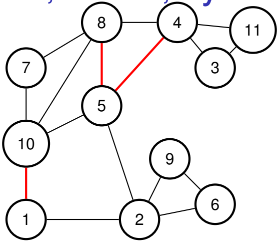
Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path?

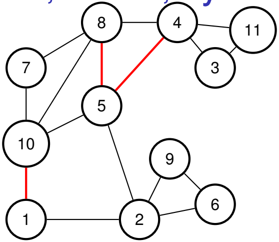
Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$?

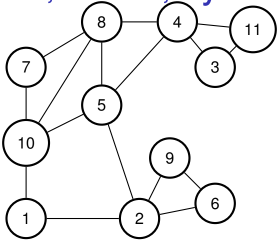
Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Paths, walks, cycles, tour.

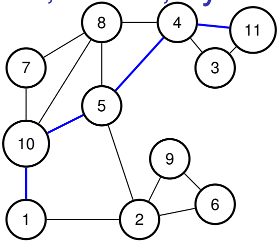


A path in a graph is a sequence of edges.

Path? $\{1, 10\}$, $\{8, 5\}$, $\{4, 5\}$? No!

Path?

Paths, walks, cycles, tour.

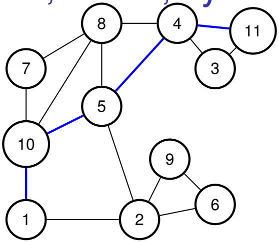


A path in a graph is a sequence of edges.

Path? $\{1,10\}, \{8,5\}, \{4,5\}$? No!

Path? $\{1,10\}, \{10,5\}, \{5,4\}, \{4,11\}$?

Paths, walks, cycles, tour.

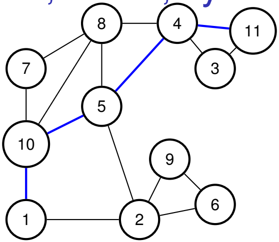


A path in a graph is a sequence of edges.

Path? $\{1, 10\}, \{8, 5\}, \{4, 5\}$? No!

Path? $\{1, 10\}, \{10, 5\}, \{5, 4\}, \{4, 11\}$? Yes!

Paths, walks, cycles, tour.



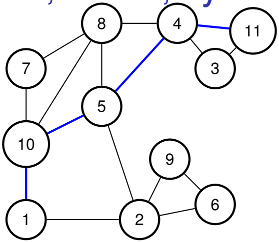
A path in a graph is a sequence of edges.

Path? $\{1,10\}, \{8,5\}, \{4,5\}$? No!

Path? $\{1,10\}, \{10,5\}, \{5,4\}, \{4,11\}$? Yes!

Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

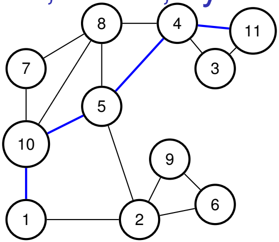
Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check!

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

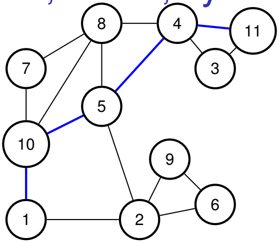
Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path?

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

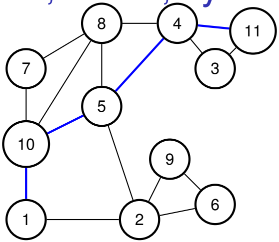
Path? $\{1,10\}, \{8,5\}, \{4,5\}$? No!

Path? $\{1,10\}, \{10,5\}, \{5,4\}, \{4,11\}$? Yes!

Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.

Quick Check! Length of path? k vertices

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

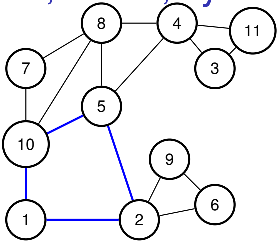
Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

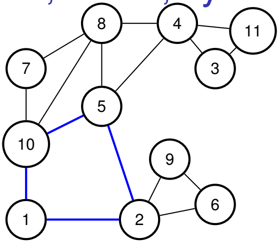
Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1)

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

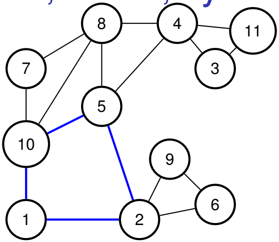
Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1) Length of cycle?

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

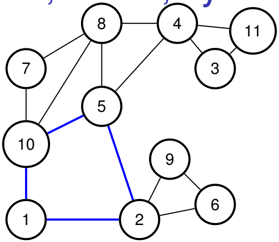
Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1}, v_1) Length of cycle? $k-1$ vertices and edges!

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

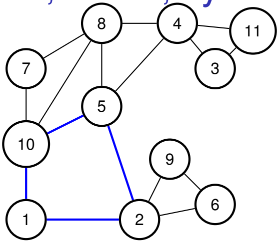
Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1) Length of cycle? $k-1$ vertices and edges!

Path is usually simple.

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

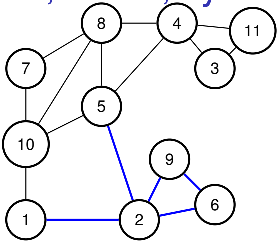
Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1) Length of cycle? $k-1$ vertices and edges!

Path is usually simple. No repeated vertex!

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

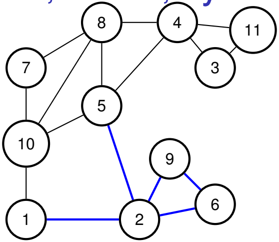
Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1) Length of cycle? $k-1$ vertices and edges!

Path is usually simple. No repeated vertex!

Walk is sequence of edges with possible repeated vertex or edge.

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

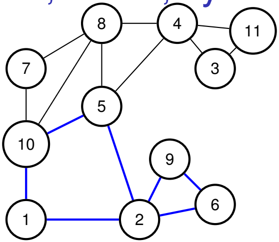
Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1) Length of cycle? $k-1$ vertices and edges!

Path is usually simple. No repeated vertex!

Walk is sequence of edges with possible repeated vertex or edge.

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

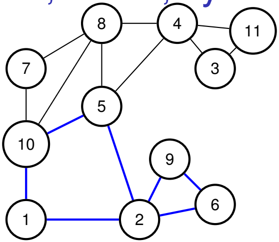
Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1) Length of cycle? $k-1$ vertices and edges!

Path is usually simple. No repeated vertex!

Walk is sequence of edges with possible repeated vertex or edge.

Tour is walk that starts and ends at the same node.

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

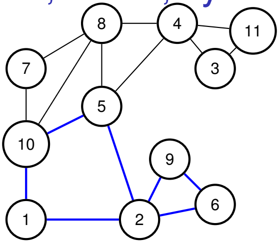
Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1) Length of cycle? $k-1$ vertices and edges!

Path is usually simple. No repeated vertex!

Walk is sequence of edges with possible repeated vertex or edge.

Tour is walk that starts and ends at the same node.

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1) Length of cycle? $k-1$ vertices and edges!

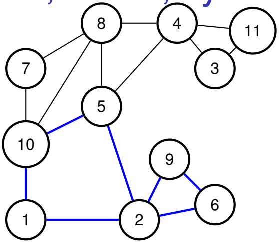
Path is usually simple. No repeated vertex!

Walk is sequence of edges with possible repeated vertex or edge.

Tour is walk that starts and ends at the same node.

Quick Check!

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1) Length of cycle? $k-1$ vertices and edges!

Path is usually simple. No repeated vertex!

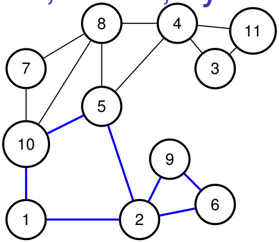
Walk is sequence of edges with possible repeated vertex or edge.

Tour is walk that starts and ends at the same node.

Quick Check!

Path is to Walk as Cycle is to ??

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1,v_2),(v_2,v_3),\ldots(v_{k-1},v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1},v_1) Length of cycle? $k-1$ vertices and edges!

Path is usually simple. No repeated vertex!

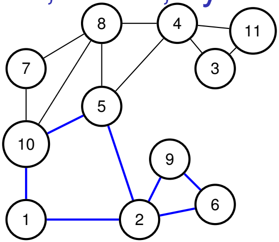
Walk is sequence of edges with possible repeated vertex or edge.

Tour is walk that starts and ends at the same node.

Quick Check!

Path is to Walk as Cycle is to ?? Tour!

Paths, walks, cycles, tour.



A path in a graph is a sequence of edges.

Path? $\{1,10\}$, $\{8,5\}$, $\{4,5\}$? No!

Path? $\{1,10\}$, $\{10,5\}$, $\{5,4\}$, $\{4,11\}$? Yes!

Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.

Quick Check! Length of path? k vertices or $k-1$ edges.

Cycle: Path from v_1 to v_{k-1}, + edge (v_{k-1}, v_1) Length of cycle? $k-1$ vertices and edges!

Path is usually simple. No repeated vertex!

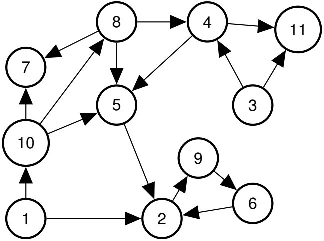
Walk is sequence of edges with possible repeated vertex or edge.

Tour is walk that starts and ends at the same node.

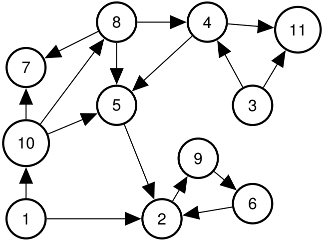
Quick Check!

Path is to Walk as Cycle is to ?? Tour!

Directed Paths.

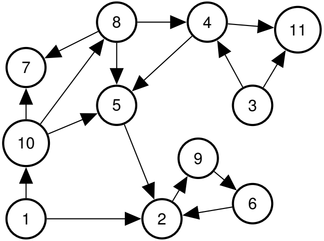


Directed Paths.



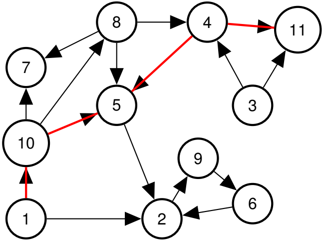
Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.

Directed Paths.



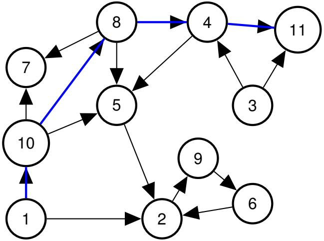
Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.

Directed Paths.



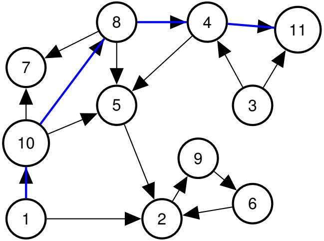
Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.

Directed Paths.



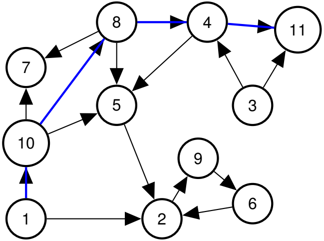
Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.

Directed Paths.



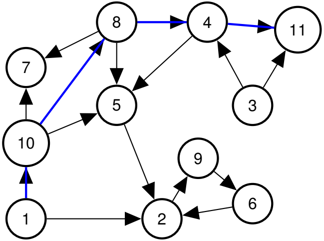
Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.
Paths,

Directed Paths.



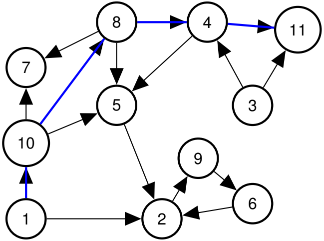
Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.
Paths, walks,

Directed Paths.



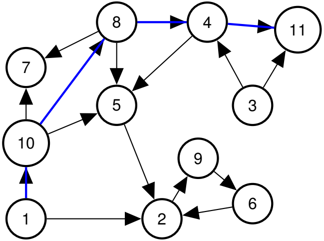
Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.
Paths, walks, cycles,

Directed Paths.



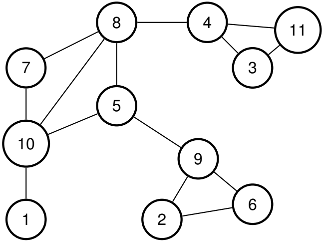
Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.
Paths, walks, cycles, tours

Directed Paths.



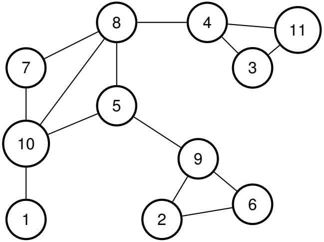
Path: $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$.
Paths, walks, cycles, tours ... are analagous to undirected now.

Connectivity: undirected graph.



u and *v* are connected if there is a path between *u* and *v*.

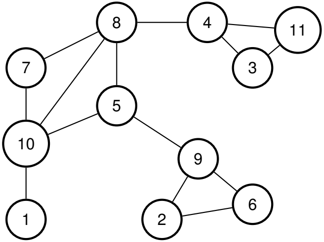
Connectivity: undirected graph.



u and *v* are connected if there is a path between *u* and *v*.

A connected graph is a graph where all pairs of vertices are connected.

Connectivity: undirected graph.

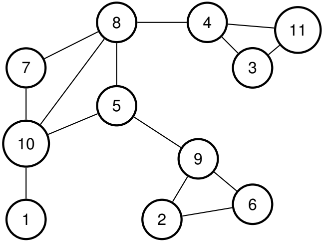


u and *v* are connected if there is a path between *u* and *v*.

A connected graph is a graph where all pairs of vertices are connected.

If one vertex *x* is connected to every other vertex.

Connectivity: undirected graph.

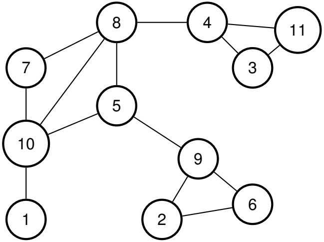


u and *v* are connected if there is a path between *u* and *v*.

A connected graph is a graph where all pairs of vertices are connected.

If one vertex *x* is connected to every other vertex.
Is graph connected?

Connectivity: undirected graph.

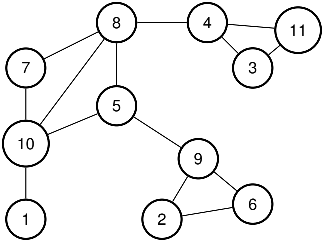


u and *v* are connected if there is a path between *u* and *v*.

A connected graph is a graph where all pairs of vertices are connected.

If one vertex *x* is connected to every other vertex.
Is graph connected? Yes?

Connectivity: undirected graph.

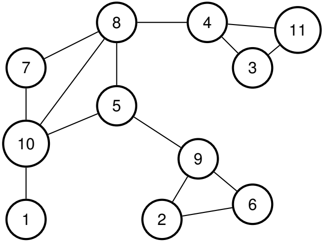


u and v are connected if there is a path between u and v.

A connected graph is a graph where all pairs of vertices are connected.

If one vertex x is connected to every other vertex.
Is graph connected? Yes? No?

Connectivity: undirected graph.



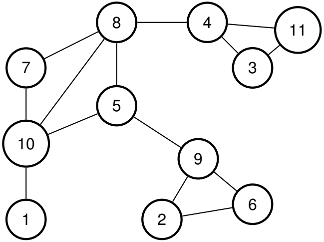
u and *v* are connected if there is a path between *u* and *v*.

A connected graph is a graph where all pairs of vertices are connected.

If one vertex *x* is connected to every other vertex.
Is graph connected? Yes? No?

Proof:

Connectivity: undirected graph.



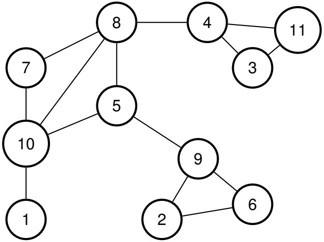
u and *v* are connected if there is a path between *u* and *v*.

A connected graph is a graph where all pairs of vertices are connected.

If one vertex *x* is connected to every other vertex.
Is graph connected? Yes? No?

Proof: Use path from *u* to *x* and then from *x* to *v*.

Connectivity: undirected graph.



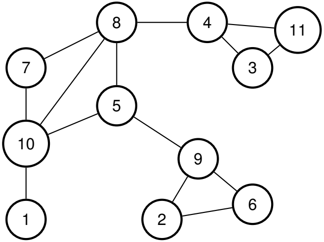
u and *v* are connected if there is a path between *u* and *v*.

A connected graph is a graph where all pairs of vertices are connected.

If one vertex *x* is connected to every other vertex.
Is graph connected? Yes? No?

Proof: Use path from *u* to *x* and then from *x* to *v*. □

Connectivity: undirected graph.



u and *v* are connected if there is a path between *u* and *v*.

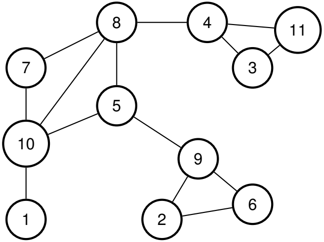
A connected graph is a graph where all pairs of vertices are connected.

If one vertex *x* is connected to every other vertex.
Is graph connected? Yes? No?

Proof: Use path from *u* to *x* and then from *x* to *v*. □

May not be simple!

Connectivity: undirected graph.



u and *v* are connected if there is a path between *u* and *v*.

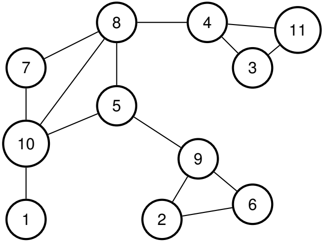
A connected graph is a graph where all pairs of vertices are connected.

If one vertex *x* is connected to every other vertex.
Is graph connected? Yes? No?

Proof: Use path from *u* to *x* and then from *x* to *v*. □

May not be simple!
Either modify definition to walk.

Connectivity: undirected graph.



u and *v* are connected if there is a path between *u* and *v*.

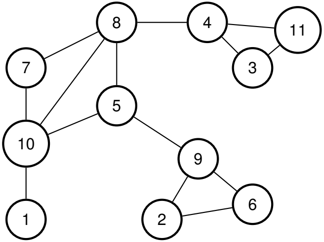
A connected graph is a graph where all pairs of vertices are connected.

If one vertex *x* is connected to every other vertex.
Is graph connected? Yes? No?

Proof: Use path from *u* to *x* and then from *x* to *v*. □

May not be simple!
Either modify definition to walk.
Or cut out cycles.

Connectivity: undirected graph.



u and *v* are connected if there is a path between *u* and *v*.

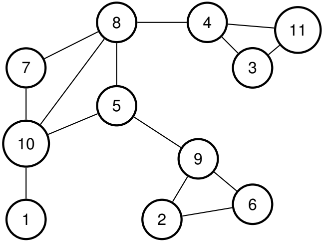
A connected graph is a graph where all pairs of vertices are connected.

If one vertex *x* is connected to every other vertex.
Is graph connected? Yes? No?

Proof: Use path from *u* to *x* and then from *x* to *v*. □

May not be simple!
Either modify definition to walk.
Or cut out cycles. .

Connectivity: undirected graph.



u and v are connected if there is a path between u and v.

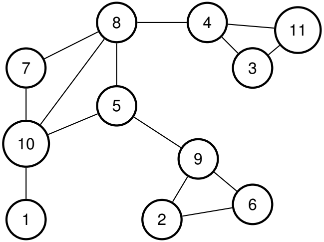
A connected graph is a graph where all pairs of vertices are connected.

If one vertex x is connected to every other vertex.
Is graph connected? Yes? No?

Proof: Use path from u to x and then from x to v. □

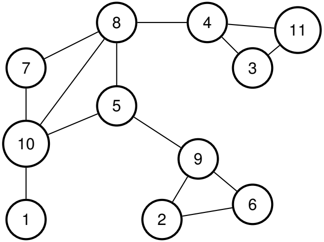
May not be simple!
Either modify definition to walk.
Or cut out cycles. .

Connected Components: Quiz.



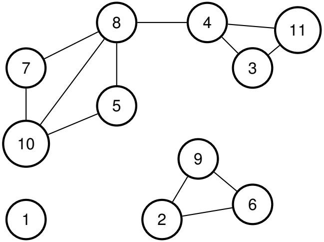
Is graph above connected?

Connected Components: Quiz.



Is graph above connected? Yes!

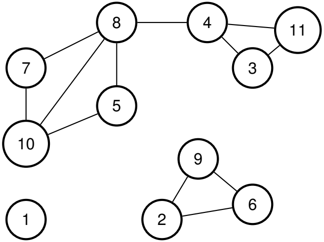
Connected Components: Quiz.



Is graph above connected? Yes!

How about now?

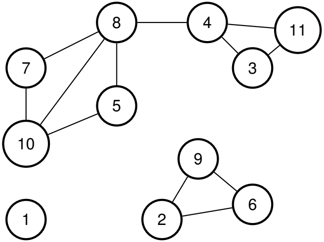
Connected Components: Quiz.



Is graph above connected? Yes!

How about now? No!

Connected Components: Quiz.

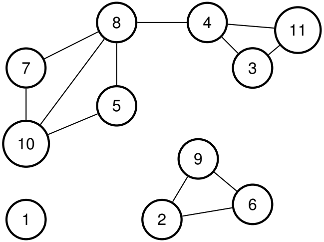


Is graph above connected? Yes!

How about now? No!

Connected Components?

Connected Components: Quiz.

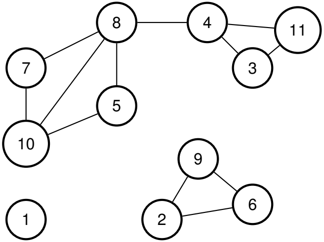


Is graph above connected? Yes!

How about now? No!

Connected Components? $\{1\},\{10,7,5,8,4,3,11\},\{2,9,6\}$.

Connected Components: Quiz.



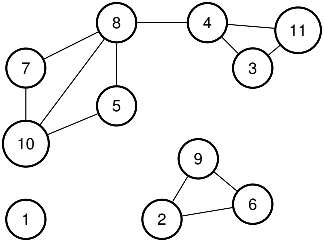
Is graph above connected? Yes!

How about now? No!

Connected Components? $\{1\}, \{10, 7, 5, 8, 4, 3, 11\}, \{2, 9, 6\}$.

Connected component - maximal set of connected vertices.

Connected Components: Quiz.

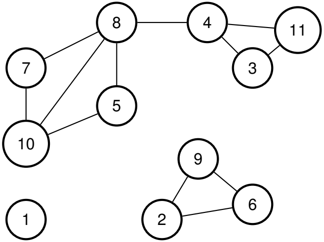


Is graph above connected? Yes!

How about now? No!

Connected Components? $\{1\}, \{10, 7, 5, 8, 4, 3, 11\}, \{2, 9, 6\}$.
 Connected component - maximal set of connected vertices.
Quick Check: Is $\{10, 7, 5\}$ a connected component?

Connected Components: Quiz.



Is graph above connected? Yes!

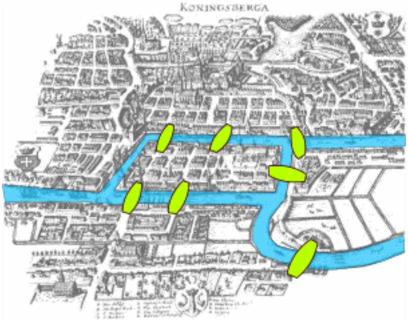
How about now? No!

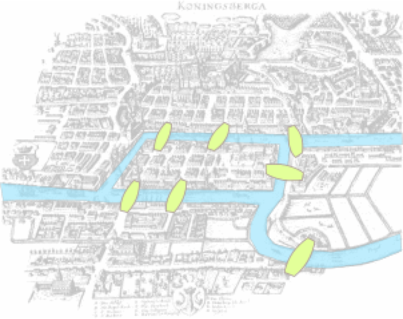
Connected Components? $\{1\}, \{10, 7, 5, 8, 4, 3, 11\}, \{2, 9, 6\}$.
Connected component - maximal set of connected vertices.
Quick Check: Is $\{10, 7, 5\}$ a connected component? No.

Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.

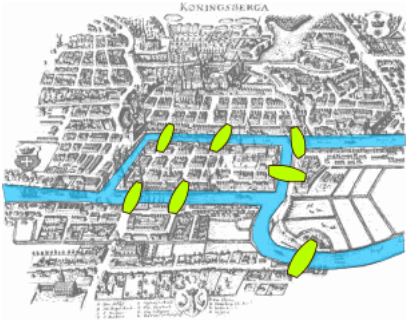


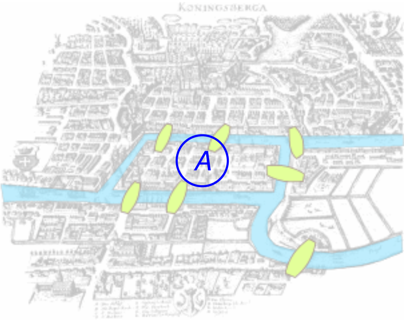


Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.

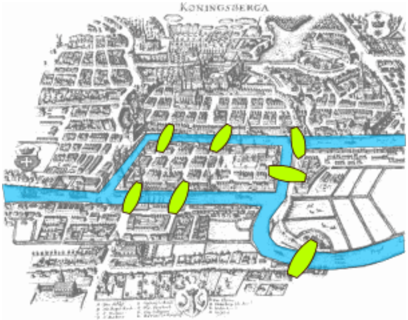


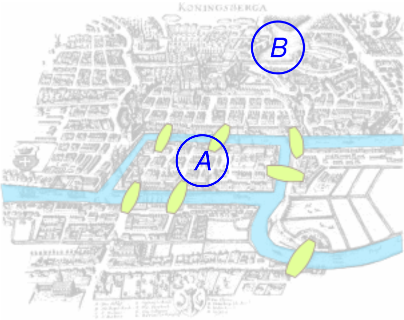


Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.

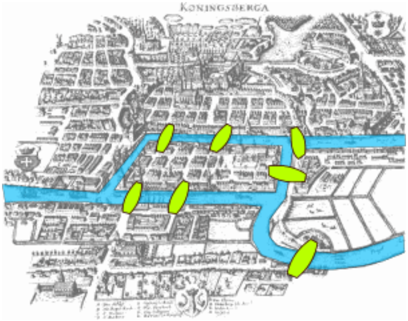


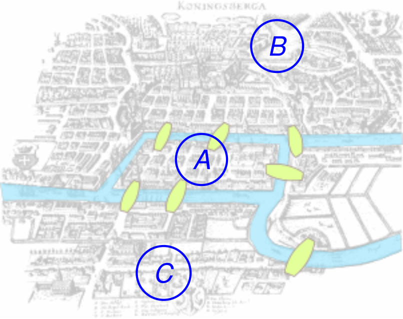


Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

"Konigsberg bridges" by Bogdan Giuşcă - License.

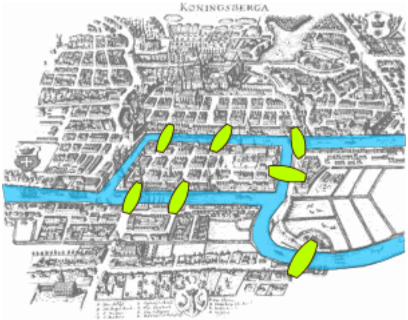


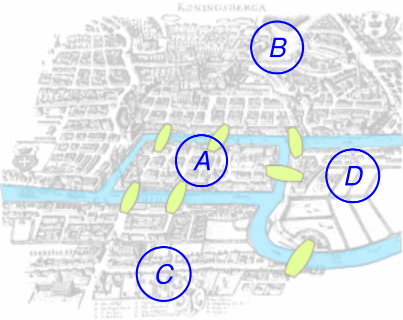


Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.

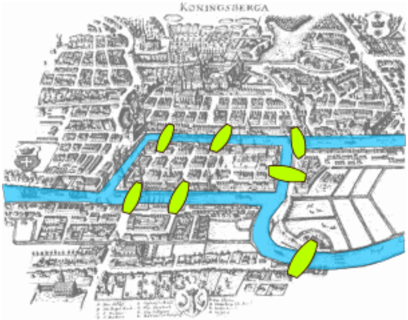


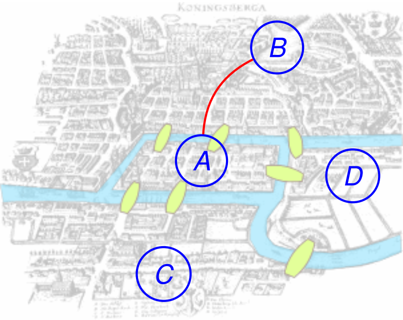


Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.

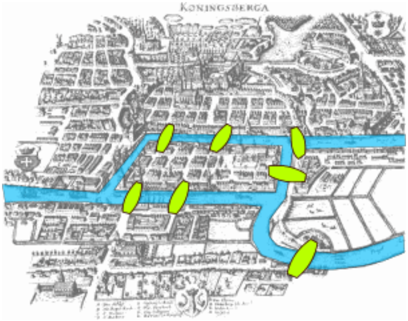


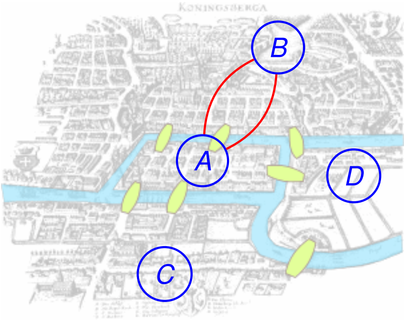


Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

"Konigsberg bridges" by Bogdan Giuşcă - License.

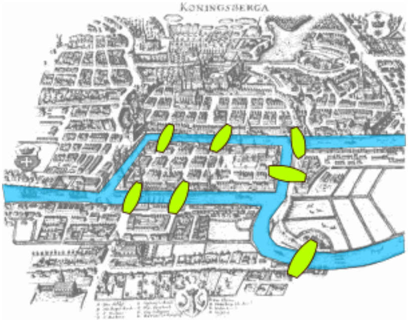


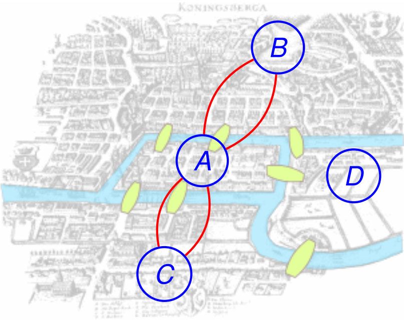


Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.

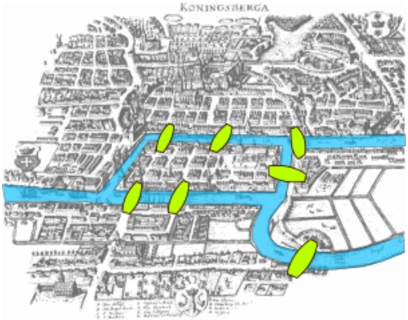


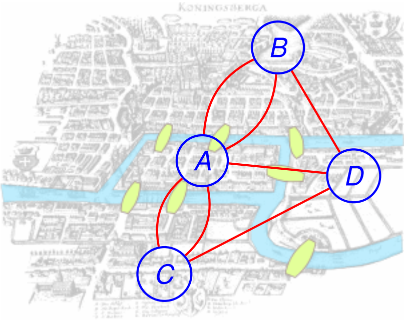


Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.

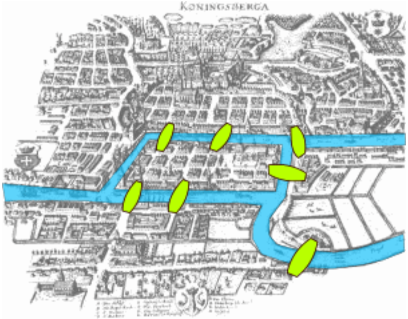


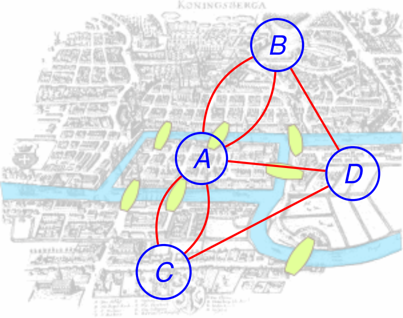


Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.



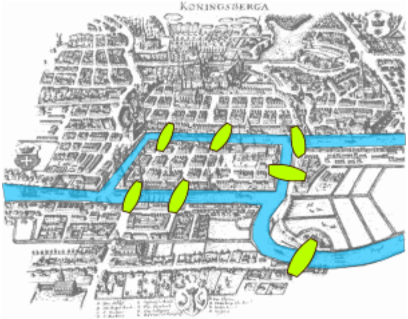


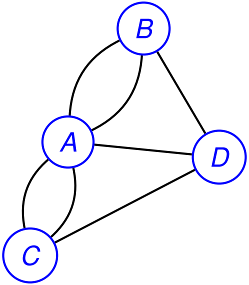
Can you draw a tour in the graph where you visit each edge once?

Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.



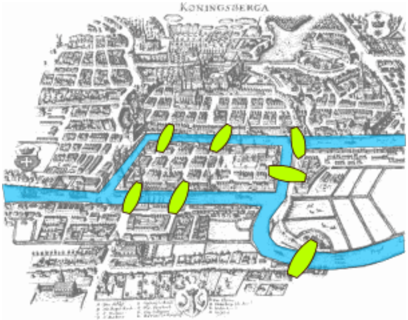


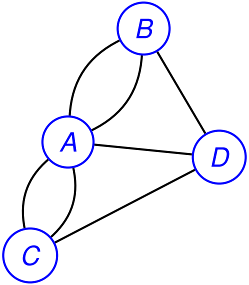
Can you draw a tour in the graph where you visit each edge once?
Yes?

Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.



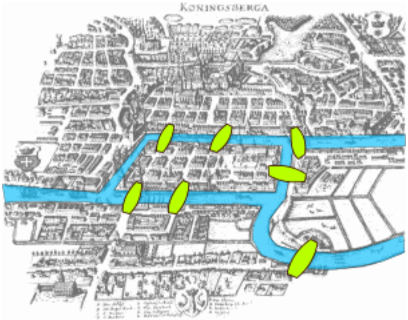


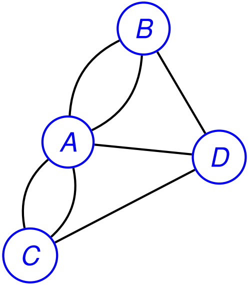
Can you draw a tour in the graph where you visit each edge once?
Yes? No?

Konigsberg bridges problem.

Can you make a tour visiting each bridge exactly once?

“Konigsberg bridges” by Bogdan Giuşcă - License.





Can you draw a tour in the graph where you visit each edge once?
Yes? No?
We will see!

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. $\square$

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

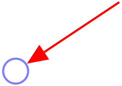
Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □



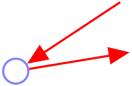
When you enter,

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □



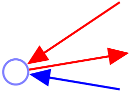
When you enter, you can leave.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □



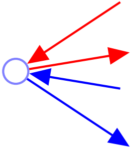
When you enter, you can leave.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □



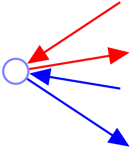
When you enter, you can leave.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □



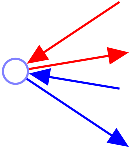
When you enter, you can leave.
For starting node,

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □



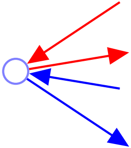
When you enter, you can leave.
For starting node, tour leaves first

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □



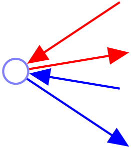
When you enter, you can leave.
For starting node, tour leaves first ....then enters at end.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\Longrightarrow$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □



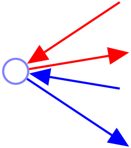
When you enter, you can leave.
For starting node, tour leaves first ....then enters at end.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\implies$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □



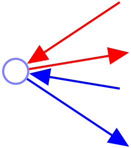
When you enter, you can leave.
For starting node, tour leaves first ....then enters at end.
Not The Hotel California.

Eulerian Tour

Eulerian Tour visits every vertex using each edge exactly once.

Theorem: Any undirected graph has an Eulerian tour if and only if all vertices have even degree and is connected.
Proof of only if: Eulerian $\Longrightarrow$ connected and all even degree.

Eulerian Tour is connected so graph is connected.
Tour enters and leaves vertex v on each visit.
Uses two incident edges per visit. Tour uses all incident edges.
Therefore v has even degree. □



When you enter, you can leave.
For starting node, tour leaves first ....then enters at end.
Not The Hotel California.

(Timestamp: 4:02).

Finding a tour!

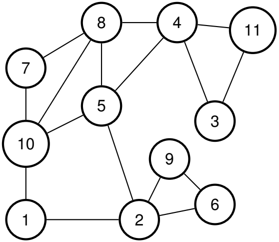
Proof of if: Even + connected $\Longrightarrow$ Eulerian Tour.
We will give an algorithm.

Finding a tour!

Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.

Finding a tour!

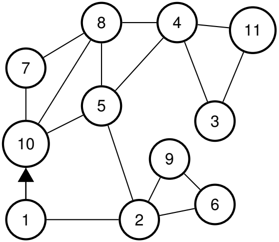
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges

Finding a tour!

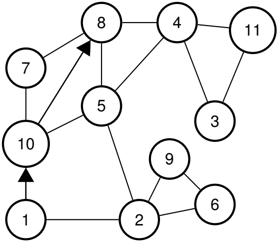
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on “unused” edges

Finding a tour!

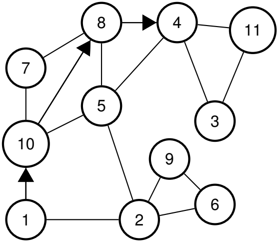
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges

Finding a tour!

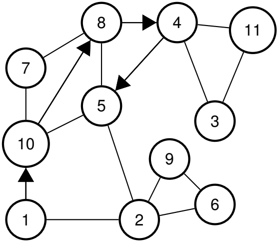
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges

Finding a tour!

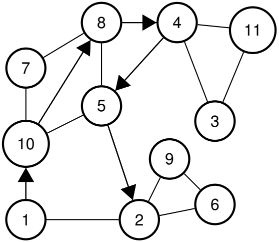
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on “unused” edges

Finding a tour!

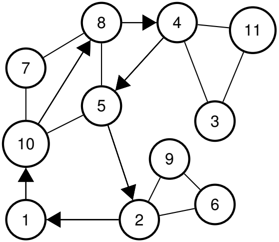
Proof of if: Even + connected $\Longrightarrow$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges

Finding a tour!

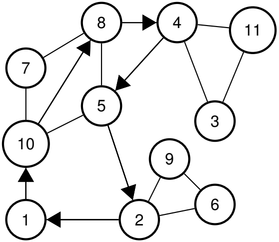
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.

Finding a tour!

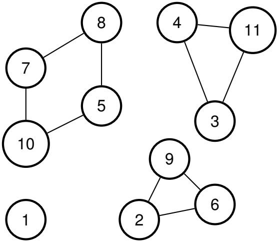
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.

Finding a tour!

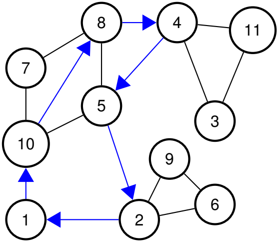
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
   ... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.

Finding a tour!

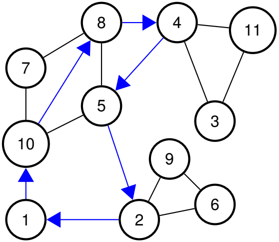
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \dots, G_k$ be connected components.
   Each is touched by C.

Finding a tour!

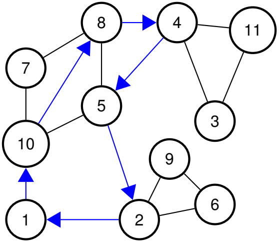
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on “unused” edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why?

Finding a tour!

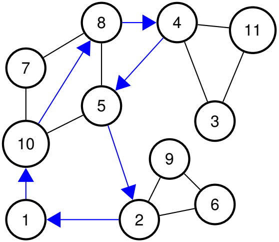
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.

Finding a tour!

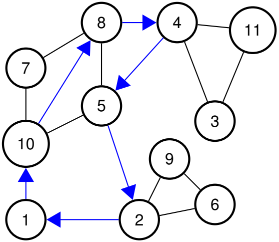
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.
   Let v_i be (first) node in G_i touched by C.

Finding a tour!

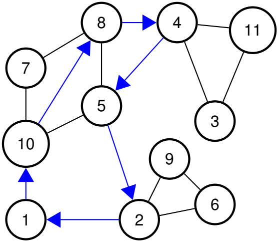
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
    Example: $v_1 = 1$,

Finding a tour!

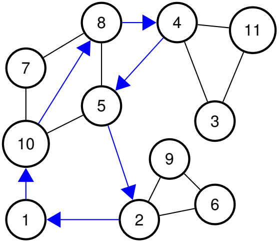
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on “unused” edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
    Example: $v_1 = 1$, $v_2 = 10$,

Finding a tour!

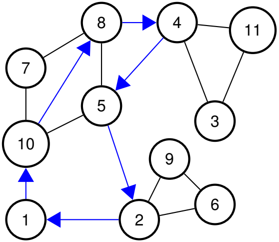
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
    Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$,

Finding a tour!

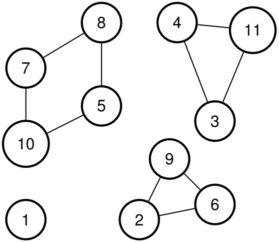
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
Each is touched by C.
Why? G was connected.
Let v_i be (first) node in G_i touched by C.
Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.

Finding a tour!

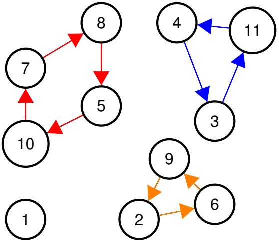
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
    Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.
4. Recurse on $G_1, \ldots, G_k$ starting from v_i

Finding a tour!

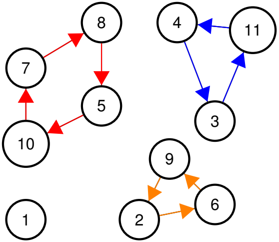
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
    Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.
4. Recurse on $G_1, \ldots, G_k$ starting from v_i

Finding a tour!

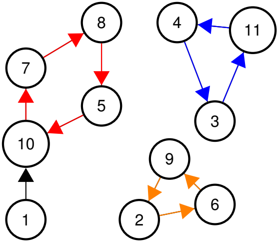
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
    Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.
4. Recurse on $G_1, \ldots, G_k$ starting from v_i
5. Splice together.

Finding a tour!

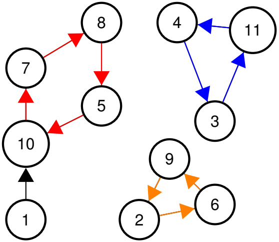
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
    Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.
4. Recurse on $G_1, \ldots, G_k$ starting from v_i
5. Splice together.
   1,10

Finding a tour!

Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.

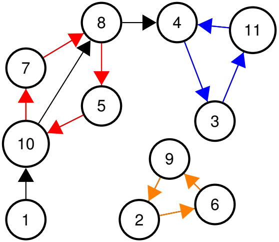


1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
    Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.
4. Recurse on $G_1, \ldots, G_k$ starting from v_i
5. Splice together.
   1,10,7,8,5,10

Finding a tour!

Proof of if: Even + connected $\implies$ Eulerian Tour.

We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges

... till you get back to v.

2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.

   Each is touched by C.

   Why? G was connected.

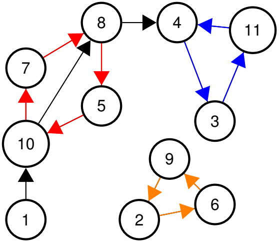
   Let v_i be (first) node in G_i touched by C.

   Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.
4. Recurse on $G_1, \ldots, G_k$ starting from v_i
5. Splice together.

   1,10,7,8,5,10 ,8,4

Finding a tour!

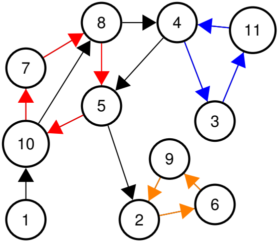
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
      Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
    Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.
4. Recurse on $G_1, \ldots, G_k$ starting from v_i
5. Splice together.
   1,10,7,8,5,10 ,8,4,3,11,4

Finding a tour!

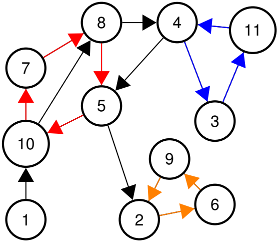
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
   ... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
   Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
   Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.
4. Recurse on $G_1, \ldots, G_k$ starting from v_i
5. Splice together.
   1,10,7,8,5,10 ,8,4,3,11,4 5,2

Finding a tour!

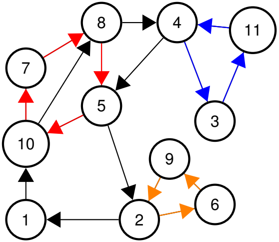
Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
   Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
   Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.
4. Recurse on $G_1, \ldots, G_k$ starting from v_i
5. Splice together.
   1,10,7,8,5,10 ,8,4,3,11,4 5,2,6,9,2

Finding a tour!

Proof of if: Even + connected $\implies$ Eulerian Tour.
We will give an algorithm. First by picture.



1. Take a walk starting from v (1) on "unused" edges
... till you get back to v.
2. Remove tour, C.
3. Let $G_1, \ldots, G_k$ be connected components.
   Each is touched by C.
   Why? G was connected.
   Let v_i be (first) node in G_i touched by C.
   Example: $v_1 = 1$, $v_2 = 10$, $v_3 = 4$, $v_4 = 2$.
4. Recurse on $G_1, \ldots, G_k$ starting from v_i
5. Splice together.
   1,10,7,8,5,10 ,8,4,3,11,4 5,2,6,9,2 and to 1!

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.
Resulting graph may be disconnected. (Removed edges!)

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. $\square$

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
  Why is there a v_i in C?
    G was connected $\implies$ path from G_i to rest.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.
Prf: Tour C has even incidences to any vertex v.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.
Prf: Tour C has even incidences to any vertex v. □

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.
Prf: Tour C has even incidences to any vertex v. □

3. Find tour T_i of G_i

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.
Prf: Tour C has even incidences to any vertex v. □

3. Find tour T_i of G_i starting/ending at v_i.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.
Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.
Prf: Tour C has even incidences to any vertex v. □

3. Find tour T_i of G_i starting/ending at v_i. Induction.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\Longrightarrow$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.
Prf: Tour C has even incidences to any vertex v. □

3. Find tour T_i of G_i starting/ending at v_i. Induction.
4. Splice T_i into C where v_i first appears in C.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.
Prf: Tour C has even incidences to any vertex v. □

3. Find tour T_i of G_i starting/ending at v_i. Induction.
4. Splice T_i into C where v_i first appears in C.

Visits every edge once:
Visits edges in C

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.
Prf: Tour C has even incidences to any vertex v. □

3. Find tour T_i of G_i starting/ending at v_i. Induction.
4. Splice T_i into C where v_i first appears in C.

Visits every edge once:
Visits edges in C exactly once.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.
Prf: Tour C has even incidences to any vertex v. □

3. Find tour T_i of G_i starting/ending at v_i. Induction.
4. Splice T_i into C where v_i first appears in C.

Visits every edge once:
Visits edges in C exactly once.
By induction for all edges in each G_i.

Recursive/Inductive Algorithm.

1. Take a walk from arbitrary node v, until you get back to v.

Claim: Do get back to v!
Proof of Claim: Even degree. If enter, can leave except for v. □

2. Remove cycle, C, from G.

Resulting graph may be disconnected. (Removed edges!)
Let components be $G_1, \ldots, G_k$.
Let v_i be first vertex of C that is in G_i.
Why is there a v_i in C?
G was connected $\implies$ path from G_i to rest.
a vertex in G_i must be incident to a removed edge in C.

Claim: Each vertex in each G_i has even degree and is connected.
Prf: Tour C has even incidences to any vertex v. □

3. Find tour T_i of G_i starting/ending at v_i. Induction.
4. Splice T_i into C where v_i first appears in C.

Visits every edge once:
Visits edges in C exactly once.
By induction for all edges in each G_i. □

Poll: Euler concepts.

Mark correct statements for a connected graph where all vertices have even degree. (Below, tours uses edges at most once, but may involve a vertex several times.

Poll: Euler concepts.

Mark correct statements for a connected graph where all vertices have even degree. (Below, tours uses edges at most once, but may involve a vertex several times.

(A) Removing a tour leaves a graph of even degree.
(B) A tour connecting a set of connected components, each with a Eulerian tour is really cool! Eulerian even.
(C) There is no hotel california in this graph.
(D) After removing a set of edges E' in a connected graph, every connected component is incident to an edge in E'
(E) If one walks on new edges, starting at v, one must eventually get back to v.
(F) Removing a tour leaves a connected graph.

Poll: Euler concepts.

Mark correct statements for a connected graph where all vertices have even degree. (Below, tours uses edges at most once, but may involve a vertex several times.

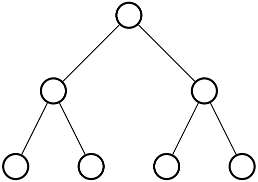
(A) Removing a tour leaves a graph of even degree.
(B) A tour connecting a set of connected components, each with a Eulerian tour is really cool! Eulerian even.
(C) There is no hotel california in this graph.
(D) After removing a set of edges E' in a connected graph, every connected component is incident to an edge in E'
(E) If one walks on new edges, starting at v, one must eventually get back to v.
(F) Removing a tour leaves a connected graph.

Only (F) is false.

A Tree, a tree.

Graph $G = (V, E)$.

Binary Tree!



More generally.

Trees.

Definitions:

Trees.

Definitions:

A connected graph without a cycle.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V| - 1$ edges.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V| - 1$ edges.
A connected graph where any edge removal disconnects it.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V|-1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Trees.

Definitions:

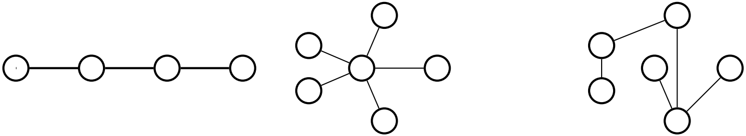
A connected graph without a cycle.
A connected graph with $|V|-1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V| - 1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



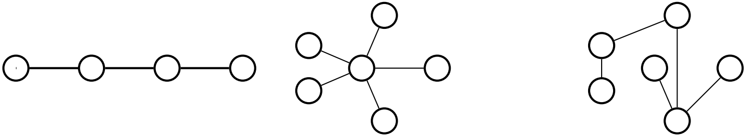
no cycle and connected?

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V| - 1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



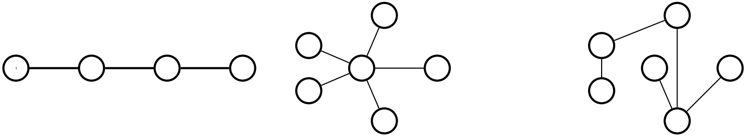
no cycle and connected? Yes.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V| - 1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



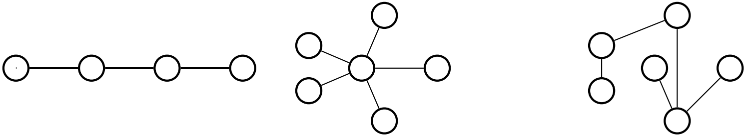
no cycle and connected? Yes.
$|V| - 1$ edges and connected?

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V|-1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



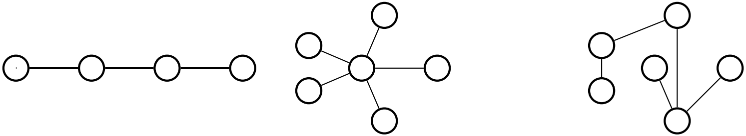
no cycle and connected? Yes.
$|V|-1$ edges and connected? Yes.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V|-1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



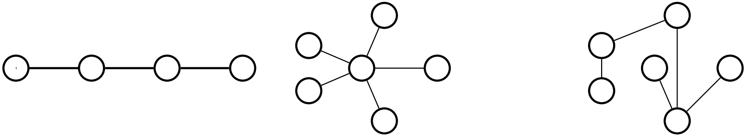
no cycle and connected? Yes.
$|V|-1$ edges and connected? Yes.
removing any edge disconnects it.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V| - 1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



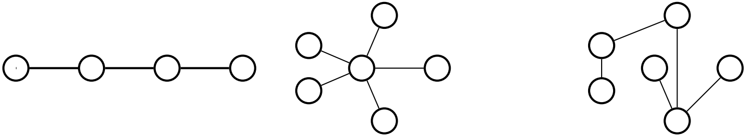
no cycle and connected? Yes.
$|V| - 1$ edges and connected? Yes.
removing any edge disconnects it. Harder to check.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V|-1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



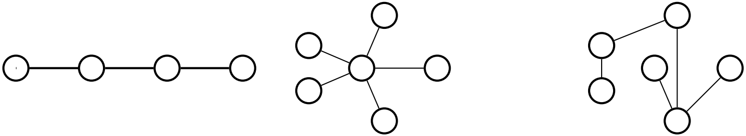
no cycle and connected? Yes.
$|V|-1$ edges and connected? Yes.
removing any edge disconnects it. Harder to check. but yes.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V|-1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



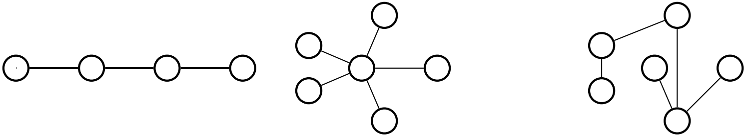
no cycle and connected? Yes.
$|V|-1$ edges and connected? Yes.
removing any edge disconnects it. Harder to check. but yes.
Adding any edge creates cycle.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V|-1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



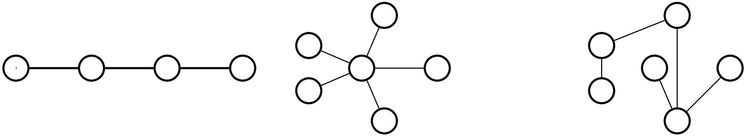
no cycle and connected? Yes.
$|V|-1$ edges and connected? Yes.
removing any edge disconnects it. Harder to check. but yes.
Adding any edge creates cycle. Harder to check.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V| - 1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



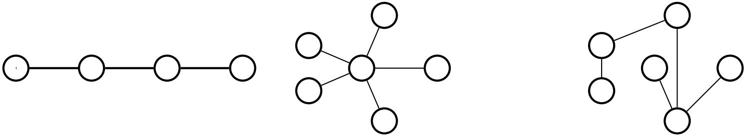
no cycle and connected? Yes.
$|V| - 1$ edges and connected? Yes.
removing any edge disconnects it. Harder to check. but yes.
Adding any edge creates cycle. Harder to check. but yes.

Trees.

Definitions:

A connected graph without a cycle.
A connected graph with $|V| - 1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



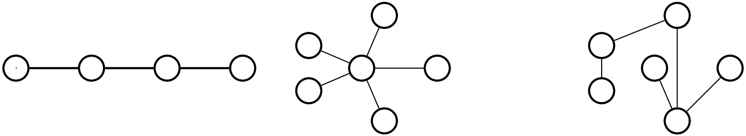
no cycle and connected? Yes.
$|V| - 1$ edges and connected? Yes.
removing any edge disconnects it. Harder to check. but yes.
Adding any edge creates cycle. Harder to check. but yes.

Trees.

Definitions:

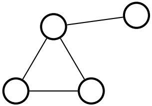
A connected graph without a cycle.
A connected graph with $|V|-1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



no cycle and connected? Yes.
$|V|-1$ edges and connected? Yes.
removing any edge disconnects it. Harder to check. but yes.
Adding any edge creates cycle. Harder to check. but yes.

To tree or not to tree!

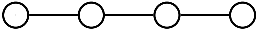


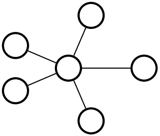
Trees.

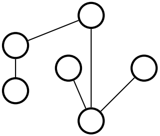
Definitions:

A connected graph without a cycle.
A connected graph with $|V|-1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.

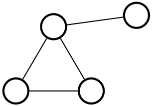


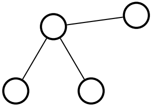




no cycle and connected? Yes.
$|V|-1$ edges and connected? Yes.
removing any edge disconnects it. Harder to check. but yes.
Adding any edge creates cycle. Harder to check. but yes.

To tree or not to tree!



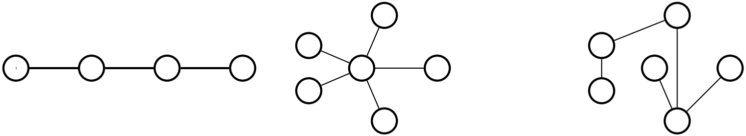


Trees.

Definitions:

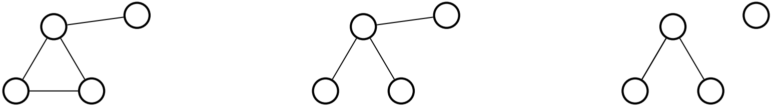
A connected graph without a cycle.
A connected graph with $|V|-1$ edges.
A connected graph where any edge removal disconnects it.
A connected graph where any edge addition creates a cycle.

Some trees.



no cycle and connected? Yes.
$|V|-1$ edges and connected? Yes.
removing any edge disconnects it. Harder to check. but yes.
Adding any edge creates cycle. Harder to check. but yes.

To tree or not to tree!



Equivalence of Definitions.

Theorem:
"G connected and has $|V|-1$ edges" $\equiv$
"G is connected and has no cycles."

Equivalence of Definitions.

Theorem:
"G connected and has $|V| - 1$ edges" $\equiv$
"G is connected and has no cycles."

Lemma: If v is degree 1 in connected graph G, $G - v$ is connected.
Proof:
For $x \neq v, y \neq v \in V$,

Equivalence of Definitions.

Theorem:
"G connected and has $|V| - 1$ edges" $\equiv$
"G is connected and has no cycles."

Lemma: If v is degree 1 in connected graph G, $G - v$ is connected.
Proof:
For $x \neq v, y \neq v \in V$,
there is path between x and y in G since connected.

Equivalence of Definitions.

Theorem:
"G connected and has $|V|-1$ edges" $\equiv$
"G is connected and has no cycles."

Lemma: If v is degree 1 in connected graph G, $G-v$ is connected.
Proof:
For $x \neq v, y \neq v \in V$,
there is path between x and y in G since connected.
and does not use v (degree 1)

Equivalence of Definitions.

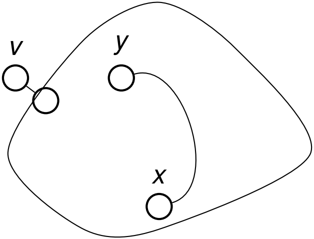
Theorem:
"G connected and has $|V|-1$ edges" $\equiv$
"G is connected and has no cycles."

Lemma: If v is degree 1 in connected graph G, $G-v$ is connected.
Proof:
For $x \neq v, y \neq v \in V$,
there is path between x and y in G since connected.
and does not use v (degree 1)
$\implies G-v$ is connected.

Equivalence of Definitions.

Theorem:
"G connected and has $|V|-1$ edges" $\equiv$
"G is connected and has no cycles."

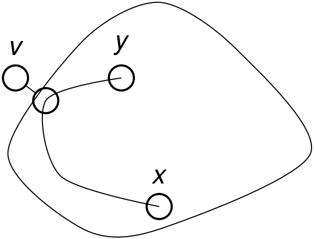
Lemma: If v is degree 1 in connected graph G, $G-v$ is connected.
Proof:
For $x \neq v, y \neq v \in V$,
there is path between x and y in G since connected.
and does not use v (degree 1)
$\implies$ $G-v$ is connected. □



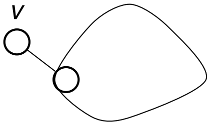
Equivalence of Definitions.

Theorem:
"G connected and has $|V|-1$ edges" $\equiv$
"G is connected and has no cycles."

Lemma: If v is degree 1 in connected graph G, $G-v$ is connected.
Proof:
For $x \neq v, y \neq v \in V$,
there is path between x and y in G since connected.
and does not use v (degree 1)
$\implies G-v$ is connected. □



Proof of only if.



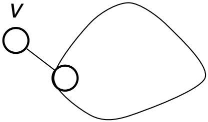
Thm:
"G connected and has $|V| - 1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$:

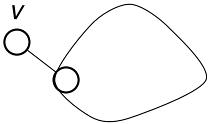
Proof of only if.

Thm:
"G connected and has $|V|-1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.



Proof of only if.

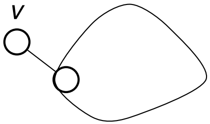


Thm:
"G connected and has $|V| - 1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V| = 1$. $0 = |V| - 1$ edges and has no cycles.

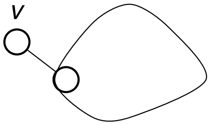
Proof of only if.

Thm:
"G connected and has $|V|-1$ edges" $\implies$
"G is connected and has no cycles."



Proof of $\implies$: By induction on $|V|$.
Base Case: $|V|=1$. $0=|V|-1$ edges and has no cycles.

Proof of only if.

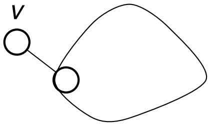


Thm:
"G connected and has $|V|-1$ edges" $\Longrightarrow$
"G is connected and has no cycles."

Proof of $\Longrightarrow$: By induction on $|V|$.
Base Case: $|V|=1$. $0=|V|-1$ edges and has no cycles.

Induction Step:

Proof of only if.

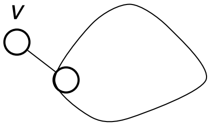


Thm:
"G connected and has $|V|-1$ edges" $\Longrightarrow$
"G is connected and has no cycles."

Proof of $\Longrightarrow$: By induction on $|V|$.
Base Case: $|V|=1$. $0=|V|-1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.

Proof of only if.

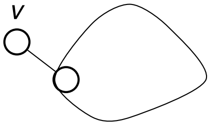


Thm:
"G connected and has $|V|-1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V|=1$. $0=|V|-1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\implies$ every vertex degree ≥ 1.

Proof of only if.

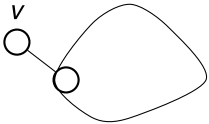


Thm:
"G connected and has $|V|-1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V|=1$. $0=|V|-1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\implies$ every vertex degree ≥ 1.
Sum of degrees is $2|E|=2(|V|-1)=2|V|-2$

Proof of only if.

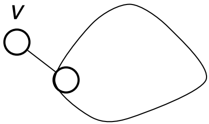


Thm:
"G connected and has $|V|-1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V| = 1$. $0 = |V| - 1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\implies$ every vertex degree ≥ 1.
Sum of degrees is $2|E| = 2(|V|-1) = 2|V|-2$
Average degree $(2|V|-2)/|V| = 2-(2/|V|)$.

Proof of only if.

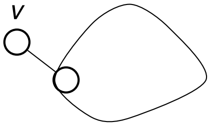


Thm:
"G connected and has $|V|-1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V| = 1$. $0 = |V|-1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\implies$ every vertex degree ≥ 1.
Sum of degrees is $2|E| = 2(|V|-1) = 2|V|-2$
Average degree $(2|V|-2)/|V| = 2-(2/|V|)$. Must be a degree 1 vertex.

Proof of only if.

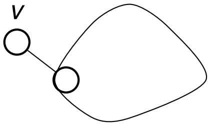


Thm:
"G connected and has $|V|-1$ edges" $\Longrightarrow$
"G is connected and has no cycles."

Proof of $\Longrightarrow$: By induction on $|V|$.
Base Case: $|V|=1$. $0=|V|-1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\Longrightarrow$ every vertex degree ≥ 1.
Sum of degrees is $2|E|=2(|V|-1)=2|V|-2$
Average degree $(2|V|-2)/|V|=2-(2/|V|)$. Must be a degree 1 vertex.
Cuz not everyone is bigger than average!

Proof of only if.

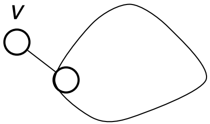


Thm:
"G connected and has $|V| - 1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V| = 1$. $0 = |V| - 1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\implies$ every vertex degree ≥ 1.
Sum of degrees is $2|E| = 2(|V| - 1) = 2|V| - 2$
Average degree $(2|V| - 2)/|V| = 2 - (2/|V|)$. Must be a degree 1 vertex.
Cuz not everyone is bigger than average! □

Proof of only if.

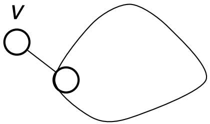


Thm:
"G connected and has $|V|-1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V|=1$. $0=|V|-1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\implies$ every vertex degree ≥ 1.
Sum of degrees is $2|E|=2(|V|-1)=2|V|-2$
Average degree $(2|V|-2)/|V|=2-(2/|V|)$. Must be a degree 1 vertex.
Cuz not everyone is bigger than average! □
By degree 1 removal lemma, $G-v$ is connected.

Proof of only if.

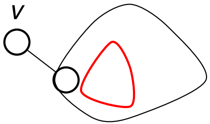


Thm:
"G connected and has $|V|-1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V|=1$. $0=|V|-1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\implies$ every vertex degree ≥ 1.
Sum of degrees is $2|E|=2(|V|-1)=2|V|-2$
Average degree $(2|V|-2)/|V|=2-(2/|V|)$. Must be a degree 1 vertex.
Cuz not everyone is bigger than average! □
By degree 1 removal lemma, $G-v$ is connected.
$G-v$ has $|V|-1$ vertices and $|V|-2$ edges so by induction

Proof of only if.

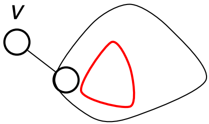


Thm:
"G connected and has $|V| - 1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V| = 1$. $0 = |V| - 1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\implies$ every vertex degree ≥ 1.
Sum of degrees is $2|E| = 2(|V| - 1) = 2|V| - 2$
Average degree $(2|V| - 2)/|V| = 2 - (2/|V|)$. Must be a degree 1 vertex.
Cuz not everyone is bigger than average! □
By degree 1 removal lemma, $G - v$ is connected.
$G - v$ has $|V| - 1$ vertices and $|V| - 2$ edges so by induction
$\implies$ no cycle in $G - v$.

Proof of only if.

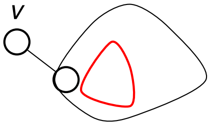


Thm:
"G connected and has $|V|-1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V| = 1$. $0 = |V| - 1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\implies$ every vertex degree ≥ 1.
Sum of degrees is $2|E| = 2(|V|-1) = 2|V|-2$
Average degree $(2|V|-2)/|V| = 2-(2/|V|)$. Must be a degree 1 vertex.
Cuz not everyone is bigger than average! □
By degree 1 removal lemma, $G - v$ is connected.
$G - v$ has $|V|-1$ vertices and $|V|-2$ edges so by induction
$\implies$ no cycle in $G - v$.
And no cycle in G since degree 1 cannot participate in cycle.

Proof of only if.



Thm:
"G connected and has $|V|-1$ edges" $\implies$
"G is connected and has no cycles."

Proof of $\implies$: By induction on $|V|$.
Base Case: $|V|=1$. $0=|V|-1$ edges and has no cycles.

Induction Step:
Claim: There is a degree 1 node.
Proof: First, connected $\implies$ every vertex degree ≥ 1.
Sum of degrees is $2|E|=2(|V|-1)=2|V|-2$
Average degree $(2|V|-2)/|V|=2-(2/|V|)$. Must be a degree 1 vertex.
Cuz not everyone is bigger than average! □
By degree 1 removal lemma, $G-v$ is connected.
$G-v$ has $|V|-1$ vertices and $|V|-2$ edges so by induction
$\implies$ no cycle in $G-v$.
And no cycle in G since degree 1 cannot participate in cycle. □

Proof of if

Thm:
“G is connected and has no cycles”
$\implies$ “G connected and has $|V| - 1$ edges”
Proof:

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.
Proof of Claim:
Can't visit more than once since no cycle.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.
Proof of Claim:
Can't visit more than once since no cycle.
Entered.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.
Proof of Claim:
Can't visit more than once since no cycle.
Entered. Didn't leave.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.
Proof of Claim:
Can't visit more than once since no cycle.
Entered. Didn't leave. Only one incident edge.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.
Proof of Claim:
Can't visit more than once since no cycle.
Entered. Didn't leave. Only one incident edge. □
Removing node doesn't create cycle.

Proof of if

Thm:
"G is connected and has no cycles"
$\Longrightarrow$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.
Proof of Claim:
Can't visit more than once since no cycle.
Entered. Didn't leave. Only one incident edge. $\square$
Removing node doesn't create cycle.
New graph is connected.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.
Proof of Claim:
Can't visit more than once since no cycle.
Entered. Didn't leave. Only one incident edge. □
Removing node doesn't create cycle.
New graph is connected.
Removing degree 1 node doesn't disconnect from Degree 1 lemma.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.
Proof of Claim:
Can't visit more than once since no cycle.
Entered. Didn't leave. Only one incident edge. $\square$
Removing node doesn't create cycle.
New graph is connected.
Removing degree 1 node doesn't disconnect from Degree 1 lemma.
By induction $G - v$ has $|V| - 2$ edges.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V|-1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.
Proof of Claim:
Can't visit more than once since no cycle.
Entered. Didn't leave. Only one incident edge. □
Removing node doesn't create cycle.
New graph is connected.
Removing degree 1 node doesn't disconnect from Degree 1 lemma.
By induction $G-v$ has $|V|-2$ edges.
G has one more or $|V|-1$ edges.

Proof of if

Thm:
"G is connected and has no cycles"
$\implies$ "G connected and has $|V| - 1$ edges"
Proof:
Walk from a vertex using untraversed edges.
Until get stuck.
Claim: Degree 1 vertex.
Proof of Claim:
Can't visit more than once since no cycle.
Entered. Didn't leave. Only one incident edge. $\square$
Removing node doesn't create cycle.
New graph is connected.
Removing degree 1 node doesn't disconnect from Degree 1 lemma.
By induction $G - v$ has $|V| - 2$ edges.
G has one more or $|V| - 1$ edges. $\square$

Poll: Oh tree, beautiful tree.

Let G be a connected graph with $|V| - 1$ **edges.**

Poll: Oh tree, beautiful tree.

Let G be a connected graph with $|V| - 1$ edges.

(A) Removing a degree 1 vertex can disconnect the graph.
(B) One can use induction on smaller objects.
(C) The average degree is $2 - 2/|V|$.
(D) There is a hotel california: a degree 1 vertex.
(E) Everyone can be bigger than average.

Poll: Oh tree, beautiful tree.

Let G be a connected graph with $|V|-1$ edges.

(A) Removing a degree 1 vertex can disconnect the graph.
(B) One can use induction on smaller objects.
(C) The average degree is $2-2/|V|$.
(D) There is a hotel california: a degree 1 vertex.
(E) Everyone can be bigger than average.

(B), (C), (D) are true

Lecture Summary.

Graphs.

Lecture Summary.

Graphs.
  Basics.

Lecture Summary.

Graphs.

Basics.

Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.

Lecture Summary.

Graphs.
- Basics.
- Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.
  - Connected Component.

Lecture Summary.

Graphs.
- Basics.
- Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.
  - Connected Component.
    - maximal set of vertices that are connected.

Lecture Summary.

Graphs.
- Basics.
- Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.
  - Connected Component.
    - maximal set of vertices that are connected.

Algorithm for Eulerian Tour.

Lecture Summary.

Graphs.
- Basics.
- Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.
  - Connected Component.
    - maximal set of vertices that are connected.

Algorithm for Eulerian Tour.
- Take a walk until stuck to form tour.

Lecture Summary.

Graphs.
- Basics.
- Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.
  - Connected Component.
    - maximal set of vertices that are connected.

Algorithm for Eulerian Tour.
- Take a walk until stuck to form tour.
- Remove tour.

Lecture Summary.

Graphs.
- Basics.
- Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.
  - Connected Component.
    - maximal set of vertices that are connected.

Algorithm for Eulerian Tour.
- Take a walk until stuck to form tour.
- Remove tour.
- Recurse on connected components.

Lecture Summary.

Graphs.
- Basics.
- Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.
  - Connected Component.
    - maximal set of vertices that are connected.

Algorithm for Eulerian Tour.
- Take a walk until stuck to form tour.
- Remove tour.
- Recurse on connected components.

Lecture Summary.

Graphs.
Basics.
Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.
Connected Component.
maximal set of vertices that are connected.
Algorithm for Eulerian Tour.
Take a walk until stuck to form tour.
Remove tour.
Recurse on connected components.

Trees: degree 1 lemma $\implies$ equivalence of several definitions.

Lecture Summary.

Graphs.
 Basics.
 Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.
  Connected Component.
   maximal set of vertices that are connected.
Algorithm for Eulerian Tour.
  Take a walk until stuck to form tour.
  Remove tour.
  Recurse on connected components.

Trees: degree 1 lemma $\implies$ equivalence of several definitions.
 G: n vertices and $n-1$ edges and connected.
  remove degree 1 vertex.
 $n-1$ vertices, $n-2$ edges and connected $\implies$ acyclic.
  (Ind. Hyp.)
 degree 1 vertex is not in a cycle.
 G is acyclic.

Lecture Summary.

Graphs.
- Basics.
- Degree, Incidence, Sum of degrees is $2|E|$. Connectivity.
  - Connected Component.
    - maximal set of vertices that are connected.

Algorithm for Eulerian Tour.
- Take a walk until stuck to form tour.
- Remove tour.
- Recurse on connected components.

Trees: degree 1 lemma $\implies$ equivalence of several definitions.
- G: n vertices and $n-1$ edges and connected.
  - remove degree 1 vertex.
  - $n-1$ vertices, $n-2$ edges and connected $\implies$ acyclic.
    - (Ind. Hyp.)
  - degree 1 vertex is not in a cycle.
- G is acyclic.