Warm-Up + Graphs

Outline

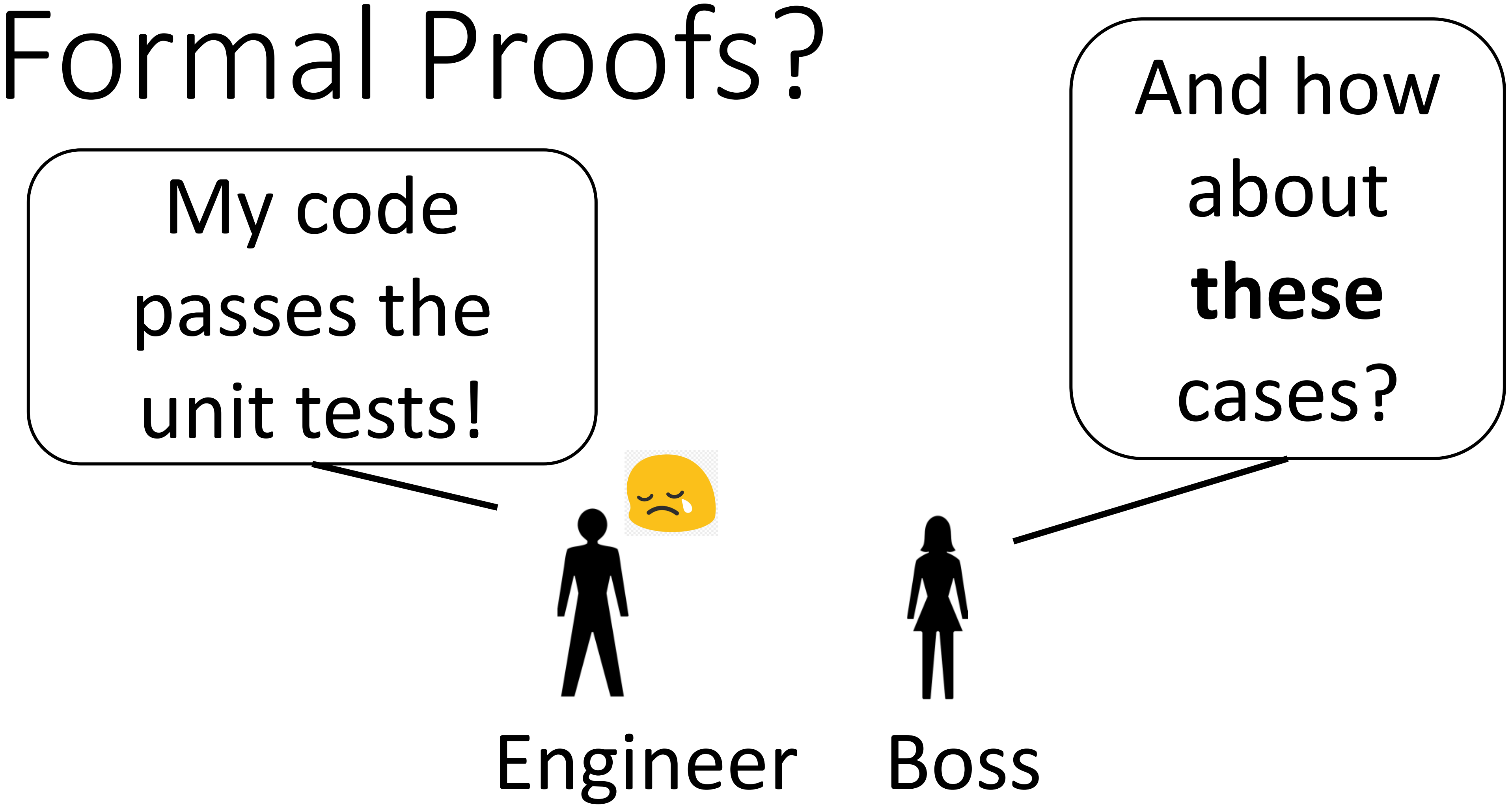
- Why do we want Formal Proofs?
- Asymptotic analysis
- Graphs
  - Recap on terminology

Outline

- Why do we want Formal Proofs?
- Asymptotic analysis
- Graphs
  - Recap on terminology

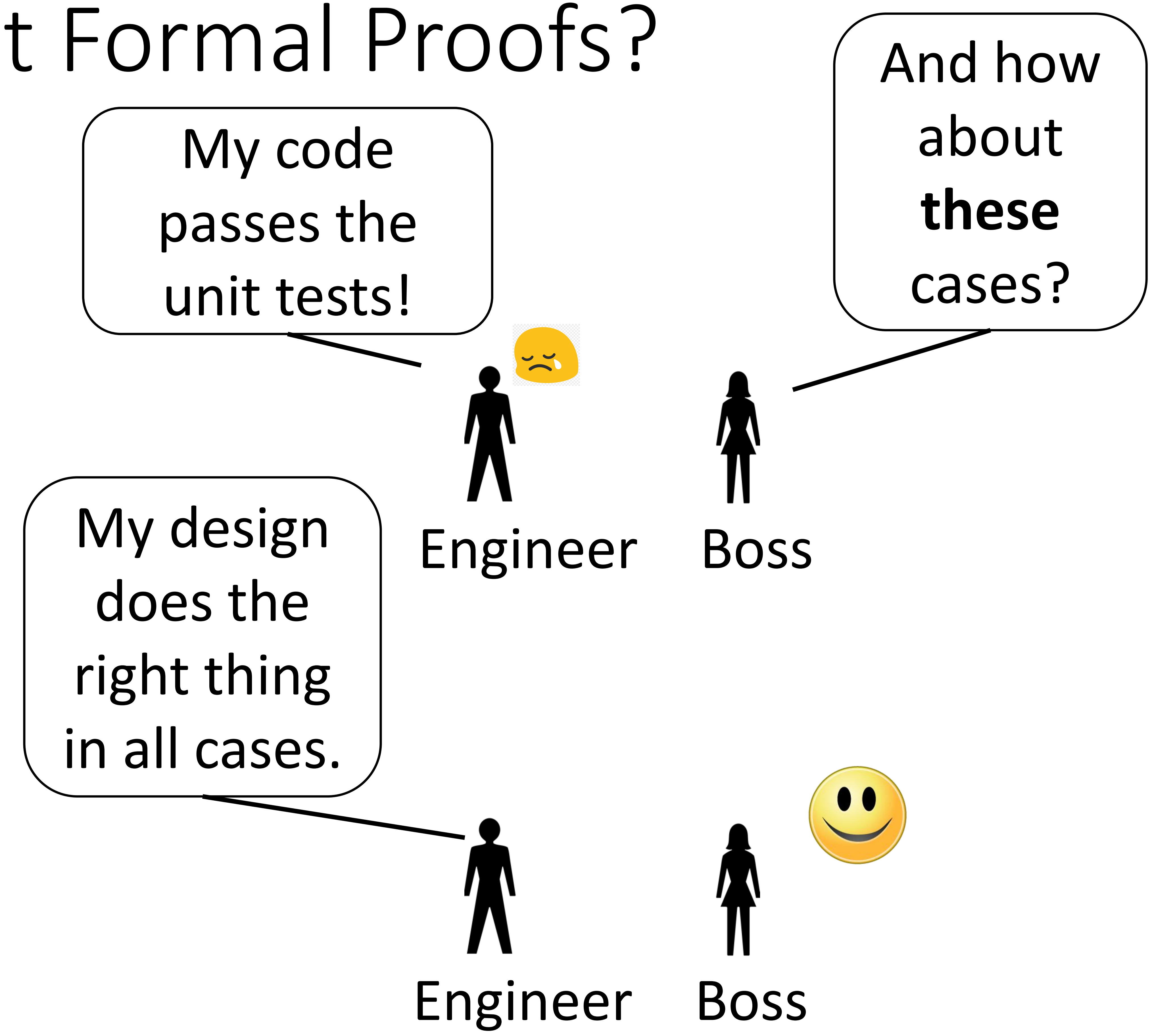
Learning objectives:
You are able to
- name and describe basic graph properties
- compare runtimes of algorithms
- describe the concepts of correctness and the asymptotic runtime of an algorithm

Why do we want Formal Proofs?



Why do we want Formal Proofs?

Understand and explain how the algorithm you designed works!



Why do we want Formal Proofs?

Correctness proof

↓

I am sure that my algorithm works

Runtime analysis, usually worst case

↓

I am sure that my algorithm *always* works fast

Outline

- Why do we want Formal Proofs?
- Asymptotic analysis
- Graphs
    - Recap on terminology

Asymptotic Analysis

Runtime analysis:
Count the number of "primitive operations".

Asymptotic Analysis

Runtime analysis:
Count the number of "primitive operations".

Calculate sum (n):
Return $n + n$

Asymptotic Analysis

Runtime analysis:
Count the number of "primitive operations".

Calculate sum (n):
Return $n + n$

Runtime of the naïve algorithm:
Roughly $\log n$ additions of constant size numbers.

An addition of constant size numbers takes a constant number of operations on a CPU.

Asymptotic Analysis

Independent of the underlying architecture

Runtime of the naïve algorithm:
Roughly $\log n$ additions of constant size numbers.

An addition of constant size numbers takes a constant number of operations on a CPU.

Calculate sum (n):
Return $n + n$

Independent of the programming language

Runtime is *roughly* logarithmic in the input size

The O-notation

Definition:
For two functions f and g, we write $f(n) = O(g(n))$, if there are some constants $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Suppose that $f(n) = O(n \log n)$ is an upper bound on the runtime of an algorithm A any input of size n for any n.

Then we say that the runtime of A is $O(n \log n)$.

You can of course replace $O(n \log n)$ with any other function $g(n)$.

More definitions:

$f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$

$f(n) = \Theta(g(n))$ if $g(n) = O(f(n)$ and $f(n) = O(g(n))$

$f(n) = o(g(n))$ if for every constant c there is an n_0 such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

$f(n) = \omega(g(n))$ if $g(n) = o(f(n))$.

The O-notation

Definition:
For two functions f and g, we write $f(n) = O(g(n))$, if there are some constants $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Suppose that $f(n) = O(n \log n)$ is an upper bound on the runtime of an algorithm A any input of size n for any n.

Then we say that the runtime of A is $O(n \log n)$.

You can of course replace $O(n \log n)$ with any other function $g(n)$.

More definitions:

$f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$

$f(n) = \Theta(g(n))$ if $g(n) = O(f(n)$ and $f(n) = O(g(n))$

$f(n) = o(g(n))$ if for every constant c there is an n_0 such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

$f(n) = \omega(g(n))$ if $g(n) = o(f(n))$.

The O-notation

Definition:
For two functions f and g, we write $f(n) = O(g(n))$, if there are some constants $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Suppose that $f(n) = O(n \log n)$ is an upper bound on the runtime of an algorithm A any input of size n for any n.

Then we say that the runtime of A is $O(n \log n)$.

You can of course replace $O(n \log n)$ with any other function $g(n)$.

More definitions:

$f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$

$f(n) = \Theta(g(n))$ if $g(n) = O(f(n)$ and $f(n) = O(g(n))$

$f(n) = o(g(n))$ if for every constant c there is an n_0 such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

$f(n) = \omega(g(n))$ if $g(n) = o(f(n))$.

The O-notation

Definition:
For two functions f and g, we write $f(n) = O(g(n))$, if there are some constants $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Suppose that $f(n) = O(n \log n)$ is an upper bound on the runtime of an algorithm A any input of size n for any n.

Then we say that the runtime of A is $O(n \log n)$.

You can of course replace $O(n \log n)$ with any other function $g(n)$.

More definitions:

$f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$

$f(n) = \Theta(g(n))$ if $g(n) = O(f(n)$ and $f(n) = O(g(n))$

$f(n) = o(g(n))$ if for every constant c there is an n_0 such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

$f(n) = \omega(g(n))$ if $g(n) = o(f(n))$.

Upper Bounds vs Lower Bounds

An upper bound:
Suppose that $f(n) = O(n \log n)$ is an upper bound on the runtime of an algorithm A any input of size n for any n.

Then we say that the runtime of A is $O(n \log n)$.

Upper Bounds vs Lower Bounds

An upper bound:
Suppose that $f(n) = O(n \log n)$ is an upper bound on the runtime of an algorithm A any input of size n for any n.

Then we say that the runtime of A is $O(n \log n)$.

Need to hold for any possible input instance!

Upper Bounds vs Lower Bounds

An upper bound:
Suppose that $f(n) = O(n \log n)$ is an upper bound on the runtime of an algorithm A any input of size n for any n.

Then we say that the runtime of A is $O(n \log n)$.

Need to hold for any possible input instance!

A lower bound:
For any constant n_0, there is an input instance of size $n \geq n_0$ such that the runtime of an algorithm A is $f(n) = \Omega(n \log n)$.

Then we say that the runtime of A is $\Omega(n \log n)$.

Enough to find *one* input instance that scales arbitrarily large.

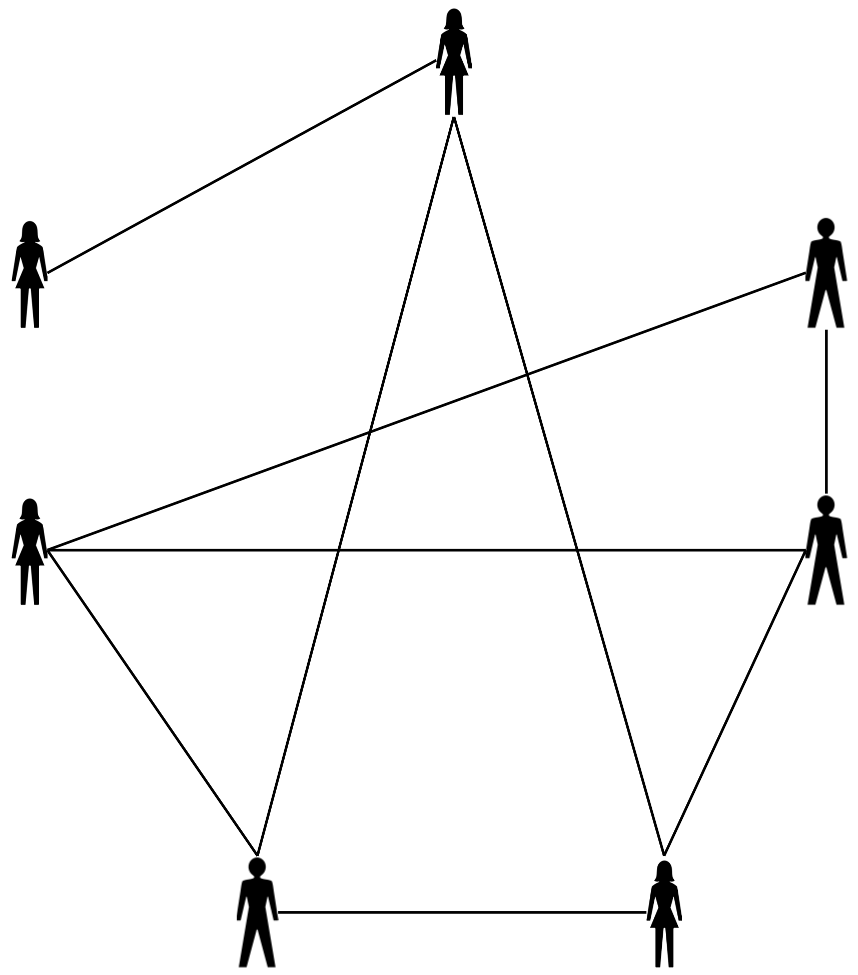
Outline

- Why do we want Formal Proofs?
- Asymptotic analysis
- Graphs
    - Recap on terminology

Graphs

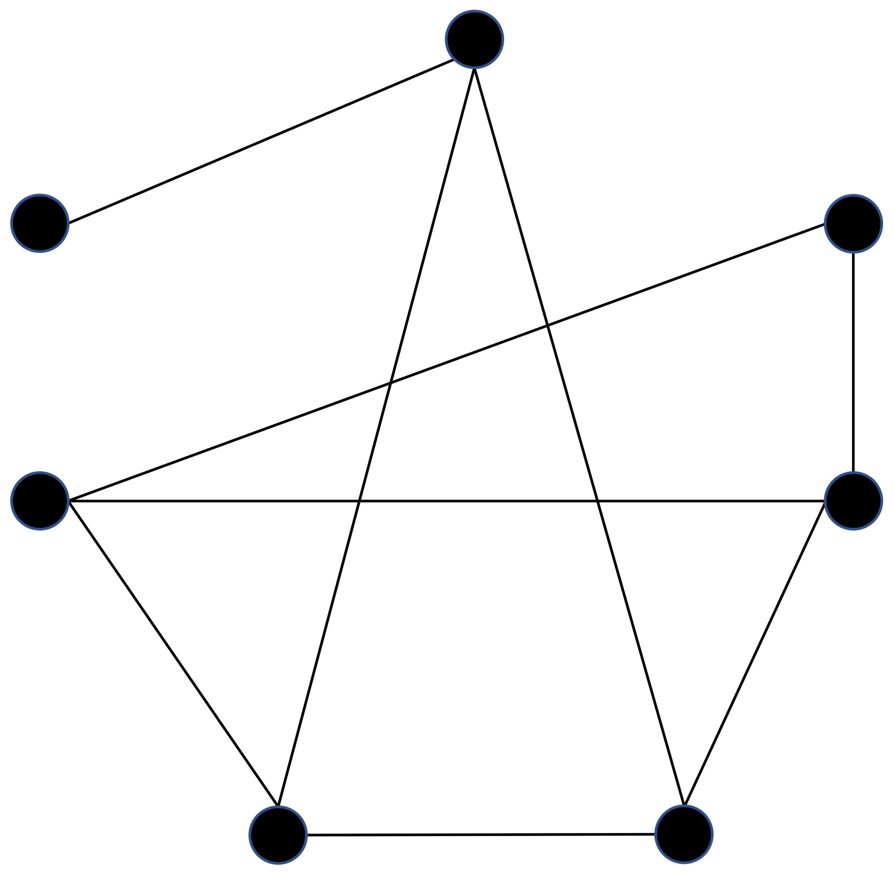
- A set V of entities, the **nodes**:
  - People in a social network
  - Junctions in a road network
  - Entries in a database
  - ...

- A set $E \subseteq (V \times V)$ of relations, the **edges:**
  - Who knows who
  - Junctions connected by a road
  - Fields have the same value



Graphs

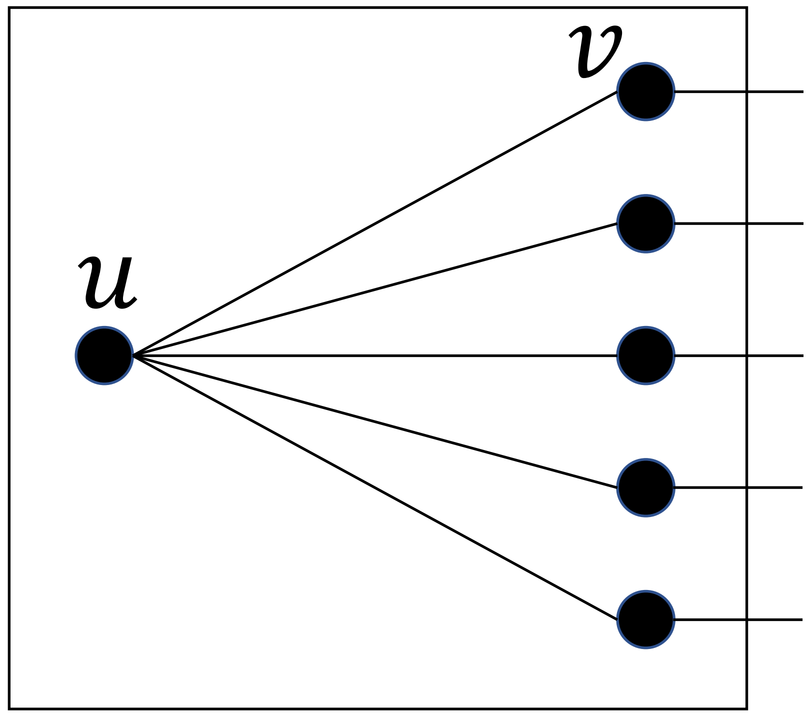
- A set V of entities, the **nodes**:
  - People in a social network
  - Junctions in a road network
  - Entries in a database
  - …
- A set $E \subseteq (V \times V)$ of relations, the **edges:**
  - Who knows who
  - Junctions connected by a road
  - Fields have the same value



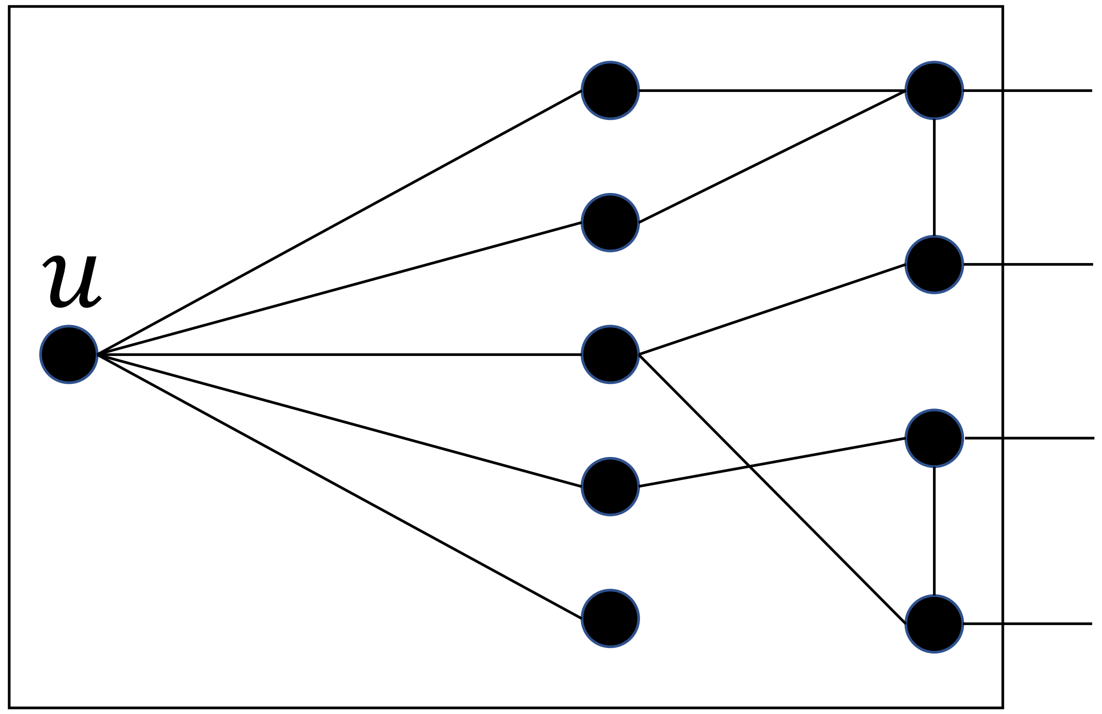
Graphs: Useful Terms

Undirected graph $G = (V, E)$

- Node v is called a *neighbor* of u if $\{u, v\} \in E$
- Neighborhood $N(u)$ of node u is the set of neighbors (and usually includes u).
- The i-hop neighborhood $N^i(u)$ of node u is the set of nodes within distance i
- Degree $\deg(u) = |N(u)| - 1$
- The maximum degree of G is $\Delta = \Delta(G) = \max\{\deg(v) \mid v \in V\}$



$N^1(u) = N(u)$



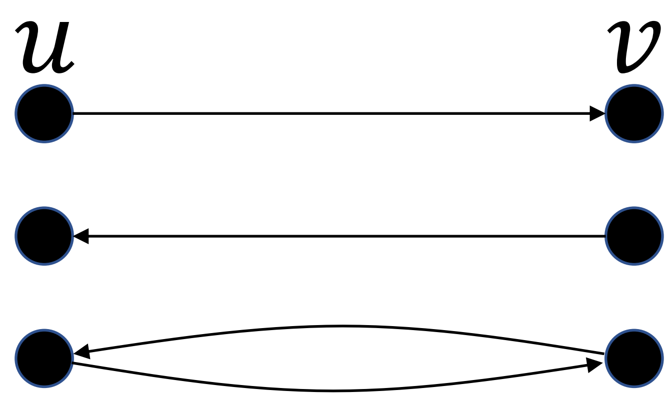
$N^2(u)$

Graphs: Useful Terms

Undirected graph $G = (V, E)$

- Node v is called a *neighbor* of u if $\{u, v\} \in E$
- Neighborhood $N(u)$ of node u is the set of neighbors (and usually includes u).
- The i-hop neighborhood $N^i(u)$ of node u is the set of nodes within distance i
- Degree $\deg(u) = |N(u)| - 1$
- The maximum degree of G is $\Delta = \Delta(G) = \max\{\deg(v) \mid v \in V\}$

In a directed graph, edges are directed from endpoint to the other



Subgraphs and Trees

Subgraph: $G' = (V', E')$ of $G = (V, E)$

- $V' \subseteq V$, $E' \subseteq E$.
- G' does not need to be connected.
- Sometimes we write: $G' \subseteq G$

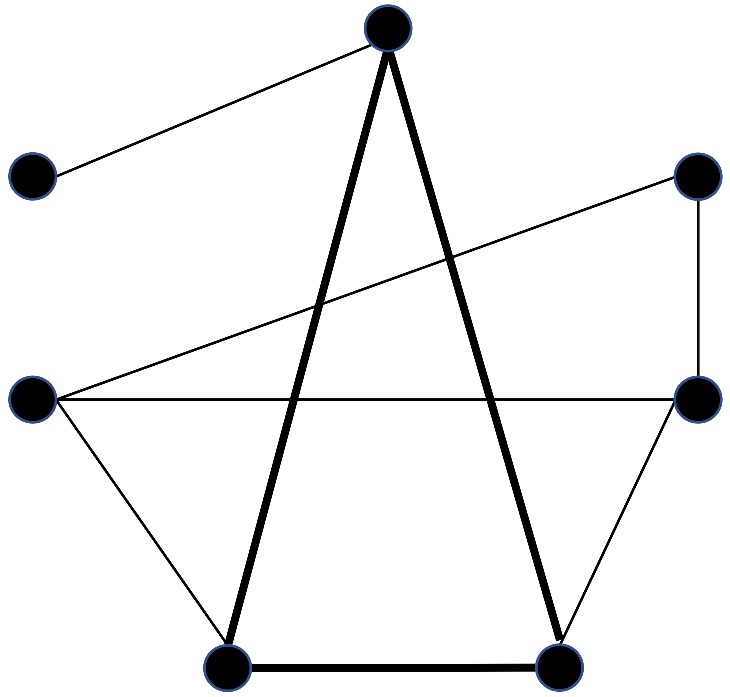
Cycle: $C = (V, E)$

- C is a connected graph
- For all $v \in V$, $\deg(v) = 2$

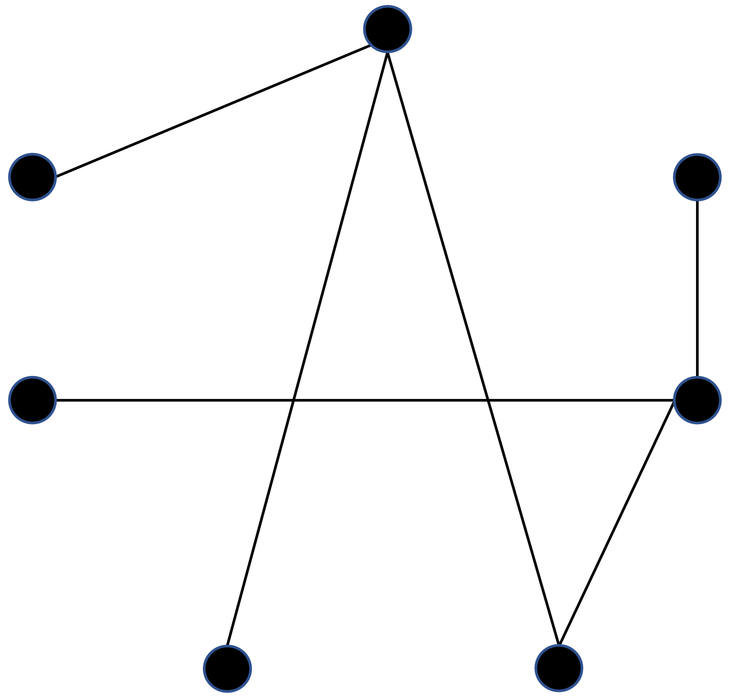
Tree: $T = (V, E)$

- T is a graph
- T *contains* no cycles, i.e., no subgraph of T is a cycle

Bold edges form a cycle of length 3 (triangle). This graph is not a tree.



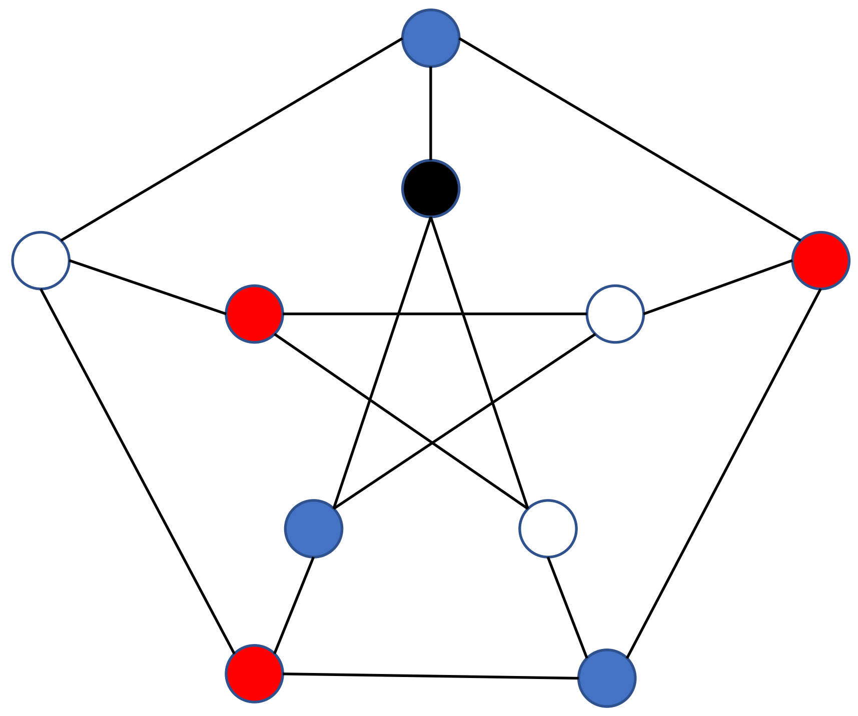
This graph is a tree.



Graph Colorings

Undirected graph $G = (V, E)$

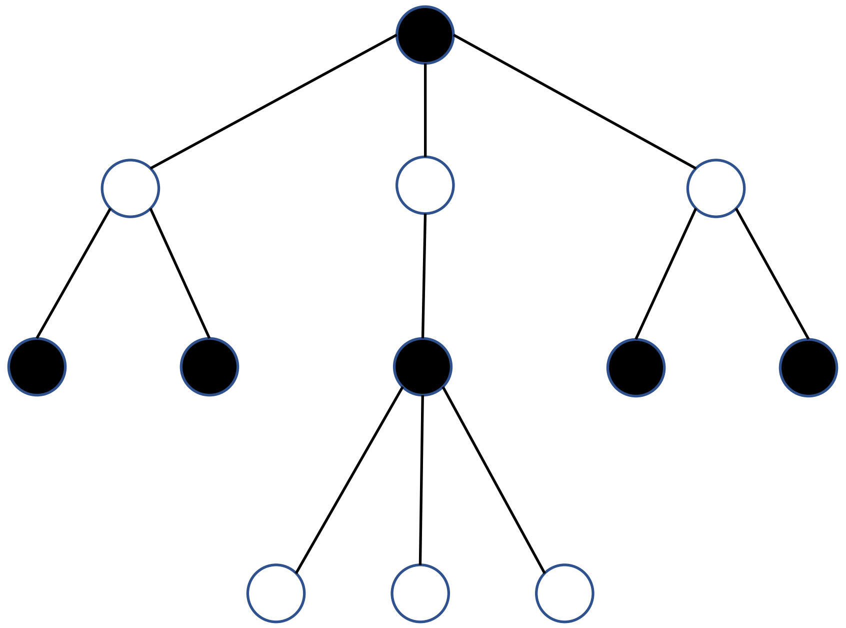
- A c-coloring of G is a mapping $\phi(v): V \to \{1, \dots, c\}$ such that for all nodes u and v that are neighbors, it holds that $\phi(v) \neq \phi(u)$



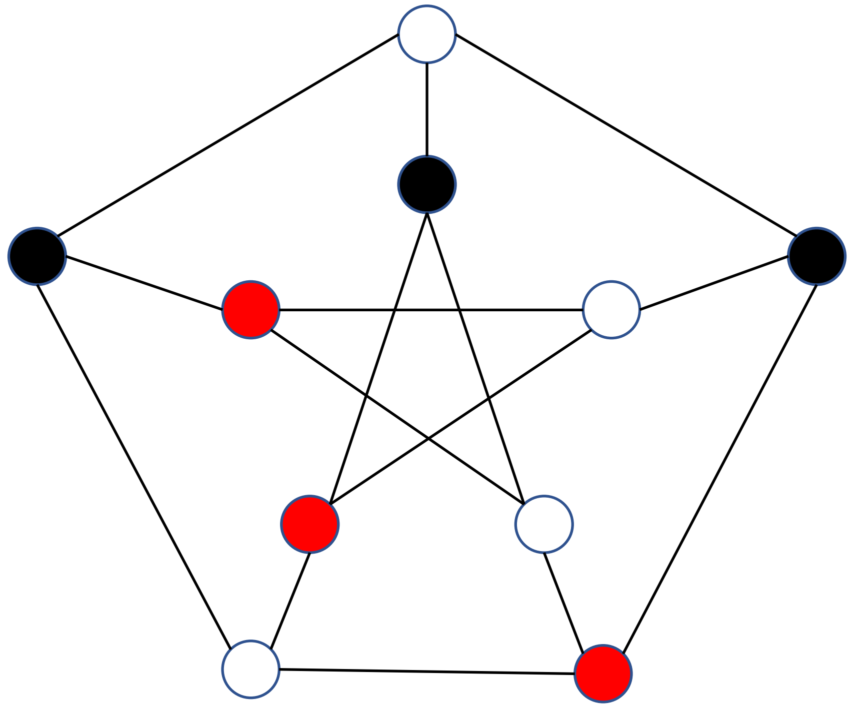
Petersen graph
4-coloring

Graph Colorings

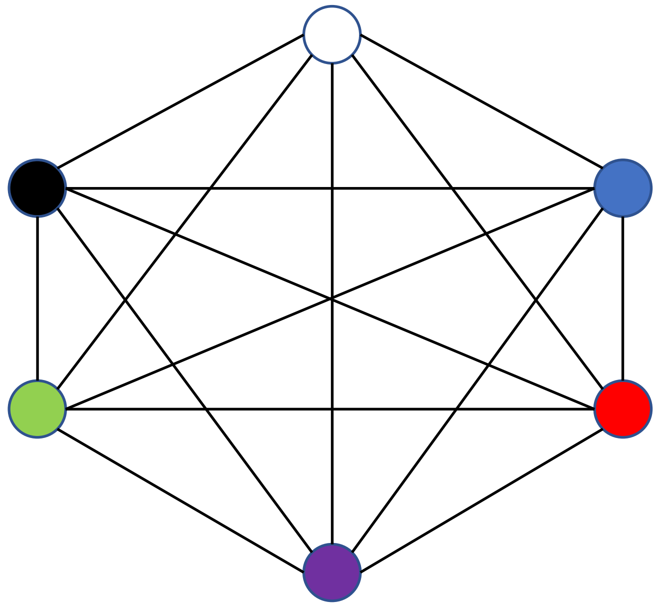
Chromatic number (or index) $\chi(G)$ is the smallest number of colors needed to color G



Tree
$\chi(G) = 2$



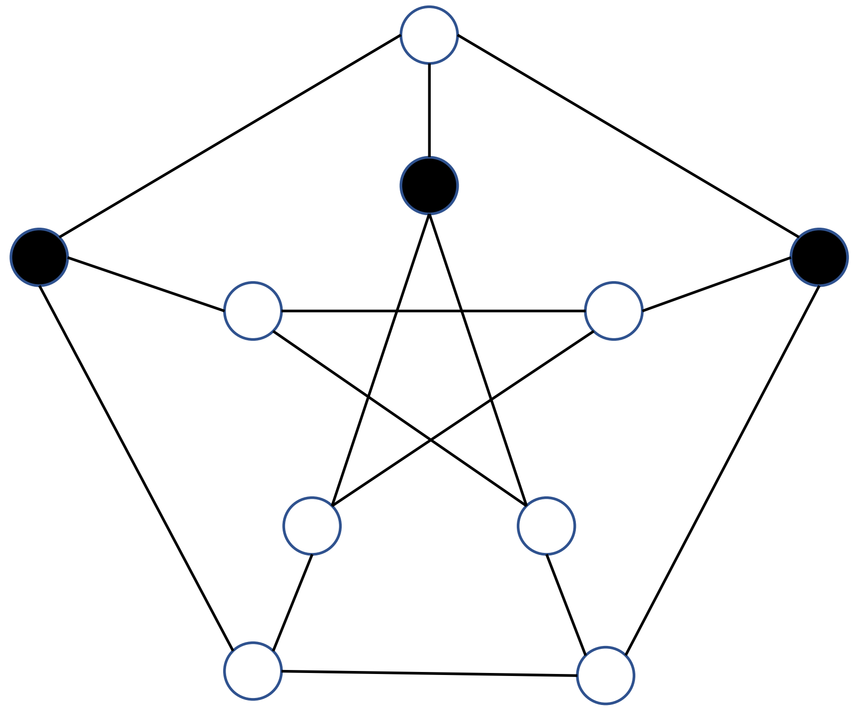
Petersen graph
$\chi(G) = 3$



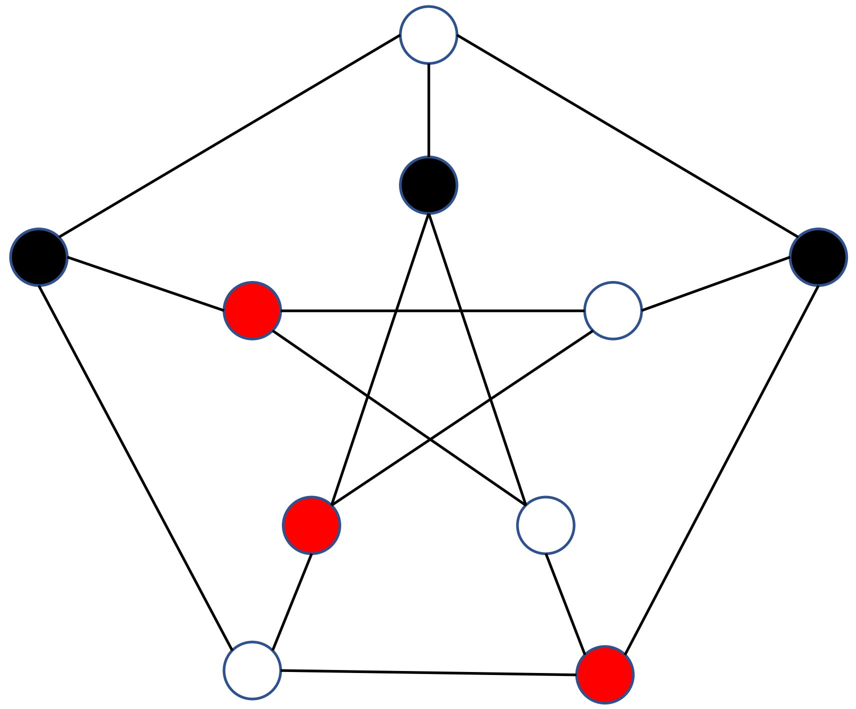
Δ-clique (complete graph)
$\chi(G) = \Delta + 1$

Independent Sets

An *independent set* $I \subseteq V$ such that no two nodes are adjacent, i.e., for any $u, v \in I$, $\{u, v\} \notin E$.

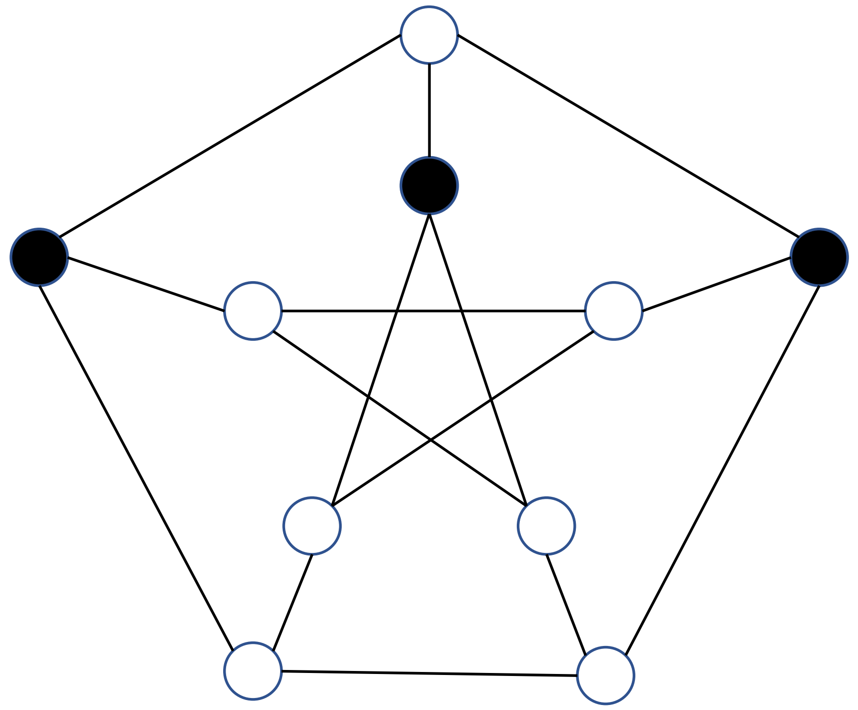


In a graph coloring, each color class is an independent set.



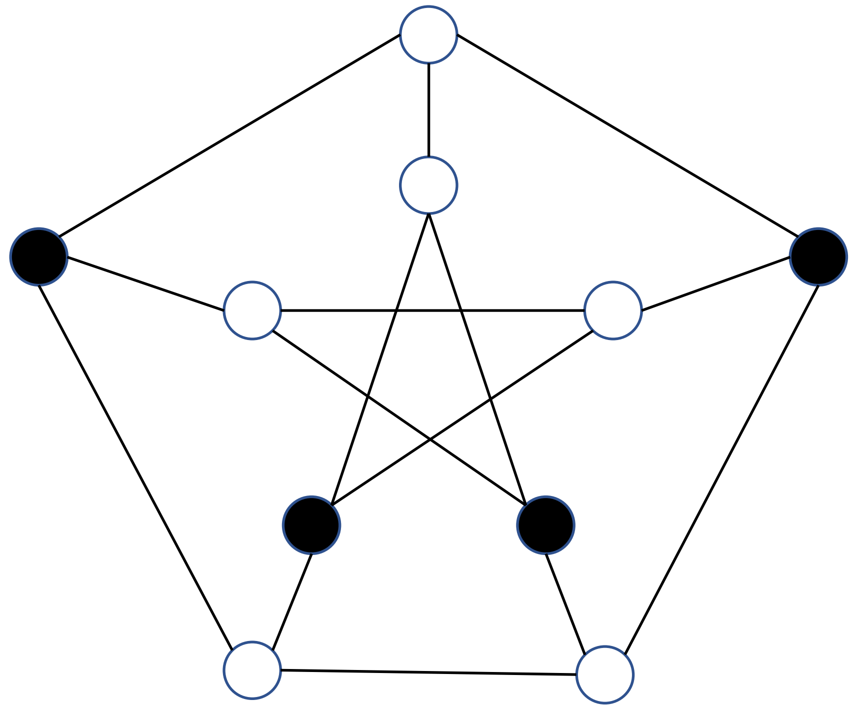
Independent Sets

An independent set is *maximal* if no node can be added to it without breaking the independence.



Maximal

A maximum independent set is an independent set of maximum size. The size of the maximum independent set is often referred to as the *independence number.*

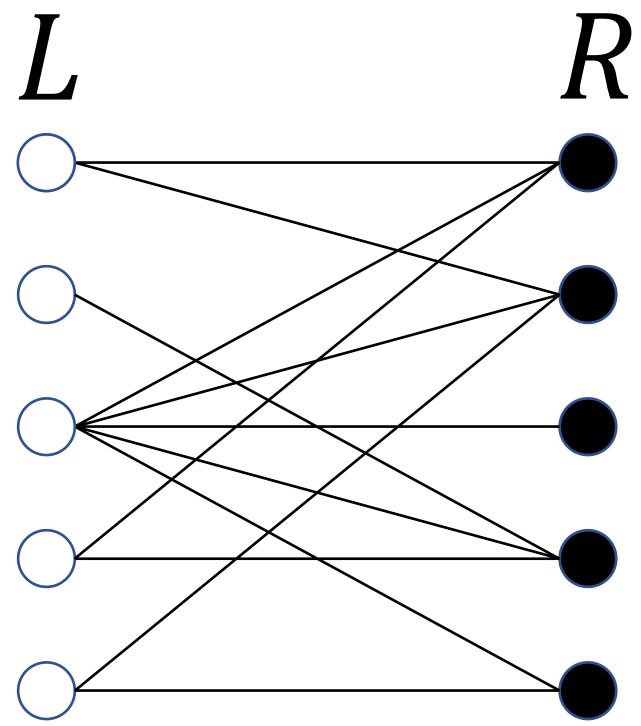


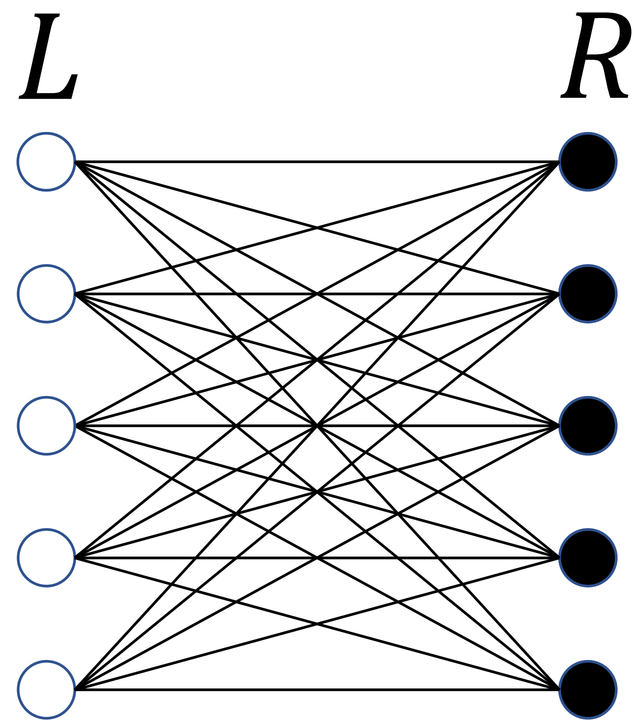
Maximum

Bipartite Graphs

Undirected graph $G = (L \cup R, E)$

- Is called *bipartite* if G can be two-colored
- Is called *complete bipartite* if $\{u, v\} \in E$ for all $u \in L$ and $v \in R$.



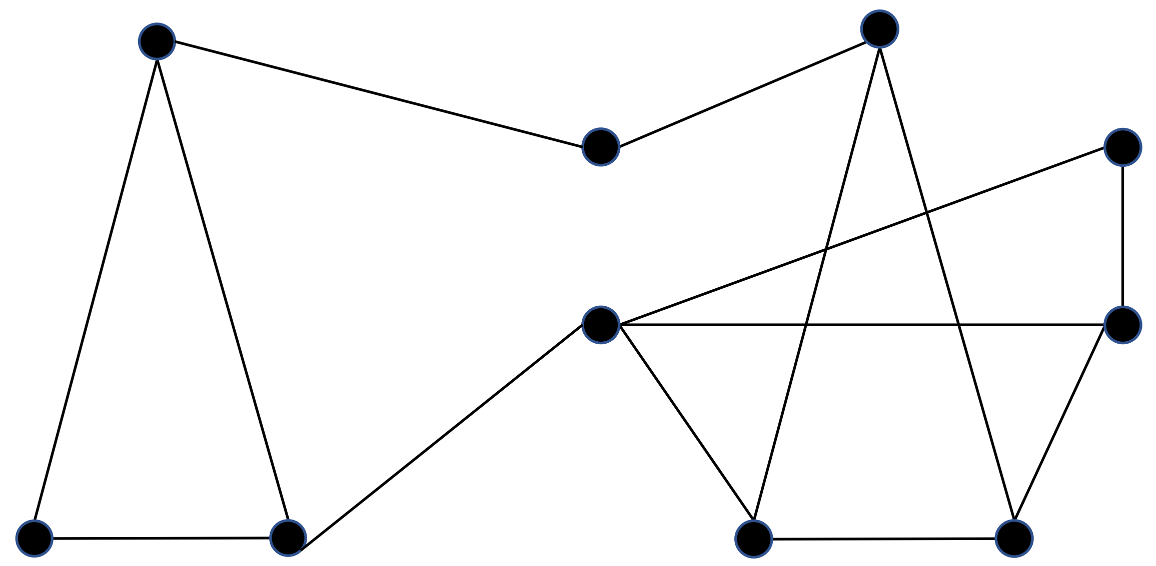


Complete bipartite

Graph Connectivity

An undirected graph $G = (V, E)$ is connected if there is a path between any pair of nodes $u, v \in V$.

A directed graph is connected if there is a directed path between any pair of nodes.

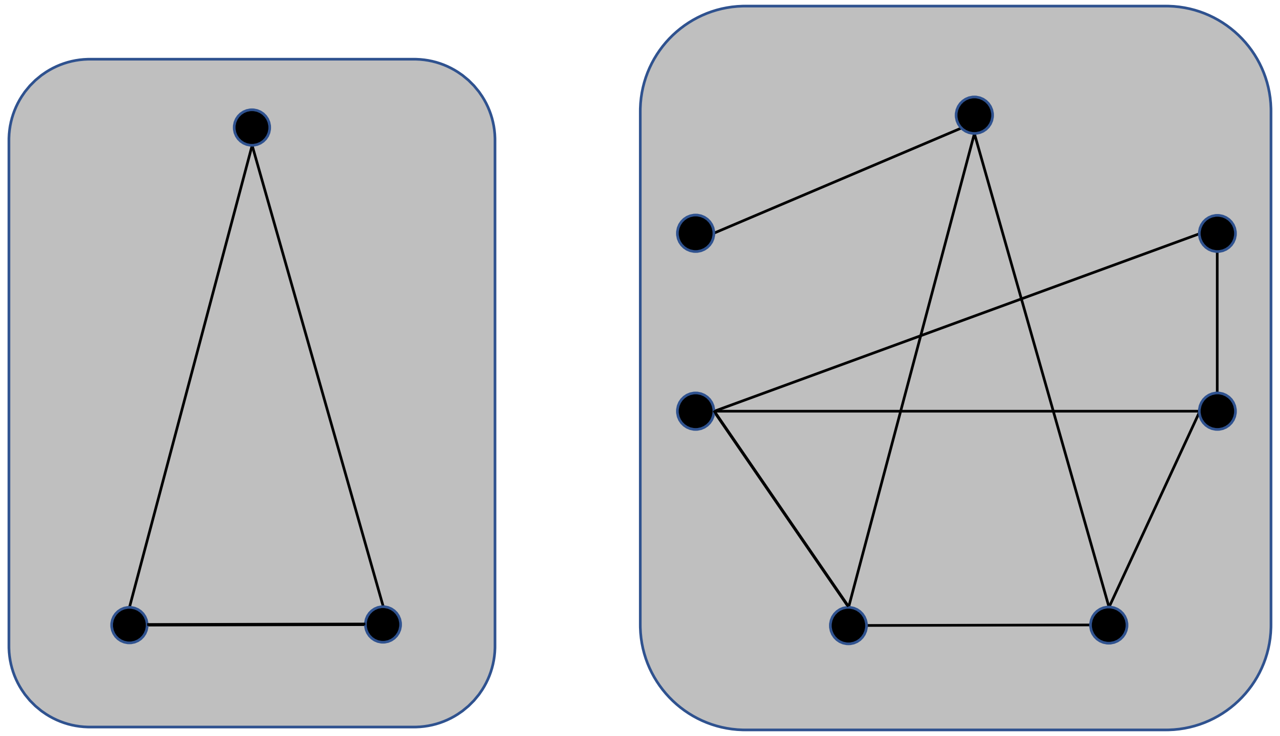


Connected graph

Graph Connectivity

An undirected graph $G = (V, E)$ is connected if there is a path between any pair of nodes $u, v \in V$.

A directed graph is connected if there is a directed path between any pair of nodes.

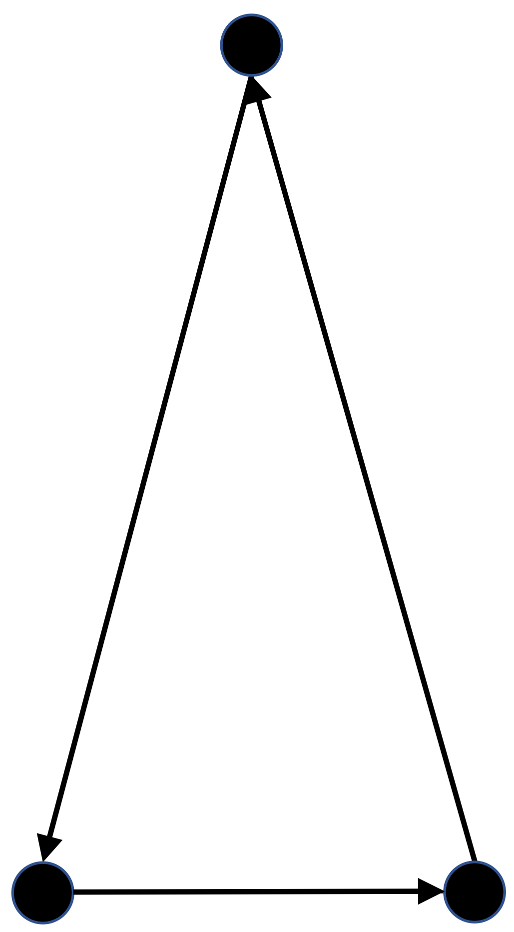


Connected components

Graph Connectivity

An undirected graph $G = (V, E)$ is connected if there is a path between any pair of nodes $u, v \in V$.

A directed graph is connected if there is a directed path between any pair of nodes.



The Handshaking Lemma

Lemma: Let $G = (V, E)$ be a graph.

$$\sum_{v \in V} \deg(v) = 2|E|$$

The Handshaking Lemma

Lemma: Let $G = (V, E)$ be a graph.

$$\sum_{v \in V} \deg(v) = 2|E|$$

Proof: In the summation, each edge $e \in E$ is counted exactly twice, once per endpoint.

The Handshaking Lemma

Lemma: Let $G = (V, E)$ be a graph.

$$\sum_{v \in V} \deg(v) = 2|E|$$

Proof: In the summation, each edge $e \in E$ is counted exactly twice, once per endpoint.

Consequence: There is always an even number of odd degree nodes in any graph.

For more applications, check for example the Seven Bridges of Königsberg (Euler tours)

Wrap-up

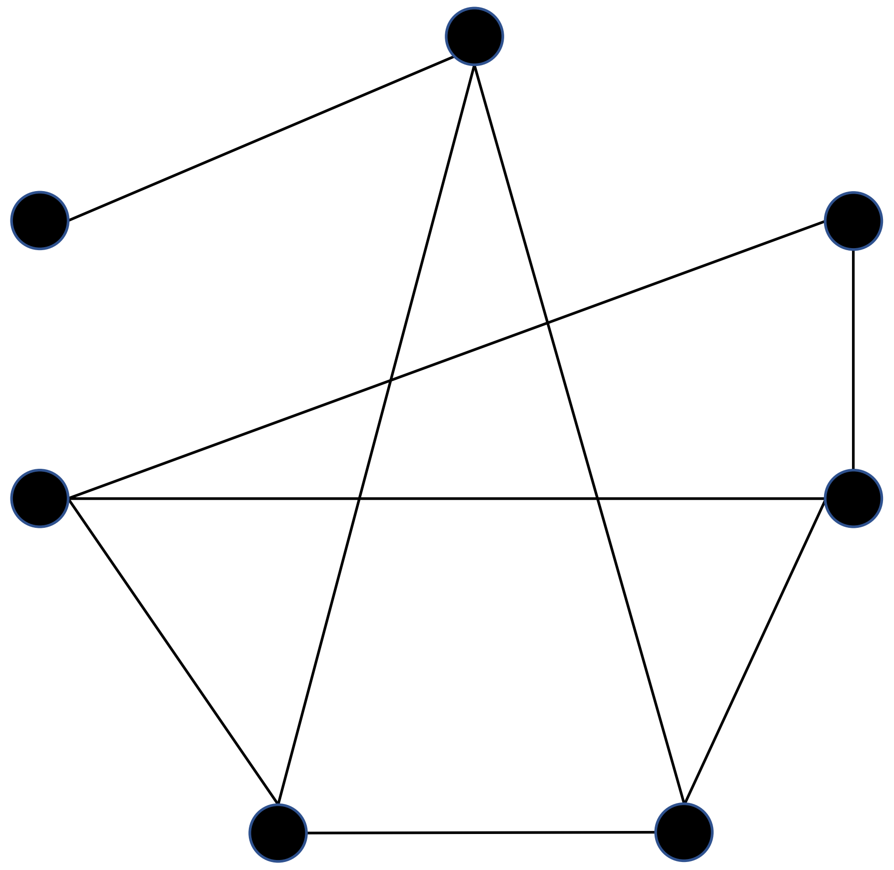
Correctness proof

I am sure that my algorithm works

Runtime analysis, usually worst case

I am sure that my algorithm *always* works fast

Asymptotic analysis



Graphs