IF223 - Algorithmique Distribuée

2021-2022

Rohan Fossé - rohan.fosse@labri.fr

Graph Theory : Reminder

Graph G = (V, E)

V = **set of nodes** (finite, non-empty)

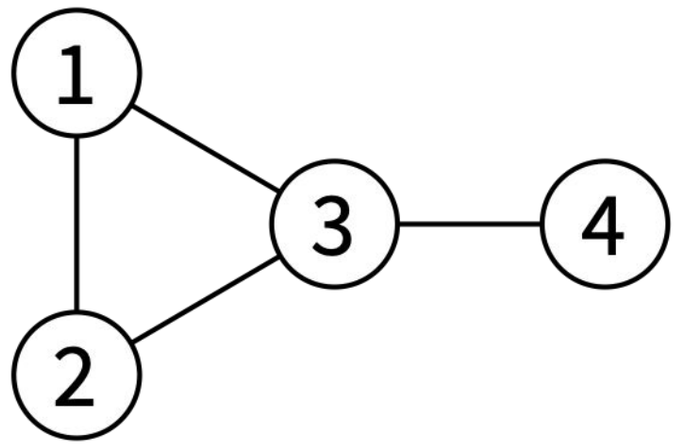
E = **set of edges** (unordered pairs of nodes)

$G = (V, E)$

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

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

G:



Graph G = (V, E)

V = **set of nodes** (finite, non-empty)

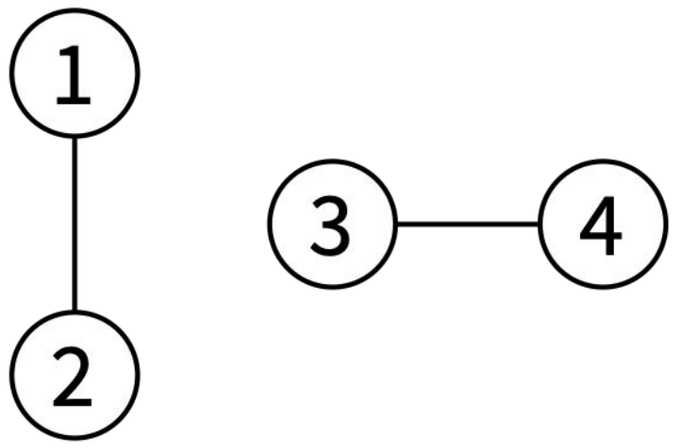
E = **set of edges** (unordered pairs of nodes)

$G = (V, E)$

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

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

G:



Graph G = (V, E)

V = **set of nodes** (finite, non-empty)

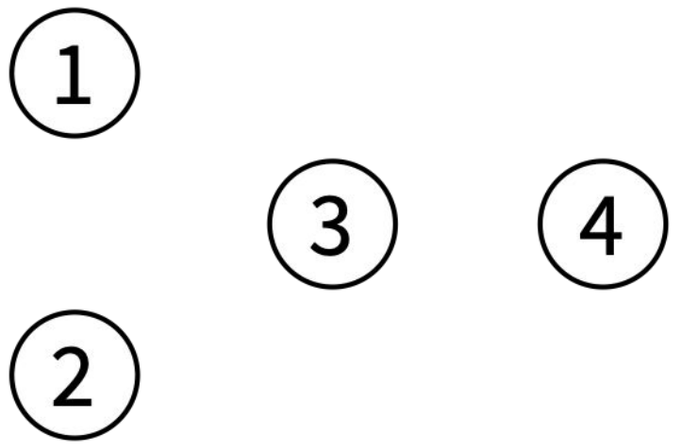
E = **set of edges** (unordered pairs of nodes)

$G = (V, E)$

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

$E = \varnothing$

G:



Graph G = (V, E)

V = **set of nodes** (a.k.a. "vertices")
E = **set of edges**

Usually nodes are denoted with *u*, *v*
(if more nodes needed: *s*, *t*, *u*, *v*, *u'*, *v'*, v_1, v_2, etc.),
edges are denoted with e, e', e_1, e_2, etc.

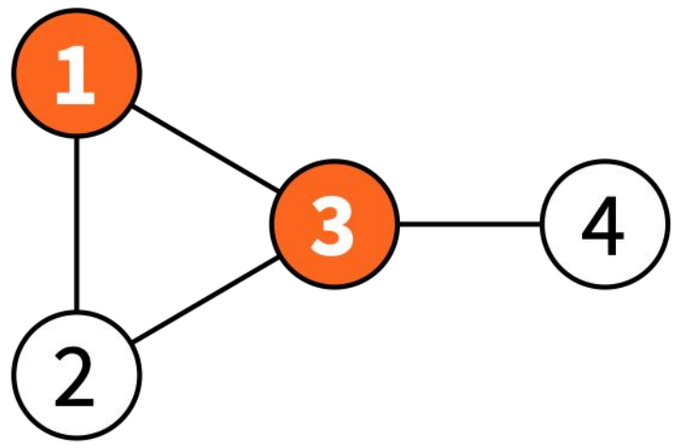
Convention: $n = |V|$, $m = |E|$

Graph G = (V, E)

u and *v* are **adjacent nodes**
= **nodes** *u* and *v* are **neighbours**
= there is an **edge** {*u*, *v*}

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

G:



Graph G = (V, E)

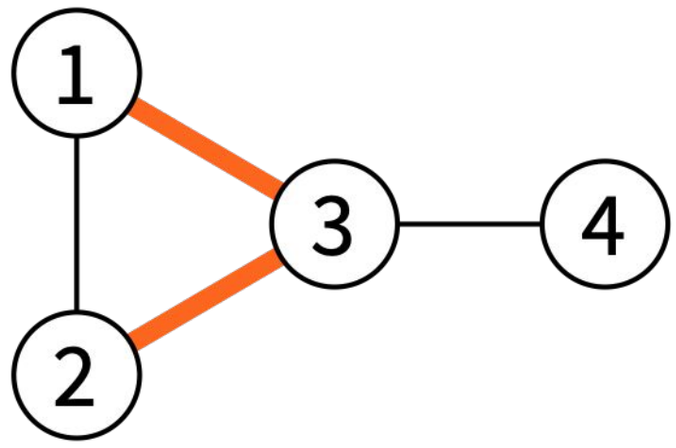
e_1 and e_2 are "adjacent edges"
= they share an endpoint
= their intersection is non-empty

$G = (V, E)$

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

$E = \{ \{1,2\}, \mathbf{\{1,3\}}, \mathbf{\{2,3\}}, \{3,4\} \}$

G:

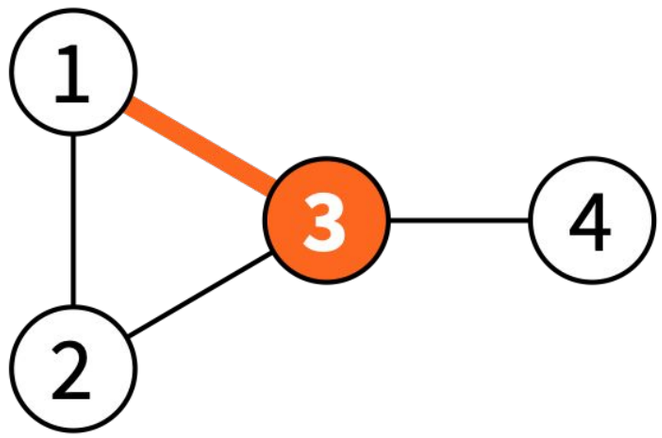


Graph G = (V, E)

Node *v* is incident to edge *e*
= *v* is an endpoint of *e*
= *v* is a member of *e*

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

G:



Graph G = (V, E)

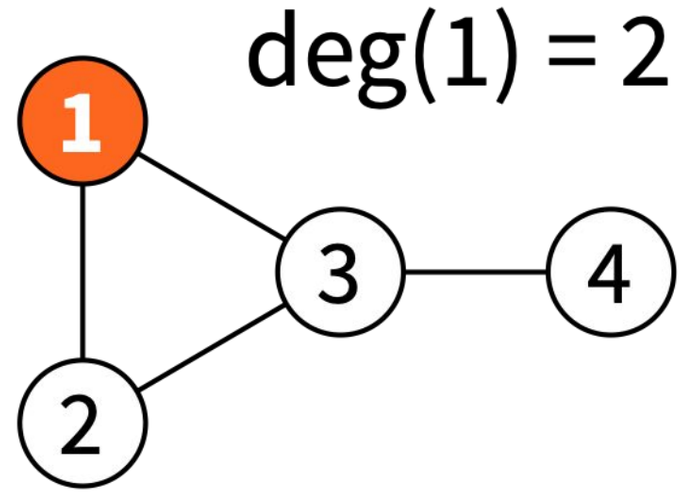
Node of degree *k*
= # node adjacent to *k* nodes
= # node incident to *k* edges

$G = (V, E)$

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

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

G:



Graph G = (V, E)

***k*-regular graph**

= all nodes have degree *k*

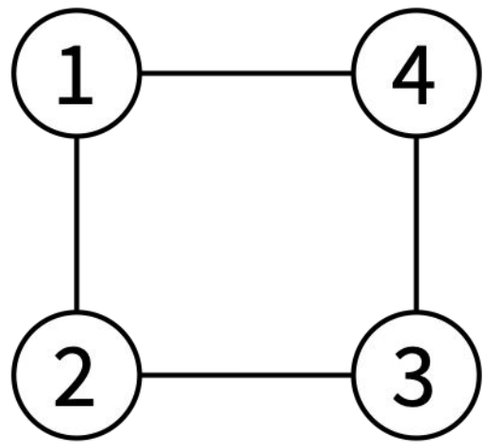
= all nodes have *k* neighbours

$G = (V, E)$

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

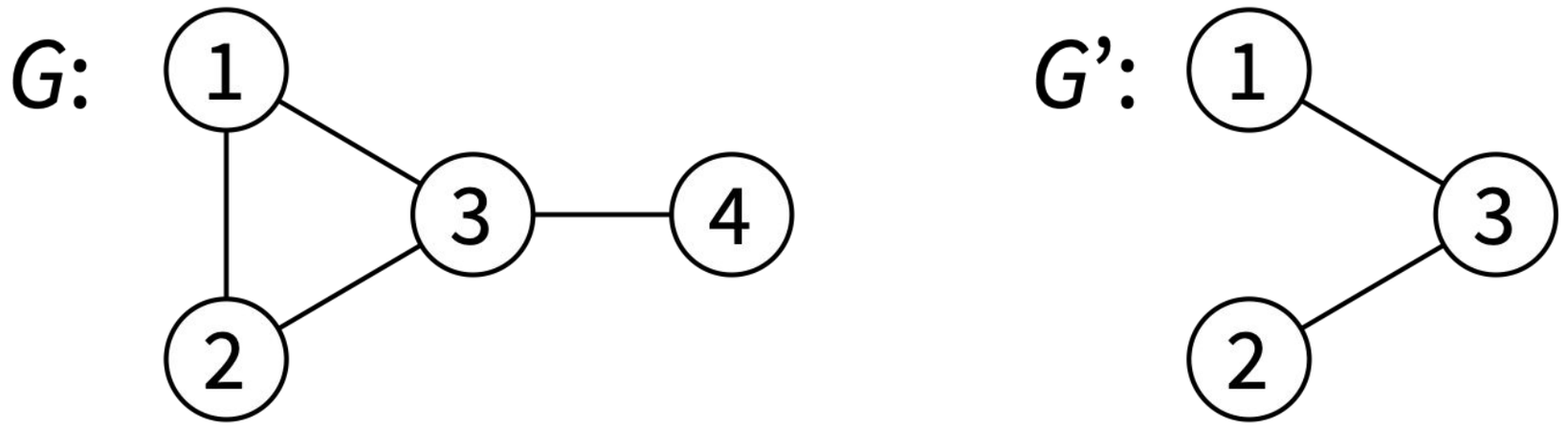
$E = \{ \{1,2\}, \{2,3\}, \{3,4\}, \{1,4\} \}$

G:



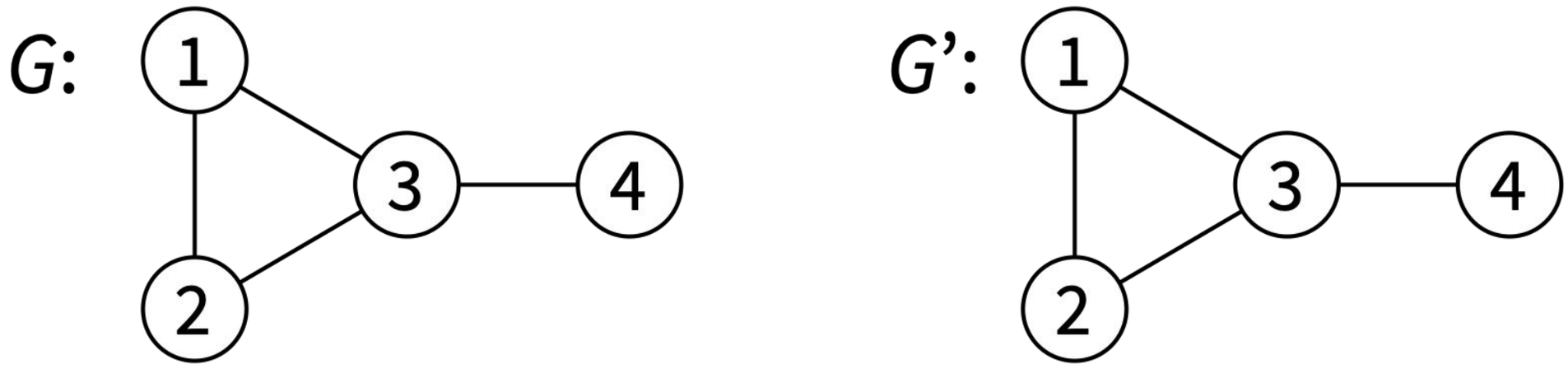
Subgraph

Graph $G' = (V', E')$ is a subgraph of $G = (V, E)$:
$V' \subseteq V$ and $E' \subseteq E$



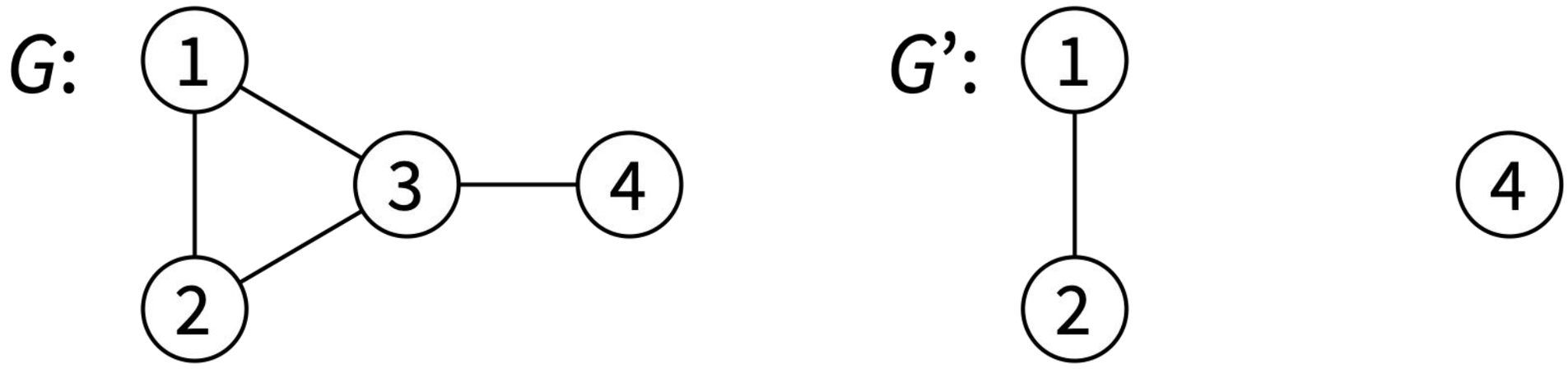
Subgraph

Graph $G' = (V', E')$ is a subgraph of $G = (V, E)$:
$V' \subseteq V$ and $E' \subseteq E$



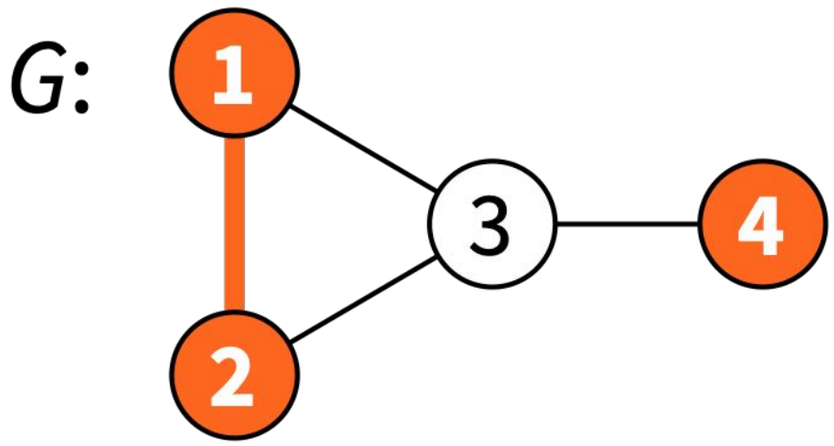
Subgraph

Graph $G' = (V', E')$ is a subgraph of $G = (V, E)$:
$V' \subseteq V$ and $E' \subseteq E$



Subgraph

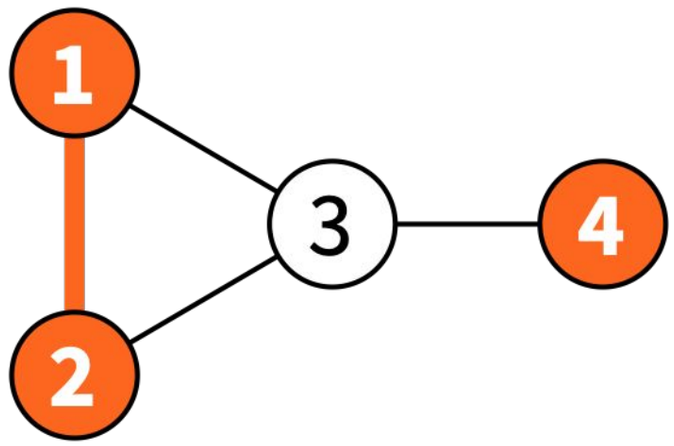
Graph $G' = (V', E')$ is a subgraph of $G = (V, E)$:
$V' \subseteq V$ and $E' \subseteq E$



Induced Subgraph

Subgraph induced by nodes *V'*
= all nodes of *V'* and all edges that connect them

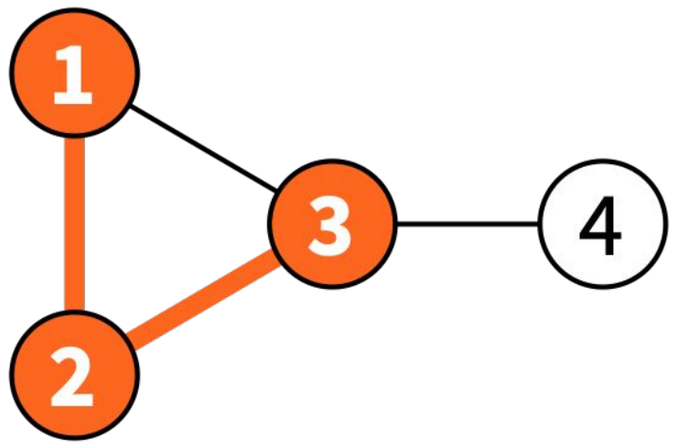
G:



Induced Subgraph

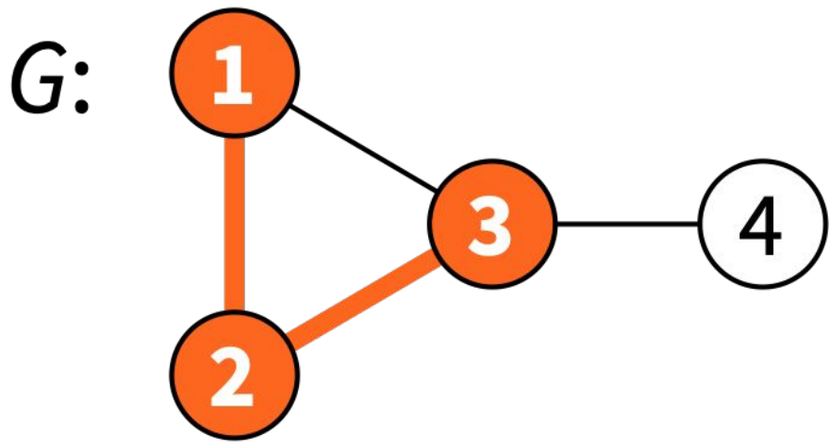
Subgraph induced by edges *E'*
= all edges of *E'* and all of their endpoints

G:



Induced Subgraph

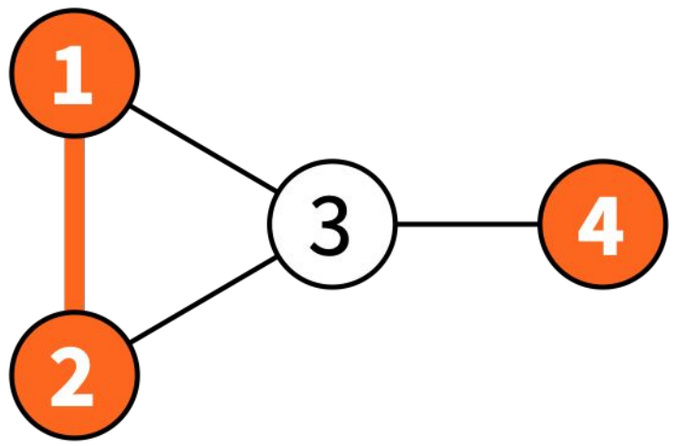
This is *not* a subgraph induced by any set of nodes — why?



Induced Subgraph

This is *not* a subgraph induced by any set of edges — why?

G:

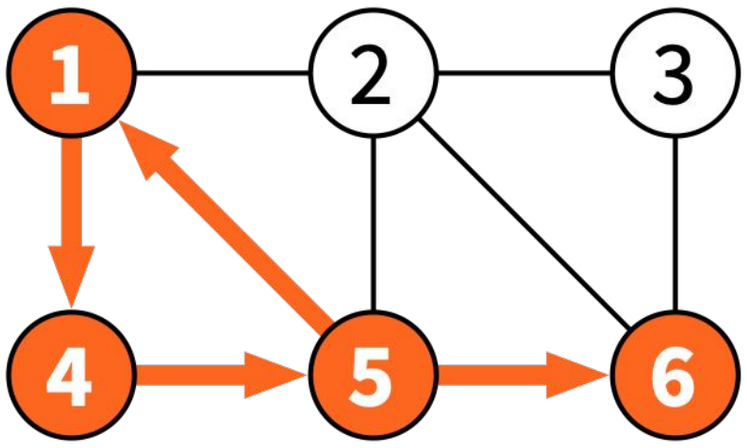


Walks, paths and connectivity

Walk = alternating sequence of incident nodes and edges

$w = (5, \{5,1\}, 1, \{1,4\}, 4, \{4,5\}, 5, \{5,6\}, 6)$

G:



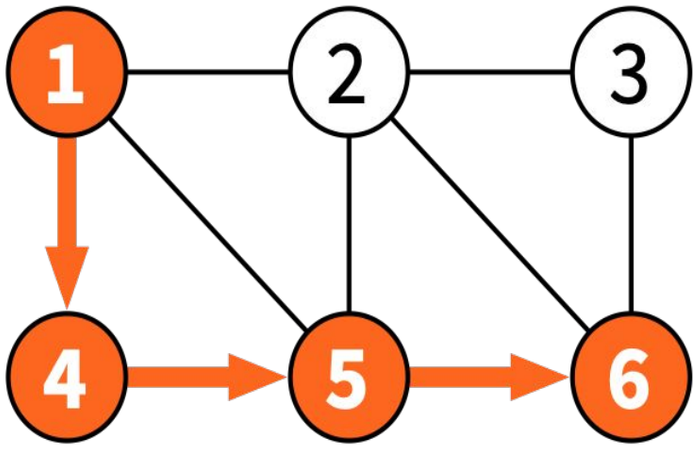
Walks, paths and connectivity

Path = walk visiting each node at most once

Length of a path = number of *edges*

$w = (1, \{1,4\}, 4, \{4,5\}, 5, \{5,6\}, 6)$

G:

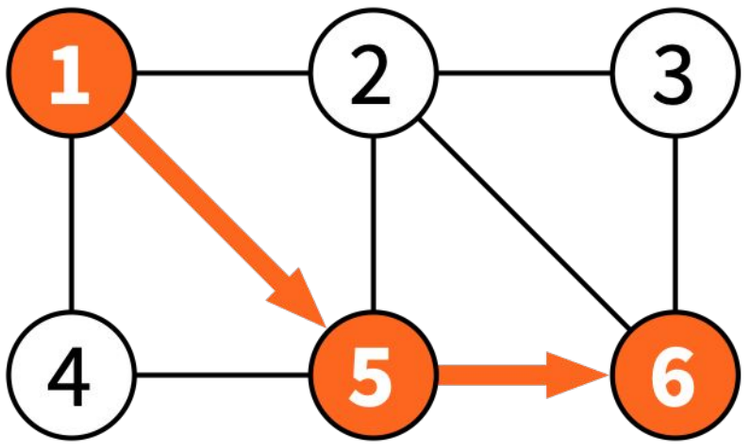


Walks, paths and connectivity

Distance = length of a ***shortest path***

$w = (1, \{1,5\}, 5, \{5,6\}, 6)$

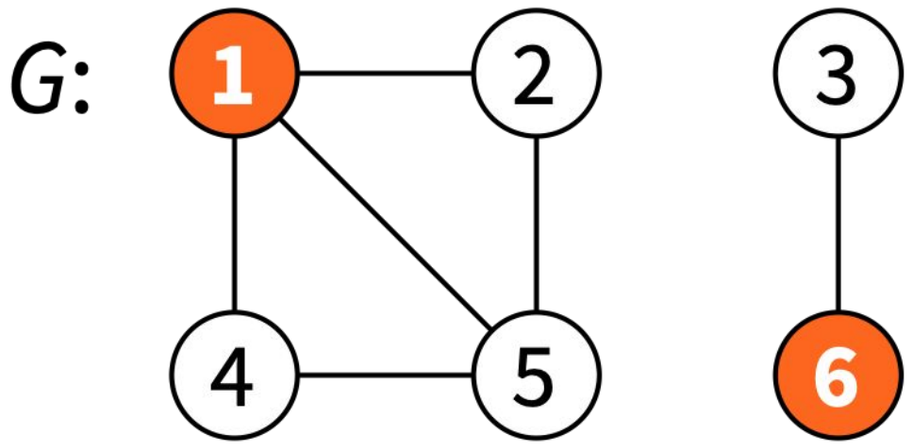
G:



dist(1, 6) = 2

Walks, paths and connectivity

Distance = length of a *shortest path*
(infinite if no such path exists)



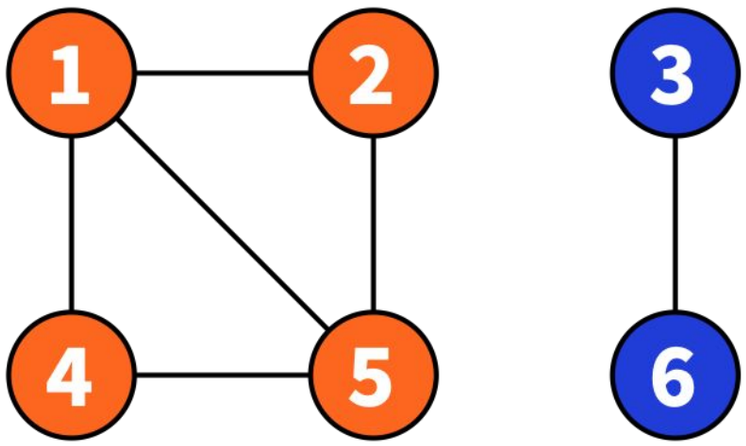
dist(1, 6) = ∞

Walks, paths and connectivity

Connected component C**:**

there is a path between any two nodes of C

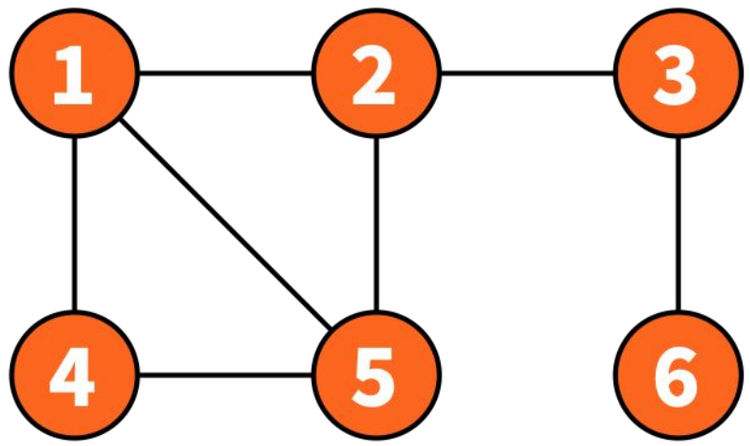
G:



Walks, paths and connectivity

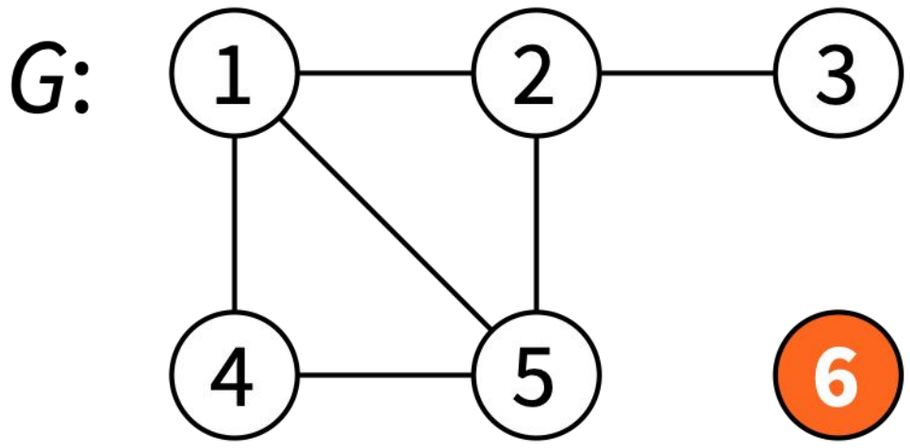
Graph is connected if only 1 connected component

G:



Walks, paths and connectivity

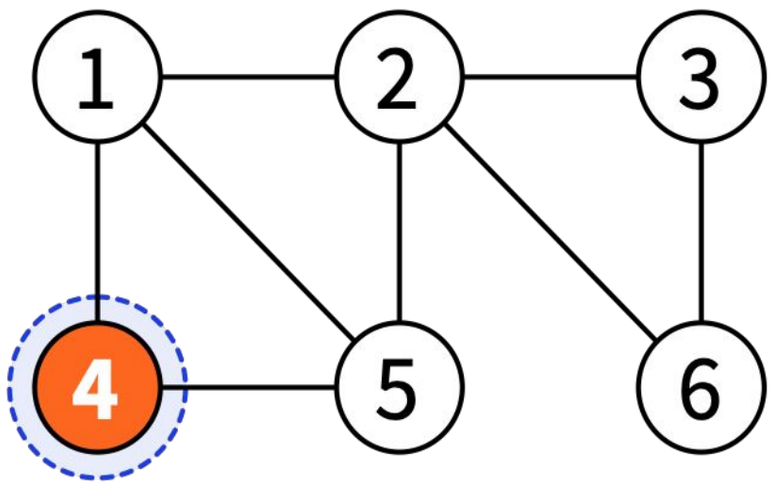
Isolated node = node of degree 0



Walks, paths and connectivity

ball(v, r) = "radius-*r* neighbourhood of *v*"
= nodes at distance at most *r* from node *v*

G:

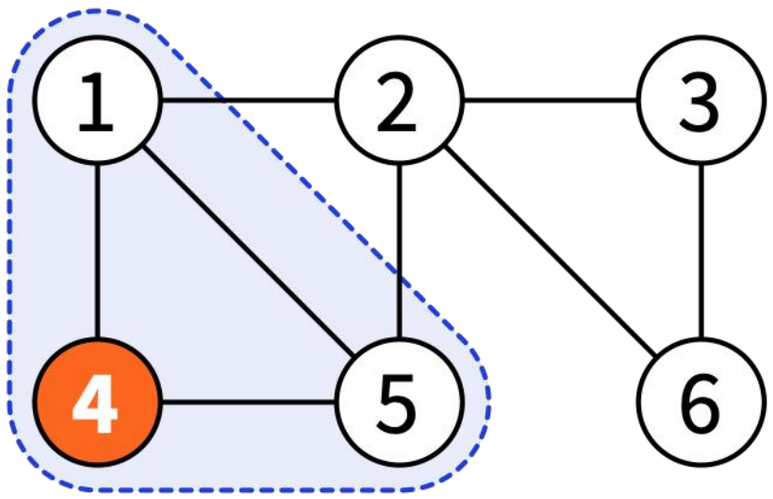


ball(4, 0) = {4}
ball(4, 1) = {4, 1, 5}
ball(4, 2) = {4, 1, 5, 2}
ball(4, 3) = *V*

Walks, paths and connectivity

ball(v, r) = "radius-*r* neighbourhood of *v*"
= nodes at distance at most *r* from node *v*

G:



ball(4, 0) = {4}

ball(4, 1) = {4, 1, 5}

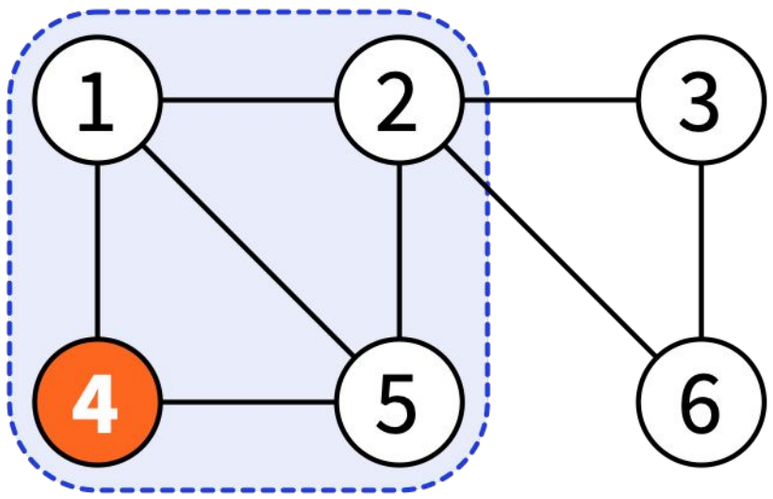
ball(4, 2) = {4, 1, 5, 2}

ball(4, 3) = *V*

Walks, paths and connectivity

ball(v, r) = "radius-*r* neighbourhood of *v*"
= nodes at distance at most *r* from node *v*

G:

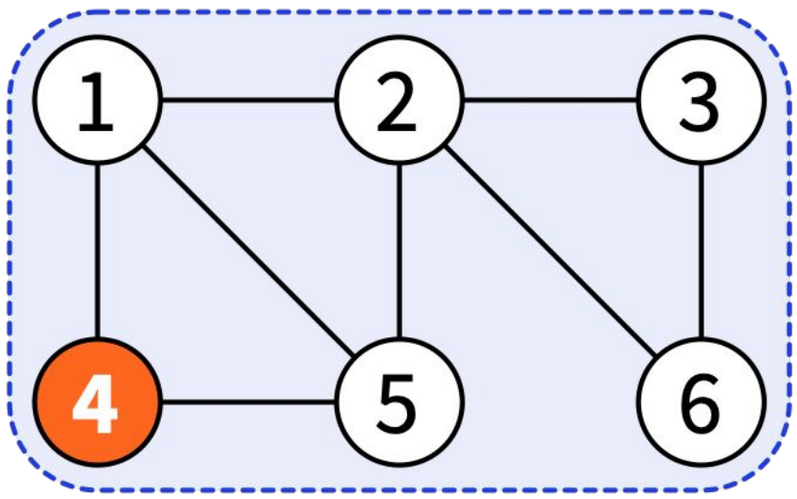


ball(4, 0) = {4}
ball(4, 1) = {4, 1, 5}
ball(4, 2) = {4, 1, 5, 2}
ball(4, 3) = *V*

Walks, paths and connectivity

ball(v, r) = "radius-*r* neighbourhood of *v*"
= nodes at distance at most *r* from node *v*

G:

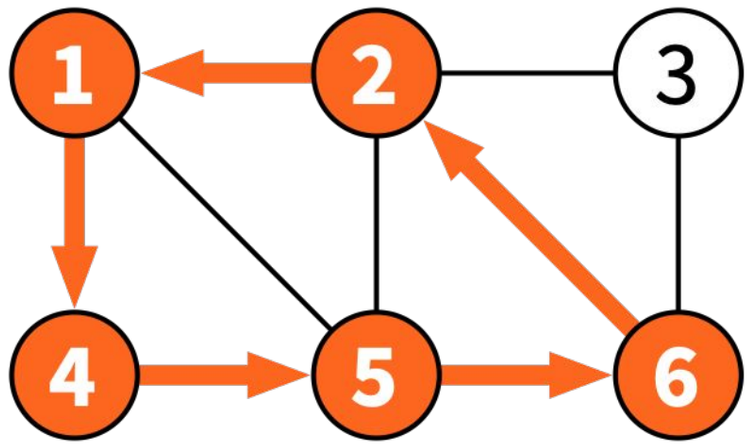


ball(4, 0) = {4}
ball(4, 1) = {4, 1, 5}
ball(4, 2) = {4, 1, 5, 2}
ball(4, 3) = *V*

Walks, paths and connectivity

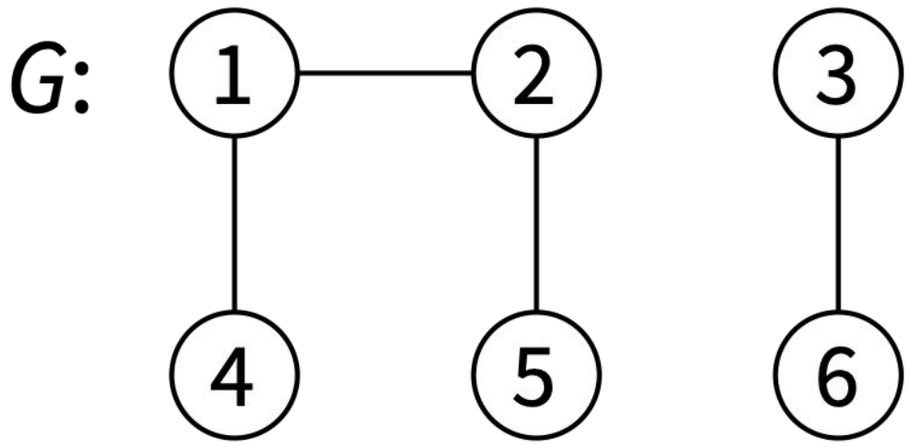
Cycle = closed walk that visits each node and each edge at most once (length ≥ 3)

G:



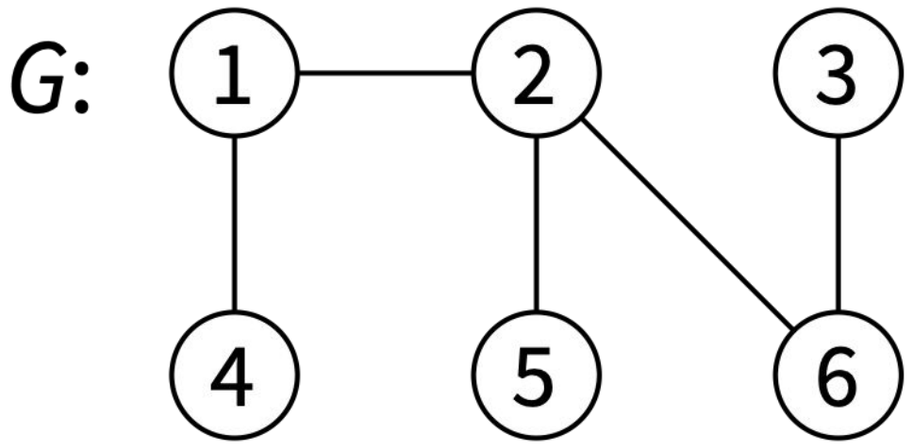
Walks, paths and connectivity

Acyclic graph = graph without any cycles



Walks, paths and connectivity

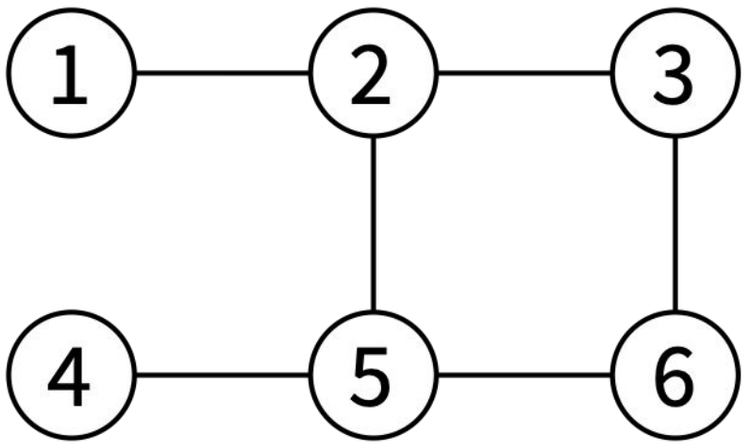
Tree = connected acyclic graph
Forest = acyclic graph



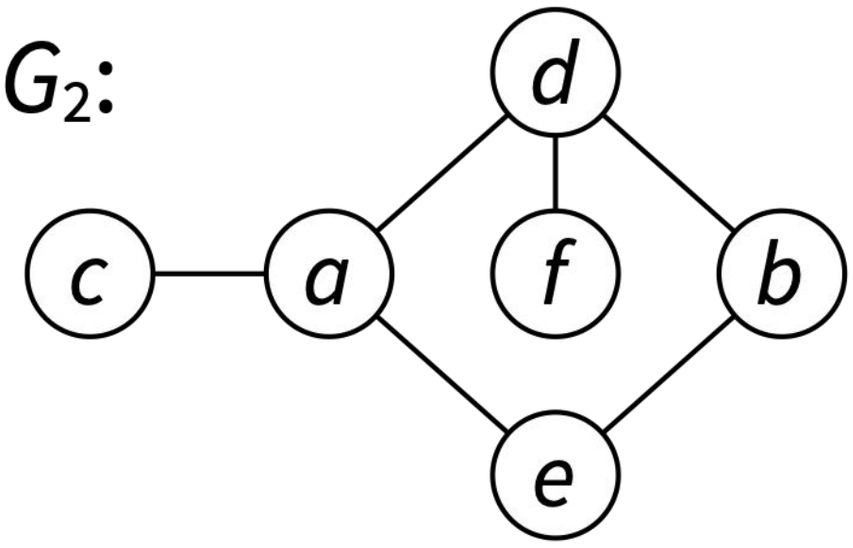
Isomorphism

Isomorphism from $G_1 = (V_1, E_1)$ to $G_2 = (V_2, E_2)$: bijection f: $V_1 \rightarrow V_2$ that preserves adjacency

G_1:



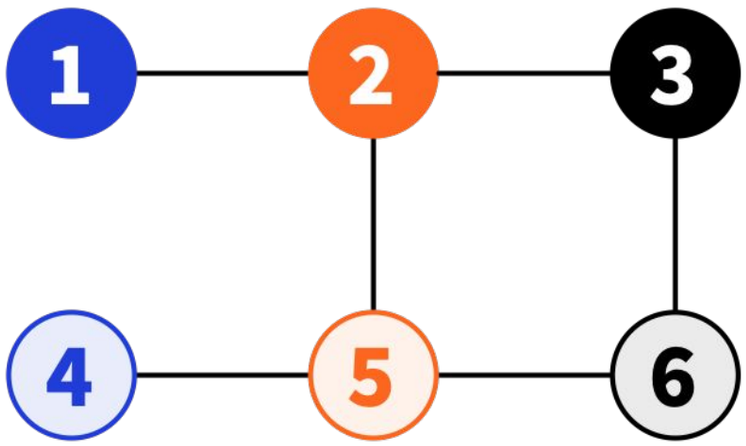
G_2:



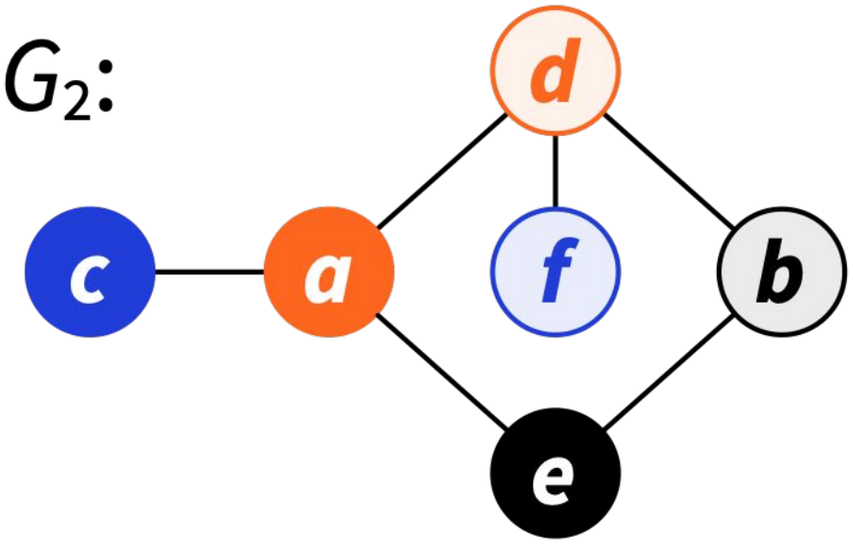
Isomorphism

Isomorphism from $G_1 = (V_1, E_1)$ to $G_2 = (V_2, E_2)$: bijection f: $V_1 \longrightarrow V_2$ that preserves adjacency

G_1:



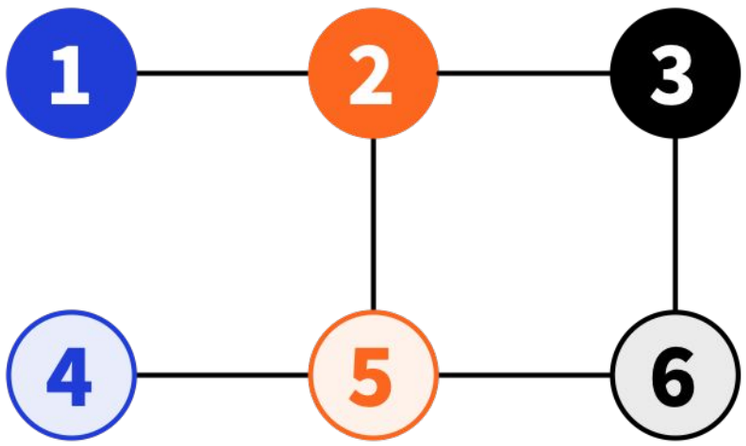
G_2:



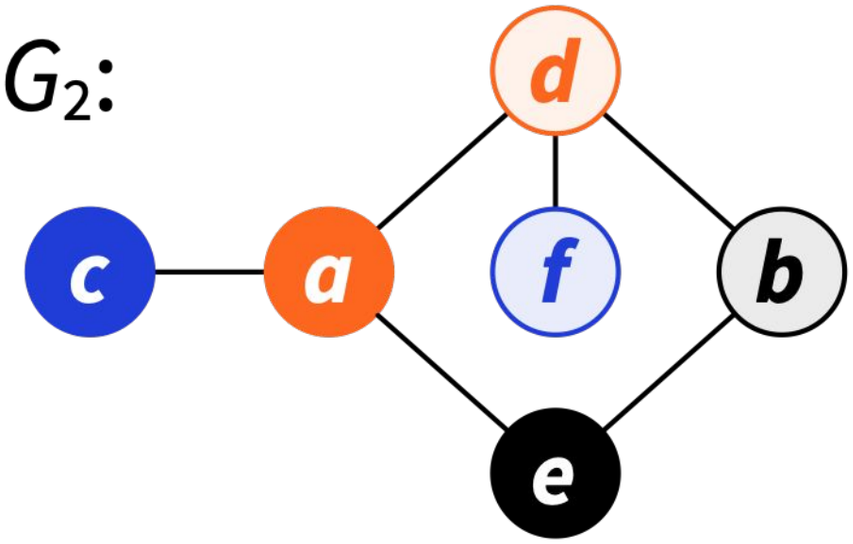
Isomorphism

Graphs are isomorphic if there exists an isomorphism from one to another

G_1:

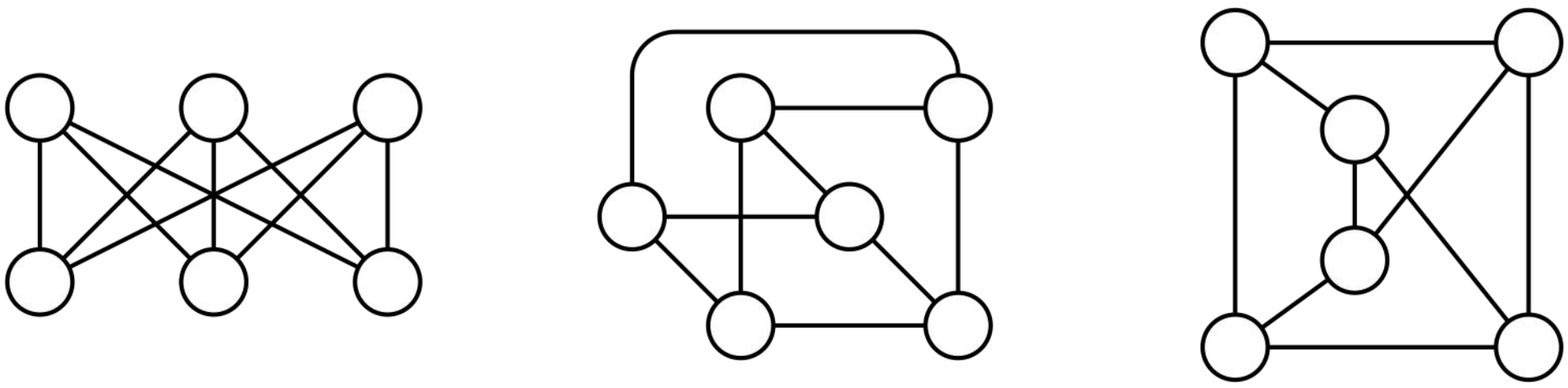


G_2:



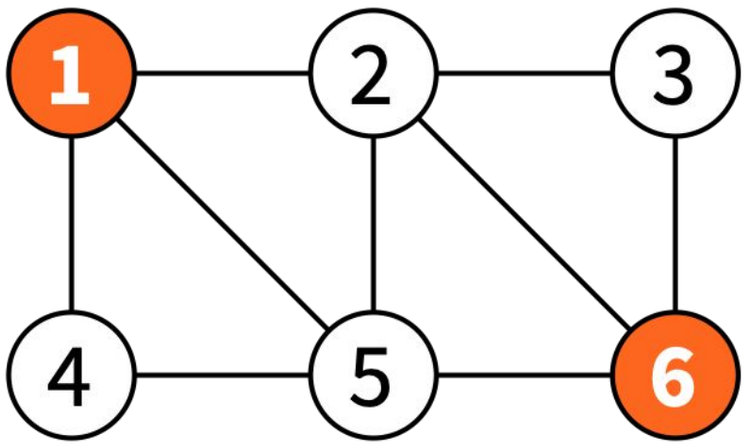
Isomorphism

Graphs are isomorphic if there exists an isomorphism from one to another



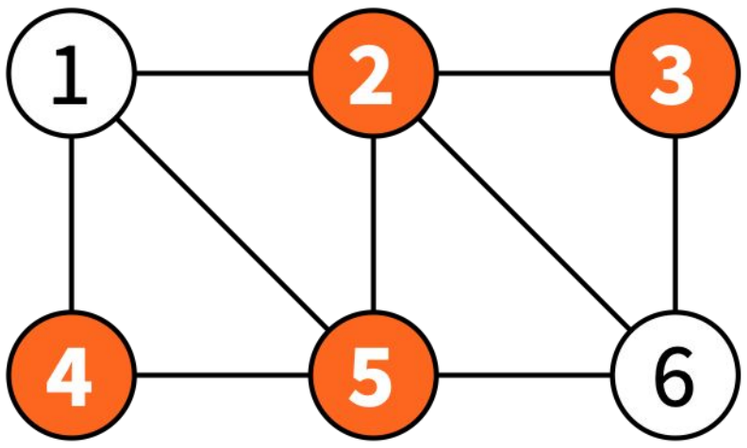
Graph problems

Independent set: non-adjacent nodes



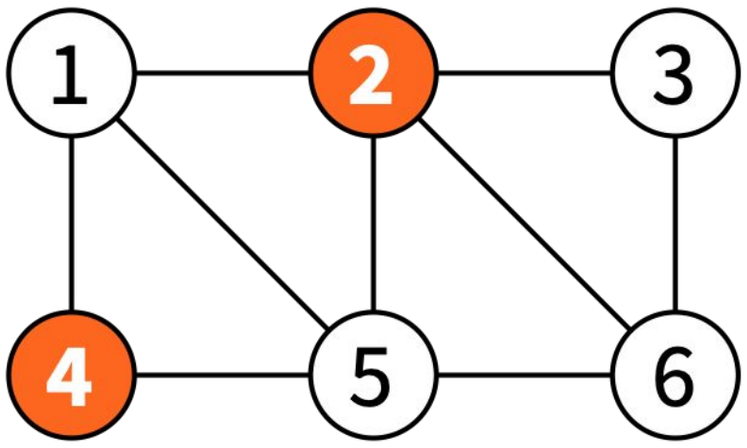
Graph problems

Vertex cover: at least one endpoint of each edge (all edges are "covered" with these nodes)



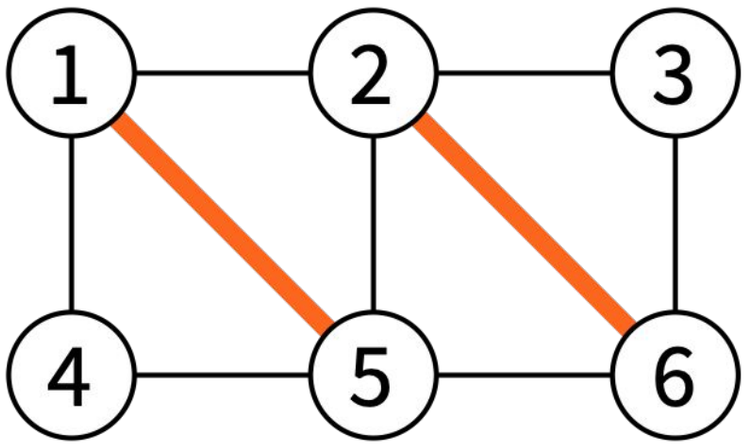
Graph problems

Dominating set: all other nodes have a neighbour in this set



Graph problems

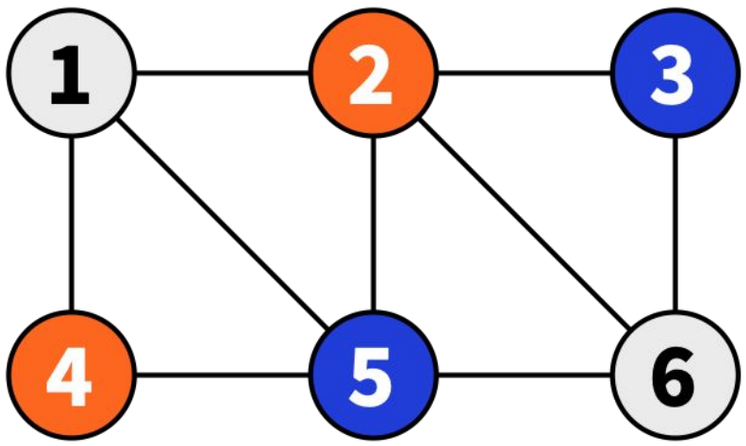
Matching: non-adjacent edges



Graph problems

Vertex coloring:

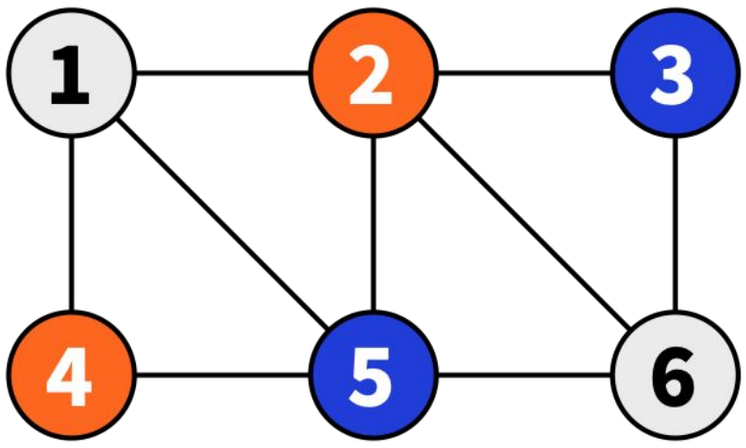
adjacent nodes have different colours



Graph problems

Vertex coloring:

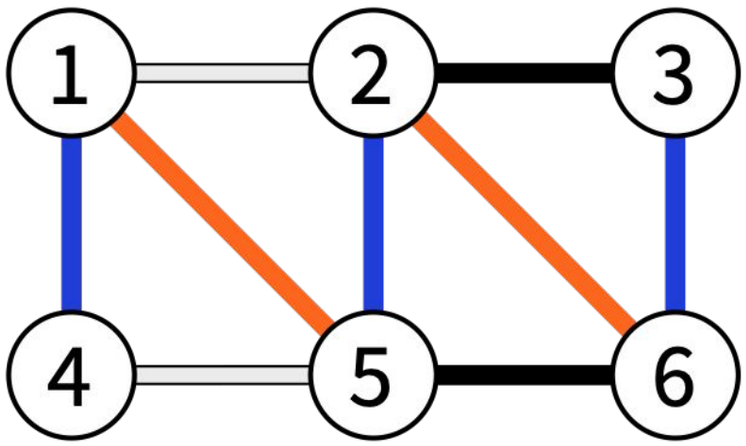
each colour class is an independent set



Graph problems

Edge coloring:

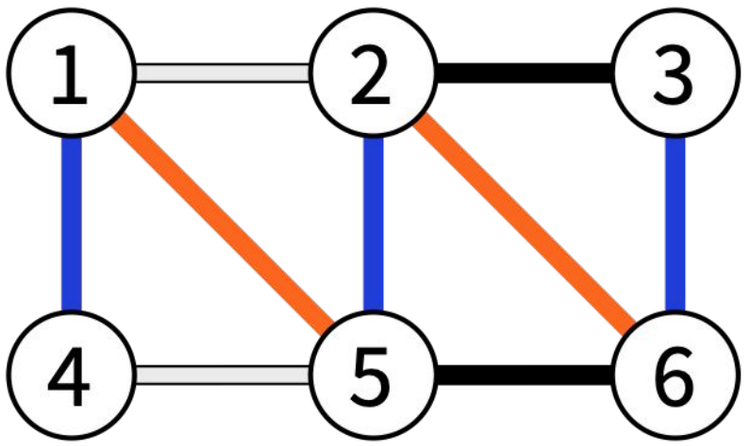
adjacent edges have different colours



Graph problems

Edge coloring:

each color class is a matching

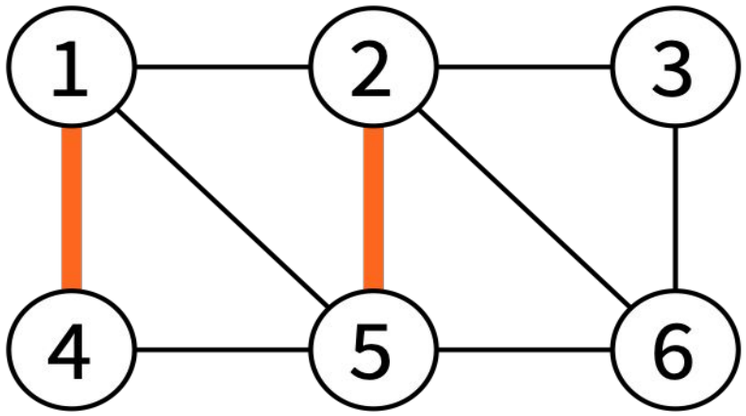


Maximisation problems

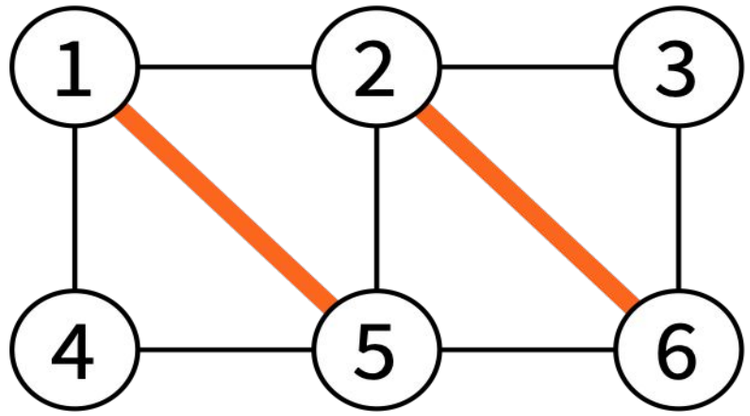
- maximal = **cannot add anything**
- maximum = **largest possible size**
- x-approximation = **at least $1/x$ times maximum**

Maximisation problems

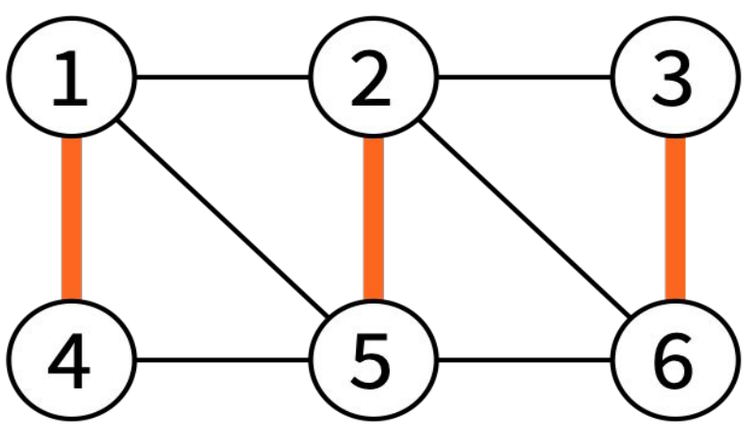
Matching



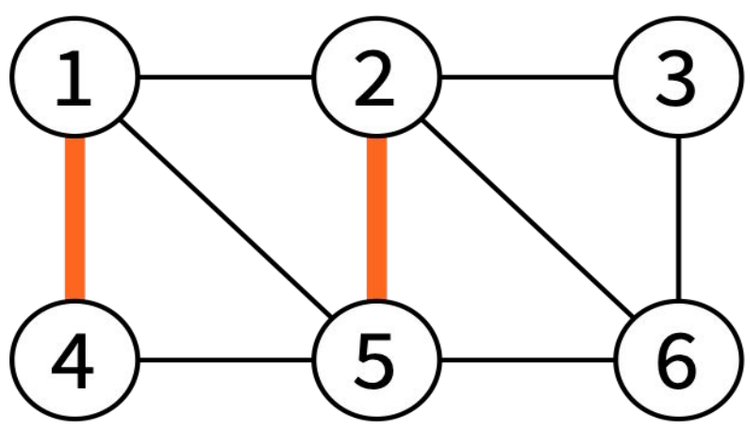
Maximal matching



Maximum matching

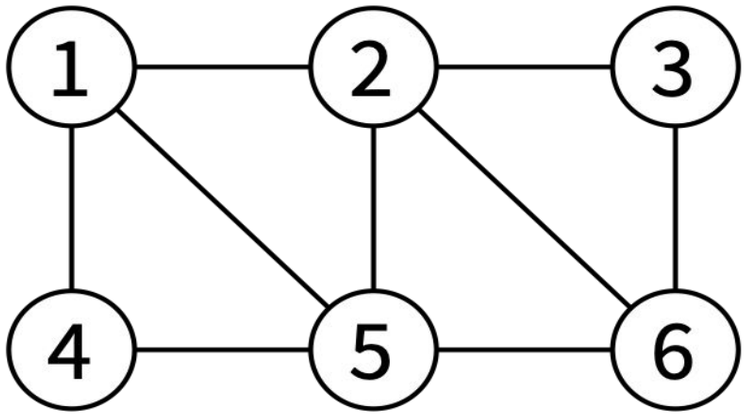


2-approximation

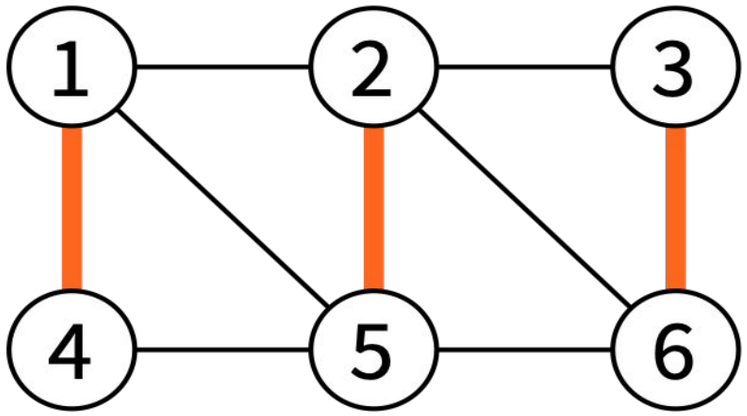


Maximisation problems

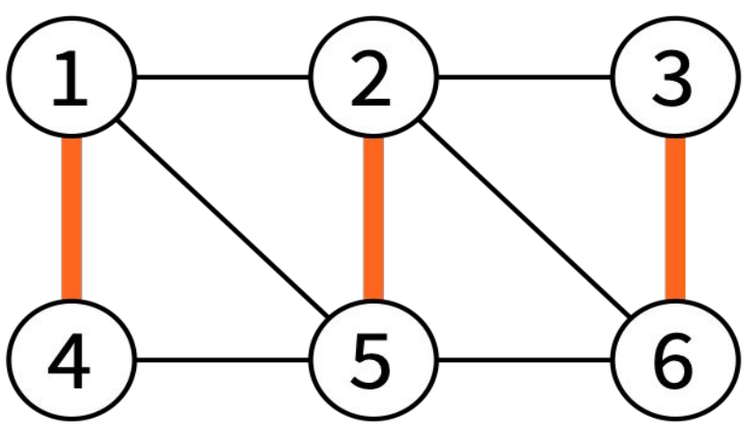
Matching



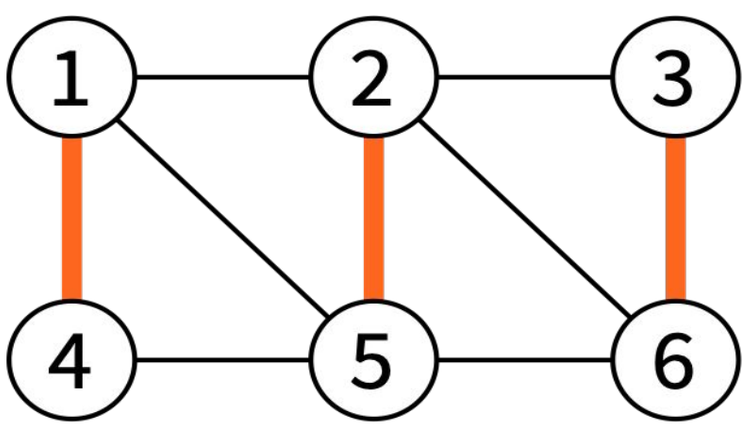
Maximal matching



Maximum matching



2-approximation

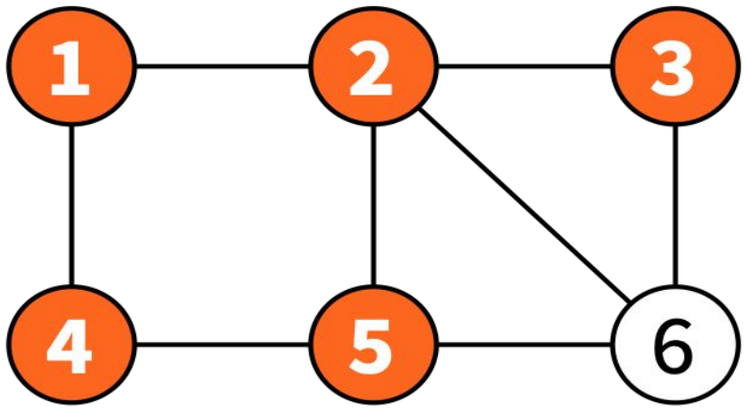


Minimisation problems

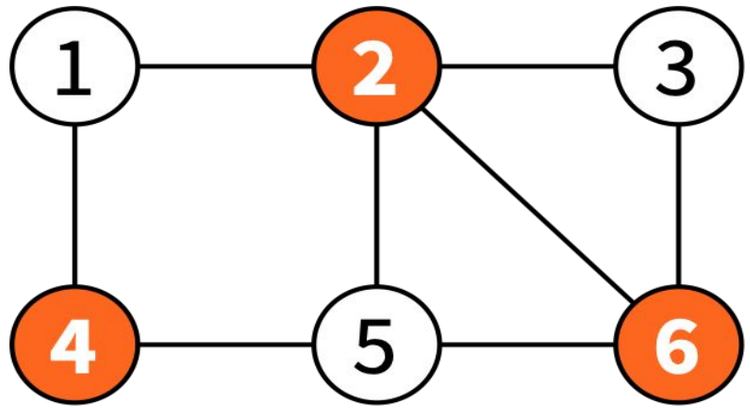
- minimal = **cannot remove anything**
- minimum = **smallest possible size**
- *x*-approximation =
  at most *x* times minimum

Minimisation problems

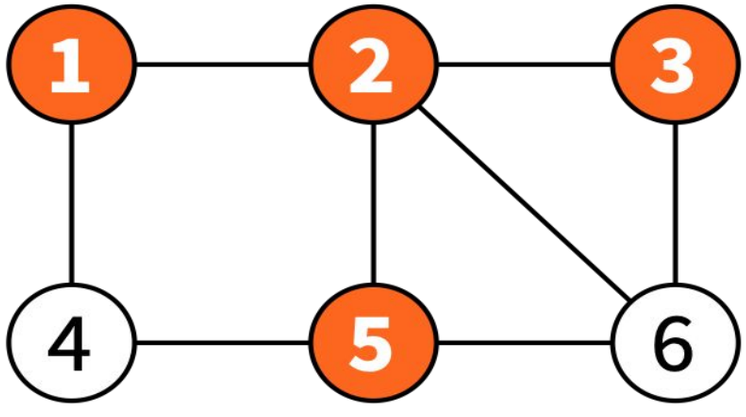
Vertex cover (VC)



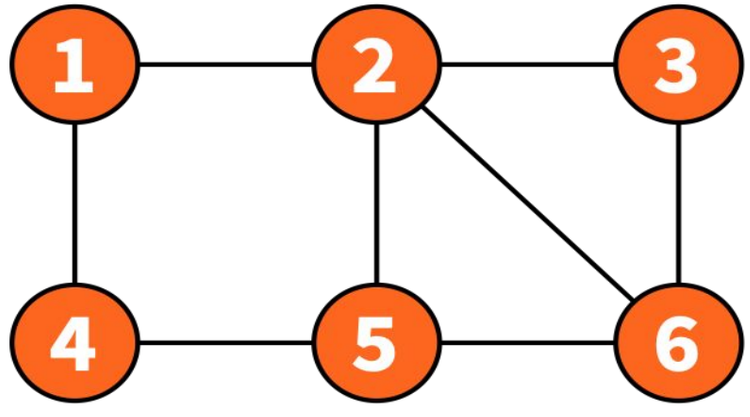
Minimum VC



Minimal VC



2-approximation



Approximation

- **Approximations are always feasible solutions!**
- **“2-approximation of minimum vertex cover”**
  - vertex cover
  - ≤ 2 times as large as minimum vertex cover

Optimization problem

An **optimization problem** has the following characteristics:

- A solution is a subset of one of the data in the problem;
- There are in general several admissible solutions;
- Each (admissible) solution has an associated value (usually a cost or a gain).

The **optimization problem** consists not only in finding an admissible solution, but also in finding a admissible solution, but to find a solution of minimal value or maximum value.

Example : giving change

- Problem : we have coins of values v1, v2, .., vn ;
- Solution : a sequence of (values of) coins with total S ;
- Best solution : the one using the least number of coins.

Greedy Algorithms

It is a particular strategy to solve an optimization problem.

A greedy algorithm is an algorithm that builds a solution :

- element by element without ever going back ;

- based on local considerations.

Example of greedy algorithm

Greedy algorithm

1. As long as S > 0 ;

2. Choose the piece with the largest value v less than S ;

3. Repeat with S - v.

Example of greedy algorithm

Greedy algorithm

1. As long as S > 0 ;

2. Choose the piece with the largest value v less than S ;

3. Repeat with S - v.

Correct example

S =16 with pieces of 10,5,2, 1:

- 16 - 10 = 6
- 6 - 5 = 1
- 1 - 1 = 0

3 pieces

Example of greedy algorithm

Greedy algorithm

1. As long as S > 0 ;

2. Choose the piece with the largest value v less than S ;

3. Repeat with S - v.

Correct example

S =16 with pieces of 10,5,2, 1:

- 16 - 10 = 6
- 6 - 5 = 1
- 1 - 1 = 0

3 pieces

Incorrect example

S=16 with pieces of 9,8,1:

- 16 - 9 = 7
- 7 - 1 = 6
- …

8 pieces (best case is 8 + 8 = 16)

Coloration of a graph with a greedy algorithm

Idea of the algorithm

We have a graph G = (V , E). For any v ∈ V :

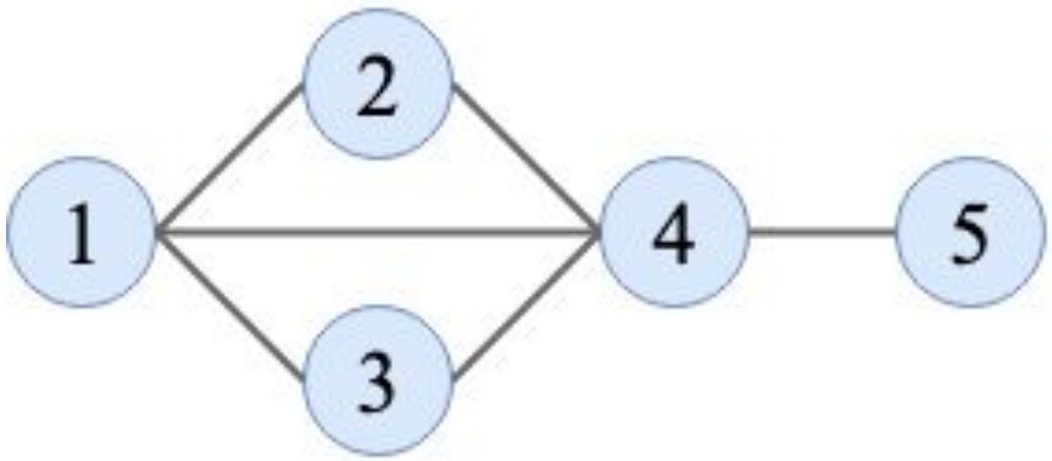
- We look at the set of colors already assigned to the neighbors of v.
- We deduce the smallest natural number that does not belong to this set.
- We assign this color to v.

Coloration of a graph with a greedy algorithm

Idea of the algorithm

We have a graph G = (V , E). For any v ∈ V :

- We look at the set of colors already assigned to the neighbors of v.
- We deduce the smallest natural number that does not belong to this set.
- We assign this color to v.

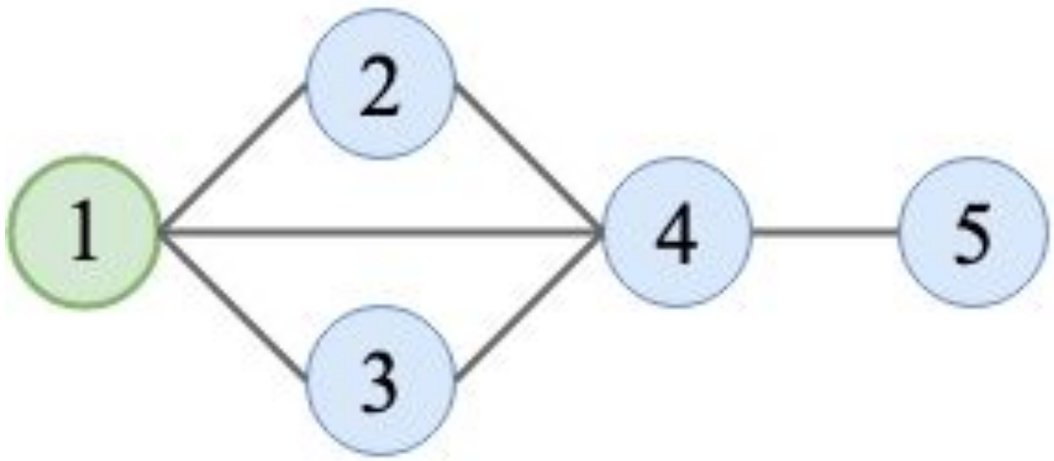


Coloration of a graph with a greedy algorithm

Idea of the algorithm

We have a graph G = (V , E). For any v ∈ V :

- We look at the set of colors already assigned to the neighbors of v.
- We deduce the smallest natural number that does not belong to this set.
- We assign this color to v.

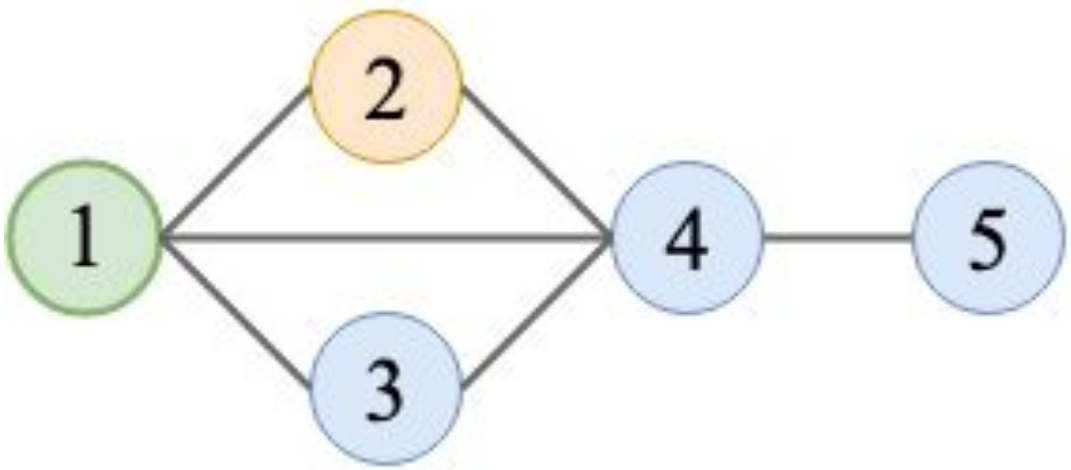


Coloration of a graph with a greedy algorithm

Idea of the algorithm

We have a graph G = (V , E). For any v ∈ V :

- We look at the set of colors already assigned to the neighbors of v.
- We deduce the smallest natural number that does not belong to this set.
- We assign this color to v.

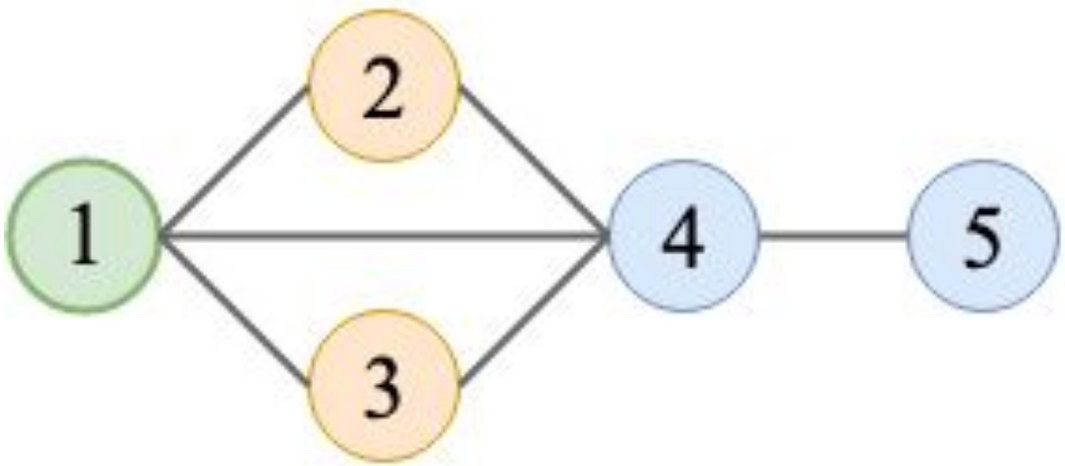


Coloration of a graph with a greedy algorithm

Idea of the algorithm

We have a graph G = (V , E). For any v ∈ V :

- We look at the set of colors already assigned to the neighbors of v.
- We deduce the smallest natural number that does not belong to this set.
- We assign this color to v.

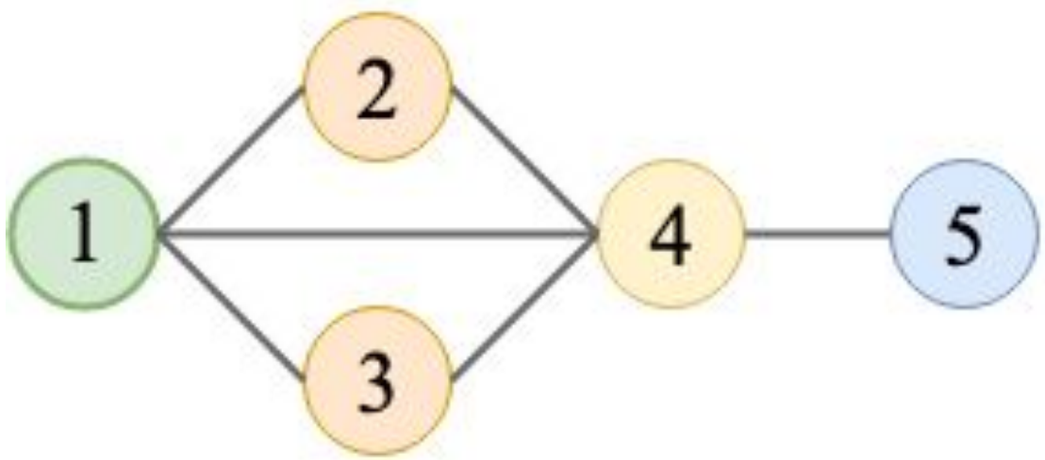


Coloration of a graph with a greedy algorithm

Idea of the algorithm

We have a graph G = (V , E). For any v ∈ V :

- We look at the set of colors already assigned to the neighbors of v.
- We deduce the smallest natural number that does not belong to this set.
- We assign this color to v.

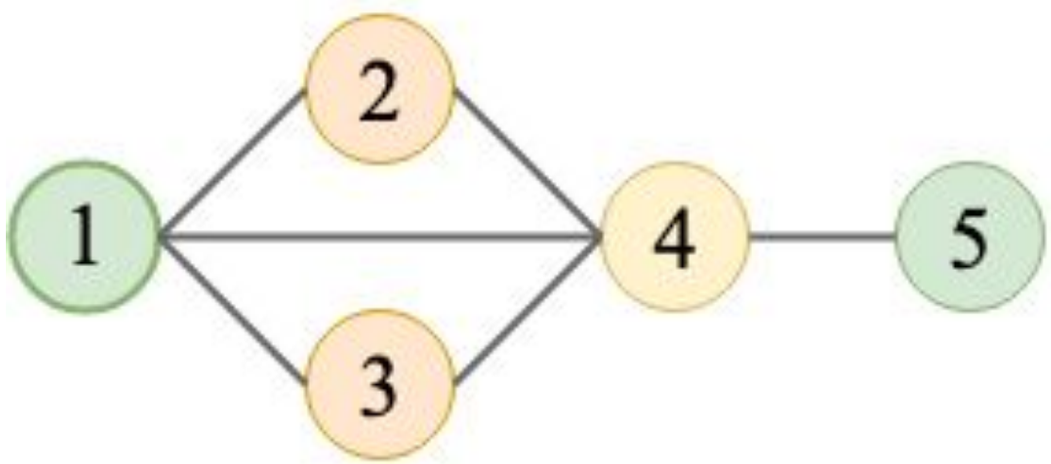


Coloration of a graph with a greedy algorithm

Idea of the algorithm

We have a graph G = (V , E). For any v ∈ V :

- We look at the set of colors already assigned to the neighbors of v.
- We deduce the smallest natural number that does not belong to this set.
- We assign this color to v.



Reminders about tree

a tree is a connected graph without cycle.
Be careful, the notion is different from the trees usually used in algo:
- No arity constraint ;
- No notion of root

Properties

- Between two given vertices of a tree, there is always exactly an (elementary) chain;
- A tree with n vertices has n - 1 edges.

Spanning Tree

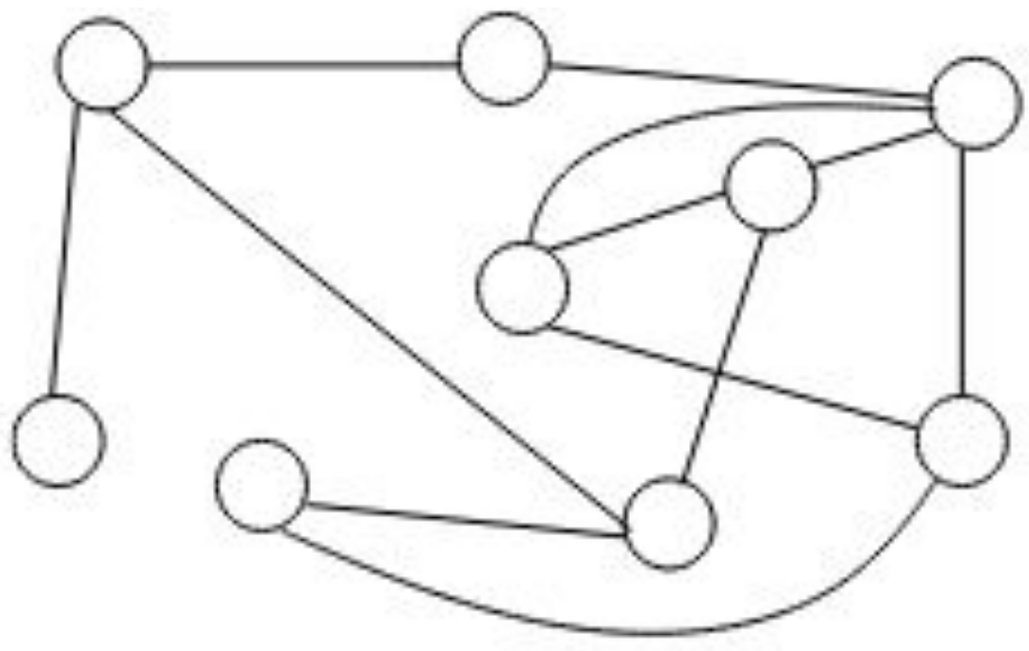
A subgraph of G is said to be covering if it contains all vertices of G.

Be careful, a covering subgraph is not necessarily connected.

Spanning Tree

A subgraph of G is said to be covering if it contains all vertices of G.

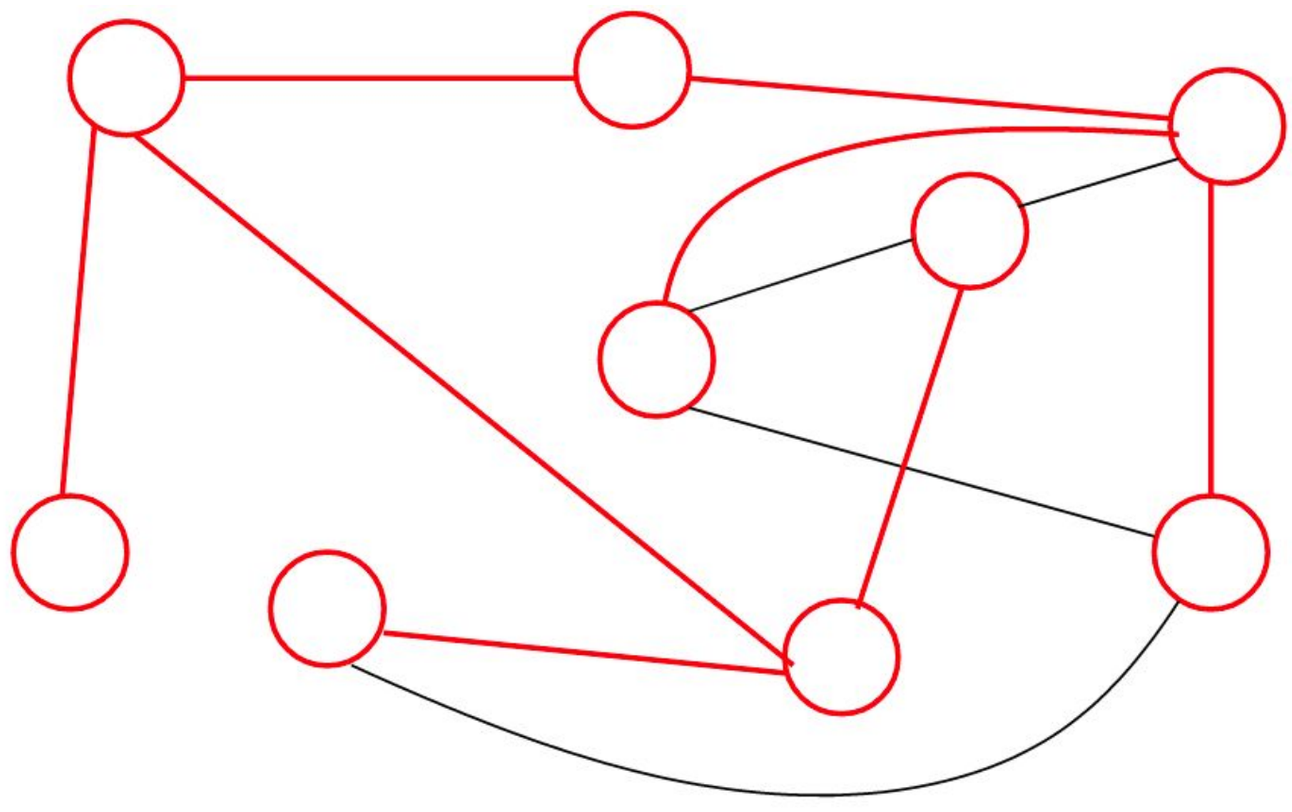
Be careful, a covering subgraph is not necessarily connected.



Spanning Tree

A subgraph of G is said to be covering if it contains all vertices of G.

Be careful, a covering subgraph is not necessarily connected.



Spanning Tree

If A is a **spanning subgraph** of a graph G with V vertices the following characterizations are equivalent:

- A is a spanning tree of G ;

- A is cycle-free and has V - 1 edges;

- A is connected

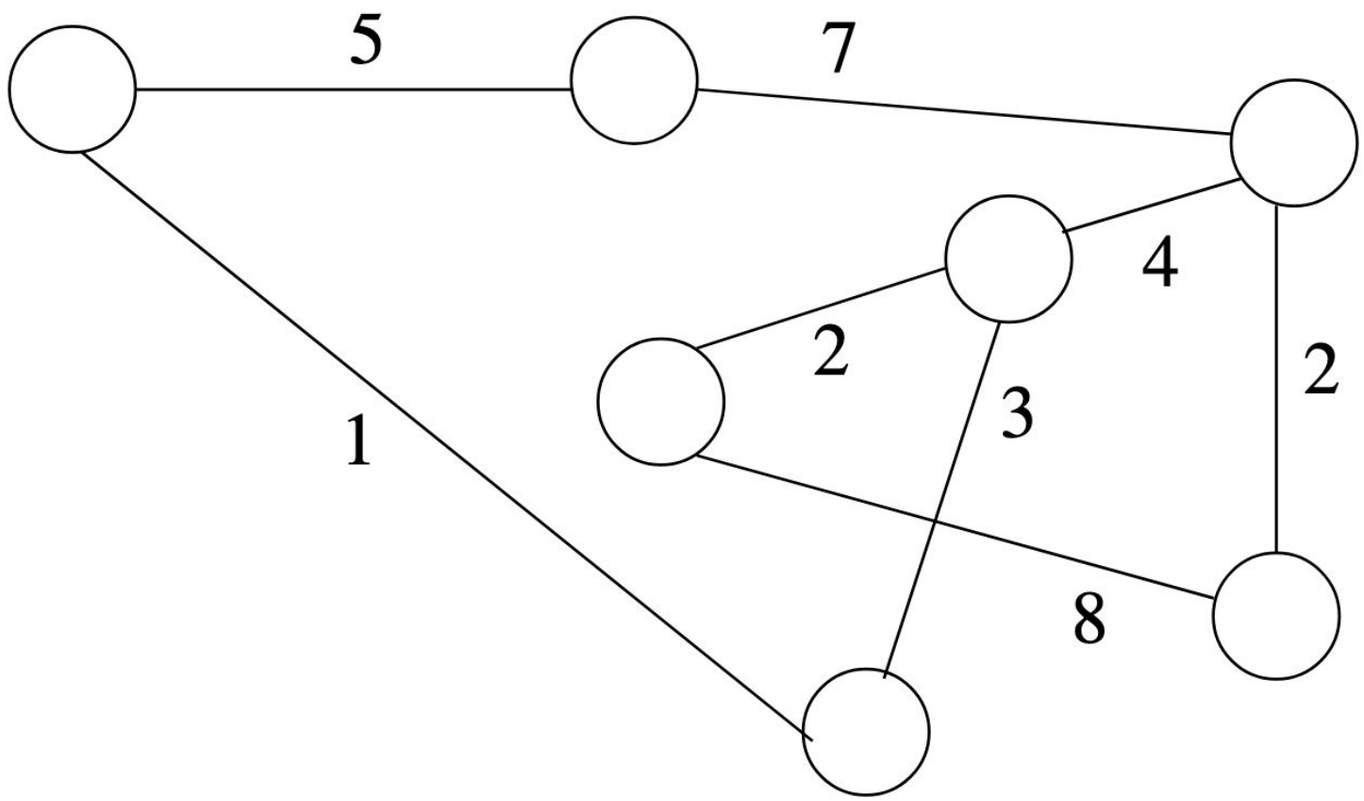
- We cannot add an edge to A without creating a cycle;

- We cannot remove an edge from A without breaking its connectedness

Spanning Tree in a weighted graph

Minimum weight spanning tree

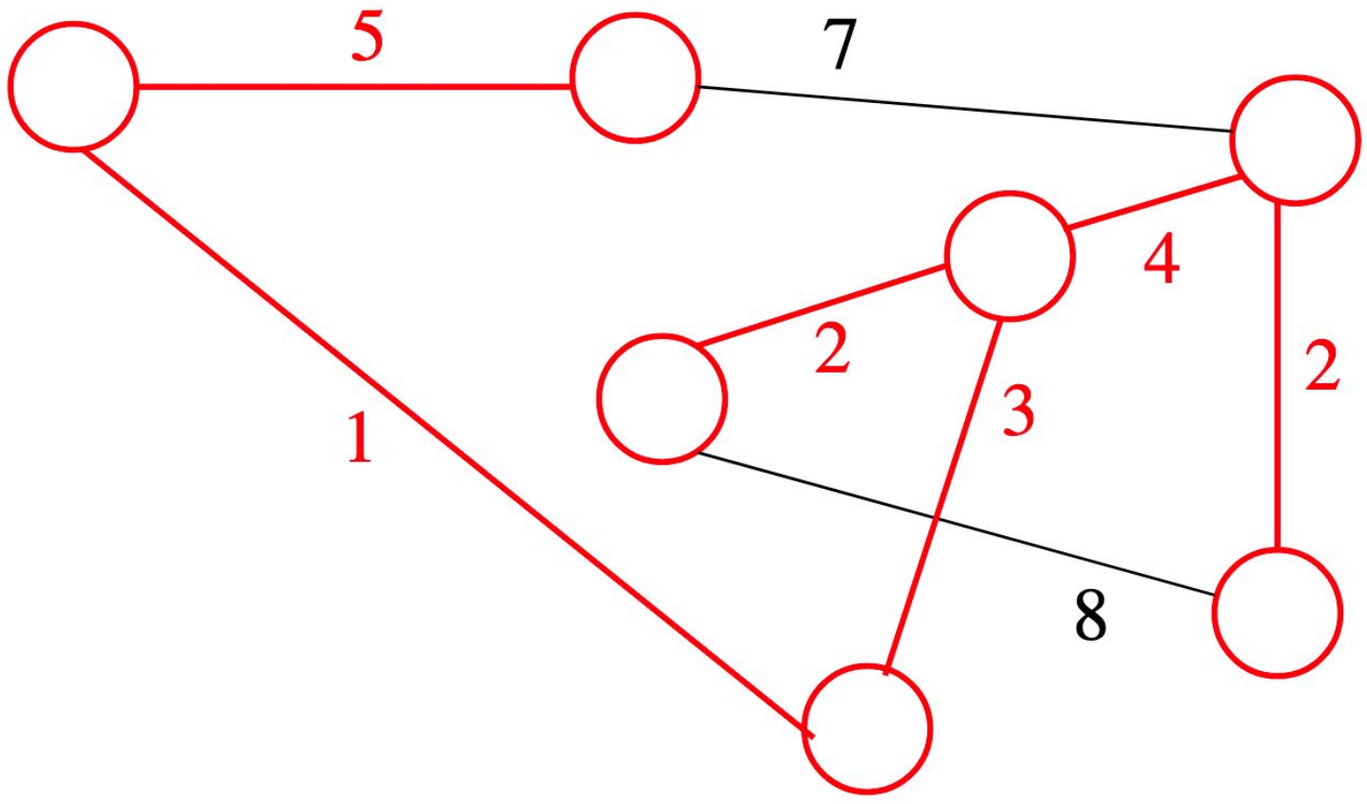
The weight of a subgraph is the sum of the weights of the edges. We are looking for a minimal weight spanning tree (MST).



Spanning Tree in a weighted graph

Minimum weight spanning tree

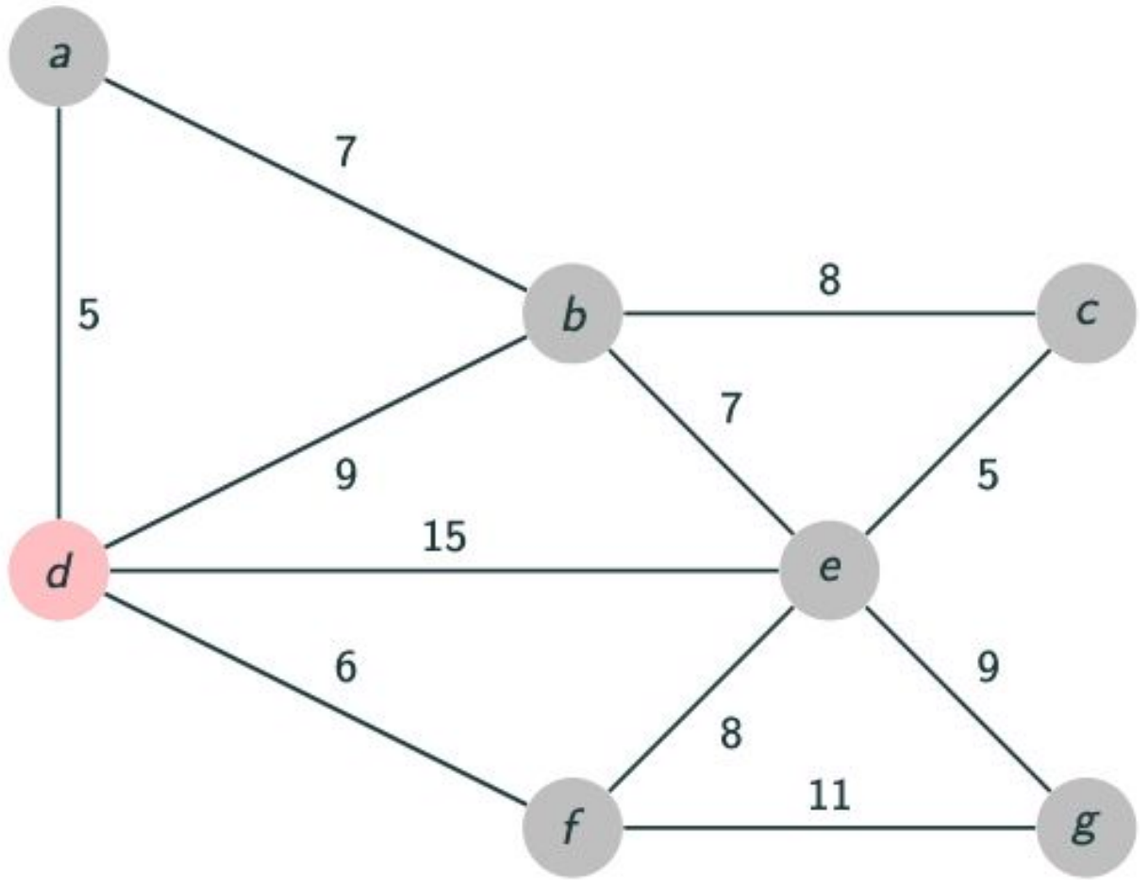
The weight of a subgraph is the sum of the weights of the edges. We are looking for a minimal weight spanning tree (MST).



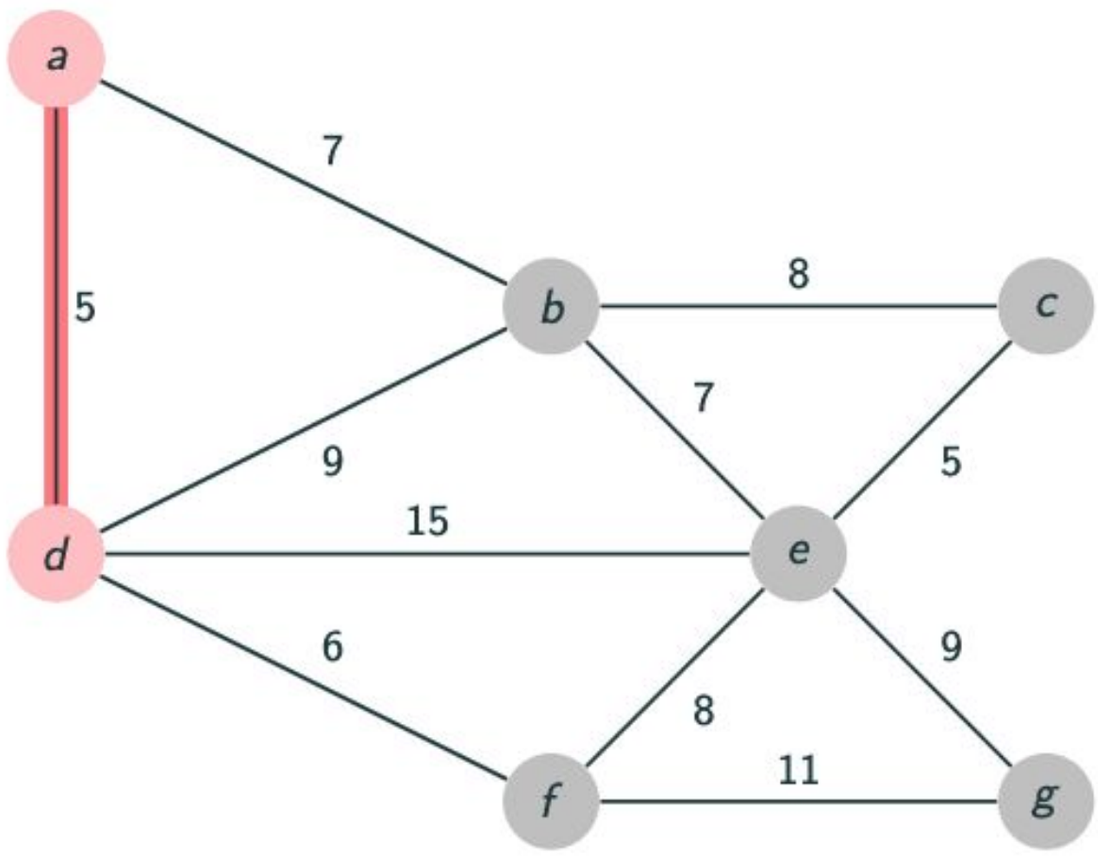
Prim's Algorithm

```
A := (∅, ∅)
ajouterSommet( choisirSommet(G), A )
while nbSommets(A) < V
    // Recherche de l'arête min incidente à A
    m := +∞
    foreach x ∈ A do
        foreach y ∈ ensVoisins(x) do
            if y ∉ ensSommets(A) et poids(x, y) < m
                m := poids(x, y)
                ymin := y
    ajouterSommet( ymin, A )
    ajouterArete( (x, ymin), A )
return A
```

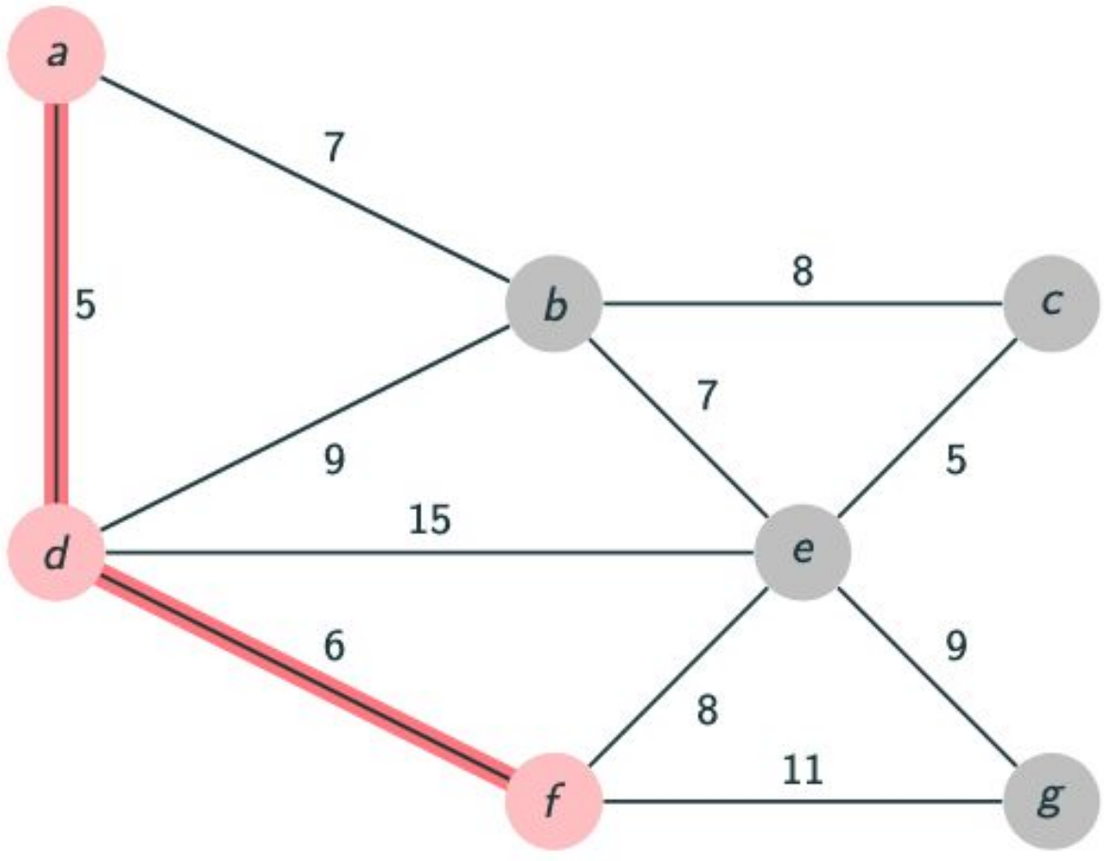
Prim's Algorithm on an example



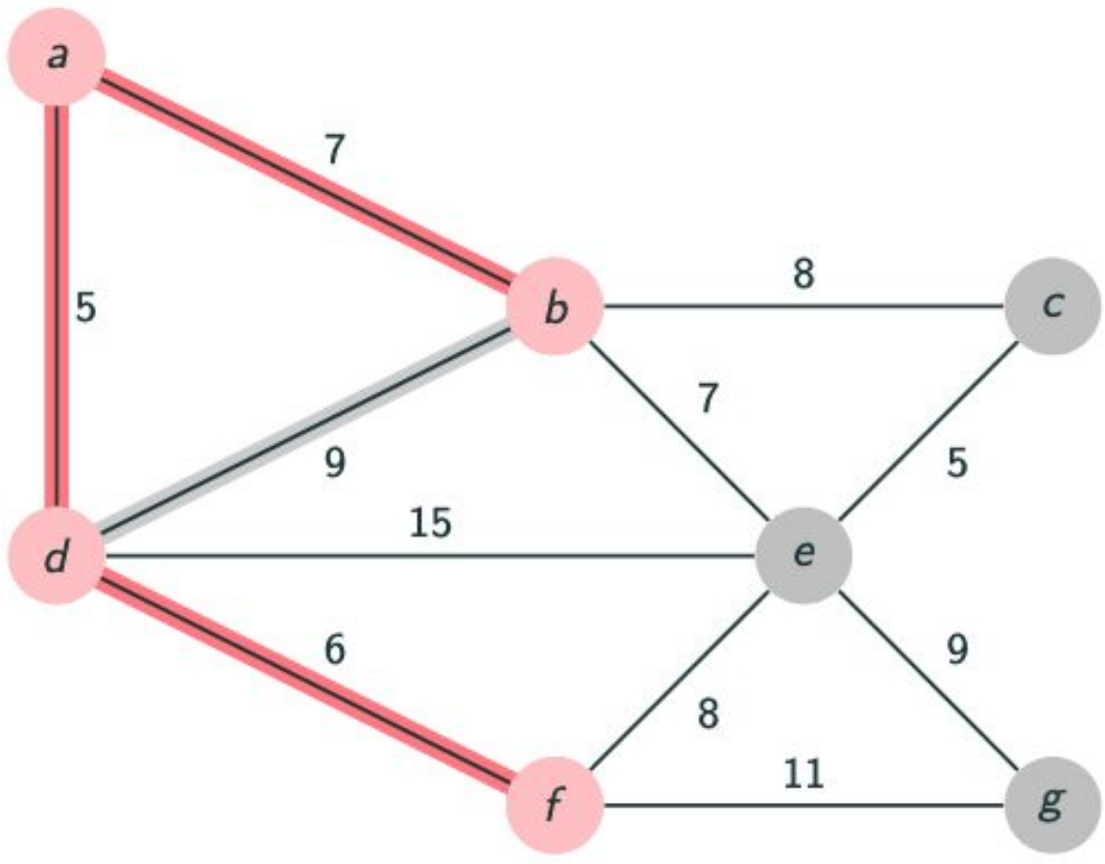
Prim's Algorithm on an example



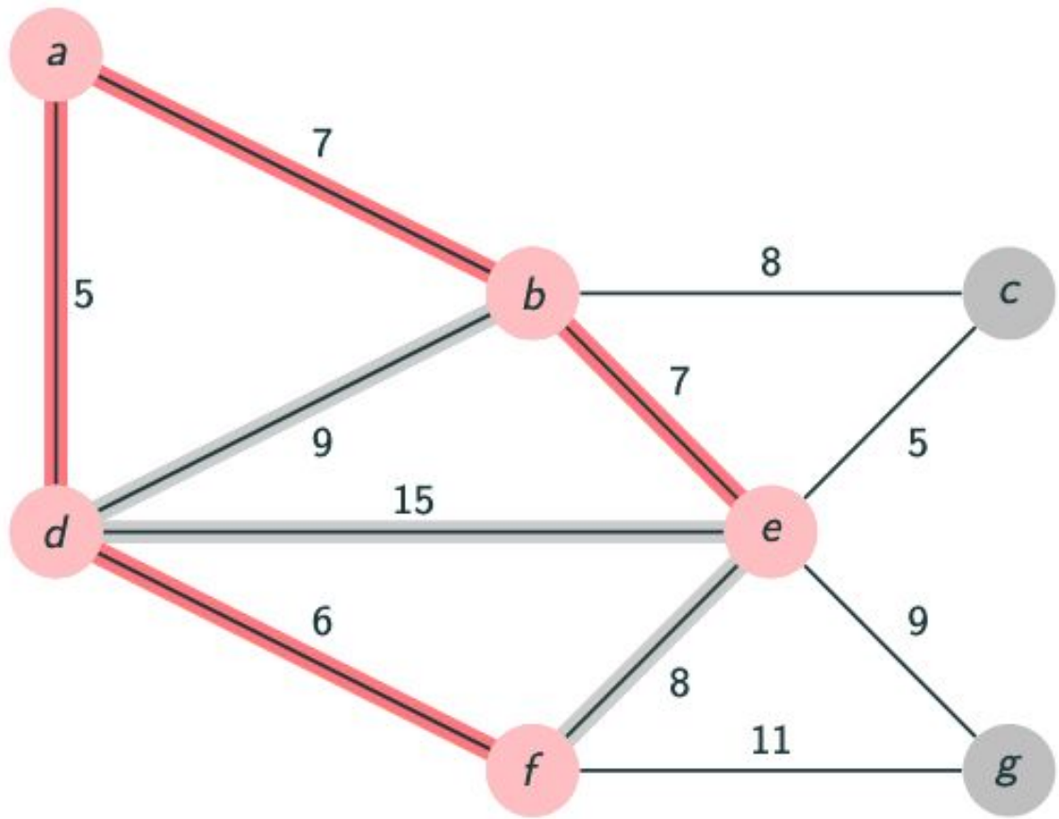
Prim's Algorithm on an example



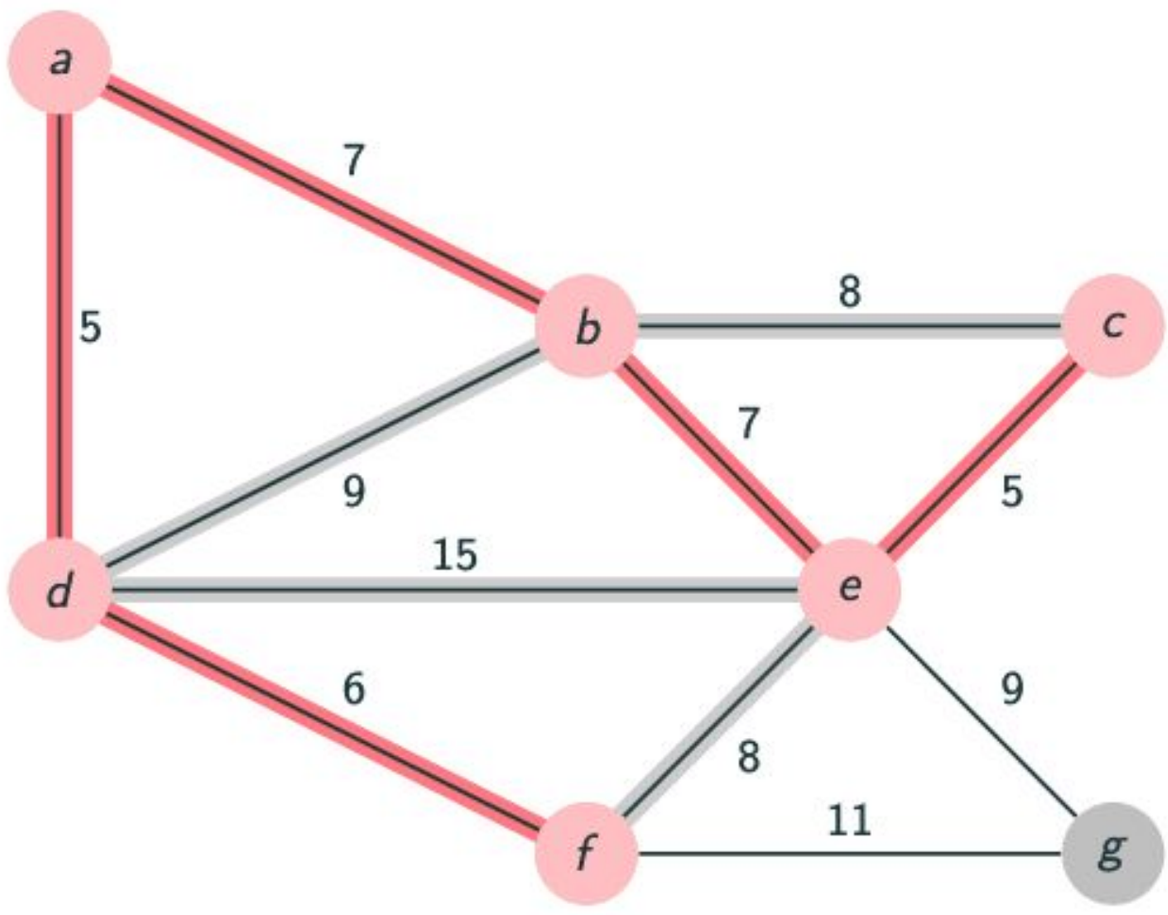
Prim's Algorithm on an example



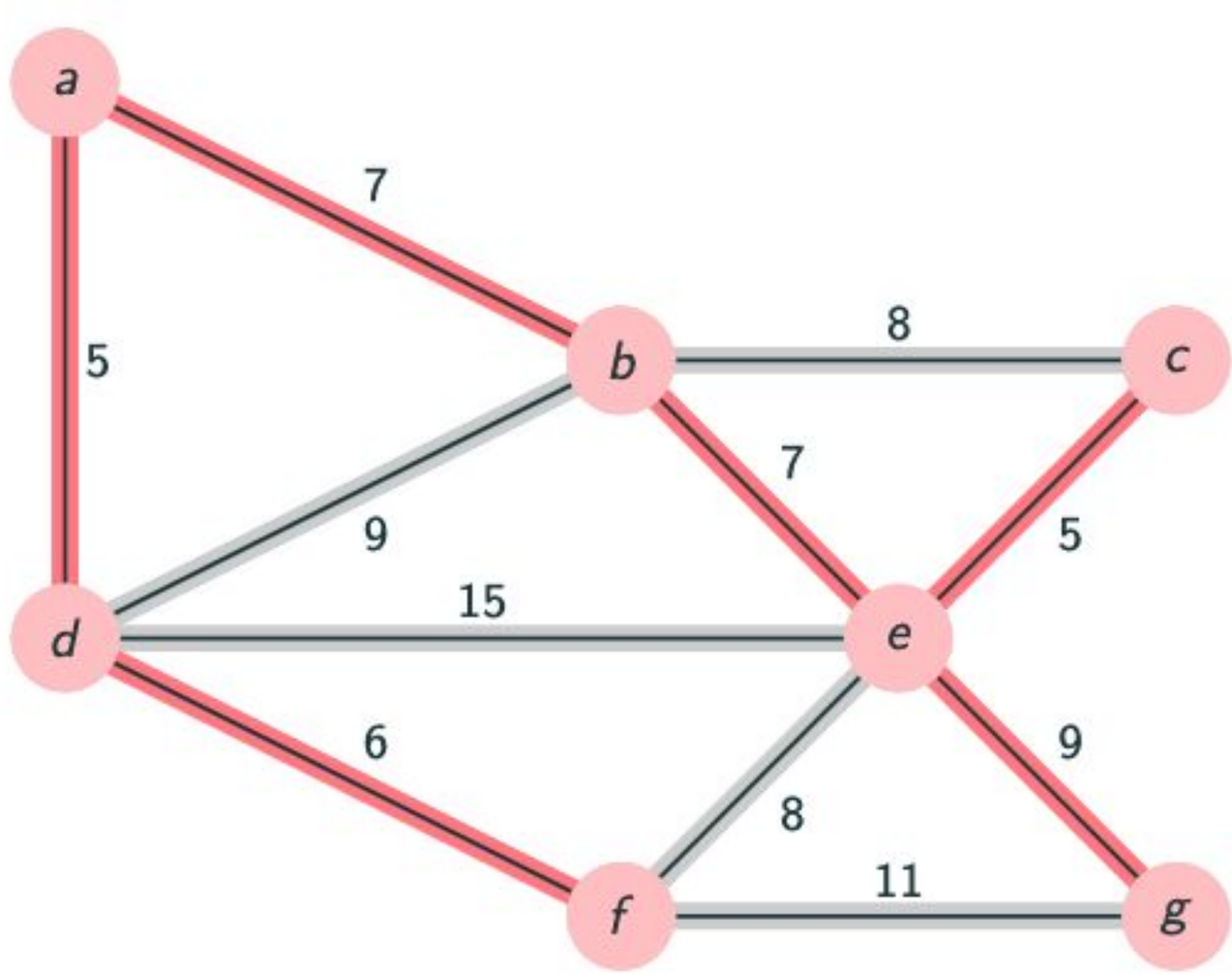
Prim's Algorithm on an example



Prim's Algorithm on an example



Prim's Algorithm on an example



Is it a greedy algorithm ?

We find the characteristics of a greedy algorithm:

- The constructed solution is a set (of edges) ;
- The admissible solutions are covering trees;
- We are looking for a solution of minimal total weight;
- Prim's algorithm proceeds only by adding edges (and vertices) in A;
- he choice of the next edge is based only on its weight.
-

The algorithm therefore has a cost in $O(V \times C)$ where C is the cost of choosing the next edge;

Graph theory and distributed algorithms

- **Network ≈ graph:** node ≈ computer, edge ≈ link
- **Graph theory used to:**
  - define: model of computing,
    what we want to solve, what we assume …
  - prove: correctness of algorithms,
    time complexity, impossibility results …

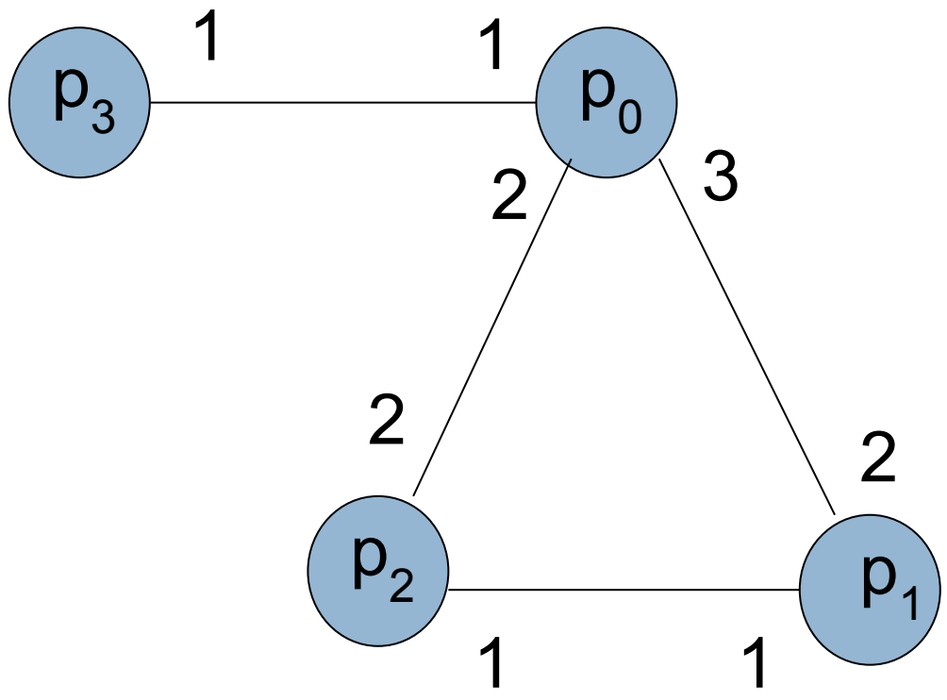
Distributed Algorithm : Message Passing Model

Message-Passing Model

- In a message-passing system, processors communicate by sending messages over communication channels, where each channel provides a bidirectional connection between two specific processors.
- The pattern of connections provided by the channels describes the **topology** of the system.
- The topology is represented by an undirected graph in which
  - each node represents a processor and
  - an edge exists between two nodes if and only if there is a communication channel between the corresponding processors.

Message-Passing Model

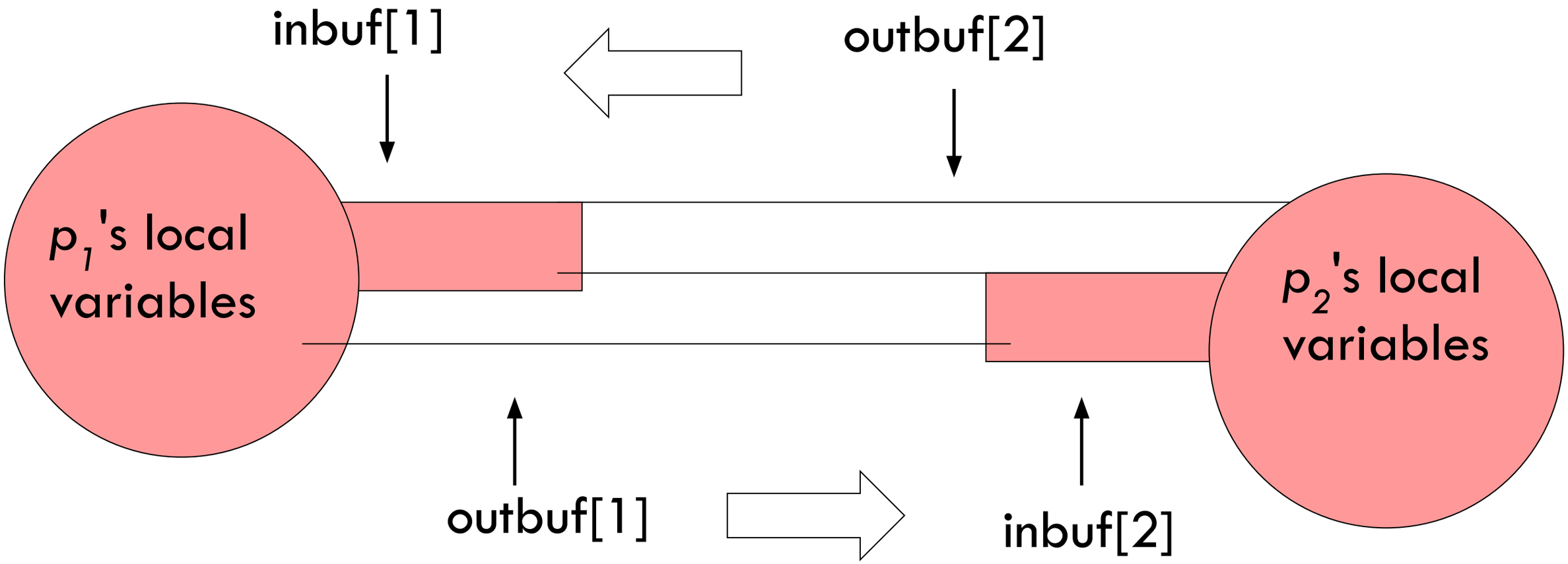
- processors are p_0, p_1, …, p_{n-1} (nodes of graph)
- each processor p_i labels its incident channels with the integers 1, 2, …r where r is the degree of p_i
  - 1, 2, 3 for p_0 in the exemple



Modeling Processors and Channels

- Processor is a state machine including
  - local state of the processor
  - mechanisms for modeling channels
- Channel directed from processor p_i to processor p_j is modeled in two pieces:
  - *outbuf* variable of p_i : msgs p_i has sent to p_j *and not yet delivered to* p_j
  - *inbuf* variable of p_j : msgs *delivered by* p_i *and not yet processed by* p_j
- *Outbuf* corresponds to physical channel, *inbuf* to incoming message queue

Modeling Processors and Channels



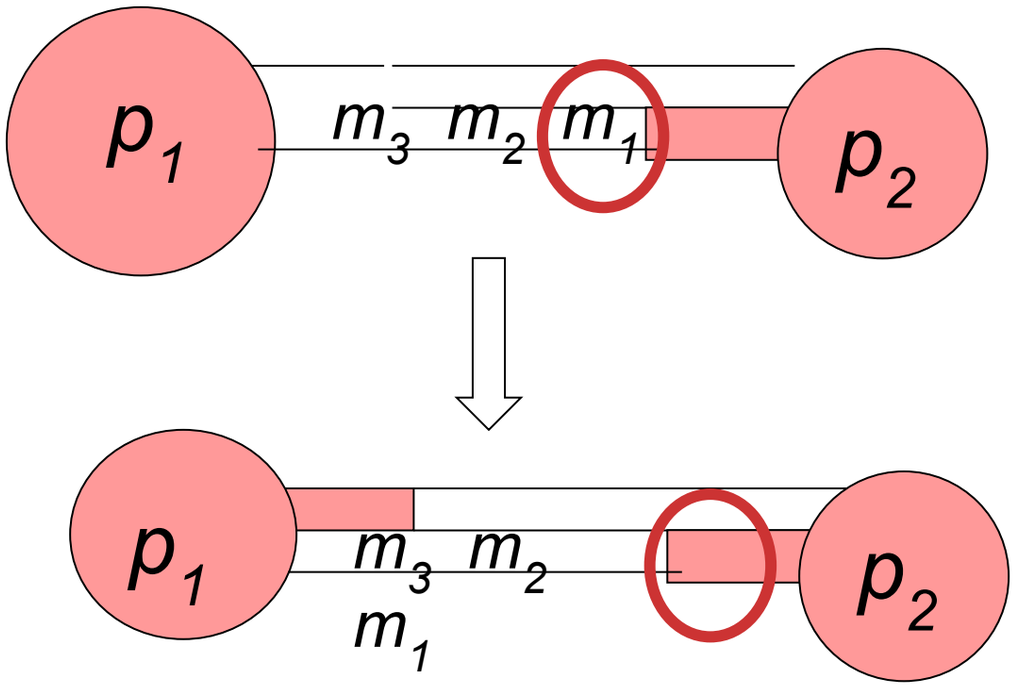
Pink area (**local vars + inbuf**) is ***accessible* state for a processor.**

Processor state and configuration

- The *state* of a processor p_i contains 2r components where r is the number of neighbors of p_i . In particular for every neighbor j
  - *outbuf[j]* : is the set of msgs p_i has sent to p_j *and not yet delivered to* p_j
  - *inbuf[j]* variable of p_i : msgs from p_j *delivered by* p_i *and not yet processed by* p_i
- A *configuration* is a vector $C=(q_0,q_1,\ldots,q_{n-1})$ where q_i is a state of process p_i
  - An initial configuration is a vector $(q_0,q_1,\ldots,q_{n-1})$ such that q_i is the initial state of process p_i

A Deliver Event

- Moves a message from sender's outbuf to receiver's inbuf; message will be available next time receiver takes a step



A Computation Event

- Occurs at one processor and represents a computation step where the process applies its transition function :
  - Start with old accessible state (local vars + incoming messages)
  - Apply transition function of processor's state machine; handles all incoming messages
  - End with new accessible state with empty inbufs, and new outgoing messages

Execution

- The behavior of the system over time is modeled as an execution : a sequence of configurations alternating with events :
  - config, event, config, event, config, …
- in first config: each processor is in initial state and all inbufs are empty
- for each consecutive (config, event, config), new config is same as old config except:
  - if delivery event: specified msg is transferred from sender's outbuf to receiver's inbuf
  - if computation event: specified processor's state (including outbufs) change according to transition function

Asynchronous Systems

- A system is said to be asynchronous if there is no fixed upper bound on how it takes for a message to be delivered or how much time elapses between two consecutive steps of a processor (ex. Internet)
- Observation : there are usually upper bounds on message delays and processors step times, but sometimes they are very large, infrequently reached and they can vary over time :

  ☞ It is often desirable to write an algorithm independent of any timing parameters

Asynchronous Executions

- An execution is *admissible for the asynchronous model* if:
  - every message in an outbuf is eventually delivered
  - every processor takes an infinite number of steps
- No constraints on when these events take place: arbitrary message delays and relative processor speeds are not ruled out
- Models reliable system (no message is lost and no processor stops working)

Synchronous Message Passing Systems

- An execution is *admissible* for the synchronous model if it is an infinite sequence of "rounds"
- A"round" is a sequence of deliver events that move all messages in transit into inbuf's, followed by a sequence of computation events, one for each processor

Synchronous Message Passing Systems

- The new definition of admissible captures lockstep unison feature of synchronous model.
- This definition also implies
  - every message sent is delivered
  - every processor takes an infinite number of steps.
- **Time** is measured as number of rounds until termination.

Message Complexity Measure

- **Message complexity:** maximum number of messages sent in any admissible execution

- This is a worst-case measure

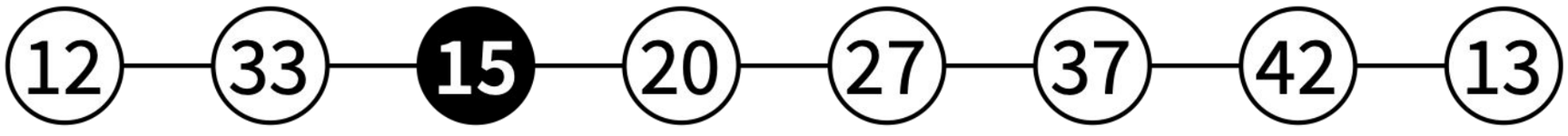
Distributed Algorithm : Locality

Locality

- **Output of a node can only depend on what it knows**
- **After *T* time steps, a node can only know about things up to distance *T***

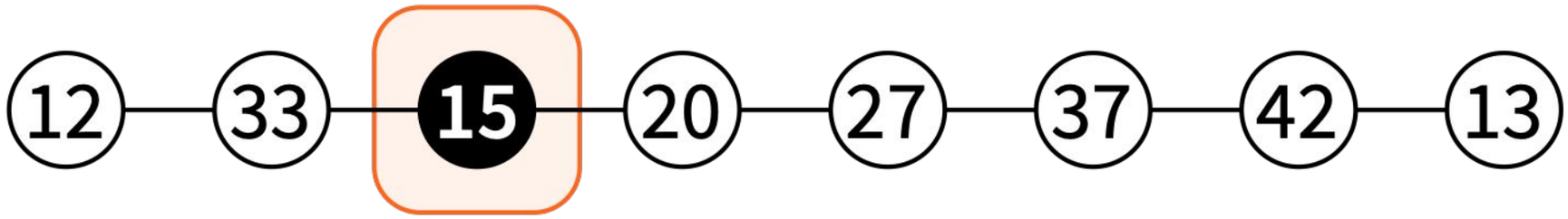
Locality

- **Who knows that node 15 exists?**
  - initially, only node 15
  - everyone else has to learn it by exchanging messages



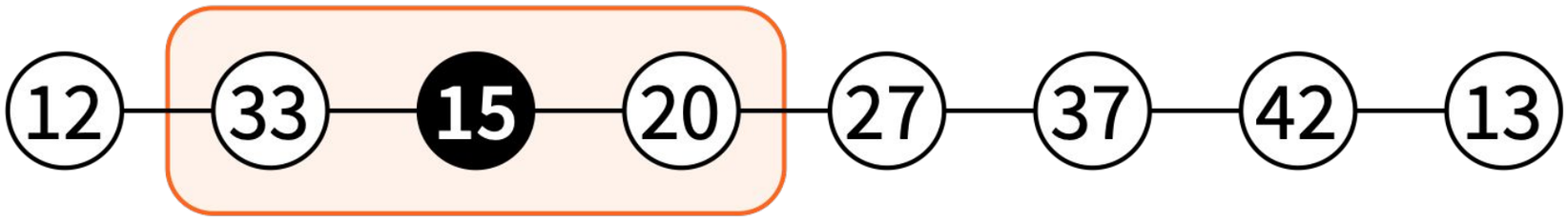
Locality

- **Who knows about node 15 at time $T = 0$?**
  - initial state, before we exchange any messages



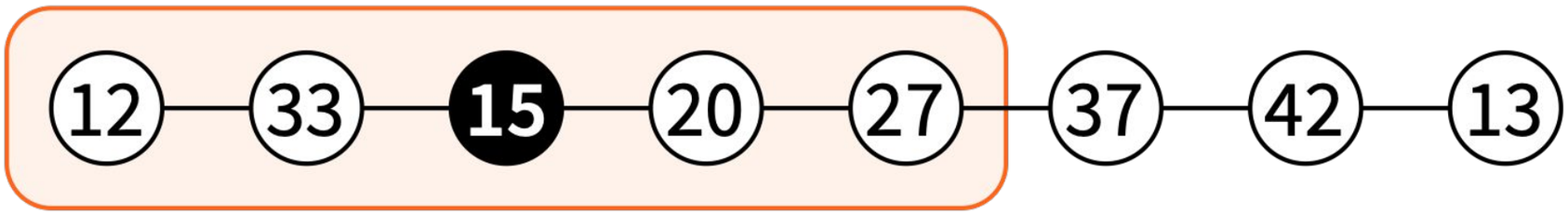
Locality

- **Who knows about node 15 at time $T = 1$?**
  - after 1 communication round



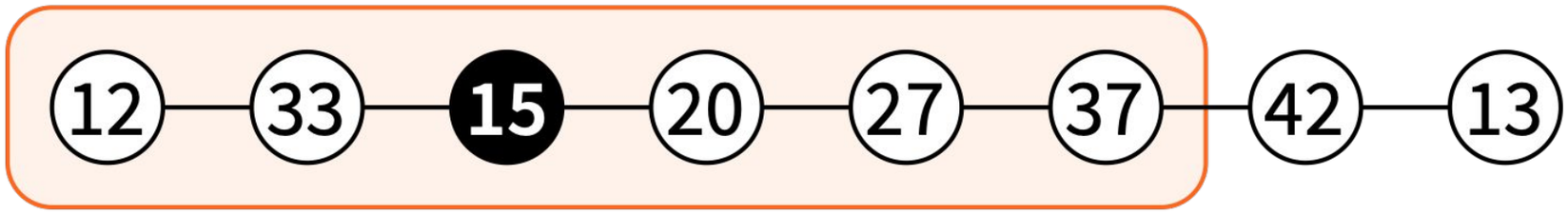
Locality

- **Who knows about node 15 at time $T = 2$?**
  - after 2 communication rounds



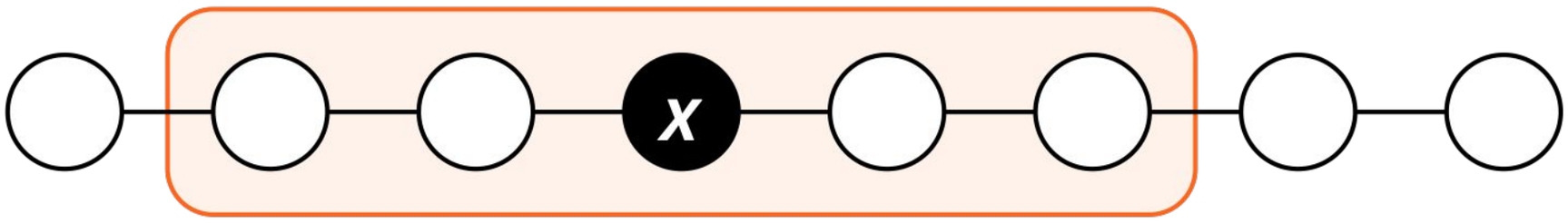
Locality

- **Who knows about node 15 at time $T = 3$?**
  - after 3 communication rounds



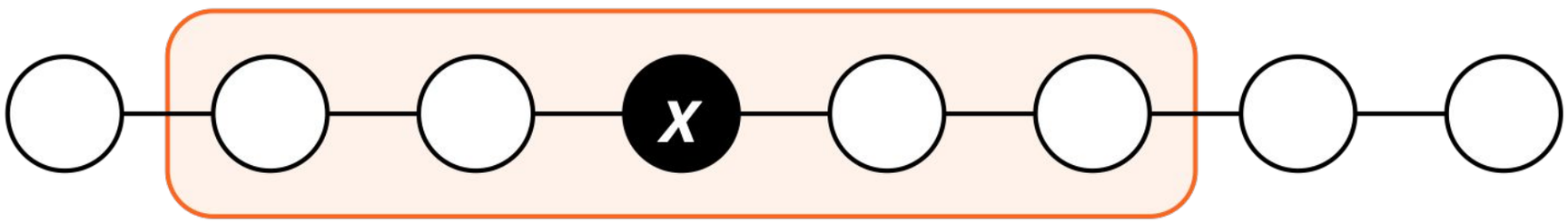
Locality

- **After *T* communication rounds, only nodes up to distance *T* from node *x* can know anything about node *x***
  - distance = “number of hops”



Locality

- **After** *T* communication rounds**,**
  node *x* **can only know about other nodes that are within** distance *T* **from it**
  - distance = “number of hops”



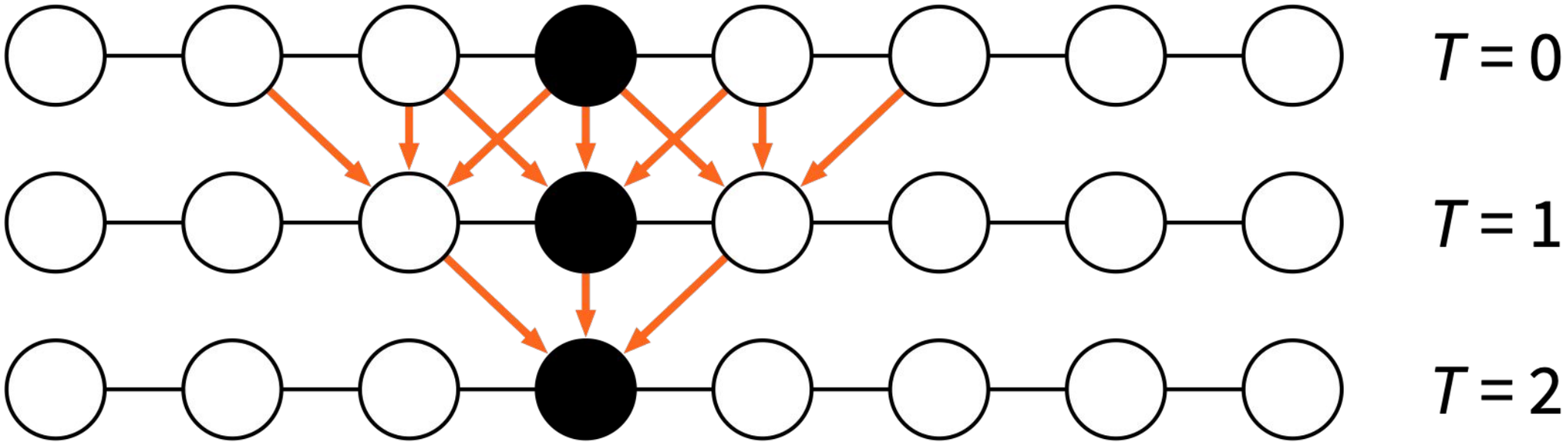
Locality

- **My state at time T only depends on:**
  - my state at time $T - 1$, and
  - messages that I received on round T,

  which only depend on:
    - the state of my neighbours at time $T - 1$

Locality

- **State at time *T* only depends on initial information within distance *T***

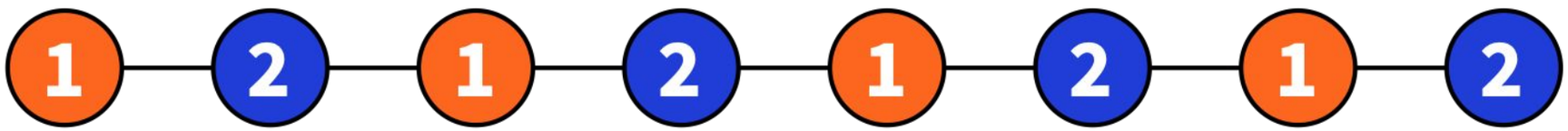


Locality

- **Time = distance**
- **Fast algorithm = “local” algorithm**
  - outputs only depend on local neighbourhoods

Using locality to prove lower bounds

- **Example: 2-colouring of a path**
- **Upper bound: possible in time $O(n)$**
- **Lower bound: not possible in time $o(n)$**



Algorithm for 2-coloring

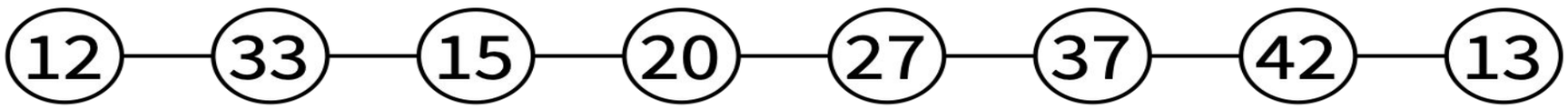
- **Assumption: path, unique identifiers**

Algorithm for 2-coloring

- **Assumption: path, unique identifiers**
- **Two phases:**
  - find the endpoint with smaller identifier
  - starting from this end, assign colours 1, 2, 1, 2, …

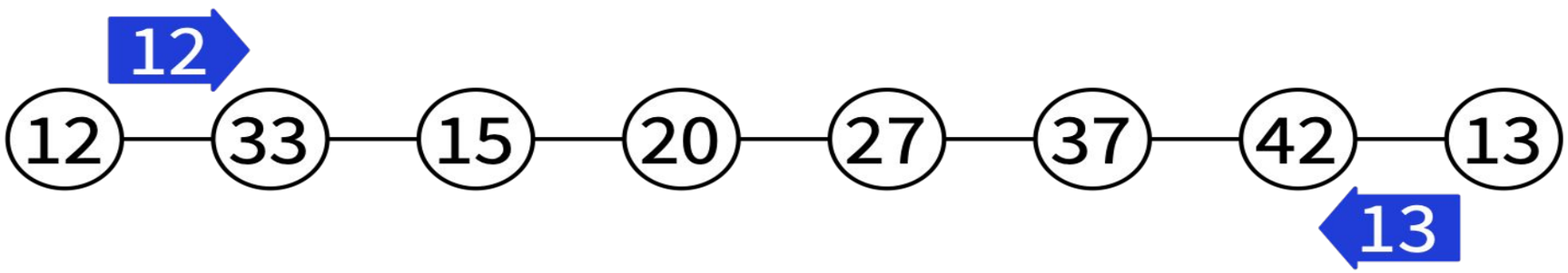
Algorithm for 2-coloring

- **Messages:**
  - “**ID *x***” = there is an endpoint with identifier *x*
  - “**colour *c***” = my colour is *c*



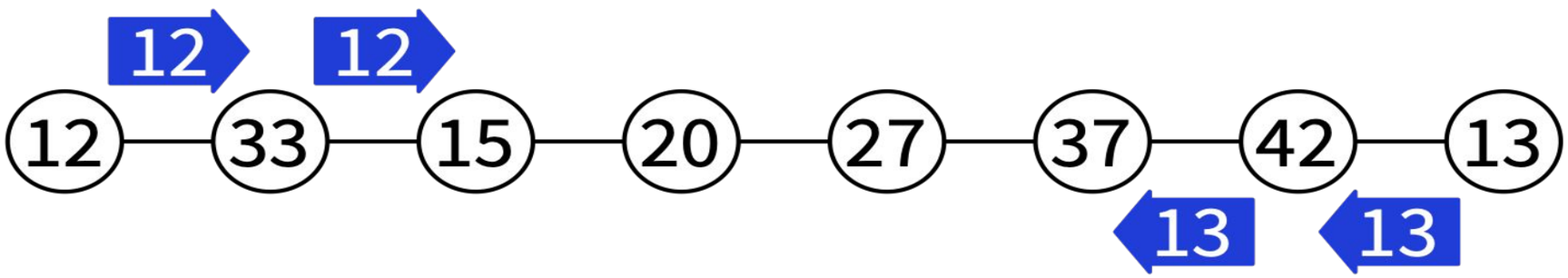
Algorithm for 2-coloring

- **Messages:**
  - "**ID x**" = there is an endpoint with identifier *x*
  - "**colour c**" = my colour is c



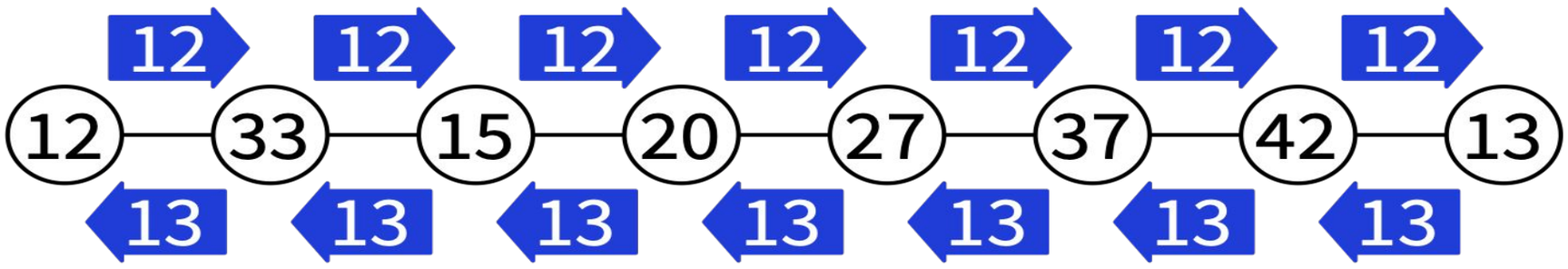
Algorithm for 2-coloring

- **Messages:**
  - "**ID *x***" = there is an endpoint with identifier *x*
  - "**colour *c***" = my colour is *c*



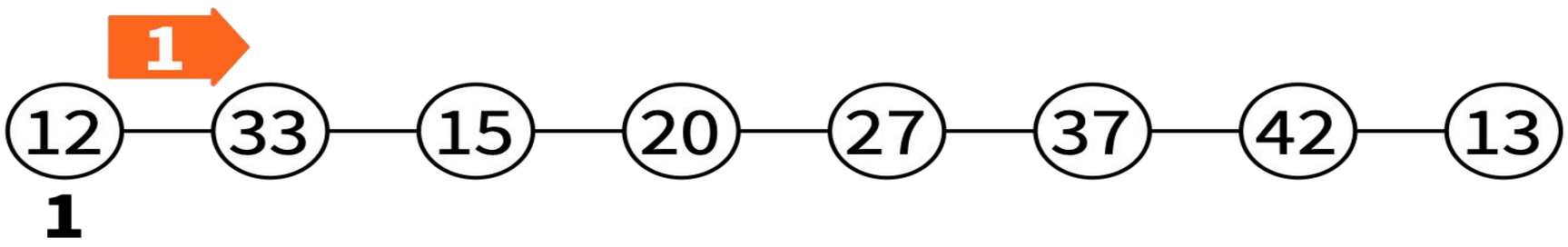
Algorithm for 2-coloring

- **Messages:**
  - "**ID x**" = there is an endpoint with identifier *x*
  - "**colour c**" = my colour is c



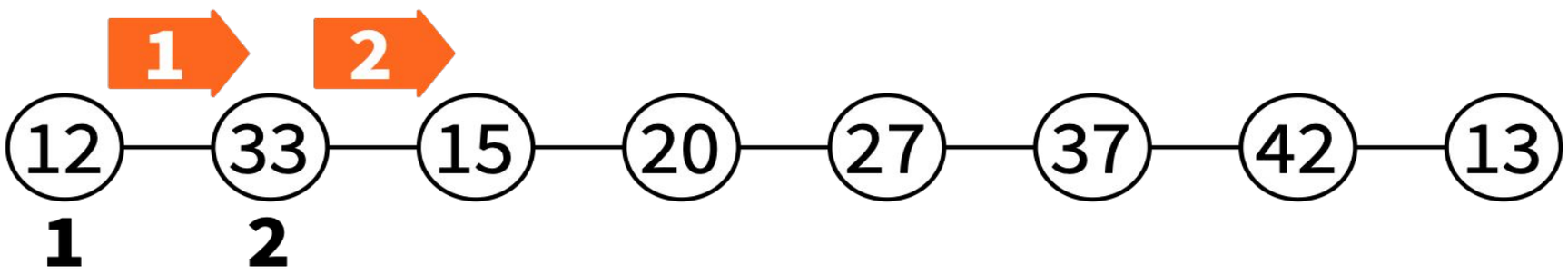
Algorithm for 2-coloring

- **Messages:**
  - "**ID *x***" = there is an endpoint with identifier *x*
  - "**colour c**" = my colour is c



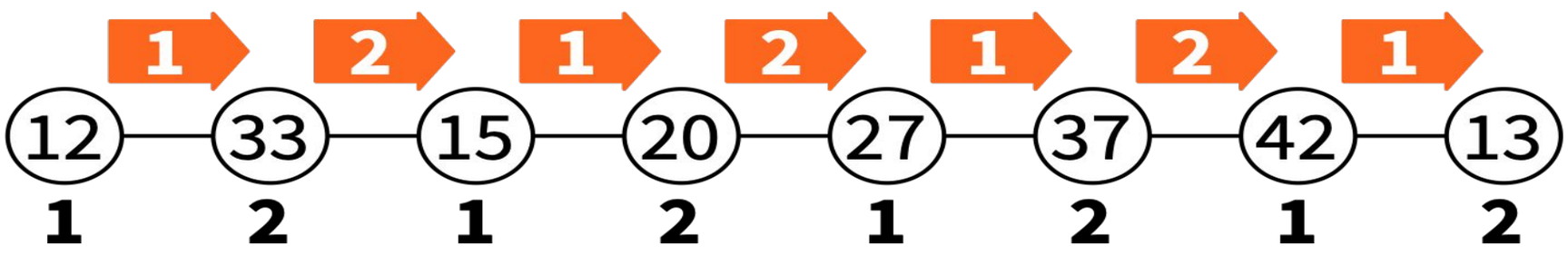
Algorithm for 2-coloring

- **Messages:**
  - "**ID *x***" = there is an endpoint with identifier *x*
  - "**colour c**" = my colour is c



Algorithm for 2-coloring

- **Messages:**
  - "**ID *x***" = there is an endpoint with identifier *x*
  - "**colour *c***" = my colour is c



BASIC DISTRIBUTED ALGORITHMS IN MESSAGE-PASSING SYSTEMS

Broadcast on a spanning tree

Broadcast over a Rooted Spanning Tree

- Problem : (Single message) Broadcast a distinguished processor p_r has a message M it wishes to send to all other processors
- Suppose processors already have information about a rooted spanning tree of the communication topology
  - tree: connected graph with no cycles
  - spanning tree: contains all processors
  - rooted: there is a unique root node p_r

Broadcast Over a Rooted Spanning Tree

- root initially sends *M* to its children
- when a processor receives *M* from its parent
  - sends *M* to its children
  - terminates (sets a local boolean to true)
- Implemented via *parent* and *children* local variables at each processor
  - indicate which incident channels lead to parent and children in the rooted spanning tree

Broadcast Over a Rooted Spanning Tree : pseudo-code

Algorithm 1 Spanning tree broadcast algorithm.

Initially ⟨M⟩ is in transit from p_r to all its children in the spanning tree.

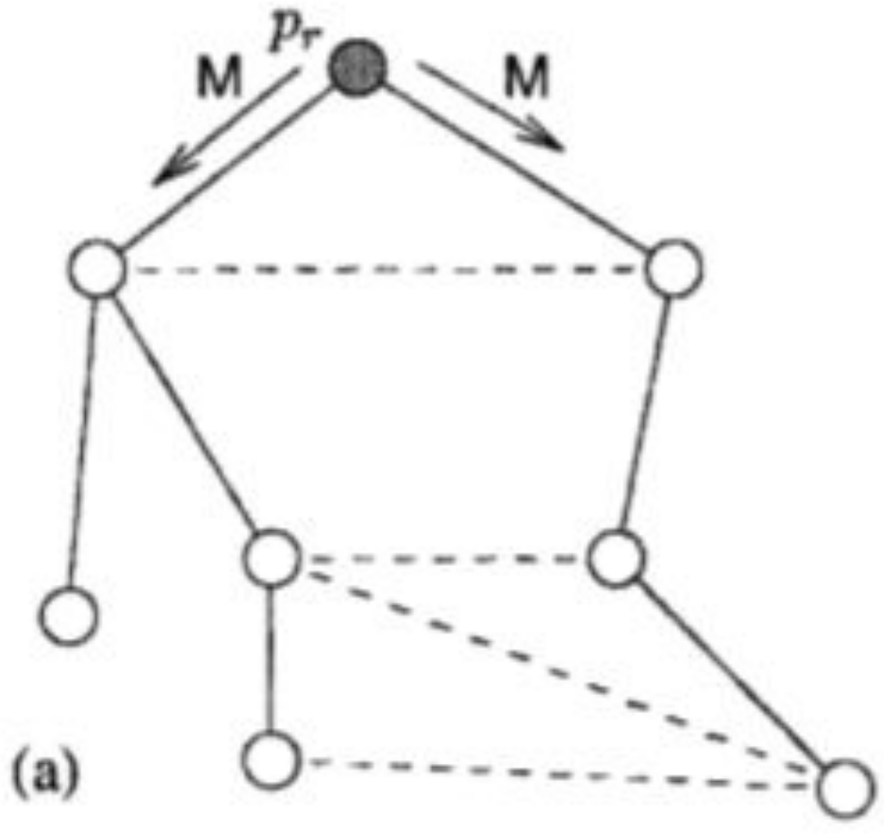
Code for p_r:

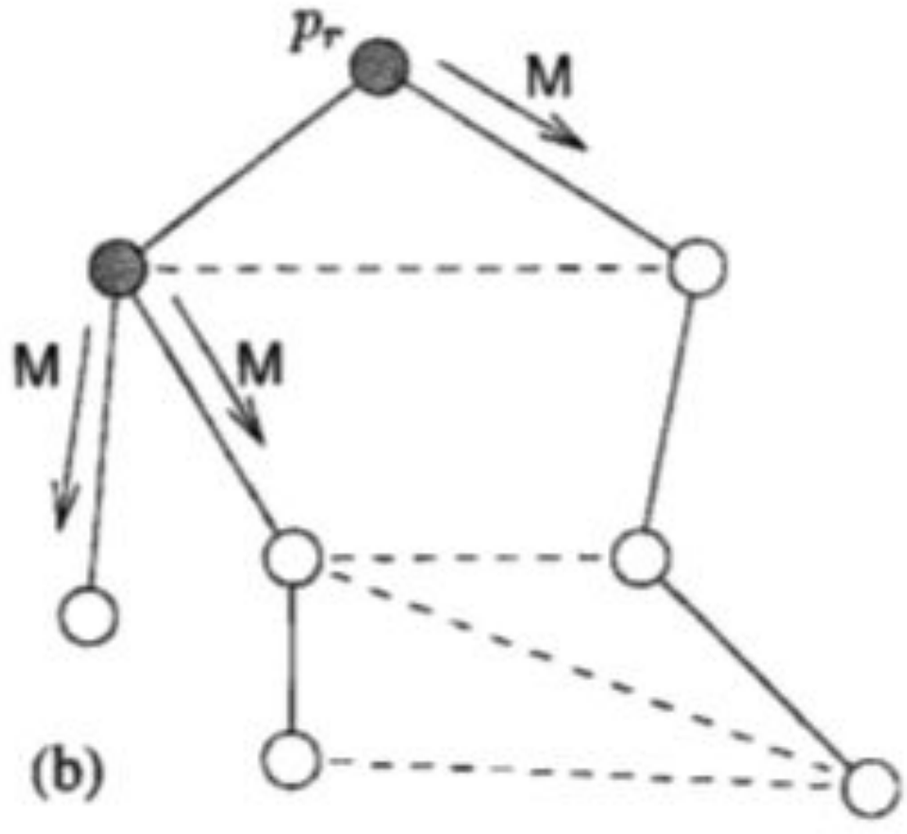
1: upon receiving no message: // first computation event by p_r
2: terminate

Code for p_i, $0 \leq i \leq n-1, i \neq r$:

3: upon receiving ⟨M⟩ from parent:
4: send ⟨M⟩ to all children
5: terminate

Broadcast Over a Rooted Spanning Tree : two steps in an asynchronous execution





Message Complexity Analysis

- Number of messages is $n - 1$, since
  - one message is sent over each spanning tree edge
  - A spanning tree of n nodes has exactly n-1 edges
- Same message complexity for synchronous and asynchronous model

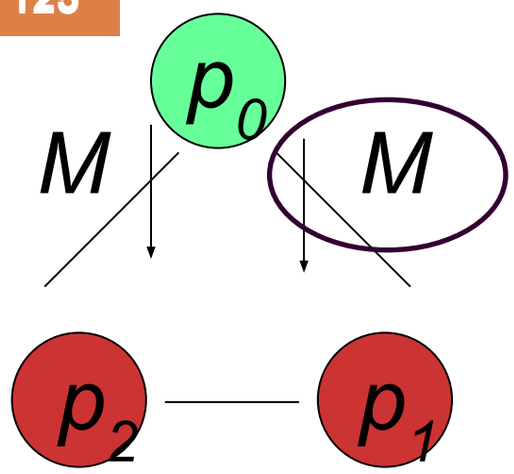
Broadcast without a spanning tree

- The algorithm starts from a given processor p_r which sends the message M to all its neighbors
- When processor p_i receives M for the first time, from some neighboring processor p_j
  - It sends M to its neighbors except p_j
- Message complexity : a processor will not send <M> more than once on any communication channel
  - <M> is sent at most twice on each communication channel (once by each processor using the channel)

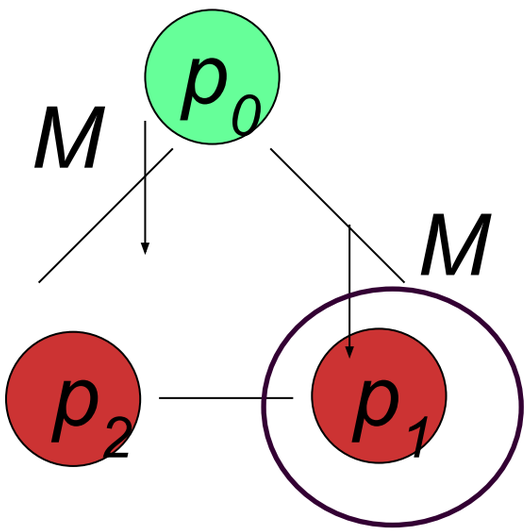
A simple algorithm to solve broadcast : called Flooding

- Each processor's local state consists of variable *color*, either red or green
- Initially:
  - p_0: *color* = green, all outbufs contain *M*
  - others: *color* = red, all outbufs empty
- Transition: If *M* is in an inbuf[j] and *color* = red, then change *color* to green and send *M* on all outbufs excluding outbuf[j]

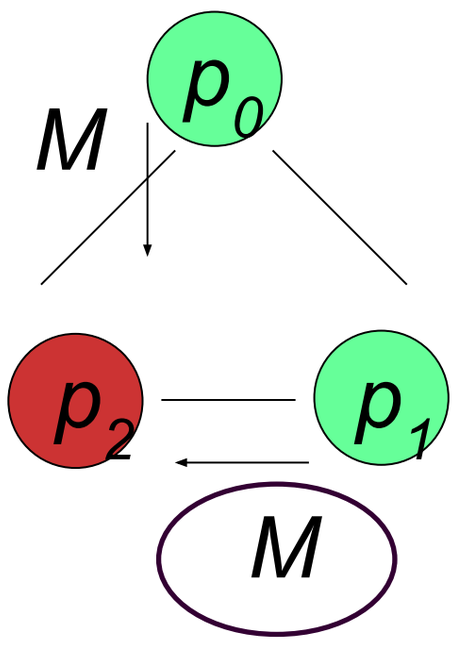
Example: (Asynchronous) Flooding



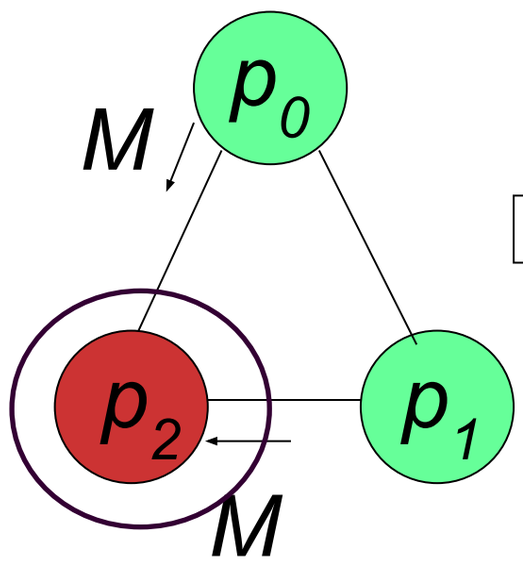
deliver event at p_1 from p_0



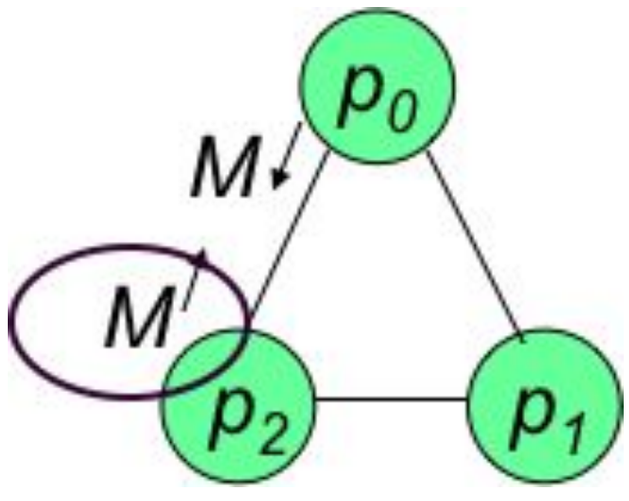
computation event by p_1



deliver event at p_2 from p_1



computation event by p_2



etc. to deliver all messages

Nondeterminism

- The previous execution is not the only admissible execution of the Flooding algorithm on that triangle.
- There are several, depending on the order in which messages are delivered.
- For instance, the message from p_0 could arrive at p_2 before the message from p_1 does.

Termination

- For technical reasons, admissible executions are defined as infinite.
- But often algorithms terminate.
- To model algorithm termination, identify *terminated* states of processors: states which, once entered, are never left
- Execution has *terminated* when all processors are terminated and no messages are in transit (in inbufs or outbufs)

Termination of Flooding Algorithm

- Define *terminated* processor states as those in which *color* = green

Message Complexity of Flooding Algorithm

- Message complexity: one message is sent over each edge in each direction. So number is $2m$, where m = number of edges
- m can be n(n-1)/2 where n is the number of processes

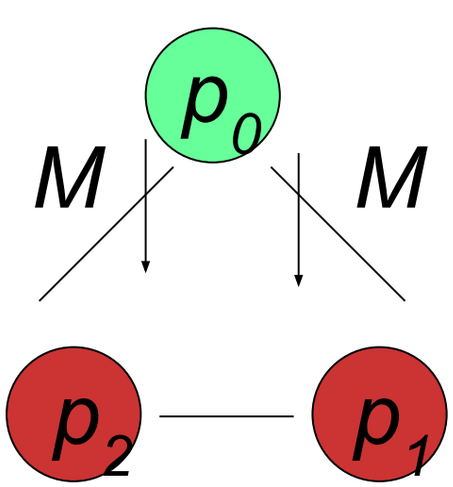
Example of Synchronous Model

- Suppose flooding algorithm is executed in synchronous model on the triangle.
- Round 1:
  - deliver M to p_1 from p_0
  - deliver M to p_2 from p_0
  - p_0 does nothing (as it has no incoming messages)
  - p_1 receives M, turns green and sends M to p_0 and p_1
  - p_2 receives M, turns green and sends M to p_0 and p_1

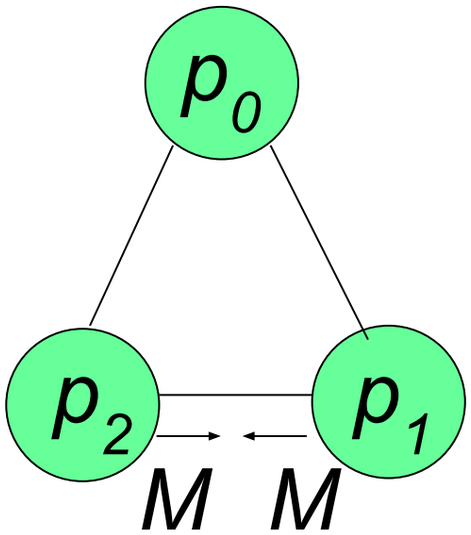
Example of Synchronous Model

- Round 2:
  - deliver M to p_0 from p_1
  - deliver M to p_0 from p_2
  - deliver M to p_1 from p_2
  - deliver M to p_2 from p_1
  - p_0 does nothing since its *color* variable is already green
  - p_1 does nothing since its *color* variable is already green
  - p_2 does nothing since its *color* variable is already green

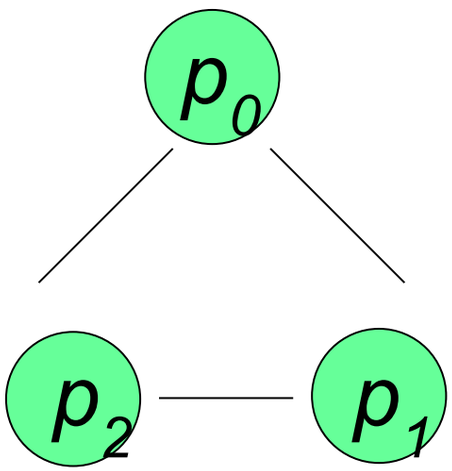
Example of Synchronous Model



round 1 events



round 2 events



Finding a Spanning Tree Given a Root

- Having a spanning tree is very convenient.
- How do you get one?
- Suppose a distinguished processor is known, to serve as the root.

Finding a Spanning Tree Given a Root : Algorithm description

- root sends M to all its neighbors

1. when non-root process *first* gets M
   - set the sender as its parent
   - send "parent" msg to sender
   - send M to all other neighbors (if no other neighbors, then terminate)
2. when get M otherwise
   - send ”already" msg to sender

- Note : it is possible that a process receives M for the first time from several processes (all these messages are in the inbufs). In this case pick one and execute 1. Execute 2. for the other messages.

Finding a Spanning Tree Given a Root : Algorithm description

- Each process pi manages three local variables :
  - *parent* is used to store the process that will be the parent of p_i in the spanning tree. It is initially equal to $\perp$
  - *children* is used to store the processes that will be the children of pi in the spanning tree. Initially it is equal to the empty set.
  - *other* to store the other processes. Initially it is equal to the empty set.
- "terminate" (for example in line 14) indicates that the process pi has entered terminated state.
  - use "parent" and "already" msgs to set children variables and know when to terminate (after hearing from all neighbors)

Finding a Spanning Tree Given a Root : Algorithm pseudo-code for processor p_i


```
1:  upon receiving no message:
2:      if p_i = p_r and parent = ⊥ then          // root has not yet sent ⟨M⟩
3:          send ⟨M⟩ to all neighbors
4:          parent := p_i

5:  upon receiving ⟨M⟩ from neighbor p_j:
6:      if parent = ⊥ then                        // p_i has not received ⟨M⟩ before
7:          parent := p_j
8:          send ⟨parent⟩ to p_j
9:          send ⟨M⟩ to all neighbors except p_j
10:     else send ⟨already⟩ to p_j

11: upon receiving ⟨parent⟩ from neighbor p_j:
12:     add p_j to children
13:     if children ∪ other contains all neighbors except parent then
14:         terminate

15: upon receiving ⟨already⟩ from neighbor p_j:
16:     add p_j to other
17:     if children ∪ other contains all neighbors except parent then
18:         terminate
```