Graphs

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

The graph data model

A graph consists of *nodes* and *edges*:

Nodes denote pieces of information;

Edges denote relationships between these pieces.

Given a graph data set, one can often *derive* other information or relationships.

The graph data model

A graph consists of *nodes* and *edges*:

Nodes denote pieces of information;

Edges denote relationships between these pieces.

Given a graph data set, one can often *derive* other information or relationships.

Many variations

- Nodes and edges can have *labels*;
- Nodes and edges can carry *weights*; and
- Edges can be *directed* or *undirected*.

The graph data model

A graph consists of *nodes* and *edges*:

Nodes denote pieces of information;

Edges denote relationships between these pieces.

Given a graph data set, one can often *derive* other information or relationships.

Many variations

- Nodes and edges can have *labels*;
- Nodes and edges can carry *weights*; and
- Edges can be *directed* or *undirected*.

Most data sources can be modeled as graphs, e.g., “Big Data”.

The graph data model

A graph consists of *nodes* and *edges*:

Nodes denote pieces of information;

Edges denote relationships between these pieces.

Given a graph data set, one can often *derive* other information or relationships.

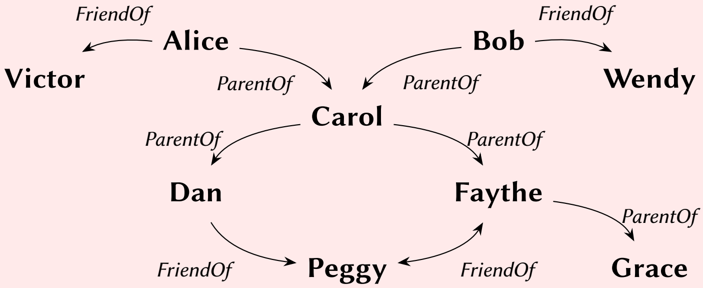
Many variations

- Nodes and edges can have *labels*;
- Nodes and edges can carry *weights*; and
- Edges can be *directed* or *undirected*.

Most data sources can be modeled as graphs, e.g., “Big Data”.
Standard graph algorithms can be used to solve many *different* problems.

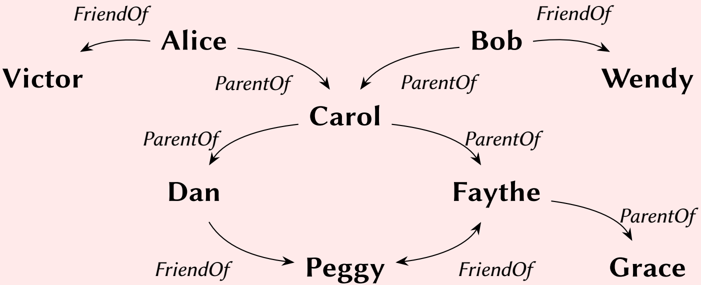
Example: Social networks

Source: Hellings et al., 2021.



Example: Social networks

Source: Hellings et al., 2021.

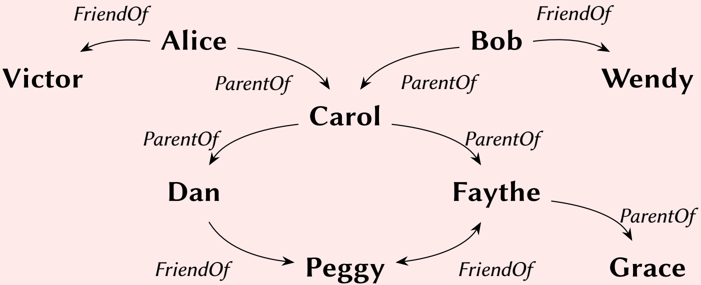


Nodes People.

Edges Relationships between them.

Example: Social networks

Source: Hellings et al., 2021.



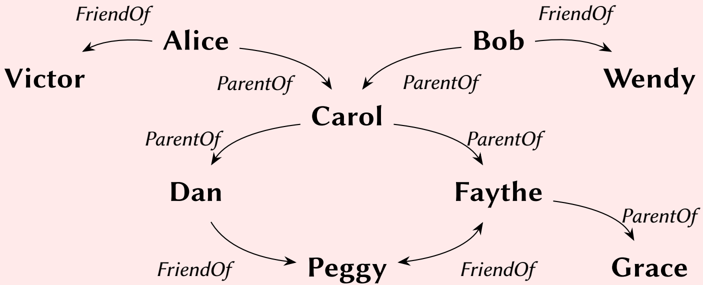
Nodes People.

Edges Relationships between them.

We can derive *GrandParentOf*, *AncestorOf*, ...

Example: Social networks

Source: Hellings et al., 2021.



Nodes People.

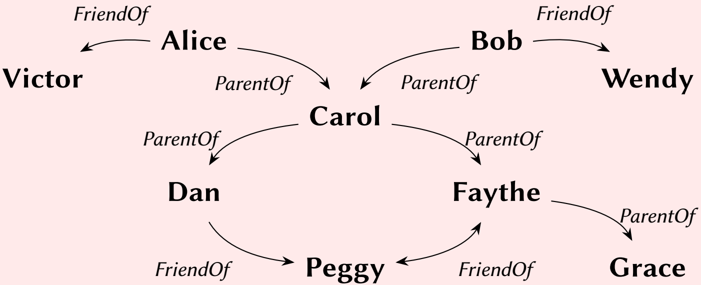
Edges Relationships between them.

We can derive *GrandParentOf*, *AncestorOf* , ...

How can one contact someone else via a friend-of-a-friend?

Example: Social networks

Source: Hellings et al., 2021.



Nodes People.

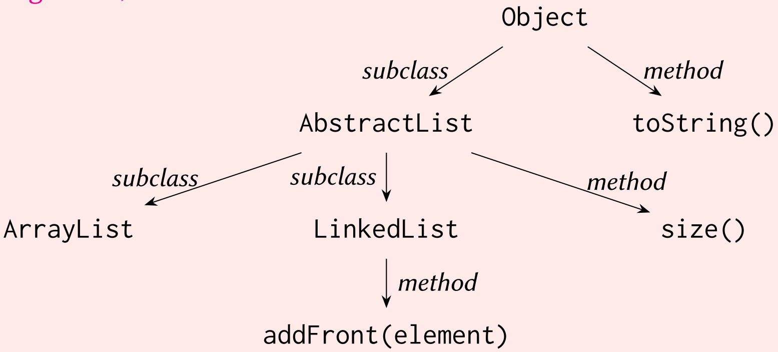
Edges Relationships between them.

We can derive *GrandParentOf*, *AncestorOf* , ...

How can one contact someone else via a friend-of-a-friend? → A shortest path!

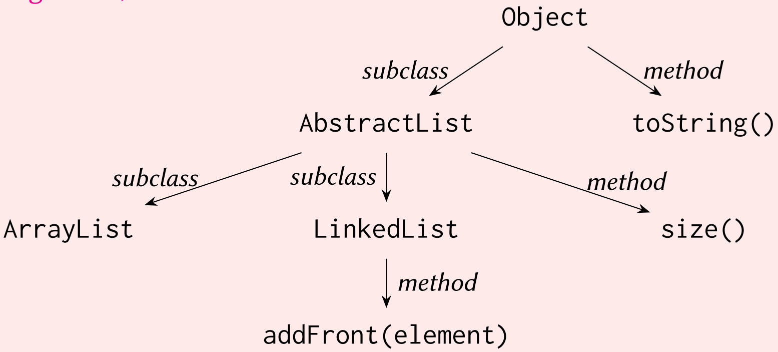
Example: Class hierarchy (trees)

Source: Hellings et al., 2020.



Example: Class hierarchy (trees)

Source: Hellings et al., 2020.

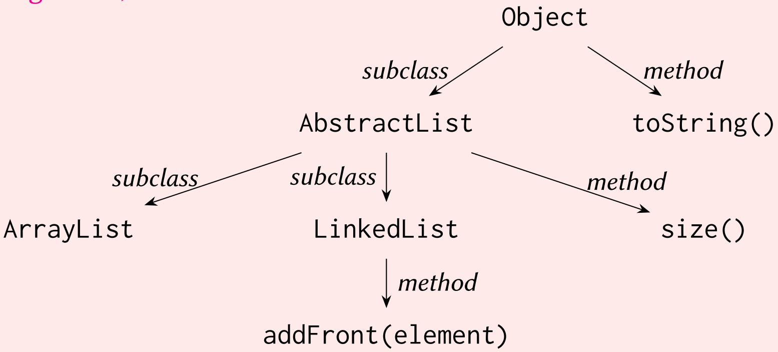


Nodes Names (classes, methods).

Edges Membership (subclass, method).

Example: Class hierarchy (trees)

Source: Hellings et al., 2020.



Nodes Names (classes, methods).

Edges Membership (subclass, method).

Question: does `LinkedList` have a method `toString()`?

Example: Rail Network

Source: *Classical geographer* at *Wikimedia Commons*.

Example: Rail Network

Source: *Classical geographer* at Wikimedia Commons.

Nodes Train stations.
Edges Rail connections.

Example: Rail Network

Source: *Classical geographer* at *Wikimedia Commons*.

Nodes Train stations.
Edges Rail connections.
Weights Maximum speed.

Example: Rail Network

Source: *Classical geographer* at Wikimedia Commons.

Nodes	Train stations.
Edges	Rail connections.
Weights	Maximum speed.

The shortest path problem

Which route should a train take to connect stations A and B (with minimal travel time)?

Example: Air traffic data

Source: *On-Time : Reporting Carrier On-Time Performance* at Bureau of Transportation Statistics.

OP_CARRIER	TAIL_NUM	ORIGIN	DEST	DEP_TIME	ARR_DELAY	DISTANCE
DL	N102DN	ATL	ORD	1329	-4.00	606.00
UA	N12754	BOS	EWR	1501	-14.00	200.00
AA	N103NN	SFO	JFK	554	-12.00	2586.00
WN	N935WN	SFO	DEN	1313	6.00	967.00

Example: Air traffic data

Source: *On-Time : Reporting Carrier On-Time Performance* at Bureau of Transportation Statistics.

OP_CARRIER	TAIL_NUM	ORIGIN	DEST	DEP_TIME	ARR_DELAY	DISTANCE
DL	N102DN	ATL	ORD	1329	-4.00	606.00
UA	N12754	BOS	EWR	1501	-14.00	200.00
AA	N103NN	SFO	JFK	554	-12.00	2586.00
WN	N935WN	SFO	DEN	1313	6.00	967.00

Nodes Air ports.

Edges Flights.

Example: Air traffic data

Source: *On-Time : Reporting Carrier On-Time Performance* at Bureau of Transportation Statistics.

OP_CARRIER	TAIL_NUM	ORIGIN	DEST	DEP_TIME	ARR_DELAY	DISTANCE
DL	N102DN	ATL	ORD	1329	-4.00	606.00
UA	N12754	BOS	EWR	1501	-14.00	200.00
AA	N103NN	SFO	JFK	554	-12.00	2586.00
WN	N935WN	SFO	DEN	1313	6.00	967.00

Nodes Air ports.

Edges Flights.

Weights Several: flight delay, distance, flight duration, ….

Example: Air traffic data

Source: *On-Time : Reporting Carrier On-Time Performance* at Bureau of Transportation Statistics.

OP_CARRIER	TAIL_NUM	ORIGIN	DEST	DEP_TIME	ARR_DELAY	DISTANCE
DL	N102DN	ATL	ORD	1329	-4.00	606.00
UA	N12754	BOS	EWR	1501	-14.00	200.00
AA	N103NN	SFO	JFK	554	-12.00	2586.00
WN	N935WN	SFO	DEN	1313	6.00	967.00

Nodes Air ports.

Edges Flights.

Weights Several: flight delay, distance, flight duration, ...

Which airports can I reach starting at X in at-most N stops?

Which airports can I reach starting at X in 7 hours of travel time?

Example: Air traffic data

Source: *On-Time : Reporting Carrier On-Time Performance* at Bureau of Transportation Statistics.

OP_CARRIER	TAIL_NUM	ORIGIN	DEST	DEP_TIME	ARR_DELAY	DISTANCE
DL	N102DN	ATL	ORD	1329	-4.00	606.00
UA	N12754	BOS	EWR	1501	-14.00	200.00
AA	N103NN	SFO	JFK	554	-12.00	2586.00
WN	N935WN	SFO	DEN	1313	6.00	967.00

Nodes Air ports.

Edges Flights.

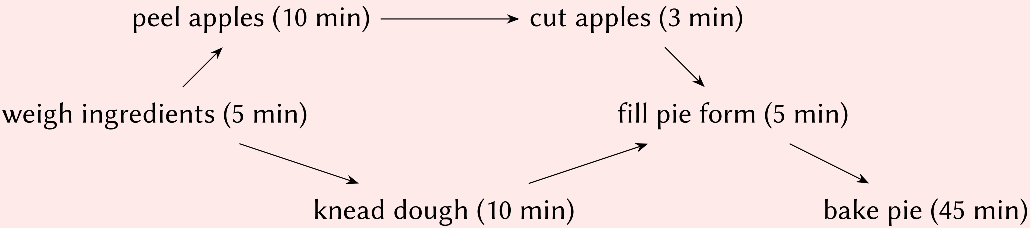
Weights Several: flight delay, distance, flight duration, ….

Which airports can I reach starting at X in at-most N stops?

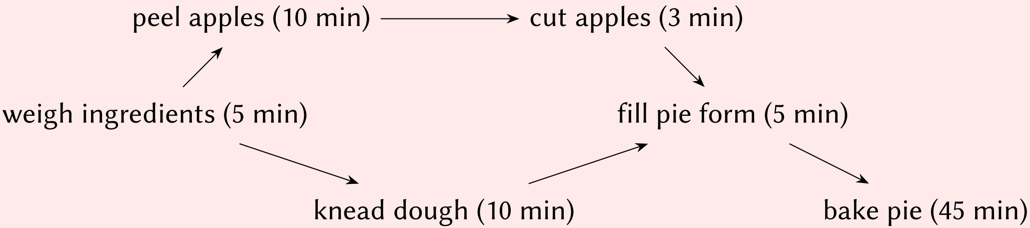
Which airports can I reach starting at X in 7 hours of travel time?

This data has a time component: a *temporal graph*.

Example: Schedules with dependencies (DAGs)



Example: Schedules with dependencies (DAGs)

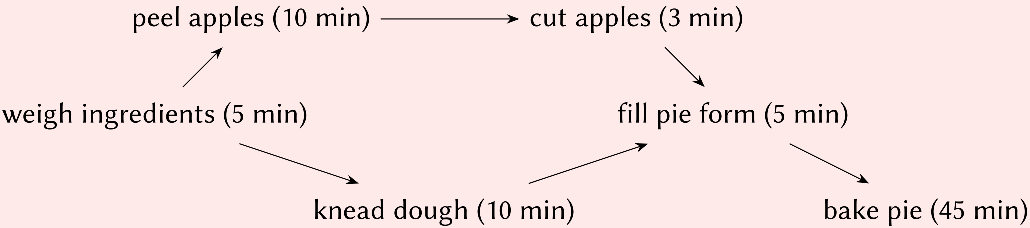


Nodes Steps of recipe.

Edges Dependencies between steps.

Weights Duration of each step.

Example: Schedules with dependencies (DAGs)



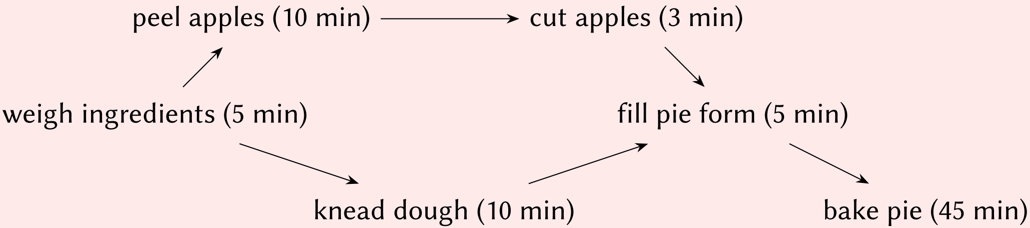
Nodes Steps of recipe.

Edges Dependencies between steps.

Weights Duration of each step.

Which tasks can I do concurrently?
How fast can a group bake a pie?

Example: Schedules with dependencies (DAGs)



Nodes Steps of recipe.

Edges Dependencies between steps.

Weights Duration of each step.

Which tasks can I do concurrently?

How fast can a group bake a pie? $\longrightarrow$ A *longest* path problem
(that we can turn into a *shortest* path problem).

Rational for graphs and graph algorithms

Let D be some dataset and let P be some *computational problem*.

Rational for graphs and graph algorithms

Let D be some dataset and let P be some *computational problem*.

- One can often translate (part of) D to some graph representation G.

Rational for graphs and graph algorithms

Let D be some dataset and let P be some *computational problem*.

- One can often translate (part of) D to some graph representation G.
- Next, one can translate problem P to a *graph problem* S.

Rational for graphs and graph algorithms

Let D be some dataset and let P be some *computational problem*.

- One can often translate (part of) D to some graph representation G.
- Next, one can translate problem P to a *graph problem* S.
- On G, we can run some *standard graph algorithm* to *solve* S.

Rational for graphs and graph algorithms

Let D be some dataset and let P be some *computational problem*.

- One can often translate (part of) D to some graph representation G.
- Next, one can translate problem P to a *graph problem* S.
- On G, we can run some *standard graph algorithm* to *solve* S.
- The solution for S can be translated to a solution for problem P.

Rational for graphs and graph algorithms

Let D be some dataset and let P be some *computational problem*.

- One can often translate (part of) D to some graph representation G.
- Next, one can translate problem P to a *graph problem* S.
- On G, we can run some *standard graph algorithm* to *solve* S.
- The solution for S can be translated to a solution for problem P.

Often, one can do these steps *implicitly*.

Rational for graphs and graph algorithms

Let D be some dataset and let P be some *computational problem*.

- One can often translate (part of) D to some graph representation G.
- Next, one can translate problem P to a *graph problem* S.
- On G, we can run some *standard graph algorithm* to *solve* S.
- The solution for S can be translated to a solution for problem P.

Often, one can do these steps *implicitly*.

We will see examples of this in the lectures and assignments!

Selected topics on graphs

- Formalization.
- Data structures to represent graphs.
- Traversing graphs:
  Reachability, finding cycles, shortest paths (without weights), topological sort, ...
- Minimum spanning trees.
- Finding shortest-paths (with weights).

Undirected graphs

Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E}$ a collection of *undirected edges* that consist of *node pairs*.

Undirected graphs

Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E}$ a collection of *undirected edges* that consist of *node pairs*.
  Undirected: if $(v, w) \in \mathcal{E}$, then also $(w, v) \in \mathcal{E}$!

Undirected graphs

Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E}$ a collection of *undirected edges* that consist of *node pairs*.
  Typically: at-most one edge between nodes: $\mathcal{E}$ is a set with $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$.

Undirected graphs

Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

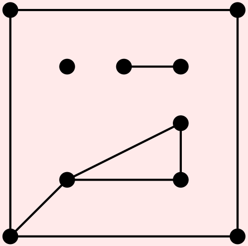
- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.
  Typically: at-most one edge between nodes: $\mathcal{E}$ is a set with $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$.

Undirected graphs

Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.

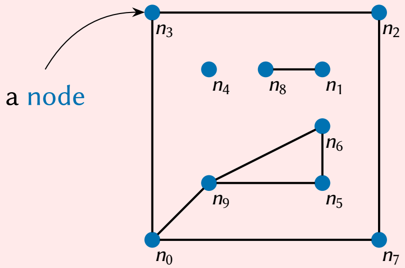


Undirected graphs

Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.

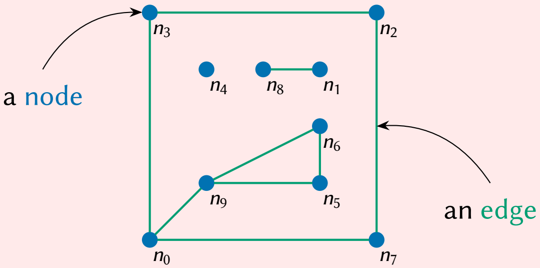


Undirected graphs

Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.



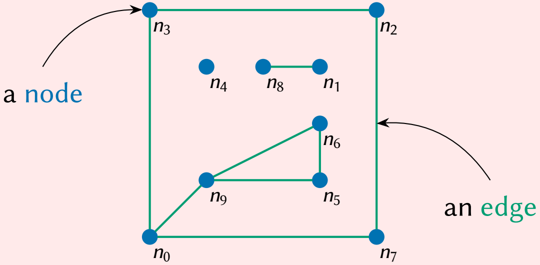
Undirected graphs

Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.

Nodes have unique identities, e.g., they are assigned unique numbers.



Undirected graphs

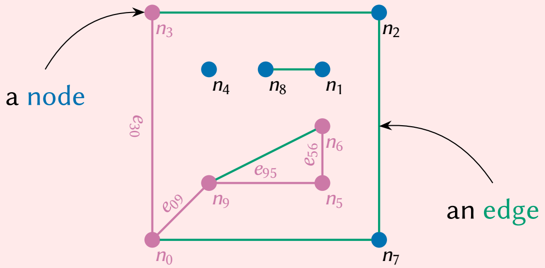
Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.

A *path* is a sequence of nodes and edges connecting two nodes.

Example: $n_3\, e_{30}\, n_0\, e_{09}\, n_9\, e_{95}\, n_5\, e_{56}\, n_6$.



Undirected graphs

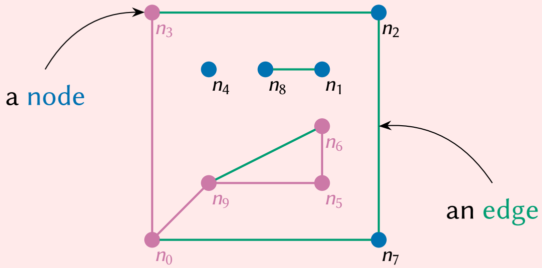
Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.

A *path* is a sequence of nodes and edges connecting two nodes.

Example: $n_3 n_0 n_9 n_5 n_6$ (at-most one edge per node pair).



Undirected graphs

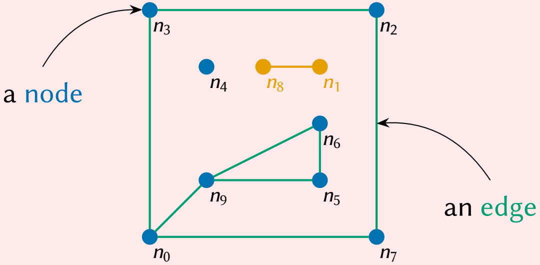
Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.

Two nodes are *connected* if there is a path between them.

Connected component: maximal subgraph in which all node pairs are connected.



Undirected graphs

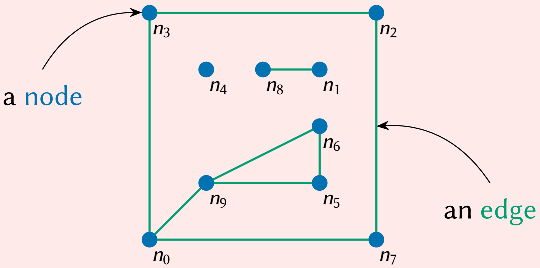
Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.

A graph is *connected* if all node pairs are connected.

This graph is *not* connected: there are three disconnected components!



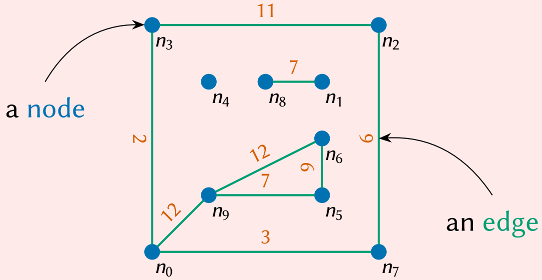
Undirected graphs

Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.

In a *weighted undirected graph*, each edge has a weight.
Typically modeled via a *weight function* *weight*, e.g., $\mathit{weight} : \mathcal{E} \to \mathbb{N}$.



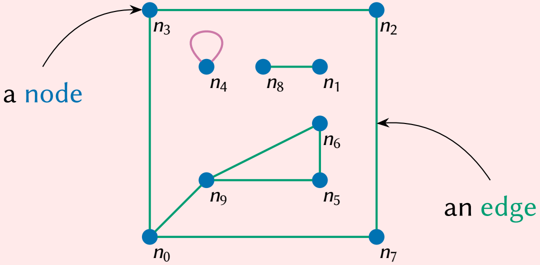
Undirected graphs

Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.

We can have edges from nodes to themselves: self-loops.
(we will mostly ignore self-loops).

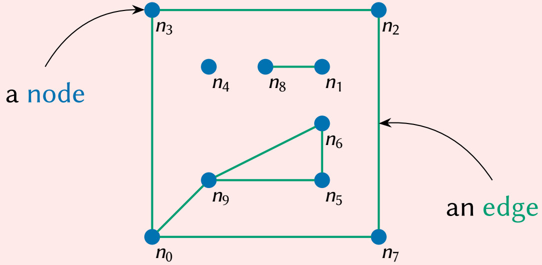


Undirected graphs

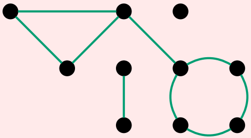
Definition

An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.



The same graph?

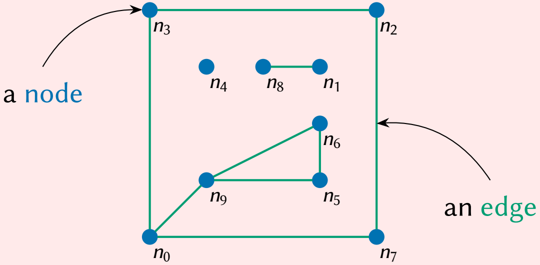


Undirected graphs

Definition

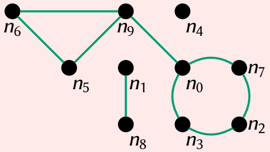
An *undirected graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *undirected edges* that consist of *node pairs*.



The same graph?

Yes

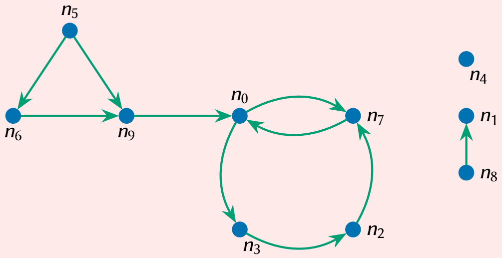


Directed graphs

Definition

A *directed graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *edges* that consist of *node pairs*.



Directed graphs

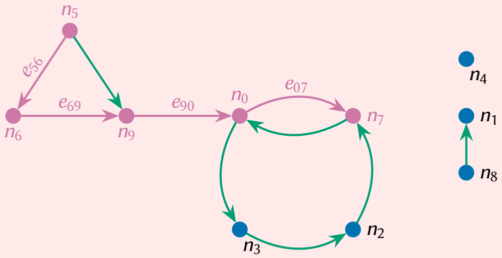
Definition

A *directed graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *edges* that consist of *node pairs*.

A *path* is a sequence of nodes and edges connecting two nodes.

Example: $n_5\, e_{56}\, n_6\, e_{69}\, n_9\, e_{90}\, n_0\, e_{07}\, n_7$.



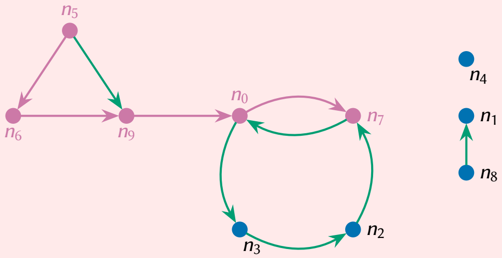
Directed graphs

Definition

A *directed graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *edges* that consist of *node pairs*.

A *path* is a sequence of nodes and edges connecting two nodes.
Example: $n_5 n_6 n_9 n_0 n_7$ (at-most one edge per node pair).



Directed graphs

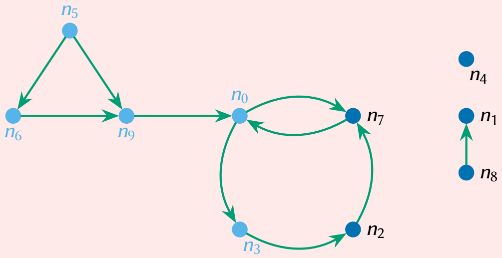
Definition

A *directed graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *edges* that consist of *node pairs*.

A *path* is a sequence of nodes and edges connecting two nodes.

$n_3 n_0 n_9 n_5 n_6$ does not follow direction $\longrightarrow$ *not* a path!



Directed graphs

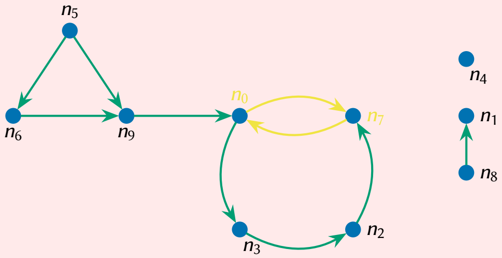
Definition

A *directed graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *edges* that consist of *node pairs*.

A *cycle* is a path with at-least one edge from a node to itself.

Example: the cycles $n_0 n_7$ and $n_7 n_0$.



Directed graphs

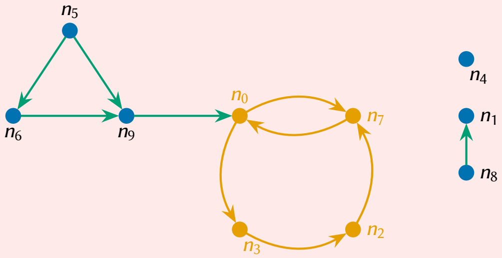
Definition

A *directed graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *edges* that consist of *node pairs*.

Two nodes are *strongly connected* if there is a path between them.

Strongly …component: maximal subgraph in which all node pairs are strongly connected.



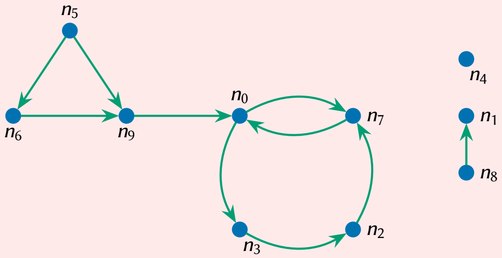
Directed graphs

Definition

A *directed graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *edges* that consist of *node pairs*.

A graph is *strongly connected* if all node pairs are strongly connected.
This graph is *not* strongly connected: e.g., no paths toward n_4.



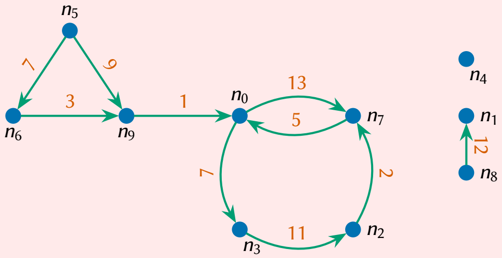
Directed graphs

Definition

A *directed graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *edges* that consist of *node pairs*.

In a *weighted directed graph*, each edge has a weight.
Typically modeled via a *weight function* *weight*, e.g., $\mathit{weight} : \mathcal{E} \to \mathbb{N}$.



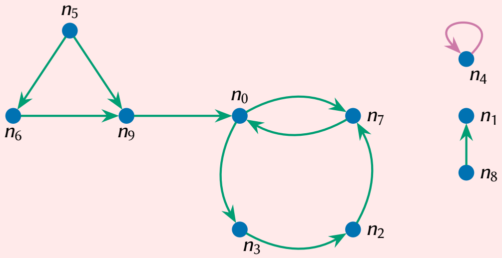
Directed graphs

Definition

A *directed graph* is a pair $(\mathcal{N}, \mathcal{E})$ with

- $\mathcal{N}$ a set of *nodes* (or *vertices*); and
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a set of *edges* that consist of *node pairs*.

We can have edges from nodes to themselves: self-loops.
(we will mostly ignore self-loops).



Implementing graphs

Consider a directed or undirected graph,
possibly with a weight function.

Which basic operations do we want?

Implementing graphs

Consider a directed or undirected graph,
possibly with a weight function.

Which basic operations do we want?

- Adding and removing nodes?
- Adding and removing edges?

Implementing graphs

Consider a directed or undirected graph,
possibly with a weight function.

Which basic operations do we want?

- Adding and removing nodes?
- Adding and removing edges?
- Check whether an edge exists between a node pair?

Implementing graphs

Consider a directed or undirected graph,
possibly with a weight function.

Which basic operations do we want?

- Adding and removing nodes?
- Adding and removing edges?
- Check whether an edge exists between a node pair?
- Iterate over all (incoming and outgoing) edges of a node?

Implementing graphs

Consider a directed or undirected graph,
possibly with a weight function.

Which basic operations do we want?

- Adding and removing nodes?
- Adding and removing edges?
- Check whether an edge exists between a node pair?
- Iterate over all (incoming and outgoing) edges of a node?
- Given an edge, check or change the weight?

The matrix representation

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.
Assume each node $n \in \mathcal{N}$ has a unique identifier $\text{id}(n)$ with $0 \leq \text{id}(n) < |N|$.

Matrix representation

Let M be a $|\mathcal{N}| \times |\mathcal{N}|$-*matrix* ($M$ is a *two-dimensional array*).
For every pair of nodes (m, n), set $M[\text{id}(m), \text{id}(n)] := (m, n) \in \mathcal{E}$.

The matrix representation

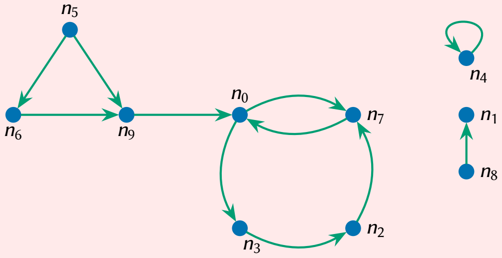
Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.

Assume each node $n \in \mathcal{N}$ has a unique identifier $\mathrm{id}(n)$ with $0 \leq \mathrm{id}(n) < |\mathcal{N}|$.

Matrix representation

Let M be a $|\mathcal{N}| \times |\mathcal{N}|$-*matrix* ($M$ is a *two-dimensional array*).

For every pair of nodes (m, n), set $M[\mathrm{id}(m), \mathrm{id}(n)] := (m, n) \in \mathcal{E}$.

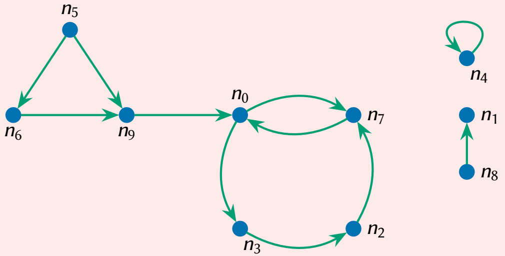


The matrix representation

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.
Assume each node $n \in \mathcal{N}$ has a unique identifier $\text{id}(n)$ with $0 \leq \text{id}(n) < |\mathcal{N}|$.

Matrix representation

Let M be a $|\mathcal{N}| \times |\mathcal{N}|$*-matrix* ($M$ is a *two-dimensional array*).
For every pair of nodes (m, n), set $M[\text{id}(m), \text{id}(n)] := (m, n) \in \mathcal{E}$.



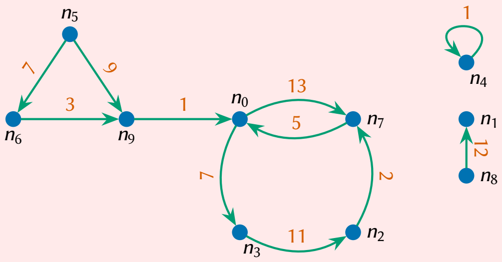
	0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0
3	0	0	1	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0	0	0	1
7	1	0	0	0	0	0	0	0	0	0
8	0	1	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0	0

The matrix representation

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.
Assume each node $n \in \mathcal{N}$ has a unique identifier $\mathrm{id}(n)$ with $0 \leq \mathrm{id}(n) < |\mathcal{N}|$.

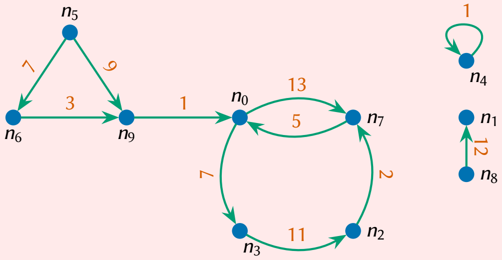
Matrix representation

Let M be a *$|\mathcal{N}| \times |\mathcal{N}|$-matrix* ($M$ is a *two-dimensional array*).
For every pair of nodes (m, n), set $M[\mathrm{id}(m), \mathrm{id}(n)] := (m, n) \in \mathcal{E}$.



	0	1	2	3	4	5	6	7	8	9
0	*	*	*	7	*	*	*	13	*	*
1	*	*	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	2	*	*
3	*	*	11	*	*	*	*	*	*	*
4	*	*	*	*	1	*	*	*	*	*
5	*	*	*	*	*	*	7	*	*	9
6	*	*	*	*	*	*	*	*	*	3
7	5	*	*	*	*	*	*	*	*	*
8	*	12	*	*	*	*	*	*	*	*
9	1	*	*	*	*	*	*	*	*	*

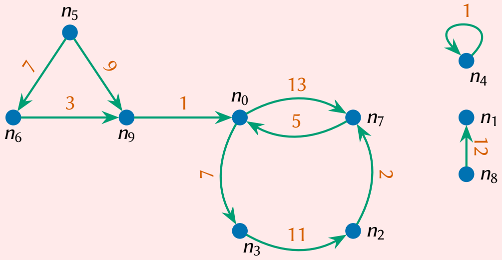
The matrix representation



	0	1	2	3	4	5	6	7	8	9
0	*	*	*	7	*	*	*	13	*	*
1	*	*	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	2	*	*
3	*	*	11	*	*	*	*	*	*	*
4	*	*	*	*	1	*	*	*	*	*
5	*	*	*	*	*	*	7	*	*	9
6	*	*	*	*	*	*	*	*	*	3
7	5	*	*	*	*	*	*	*	*	*
8	*	12	*	*	*	*	*	*	*	*
9	1	*	*	*	*	*	*	*	*	*

- Adding and removing nodes?
- Adding and removing edges (n, m)?
- Check whether an edge (n, m) exists?
- Iterate over all incoming edges of node n?
- Iterate over all outgoing edges of node n?
- Check or change the weight of (n, m)?

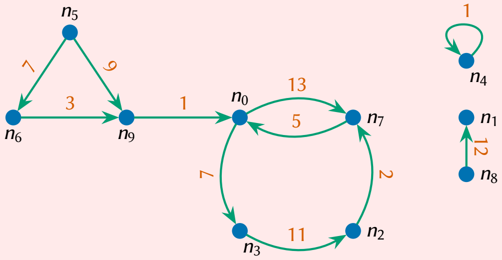
The matrix representation



	0	1	2	3	4	5	6	7	8	9
0	*	*	*	7	*	*	*	13	*	*
1	*	*	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	2	*	*
3	*	*	11	*	*	*	*	*	*	*
4	*	*	*	*	1	*	*	*	*	*
5	*	*	*	*	*	*	7	*	*	9
6	*	*	*	*	*	*	*	*	*	3
7	5	*	*	*	*	*	*	*	*	*
8	*	12	*	*	*	*	*	*	*	*
9	1	*	*	*	*	*	*	*	*	*

- Adding and removing nodes?
- Adding and removing edges (n, m)? $\longrightarrow \Theta(1)$
- Check whether an edge (n, m) exists? $\longrightarrow \Theta(1)$
- Iterate over all incoming edges of node n?
- Iterate over all outgoing edges of node n?
- Check or change the weight of (n, m)? $\longrightarrow \Theta(1)$

The matrix representation



	0	1	2	3	4	5	6	7	8	9
0	*	*	*	7	*	*	*	13	*	*
1	*	*	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	2	*	*
3	*	*	11	*	*	*	*	*	*	*
4	*	*	*	*	1	*	*	*	*	*
5	*	*	*	*	*	*	7	*	*	9
6	*	*	*	*	*	*	*	*	*	3
7	5	*	*	*	*	*	*	*	*	*
8	*	12	*	*	*	*	*	*	*	*
9	1	*	*	*	*	*	*	*	*	*

- Adding and removing nodes? $\longrightarrow \Theta\left(|\mathcal{N}|^2\right)$ (copy to new matrix).
- Adding and removing edges (n, m)? $\longrightarrow \Theta\left(1\right)$
- Check whether an edge (n, m) exists? $\longrightarrow \Theta\left(1\right)$
- Iterate over all incoming edges of node n? $\longrightarrow \Theta\left(|\mathcal{N}|\right)$ (scan a column)
- Iterate over all outgoing edges of node n? $\longrightarrow \Theta\left(|\mathcal{N}|\right)$ (scan a row)
- Check or change the weight of (n, m)? $\longrightarrow \Theta\left(1\right)$

The adjacency list representation

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.
Assume each node $n \in \mathcal{N}$ has a unique identifier $\text{id}(n)$ with $0 \leq \text{id}(n) < |\mathcal{N}|$.

Adjacency list representation

Let $A[0 \dots |\mathcal{N}|)$ be an array of *bags*.
For every edge $(m, n) \in \mathcal{E}$, Add (m, n) to the bag $A[\text{id}(m)]$.

The adjacency list representation

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.
Assume each node $n \in \mathcal{N}$ has a unique identifier $\mathrm{id}(n)$ with $0 \leq \mathrm{id}(n) < |N|$.

Adjacency list representation

Let $A[0 \ldots |\mathcal{N}|)$ be an array of *bags*.
For every edge $(m, n) \in \mathcal{E}$, Add (m, n) to the bag $A[\mathrm{id}(m)]$.

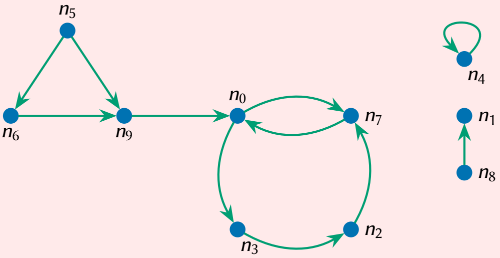
- The *standard* adjacency list stores *outgoing* edges.
  If needed, one can also store *incoming edges* or *both*.
- $A[i]$ is a *bag*, e.g., linked list, dynamic array, search tree, hash table, ...
- A can be a *dynamic array* to support adding nodes efficiently.
- A can be a *dictionary* mapping nodes onto their adjacency lists.
  Useful when nodes do not have identifiers, not all nodes have edges, ...

The adjacency list representation

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.
Assume each node $n \in \mathcal{N}$ has a unique identifier $\text{id}(n)$ with $0 \leq \text{id}(n) < |\mathcal{N}|$.

Adjacency list representation

Let $A[0 \dots |\mathcal{N}|)$ be an array of *bags*.
For every edge $(m, n) \in \mathcal{E}$, Add (m, n) to the bag $A[\text{id}(m)]$.



The adjacency list representation

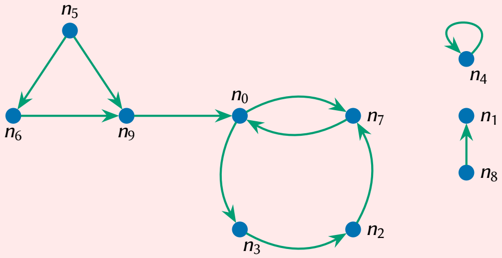
Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.

Assume each node $n \in \mathcal{N}$ has a unique identifier $\text{id}(n)$ with $0 \leq \text{id}(n) < |\mathcal{N}|$.

Adjacency list representation

Let $A[0 \dots |\mathcal{N}|)$ be an array of *bags*.

For every edge $(m, n) \in \mathcal{E}$, Add (m, n) to the bag $A[\text{id}(m)]$.



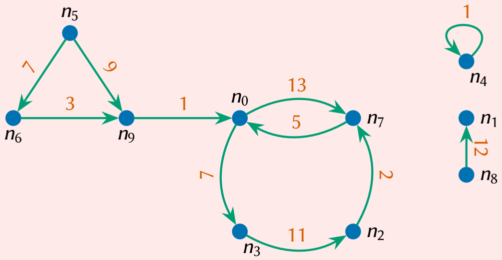
0	$[(n_0, n_3), (n_0, n_7)]$
1	$[]$
2	$[(n_2, n_7)]$
3	$[(n_3, n_2)]$
4	$[(n_4, n_4)]$
5	$[(n_5, n_6), (n_5, n_9)]$
6	$[(n_6, n_9)]$
7	$[(n_7, n_0)]$
8	$[(n_8, n_1)]$
9	$[(n_9, n_0)]$

The adjacency list representation

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.
Assume each node $n \in \mathcal{N}$ has a unique identifier $\mathrm{id}(n)$ with $0 \leq \mathrm{id}(n) < |\mathcal{N}|$.

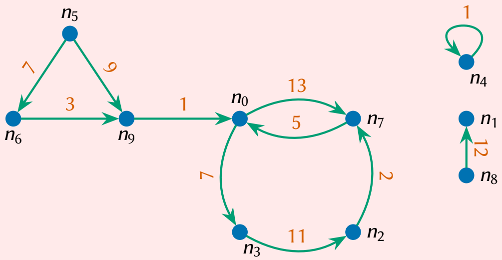
Adjacency list representation

Let $A[0 \ldots |\mathcal{N}|)$ be an array of *bags*.
For every edge $(m, n) \in \mathcal{E}$, Add (m, n) to the bag $A[\mathrm{id}(m)]$.



0	$[(n_0, n_3) : 7, (n_0, n_7) : 13]$
1	$[]$
2	$[(n_2, n_7) : 2]$
3	$[(n_3, n_2) : 11]$
4	$[(n_4, n_4) : 1]$
5	$[(n_5, n_6) : 7, (n_5, n_9) : 9]$
6	$[(n_6, n_9) : 3]$
7	$[(n_7, n_0) : 5]$
8	$[(n_8, n_1) : 12]$
9	$[(n_9, n_0) : 1]$

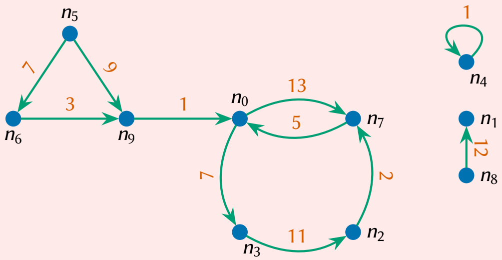
The adjacency list representation



0	$[(n_0, n_3) : 7, (n_0, n_7) : 13]$
1	$[]$
2	$[(n_2, n_7) : 2]$
3	$[(n_3, n_2) : 11]$
4	$[(n_4, n_4) : 1]$
5	$[(n_5, n_6) : 7, (n_5, n_9) : 9]$
6	$[(n_6, n_9) : 3]$
7	$[(n_7, n_0) : 5]$
8	$[(n_8, n_1) : 12]$
9	$[(n_9, n_0) : 1]$

- Adding and removing nodes?
- Adding and removing edges (n, m)?
- Check whether an edge (n, m) exists?
- Iterate over all *incoming* edges of node n?
- Iterate over all *outgoing* edges of node n?
- Check or change the weight of (n, m)?

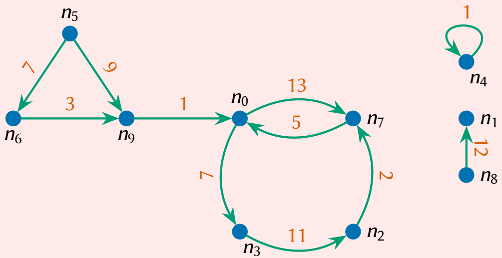
The adjacency list representation



0	$[(n_0, n_3) : 7, (n_0, n_7) : 13]$
1	$[]$
2	$[(n_2, n_7) : 2]$
3	$[(n_3, n_2) : 11]$
4	$[(n_4, n_4) : 1]$
5	$[(n_5, n_6) : 7, (n_5, n_9) : 9]$
6	$[(n_6, n_9) : 3]$
7	$[(n_7, n_0) : 5]$
8	$[(n_8, n_1) : 12]$
9	$[(n_9, n_0) : 1]$

- Adding and removing nodes?
- Adding and removing edges (n, m)?
- Check whether an edge (n, m) exists?
- Iterate over all *incoming* edges of node n?
- Iterate over all *outgoing* edges of node n?
- Check or change the weight of (n, m)? $\longrightarrow \Theta(1)$

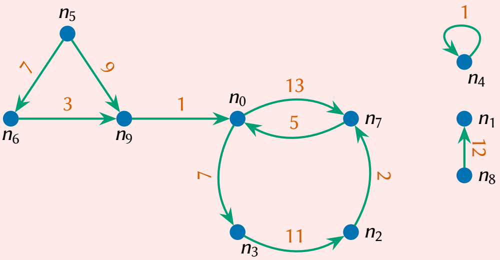
The adjacency list representation



0	$[(n_0, n_3) : 7, (n_0, n_7) : 13]$
1	$[]$
2	$[(n_2, n_7) : 2]$
3	$[(n_3, n_2) : 11]$
4	$[(n_4, n_4) : 1]$
5	$[(n_5, n_6) : 7, (n_5, n_9) : 9]$
6	$[(n_6, n_9) : 3]$
7	$[(n_7, n_0) : 5]$
8	$[(n_8, n_1) : 12]$
9	$[(n_9, n_0) : 1]$

- Adding and removing nodes? $\rightarrow \Theta(|\mathcal{N}|)$ (copy array).
- Adding and removing edges (n, m)? $\rightarrow \Theta(|\mathcal{N}|)$ (adding to bag).
- Check whether an edge (n, m) exists? $\rightarrow \Theta(|\mathcal{N}|)$ (searching bag)
- Iterate over all *incoming* edges of node n? $\rightarrow \Theta(|\mathcal{E}|)$ (scan all bags)
- Iterate over all *outgoing* edges of node n? $\rightarrow \Theta(|\mathcal{N}|)$ (scan a bag)
- Check or change the weight of (n, m)? $\rightarrow \Theta(1)$

The adjacency list representation



0	$[(n_0, n_3) : 7, (n_0, n_7) : 13]$
1	$[]$
2	$[(n_2, n_7) : 2]$
3	$[(n_3, n_2) : 11]$
4	$[(n_4, n_4) : 1]$
5	$[(n_5, n_6) : 7, (n_5, n_9) : 9]$
6	$[(n_6, n_9) : 3]$
7	$[(n_7, n_0) : 5]$
8	$[(n_8, n_1) : 12]$
9	$[(n_9, n_0) : 1]$

Let $\text{out}(n) = \{(n, m) \in \mathcal{E}\}$ be all *outgoing* edges of node n.

- Adding and removing nodes? $\longrightarrow \Theta(|\mathcal{N}|)$ (copy array).
- Adding and removing edges (n, m)? $\longrightarrow \Theta(|\text{out}(n)|)$ (adding to bag).
- Check whether an edge (n, m) exists? $\longrightarrow \Theta(|\text{out}(n)|)$ (searching bag)
- Iterate over all *incoming* edges of node n? $\longrightarrow \Theta(|\mathcal{E}|)$ (scan all bags)
- Iterate over all *outgoing* edges of node n? $\longrightarrow \Theta(|\text{out}(n)|)$ (scan a bag)
- Check or change the weight of (n, m)? $\longrightarrow \Theta(1)$

A comparison of representations

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.

Dense graph graph $\mathcal{G}$ is *dense* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|^2\right)$.

Sparse graph graph $\mathcal{G}$ is *spase* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|\right)$.

A comparison of representations

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.

Dense graph: graph $\mathcal{G}$ is *dense* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|^2\right)$. $\longrightarrow$ most node pairs are edges!

Sparse graph: graph $\mathcal{G}$ is *spase* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|\right)$. $\longrightarrow$ most node pairs are *not* edges!

	Matrix		Adjacency List	
	Sparse	Dense	Sparse	Dense
Memory usage	$\Theta\left(\lvert\mathcal{N}\rvert^2\right)$		$\Theta\left(\lvert\mathcal{N}\rvert + \lvert\mathcal{E}\rvert\right)$	
Adding nodes	$\Theta\left(\lvert\mathcal{N}\rvert^2\right)$		$\Theta\left(\lvert\mathcal{N}\rvert\right)$	
Adding edge (n, m)	$\Theta\left(1\right)$		$\Theta\left(\lvert\text{out}(n)\rvert\right)$	
Checking edge (n, m)	$\Theta\left(1\right)$		$\Theta\left(\lvert\text{out}(n)\rvert\right)$	
Incoming edges of n	$\Theta\left(\lvert\mathcal{N}\rvert\right)$		$\Theta\left(\lvert\mathcal{E}\rvert\right)$	
Outgoing edges of n	$\Theta\left(\lvert\mathcal{N}\rvert\right)$		$\Theta\left(\lvert\text{out}(n)\rvert\right)$	
Weight of edge (n, m)	$\Theta\left(1\right)$		$\Theta\left(\lvert\text{out}(n)\rvert\right)$	

A comparison of representations

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.

Dense graph graph $\mathcal{G}$ is *dense* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|^2\right)$. $\rightarrow$ most node pairs are edges!

Sparse graph graph $\mathcal{G}$ is *spase* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|\right)$. $\rightarrow$ most node pairs are *not* edges!

	Matrix		Adjacency List	
	Sparse	Dense	Sparse	Dense
Memory usage	$\Theta\left(\lvert\mathcal{N}\rvert^2\right)$	$\Theta\left(\lvert\mathcal{N}\rvert^2\right)$	$\Theta\left(\lvert\mathcal{N}\rvert + \lvert\mathcal{E}\rvert\right)$	$\Theta\left(\lvert\mathcal{N}\rvert^2\right)$
Adding nodes	$\Theta\left(\lvert\mathcal{N}\rvert^2\right)$	$\Theta\left(\lvert\mathcal{N}\rvert^2\right)$	$\Theta\left(\lvert\mathcal{N}\rvert\right)$	$\Theta\left(\lvert\mathcal{N}\rvert^2\right)$
Adding edge (n, m)	$\Theta(1)$	$\Theta(1)$	$\Theta\left(\lvert\text{out}(n)\rvert\right)$	$\Theta\left(\lvert\text{out}(n)\rvert\right)$
Checking edge (n, m)	$\Theta(1)$	$\Theta(1)$	$\Theta\left(\lvert\text{out}(n)\rvert\right)$	$\Theta\left(\lvert\text{out}(n)\rvert\right)$
Incoming edges of n	$\Theta\left(\lvert\mathcal{N}\rvert\right)$	$\Theta\left(\lvert\mathcal{N}\rvert\right)$	$\Theta\left(\lvert\mathcal{E}\rvert\right)$	$\Theta\left(\lvert\mathcal{N}\rvert^2\right)$
Outgoing edges of n	$\Theta\left(\lvert\mathcal{N}\rvert\right)$	$\Theta\left(\lvert\mathcal{N}\rvert\right)$	$\Theta\left(\lvert\text{out}(n)\rvert\right)$	$\Theta\left(\lvert\text{out}(n)\rvert\right)$
Weight of edge (n, m)	$\Theta(1)$	$\Theta(1)$	$\Theta\left(\lvert\text{out}(n)\rvert\right)$	$\Theta\left(\lvert\text{out}(n)\rvert\right)$

A comparison of representations

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.

Dense graph graph $\mathcal{G}$ is *dense* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|^2\right)$. $\rightarrow$ most node pairs are edges!

Sparse graph graph $\mathcal{G}$ is *spase* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|\right)$. $\rightarrow$ most node pairs are *not* edges!

Which representation is the best?

- Sparse graphs?
- Dense graphs?
- Small graphs of at-most 16 nodes?

A comparison of representations

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.

Dense graph graph $\mathcal{G}$ is *dense* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|^2\right)$. $\longrightarrow$ most node pairs are edges!

Sparse graph graph $\mathcal{G}$ is *spase* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|\right)$. $\longrightarrow$ most node pairs are *not* edges!

Which representation is the best?

- Sparse graphs? $\longrightarrow$ usually adjacency list.
- Dense graphs? $\longrightarrow$ usually matrix.
- Small graphs of at-most 16 nodes? $\longrightarrow$ likely matrix.

A comparison of representations

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.

Dense graph graph $\mathcal{G}$ is *dense* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|^2\right)$. $\longrightarrow$ most node pairs are edges!

Sparse graph graph $\mathcal{G}$ is *spase* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|\right)$. $\longrightarrow$ most node pairs are *not* edges!

Which representation is the best?

- Sparse graphs? $\longrightarrow$ usually adjacency list.
- Dense graphs? $\longrightarrow$ usually matrix.
- Small graphs of at-most 16 nodes? $\longrightarrow$ likely matrix.

Depends a lot on the type of operations.

E.g., graph operations in terms of *matrices* are easier to implement on GPUs.

A comparison of representations

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph.

Dense graph graph $\mathcal{G}$ is *dense* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|^2\right)$. $\longrightarrow$ most node pairs are edges!

Sparse graph graph $\mathcal{G}$ is *spase* if $|\mathcal{E}| = \Theta\left(|\mathcal{N}|\right)$. $\longrightarrow$ most node pairs are *not* edges!

Which representation is the best?

- Sparse graphs? $\longrightarrow$ usually adjacency list.
- Dense graphs? $\longrightarrow$ usually matrix.
- Small graphs of at-most 16 nodes? $\longrightarrow$ likely matrix.

Depends a lot on the type of operations.

E.g., graph operations in terms of *matrices* are easier to implement on GPUs.

Many alternatives exist

- Simply storing the set of *edges* (e.g., as a *relational table* in a database);
- *Compressed matrices* for GPU operations on sparse graphs (e.g., in machine learning);
- ....

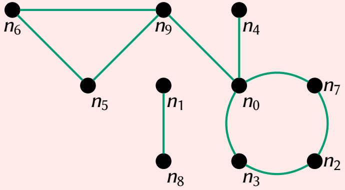
Traversing undirected graphs: Depth-first

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if ¬marked[m] then
        marked[m] := true.
        DFS-R(𝒢, marked, m).
```

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ 𝒩}.
DFS-R(𝒢, marked, s).
```



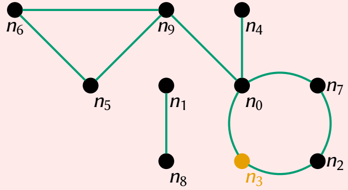
Traversing undirected graphs: Depth-first

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
        $marked[m] :=$ `true`.
        DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



Traversing undirected graphs: Depth-first

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

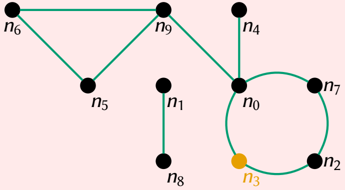
> **for all** $(n, m) \in \mathcal{E}$ **do**
>   **if** $\neg marked[m]$ **then**
> $marked[m] :=$ `true`.
>   DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

> $marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
> DFS-R($\mathcal{G}$, *marked*, s).

$marked =$

n_0	`false`
n_1	`false`
n_2	`false`
n_3	`true`
n_4	`false`
n_5	`false`
n_6	`false`
n_7	`false`
n_8	`false`
n_9	`false`



Traversing undirected graphs: Depth-first

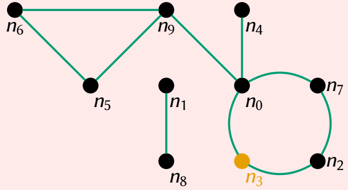
Called with $n = n_3$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if ¬marked[m] then
        marked[m] := true.
        DFS-R(𝒢, marked, m).
```

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ 𝒩}.
DFS-R(𝒢, marked, s).
```



marked =

n_0	false
n_1	false
n_2	false
n_3	true
n_4	false
n_5	false
n_6	false
n_7	false
n_8	false
n_9	false

Traversing undirected graphs: Depth-first

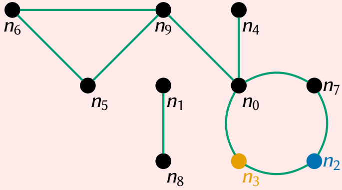
Called with $n = n_3$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $marked$, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
        $marked[m] := \texttt{true}$.
        DFS-R($\mathcal{G}$, $marked$, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, $marked$, s).



$marked =$

n_0	false
n_1	false
n_2	true
n_3	true
n_4	false
n_5	false
n_6	false
n_7	false
n_8	false
n_9	false

Traversing undirected graphs: Depth-first

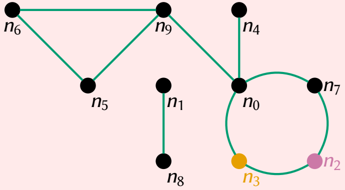
Called with $n = n_3$, n_2.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m]$:= true.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DEPTHFIRSTR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



marked =

n_0	false
n_1	false
n_2	true
n_3	true
n_4	false
n_5	false
n_6	false
n_7	false
n_8	false
n_9	false

Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $marked$, $n \in \mathcal{N}$):

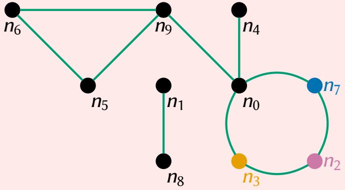
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] :=$ true.
    DFS-R($\mathcal{G}$, $marked$, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, $marked$, s).

$marked =$

n_0	false
n_1	false
n_2	true
n_3	true
n_4	false
n_5	false
n_6	false
n_7	true
n_8	false
n_9	false



Traversing undirected graphs: Depth-first

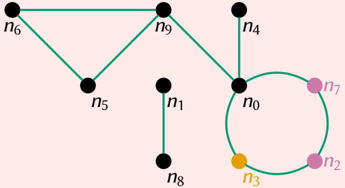
Called with $n = n_3, n_2, n_7$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] :=$ `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



marked =

n_0	`false`
n_1	`false`
n_2	`true`
n_3	`true`
n_4	`false`
n_5	`false`
n_6	`false`
n_7	`true`
n_8	`false`
n_9	`false`

Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $marked$, $n \in \mathcal{N}$):

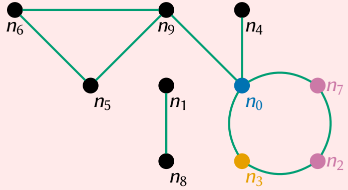
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m]$:= true.
    DFS-R($\mathcal{G}$, $marked$, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, $marked$, s).

$marked =$

n_0	true
n_1	false
n_2	true
n_3	true
n_4	false
n_5	false
n_6	false
n_7	true
n_8	false
n_9	false



Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7, n_0$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

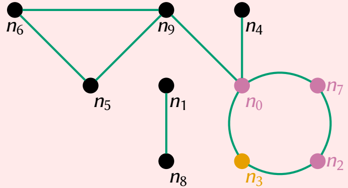
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] :=$ true.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

marked =

n_0	true
n_1	false
n_2	true
n_3	true
n_4	false
n_5	false
n_6	false
n_7	true
n_8	false
n_9	false



Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7, n_0$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

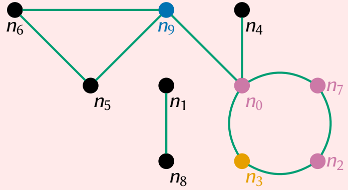
```
for all (n, m) ∈ E do
    if ¬marked[m] then
        marked[m] := true.
        DFS-R(G, marked, m).
```

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ N}.
DFS-R(G, marked, s).
```

$marked =$

n_0	true
n_1	false
n_2	true
n_3	true
n_4	false
n_5	false
n_6	false
n_7	true
n_8	false
n_9	true



Traversing undirected graphs: Depth-first

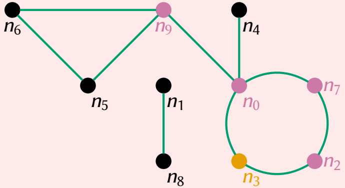
Called with $n = n_3, n_2, n_7, n_0, n_9$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] :=$ true.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



marked =

n_0	true
n_1	false
n_2	true
n_3	true
n_4	false
n_5	false
n_6	false
n_7	true
n_8	false
n_9	true

Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7, n_0, n_9$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

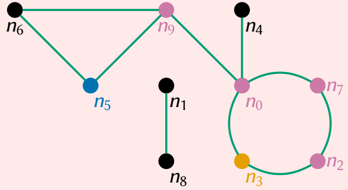
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] := \texttt{true}$.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

$marked =$

n_0	true
n_1	false
n_2	true
n_3	true
n_4	false
n_5	true
n_6	false
n_7	true
n_8	false
n_9	true



Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7, n_0, n_9, n_5$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

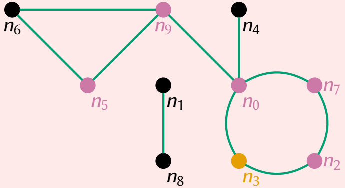
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] :=$ `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

marked =

n_0	true
n_1	false
n_2	true
n_3	true
n_4	false
n_5	true
n_6	false
n_7	true
n_8	false
n_9	true



Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7, n_0, n_9, n_5$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

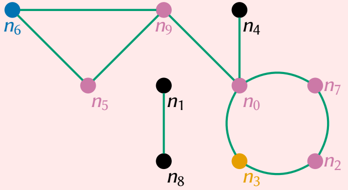
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] :=$ `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

marked =

n_0	`true`
n_1	`false`
n_2	`true`
n_3	`true`
n_4	`false`
n_5	`true`
n_6	`true`
n_7	`true`
n_8	`false`
n_9	`true`



Traversing undirected graphs: Depth-first

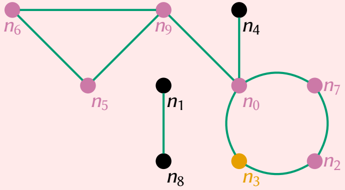
Called with $n = n_3, n_2, n_7, n_0, n_9, n_5, n_6$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m]$:= `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



marked =

n_0	`true`
n_1	`false`
n_2	`true`
n_3	`true`
n_4	`false`
n_5	`true`
n_6	`true`
n_7	`true`
n_8	`false`
n_9	`true`

Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7, n_0, n_9, n_5, n_6$.

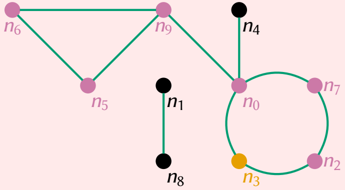
Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if ¬marked[m] then
        marked[m] := true.
        DFS-R(𝒢, marked, m).
```

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ 𝒩}.
DFS-R(𝒢, marked, s).
```

$$marked = \begin{array}{c|c|} n_0 & \texttt{true} \\ n_1 & \texttt{false} \\ n_2 & \texttt{true} \\ n_3 & \texttt{true} \\ n_4 & \texttt{false} \\ n_5 & \texttt{true} \\ n_6 & \texttt{true} \\ n_7 & \texttt{true} \\ n_8 & \texttt{false} \\ n_9 & \texttt{true} \end{array}$$



Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7, n_0, n_9, n_5$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

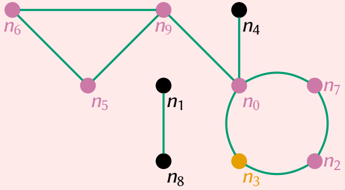
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m]$:= true.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

marked =

n_0	true
n_1	false
n_2	true
n_3	true
n_4	false
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true



Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7, n_0, n_9$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

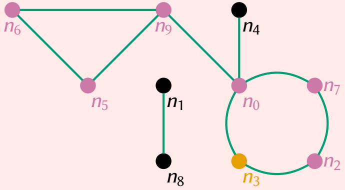
for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
        $marked[m]$:= `true`.
        DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

marked =

n_0	`true`
n_1	`false`
n_2	`true`
n_3	`true`
n_4	`false`
n_5	`true`
n_6	`true`
n_7	`true`
n_8	`false`
n_9	`true`



Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7, n_0$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

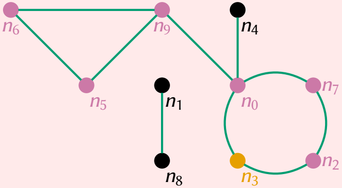
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] :=$ `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

$marked =$

n_0	true
n_1	false
n_2	true
n_3	true
n_4	false
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true



Traversing undirected graphs: Depth-first

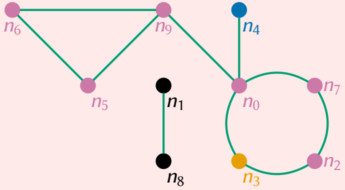
Called with $n = n_3, n_2, n_7, n_0, n_4$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m]$:= `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



marked =

n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true

Traversing undirected graphs: Depth-first

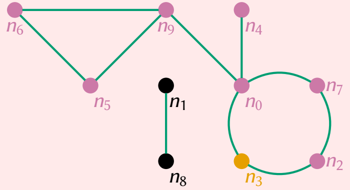
Called with $n = n_3, n_2, n_7, n_0, n_4$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg$*marked*$[m]$ **then**
    marked$[m]$:= true.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

marked := $\{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



marked =

n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true

Traversing undirected graphs: Depth-first

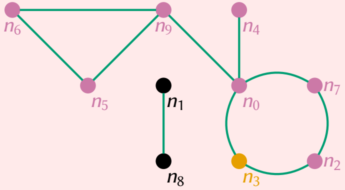
Called with $n = n_3, n_2, n_7, n_0$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if ¬marked[m] then
        marked[m] := true.
        DFS-R(𝒢, marked, m).
```

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ 𝒩}.
DFS-R(𝒢, marked, s).
```



$marked =$

n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true

Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2, n_7$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

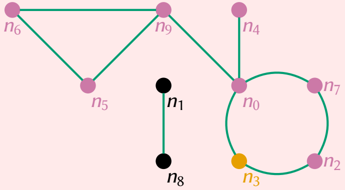
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] :=$ `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

marked =

n_0	`true`
n_1	`false`
n_2	`true`
n_3	`true`
n_4	`true`
n_5	`true`
n_6	`true`
n_7	`true`
n_8	`false`
n_9	`true`



Traversing undirected graphs: Depth-first

Called with $n = n_3, n_2$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

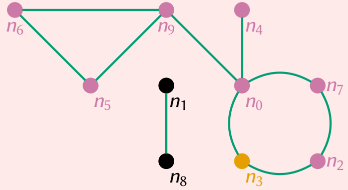
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] :=$ `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

marked =

n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true



Traversing undirected graphs: Depth-first

Called with $n = n_3$.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

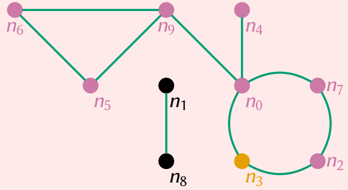
```
for all (n, m) ∈ ℰ do
    if ¬marked[m] then
        marked[m] := true.
        DFS-R(𝒢, marked, m).
```

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ 𝒩}.
DFS-R(𝒢, marked, s).
```

$marked =$

n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true



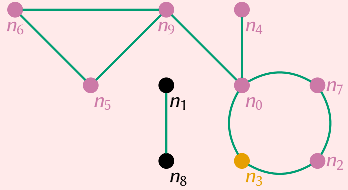
Traversing undirected graphs: Depth-first

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $marked$, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] :=$ `true`.
    DFS-R($\mathcal{G}$, $marked$, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

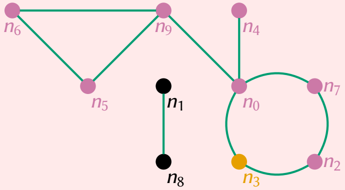
$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, $marked$, s).



$marked =$

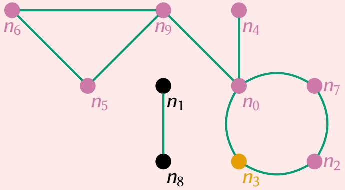
n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true

Traversing undirected graphs: Depth-first



What can we learn from this depth-first search?

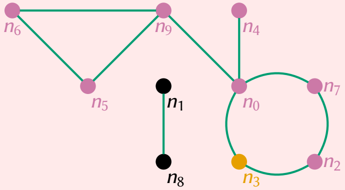
Traversing undirected graphs: Depth-first



What can we learn from this depth-first search?

- We found all nodes to which n_3 is *connected* (nodes one can reach from n_3).

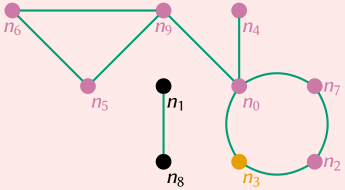
Traversing undirected graphs: Depth-first



What can we learn from this depth-first search?

- We found all nodes to which n_3 is *connected* (nodes one can reach from n_3).
- $\mathcal{G}$ is *not* a connected graph.

Traversing undirected graphs: Depth-first



What can we learn from this depth-first search?

- We found all nodes to which n_3 is *connected* (nodes one can reach from n_3).
- $\mathcal{G}$ is *not* a connected graph.
- The order of recursive calls was:

$$n = n_3, n_2, n_7, n_0, \begin{cases} n_9, n_5, n_6; \\ n_4. \end{cases}$$

This order provides a path from n_3 to *every* node it is connected to!

Traversing undirected graphs: Depth-first

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m] := \texttt{true}$.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

Complexity

Traversing undirected graphs: Depth-first

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $marked$, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if ¬marked[m] then
        marked[m] := true.
        DFS-R(𝒢, marked, m).
```

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ 𝒩}.
DFS-R(𝒢, marked, s).
```

Complexity

- We need $|\mathcal{N}|$ memory for *marked* and the at-most $|\mathcal{N}|$ recursive calls.
- We inspect each node once and traverse each edge once: $\Theta(|\mathcal{N}| + |\mathcal{E}|)$ (if we use the adjacency list representation).

Problem: Connected components

Problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

Provide an algorithm that can find all connected components in $\mathcal{G}$.

Problem: Connected components

Problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
Provide an algorithm that can find all connected components in $\mathcal{G}$.

Solution

Remark: $\textsc{DepthFirstR}(\mathcal{G}, n)$ will find all nodes in the connected component of n.

Problem: Connected components

Problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
Provide an algorithm that can find all connected components in $\mathcal{G}$.

Solution

Remark: DepthFirstR($\mathcal{G}$, n) will find all nodes in the connected component of n.

Algorithm DFS-CC-R($\mathcal{G}$, cc, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if cc[m] = unmarked then
        cc[m] := cc[n].
        DFS-CC-R(𝒢, cc, m).
```

Algorithm Components($\mathcal{G}$, $s \in \mathcal{N}$):

```
cc := {n ↦ unmarked}.
for all n ∈ 𝒩 do
    if cc[n] = unmarked then
        cc[n] := n.
        DFS-CC-R(𝒢, cc, n).
```

Problem: Connected components

Problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
Provide an algorithm that can find all connected components in $\mathcal{G}$.

Solution

Remark: DEPTHFIRSTR($\mathcal{G}$, n) will find all nodes in the connected component of n.

Algorithm DFS-CC-R($\mathcal{G}$, cc, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $cc[m] = \mathit{unmarked}$ **then**
    $cc[m] := cc[n]$.
    DFS-CC-R($\mathcal{G}$, cc, m).

Algorithm COMPONENTS($\mathcal{G}$, $s \in \mathcal{N}$):

$cc := \{n \mapsto \mathit{unmarked}\}$.
for all $n \in \mathcal{N}$ **do**
  if $cc[n] = \mathit{unmarked}$ **then**
    $cc[n] := n$.
    DFS-CC-R($\mathcal{G}$, cc, n).

We inspect each node once and traverse each edge once: $\Theta(|\mathcal{N}| + |\mathcal{E}|)$.

Problem: Two-colorability

Problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which:

- the nodes $\mathcal{N}$ represent competitors;
- the edges $\mathcal{E}$ represent rivalries.

Can we divide the nodes into two teams such that no rivals are in the same team?

Problem: Two-colorability

Problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which:

- the nodes $\mathcal{N}$ represent competitors;
- the edges $\mathcal{E}$ represent rivalries.

Can we divide the nodes into two teams such that no rivals are in the same team?

Definition

Graph $\mathcal{G}$ is *bipartite* if we can partition the nodes in two sets such that no two nodes in the same set share an edge.

Problem: Two-colorability

Problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which:

- the nodes $\mathcal{N}$ represent competitors;
- the edges $\mathcal{E}$ represent rivalries.

Can we divide the nodes into two teams such that no rivals are in the same team?

Definition

Graph $\mathcal{G}$ is *bipartite* if we can partition the nodes in two sets such that no two nodes in the same set share an edge.

The two-colorability problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. Find a coloring of the nodes $\mathcal{N}$ (if possible) using two colors such that nodes $(n, m) \in \mathcal{E}$ have different colors.

Problem: Two-colorability

The two-colorability problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. Find a coloring of the nodes $\mathcal{N}$ (if possible) using two colors such that nodes $(n, m) \in \mathcal{E}$ have different colors.

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
    if $colors[m] = 0$ **then**
        $colors[m] := -colors[n]$.
        DFS-TC-R($\mathcal{G}$, *colors*, m).
    else if $colors[m] = colors[n]$ **then**
        This graph is *not* bipartite.

Algorithm TwoColors($\mathcal{G}$):

$colors := \{n \mapsto 0 \mid n \in \mathcal{N}\}$.
for all $n \in \mathcal{N}$ **do**
    if $colors[n] = 0$ **then**
        $colors[n] := 1$.
        DFS-TC-R($\mathcal{G}$, *colors*, n).

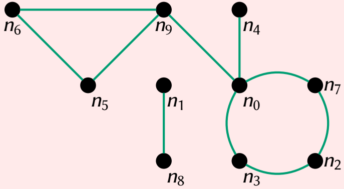
Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
    if $colors[m] = 0$ **then**
        $colors[m] := -colors[n]$.
        DFS-TC-R($\mathcal{G}$, *colors*, m).
    else if $colors[m] = colors[n]$ **then**
        This graph is *not* bipartite.

Algorithm TwoColors($\mathcal{G}$):

$colors := \{n \mapsto 0 \mid n \in \mathcal{N}\}$.
for all $n \in \mathcal{N}$ **do**
    if $colors[n] = 0$ **then**
        $colors[n] := 1$.
        DFS-TC-R($\mathcal{G}$, *colors*, n).



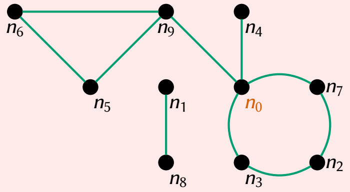
Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if colors[m] = 0 then
        colors[m] := −colors[n].
        DFS-TC-R(𝒢, colors, m).
    else if colors[m] = colors[n] then
        This graph is not bipartite.
```

Algorithm TwoColors($\mathcal{G}$):

```
colors := {n ↦ 0 | n ∈ 𝒩}.
for all n ∈ 𝒩 do
    if colors[n] = 0 then
        colors[n] := 1.
        DFS-TC-R(𝒢, colors, n).
```



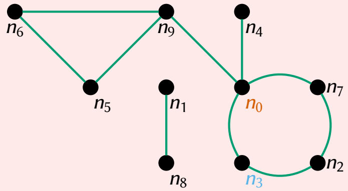
Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

- **for all** $(n, m) \in \mathcal{E}$ **do**
  - **if** $colors[m] = 0$ **then**
    - $colors[m] := -colors[n]$.
    - DFS-TC-R($\mathcal{G}$, *colors*, m).
  - **else if** $colors[m] = colors[n]$ **then**
    - This graph is *not* bipartite.

Algorithm TwoColors($\mathcal{G}$):

- $colors := \{n \mapsto 0 \mid n \in \mathcal{N}\}$.
- **for all** $n \in \mathcal{N}$ **do**
  - **if** $colors[n] = 0$ **then**
    - $colors[n] := 1$.
    - DFS-TC-R($\mathcal{G}$, *colors*, n).



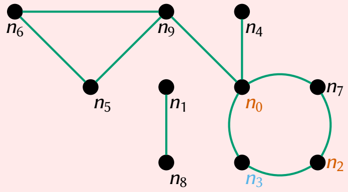
Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $colors[m] = 0$ **then**
    $colors[m] := -colors[n]$.
    DFS-TC-R($\mathcal{G}$, *colors*, m).
  else if $colors[m] = colors[n]$ **then**
    This graph is *not* bipartite.

Algorithm TwoColors($\mathcal{G}$):

$colors := \{n \mapsto 0 \mid n \in \mathcal{N}\}$.
for all $n \in \mathcal{N}$ **do**
  if $colors[n] = 0$ **then**
    $colors[n] := 1$.
    DFS-TC-R($\mathcal{G}$, *colors*, n).



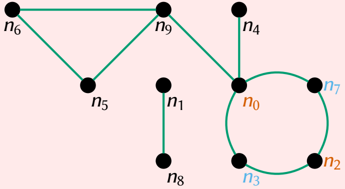
Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

- **for all** $(n, m) \in \mathcal{E}$ **do**
    - **if** $colors[m] = 0$ **then**
        - $colors[m] := -colors[n]$.
        - DFS-TC-R($\mathcal{G}$, *colors*, m).
    - **else if** $colors[m] = colors[n]$ **then**
        - This graph is *not* bipartite.

Algorithm TwoColors($\mathcal{G}$):

- $colors := \{n \mapsto 0 \mid n \in \mathcal{N}\}$.
- **for all** $n \in \mathcal{N}$ **do**
    - **if** $colors[n] = 0$ **then**
        - $colors[n] := 1$.
        - DFS-TC-R($\mathcal{G}$, *colors*, n).



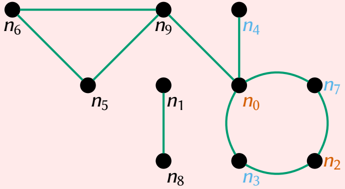
Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $colors[m] = 0$ **then**
    $colors[m] := -colors[n]$.
    DFS-TC-R($\mathcal{G}$, *colors*, m).
  else if $colors[m] = colors[n]$ **then**
    This graph is *not* bipartite.

Algorithm TwoColors($\mathcal{G}$):

$colors := \{n \mapsto 0 \mid n \in \mathcal{N}\}$.
for all $n \in \mathcal{N}$ **do**
  if $colors[n] = 0$ **then**
    $colors[n] := 1$.
    DFS-TC-R($\mathcal{G}$, *colors*, n).



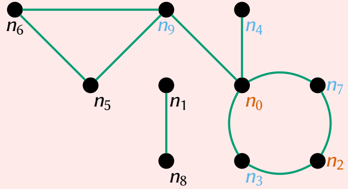
Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $colors[m] = 0$ **then**
    $colors[m] := -colors[n]$.
    DFS-TC-R($\mathcal{G}$, *colors*, m).
  else if $colors[m] = colors[n]$ **then**
    This graph is *not* bipartite.

Algorithm TwoColors($\mathcal{G}$):

$colors := \{n \mapsto 0 \mid n \in \mathcal{N}\}$.
for all $n \in \mathcal{N}$ **do**
  if $colors[n] = 0$ **then**
    $colors[n] := 1$.
    DFS-TC-R($\mathcal{G}$, *colors*, n).



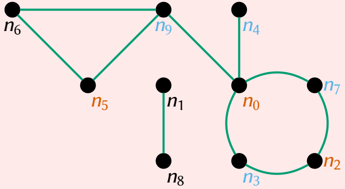
Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

- **for all** $(n, m) \in \mathcal{E}$ **do**
  - **if** $colors[m] = 0$ **then**
    - $colors[m] := -colors[n]$.
    - DFS-TC-R($\mathcal{G}$, *colors*, m).
  - **else if** $colors[m] = colors[n]$ **then**
    - This graph is *not* bipartite.

Algorithm TwoColors($\mathcal{G}$):

- $colors := \{n \mapsto 0 \mid n \in \mathcal{N}\}$.
- **for all** $n \in \mathcal{N}$ **do**
  - **if** $colors[n] = 0$ **then**
    - $colors[n] := 1$.
    - DFS-TC-R($\mathcal{G}$, *colors*, n).



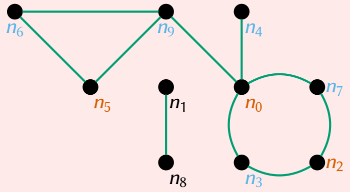
Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if colors[m] = 0 then
        colors[m] := −colors[n].
        DFS-TC-R(𝒢, colors, m).
    else if colors[m] = colors[n] then
        This graph is not bipartite.
```

Algorithm TwoColors($\mathcal{G}$):

```
colors := {n ↦ 0 | n ∈ 𝒩}.
for all n ∈ 𝒩 do
    if colors[n] = 0 then
        colors[n] := 1.
        DFS-TC-R(𝒢, colors, n).
```



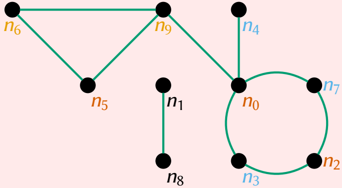
Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
    if $colors[m] = 0$ **then**
        $colors[m] := -colors[n]$.
        DFS-TC-R($\mathcal{G}$, *colors*, m).
    else if $colors[m] = colors[n]$ **then**
        This graph is *not* bipartite.

Algorithm TwoColors($\mathcal{G}$):

$colors := \{n \mapsto 0 \mid n \in \mathcal{N}\}$.
for all $n \in \mathcal{N}$ **do**
    if $colors[n] = 0$ **then**
        $colors[n] := 1$.
        DFS-TC-R($\mathcal{G}$, *colors*, n).



Problem: Two-colorability

Algorithm DFS-TC-R($\mathcal{G}$, *colors*, $n \in \mathcal{N}$):

- **for all** $(n, m) \in \mathcal{E}$ **do**
  - **if** $colors[m] = 0$ **then**
    - $colors[m] := -colors[n]$.
    - DFS-TC-R($\mathcal{G}$, *colors*, m).
  - **else if** $colors[m] = colors[n]$ **then**
    - This graph is *not* bipartite.

Algorithm TwoColors($\mathcal{G}$):

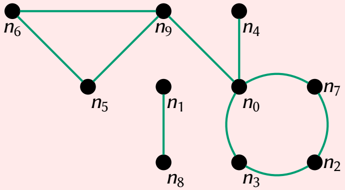
- $colors := \{n \mapsto 0 \mid n \in \mathcal{N}\}$.
- **for all** $n \in \mathcal{N}$ **do**
  - **if** $colors[n] = 0$ **then**
    - $colors[n] := 1$.
    - DFS-TC-R($\mathcal{G}$, *colors*, n).

We inspect each node once and traverse each edge once: $\Theta(|\mathcal{N}| + |\mathcal{E}|)$.

Traversing undirected graphs: Breadth-first

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E}), s \in \mathcal{N}$):

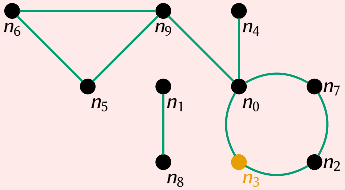
$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
Q := a queue holding only s.
while $\neg$Empty(Q) **do**
  $n :=$ Dequeue(Q).
  for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
      $marked[m] :=$ `true`.
      Enqueue(S, m).



Traversing undirected graphs: Breadth-first

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
Q := a queue holding only s.
while $\neg$Empty(Q) **do**
  $n :=$ Dequeue(Q).
  for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
      $marked[m] :=$ `true`.
      Enqueue(S, m).



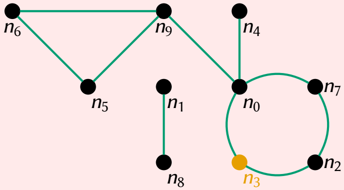
Traversing undirected graphs: Breadth-first

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
Q := a queue holding only s.
while $\neg$Empty(Q) **do**
  n := Dequeue(Q).
  for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
      $marked[m]$:= true.
      Enqueue(S, m).

$marked =$

n_0	false
n_1	false
n_2	false
n_3	true
n_4	false
n_5	false
n_6	false
n_7	false
n_8	false
n_9	false



Traversing undirected graphs: Breadth-first

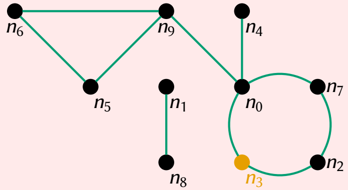
$Q : [n_3]$.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E}), s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ 𝒩}.
Q := a queue holding only s.
while ¬Empty(Q) do
    n := Dequeue(Q).
    for all (n, m) ∈ ℰ do
        if ¬marked[m] then
            marked[m] := true.
            Enqueue(S, m).
```

$marked =$

n_0	false
n_1	false
n_2	false
n_3	true
n_4	false
n_5	false
n_6	false
n_7	false
n_8	false
n_9	false



Traversing undirected graphs: Breadth-first

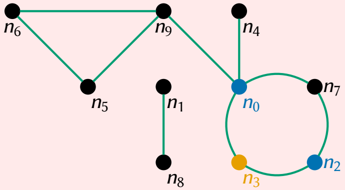
$Q : [n_0, n_2]$, $n = n_3$.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
Q := a queue holding only s.
while $\neg$EMPTY(Q) **do**
  $n :=$ DEQUEUE(Q).
  for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
      $marked[m] :=$ true.
      ENQUEUE(S, m).

$marked =$

n_0	true
n_1	false
n_2	true
n_3	true
n_4	false
n_5	false
n_6	false
n_7	false
n_8	false
n_9	false



Traversing undirected graphs: Breadth-first

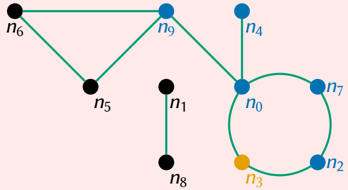
$Q : [n_2, n_7, n_4, n_9]$, $n = n_0$.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ N}.
Q := a queue holding only s.
while ¬EMPTY(Q) do
    n := DEQUEUE(Q).
    for all (n, m) ∈ E do
        if ¬marked[m] then
            marked[m] := true.
            ENQUEUE(S, m).
```

$marked =$

n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	false
n_6	false
n_7	true
n_8	false
n_9	true



Traversing undirected graphs: Breadth-first

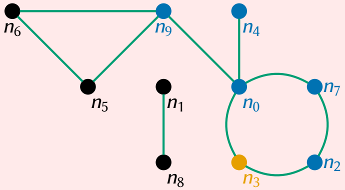
$Q : [n_7, n_4, n_9]$, $n = n_2$.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
Q := a queue holding only s.
while $\neg$Empty(Q) **do**
  $n :=$ Dequeue(Q).
  for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
      $marked[m] :=$ `true`.
      Enqueue(S, m).

$marked =$

n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	false
n_6	false
n_7	true
n_8	false
n_9	true

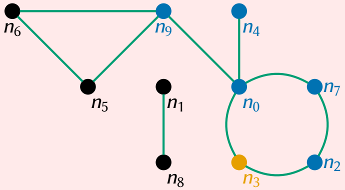


Traversing undirected graphs: Breadth-first

$Q : [n_4, n_9]$, $n = n_7$.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ N}.
Q := a queue holding only s.
while ¬EMPTY(Q) do
    n := DEQUEUE(Q).
    for all (n, m) ∈ E do
        if ¬marked[m] then
            marked[m] := true.
            ENQUEUE(S, m).
```



marked =

n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	false
n_6	false
n_7	true
n_8	false
n_9	true

Traversing undirected graphs: Breadth-first

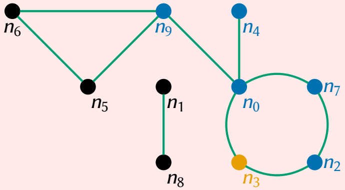
$Q : [n_9]$, $n = n_4$.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E}), s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
$Q :=$ a queue holding only s.
while $\neg$EMPTY(Q) **do**
    $n :=$ DEQUEUE(Q).
    for all $(n, m) \in \mathcal{E}$ **do**
        if $\neg marked[m]$ **then**
            $marked[m] :=$ true.
            ENQUEUE(S, m).

$marked =$

n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	false
n_6	false
n_7	true
n_8	false
n_9	true

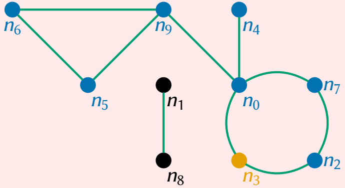


Traversing undirected graphs: Breadth-first

$Q : [n_6, n_5], n = n_9.$

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E}), s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
$Q :=$ a queue holding only s.
while $\neg$EMPTY(Q) **do**
  $n :=$ DEQUEUE(Q).
  for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
      $marked[m] :=$ true.
      ENQUEUE(S, m).



$marked =$

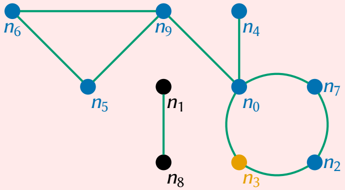
n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true

Traversing undirected graphs: Breadth-first

$Q : [n_5]$, $n = n_6$.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ 𝒩}.
Q := a queue holding only s.
while ¬Empty(Q) do
    n := Dequeue(Q).
    for all (n, m) ∈ ℰ do
        if ¬marked[m] then
            marked[m] := true.
            Enqueue(S, m).
```



marked =

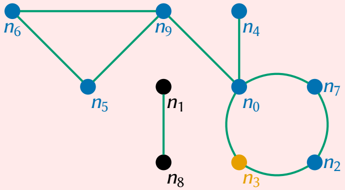
n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true

Traversing undirected graphs: Breadth-first

$Q : [], n = n_5.$

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E}), s \in \mathcal{N}$):

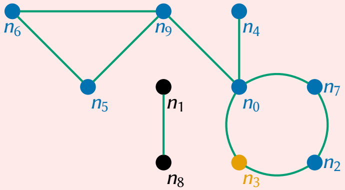
```
marked := {n ↦ (n ≠ s) | n ∈ N}.
Q := a queue holding only s.
while ¬Empty(Q) do
    n := Dequeue(Q).
    for all (n, m) ∈ E do
        if ¬marked[m] then
            marked[m] := true.
            Enqueue(S, m).
```



$marked =$

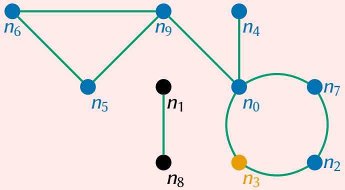
n_0	true
n_1	false
n_2	true
n_3	true
n_4	true
n_5	true
n_6	true
n_7	true
n_8	false
n_9	true

Traversing undirected graphs: Breadth-first



What can we learn from this breadth-first search?

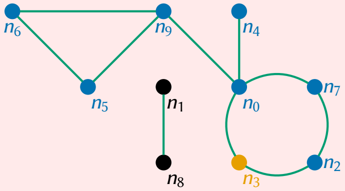
Traversing undirected graphs: Breadth-first



What can we learn from this breadth-first search?

- We found all nodes to which n_3 is *connected* (nodes one can reach from n_3).

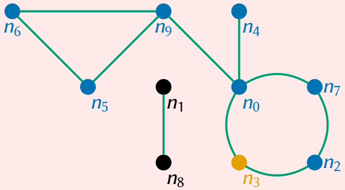
Traversing undirected graphs: Breadth-first



What can we learn from this breadth-first search?

- We found all nodes to which n_3 is *connected* (nodes one can reach from n_3).
- $\mathcal{G}$ is *not* a connected graph.

Traversing undirected graphs: Breadth-first

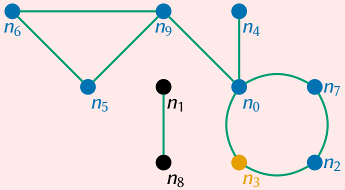


What can we learn from this breadth-first search?

- We found all nodes to which n_3 is *connected* (nodes one can reach from n_3).
- $\mathcal{G}$ is *not* a connected graph.

Breadth-first search is *similar* to depth-first search!

Traversing undirected graphs: Breadth-first



Complexity

- We need $|\mathcal{N}|$ memory for *marked*.
- We inspect each node once and traverse each edge once: $\Theta(|\mathcal{N}| + |\mathcal{E}|)$ (if we use the adjacency list representation).

Problem: Single-source shortest path

Problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ *without weight* and node $s \in \mathcal{N}$, find a shortest path from node s to all nodes s can reach.

Problem: Single-source shortest path

Problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ *without weight* and node $s \in \mathcal{N}$, find a shortest path from node s to all nodes s can reach.

Observe

Breadth-first search visits nodes on increasing distance to s.
First: all nodes at distance 1, then all nodes at distance 2, ...

Problem: Single-source shortest path

Problem

Given an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ *without weight* and node $s \in \mathcal{N}$, find a shortest path from node s to all nodes s can reach.

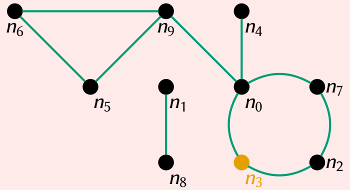
Algorithm BFS-SSSP($\mathcal{G}$, $s \in \mathcal{N}$):

$\mathit{distance} := \{n \mapsto \infty \mid n \in \mathcal{N}\}$.
$\mathit{distance}[s] := 0$.
Q := a queue holding only s.
while $\neg\textsc{Empty}(Q)$ **do**
    $n := \textsc{Dequeue}(Q)$.
    for all $(n, m) \in \mathcal{E}$ **do**
        if $\mathit{distance}[m] = \infty$ **then**
            $\mathit{distance}[m] := \mathit{distance}[n] + 1$.
            $\textsc{Enqueue}(Q, m)$.

Problem: Single-source shortest path

Algorithm BFS-SSSP($\mathcal{G}$, $s \in \mathcal{N}$):

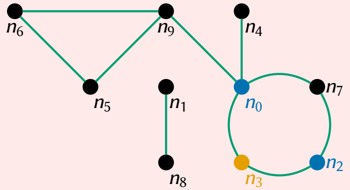
$distance := \{n \mapsto \infty \mid n \in \mathcal{N}\}$.
$distance[s] := 0$.
Q := a queue holding only s.
while $\neg$Empty(Q) **do**
    $n :=$ Dequeue(Q).
    for all $(n, m) \in \mathcal{E}$ **do**
        if $distance[m] = \infty$ **then**
            $distance[m] := distance[n] + 1$.
            Enqueue(Q, m).



Problem: Single-source shortest path

Algorithm BFS-SSSP($\mathcal{G}$, $s \in \mathcal{N}$):

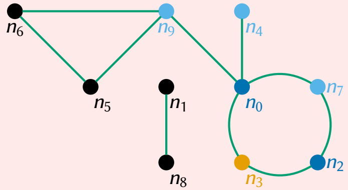
$distance := \{n \mapsto \infty \mid n \in \mathcal{N}\}$.
$distance[s] := 0$.
Q := a queue holding only s.
while $\neg$Empty(Q) **do**
    $n :=$ Dequeue(Q).
    for all $(n, m) \in \mathcal{E}$ **do**
        if $distance[m] = \infty$ **then**
            $distance[m] := distance[n] + 1$.
            Enqueue(Q, m).



Problem: Single-source shortest path

Algorithm BFS-SSSP($\mathcal{G}$, $s \in \mathcal{N}$):

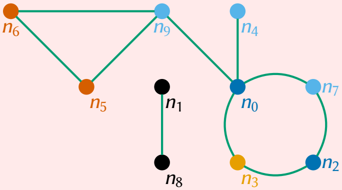
$\mathit{distance} := \{n \mapsto \infty \mid n \in \mathcal{N}\}$.
$\mathit{distance}[s] := 0$.
Q := a queue holding only s.
while $\neg$Empty(Q) **do**
    $n :=$ Dequeue(Q).
    for all $(n, m) \in \mathcal{E}$ **do**
        if $\mathit{distance}[m] = \infty$ **then**
            $\mathit{distance}[m] := \mathit{distance}[n] + 1$.
            Enqueue(Q, m).



Problem: Single-source shortest path

Algorithm BFS-SSSP($\mathcal{G}$, $s \in \mathcal{N}$):

$distance := \{n \mapsto \infty \mid n \in \mathcal{N}\}$.
$distance[s] := 0$.
$Q :=$ a queue holding only s.
while $\neg$Empty(Q) **do**
  $n :=$ Dequeue(Q).
  for all $(n, m) \in \mathcal{E}$ **do**
    if $distance[m] = \infty$ **then**
      $distance[m] := distance[n] + 1$.
      Enqueue(Q, m).



Problem: Single-source shortest path

Algorithm BFS-SSSP($\mathcal{G}$, $s \in \mathcal{N}$):

$distance := \{n \mapsto \infty \mid n \in \mathcal{N}\}$.

$distance[s] := 0$.

Q := a queue holding only s.

while $\neg$Empty(Q) **do**

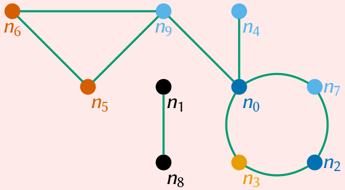
  $n :=$ Dequeue(Q).

  for all $(n, m) \in \mathcal{E}$ **do**

    if $distance[m] = \infty$ **then**

      $distance[m] := distance[n] + 1$.   *include (n, m) in the shortest path.*

      Enqueue(Q, m).



Problem: Single-source shortest path

Algorithm BFS-SSSP($\mathcal{G}$, $s \in \mathcal{N}$):

$distance := \{n \mapsto \infty \mid n \in \mathcal{N}\}$.
$distance[s] := 0$.
$Q :=$ a queue holding only s.
while $\neg$Empty(Q) **do**
    $n :=$ Dequeue(Q).
    for all $(n, m) \in \mathcal{E}$ **do**
        if $distance[m] = \infty$ **then**
            $distance[m] := distance[n] + 1$.   *include (n, m) in the shortest path.*
            Enqueue(Q, m).

We inspect each node once and traverse each edge once: $\Theta(|\mathcal{N}| + |\mathcal{E}|)$.

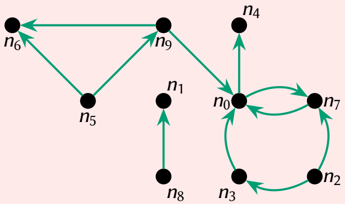
Traversing directed graphs: Depth-first

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
        $marked[m]$:= `true`.
        DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



Traversing directed graphs: Depth-first

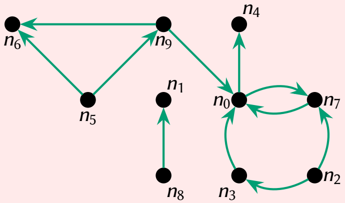
Same algorithm as for *undirected* graphs.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if ¬marked[m] then
        marked[m] := true.
        DFS-R(𝒢, marked, m).
```

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ 𝒩}.
DFS-R(𝒢, marked, s).
```



Traversing directed graphs: Depth-first

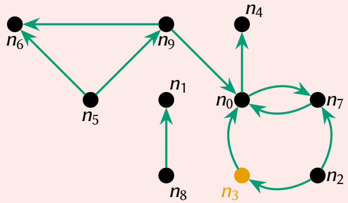
Same algorithm as for *undirected* graphs.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg$*marked*$[m]$ **then**
    marked$[m]$:= `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

marked := $\{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



Traversing directed graphs: Depth-first

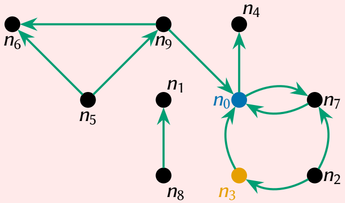
Same algorithm as for *undirected* graphs.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
        $marked[m]$:= `true`.
        DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



Traversing directed graphs: Depth-first

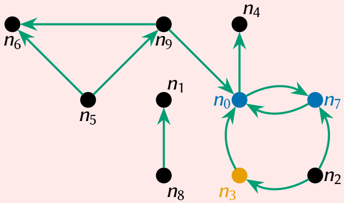
Same algorithm as for *undirected* graphs.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m]$:= `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).



Traversing directed graphs: Depth-first

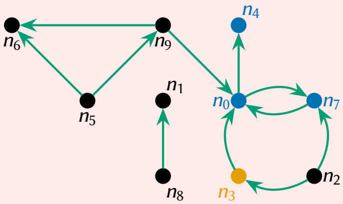
Same algorithm as for *undirected* graphs.

Algorithm DFS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$):

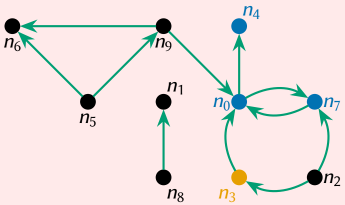
for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg marked[m]$ **then**
    $marked[m]$:= `true`.
    DFS-R($\mathcal{G}$, *marked*, m).

Algorithm DepthFirstR($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
DFS-R($\mathcal{G}$, *marked*, s).

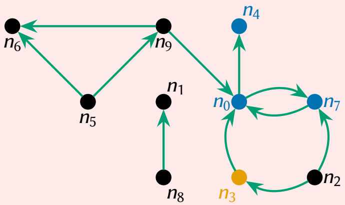


Traversing directed graphs: Depth-first



What can we learn from this depth-first search?

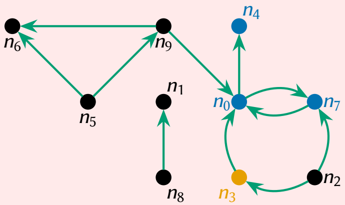
Traversing directed graphs: Depth-first



What can we learn from this depth-first search?

- We found all nodes to which n_3 is *strongly connected* (nodes one can reach from n_3).

Traversing directed graphs: Depth-first



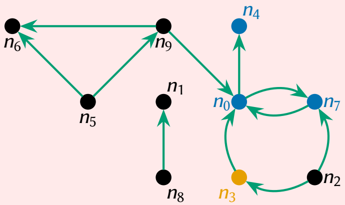
What can we learn from this depth-first search?

- We found all nodes to which n_3 is *strongly connected* (nodes one can reach from n_3).
- The order of recursive calls was:

$$n = n_3, n_0, \begin{cases} n_7; \\ n_4. \end{cases}$$

This order provides a path from n_3 to *every* node it is strongly connected to!

Traversing directed graphs: Depth-first



What can we learn from this depth-first search?

- We found all nodes to which n_3 is *strongly connected* (nodes one can reach from n_3).
- The order of recursive calls was:

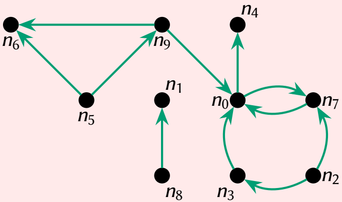
$$n = n_3, n_0, \begin{cases} n_7; \\ n_4. \end{cases}$$

  This order provides a path from n_3 to *every* node it is strongly connected to!
- Depth-first search does *not* tell us whether a graph is strongly connected!

Traversing directed graphs: Breadth-first

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E}), s \in \mathcal{N}$):

```
marked := {n ↦ (n ≠ s) | n ∈ 𝒩}.
Q := a queue holding only s.
while ¬Empty(Q) do
    n := Dequeue(Q).
    for all (n, m) ∈ ℰ do
        if ¬marked[m] then
            marked[m] := true.
            Enqueue(S, m).
```

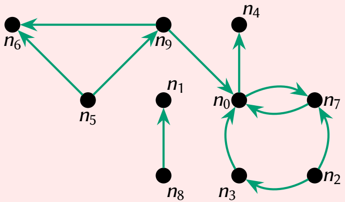


Traversing directed graphs: Breadth-first

Same algorithm as for *undirected* graphs.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$\mathit{marked} := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
$Q :=$ a queue holding only s.
while $\neg$Empty(Q) **do**
  $n :=$ Dequeue(Q).
  for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg \mathit{marked}[m]$ **then**
      $\mathit{marked}[m] :=$ `true`.
      Enqueue(S, m).

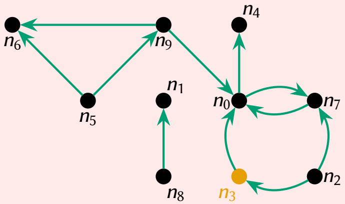


Traversing directed graphs: Breadth-first

Same algorithm as for *undirected* graphs.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
$Q :=$ a queue holding only s.
while $\neg$Empty(Q) **do**
    $n :=$ Dequeue(Q).
    for all $(n, m) \in \mathcal{E}$ **do**
        if $\neg marked[m]$ **then**
            $marked[m]$:= `true`.
            Enqueue(S, m).

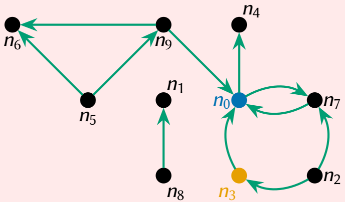


Traversing directed graphs: Breadth-first

Same algorithm as for *undirected* graphs.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
Q := a queue holding only s.
while $\neg$Empty(Q) **do**
  $n :=$ Dequeue(Q).
  for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
      $marked[m] :=$ `true`.
      Enqueue(S, m).

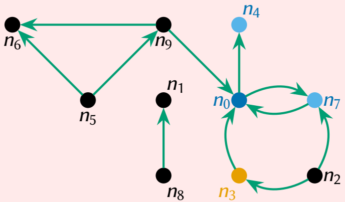


Traversing directed graphs: Breadth-first

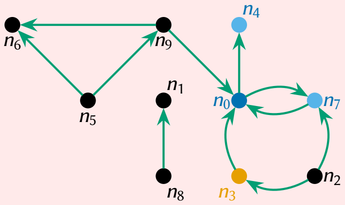
Same algorithm as for *undirected* graphs.

Algorithm BFS($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$):

$marked := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
Q := a queue holding only s.
while $\neg$Empty(Q) **do**
  $n :=$ Dequeue(Q).
  for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg marked[m]$ **then**
      $marked[m]$:= `true`.
      Enqueue(S, m).

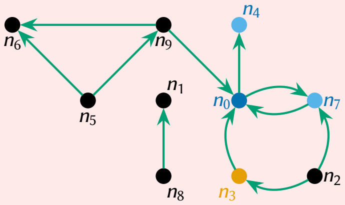


Traversing directed graphs: Breadth-first



What can we learn from this breadth-first search?

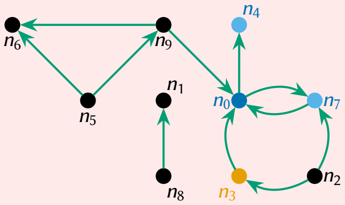
Traversing directed graphs: Breadth-first



What can we learn from this breadth-first search?

- We found all nodes to which n_3 is *strongly connected* (nodes one can reach from n_3).

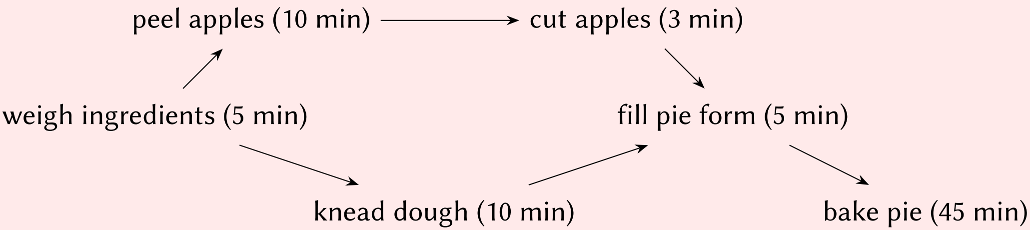
Traversing directed graphs: Breadth-first



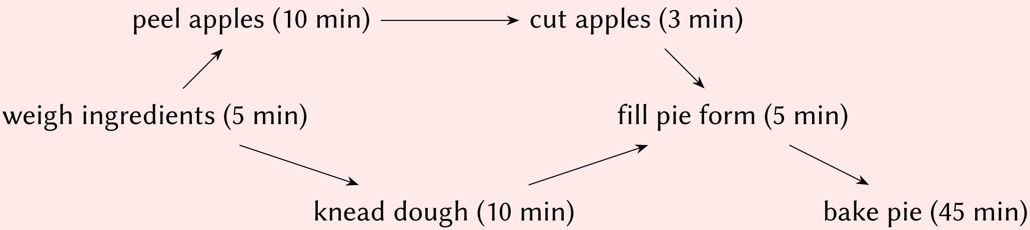
What can we learn from this breadth-first search?

- We found all nodes to which n_3 is *strongly connected* (nodes one can reach from n_3).
- We can also easily find shortest directed paths node n_3 can reach.

Problem: Cyclic dependencies



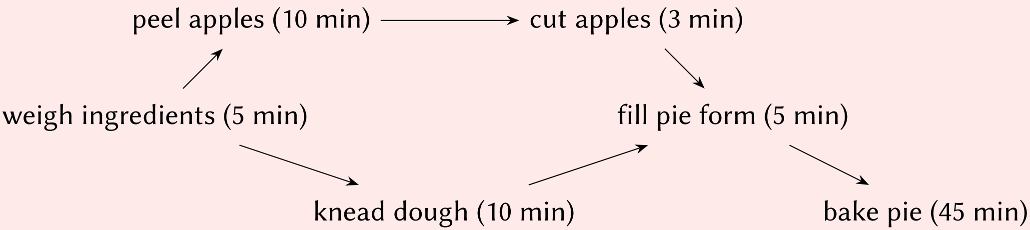
Problem: Cyclic dependencies



Problem

We can only satisfy the schedule if the graph of dependencies is *acyclic*:
Acylcic graph: there are *no* directed cycles.

Problem: Cyclic dependencies



Problem

We can only satisfy the schedule if the graph of dependencies is *acyclic*:
Acylcic graph: there are *no* directed cycles.

There is no path with at-least one edge from a node n to itself.

Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

Assume node m is the fist node visited during depth-first search with a path to itself.

Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

Assume node m is the fist node visited during depth-first search with a path to itself.

- The traversal started at node s and we visited the path $sn_1 \dots n_i m$ to reach m.

Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

Assume node m is the fist node visited during depth-first search with a path to itself.

- The traversal started at node s and we visited the path $sn_1 \ldots n_i m$ to reach m.
- m cannot reach any of $sn_1 \ldots n_i$: m is the *first* node on a cycle.

Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

Assume node m is the fist node visited during depth-first search with a path to itself.

- The traversal started at node s and we visited the path $sn_1 \dots n_i m$ to reach m.
- m cannot reach any of $sn_1 \dots n_i$: m is the *first* node on a cycle.
- From m, we will visit a path $mn'_1 \dots n'_j w$ to some node w such that node w has an edge to node n. *Why?*

Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

Assume node m is the fist node visited during depth-first search with a path to itself.

- The traversal started at node s and we visited the path $sn_1 \ldots n_i m$ to reach m.
- m cannot reach any of $sn_1 \ldots n_i$: m is the *first* node on a cycle.
- From m, we will visit a path $mn'_1 \ldots n'_j w$ to some node w such that node w has an edge to node n. *Why?*
  - Node m can reach itself as *it is on a cycle*.
    Hence, if we *started* at node m, we will eventually find node m.

Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

Assume node m is the fist node visited during depth-first search with a path to itself.

- The traversal started at node s and we visited the path $sn_1 \dots n_i m$ to reach m.
- m cannot reach any of $sn_1 \dots n_i$: m is the *first* node on a cycle.
- From m, we will visit a path $mn'_1 \dots n'_j w$ to some node w such that node w has an edge to node n. *Why?*
  - Node m can reach itself as *it is on a cycle*. Hence, if we *started* at node m, we will eventually find node m.
  - We could have started at a node $s \neq m$, however. But: the nodes $sn_1 \dots n_i$ are not part of a cycle. Hence, m cannot reach them!

Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
Assume node m is the fist node visited during depth-first search with a path to itself.

- The traversal started at node s and we visited the path $sn_1 \dots n_i m$ to reach m.
- m cannot reach any of $sn_1 \dots n_i$: m is the *first* node on a cycle.
- From m, we will visit a path $mn'_1 \dots n'_j w$ to some node w such that node w has an edge to node n. *Why?*
  - Node m can reach itself as *it is on a cycle*.
    Hence, if we *started* at node m, we will eventually find node m.
  - We could have started at a node $s \neq m$, however.
    But: the nodes $sn_1 \dots n_i$ are not part of a cycle. Hence, m cannot reach them!

Conclusion. Depth-first search can find cycles:
We simply have to detect nodes that reach themselves!

Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

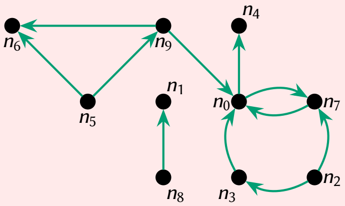
Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

Algorithm DFS-C-R($\mathcal{G}$, *marked*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ E do
    if marked[m] = unmarked then
        marked[m] := inspecting.
        DFS-C-R(G, marked, m).
    else if marked[m] = inspecting then
        Found a path that contains a cycle.
marked[m] := inspected.
```

Algorithm DFS-Cycle($\mathcal{G}$):

```
marked := {n ↦ unmarked | n ∈ N}.
for all n ∈ N do
    if marked[n] = unmarked then
        marked[n] := inspecting.
        DFS-C-R(G, marked, n).
```



Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

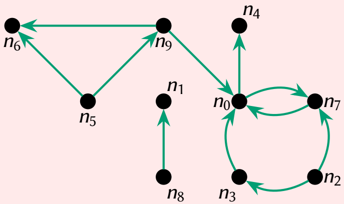
Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

Algorithm DFS-C-R($\mathcal{G}$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $marked[m] = unmarked$ **then**
    $marked[m] := inspecting$.
    DFS-C-R($\mathcal{G}$, *marked*, m).
  else if $marked[m] = inspecting$ **then**
    Found a path that contains a cycle.
$marked[m] := inspected$.

Algorithm DFS-Cycle($\mathcal{G}$):

$marked := \{n \mapsto unmarked \mid n \in \mathcal{N}\}$.
for all $n \in \mathcal{N}$ **do**
  if $marked[n] = unmarked$ **then**
    $marked[n] := inspecting$.
    DFS-C-R($\mathcal{G}$, *marked*, n).



Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

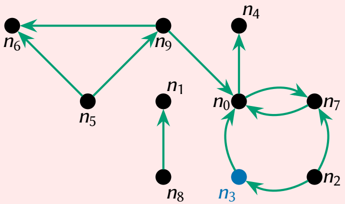
Called with $n = n_3$.

Algorithm DFS-C-R($\mathcal{G}$, *marked*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if marked[m] = unmarked then
        marked[m] := inspecting.
        DFS-C-R(𝒢, marked, m).
    else if marked[m] = inspecting then
        Found a path that contains a cycle.
marked[m] := inspected.
```

Algorithm DFS-CYCLE($\mathcal{G}$):

```
marked := {n ↦ unmarked | n ∈ 𝒩}.
for all n ∈ 𝒩 do
    if marked[n] = unmarked then
        marked[n] := inspecting.
        DFS-C-R(𝒢, marked, n).
```



Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

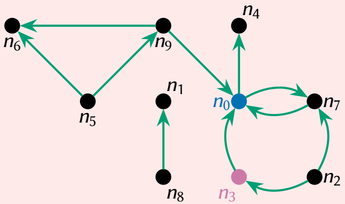
Called with $n = n_3$, n_0.

Algorithm DFS-C-R($\mathcal{G}$, *marked*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ ℰ do
    if marked[m] = unmarked then
        marked[m] := inspecting.
        DFS-C-R(𝒢, marked, m).
    else if marked[m] = inspecting then
        Found a path that contains a cycle.
marked[m] := inspected.
```

Algorithm DFS-CYCLE($\mathcal{G}$):

```
marked := {n ↦ unmarked | n ∈ 𝒩}.
for all n ∈ 𝒩 do
    if marked[n] = unmarked then
        marked[n] := inspecting.
        DFS-C-R(𝒢, marked, n).
```



Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

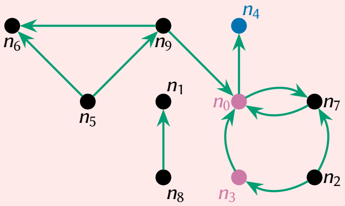
Called with $n = n_3, n_0, n_4$.

Algorithm DFS-C-R($\mathcal{G}$, *marked*, $n \in \mathcal{N}$):

```
for all (n, m) ∈ E do
    if marked[m] = unmarked then
        marked[m] := inspecting.
        DFS-C-R(G, marked, m).
    else if marked[m] = inspecting then
        Found a path that contains a cycle.
marked[m] := inspected.
```

Algorithm DFS-Cycle($\mathcal{G}$):

```
marked := {n ↦ unmarked | n ∈ N}.
for all n ∈ N do
    if marked[n] = unmarked then
        marked[n] := inspecting.
        DFS-C-R(G, marked, n).
```



Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

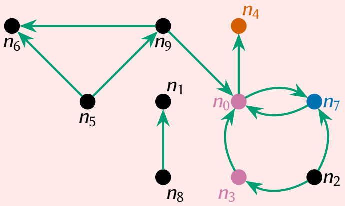
Called with $n = n_3, n_0, n_7$.

Algorithm DFS-C-R($\mathcal{G}$, *marked*, $n \in \mathcal{N}$):

for all $(n, m) \in \mathcal{E}$ **do**
  if $marked[m] = unmarked$ **then**
    $marked[m] := inspecting$.
    DFS-C-R($\mathcal{G}$, *marked*, m).
  else if $marked[m] = inspecting$ **then**
    Found a path that contains a cycle.
$marked[m] := inspected$.

Algorithm DFS-Cycle($\mathcal{G}$):

$marked := \{n \mapsto unmarked \mid n \in \mathcal{N}\}$.
for all $n \in \mathcal{N}$ **do**
  if $marked[n] = unmarked$ **then**
    $marked[n] := inspecting$.
    DFS-C-R($\mathcal{G}$, *marked*, n).



Problem: Cyclic dependencies

Find a *directed* cycle: a path from a node to itself

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

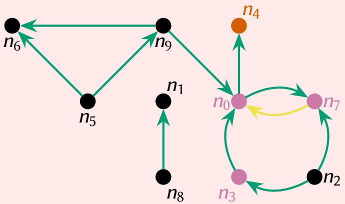
Called with $n = n_3$, n_0, n_7.

Algorithm DFS-C-R($\mathcal{G}$, *marked*, $n \in \mathcal{N}$):

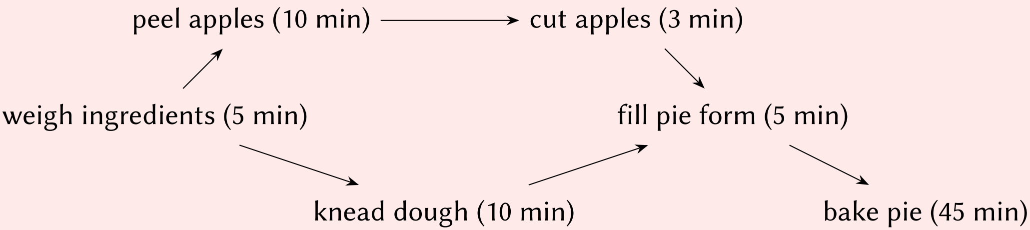
for all $(n, m) \in \mathcal{E}$ **do**
  if $marked[m] = unmarked$ **then**
    $marked[m] := inspecting$.
    DFS-C-R($\mathcal{G}$, *marked*, m).
  else if $marked[m] = inspecting$ **then**
    Found a path that contains a cycle.
$marked[m] := inspected$.

Algorithm DFS-Cycle($\mathcal{G}$):

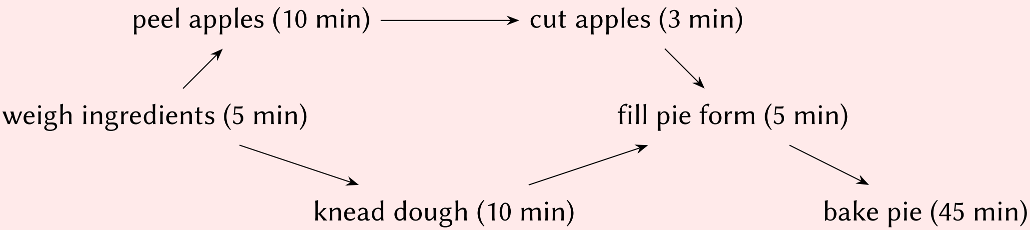
$marked := \{n \mapsto unmarked \mid n \in \mathcal{N}\}$.
for all $n \in \mathcal{N}$ **do**
  if $marked[n] = unmarked$ **then**
    $marked[n] := inspecting$.
    DFS-C-R($\mathcal{G}$, *marked*, n).



Problem: Ordering tasks



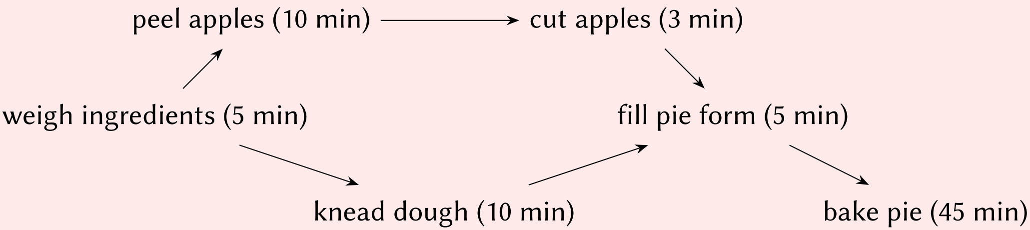
Problem: Ordering tasks



Problem

To schedule the tasks, we need to order tasks based on their dependencies.

Problem: Ordering tasks

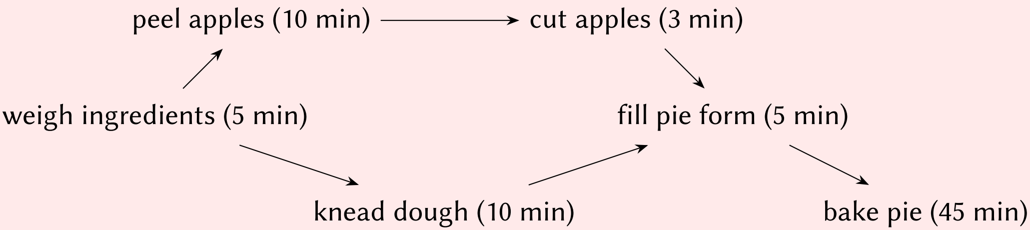


Problem

To schedule the tasks, we need to order tasks based on their dependencies.

Topological order: an order on nodes such that,
for every directed edge (m, n), m is ordered before n.

Problem: Ordering tasks



Problem

To schedule the tasks, we need to order tasks based on their dependencies.

Topological order: an order on nodes such that,
for every directed edge (m, n), m is ordered before n.

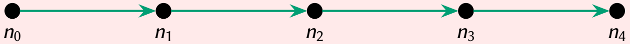
We *cannot* have a topological order if the graph is cyclic.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.

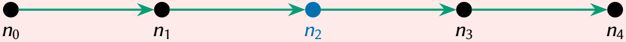


Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



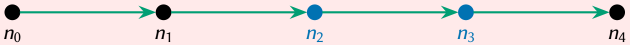
We inspect the nodes in the order: n_2.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



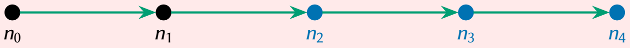
We inspect the nodes in the order: n_2, n_3.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m, then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



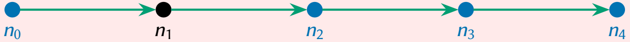
We inspect the nodes in the order: n_2, n_3, n_4.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



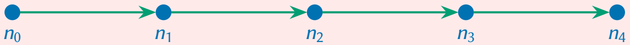
We inspect the nodes in the order: n_2, n_3, n_4, n_0.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



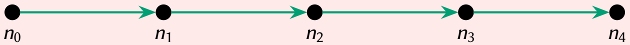
We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

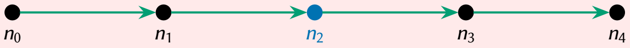
We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

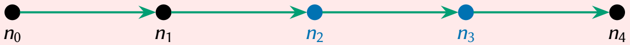
We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

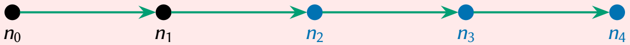
We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

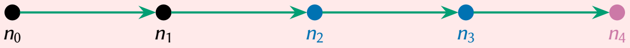
We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

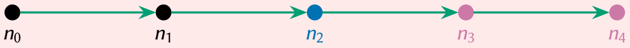
We finish inspecting the nodes in the order n_4.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

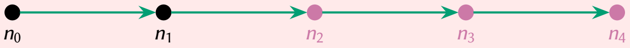
We finish inspecting the nodes in the order n_4, n_3.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

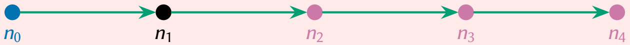
We finish inspecting the nodes in the order n_4, n_3, n_2.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

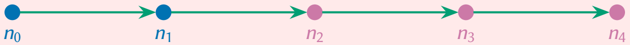
We finish inspecting the nodes in the order n_4, n_3, n_2.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

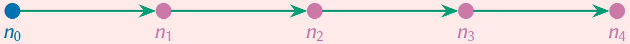
We finish inspecting the nodes in the order n_4, n_3, n_2.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

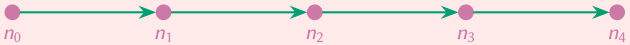
We finish inspecting the nodes in the order n_4, n_3, n_2, n_1.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

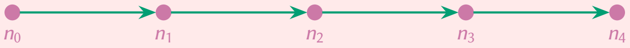
We finish inspecting the nodes in the order n_4, n_3, n_2, n_1, n_0.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

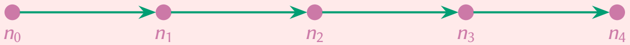
We finish inspecting the nodes in the order n_4, n_3, n_2, n_1, n_0.
This ordering is in reverse: we added to the *back* in this example.

Problem: Ordering tasks

Determine a *topological order*

Depth-first search seems related: if we reach node n after inspecting m,
then m should definitely come before n in the order.

Consider first starting depth-first search at n_2, and then starting at n_0.



We inspect the nodes in the order: n_2, n_3, n_4, n_0, n_1.

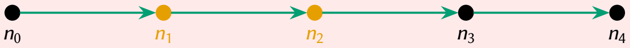
We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

We finish inspecting the nodes in the order n_4, n_3, n_2, n_1, n_0.
This ordering is in reverse: we added to the *back* in this example.

We need to prove that this is correct!

Problem: Ordering tasks

Determine a *topological order*



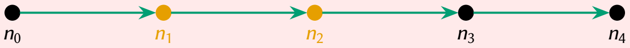
We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

Theorem

Let $(m, n) \in \mathcal{E}$ be an edge in an acyclic graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
Any depth-first search on $\mathcal{G}$ will finish inspecting n before m (hence, m is placed before n in our order).

Problem: Ordering tasks

Determine a *topological order*



We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

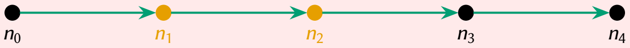
Theorem
*Let $(m, n) \in \mathcal{E}$ be an edge in an acyclic graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
Any depth-first search on $\mathcal{G}$ will finish inspecting n before m.*

Proof. We consider two cases:

- *When we run depth-first search for m, n is already marked.*
- *When we run depth-first search for m, n is not yet marked.*

Problem: Ordering tasks

Determine a *topological order*



We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

Theorem

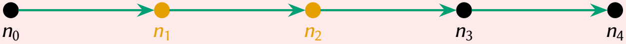
Let $(m, n) \in \mathcal{E}$ be an edge in an acyclic graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
Any depth-first search on $\mathcal{G}$ will finish inspecting n before m.

Proof. We consider two cases:

- *When we run depth-first search for m, n is already marked*.
  Graph is acylcic: n cannot reach m, hence we finished inspecting n already.
- *When we run depth-first search for m, n is not yet marked*.

Problem: Ordering tasks

Determine a *topological order*



We annotate depth-first search to collect ordering information:
When we *finish* inspecting a node, we add it to the *front* of our order.

Theorem

Let $(m, n) \in \mathcal{E}$ be an edge in an acyclic graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. Any depth-first search on $\mathcal{G}$ will finish inspecting n before m.

Proof. We consider two cases:

- *When we run depth-first search for m, n is already marked*.
  Graph is acylcic: n cannot reach m, hence we finished inspecting n already.
- *When we run depth-first search for m, n is not yet marked*.
  We find n while inspecting m, hence we finished inspecting n before m.

Problem: Ordering tasks

Algorithm DFS-TS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$, *order*):

for all $(n, m) \in \mathcal{E}$ **do**
    if $\neg$*marked*[m] **then**
        marked[m] := `true`.
        DFS-TS-R($\mathcal{G}$, *marked*, m, *order*).
Add n to the front of *order*.

Algorithm TopologicalSort($\mathcal{G} = (\mathcal{N}, \mathcal{E})$):

marked, *order* := $\{n \mapsto$ `false` $\mid n \in \mathcal{N}\}$, [].
for all $n \in \mathcal{N}$ **do**
    if $\neg$*marked*[n] **then**
        marked[n] := `true`.
        DFS-TS-R($\mathcal{G}$, *marked*, n, *order*).
return *order*.

Problem: Ordering tasks

Algorithm DFS-TS-R($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *marked*, $n \in \mathcal{N}$, *order*):

for all $(n, m) \in \mathcal{E}$ **do**
  if $\neg$*marked*[m] **then**
    marked[m] := `true`.
    DFS-TS-R($\mathcal{G}$, *marked*, m, *order*).
Add n to the front of *order*.

Algorithm TopologicalSort($\mathcal{G} = (\mathcal{N}, \mathcal{E})$):

marked, *order* := $\{n \mapsto$ `false` $\mid n \in \mathcal{N}\}$, [].
for all $n \in \mathcal{N}$ **do**
  if $\neg$*marked*[n] **then**
    marked[n] := `true`.
    DFS-TS-R($\mathcal{G}$, *marked*, n, *order*).
return *order*.

We can easily integrate a cycle-detection step into TopologicalSort.

Problem: Reverse reachability

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and node s.
Depth-first search can find all nodes reachable from node s.

Problem: Reverse reachability

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and node s.
Depth-first search can find all nodes reachable from node s.

Problem: Reverse reachability

How to find all nodes that can reach node s?
E.g., in a one-way communication network:
which participants can communicate messages to s?

Problem: Reverse reachability

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and node s.
Depth-first search can find all nodes reachable from node s.

Problem: Reverse reachability

How to find all nodes that can reach node s?
E.g., in a one-way communication network:
which participants can communicate messages to s?

Solution

Reverse all edges in $\mathcal{G}$ and perform depth-first search on the resulting graph.
Hence, $\textsc{DepthFirstR}(\mathcal{G}', s)$ with $\mathcal{G}' = (\mathcal{N}, \{(n, m) \mid (m, n) \in \mathcal{E}\})$.

Problem: Healthy network

Problem

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which

- the nodes $\mathcal{N}$ represent network devices; and
- the edges $\mathcal{E}$ are network connections.

Can all network devices communicate with all other network devices?

Problem: Healthy network

Problem

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which

- the nodes $\mathcal{N}$ represent network devices; and
- the edges $\mathcal{E}$ are network connections.

Can all network devices communicate with all other network devices?

Problem

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
A graph is *strongly connected* if all node pairs are strongly connected.

Problem: Healthy network

Problem

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
A graph is *strongly connected* if all node pairs are strongly connected.

Observations

There must be a directed path between all pairs of nodes.

Problem: Healthy network

Problem

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

A graph is *strongly connected* if all node pairs are strongly connected.

Observations

There must be a directed path between all pairs of nodes.

Now consider an arbitrary node $s \in \mathcal{N}$.

1. All nodes must have a path to node s.
2. Node s must have a path to all nodes.

Problem: Healthy network

Problem

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

A graph is *strongly connected* if all node pairs are strongly connected.

Observations

There must be a directed path between all pairs of nodes.

Now consider an arbitrary node $s \in \mathcal{N}$.

1. All nodes must have a path to node s.
2. Node s must have a path to all nodes.

We can route all paths proving "strongly connected" via node s.

Problem: Healthy network

Problem

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
A graph is *strongly connected* if all node pairs are strongly connected.

Observations

There must be a directed path between all pairs of nodes.

Now consider an arbitrary node $s \in \mathcal{N}$.

1. All nodes must have a path to node s. $\longrightarrow$ Use *reverse reachability*.
2. Node s must have a path to all nodes. $\longrightarrow$ Use *reachability*.

We can route all paths proving "strongly connected" via node s.

Solution

Use *reverse reachability* and *reachability*.
Both can be done via depth-first search.

Problem: Subcommunities

Problem

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which

- the nodes $\mathcal{N}$ represent social media accounts; and
- the edges $\mathcal{E}$ are interactions between accounts.

We want to find subcommunities (and echo chambers) by looking groups of accounts that all have direct-or-indirect interactions with each other.

Problem: Subcommunities

Problem

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which

- the nodes $\mathcal{N}$ represent social media accounts; and
- the edges $\mathcal{E}$ are interactions between accounts.

We want to find subcommunities (and echo chambers) by looking groups of accounts that all have direct-or-indirect interactions with each other.

Problem

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.
Find all *strongly connected components*.

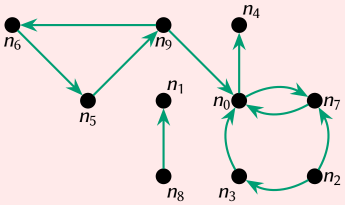
Problem: Subcommunities

Observations

For each $n \in \mathcal{N}$, let $\mathrm{scc}(n)$ be all nodes in the strongly connected component of n.

Consider the graph $\mathcal{G}_{\mathrm{SCC}} = (\mathcal{N}_{\mathrm{SCC}}, \mathcal{E}_{\mathrm{SCC}})$ obtained by *merging* the strongly connected components in $\mathcal{G}$:

- $\mathcal{N}_{\mathrm{SCC}} = \{\mathrm{scc}(n) \mid n \in \mathcal{N}\}$;
- $\mathcal{E}_{\mathrm{SCC}} = \{(\mathrm{scc}(m), \mathrm{scc}(n)) \mid (m, n) \in \mathcal{E}\}$.



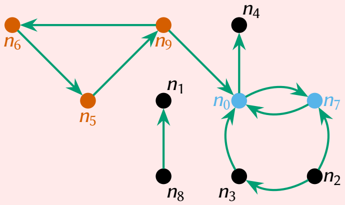
Problem: Subcommunities

Observations

For each $n \in \mathcal{N}$, let $\mathrm{scc}(n)$ be all nodes in the strongly connected component of n.

Consider the graph $\mathcal{G}_{\mathrm{SCC}} = (\mathcal{N}_{\mathrm{SCC}}, \mathcal{E}_{\mathrm{SCC}})$ obtained by *merging* the strongly connected components in $\mathcal{G}$:

- $\mathcal{N}_{\mathrm{SCC}} = \{\mathrm{scc}(n) \mid n \in \mathcal{N}\}$;
- $\mathcal{E}_{\mathrm{SCC}} = \{(\mathrm{scc}(m), \mathrm{scc}(n)) \mid (m, n) \in \mathcal{E}\}$.



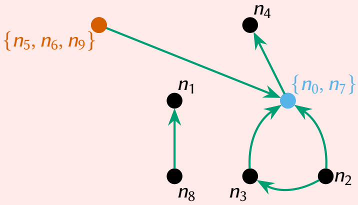
Problem: Subcommunities

Observations

For each $n \in \mathcal{N}$, let $\mathrm{scc}(n)$ be all nodes in the strongly connected component of n.

Consider the graph $\mathcal{G}_{\mathrm{SCC}} = (\mathcal{N}_{\mathrm{SCC}}, \mathcal{E}_{\mathrm{SCC}})$ obtained by *merging* the strongly connected components in $\mathcal{G}$:

- $\mathcal{N}_{\mathrm{SCC}} = \{\mathrm{scc}(n) \mid n \in \mathcal{N}\}$;
- $\mathcal{E}_{\mathrm{SCC}} = \{(\mathrm{scc}(m), \mathrm{scc}(n)) \mid (m, n) \in \mathcal{E}\}$.



Problem: Subcommunities

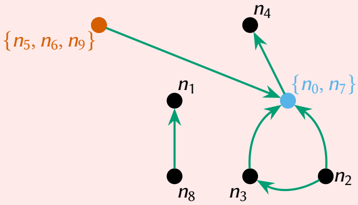
Observations

For each $n \in \mathcal{N}$, let $\mathrm{scc}(n)$ be all nodes in the strongly connected component of n.

Consider the graph $\mathcal{G}_{\mathrm{SCC}} = (\mathcal{N}_{\mathrm{SCC}}, \mathcal{E}_{\mathrm{SCC}})$ obtained by *merging* the strongly connected components in $\mathcal{G}$:

- $\mathcal{N}_{\mathrm{SCC}} = \{\mathrm{scc}(n) \mid n \in \mathcal{N}\}$;
- $\mathcal{E}_{\mathrm{SCC}} = \{(\mathrm{scc}(m), \mathrm{scc}(n)) \mid (m, n) \in \mathcal{E}\}$.

The resulting graph $\mathcal{G}_{\mathrm{SCC}}$ is *acyclic*.



Problem: Subcommunities

Observations

For each $n \in \mathcal{N}$, let $\mathrm{scc}(n)$ be all nodes in the strongly connected component of n.

Consider the graph $\mathcal{G}_{\mathrm{SCC}} = (\mathcal{N}_{\mathrm{SCC}}, \mathcal{E}_{\mathrm{SCC}})$ obtained by *merging* the strongly connected components in $\mathcal{G}$:

- $\mathcal{N}_{\mathrm{SCC}} = \{\mathrm{scc}(n) \mid n \in \mathcal{N}\}$;
- $\mathcal{E}_{\mathrm{SCC}} = \{(\mathrm{scc}(m), \mathrm{scc}(n)) \mid (m, n) \in \mathcal{E}\}$.

The resulting graph $\mathcal{G}_{\mathrm{SCC}}$ is *acyclic*.

Question: What would a *topological sort* of $\mathcal{G}$ produce?

Problem: Subcommunities

Observations

For each $n \in \mathcal{N}$, let $\text{scc}(n)$ be all nodes in the strongly connected component of n.

Consider the graph $\mathcal{G}_{\text{SCC}} = (\mathcal{N}_{\text{SCC}}, \mathcal{E}_{\text{SCC}})$ obtained by *merging* the strongly connected components in $\mathcal{G}$:

- $\mathcal{N}_{\text{SCC}} = \{\text{scc}(n) \mid n \in \mathcal{N}\}$;
- $\mathcal{E}_{\text{SCC}} = \{(\text{scc}(m), \text{scc}(n)) \mid (m, n) \in \mathcal{E}\}$.

The resulting graph $\mathcal{G}_{\text{SCC}}$ is *acyclic*.

Question: What would a *topological sort* of $\mathcal{G}$ produce?
A node order consistent with a topological sort of $\mathcal{G}_{\text{scc}}$:
the strongly connected components stick together.

Problem: Subcommunities

Observations

For each $n \in \mathcal{N}$, let $\text{scc}(n)$ be all nodes in the strongly connected component of n.

Consider the graph $\mathcal{G}_{\text{SCC}} = (\mathcal{N}_{\text{SCC}}, \mathcal{E}_{\text{SCC}})$ obtained by *merging* the strongly connected components in $\mathcal{G}$:

- $\mathcal{N}_{\text{SCC}} = \{\text{scc}(n) \mid n \in \mathcal{N}\}$;
- $\mathcal{E}_{\text{SCC}} = \{(\text{scc}(m), \text{scc}(n)) \mid (m, n) \in \mathcal{E}\}$.

The resulting graph $\mathcal{G}_{\text{SCC}}$ is *acyclic*.

Question: What would a *topological sort* of $\mathcal{G}$ produce?
A node order consistent with a topological sort of $\mathcal{G}_{\text{scc}}$:
the strongly connected components stick together.

We just do not know where one strongly connected component ends and the next begins.

Problem: Subcommunities

Algorithm $\textsc{StronglyConnectedComponent}(\mathcal{G} = (\mathcal{N}, \mathcal{E}))$:

Let $n_0, \ldots, n_{|\mathcal{N}|}$ be a topological sort of $\mathcal{N}$.
$marked := \{n \mapsto \texttt{false} \mid n \in \mathcal{N}\}$.
for $i := 0$ upto $|\mathcal{N}|$ **do**
    if $\neg marked[n_i]$ **then**
        DFS-R$((\mathcal{N}, (\{(n, m) \mid (m, n) \in \mathcal{E}\})), marked, n_i)$ (*reverse reachability*).

Problem: Subcommunities

Algorithm $\textsc{StronglyConnectedComponent}(\mathcal{G} = (\mathcal{N}, \mathcal{E}))$:

Let $n_0, \ldots, n_{|\mathcal{N}|}$ be a topological sort of $\mathcal{N}$.
$\mathit{marked} := \{n \mapsto \texttt{false} \mid n \in \mathcal{N}\}$.
for $i := 0$ upto $|\mathcal{N}|$ **do**
    if $\neg\mathit{marked}[n_i]$ **then**
        n_i is the start of a strongly connected component.

        DFS-R$((\mathcal{N}, (\{(n, m) \mid (m, n) \in \mathcal{E}\})), \mathit{marked}, n_i)$ (*reverse reachability*).

Problem: Subcommunities

Algorithm $\textsc{StronglyConnectedComponent}(\mathcal{G} = (\mathcal{N}, \mathcal{E}))$:

Let $n_0, \ldots, n_{|\mathcal{N}|}$ be a topological sort of $\mathcal{N}$.
$marked := \{n \mapsto \texttt{false} \mid n \in \mathcal{N}\}$.
for $i := 0$ upto $|\mathcal{N}|$ **do**
    if $\neg marked[n_i]$ **then**
        n_i is the start of a strongly connected component.
        Find all nodes not-yet-visited that can *reach* node n_i.
        DFS-R$((\mathcal{N}, (\{(n, m) \mid (m, n) \in \mathcal{E}\})), marked, n_i)$ (*reverse reachability*).

Problem: Subcommunities

Algorithm StronglyConnectedComponent($\mathcal{G} = (\mathcal{N}, \mathcal{E})$):

```
Let n_0, ..., n_|N| be a topological sort of N.
marked := {n ↦ false | n ∈ N}.
for i := 0 upto |N| do
    if ¬marked[n_i] then
        n_i is the start of a strongly connected component.
        Find all nodes not-yet-visited that can reach node n_i.
        DFS-R((N, ({(n, m) | (m, n) ∈ E})), marked, n_i) (reverse reachability).
```

A node that can reach n_i comes before n_i in the topological sort
unless it is part of the same strongly connected component!

Problem: Subcommunities

Algorithm $\textsc{StronglyConnectedComponent}(\mathcal{G} = (\mathcal{N}, \mathcal{E}))$:

```
Let n_0, ..., n_|N| be a topological sort of N.
marked := {n ↦ false | n ∈ N}.
for i := 0 upto |N| do
    if ¬marked[n_i] then
        n_i is the start of a strongly connected component.
        Find all nodes not-yet-visited that can reach node n_i.
        DFS-R((N, ({(n, m) | (m, n) ∈ E})), marked, n_i) (reverse reachability).
```

A node that can reach n_i comes before n_i in the topological sort
unless it is part of the same strongly connected component!

The book presents a variation of the above:
they perform a reverse-topological sort instead of performing reverse reachability.

Problem: Indirect flight connections

Problem: Indirect flight connections

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which

- the nodes $\mathcal{N}$ represent airports; and
- the edges $\mathcal{E}$ are flights between airports.

Construct the edge relation that relates airports m to n if one can fly from m to n (via zero-or-more stops):

$$\{(m, n) \mid \text{there is a sequence of flights connecting } m \text{ to } n\}.$$

Problem: Indirect flight connections

Problem: Indirect flight connections

Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which

- the nodes $\mathcal{N}$ represent airports; and
- the edges $\mathcal{E}$ are flights between airports.

Construct the edge relation that relates airports m to n if one can fly from m to n (via zero-or-more stops):

$$\{(m, n) \mid \text{there is a sequence of flights connecting } m \text{ to } n\}.$$

Definition

The *transitive closure* of a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is the graph $\mathcal{G}_{\text{tc}} = (\mathcal{N}, \mathcal{E}_{\text{tc}})$ with

$$\mathcal{E}_{\text{tc}} = \{(m, n) \mid \text{there is a path from } m \text{ to } n \text{ in } \mathcal{G}\}.$$

Problem: Indirect flight connections

Definition

The *transitive closure* of a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is the graph $\mathcal{G}_{\text{tc}} = (\mathcal{N}, \mathcal{E}_{\text{tc}})$ with

$$\mathcal{E}_{\text{tc}} = \{(m, n) \mid \text{there is a path from } m \text{ to } n \text{ in } \mathcal{G}\}.$$

Solution

$$\mathcal{E}_{\text{tc}} = \{(m, n) \mid \textsc{DepthFirstR}(\mathcal{G}, m) \text{ visits node } n\}.$$

Problem: Indirect flight connections

Definition

The *transitive closure* of a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is the graph $\mathcal{G}_{\text{tc}} = (\mathcal{N}, \mathcal{E}_{\text{tc}})$ with

$$\mathcal{E}_{\text{tc}} = \{(m, n) \mid \text{there is a path from } m \text{ to } n \text{ in } \mathcal{G}\}.$$

Solution

$$\mathcal{E}_{\text{tc}} = \{(m, n) \mid \textsc{DepthFirstR}(\mathcal{G}, m) \text{ visits node } n\}.$$

Complexity

- Runtime complexity is $\Theta\left(|\mathcal{N}|(|\mathcal{N}| + |\mathcal{E}|)\right)$: we run $|\mathcal{N}|$ depth-first searches.
- Memory complexity is $\Theta\left(|\mathcal{N}| + |\mathcal{E}_{\text{tc}}|\right)$: $|\mathcal{E}_{\text{tc}}|$ is likely to be $\Theta\left(|\mathcal{N}|^2\right)$.

Problem: Indirect flight connections

Definition

The *transitive closure* of a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is the graph $\mathcal{G}_{\text{tc}} = (\mathcal{N}, \mathcal{E}_{\text{tc}})$ with

$$\mathcal{E}_{\text{tc}} = \{(m, n) \mid \text{there is a path from } m \text{ to } n \text{ in } \mathcal{G}\}.$$

Solution

$$\mathcal{E}_{\text{tc}} = \{(m, n) \mid \textsc{DepthFirstR}(\mathcal{G}, m) \text{ visits node } n\}.$$

Complexity

- Runtime complexity is $\Theta(|\mathcal{N}|(|\mathcal{N}| + |\mathcal{E}|))$: we run $|\mathcal{N}|$ depth-first searches.
- Memory complexity is $\Theta(|\mathcal{N}| + |\mathcal{E}_{\text{tc}}|)$: $|\mathcal{E}_{\text{tc}}|$ is likely to be $\Theta(|\mathcal{N}|^2)$.

Can we do significantly better? Huge open research question!