Algorithms and Data Structures

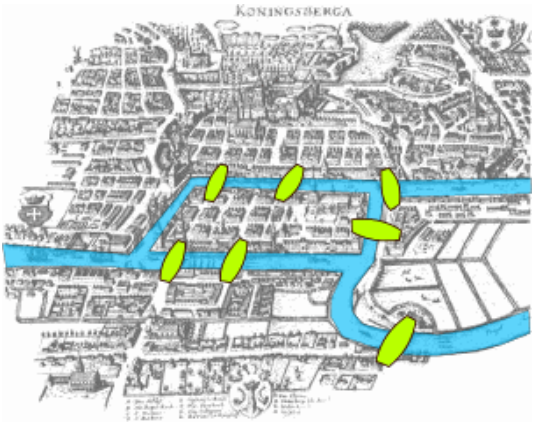
Part 5: Elementary Graph Algorithms

Department of Mathematics
Stockholm University

Elementary Graph Algorithms

Part 5: Elementary Graph Algorithms

Seven Bridges of Königsberg (Euler, 1736)

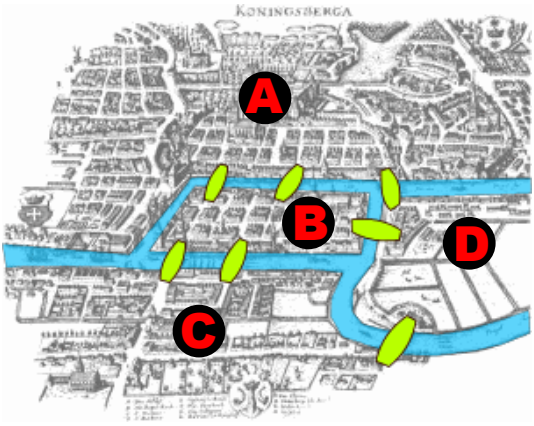


The “first problem” in graph theory:

Is there a walk through the city that would cross each of those bridges once and only once.

Original article: `https://scholarlycommons.pacific.edu/euler-works/`

Seven Bridges of Königsberg (Euler, 1736)

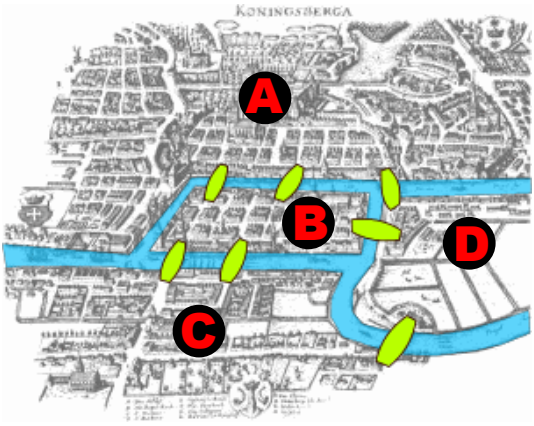


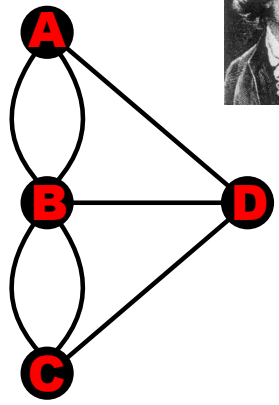
The “first problem” in graph theory:

Is there a walk through the city that would cross each of those bridges once and only once.

Original article: `https://scholarlycommons.pacific.edu/euler-works/`

Seven Bridges of Königsberg (Euler, 1736)





The “first problem” in graph theory:

Is there a walk through the city that would cross each of those bridges once and only once.

Original article: `https://scholarlycommons.pacific.edu/euler-works/`

Graphs are used to abstract and to show relationships

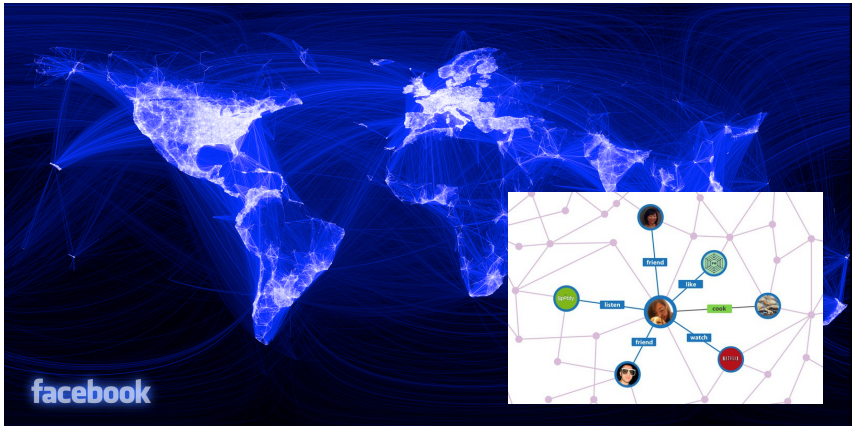
Metro Network

Vertices = metro stations
Edges = direct connections between stations

Source https://tunnelbanakarta.se/

Graphs are used to abstract and to show relationships

Social Networks

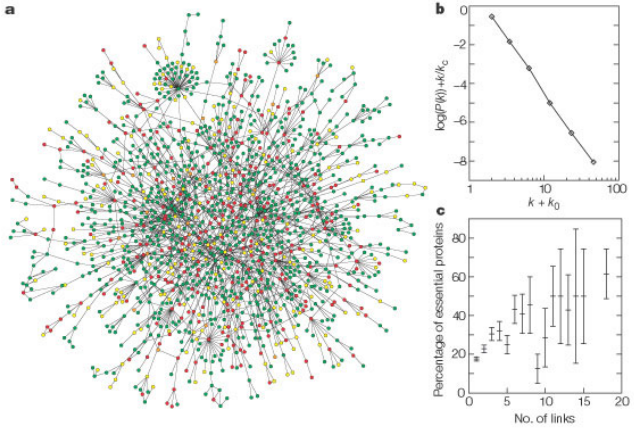


Vertices = user accounts
Edges = two accounts are "friends"

Source: `http://blog.revolutionanalytics.com`

Graphs are used to abstract and to show relationships

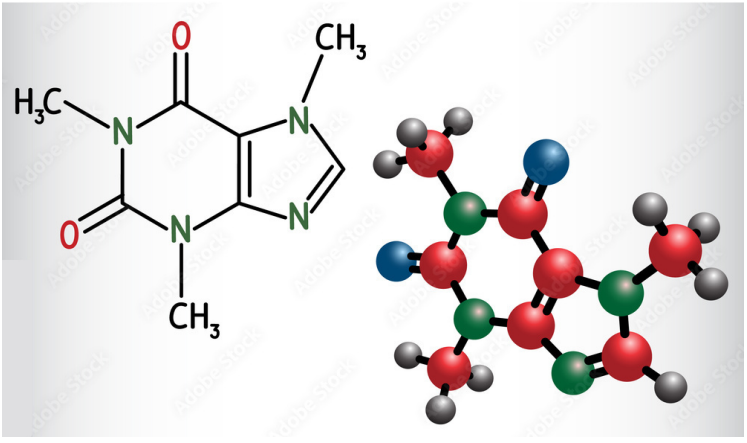
Graph of Protein-Interactions (yeast)



Vertices = proteins
Edges = two proteins interact

Source https://www.nature.com/

Graphs are used to abstract and to show relationships

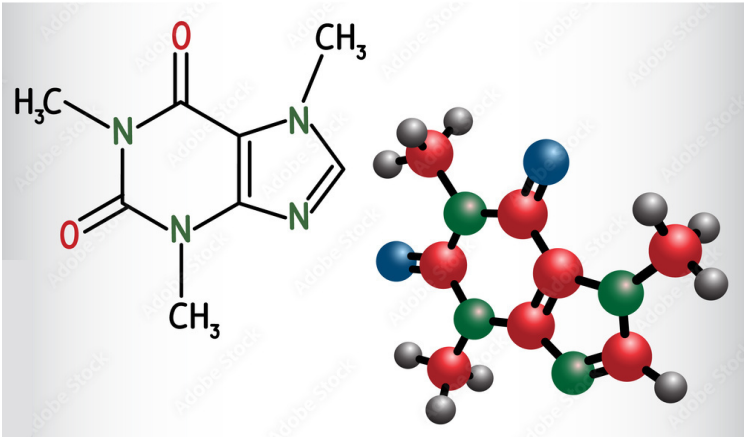


$C_8H_{10}N_4O_2$

Vertices = atoms
Edges = chemical bonds

Source https://stock.adobe.com/

Graphs are used to abstract and to show relationships



$C_8H_{10}N_4O_2$

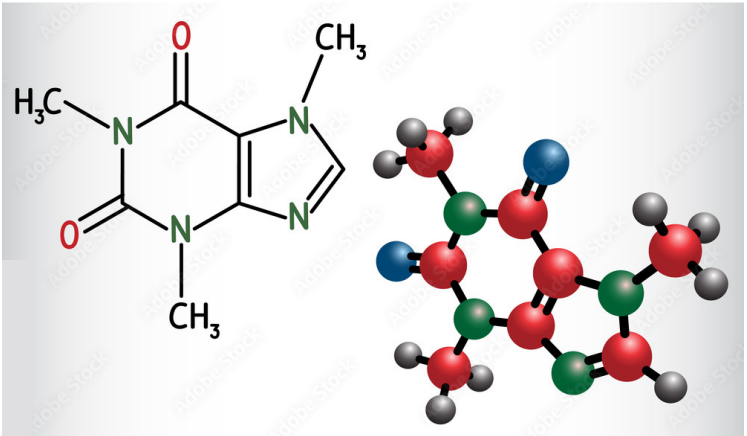
Vertices = atoms
Edges = chemical bonds

Any idea what this molecule is?

(Hint: makes you awake and can be transformed by mathematicians into theorems)

Source https://stock.adobe.com/

Graphs are used to abstract and to show relationships



$C_8H_{10}N_4O_2$ = caffeine

Vertices = atoms
Edges = chemical bonds

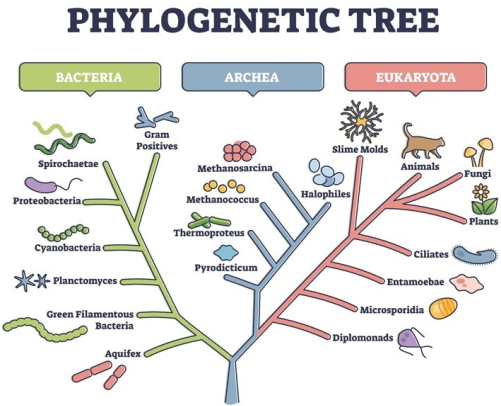
Any idea what this molecule is?

(Hint: makes you awake and can be transformed by mathematicians into theorems)

Source https://stock.adobe.com/

Graphs are used to abstract and to show relationships

Phylogenetics Trees



Vertices = Taxa (e.g. genes or species)
Edges = ancestor relationship

Source: https://www.azolifesciences.com/

Graphs are used to abstract

Graphs can be used to model various types of real-world scenarios.

Given such a graph, it is therefore of interest, to understand its structure (what does structure mean?).

$\Longrightarrow$ we need algorithms to analyze them.

Here, we focus on simple structural properties and basic algorithms to compute them.

Graphs are used to abstract

Graphs can be used to model various types of real-world scenarios.

Given such a graph, it is therefore of interest, to understand its structure (what does structure mean?).

$\Longrightarrow$ we need algorithms to analyze them.

Here, we focus on simple structural properties and basic algorithms to compute them.

Graphs are used to abstract

Graphs can be used to model various types of real-world scenarios.

Given such a graph, it is therefore of interest, to understand its structure (what does structure mean?).

$\Longrightarrow$ we need algorithms to analyze them.

Here, we focus on simple structural properties and basic algorithms to compute them.

Graphs are used to abstract

Graphs can be used to model various types of real-world scenarios.

Given such a graph, it is therefore of interest, to understand its structure (what does structure mean?).

$\Longrightarrow$ we need algorithms to analyze them.

Here, we focus on simple structural properties and basic algorithms to compute them.

Graphs: Basics

Definition

A tuple (V, E) is called an ***undirected graph*** if

V is a finite set, and

E is a set of unordered pairs of elements in V.

V is called the **vertex set**, and the elements of V are called **vertices** (often also **nodes**).
E is called the **edge set**, and the elements of E are called **edges**.

For now, we consider undirected graphs only and call them just **graphs**

Example. $V = \{1, 2, 3, 4, 5\}$

$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$

Graphs: Basics

Definition

A tuple (V, E) is called an ***undirected graph*** if

V is a finite set, and

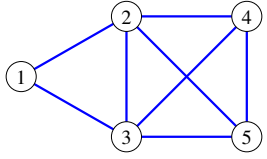
E is a set of unordered pairs of elements in V.

V is called the **vertex set**, and the elements of V are called **vertices** (often also **nodes**).
E is called the **edge set**, and the elements of E are called **edges**.

For now, we consider undirected graphs only and call them just **graphs**

Example. $V = \{1, 2, 3, 4, 5\}$

$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$



Graphs: Basics

Definition

A tuple (V, E) is called an ***undirected graph*** if

V is a finite set, and

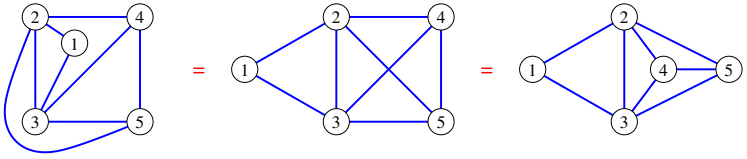
E is a set of unordered pairs of elements in V.

V is called the **vertex set**, and the elements of V are called **vertices** (often also **nodes**).
E is called the **edge set**, and the elements of E are called **edges**.

For now, we consider undirected graphs only and call them just **graphs**

Example. $V = \{1, 2, 3, 4, 5\}$

$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$



Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Proof: Let $T = (V, E)$ be a tree. By definition, T is connected.

$\Rightarrow$ for all $u, v \in V$ there is a uv-path in T.

In particular, there cannot be two uv-paths in T since, otherwise, we can find simple cycles.

$\Rightarrow$ For all $u, v \in V$ there is a *unique* uv-path in T

$\Rightarrow$ For $e = \{u, v\} \in E$, the unique uv-path in T is precisely the edge $\{u, v\}$.

$\Rightarrow$ $(V, E \setminus e)$ does not contain a uv-path and is, therefore, not connected.

□

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Proof: Let $T = (V, E)$ be a tree. By definition, T is connected.

$\Rightarrow$ for all $u, v \in V$ there is a uv-path in T.

In particular, there cannot be two uv-paths in T since, otherwise, we can find simple cycles.

$\Rightarrow$ For all $u, v \in V$ there is a *unique* uv-path in T

$\Rightarrow$ For $e = \{u, v\} \in E$, the unique uv-path in T is precisely the edge $\{u, v\}$.

$\Rightarrow$ $(V, E \setminus e)$ does not contain a uv-path and is, therefore, not connected.

□

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Proof: Let $T = (V, E)$ be a tree. By definition, T is connected.

$\Rightarrow$ for all $u, v \in V$ there is a uv-path in T.

In particular, there cannot be two uv-paths in T since, otherwise, we can find simple cycles.

$\Rightarrow$ For all $u, v \in V$ there is a *unique* uv-path in T

$\Rightarrow$ For $e = \{u, v\} \in E$, the unique uv-path in T is precisely the edge $\{u, v\}$.

$\Rightarrow$ $(V, E \setminus e)$ does not contain a uv-path and is, therefore, not connected.

□

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Proof: Let $T = (V, E)$ be a tree. By definition, T is connected.

$\Rightarrow$ for all $u, v \in V$ there is a uv-path in T.

In particular, there cannot be two uv-paths in T since, otherwise, we can find simple cycles.

$\Rightarrow$ For all $u, v \in V$ there is a *unique* uv-path in T

$\Rightarrow$ For $e = \{u, v\} \in E$, the unique uv-path in T is precisely the edge $\{u, v\}$.

$\Rightarrow$ $(V, E \setminus e)$ does not contain a uv-path and is, therefore, not connected.

□

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Proof: Let $T = (V, E)$ be a tree. By definition, T is connected.

$\Rightarrow$ for all $u, v \in V$ there is a uv-path in T.

In particular, there cannot be two uv-paths in T since, otherwise, we can find simple cycles.

$\Rightarrow$ For all $u, v \in V$ there is a *unique* uv-path in T

$\Rightarrow$ For $e = \{u, v\} \in E$, the unique uv-path in T is precisely the edge $\{u, v\}$.

$\Rightarrow$ $(V, E \setminus e)$ does not contain a uv-path and is, therefore, not connected.

□

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Proof: Let $T = (V, E)$ be a tree. By definition, T is connected.

$\Rightarrow$ for all $u, v \in V$ there is a uv-path in T.

In particular, there cannot be two uv-paths in T since, otherwise, we can find simple cycles.

$\Rightarrow$ For all $u, v \in V$ there is a *unique* uv-path in T

$\Rightarrow$ For $e = \{u, v\} \in E$, the unique uv-path in T is precisely the edge $\{u, v\}$.

$\Rightarrow$ $(V, E \setminus e)$ does not contain a uv-path and is, therefore, not connected.

□

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

We show $|E| = |V| - 1$ by induction on $|V|$.

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

We show $|E| = |V| - 1$ by induction on $|V|$.

Base case:	A tree with one vertex has no edges $\Rightarrow |V| - 1 = 0 = |E|$
	A tree with two vertices contains exactly one edge edge $\Rightarrow |V| - 1 = 1 = |E|$
Ind.-Hyp.:	Assume that statement is true for all trees with $< k$ vertices, $k \geq 2$

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

We show $|E| = |V| - 1$ by induction on $|V|$.

Base case:	A tree with one vertex has no edges $\Rightarrow |V| - 1 = 0 = |E|$
	A tree with two vertices contains exactly one edge edge $\Rightarrow |V| - 1 = 1 = |E|$
Ind.-Hyp.:	Assume that statement is true for all trees with $< k$ vertices, $k \geq 2$

Let $T = (V, E)$ be a tree with $|V| = k$, $k \geq 2$. At least one edge $e \in E$ must exist, else T would be disconnected.

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

We show $|E| = |V| - 1$ by induction on $|V|$.

Base case:	A tree with one vertex has no edges $\Rightarrow |V| - 1 = 0 = |E|$
	A tree with two vertices contains exactly one edge edge $\Rightarrow |V| - 1 = 1 = |E|$
Ind.-Hyp.:	Assume that statement is true for all trees with $< k$ vertices, $k \geq 2$

Let $T = (V, E)$ be a tree with $|V| = k$, $k \geq 2$. At least one edge $e \in E$ must exist, else T would be disconnected.

By the previous lemma, $T' = (V, E \setminus \{e\})$ is disconnected.

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

We show $|E| = |V| - 1$ by induction on $|V|$.

Base case:	A tree with one vertex has no edges $\Rightarrow |V| - 1 = 0 = |E|$
	A tree with two vertices contains exactly one edge edge $\Rightarrow |V| - 1 = 1 = |E|$
Ind.-Hyp.:	Assume that statement is true for all trees with $< k$ vertices, $k \geq 2$

Let $T = (V, E)$ be a tree with $|V| = k$, $k \geq 2$. At least one edge $e \in E$ must exist, else T would be disconnected.

By the previous lemma, $T' = (V, E \setminus \{e\})$ is disconnected.

In particular, e "links" two subgraphs in T' and these two subgraphs must be connected, i.e., T' consists of exactly two connected subgraphs $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$.

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

We show $|E| = |V| - 1$ by induction on $|V|$.

Base case:	A tree with one vertex has no edges $\Rightarrow |V| - 1 = 0 = |E|$
	A tree with two vertices contains exactly one edge edge $\Rightarrow |V| - 1 = 1 = |E|$
Ind.-Hyp.:	Assume that statement is true for all trees with $< k$ vertices, $k \geq 2$

Let $T = (V, E)$ be a tree with $|V| = k$, $k \geq 2$. At least one edge $e \in E$ must exist, else T would be disconnected.

By the previous lemma, $T' = (V, E \setminus \{e\})$ is disconnected.

In particular, e "links" two subgraphs in T' and these two subgraphs must be connected, i.e., T' consists of exactly two connected subgraphs $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$.

Since we have not added edges, T_1 and T_2 are acyclic and, therefore, trees.

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

We show $|E| = |V| - 1$ by induction on $|V|$.

Base case:	A tree with one vertex has no edges $\Rightarrow |V| - 1 = 0 = |E|$
	A tree with two vertices contains exactly one edge edge $\Rightarrow |V| - 1 = 1 = |E|$
Ind.-Hyp.:	Assume that statement is true for all trees with $< k$ vertices, $k \geq 2$

Let $T = (V, E)$ be a tree with $|V| = k$, $k \geq 2$. At least one edge $e \in E$ must exist, else T would be disconnected.

By the previous lemma, $T' = (V, E \setminus \{e\})$ is disconnected.

In particular, e "links" two subgraphs in T' and these two subgraphs must be connected, i.e., T' consists of exactly two connected subgraphs $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$.

Since we have not added edges, T_1 and T_2 are acyclic and, therefore, trees.

By Ind.-Hyp. $|E_1| = |V_1| - 1$ and $|E_2| = |V_2| - 1$.

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

We show $|E| = |V| - 1$ by induction on $|V|$.

Base case:	A tree with one vertex has no edges $\Rightarrow |V| - 1 = 0 = |E|$
	A tree with two vertices contains exactly one edge edge $\Rightarrow |V| - 1 = 1 = |E|$
Ind.-Hyp.:	Assume that statement is true for all trees with $< k$ vertices, $k \geq 2$

Let $T = (V, E)$ be a tree with $|V| = k$, $k \geq 2$. At least one edge $e \in E$ must exist, else T would be disconnected.

By the previous lemma, $T' = (V, E \setminus \{e\})$ is disconnected.

In particular, e "links" two subgraphs in T' and these two subgraphs must be connected, i.e., T' consists of exactly two connected subgraphs $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$.

Since we have not added edges, T_1 and T_2 are acyclic and, therefore, trees.

By Ind.-Hyp. $|E_1| = |V_1| - 1$ and $|E_2| = |V_2| - 1$.

Note that $|V| = |V_1| + |V_2|$ and $|E| = |E_1| + |E_2| + 1$.

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

We show $|E| = |V| - 1$ by induction on $|V|$.

Base case:	A tree with one vertex has no edges $\Rightarrow |V| - 1 = 0 = |E|$
	A tree with two vertices contains exactly one edge edge $\Rightarrow |V| - 1 = 1 = |E|$
Ind.-Hyp.:	Assume that statement is true for all trees with $< k$ vertices, $k \geq 2$

Let $T = (V, E)$ be a tree with $|V| = k$, $k \geq 2$. At least one edge $e \in E$ must exist, else T would be disconnected.

By the previous lemma, $T' = (V, E \setminus \{e\})$ is disconnected.

In particular, e "links" two subgraphs in T' and these two subgraphs must be connected, i.e., T' consists of exactly two connected subgraphs $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$.

Since we have not added edges, T_1 and T_2 are acyclic and, therefore, trees.

By Ind.-Hyp. $|E_1| = |V_1| - 1$ and $|E_2| = |V_2| - 1$.

Note that $|V| = |V_1| + |V_2|$ and $|E| = |E_1| + |E_2| + 1$.

Hence, $|E| = |E_1| + |E_2| + 1 = (|V_1| - 1) + (|V_2| - 1) + 1 = |V_1| + |V_2| - 1 = |V| - 1$.

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *if-direction:* Let $T = (V, E)$ be connected and $|E| = |V| - 1$.

Assume, for contradiction, that T contains *simple* cycle $C = (v_0, v_1, \ldots, v_k, v_0)$, i.e., $P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $e = \{v_k, v_0\} \in E$.

Removing e from T results in $T_1 = (V, E \setminus \{e\})$.

Note T_1 remains connected but it does not contain the cycle C anymore. If T_1 is a tree, we stop.

If T_1 is not a tree„ we continue with the latter process by taking the next simple cycle C' in T_1.
remove an edge from C' to get a connected graph T_2 that does neither contain C nor C'.

After k steps this process must terminate, i.e., we obtain a tree $T_k = (V, \tilde{E})$.

We already proved: $|\tilde{E}| = |V| - 1$ (*only-if-direction*).

By assumption, $|E| = |V| - 1$ and thus, $|\tilde{E}| = |E|$ must hold.

$\Rightarrow E = \tilde{E}$, i.e., there cannot be simple cycles in T and, since T is connected, it is a tree. □

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *if-direction:* Let $T = (V, E)$ be connected and $|E| = |V| - 1$.

Assume, for contradiction, that T contains *simple* cycle $C = (v_0, v_1, \ldots, v_k, v_0)$, i.e., $P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $e = \{v_k, v_0\} \in E$.

Removing e from T results in $T_1 = (V, E \setminus \{e\})$.

Note T_1 remains connected but it does not contain the cycle C anymore. If T_1 is a tree, we stop.

If T_1 is not a tree„ we continue with the latter process by taking the next simple cycle C' in T_1.
remove an edge from C' to get a connected graph T_2 that does neither contain C nor C'.

After k steps this process must terminate, i.e., we obtain a tree $T_k = (V, \tilde{E})$.

We already proved: $|\tilde{E}| = |V| - 1$ (*only-if-direction*).

By assumption, $|E| = |V| - 1$ and thus, $|\tilde{E}| = |E|$ must hold.

$\Rightarrow E = \tilde{E}$, i.e., there cannot be simple cycles in T and, since T is connected, it is a tree. □

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *if-direction:* Let $T = (V, E)$ be connected and $|E| = |V| - 1$.

Assume, for contradiction, that T contains *simple* cycle $C = (v_0, v_1, \ldots, v_k, v_0)$, i.e., $P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $e = \{v_k, v_0\} \in E$.

Removing e from T results in $T_1 = (V, E \setminus \{e\})$.

Note T_1 remains connected but it does not contain the cycle C anymore. If T_1 is a tree, we stop.

If T_1 is not a tree„ we continue with the latter process by taking the next simple cycle C' in T_1.
remove an edge from C' to get a connected graph T_2 that does neither contain C nor C'.

After k steps this process must terminate, i.e., we obtain a tree $T_k = (V, \tilde{E})$.

We already proved: $|\tilde{E}| = |V| - 1$ (*only-if-direction*).

By assumption, $|E| = |V| - 1$ and thus, $|\tilde{E}| = |E|$ must hold.

$\Rightarrow E = \tilde{E}$, i.e., there cannot be simple cycles in T and, since T is connected, it is a tree. □

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *if-direction:* Let $T = (V, E)$ be connected and $|E| = |V| - 1$.

Assume, for contradiction, that T contains *simple* cycle $C = (v_0, v_1, \ldots, v_k, v_0)$, i.e., $P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $e = \{v_k, v_0\} \in E$.

Removing e from T results in $T_1 = (V, E \setminus \{e\})$.

Note T_1 remains connected but it does not contain the cycle C anymore. If T_1 is a tree, we stop.

If T_1 is not a tree„ we continue with the latter process by taking the next simple cycle C' in T_1.
remove an edge from C' to get a connected graph T_2 that does neither contain C nor C'.

After k steps this process must terminate, i.e., we obtain a tree $T_k = (V, \tilde{E})$.

We already proved: $|\tilde{E}| = |V| - 1$ (*only-if-direction*).

By assumption, $|E| = |V| - 1$ and thus, $|\tilde{E}| = |E|$ must hold.

$\Rightarrow E = \tilde{E}$, i.e., there cannot be simple cycles in T and, since T is connected, it is a tree. □

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *if-direction:* Let $T = (V, E)$ be connected and $|E| = |V| - 1$.

Assume, for contradiction, that T contains *simple* cycle $C = (v_0, v_1, \ldots, v_k, v_0)$, i.e., $P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $e = \{v_k, v_0\} \in E$.

Removing e from T results in $T_1 = (V, E \setminus \{e\})$.

Note T_1 remains connected but it does not contain the cycle C anymore. If T_1 is a tree, we stop.

If T_1 is not a tree„ we continue with the latter process by taking the next simple cycle C' in T_1.
remove an edge from C' to get a connected graph T_2 that does neither contain C nor C'.

After k steps this process must terminate, i.e., we obtain a tree $T_k = (V, \tilde{E})$.

We already proved: $|\tilde{E}| = |V| - 1$ (*only-if-direction*).

By assumption, $|E| = |V| - 1$ and thus, $|\tilde{E}| = |E|$ must hold.

$\Rightarrow E = \tilde{E}$, i.e., there cannot be simple cycles in T and, since T is connected, it is a tree. □

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *if-direction:* Let $T = (V, E)$ be connected and $|E| = |V| - 1$.

Assume, for contradiction, that T contains *simple* cycle $C = (v_0, v_1, \ldots, v_k, v_0)$, i.e., $P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $e = \{v_k, v_0\} \in E$.

Removing e from T results in $T_1 = (V, E \setminus \{e\})$.

Note T_1 remains connected but it does not contain the cycle C anymore. If T_1 is a tree, we stop.

If T_1 is not a tree„ we continue with the latter process by taking the next simple cycle C' in T_1.
remove an edge from C' to get a connected graph T_2 that does neither contain C nor C'.

After k steps this process must terminate, i.e., we obtain a tree $T_k = (V, \tilde{E})$.

We already proved: $|\tilde{E}| = |V| - 1$ (*only-if-direction*).

By assumption, $|E| = |V| - 1$ and thus, $|\tilde{E}| = |E|$ must hold.

$\Rightarrow E = \tilde{E}$, i.e., there cannot be simple cycles in T and, since T is connected, it is a tree. □

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *if-direction:* Let $T = (V, E)$ be connected and $|E| = |V| - 1$.

Assume, for contradiction, that T contains *simple* cycle $C = (v_0, v_1, \ldots, v_k, v_0)$, i.e., $P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $e = \{v_k, v_0\} \in E$.

Removing e from T results in $T_1 = (V, E \setminus \{e\})$.

Note T_1 remains connected but it does not contain the cycle C anymore. If T_1 is a tree, we stop.

If T_1 is not a tree„ we continue with the latter process by taking the next simple cycle C' in T_1.
remove an edge from C' to get a connected graph T_2 that does neither contain C nor C'.

After k steps this process must terminate, i.e., we obtain a tree $T_k = (V, \tilde{E})$.

We already proved: $|\tilde{E}| = |V| - 1$ (*only-if-direction*).

By assumption, $|E| = |V| - 1$ and thus, $|\tilde{E}| = |E|$ must hold.

$\Rightarrow E = \tilde{E}$, i.e., there cannot be simple cycles in T and, since T is connected, it is a tree. □

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *if-direction:* Let $T = (V, E)$ be connected and $|E| = |V| - 1$.

Assume, for contradiction, that T contains *simple* cycle $C = (v_0, v_1, \ldots, v_k, v_0)$, i.e., $P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $e = \{v_k, v_0\} \in E$.

Removing e from T results in $T_1 = (V, E \setminus \{e\})$.

Note T_1 remains connected but it does not contain the cycle C anymore. If T_1 is a tree, we stop.

If T_1 is not a tree„ we continue with the latter process by taking the next simple cycle C' in T_1.
remove an edge from C' to get a connected graph T_2 that does neither contain C nor C'.

After k steps this process must terminate, i.e., we obtain a tree $T_k = (V, \tilde{E})$.

We already proved: $|\tilde{E}| = |V| - 1$ (*only-if-direction*).

By assumption, $|E| = |V| - 1$ and thus, $|\tilde{E}| = |E|$ must hold.

$\Rightarrow E = \tilde{E}$, i.e., there cannot be simple cycles in T and, since T is connected, it is a tree. □

Graphs: Basics

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If $T = (V, E)$ is a tree, then $(V, E \setminus e)$ is disconnected for all $e \in E$.*

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *if-direction:* Let $T = (V, E)$ be connected and $|E| = |V| - 1$.

Assume, for contradiction, that T contains *simple* cycle $C = (v_0, v_1, \ldots, v_k, v_0)$, i.e., $P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $e = \{v_k, v_0\} \in E$.

Removing e from T results in $T_1 = (V, E \setminus \{e\})$.

Note T_1 remains connected but it does not contain the cycle C anymore. If T_1 is a tree, we stop.

If T_1 is not a tree„ we continue with the latter process by taking the next simple cycle C' in T_1.
remove an edge from C' to get a connected graph T_2 that does neither contain C nor C'.

After k steps this process must terminate, i.e., we obtain a tree $T_k = (V, \tilde{E})$.

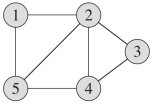
We already proved: $|\tilde{E}| = |V| - 1$ (*only-if-direction*).

By assumption, $|E| = |V| - 1$ and thus, $|\tilde{E}| = |E|$ must hold.

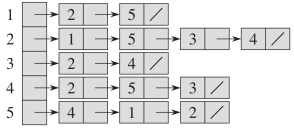
$\Rightarrow E = \tilde{E}$, i.e., there cannot be simple cycles in T and, since T is connected, it is a tree. □

How to store graphs

Before we delve into graph algorithms, let's take a closer look at how to store graphs.
Here, we assume that the vertics in $G = (V, E)$ are integers $1, \ldots, |V|$.



G



adjacency-list

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

adjacency matrix

There are two common standart ways (among others):

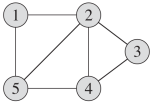
- The adjacency-list of a graph $G = (V, E)$ ..
  .. consists of an array A of $|V|$ linked lists, one for each vertex j in V and each list $A[j]$ contains all vertices $i \in N(j)$, i.e., those i for which $\{i, j\} \in E$ ($O(|V| + |E|)$ space)

- The adjacency matrix of a Boolean $|V| \times |V|$-matrix $A = (a_{ij})$ ($O(|V|^2)$ space)

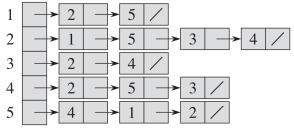
$$\text{with } a_{ij} = \begin{cases} 1 & \text{falls } \{i, j\} \in E \\ 0 & \text{sonst} \end{cases}$$

How to store graphs

Before we delve into graph algorithms, let's take a closer look at how to store graphs.
Here, we assume that the vertics in $G = (V, E)$ are integers $1, \ldots, |V|$.



G



adjacency-list

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

adjacency matrix

There are two common standart ways (among others):

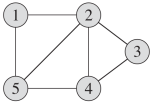
- The adjacency-list of a graph $G = (V, E)$ ..
  .. consists of an array A of $|V|$ linked lists, one for each vertex j in V and each list $A[j]$ contains all vertices $i \in N(j)$, i.e., those i for which $\{i, j\} \in E$ ($O(|V| + |E|)$ space)
  Usually used when graph is sparse, i.e., $|E|$ is much less than $|V^2|$.

- The adjacency matrix of a graph $G = (V, E)$ is a Boolean $|V| \times |V|$-matrix $A = (a_{ij})$ ($O(|V|^2)$ space)

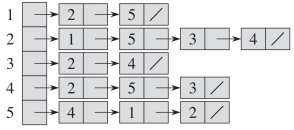
$$\text{with } a_{ij} = \begin{cases} 1 & \text{falls } \{i, j\} \in E \\ 0 & \text{sonst} \end{cases}$$

How to store graphs

Before we delve into graph algorithms, let's take a closer look at how to store graphs.
Here, we assume that the vertics in $G = (V, E)$ are integers $1, \ldots, |V|$.



G



adjacency-list

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

adjacency matrix

There are two common standart ways (among others):

- The adjacency-list of a graph $G = (V, E)$ ..

  .. consists of an array A of $|V|$ linked lists, one for each vertex j in V and each list $A[j]$ contains all vertices $i \in N(j)$, i.e., those i for which $\{i, j\} \in E$ ($O(|V| + |E|)$ space)

  Usually used when graph is sparse, i.e., $|E|$ is much less than $|V^2|$.

- The adjacency matrix of a graph $G = (V, E)$ is a Boolean $|V| \times |V|$-matrix $A = (a_{ij})$ ($O(|V|^2)$ space)

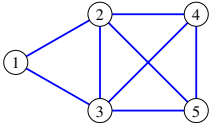
$$\text{with } a_{ij} = \begin{cases} 1 & \text{falls } \{i, j\} \in E \\ 0 & \text{sonst} \end{cases}$$

  Usually used when graph is dense, i.e., $|E| \simeq |V^2|$ or if we want to remove/insert edges in $O(1)$ time.

Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.



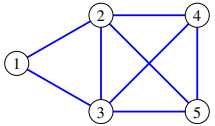
$N(4) = \{2, 3, 5\}$ and $|N(4)| = 3$

Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v \in V} |N(v)| = 2|E|$ *for every graph* $G = (V, E)$.



$N(4) = \{2, 3, 5\}$ and $|N(4)| = 3$

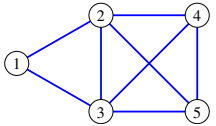
Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v \in V} |N(v)| = 2|E|$ *for every graph* $G = (V, E)$.

Proof: Each edge $\{u, w\}$ connects exactly the two vertices u and w and so it contributes 1 to $|N(u)|$ and 1 to $|N(w)|$. Thus, each edge contributes 2 to $\sum_{v \in V} |N(v)|$. Hence, $\sum_{v \in V} |N(v)|$ is equal to twice the number of edges. □



$N(4) = \{2, 3, 5\}$ and $|N(4)| = 3$

Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v \in V} |N(v)| = 2|E|$ *for every graph* $G = (V, E)$.

```
PrintSomething(G = (V, E)) //G given as adjacency-list
1  WHILE (V ≠ ∅) DO
2     Choose some v ∈ V.
3     V := V \ {v}
4     FOR (all neighbors w ∈ N(v) of v) DO
5        Print(w)
```

Question: How often is `Print`(w) executed?

Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v \in V} |N(v)| = 2|E|$ *for every graph* $G = (V, E)$.

```
PrintSomething(G = (V, E)) //G given as adjacency-list
1  WHILE (V ≠ ∅) DO
2     Choose some v ∈ V.
3     V := V \ {v}
4     FOR (all neighbors w ∈ N(v) of v) DO
5        Print(w)
```

Question: How often is `Print`(w) executed? Answer: $2|E| = O(|E|)$ times

Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v \in V} |N(v)| = 2|E|$ *for every graph* $G = (V, E)$.

```
PrintSomething(G = (V, E)) //G given as adjacency-list
1 WHILE (V ≠ ∅) DO
2     Choose some v ∈ V.
3     V := V \ {v}
4     FOR (all neighbors w ∈ N(v) of v) DO
5         Print(w)
```

Question: How often is `Print`(w) executed? Answer: $2|E| = O(|E|)$ times

Note that Space and Time-complexity of `PrintSomething`$(G = (V, E))$ is in $O(|V| + |E|)$.

Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v \in V} |N(v)| = 2|E|$ *for every graph* $G = (V, E)$.

```
PrintSomething(G = (V, E)) //G given as adjacency-list
1  WHILE (V ≠ ∅) DO
2     Choose some v ∈ V.
3     V := V \ {v}
4     FOR (all neighbors w ∈ N(v) of v) DO
5        Print(w)
```

Question: How often is `Print`(w) executed? Answer: $2|E| = O(|E|)$ times

Note that Space and Time-complexity of `PrintSomething`$(G = (V, E))$ is in $O(|V| + |E|)$.

Question: Do you now classes of graphs where space and time-complexity of `PrintSomething` is both in $O(|V|)$?

Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v \in V} |N(v)| = 2|E|$ *for every graph* $G = (V, E)$.

```
PrintSomething(G = (V, E)) //G given as adjacency-list
 1 WHILE (V ≠ ∅) DO
 2     Choose some v ∈ V.
 3     V := V \ {v}
 4     FOR (all neighbors w ∈ N(v) of v) DO
 5         Print(w)
```

Question: How often is `Print`(w) executed? Answer: $2|E| = O(|E|)$ times

Note that Space and Time-complexity of `PrintSomething`$(G = (V, E))$ is in $O(|V| + |E|)$.

Question: Do you now classes of graphs where space and time-complexity of `PrintSomething` is both in $O(|V|)$?

Answer: For trees $T = (V, E)$ we have $|E| = |V| - 1 \Rightarrow 0.5|V| \leq |E| < |V|$ whenever $|V| \geq 2$
$\Rightarrow O(|V| + |E|) = O(|V|)$ and $O(|E|) = O(|V|)$.

Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v \in V} |N(v)| = 2|E|$ *for every graph* $G = (V, E)$.

Let $G = (V, E)$ and $G' = (V', E')$ be graphs. Then,

G' is called a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$.

G' is called a spanning subgraph of G if G' is a subgraph of G with $V' = V$ (G' contains the same vertices as G).

Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

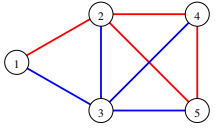
If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v \in V} |N(v)| = 2|E|$ *for every graph* $G = (V, E)$.

Let $G = (V, E)$ and $G' = (V', E')$ be graphs. Then,

G' is called a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$.

G' is called a spanning subgraph of G if G' is a subgraph of G with $V' = V$ (G' contains the same vertices as G).



a subgraph

Graphs: Basics

The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G.

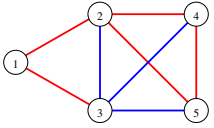
If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v \in V} |N(v)| = 2|E|$ *for every graph* $G = (V, E)$.

Let $G = (V, E)$ and $G' = (V', E')$ be graphs. Then,

G' is called a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$.

G' is called a spanning subgraph of G if G' is a subgraph of G with $V' = V$ (G' contains the same vertices as G).



a spanning subgraph

Graphs: Basics

Graphs can be used to model various types of real-world scenarios.

Given such a graph, it is therefore of interest, to understand its structure (what does structure mean?).

$\Longrightarrow$ we need algorithms to analyze them.

Simple structural properties:

- Is G connected?
- Does G contain simple cycles?
- Is G a tree?
- What is distance between two vertices u, v, i.e., length of shortest (simple) uv-path?

Graphs: Basics

Graphs can be used to model various types of real-world scenarios.

Given such a graph, it is therefore of interest, to understand its structure (what does structure mean?).

$\Longrightarrow$ we need algorithms to analyze them.

Simple structural properties:

- Is G connected?
- Does G contain simple cycles?
- Is G a tree?
- What is distance between two vertices u, v, i.e., length of shortest (simple) uv-path?

Graph Traversal

For the simple graph properties we want to test as listed above we can use graph traversal, that is:

visit the vertices in a graph beginning with a start vertex s such that

Each vertex reachable from s is visited exactly once.
The next visited vertex always has at least one neighbor in the set of previously visited nodes.

We consider here two classical graph traversal algorithms:

Breadth-First Search (BFS)

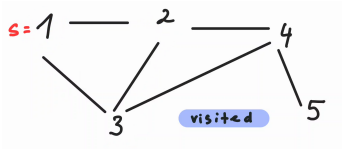
Depth-First Search (DFS)

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```

Breadth-First Search (BFS)

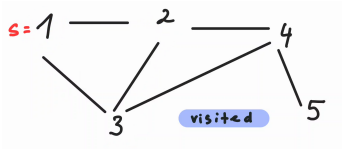
```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```

after L1-4: $Q = (1)$ `visited(1):=true`

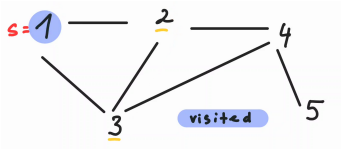


Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```

after L1-4: $Q = (1)$ visited(1):=true

L6 **(BFS-order)**	L8	L9
$v = 1$ and $Q = ()$	$Q = (2, 3)$	visited(2) = visited(3) =true

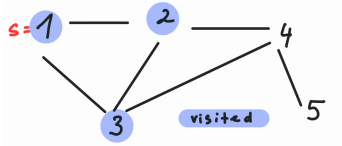


Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```

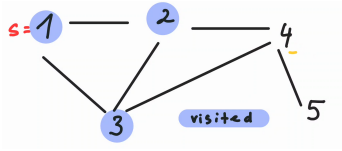
after L1-4: $Q = (1)$ `visited`(1):=`true`

L6 **(BFS-order)**	L8	L9
$v = 1$ and $Q = ()$	$Q = (2, 3)$	`visited`(2) = `visited`(3) =`true`



Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



after L1-4: $Q = (1)$ visited(1):=true

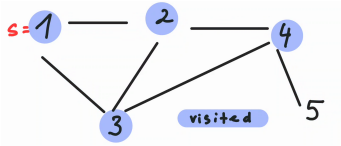
L6 **(BFS-order)**	L8	L9
$v = 1$ and $Q = ()$	$Q = (2, 3)$	visited(2) = visited(3) =true
$v = 2$ and $Q = (3)$	$Q = (3, 4)$	visited(4) =true

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```

after L1-4: $Q = (1)$ visited(1):=true

L6 **(BFS-order)**	L8	L9
$v = 1$ and $Q = ()$	$Q = (2, 3)$	visited(2) = visited(3) =true
$v = 2$ and $Q = (3)$	$Q = (3, 4)$	visited(4) =true

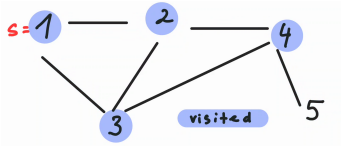


Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```

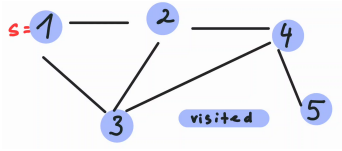
after L1-4: $Q = (1)$ visited(1):=true

L6 **(BFS-order)**	L8	L9
$v = 1$ and $Q = ()$	$Q = (2, 3)$	visited(2) = visited(3) =true
$v = 2$ and $Q = (3)$	$Q = (3, 4)$	visited(4) =true
$v = 3$ and $Q = (4)$	–	–



Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```

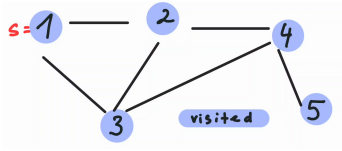


after L1-4: $Q = (1)$ visited(1):=true

L6 **(BFS-order)**	L8	L9
$v = 1$ and $Q = ()$	$Q = (2, 3)$	visited(2) = visited(3) =true
$v = 2$ and $Q = (3)$	$Q = (3, 4)$	visited(4) =true
$v = 3$ and $Q = (4)$	–	–
$v = 4$ and $Q = ()$	$Q = (5)$	visited(5) =true

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```

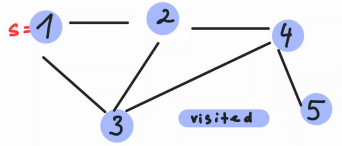


after L1-4: $Q = (1)$ visited(1):=true

L6 **(BFS-order)**	L8	L9
$v = 1$ and $Q = ()$	$Q = (2, 3)$	visited(2) = visited(3) =true
$v = 2$ and $Q = (3)$	$Q = (3, 4)$	visited(4) =true
$v = 3$ and $Q = (4)$	–	–
$v = 4$ and $Q = ()$	$Q = (5)$	visited(5) =true
$v = 5$ and $Q = ()$	–	–

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Runtime: Assume that G is stored as adjacency-list (for all v the set of neighbors $N(v)$ is available)

Only non-`visited` are added to queue and then marked as `visited`
$\Rightarrow$ each $v \in V$ is enqueued at most once, and hence dequeued at most once.

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(|V|)$.

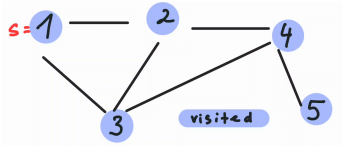
The neighbors in $N(v)$ of each vertex v are scanned only when the vertex is dequeued
$\Rightarrow$ the neighbors in each $N(v)$ are scanned at most once

The handshake-lemma implies $\sum_{v \in V} |N(v)| = 2|E| \in O(|E|)$.

$\Longrightarrow$ while-loop runs in $O(|V| + |E|)$ time.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Runtime: Assume that G is stored as adjacency-list (for all v the set of neighbors $N(v)$ is available)

L1: $O(|V|)$ time // L2-4: $O(|1|)$ time

Only non-`visited` are added to queue and then marked as `visited`
$\Rightarrow$ each $v \in V$ is enqueued at most once, and hence dequeued at most once.

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(|V|)$.

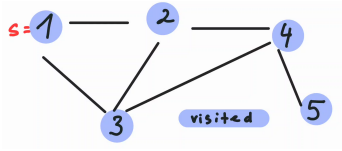
The neighbors in $N(v)$ of each vertex v are scanned only when the vertex is dequeued
$\Rightarrow$ the neighbors in each $N(v)$ are scanned at most once

The handshake-lemma implies $\sum_{v \in V} |N(v)| = 2|E| \in O(|E|)$.

$\Longrightarrow$ while-loop runs in $O(|V| + |E|)$ time.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Runtime: Assume that G is stored as adjacency-list (for all v the set of neighbors $N(v)$ is available)

L1: $O(|V|)$ time // L2-4: $O(|1|)$ time

while-loop

Only non-`visited` are added to queue and then marked as `visited`
$\Rightarrow$ each $v \in V$ is enqueued at most once, and hence dequeued at most once.

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(|V|)$.

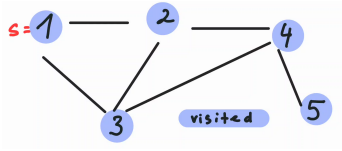
The neighbors in $N(v)$ of each vertex v are scanned only when the vertex is dequeued
$\Rightarrow$ the neighbors in each $N(v)$ are scanned at most once

The handshake-lemma implies $\sum_{v \in V} |N(v)| = 2|E| \in O(|E|)$.

$\Longrightarrow$ while-loop runs in $O(|V| + |E|)$ time.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Runtime: Assume that G is stored as adjacency-list (for all v the set of neighbors $N(v)$ is available)

L1: $O(|V|)$ time // L2-4: $O(|1|)$ time

while-loop

Only non-`visited` are added to queue and then marked as `visited`
$\Rightarrow$ each $v \in V$ is enqueued at most once, and hence dequeued at most once.

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(|V|)$.

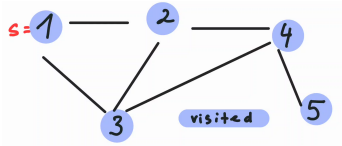
The neighbors in $N(v)$ of each vertex v are scanned only when the vertex is dequeued
$\Rightarrow$ the neighbors in each $N(v)$ are scanned at most once

The handshake-lemma implies $\sum_{v \in V} |N(v)| = 2|E| \in O(|E|)$.

$\Longrightarrow$ while-loop runs in $O(|V| + |E|)$ time.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Runtime: Assume that G is stored as adjacency-list (for all v the set of neighbors $N(v)$ is available)

L1: $O(|V|)$ time // L2-4: $O(|1|)$ time

while-loop

Only non-`visited` are added to queue and then marked as `visited`
$\Rightarrow$ each $v \in V$ is enqueued at most once, and hence dequeued at most once.

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(|V|)$.

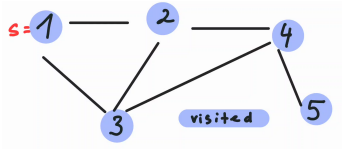
The neighbors in $N(v)$ of each vertex v are scanned only when the vertex is dequeued
$\Rightarrow$ the neighbors in each $N(v)$ are scanned at most once

The handshake-lemma implies $\sum_{v \in V} |N(v)| = 2|E| \in O(|E|)$.

$\Longrightarrow$ while-loop runs in $O(|V| + |E|)$ time.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Runtime: Assume that G is stored as adjacency-list (for all v the set of neighbors $N(v)$ is available)

L1: $O(|V|)$ time // L2-4: $O(|1|)$ time

while-loop

Only non-`visited` are added to queue and then marked as `visited`
$\Rightarrow$ each $v \in V$ is enqueued at most once, and hence dequeued at most once.

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(|V|)$.

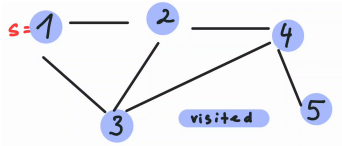
The neighbors in $N(v)$ of each vertex v are scanned only when the vertex is dequeued
$\Rightarrow$ the neighbors in each $N(v)$ are scanned at most once

The handshake-lemma implies $\sum_{v \in V} |N(v)| = 2|E| \in O(|E|)$.

$\Longrightarrow$ while-loop runs in $O(|V| + |E|)$ time.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Runtime: Assume that G is stored as adjacency-list (for all v the set of neighbors $N(v)$ is available)

L1: $O(|V|)$ time // L2-4: $O(|1|)$ time

while-loop

Only non-`visited` are added to queue and then marked as `visited`
$\Rightarrow$ each $v \in V$ is enqueued at most once, and hence dequeued at most once.

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(|V|)$.

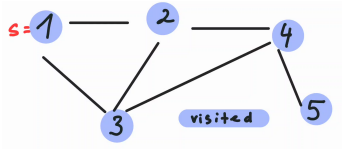
The neighbors in $N(v)$ of each vertex v are scanned only when the vertex is dequeued
$\Rightarrow$ the neighbors in each $N(v)$ are scanned at most once

The handshake-lemma implies $\sum_{v \in V} |N(v)| = 2|E| \in O(|E|)$.

$\Longrightarrow$ while-loop runs in $O(|V| + |E|)$ time.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Runtime: Assume that G is stored as adjacency-list (for all v the set of neighbors $N(v)$ is available)

L1: $O(|V|)$ time // L2-4: $O(|1|)$ time

while-loop

Only non-`visited` are added to queue and then marked as `visited`
$\Rightarrow$ each $v \in V$ is enqueued at most once, and hence dequeued at most once.

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(|V|)$.

The neighbors in $N(v)$ of each vertex v are scanned only when the vertex is dequeued
$\Rightarrow$ the neighbors in each $N(v)$ are scanned at most once

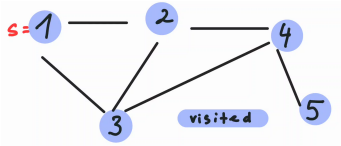
The handshake-lemma implies $\sum_{v \in V} |N(v)| = 2|E| \in O(|E|)$.

$\Longrightarrow$ while-loop runs in $O(|V| + |E|)$ time.

BFS runtime: $O(|V| + |E|)$.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



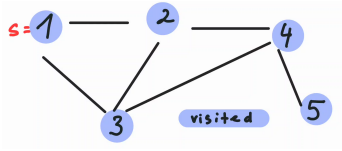
Now, lets squeeze out all structural properties of a graph G we may get using BFS.

In what follows, we say that v is marked as visited, precisely if visited(v) =true

Breadth-First Search (BFS)

`BFS`$(G = (V, E), s)$

```
1 visited(v):=false for all v ∈ V //to keep track of visited nodes
2 init empty queue Q //FIFO
3 visited(s):=true
4 Q.enqueue(s)
5 WHILE (Q ≠ ∅) DO
6     v := Q.front() and Q.dequeue()
7     FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8         Q.enqueue(w)
9         visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Assume, for contradiction, that $x \in V$ is *not* marked as `visited` after run of `BFS`(G, s).

Since s is marked as `visited`, but x is not, there are consecutive vertices v, w in $P = (s, \ldots, v, w, \ldots x)$ such that v is marked as `visited`, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

$\Rightarrow$ At some step of `BFS`(G, s), the vertex v was enqueued to Q (directly before/after v was marked as `visited`).

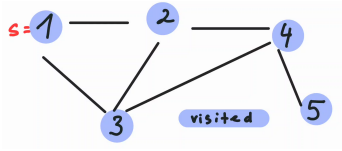
$\Rightarrow$ At some step of `BFS`(G, s), the vertex v is dequeued from Q and the for-loop is entered.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as `visited`.

Breadth-First Search (BFS)

BFS($G = (V, E)$, s)

```
1 visited(v):=false for all v ∈ V //to keep track of visited nodes
2 init empty queue Q //FIFO
3 visited(s):=true
4 Q.enqueue(s)
5 WHILE (Q ≠ ∅) DO
6     v := Q.front() and Q.dequeue()
7     FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8         Q.enqueue(w)
9         visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of BFS(G, s) $\iff$ G is connected

Assume, for contradiction, that $x \in V$ is *not* marked as `visited` after run of BFS(G, s).

Since s is marked as `visited`, but x is not, there are consecutive vertices v, w in $P = (s, \ldots, v, w, \ldots x)$ such that v is marked as `visited`, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

$\Rightarrow$ At some step of BFS(G, s), the vertex v was enqueued to Q (directly before/after v was marked as `visited`).

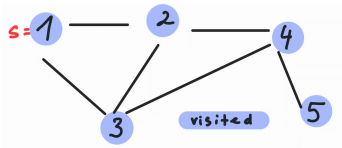
$\Rightarrow$ At some step of BFS(G, s), the vertex v is dequeued from Q and the for-loop is entered.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as `visited`.

Breadth-First Search (BFS)

BFS($G = (V, E), s$)

1 visited(v):=false for all $v \in V$ //to keep track of visited nodes
2 init empty queue Q //FIFO
3 visited(s):=true
4 Q.enqueue(s)
5 WHILE ($Q \neq \emptyset$) DO
6 $v := Q$.front() and Q.dequeue()
7 FOR (all neighbors $w \in N(v)$ of v for which visited(w)=false) DO
8 Q.enqueue(w)
9 visited(w):=true



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as visited after run of BFS(G, s) $\iff$ G is connected

Proof: "$\Leftarrow$" Suppose that G is connected. Thus, there is an sx-path P in G for all $x \in V$.

Assume, for contradiction, that $x \in V$ is *not* marked as visited after run of BFS(G, s).

Since s is marked as visited, but x is not, there are consecutive vertices v, w in $P = (s, \ldots, v, w, \ldots x)$ such that v is marked as visited, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

$\Rightarrow$ At some step of BFS(G, s), the vertex v was enqueued to Q (directly before/after v was marked as visited).

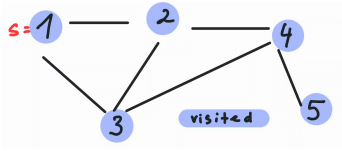
$\Rightarrow$ At some step of BFS(G, s), the vertex v is dequeued from Q and the for-loop is entered.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as visited.

Breadth-First Search (BFS)

BFS($G = (V, E)$, s)

```
1 visited(v):=false for all v ∈ V //to keep track of visited nodes
2 init empty queue Q //FIFO
3 visited(s):=true
4 Q.enqueue(s)
5 WHILE (Q ≠ ∅) DO
6     v := Q.front() and Q.dequeue()
7     FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8         Q.enqueue(w)
9         visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as visited after run of BFS(G, s) $\iff$ G is connected

Proof: "$\Leftarrow$" Suppose that G is connected. Thus, there is an sx-path P in G for all $x \in V$.

Assume, for contradiction, that $x \in V$ is *not* marked as visited after run of BFS(G, s).

Since s is marked as visited, but x is not, there are consecutive vertices v, w in $P = (s, \ldots, v, w, \ldots x)$ such that v is marked as visited, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

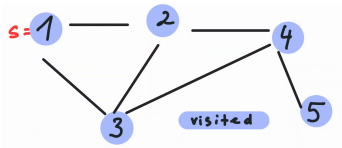
$\Rightarrow$ At some step of BFS(G, s), the vertex v was enqueued to Q (directly before/after v was marked as visited).

$\Rightarrow$ At some step of BFS(G, s), the vertex v is dequeued from Q and the for-loop is entered.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as visited.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of `BFS`(G, s) $\iff$ G is connected

Proof: "$\Leftarrow$" Suppose that G is connected. Thus, there is an sx-path P in G for all $x \in V$.

Assume, for contradiction, that $x \in V$ is *not* marked as `visited` after run of `BFS`(G, s).

Since s is marked as `visited`, but x is not, there are consecutive vertices v, w in $P = (s, \ldots, v, w, \ldots x)$ such that v is marked as `visited`, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

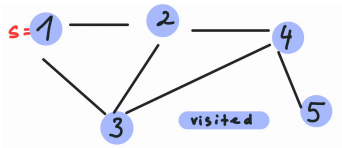
$\Rightarrow$ At some step of `BFS`(G, s), the vertex v was enqueued to Q (directly before/after v was marked as `visited`).

$\Rightarrow$ At some step of `BFS`(G, s), the vertex v is dequeued from Q and the for-loop is entered.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as `visited`.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of `BFS`(G, s) $\iff$ G is connected

Proof: "$\Leftarrow$" Suppose that G is connected. Thus, there is an sx-path P in G for all $x \in V$.

Assume, for contradiction, that $x \in V$ is *not* marked as `visited` after run of `BFS`(G, s).

Since s is marked as `visited`, but x is not, there are consecutive vertices v, w in $P = (s, \ldots, v, w, \ldots x)$ such that v is marked as `visited`, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

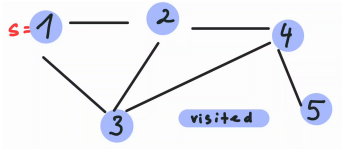
$\Rightarrow$ At some step of `BFS`(G, s), the vertex v was enqueued to Q (directly before/after v was marked as `visited`).

$\Rightarrow$ At some step of `BFS`(G, s), the vertex v is dequeued from Q and the for-loop is entered.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as `visited`.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of `BFS`(G, s) $\iff$ G is connected

Proof: "$\Leftarrow$" Suppose that G is connected. Thus, there is an sx-path P in G for all $x \in V$.

Assume, for contradiction, that $x \in V$ is *not* marked as `visited` after run of `BFS`(G, s).

Since s is marked as `visited`, but x is not, there are consecutive vertices v, w in $P = (s, \ldots, v, w, \ldots x)$ such that v is marked as `visited`, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

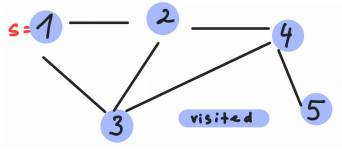
$\Rightarrow$ At some step of `BFS`(G, s), the vertex v was enqueued to Q (directly before/after v was marked as `visited`).

$\Rightarrow$ At some step of `BFS`(G, s), the vertex v is dequeued from Q and the for-loop is entered.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as `visited`.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of `BFS`(G, s) $\iff$ G is connected

Proof: "$\Leftarrow$" Suppose that G is connected. Thus, there is an sx-path P in G for all $x \in V$.

Assume, for contradiction, that $x \in V$ is *not* marked as `visited` after run of `BFS`(G, s).

Since s is marked as `visited`, but x is not, there are consecutive vertices v, w in $P = (s, \ldots, v, w, \ldots x)$ such that v is marked as `visited`, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

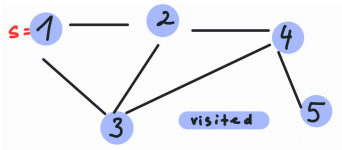
$\Rightarrow$ At some step of `BFS`(G, s), the vertex v was enqueued to Q (directly before/after v was marked as `visited`).

$\Rightarrow$ At some step of `BFS`(G, s), the vertex v is dequeued from Q and the for-loop is entered.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as `visited`.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of `BFS`(G, s) $\iff$ G is connected

Proof: "$\Leftarrow$" Suppose that G is connected. Thus, there is an sx-path P in G for all $x \in V$.

Assume, for contradiction, that $x \in V$ is *not* marked as `visited` after run of `BFS`(G, s).

Since s is marked as `visited`, but x is not, there are consecutive vertices v, w in $P = (s, \ldots, v, w, \ldots x)$ such that v is marked as `visited`, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

$\Rightarrow$ At some step of `BFS`(G, s), the vertex v was enqueued to Q (directly before/after v was marked as `visited`).

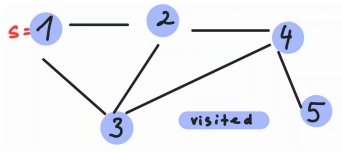
$\Rightarrow$ At some step of `BFS`(G, s), the vertex v is dequeued from Q and the for-loop is entered.

As w is a non-`visited` neighbor of v, it will be considered during the run of the for-loop for v and is thus, marked as `visited`; a contradiction $\lightning$.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as `visited`.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of `BFS`(G, s) $\iff$ G is connected

Proof: "$\Leftarrow$" Suppose that G is connected. Thus, there is an sx-path P in G for all $x \in V$.

Assume, for contradiction, that $x \in V$ is *not* marked as `visited` after run of `BFS`(G, s).

Since s is marked as `visited`, but x is not, there are consecutive vertices v, w in $P = (s, \ldots, v, w, \ldots x)$ such that v is marked as `visited`, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

$\Rightarrow$ At some step of `BFS`(G, s), the vertex v was enqueued to Q (directly before/after v was marked as `visited`).

$\Rightarrow$ At some step of `BFS`(G, s), the vertex v is dequeued from Q and the for-loop is entered.

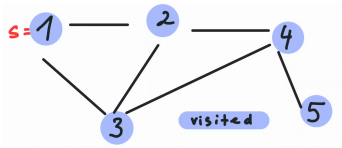
As w is a non-`visited` neighbor of v, it will be considered during the run of the for-loop for v and is thus, marked as `visited`; a contradiction $\lightning$.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as `visited`.

Breadth-First Search (BFS)

BFS($G = (V, E), s$)

```
1 visited(v):=false for all v ∈ V //to keep track of visited nodes
2 init empty queue Q //FIFO
3 visited(s):=true
4 Q.enqueue(s)
5 WHILE (Q ≠ ∅) DO
6     v := Q.front() and Q.dequeue()
7     FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8         Q.enqueue(w)
9         visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of BFS(G, s) $\iff$ G is connected

Proof: "$\Rightarrow$" Assume that all vertices in V are marked as `visited` after run of BFS(G, s).

If $V = \{s\}$, then G is connected. Assume that $|V| > 1$ and let $w \in V \setminus \{s\}$.

Since w is marked as `visited`, we have $w \in N(v_1)$ whereby $v_1 \in Q$ prior to point where w is considered in for-loop

Since v_1 was added to Q, it holds that either
- $v_1 = s$ (in which case we found an sw-path $P = (s, w)$) or
- $v_1 \in N(v_2)$ whereby $v_2 \in Q$ prior to point where v_1 is considered in for-loop

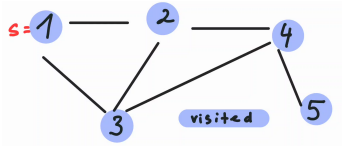
Repeating the latter, ends in a sequence of vertices $v_k, \ldots, v_1, v_0$ with $s = v_k$ and $w = v_0$ and such that $v_i \in N(v_{i+1})$, $0 \leq i \leq k-1$ and thus, we obtain an sw-path $P = (v_k, \ldots, v_0)$.

$\Rightarrow$ for all $w \in V$ there is an sw-path
$\Rightarrow$ For all $x, y \in V$ there is an sx-path and sy-path and combining these two paths yields an xy-path
$\Rightarrow$ G is connected.

□

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of `BFS`(G, s) $\iff$ G is connected

Proof: "$\Rightarrow$" Assume that all vertices in V are marked as `visited` after run of `BFS`(G, s).

If $V = \{s\}$, then G is connected. Assume that $|V| > 1$ and let $w \in V \setminus \{s\}$.

Since w is marked as `visited`, we have $w \in N(v_1)$ whereby $v_1 \in Q$ prior to point where w is considered in for-loop

Since v_1 was added to Q, it holds that either
- $v_1 = s$ (in which case we found an sw-path $P = (s, w)$) or
- $v_1 \in N(v_2)$ whereby $v_2 \in Q$ prior to point where v_1 is considered in for-loop

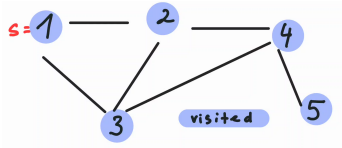
Repeating the latter, ends in a sequence of vertices $v_k, \ldots, v_1, v_0$ with $s = v_k$ and $w = v_0$ and such that $v_i \in N(v_{i+1})$, $0 \leq i \leq k-1$ and thus, we obtain an sw-path $P = (v_k, \ldots, v_0)$.

$\Rightarrow$ for all $w \in V$ there is an sw-path
$\Rightarrow$ For all $x, y \in V$ there is an sx-path and sy-path and combining these two paths yields an xy-path
$\Rightarrow$ G is connected. □

Breadth-First Search (BFS)

BFS($G = (V, E)$, s)

```
1 visited(v):=false for all v ∈ V //to keep track of visited nodes
2 init empty queue Q //FIFO
3 visited(s):=true
4 Q.enqueue(s)
5 WHILE (Q ≠ ∅) DO
6     v := Q.front() and Q.dequeue()
7     FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8         Q.enqueue(w)
9         visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of BFS(G, s) $\iff$ G is connected

Proof: "$\Rightarrow$" Assume that all vertices in V are marked as `visited` after run of BFS(G, s).

If $V = \{s\}$, then G is connected. Assume that $|V| > 1$ and let $w \in V \setminus \{s\}$.

Since w is marked as `visited`, we have $w \in N(v_1)$ whereby $v_1 \in Q$ prior to point where w is considered in for-loop

Since v_1 was added to Q, it holds that either
- $v_1 = s$ (in which case we found an sw-path $P = (s, w)$) or
- $v_1 \in N(v_2)$ whereby $v_2 \in Q$ prior to point where v_1 is considered in for-loop

Repeating the latter, ends in a sequence of vertices $v_k, \ldots, v_1, v_0$ with $s = v_k$ and $w = v_0$ and such that $v_i \in N(v_{i+1})$, $0 \leq i \leq k-1$ and thus, we obtain an sw-path $P = (v_k, \ldots, v_0)$.

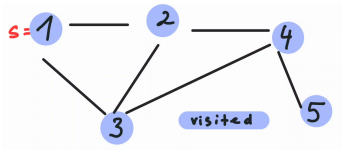
$\Rightarrow$ for all $w \in V$ there is an sw-path
$\Rightarrow$ For all $x, y \in V$ there is an sx-path and sy-path and combining these two paths yields an xy-path
$\Rightarrow$ G is connected.

□

Breadth-First Search (BFS)

BFS$(G = (V, E), s)$

1 visited(v):=false for all $v \in V$ //to keep track of visited nodes
2 init empty queue Q //FIFO
3 visited(s):=true
4 Q.enqueue(s)
5 WHILE $(Q \neq \emptyset)$ DO
6 $v := Q$.front() and Q.dequeue()
7 FOR (all neighbors $w \in N(v)$ of v for which visited(w)=false) DO
8 Q.enqueue(w)
9 visited(w):=true



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as visited after run of BFS(G, s) $\iff$ G is connected

Proof: "$\Rightarrow$" Assume that all vertices in V are marked as visited after run of BFS(G, s).

If $V = \{s\}$, then G is connected. Assume that $|V| > 1$ and let $w \in V \setminus \{s\}$.

Since w is marked as visited, we have $w \in N(v_1)$ whereby $v_1 \in Q$ prior to point where w is considered in for-loop

Since v_1 was added to Q, it holds that either
$v_1 = s$ (in which case we found an sw-path $P = (s, w)$) or
$v_1 \in N(v_2)$ whereby $v_2 \in Q$ prior to point where v_1 is considered in for-loop

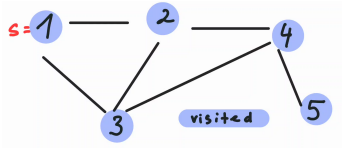
Repeating the latter, ends in a sequence of vertices $v_k, \ldots, v_1, v_0$ with $s = v_k$ and $w = v_0$ and such that $v_i \in N(v_{i+1})$, $0 \leq i \leq k-1$ and thus, we obtain an sw-path $P = (v_k, \ldots, v_0)$.

$\Rightarrow$ for all $w \in V$ there is an sw-path
$\Rightarrow$ For all $x, y \in V$ there is an sx-path and sy-path and combining these two paths yields an xy-path
$\Rightarrow$ G is connected. □

Breadth-First Search (BFS)

BFS($G = (V, E), s$)

```
1 visited(v):=false for all v ∈ V //to keep track of visited nodes
2 init empty queue Q //FIFO
3 visited(s):=true
4 Q.enqueue(s)
5 WHILE (Q ≠ ∅) DO
6     v := Q.front() and Q.dequeue()
7     FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8         Q.enqueue(w)
9         visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of BFS(G, s) $\iff$ G is connected

Proof: "$\Rightarrow$" Assume that all vertices in V are marked as `visited` after run of BFS(G, s).

If $V = \{s\}$, then G is connected. Assume that $|V| > 1$ and let $w \in V \setminus \{s\}$.

Since w is marked as `visited`, we have $w \in N(v_1)$ whereby $v_1 \in Q$ prior to point where w is considered in for-loop

Since v_1 was added to Q, it holds that either
- $v_1 = s$ (in which case we found an sw-path $P = (s, w)$) or
- $v_1 \in N(v_2)$ whereby $v_2 \in Q$ prior to point where v_1 is considered in for-loop

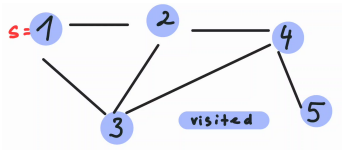
Repeating the latter, ends in a sequence of vertices $v_k, \ldots, v_1, v_0$ with $s = v_k$ and $w = v_0$ and such that $v_i \in N(v_{i+1})$, $0 \leq i \leq k-1$ and thus, we obtain an sw-path $P = (v_k, \ldots, v_0)$.

$\Rightarrow$ for all $w \in V$ there is an sw-path
$\Rightarrow$ For all $x, y \in V$ there is an sx-path and sy-path and combining these two paths yields an xy-path
$\Rightarrow$ G is connected. □

Breadth-First Search (BFS)

`BFS`$(G = (V, E), s)$

```
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as `visited` after run of `BFS`(G, s) $\iff$ G is connected

Proof: "$\Rightarrow$" Assume that all vertices in V are marked as `visited` after run of `BFS`(G, s).

If $V = \{s\}$, then G is connected. Assume that $|V| > 1$ and let $w \in V \setminus \{s\}$.

Since w is marked as `visited`, we have $w \in N(v_1)$ whereby $v_1 \in Q$ prior to point where w is considered in for-loop

Since v_1 was added to Q, it holds that either
- $v_1 = s$ (in which case we found an sw-path $P = (s, w)$) or
- $v_1 \in N(v_2)$ whereby $v_2 \in Q$ prior to point where v_1 is considered in for-loop

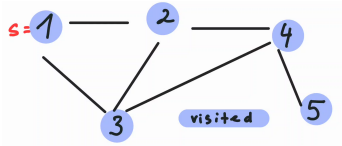
Repeating the latter, ends in a sequence of vertices $v_k, \ldots, v_1, v_0$ with $s = v_k$ and $w = v_0$ and such that $v_i \in N(v_{i+1})$, $0 \leq i \leq k-1$ and thus, we obtain an sw-path $P = (v_k, \ldots, v_0)$.

$\Rightarrow$ for all $w \in V$ there is an sw-path
$\Rightarrow$ For all $x, y \in V$ there is an sx-path and sy-path and combining these two paths yields an xy-path
$\Rightarrow$ G is connected.

□

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V //to keep track of visited nodes
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
```



Structural Property: Is $G = (V, E)$ connected?

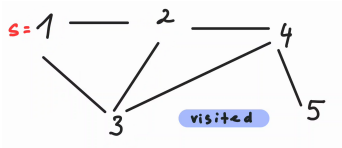
Lemma. All vertices in V are marked as `visited` after run of `BFS`(G, s) $\iff$ G is connected.

Thus, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is connected can be done $O(|V| + |E|)$ time.

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
10
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

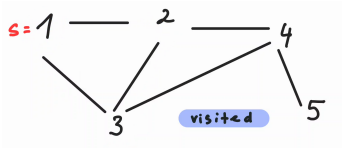
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

visited edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
10
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

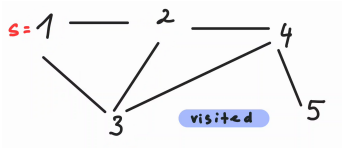
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

visited edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
10
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

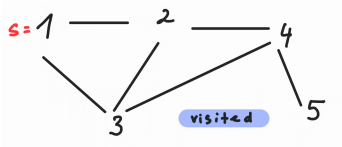
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

visited edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
10
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

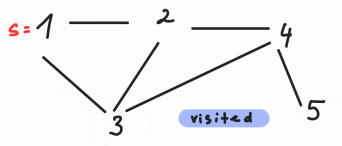
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

visited edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
BFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8          Q.enqueue(w)
9          visited(w):=true
10
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

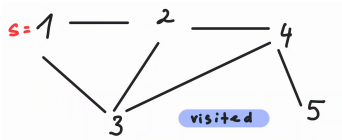
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

visited edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

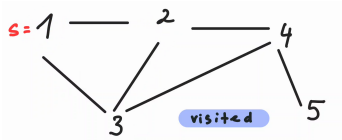
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

visited edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

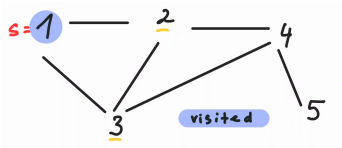
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

visited edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

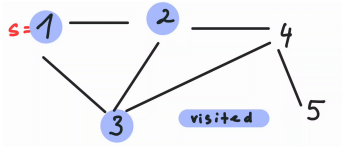
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

visited edges: $\{1,2\}$ $\{1,3\}$,$\{2,4\}$, $\{4,5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

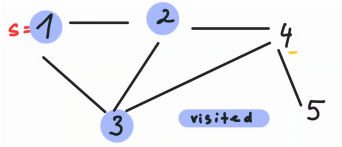
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

visited edges: $\{1, 2\}$ $\{1, 3\}$, $\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

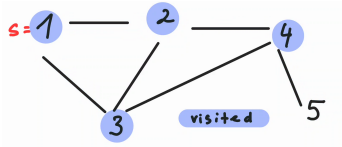
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

visited edges: $\{1, 2\}$ $\{1, 3\}$, $\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify `BFS` by marking also edges as `visited`.

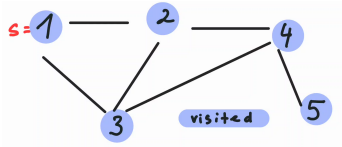
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

`visited` edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify `BFS` by marking also edges as `visited`.

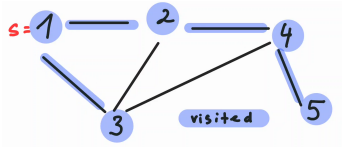
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

`visited` edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree.

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify `BFS` by marking also edges as `visited`.

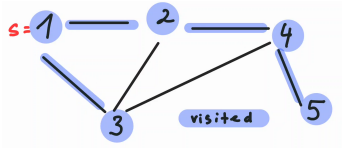
Example. Given the same order in which vertices are added to Q as in previous slides: 1, 2, 3, 4, 5

`visited` edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Breadth-First Search (BFS)

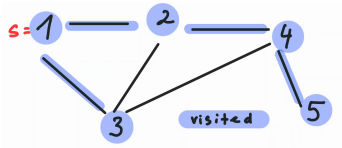
```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```

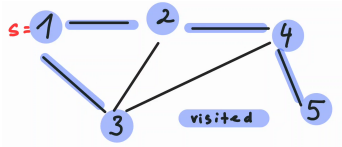


Find spanning tree of G (if there is one).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



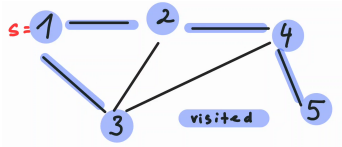
Find spanning tree of G (if there is one).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

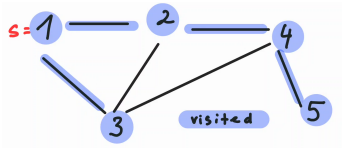
Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

Proof: We show that T_{BFS} is connected and has $|V| - 1$ edges.

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

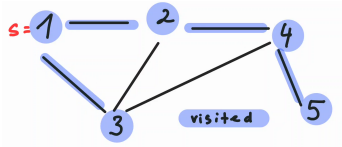
Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

Proof: We show that T_{BFS} is connected and has $|V| - 1$ edges.

Since G is connected, we can apply the same arguments as in the proof of the "⇒"-direction of the lemma "All vertices marked as `visited` iff G connected", to conclude that, for all $w \in V$, there is sw-path along `visited` vertices that consists of `visited` edges only. Hence, T_{BFS} is connected.

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

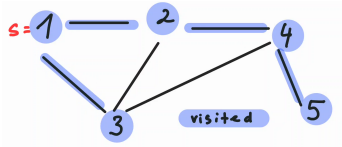
Proof: We show that T_{BFS} is connected and has $|V| - 1$ edges.

Since G is connected, we can apply the same arguments as in the proof of the "⇒"-direction of the lemma "All vertices marked as `visited` iff G connected", to conclude that, for all $w \in V$, there is sw-path along `visited` vertices that consists of `visited` edges only. Hence, T_{BFS} is connected.

Since G is connected, the latter lemma also implies that all vertices are marked as `visited`. In particular, all $|V| - 1$ vertices distinct from s must have been marked during the execution of the for-loop and each vertex is marked `visited` precisely once. Thus, exactly $|V| - 1$ edges have been marked `visited`. Hence, F contains precisely $|V| - 1$ edges.

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

Proof: We show that T_{BFS} is connected and has $|V| - 1$ edges.

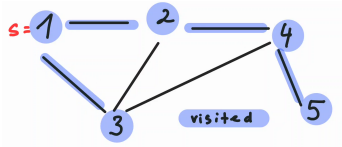
Since G is connected, we can apply the same arguments as in the proof of the "⇒"-direction of the lemma "All vertices marked as `visited` iff G connected", to conclude that, for all $w \in V$, there is sw-path along `visited` vertices that consists of `visited` edges only. Hence, T_{BFS} is connected.

Since G is connected, the latter lemma also implies that all vertices are marked as `visited`. In particular, all $|V| - 1$ vertices distinct from s must have been marked during the execution of the for-loop and each vertex is marked `visited` precisely once. Thus, exactly $|V| - 1$ edges have been marked `visited`. Hence, F contains precisely $|V| - 1$ edges.

Thus, T_{BFS} is connected and has $|V| - 1$ edges and the Tree-Theorem implies that T_{BFS} is a tree.

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

Proof: We show that T_{BFS} is connected and has $|V| - 1$ edges.

Since G is connected, we can apply the same arguments as in the proof of the "⇒"-direction of the lemma "All vertices marked as `visited` iff G connected", to conclude that, for all $w \in V$, there is sw-path along `visited` vertices that consists of `visited` edges only. Hence, T_{BFS} is connected.

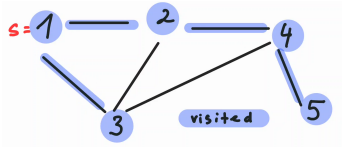
Since G is connected, the latter lemma also implies that all vertices are marked as `visited`. In particular, all $|V| - 1$ vertices distinct from s must have been marked during the execution of the for-loop and each vertex is marked `visited` precisely once. Thus, exactly $|V| - 1$ edges have been marked `visited`. Hence, F contains precisely $|V| - 1$ edges.

Thus, T_{BFS} is connected and has $|V| - 1$ edges and the Tree-Theorem implies that T_{BFS} is a tree.

Since the vertex set of T_{BFS} and G is V, T_{BFS} is a spanning tree of G □

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



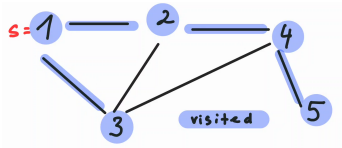
Find spanning tree of G (if there is one).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

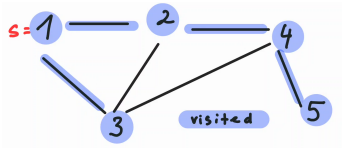
Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

Lemma. *$G = (V, E)$ is a tree if and only if G is connected and $|F| = |E|$*

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

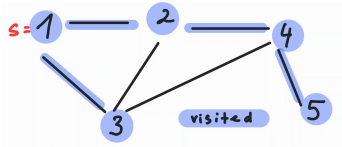
Lemma. *$G = (V, E)$ is a tree if and only if G is connected and $|F| = |E|$*

Proof: If $G = (V, E)$ is a tree, then G is connected and there is only one spanning tree and thus, $G = T_{BFS}$ (hence, $|E| = |F|$)

Suppose that G is connected and $|F| = |E|$. Since F is edge set of the spanning tree T_{BFS} of G it holds that $|F| = |V| - 1$ and thus, $|E| = |V| - 1$. By the Tree-Threorem, G is a tree.

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

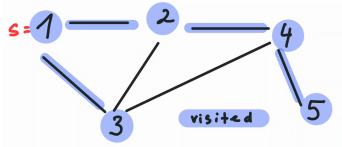
Lemma. *$G = (V, E)$ is a tree if and only if G is connected and $|F| = |E|$*

Proof: If $G = (V, E)$ is a tree, then G is connected and there is only one spanning tree and thus, $G = T_{BFS}$ (hence, $|E| = |F|$)

Suppose that G is connected and $|F| = |E|$. Since F is edge set of the spanning tree T_{BFS} of G it holds that $|F| = |V| - 1$ and thus, $|E| = |V| - 1$. By the Tree-Threorem, G is a tree.

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

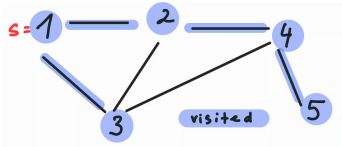
Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Summary so-far: For a given graph $G = (V, E)$, `BFS` and its modification allows us to determine in $O(|V| + |E|)$ time ...

- ... if G is connected (all vertices in V are marked as `visited`)
- ... if G is a tree. (Check first connectedness and then check if $|E| = |V| - 1$ or, alternatively, count number f of `visited` edges and compare f and $|E|$)
- ... a spanning tree $T_{BFS} = (V, F)$ of G in case G is connected.

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

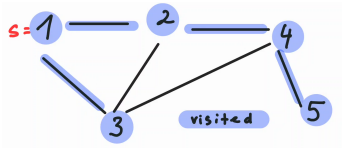
Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Summary so-far: For a given graph $G = (V, E)$, `BFS` and its modification allows us to determine in $O(|V| + |E|)$ time ...

- ... if G is connected (all vertices in V are marked as `visited`)
- ... if G is a tree. (Check first connectedness and then check if $|E| = |V| - 1$ or, alternatively, count number f of `visited` edges and compare f and $|E|$)
- ... a spanning tree $T_{BFS} = (V, F)$ of G in case G is connected.

Breadth-First Search (BFS)

```
modi_BFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V and visited(e):=false for all e ∈ E
 2  init empty queue Q //FIFO
 3  visited(s):=true
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8          Q.enqueue(w)
 9          visited(w):=true
10          visited({v, w}):=true
```



Find spanning tree of G (if there is one).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Summary so-far: For a given graph $G = (V, E)$, `BFS` and its modification allows us to determine in $O(|V| + |E|)$ time ...

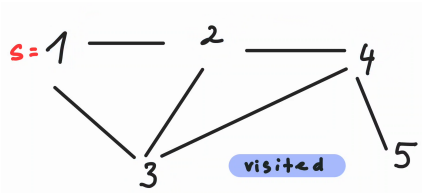
- ... if G is connected (all vertices in V are marked as `visited`)
- ... if G is a tree. (Check first connectedness and then check if $|E| = |V| - 1$ or, alternatively, count number f of `visited` edges and compare f and $|E|$)
- ... a spanning tree $T_{BFS} = (V, F)$ of G in case G is connected.

The latter results also imply

Theorem. *For each connected graph $G = (V, E)$ we have $|E| \geq |V| - 1$ and G has a spanning tree.*

Breadth-First Search (BFS)

```
   BFS(G = (V, E), s)
1  visited(v):=false                    for all v ∈ V
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8
9          Q.enqueue(w)
10         visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

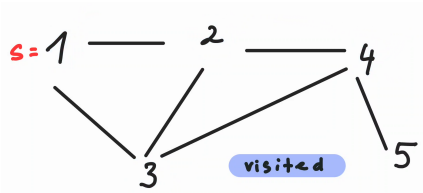
In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x,y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

Breadth-First Search (BFS)

```
   BFS(G = (V, E), s)
1  visited(v):=false                    for all v ∈ V
2  init empty queue Q //FIFO
3  visited(s):=true
4  Q.enqueue(s)
5  WHILE (Q ≠ ∅) DO
6      v := Q.front() and Q.dequeue()
7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
8
9          Q.enqueue(w)
10         visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

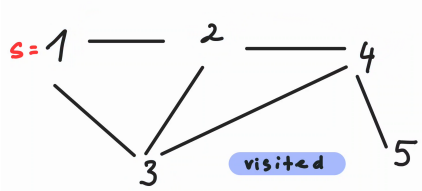
$$d(x,y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x,y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

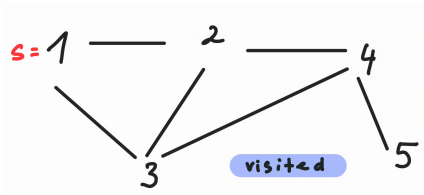
Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

To this end, consider the following modification `dist_BFS`.

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x,y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

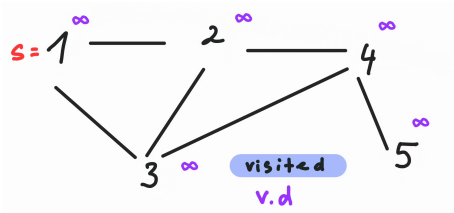
In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x, y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

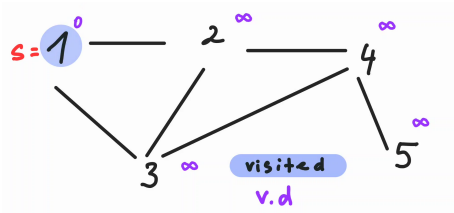
In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x, y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

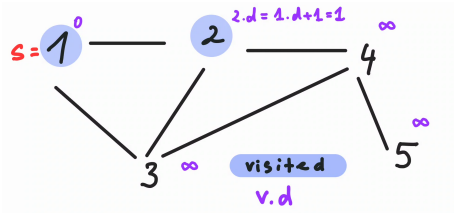
In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x, y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

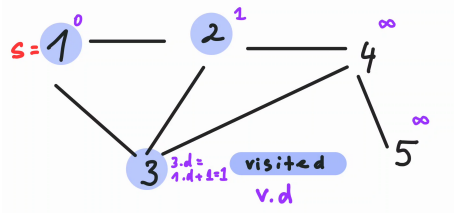
In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x, y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

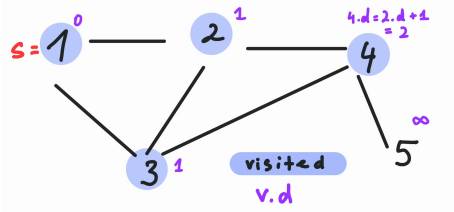
In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x, y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

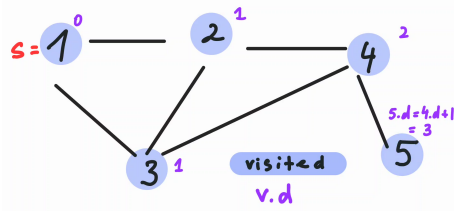
In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x, y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

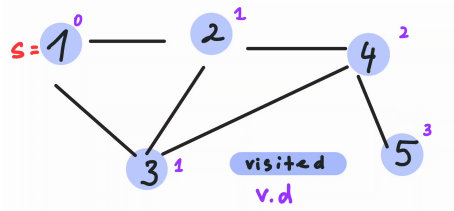
In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x, y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

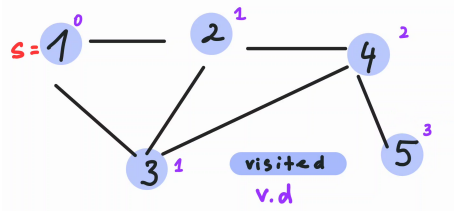
In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x, y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

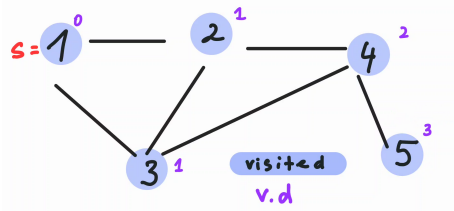
To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

What have we computed?

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x, y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

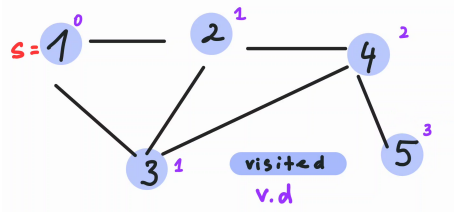
To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

What have we computed? Answer: $d(s, v)$ for all $v \in V$.

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x, y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| := k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

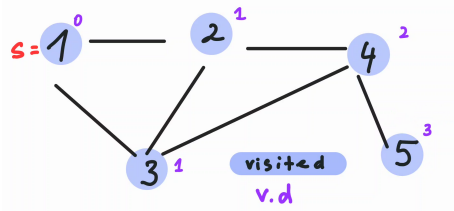
To this end, consider the following modification `dist_BFS`.

Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

What have we computed? Answer: $d(s, v)$ for all $v \in V$. Let's prove correctness!

Breadth-First Search (BFS)

```
dist_BFS(G = (V, E), s)
 1  visited(v):=false and v.d := ∞ for all v ∈ V
 2  init empty queue Q //FIFO
 3  visited(s):=true and s.d := 0
 4  Q.enqueue(s)
 5  WHILE (Q ≠ ∅) DO
 6      v := Q.front() and Q.dequeue()
 7      FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 8        w.d := v.d + 1
 9        Q.enqueue(w)
10        visited(w):=true
```



Structural Property: Distances in $G = (V, E)$?

Theorem. *dist_BFS(G,s) correctly computes the distances $d(s, v)$ between s and all $v \in V$ in $O(|V| + |E|)$ time.*

proof board (before some conclusion we can draw)

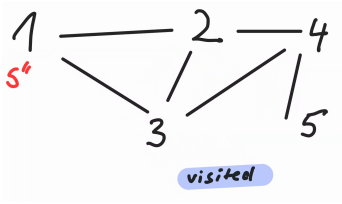
We could run dist_BFS(G,w) for each $w \in V$ as start vertex and thus, obtain the distances $d(w, v)$ for all $w, v \in V$, which implies

Lemma. *The distances between all pairs of vertices in $G = (V, E)$ can be computed $O(|V|(|V| + |E|)) = O(|V|^2 + |V||E|)$ time.*

There are many other algorithms that can be used to determine distances that even work in the case that we may have edge weights (here dist_BFS would fail in general).

Depth-First Search (DFS)

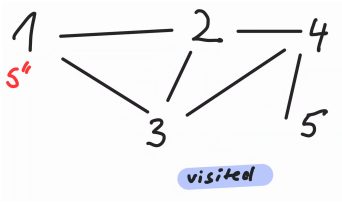
```
DFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V
 2  init empty stack S //LIFO
 3  S.push(s)
 4  WHILE (S ≠ ∅) DO
 5     v := S.top() and S.pop()
 6     IF (visited(v)≠true) THEN
 7        visited(v):=true
 8        FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 9           S.push(s)
```



Depth-First Search (DFS)

```
DFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V
 2  init empty stack S //LIFO
 3  S.push(s)
 4  WHILE (S ≠ ∅) DO
 5     v := S.top() and S.pop()
 6     IF (visited(v)≠true) THEN
 7        visited(v):=true
 8        FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 9           S.push(s)
```

after L1-4: $S = (1)$

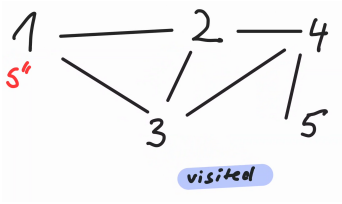


Depth-First Search (DFS)

```
DFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V
 2  init empty stack S //LIFO
 3  S.push(s)
 4  WHILE (S ≠ ∅) DO
 5     v := S.top() and S.pop()
 6     IF (visited(v)≠true) THEN
 7        visited(v):=true
 8        FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 9           S.push(s)
```

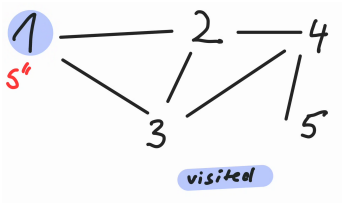
after L1-4: $S = (1)$

L5 L7 **(DFS-order)** L9



Depth-First Search (DFS)

```
DFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V
 2  init empty stack S //LIFO
 3  S.push(s)
 4  WHILE (S ≠ ∅) DO
 5     v := S.top() and S.pop()
 6     IF (visited(v)≠true) THEN
 7        visited(v):=true
 8        FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 9           S.push(s)
```

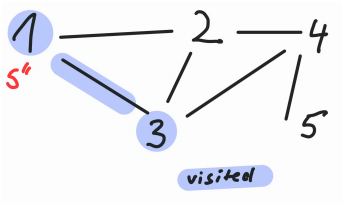


after L1-4: $S = (1)$

L5	L7 **(DFS-order)**	L9
$v = 1$ and $S = ()$	visited(1) =true	$S = (2, 3)$

Depth-First Search (DFS)

```
DFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V
 2  init empty stack S //LIFO
 3  S.push(s)
 4  WHILE (S ≠ ∅) DO
 5      v := S.top() and S.pop()
 6      IF (visited(v)≠true) THEN
 7          visited(v):=true
 8          FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 9              S.push(s)
```

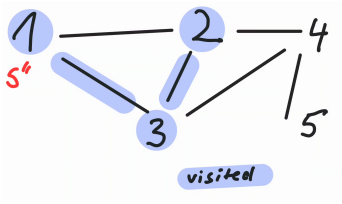


after L1-4: $S = (1)$

L5	L7 **(DFS-order)**	L9
$v = 1$ and $S = ()$	visited(1) =true	$S = (2, 3)$
$v = 3$ and $S = (2)$	visited(3) =true	$S = (2, 4, 2)$

Depth-First Search (DFS)

```
DFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V
 2  init empty stack S //LIFO
 3  S.push(s)
 4  WHILE (S ≠ ∅) DO
 5      v := S.top() and S.pop()
 6      IF (visited(v)≠true) THEN
 7          visited(v):=true
 8          FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 9              S.push(s)
```

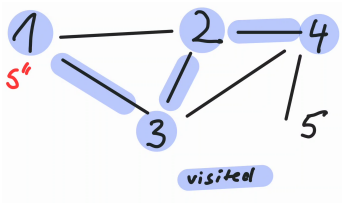


after L1-4: $S = (1)$

L5	L7 **(DFS-order)**	L9
$v = 1$ and $S = ()$	visited(1) =true	$S = (2, 3)$
$v = 3$ and $S = (2)$	visited(3) =true	$S = (2, 4, 2)$
$v = 2$ and $S = (2, 4)$	visited(2) =true	$S = (2, 4, 4)$

Depth-First Search (DFS)

```
DFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V
 2  init empty stack S //LIFO
 3  S.push(s)
 4  WHILE (S ≠ ∅) DO
 5      v := S.top() and S.pop()
 6      IF (visited(v)≠true) THEN
 7          visited(v):=true
 8          FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 9              S.push(s)
```

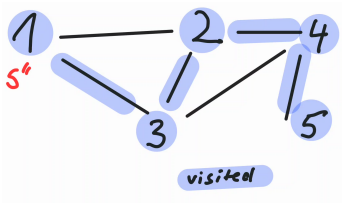


after L1-4: $S = (1)$

L5	L7 **(DFS-order)**	L9
$v = 1$ and $S = ()$	visited(1) =true	$S = (2, 3)$
$v = 3$ and $S = (2)$	visited(3) =true	$S = (2, 4, 2)$
$v = 2$ and $S = (2, 4)$	visited(2) =true	$S = (2, 4, 4)$
$v = 4$ and $S = (2, 4)$	visited(4) =true	$S = (2, 4, 5)$

Depth-First Search (DFS)

```
DFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V
 2  init empty stack S //LIFO
 3  S.push(s)
 4  WHILE (S ≠ ∅) DO
 5      v := S.top() and S.pop()
 6      IF (visited(v)≠true) THEN
 7          visited(v):=true
 8          FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 9              S.push(s)
```

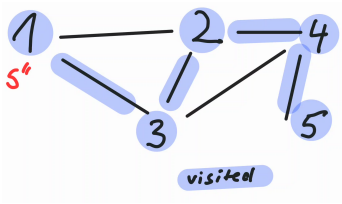


after L1-4: $S = (1)$

L5	L7 **(DFS-order)**	L9
$v = 1$ and $S = ()$	visited(1) =true	$S = (2, 3)$
$v = 3$ and $S = (2)$	visited(3) =true	$S = (2, 4, 2)$
$v = 2$ and $S = (2, 4)$	visited(2) =true	$S = (2, 4, 4)$
$v = 4$ and $S = (2, 4)$	visited(4) =true	$S = (2, 4, 5)$
$v = 5$ and $S = (2, 4)$	visited(5) =true	-

Depth-First Search (DFS)

```
DFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V
 2  init empty stack S //LIFO
 3  S.push(s)
 4  WHILE (S ≠ ∅) DO
 5     v := S.top() and S.pop()
 6     IF (visited(v)≠true) THEN
 7        visited(v):=true
 8        FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 9           S.push(s)
```

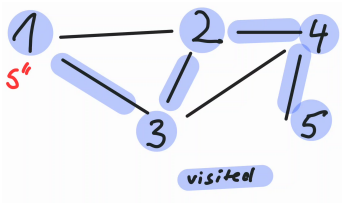


after L1-4: $S = (1)$

L5	L7 **(DFS-order)**	L9
$v = 1$ and $S = ()$	visited(1) =true	$S = (2, 3)$
$v = 3$ and $S = (2)$	visited(3) =true	$S = (2, 4, 2)$
$v = 2$ and $S = (2, 4)$	visited(2) =true	$S = (2, 4, 4)$
$v = 4$ and $S = (2, 4)$	visited(4) =true	$S = (2, 4, 5)$
$v = 5$ and $S = (2, 4)$	visited(5) =true	-
$v = 4$ and $S = (2)$	-	-

Depth-First Search (DFS)

```
DFS(G = (V, E), s)
1  visited(v):=false for all v ∈ V
2  init empty stack S //LIFO
3  S.push(s)
4  WHILE (S ≠ ∅) DO
5     v := S.top() and S.pop()
6     IF (visited(v)≠true) THEN
7        visited(v):=true
8        FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
9           S.push(s)
```

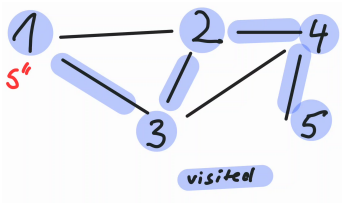


after L1-4: $S = (1)$

L5	L7 **(DFS-order)**	L9
$v = 1$ and $S = ()$	visited(1) =true	$S = (2, 3)$
$v = 3$ and $S = (2)$	visited(3) =true	$S = (2, 4, 2)$
$v = 2$ and $S = (2, 4)$	visited(2) =true	$S = (2, 4, 4)$
$v = 4$ and $S = (2, 4)$	visited(4) =true	$S = (2, 4, 5)$
$v = 5$ and $S = (2, 4)$	visited(5) =true	-
$v = 4$ and $S = (2)$	-	-
$v = 2$ and $S = ()$	-	-

Depth-First Search (DFS)

```
DFS(G = (V, E), s)
 1  visited(v):=false for all v ∈ V
 2  init empty stack S //LIFO
 3  S.push(s)
 4  WHILE (S ≠ ∅) DO
 5     v := S.top() and S.pop()
 6     IF (visited(v)≠true) THEN
 7        visited(v):=true
 8        FOR (all neighbors w ∈ N(v) of v for which visited(w)=false) DO
 9           S.push(s)
```



after L1-4: $S = (1)$

L5	L7 **(DFS-order)**	L9
$v = 1$ and $S = ()$	visited(1) =true	$S = (2, 3)$
$v = 3$ and $S = (2)$	visited(3) =true	$S = (2, 4, 2)$
$v = 2$ and $S = (2, 4)$	visited(2) =true	$S = (2, 4, 4)$
$v = 4$ and $S = (2, 4)$	visited(4) =true	$S = (2, 4, 5)$
$v = 5$ and $S = (2, 4)$	visited(5) =true	-
$v = 4$ and $S = (2)$	-	-
$v = 2$ and $S = ()$	-	-

DFS plays a particular role when considering directed graphs.

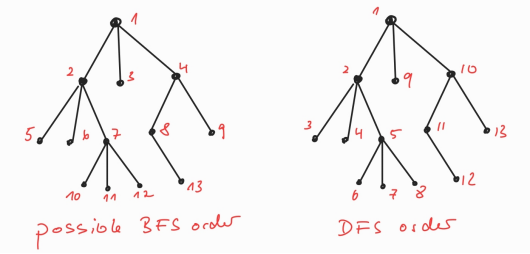
BFS and DFS

Both BFS and DFS graph traversal algorithms, that is:

they visit the vertices in a graph beginning with a start vertex *s* such that

Each vertex reachable from *s* is visited exactly once.
The next visited vertex always has at least one neighbor in the set of previously visited nodes.

BFS goes first into "breadth" while DFS goes first in "depth".



Neither the BFS- nor DFS-order is unique (e.g., we could have visited first 4 and then 3 in both trees)

Weighted Graphs and Minimum Spanning Trees

In many cases, we are interested in weighted graphs and minimum spanning trees.

Let (V, E) be a graph and $w : E \to \mathbb{R}$.

The triple $G = (V, E, w)$ is called a weighted (or edge-weighted) graph.

$w(e)$ is called the weight (or length) of edge $e \in E$.

Given: Weighted connected graph $G = (V, E, w)$.
Find: A minimum spanning tree (MST), i.e., a spanning tree $T = (V, F)$ with minimum total edge weight. That is, choose the set of edges $F \subseteq E$ such that

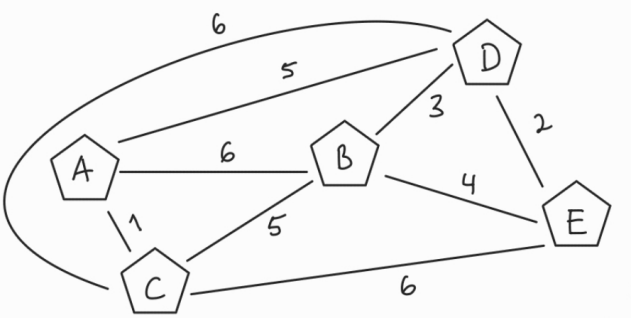
$$\sum_{e \in F} w(e)$$

is minimized.

Weighted Graphs and Minimum Spanning Trees

Example

Consider a scenario where a company needs to connect several buildings on its campus using cables or fiber optics. Each building is represented as a vertex, and the distances between the buildings are represented as edges with weights (the cost of laying cables/fiber).

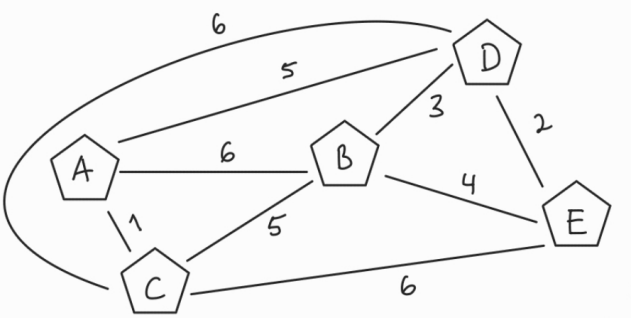


In this scenario, we want to connect all the buildings with the minimum cost possible. This is where an MST becomes useful which will give us the subset of edges that form a tree connecting all the nodes with the minimum total edge weight.

Weighted Graphs and Minimum Spanning Trees

Example

Consider a scenario where a company needs to connect several buildings on its campus using cables or fiber optics. Each building is represented as a vertex, and the distances between the buildings are represented as edges with weights (the cost of laying cables/fiber).



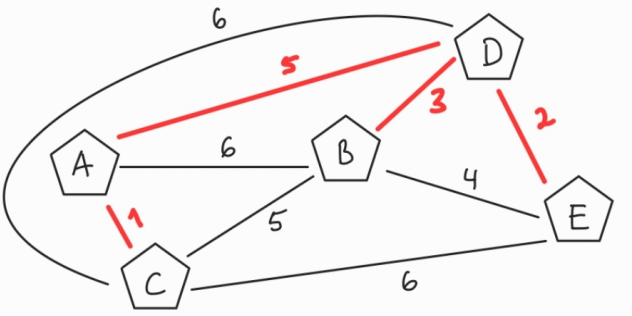
In this scenario, we want to connect all the buildings with the minimum cost possible. This is where an MST becomes useful which will give us the subset of edges that form a tree connecting all the nodes with the minimum total edge weight.

Question: What is a minimum spanning tree here?

Weighted Graphs and Minimum Spanning Trees

Example

Consider a scenario where a company needs to connect several buildings on its campus using cables or fiber optics. Each building is represented as a vertex, and the distances between the buildings are represented as edges with weights (the cost of laying cables/fiber).



In this scenario, we want to connect all the buildings with the minimum cost possible. This is where an MST becomes useful which will give us the subset of edges that form a tree connecting all the nodes with the minimum total edge weight.

Question: What is a minimum spanning tree here?

Weighted Graphs and Minimum Spanning Trees

Given: Weighted connected graph $G = (V, E, w)$.
Find: A minimum spanning tree (MST), i.e., a spanning tree $T = (V, F)$ such that $\sum_{e \in F} w(e)$ is minimized.

Question: Does BFS still work?

$\Longrightarrow$ we need a different approach.

Many optimization problems are rather difficult to solve (keyword: NP-hard).

However, somewhat surprisingly the MST problem can be solved in polynomial time with a simple greedy algorithm.

Question: What is a greedy algorithm?
Answer: A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

How does a greedy algorithm for finding an MST look like?

Weighted Graphs and Minimum Spanning Trees

Given: Weighted connected graph $G = (V, E, w)$.
Find: A minimum spanning tree (MST), i.e., a spanning tree $T = (V, F)$ such that $\sum_{e \in F} w(e)$ is minimized.

Question: Does BFS still work?
Answer: No, take graph on 3 vertices s, u, v and weights $w(\{s, v\}) = w(\{s, u\}) = 2$ and $w(\{u, v\}) = 1$.

$\Longrightarrow$ we need a different approach.

Many optimization problems are rather difficult to solve (keyword: NP-hard).

However, somewhat surprisingly the MST problem can be solved in polynomial time with a simple greedy algorithm.

Question: What is a greedy algorithm?
Answer: A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

How does a greedy algorithm for finding an MST look like?

Weighted Graphs and Minimum Spanning Trees

Given: Weighted connected graph $G = (V, E, w)$.
Find: A minimum spanning tree (MST), i.e., a spanning tree $T = (V, F)$ such that $\sum_{e \in F} w(e)$ is minimized.

Question: Does BFS still work?
Answer: No, take graph on 3 vertices s, u, v and weights $w(\{s, v\}) = w(\{s, u\}) = 2$ and $w(\{u, v\}) = 1$.

$\Longrightarrow$ we need a different approach.

Many optimization problems are rather difficult to solve (keyword: NP-hard).

However, somewhat surprisingly the MST problem can be solved in polynomial time with a simple greedy algorithm.

Question: What is a greedy algorithm?
Answer: A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

How does a greedy algorithm for finding an MST look like?

Weighted Graphs and Minimum Spanning Trees

Given: Weighted connected graph $G = (V, E, w)$.
Find: A minimum spanning tree (MST), i.e., a spanning tree $T = (V, F)$ such that $\sum_{e \in F} w(e)$ is minimized.

Question: Does BFS still work?
Answer: No, take graph on 3 vertices s, u, v and weights $w(\{s, v\}) = w(\{s, u\}) = 2$ and $w(\{u, v\}) = 1$.

$\Longrightarrow$ we need a different approach.

Many optimization problems are rather difficult to solve (keyword: NP-hard).

However, somewhat surprisingly the MST problem can be solved in polynomial time with a simple greedy algorithm.

Question: What is a greedy algorithm?
Answer: A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

How does a greedy algorithm for finding an MST look like?

Weighted Graphs and Minimum Spanning Trees

Given: Weighted connected graph $G = (V, E, w)$.
Find: A minimum spanning tree (MST), i.e., a spanning tree $T = (V, F)$ such that $\sum_{e \in F} w(e)$ is minimized.

Question: Does BFS still work?
Answer: No, take graph on 3 vertices s, u, v and weights $w(\{s, v\}) = w(\{s, u\}) = 2$ and $w(\{u, v\}) = 1$.

$\Longrightarrow$ we need a different approach.

Many optimization problems are rather difficult to solve (keyword: NP-hard).

However, somewhat surprisingly the MST problem can be solved in polynomial time with a simple greedy algorithm.

Question: What is a greedy algorithm?
Answer: A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

How does a greedy algorithm for finding an MST look like?

Weighted Graphs and Minimum Spanning Trees

Given: Weighted connected graph $G = (V, E, w)$.
Find: A minimum spanning tree (MST), i.e., a spanning tree $T = (V, F)$ such that $\sum_{e \in F} w(e)$ is minimized.

Question: Does BFS still work?
Answer: No, take graph on 3 vertices s, u, v and weights $w(\{s, v\}) = w(\{s, u\}) = 2$ and $w(\{u, v\}) = 1$.

$\Longrightarrow$ we need a different approach.

Many optimization problems are rather difficult to solve (keyword: NP-hard).

However, somewhat surprisingly the MST problem can be solved in polynomial time with a simple greedy algorithm.

Question: What is a greedy algorithm?
Answer: A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

How does a greedy algorithm for finding an MST look like?

Weighted Graphs and Minimum Spanning Trees

Given: Weighted connected graph $G = (V, E, w)$.
Find: A minimum spanning tree (MST), i.e., a spanning tree $T = (V, F)$ such that $\sum_{e \in F} w(e)$ is minimized.

Question: Does BFS still work?
Answer: No, take graph on 3 vertices s, u, v and weights $w(\{s, v\}) = w(\{s, u\}) = 2$ and $w(\{u, v\}) = 1$.

$\Longrightarrow$ we need a different approach.

Many optimization problems are rather difficult to solve (keyword: NP-hard).

However, somewhat surprisingly the MST problem can be solved in polynomial time with a simple greedy algorithm.

Question: What is a greedy algorithm?
Answer: A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

How does a greedy algorithm for finding an MST look like?

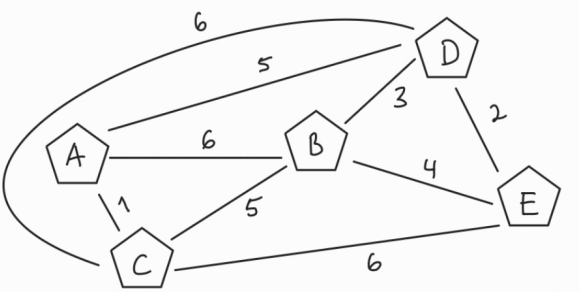
Kruskal's Algorithm

```
Kruskal(G = (V, E, w), w: E → ℝ) //m = |E|
1  Sort edges such that w(e_1) ≤ w(w_2) ··· ≤ w(e_m)
2  F := ∅, T := (V, F)
3  FOR i = 1, ..., m DO
3      IF (V, F ∪ {e_i}) is acyclic
3          F := F ∪ {e_i}
3  return T
```

Kruskal's algorithm finds the minimum spanning tree as follows: It starts by sorting edges by weight, then adds them one by one from lightest to heaviest while ensuring that the "intermediate" graph T remains a forest, until all vertices have been checked and when possible added to T

Kruskal's Algorithm

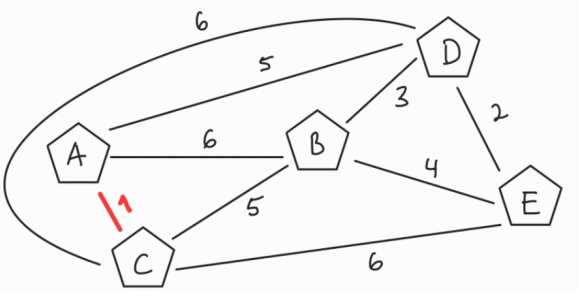
```
Kruskal(G = (V, E, w), w: E → ℝ) //m = |E|
1  Sort edges such that w(e_1) ≤ w(w_2) ··· ≤ w(e_m)
2  F := ∅, T := (V, F)
3  FOR i = 1, ..., m DO
3      IF (V, F ∪ {e_i}) is acyclic
3          F := F ∪ {e_i}
3  return T
```



$F = \{\}$

Kruskal's Algorithm

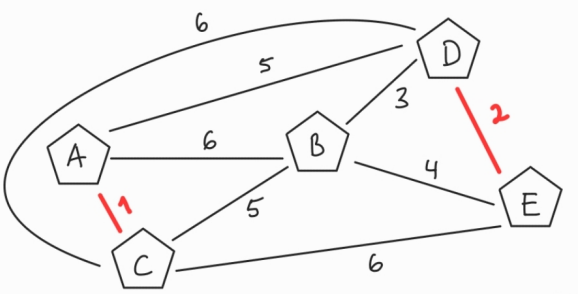
```
Kruskal(G = (V, E, w), w: E → ℝ) //m = |E|
1  Sort edges such that w(e_1) ≤ w(w_2) ··· ≤ w(e_m)
2  F := ∅, T := (V, F)
3  FOR i = 1, ..., m DO
3      IF (V, F ∪ {e_i}) is acyclic
3          F := F ∪ {e_i}
3  return T
```



$F = \{\{A, C\}\}$

Kruskal's Algorithm

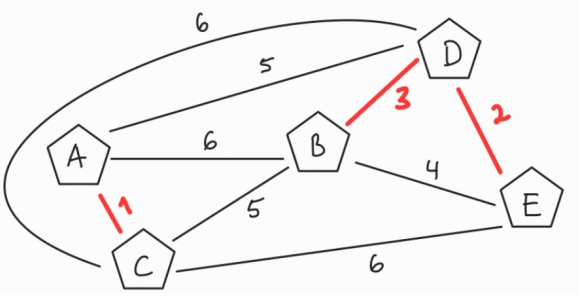
```
Kruskal(G = (V, E, w), w: E → ℝ) //m = |E|
1  Sort edges such that w(e_1) ≤ w(w_2) ··· ≤ w(e_m)
2  F := ∅, T := (V, F)
3  FOR i = 1, ..., m DO
3      IF (V, F ∪ {e_i}) is acyclic
3          F := F ∪ {e_i}
3  return T
```



$F = \{\{A, C\}, \{D, E\}\}$

Kruskal's Algorithm

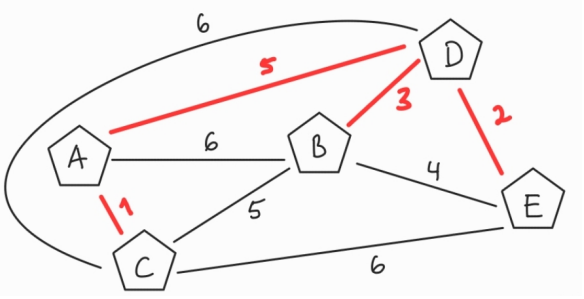
```
Kruskal(G = (V, E, w), w: E → ℝ) //m = |E|
1  Sort edges such that w(e_1) ≤ w(w_2) ··· ≤ w(e_m)
2  F := ∅, T := (V, F)
3  FOR i = 1, ..., m DO
3      IF (V, F ∪ {e_i}) is acyclic
3          F := F ∪ {e_i}
3  return T
```



$F = \{\{A, C\}, \{D, E\}, \{B, D\}\}$

Kruskal's Algorithm

```
Kruskal(G = (V, E, w), w: E → ℝ) //m = |E|
1  Sort edges such that w(e_1) ≤ w(w_2) ··· ≤ w(e_m)
2  F := ∅, T := (V, F)
3  FOR i = 1, ..., m DO
3      IF (V, F ∪ {e_i}) is acyclic
3          F := F ∪ {e_i}
3  return T
```



$F = \{\{A, C\}, \{D, E\}, \{B, D\}, \{A, D\}\}$

Kruskal's Algorithm

`Kruskal`$(G = (V, E, w), w\colon E \to \mathbb{R})$ //$m = |E|$

1 Sort edges such that $w(e_1) \leq w(w_2) \cdots \leq w(e_m)$
2 $F := \emptyset$, $T := (V, F)$
3 `FOR` $i = 1, \ldots, m$ `DO`
3 `IF` $(V, F \cup \{e_i\})$ is acyclic
3 $F := F \cup \{e_i\}$
3 `return` T

Theorem. *`Kruskal` correctly computes an MST for a given undirected, connected graph $G = (V, E)$ in $O(|E||V|)$ time.*

Proof Board.