

Datenstrukturen und Algorithmen

Vorlesung 14: Elementare Graphenalgorithmen I

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

<http://www-i2.rwth-aachen.de/i2/dsa110/>

15. Juni 2010

Übersicht

1 Graphen

- Terminologie
- Repräsentation von Graphen

2 Graphendurchlauf

- Breitensuche
- Tiefensuche
- Finden von Zusammenhangskomponenten

Übersicht

1 Graphen

- Terminologie
- Repräsentation von Graphen

2 Graphendurchlauf

- Breitensuche
- Tiefensuche
- Finden von Zusammenhangskomponenten

Die Bedeutung von Graphen

Graphen werden in vielen (Informatik-)Anwendungen verwendet:

Beispiele

- ▶ (Computer-)Netzwerke

Die Bedeutung von Graphen

Graphen werden in vielen (Informatik-)Anwendungen verwendet:

Beispiele

- ▶ (Computer-)Netzwerke
- ▶ Darstellung von topologischen Informationen (Karten, ...)

Die Bedeutung von Graphen

Graphen werden in vielen (Informatik-)Anwendungen verwendet:

Beispiele

- ▶ (Computer-)Netzwerke
- ▶ Darstellung von topologischen Informationen (Karten, ...)
- ▶ Darstellung von elektronischen Schaltungen

Die Bedeutung von Graphen

Graphen werden in vielen (Informatik-)Anwendungen verwendet:

Beispiele

- ▶ (Computer-)Netzwerke
- ▶ Darstellung von topologischen Informationen (Karten, ...)
- ▶ Darstellung von elektronischen Schaltungen
- ▶ Vorranggraphen (precedence graph), Ablaufpläne, ...

Die Bedeutung von Graphen

Graphen werden in vielen (Informatik-)Anwendungen verwendet:

Beispiele

- ▶ (Computer-)Netzwerke
- ▶ Darstellung von topologischen Informationen (Karten, ...)
- ▶ Darstellung von elektronischen Schaltungen
- ▶ Vorranggraphen (precedence graph), Ablaufpläne, ...
- ▶ Semantische Netze (z. B. Entity-Relationship-Diagramme)

Die Bedeutung von Graphen

Graphen werden in vielen (Informatik-)Anwendungen verwendet:

Beispiele

- ▶ (Computer-)Netzwerke
- ▶ Darstellung von topologischen Informationen (Karten, ...)
- ▶ Darstellung von elektronischen Schaltungen
- ▶ Vorranggraphen (precedence graph), Ablaufpläne, ...
- ▶ Semantische Netze (z. B. Entity-Relationship-Diagramme)

Wir werden uns auf fundamentale Graphalgorithmen konzentrieren.

Was ist ein gerichteter Graph? (I)

Gerichteter Graph

Ein **gerichteter Graph** (auch: **Digraph**) G ist ein Paar (V, E)

Was ist ein gerichteter Graph? (I)

Gerichteter Graph

Ein **gerichteter Graph** (auch: **Digraph**) G ist ein Paar (V, E) mit

- ▶ einer Menge Knoten (vertices) V und

Was ist ein gerichteter Graph? (I)

Gerichteter Graph

Ein **gerichteter Graph** (auch: **Digraph**) G ist ein Paar (V, E) mit

- ▶ einer Menge Knoten (vertices) V und
- ▶ einer Menge (*geordneter*) Paare von Knoten $E \subseteq V \times V$, die (gerichtete) Kanten (edges) genannt werden.

Was ist ein gerichteter Graph? (I)

Gerichteter Graph

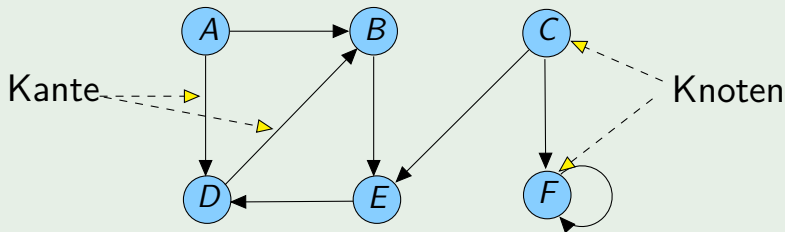
Ein **gerichteter Graph** (auch: **Digraph**) G ist ein Paar (V, E) mit

- ▶ einer Menge Knoten (vertices) V und
- ▶ einer Menge (*geordneter*) Paare von Knoten $E \subseteq V \times V$, die (gerichtete) Kanten (edges) genannt werden.
- ▶ Falls E eine Menge ungeordneter Paare ist, heißt G **ungerichtet**.

Was ist ein gerichteter Graph? (I)

Beispiel

- ▶ $V = \{ A, \dots, F \}$
- ▶ $E = \{ (A, B), (A, D), (B, E), (C, E), (C, F), (D, B), (E, D), (F, F) \}$



Terminologie bei Graphen (I)

Teilgraph

Ein **Teilgraph** (subgraph) eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit:

Terminologie bei Graphen (I)

Teilgraph

Ein **Teilgraph** (subgraph) eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit:

- ▶ $V' \subseteq V$ und $E' \subseteq E$.

Terminologie bei Graphen (I)

Teilgraph

Ein **Teilgraph** (subgraph) eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit:

- ▶ $V' \subseteq V$ und $E' \subseteq E$.
- ▶ Außerdem ist $E' \subseteq V' \times V'$ wegen der Grapheigenschaft von G' .

Terminologie bei Graphen (I)

Teilgraph

Ein **Teilgraph** (subgraph) eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit:

- ▶ $V' \subseteq V$ und $E' \subseteq E$.
- ▶ Außerdem ist $E' \subseteq V' \times V'$ wegen der Grapheigenschaft von G' .
- ▶ Ist $V' \subset V$ und $E' \subset E$, so heißt G' **echter** (proper) Teilgraph.

Terminologie bei Graphen (I)

Teilgraph

Ein **Teilgraph** (subgraph) eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit:

- ▶ $V' \subseteq V$ und $E' \subseteq E$.
- ▶ Außerdem ist $E' \subseteq V' \times V'$ wegen der Grapheigenschaft von G' .
- ▶ Ist $V' \subset V$ und $E' \subset E$, so heißt G' **echter** (proper) Teilgraph.

Symmetrischer Graph

Der Graph G heißt **symmetrisch**, wenn aus $(v, w) \in E$ folgt $(w, v) \in E$.

Terminologie bei Graphen (I)

Teilgraph

Ein **Teilgraph** (subgraph) eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit:

- ▶ $V' \subseteq V$ und $E' \subseteq E$.
- ▶ Außerdem ist $E' \subseteq V' \times V'$ wegen der Grapheigenschaft von G' .
- ▶ Ist $V' \subset V$ und $E' \subset E$, so heißt G' **echter** (proper) Teilgraph.

Symmetrischer Graph

Der Graph G heißt **symmetrisch**, wenn aus $(v, w) \in E$ folgt $(w, v) \in E$.

- ▶ Zu jedem ungerichteten Graphen gibt es korrespondierenden symmetrischen Digraphen.

Terminologie bei Graphen (II)

Vollständiger Graph

Der Graph G ist **vollständig**, wenn *jedes* Paar von Knoten mit einer Kante verbunden ist.

Terminologie bei Graphen (II)

Vollständiger Graph

Der Graph G ist **vollständig**, wenn *jedes* Paar von Knoten mit einer Kante verbunden ist.

Adjazent

Knoten w ist **adjazent** zu v , wenn $(v, w) \in E$.

Terminologie bei Graphen (II)

Vollständiger Graph

Der Graph G ist **vollständig**, wenn *jedes* Paar von Knoten mit einer Kante verbunden ist.

Adjazent

Knoten w ist **adjazent** zu v , wenn $(v, w) \in E$.

Transponieren

Transponiert man G (transpose graph), so erhält man $G^T = (V, E')$ mit $(v, w) \in E'$ gdw. $(w, v) \in E$.

Terminologie bei Graphen (II)

Vollständiger Graph

Der Graph G ist **vollständig**, wenn *jedes* Paar von Knoten mit einer Kante verbunden ist.

Adjazent

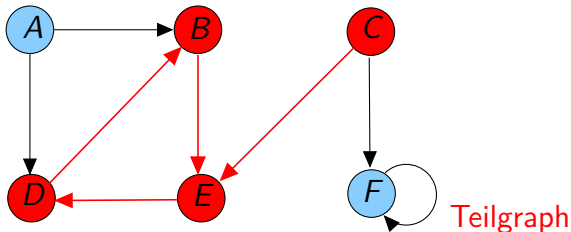
Knoten w ist **adjazent** zu v , wenn $(v, w) \in E$.

Transponieren

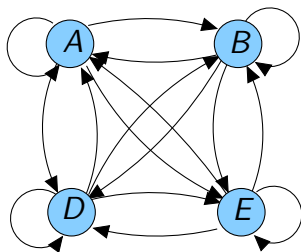
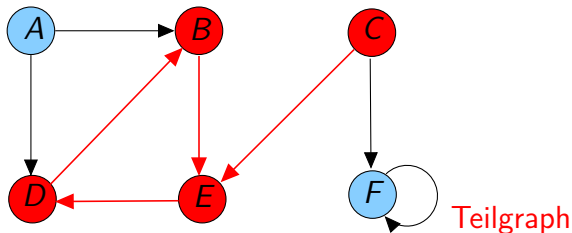
Transponiert man G (transpose graph), so erhält man $G^T = (V, E')$ mit $(v, w) \in E'$ gdw. $(w, v) \in E$.

- In G' ist die Richtung der Kanten von G gerade umgedreht.

Terminologie bei Graphen (III)



Terminologie bei Graphen (III)



Vollständiger (und symmetrischer) Digraph
auf vier Knoten

Pfade und Zyklen (I)

Weg, Pfad

Ein **Weg** von Knoten v nach w ist eine Folge von Kanten (v_i, v_{i+1}) :

$v_0 v_1 v_2 \dots v_{k-1} v_k$, so dass $v_0 = v$ und $v_k = w$

Pfade und Zyklen (I)

Weg, Pfad

Ein **Weg** von Knoten v nach w ist eine Folge von Kanten (v_i, v_{i+1}) :

$$v_0 v_1 v_2 \dots v_{k-1} v_k, \text{ so dass } v_0 = v \text{ und } v_k = w$$

Ein Weg, bei dem alle $v_i \neq v_j$ für $i \neq j$ verschieden sind, heißt **Pfad**.

Pfade und Zyklen (I)

Weg, Pfad

Ein **Weg** von Knoten v nach w ist eine Folge von Kanten (v_i, v_{i+1}) :

$$v_0 v_1 v_2 \dots v_{k-1} v_k, \text{ so dass } v_0 = v \text{ und } v_k = w$$

Ein Weg, bei dem alle $v_i \neq v_j$ für $i \neq j$ verschieden sind, heißt **Pfad**.

- Länge eines Pfades(Weges) ist die Anzahl der durchlaufenen Kanten.

Pfade und Zyklen (I)

Weg, Pfad

Ein **Weg** von Knoten v nach w ist eine Folge von Kanten (v_i, v_{i+1}) :

$$v_0 v_1 v_2 \dots v_{k-1} v_k, \text{ so dass } v_0 = v \text{ und } v_k = w$$

Ein Weg, bei dem alle $v_i \neq v_j$ für $i \neq j$ verschieden sind, heißt **Pfad**.

- Länge eines Pfades(Weges) ist die Anzahl der durchlaufenen Kanten.

Erreichbarkeit

Knoten w heißt **erreichbar** von v , wenn es einen Pfad von v nach w gibt.

Pfade und Zyklen (I)

Weg, Pfad

Ein **Weg** von Knoten v nach w ist eine Folge von Kanten (v_i, v_{i+1}) :

$$v_0 v_1 v_2 \dots v_{k-1} v_k, \text{ so dass } v_0 = v \text{ und } v_k = w$$

Ein Weg, bei dem alle $v_i \neq v_j$ für $i \neq j$ verschieden sind, heißt **Pfad**.

- Länge eines Pfades(Weges) ist die Anzahl der durchlaufenen Kanten.

Erreichbarkeit

Knoten w heißt **erreichbar** von v , wenn es einen Pfad von v nach w gibt.

Zyklus

Ein **Zyklus** ist ein nicht-leerer Weg bei dem der Startknoten auch Endknoten ist.

Pfade und Zyklen (I)

Weg, Pfad

Ein **Weg** von Knoten v nach w ist eine Folge von Kanten (v_i, v_{i+1}) :

$$v_0 v_1 v_2 \dots v_{k-1} v_k, \text{ so dass } v_0 = v \text{ und } v_k = w$$

Ein Weg, bei dem alle $v_i \neq v_j$ für $i \neq j$ verschieden sind, heißt **Pfad**.

- Länge eines Pfades(Weges) ist die Anzahl der durchlaufenen Kanten.

Erreichbarkeit

Knoten w heißt **erreichbar** von v , wenn es einen Pfad von v nach w gibt.

Zyklus

Ein **Zyklus** ist ein nicht-leerer Weg bei dem der Startknoten auch Endknoten ist.

- Ein Zyklus der Form vv heißt Schleife (loop, self-cycle).

Pfade und Zyklen (I)

Weg, Pfad

Ein **Weg** von Knoten v nach w ist eine Folge von Kanten (v_i, v_{i+1}) :

$$v_0 v_1 v_2 \dots v_{k-1} v_k, \text{ so dass } v_0 = v \text{ und } v_k = w$$

Ein Weg, bei dem alle $v_i \neq v_j$ für $i \neq j$ verschieden sind, heißt **Pfad**.

- ▶ Länge eines Pfades(Weges) ist die Anzahl der durchlaufenen Kanten.

Erreichbarkeit

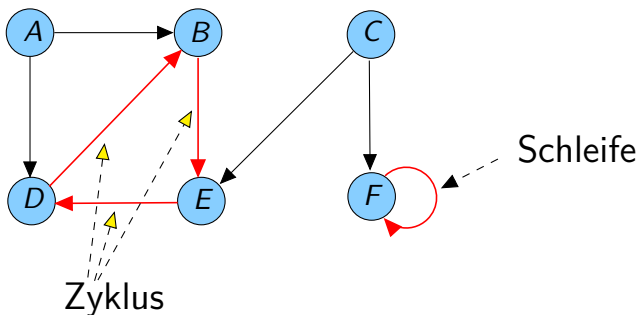
Knoten w heißt **erreichbar** von v , wenn es einen Pfad von v nach w gibt.

Zyklus

Ein **Zyklus** ist ein nicht-leerer Weg bei dem der Startknoten auch Endknoten ist.

- ▶ Ein Zyklus der Form vv heißt Schleife (loop, self-cycle).
- ▶ Ein Graph ist **azyklisch**, wenn er keine Zyklen hat.

Pfade und Zyklen (II)



$ABEDB$ und $CF F$ wären hier Beispiele für Wege.

EDB und CF sind Pfade.

Zusammenhängende Graphen (I)

Beim **ungerichteten** Graphen G :

Zusammenhang

- ▶ G heißt **zusammenhängend**, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.

Zusammenhängende Graphen (I)

Beim **ungerichteten** Graphen G :

Zusammenhang

- ▶ G heißt **zusammenhängend**, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- ▶ Eine **Zusammenhangskomponente** (connected component) von G ist ein maximaler zusammenhängender Teilgraph von G .

Zusammenhängende Graphen (I)

Beim **ungerichteten** Graphen G :

Zusammenhang

- ▶ G heißt **zusammenhängend**, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- ▶ Eine **Zusammenhangskomponente** (connected component) von G ist ein maximaler zusammenhängender Teilgraph von G .
- ▶ In einer Ansammlung von Graphen heißt ein Graph **maximal**, wenn er von keinem anderen dieser Graphen ein echter Teilgraph ist.

Zusammenhängende Graphen (II)

Beim **gerichteten** Graphen G :

Zusammenhang

- ▶ G heißt **stark zusammenhängend** (strongly connected), wenn jeder Knoten von jedem anderen aus erreichbar ist.

Zusammenhängende Graphen (II)

Beim **gerichteten** Graphen G :

Zusammenhang

- ▶ G heißt **stark zusammenhängend** (strongly connected), wenn jeder Knoten von jedem anderen aus erreichbar ist.
- ▶ G heißt **schwach zusammenhängend**, wenn der zugehörige ungerichtete Graph (wenn man alle Kanten ungerichtet macht) zusammenhängend ist.

Zusammenhängende Graphen (II)

Beim **gerichteten** Graphen G :

Zusammenhang

- ▶ G heißt **stark zusammenhängend** (strongly connected), wenn jeder Knoten von jedem anderen aus erreichbar ist.
- ▶ G heißt **schwach zusammenhängend**, wenn der zugehörige ungerichtete Graph (wenn man alle Kanten ungerichtet macht) zusammenhängend ist.
- ▶ Eine **starke Zusammenhangskomponente** von G ist ein maximaler stark zusammenhängender Teilgraph von G .

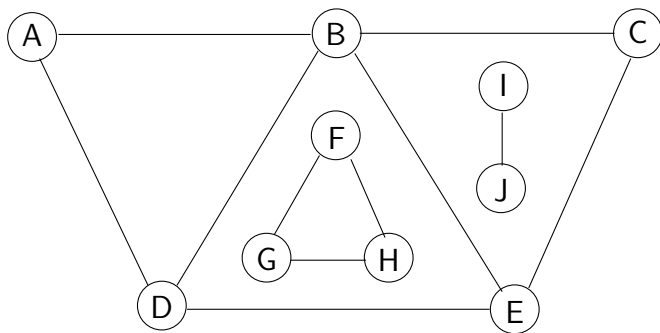
Zusammenhängende Graphen (II)

Beim **gerichteten** Graphen G :

Zusammenhang

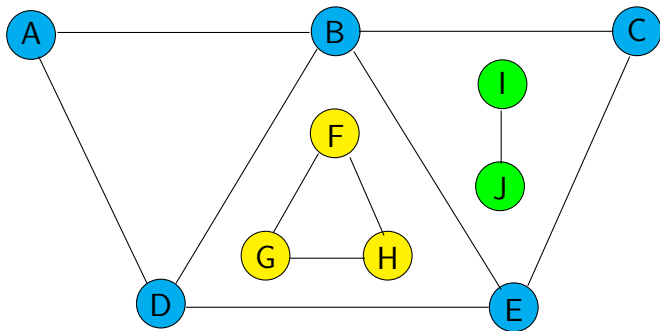
- ▶ G heißt **stark zusammenhängend** (strongly connected), wenn jeder Knoten von jedem anderen aus erreichbar ist.
- ▶ G heißt **schwach zusammenhängend**, wenn der zugehörige ungerichtete Graph (wenn man alle Kanten ungerichtet macht) zusammenhängend ist.
- ▶ Eine **starke Zusammenhangskomponente** von G ist ein maximaler stark zusammenhängender Teilgraph von G .
- ▶ Ein nicht-verbundener Graph kann **eindeutig** in verschiedene Zusammenhangskomponenten **aufgeteilt** werden.

Ungerichtete, zusammenhängende Graphen



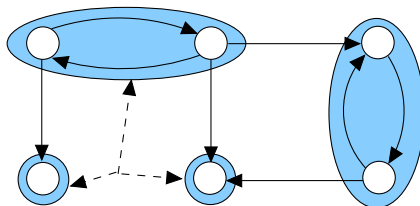
Ein ungerichteter Graph; Was sind die Zusammenhangskomponenten?

Ungerichtete, zusammenhängende Graphen



Die Zusammenhangskomponenten.

Starke Zusammenhangskomponenten



Zusammenhangskomponenten

Ein nicht-zusammenhängender Digraph, aufgeteilt in seine maximalen zusammenhängenden Teilgraphen.

Repräsentation von Graphen – Adjazenzmatrix

Sei $G = (V, E)$ mit $|V| = n$, $|E| = m$ und $V = \{v_1, \dots, v_n\}$.

Repräsentation von Graphen – Adjazenzmatrix

Sei $G = (V, E)$ mit $|V| = n$, $|E| = m$ und $V = \{v_1, \dots, v_n\}$.

Adjazenzmatrix

Die **Adjazenzmatrix**-Darstellung eines Graphen ist durch eine $n \times n$ Matrix A gegeben, wobei $A(i, j) = 1$, wenn $(v_i, v_j) \in E$, sonst 0.

Repräsentation von Graphen – Adjazenzmatrix

Sei $G = (V, E)$ mit $|V| = n$, $|E| = m$ und $V = \{v_1, \dots, v_n\}$.

Adjazenzmatrix

Die **Adjazenzmatrix**-Darstellung eines Graphen ist durch eine $n \times n$ Matrix A gegeben, wobei $A(i, j) = 1$, wenn $(v_i, v_j) \in E$, sonst 0.

- ▶ Wenn G ungerichtet ist, ergibt sich symmetrisches A (d. h. $A = A^T$). Dann muss nur die Hälfte gespeichert werden.

Repräsentation von Graphen – Adjazenzmatrix

Sei $G = (V, E)$ mit $|V| = n$, $|E| = m$ und $V = \{v_1, \dots, v_n\}$.

Adjazenzmatrix

Die **Adjazenzmatrix**-Darstellung eines Graphen ist durch eine $n \times n$ Matrix A gegeben, wobei $A(i, j) = 1$, wenn $(v_i, v_j) \in E$, sonst 0.

- ▶ Wenn G ungerichtet ist, ergibt sich symmetrisches A (d. h. $A = A^T$).
Dann muss nur die Hälfte gespeichert werden.
- ⇒ Platzbedarf: $\Theta(n^2)$.

Repräsentation von Graphen – Adjazenzliste

Adjazenzliste

Bei der Darstellung als **Array von Adjazenzlisten** gibt es ein durch die Nummer des Knoten indiziertes Array, das jeweils verkettete Listen (Adjazenzlisten) enthält.

- ▶ Der i -te Arrayeintrag enthält alle Kanten von G , die von v_i „ausgehen“.

Repräsentation von Graphen – Adjazenzliste

Adjazenzliste

Bei der Darstellung als **Array von Adjazenzlisten** gibt es ein durch die Nummer des Knoten indiziertes Array, das jeweils verkettete Listen (Adjazenzlisten) enthält.

- ▶ Der i -te Arrayeintrag enthält alle Kanten von G , die von v_i „ausgehen“.
- ▶ Ist G ungerichtet, dann werden Kanten doppelt gespeichert.

Repräsentation von Graphen – Adjazenzliste

Adjazenzliste

Bei der Darstellung als **Array von Adjazenzlisten** gibt es ein durch die Nummer des Knoten indiziertes Array, das jeweils verkettete Listen (Adjazenzlisten) enthält.

- ▶ Der i -te Arrayeintrag enthält alle Kanten von G , die von v_i „ausgehen“.
- ▶ Ist G ungerichtet, dann werden Kanten doppelt gespeichert.
- ▶ Kanten, die in G nicht vorkommen, benötigen keinen Speicherplatz.

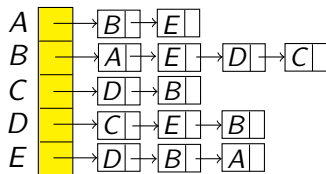
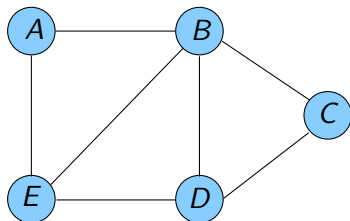
Repräsentation von Graphen – Adjazenzliste

Adjazenzliste

Bei der Darstellung als **Array von Adjazenzlisten** gibt es ein durch die Nummer des Knoten indiziertes Array, das jeweils verkettete Listen (Adjazenzlisten) enthält.

- ▶ Der i -te Arrayeintrag enthält alle Kanten von G , die von v_i „ausgehen“.
 - ▶ Ist G ungerichtet, dann werden Kanten doppelt gespeichert.
 - ▶ Kanten, die in G nicht vorkommen, benötigen keinen Speicherplatz.
- ⇒ Platzbedarf: $\Theta(n + m)$.

Darstellung eines ungerichteten Graphen

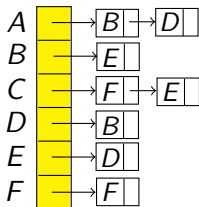
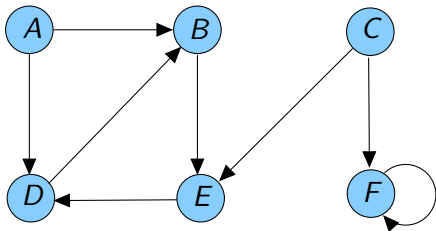


Adjazenzliste

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Adjazenzmatrix

Darstellung eines gerichteten Graphen



Adjazenzliste

$$\begin{bmatrix}
 0 & 1 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 1 & 1 \\
 0 & 1 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 1
 \end{bmatrix}$$

Adjazenzmatrix

Übersicht

1 Graphen

- Terminologie
- Repräsentation von Graphen

2 Graphendurchlauf

- Breitensuche
- Tiefensuche
- Finden von Zusammenhangskomponenten

Graphendurchlauf (I)

Viele Algorithmen untersuchen jeden Knoten (und jede Kante).

Graphendurchlauf (I)

Viele Algorithmen untersuchen jeden Knoten (und jede Kante).

Es gibt verschiedene **Graphendurchlaufstrategien** (traversal strategies), die jeden Knoten (oder jede Kante) genau einmal besuchen:

- ▶ **Tiefensuche**
- ▶ **Breitensuche**

Graphendurchlauf (I)

Viele Algorithmen untersuchen jeden Knoten (und jede Kante).

Es gibt verschiedene **Graphendurchlaufstrategien** (traversal strategies), die jeden Knoten (oder jede Kante) genau einmal besuchen:

- ▶ **Tiefensuche**
- ▶ **Breitensuche**
- ▶ Es handelt sich um Verallgemeinerungen von Strategien zur Baumtraversierung.

Graphendurchlauf (I)

Viele Algorithmen untersuchen jeden Knoten (und jede Kante).

Es gibt verschiedene **Graphendurchlaufstrategien** (traversal strategies), die jeden Knoten (oder jede Kante) genau einmal besuchen:

- ▶ **Tiefensuche**
- ▶ **Breitensuche**
- ▶ Es handelt sich um Verallgemeinerungen von Strategien zur Baumtraversierung.
- ▶ Nun müssen wir uns aber alle bereits gefundenen Knoten merken.

Graphendurchlauf (I)

Viele Algorithmen untersuchen jeden Knoten (und jede Kante).

Es gibt verschiedene **Graphendurchlaufstrategien** (traversal strategies), die jeden Knoten (oder jede Kante) genau einmal besuchen:

- ▶ **Tiefensuche**
- ▶ **Breitensuche**
- ▶ Es handelt sich um Verallgemeinerungen von Strategien zur Baumtraversierung.
- ▶ Nun müssen wir uns aber alle bereits gefundenen Knoten merken.
- ▶ Algorithmen auf dieser Basis sind in $O(|V| + |E|)$.

Graphendurchlauf (II)

Beispiele

- ▶ Finden von (starken) Zusammenhangskomponenten,
- ▶ Topologische Sortierung,
- ▶ Kritische-Pfad-Analyse,
- ▶ Finden von 2-Zusammenhangskomponenten (biconnected components),
- ▶ ... und viele weitere ...

Breitensuche

Breitensuche (Breadth-First Search, BFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert.
Die zugrundeliegende Strategie ist:

Breitensuche

Breitensuche (Breadth-First Search, BFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).

Breitensuche

Breitensuche (Breadth-First Search, BFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :

Breitensuche

Breitensuche (Breadth-First Search, BFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche „gleichzeitig“ von allen solchen w aus weiter.

Breitensuche

Breitensuche (Breadth-First Search, BFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche „gleichzeitig“ von allen solchen w aus weiter.
 - ▶ **Kein** Backtracking.

Breitensuche

Breitensuche (Breadth-First Search, BFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche „gleichzeitig“ von allen solchen w aus weiter.
 - ▶ **Kein** Backtracking.
- ▶ Markiere Knoten v als „abgeschlossen“ (BLACK).

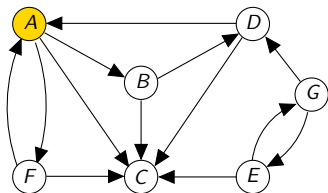
Breitensuche

Breitensuche (Breadth-First Search, BFS)

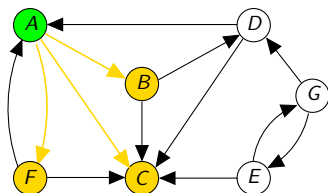
Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
 - ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche „gleichzeitig“ von allen solchen w aus weiter.
 - ▶ **Kein** Backtracking.
 - ▶ Markiere Knoten v als „abgeschlossen“ (BLACK).
- ▶ Man erhält die Menge aller Knoten, die vom Startknoten aus **erreichbar** sind.

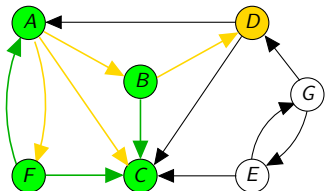
Breitensuche – Beispiel



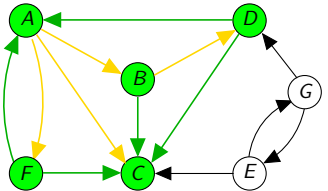
Beginn der Breitensuche



Erforsche alle folgenden nicht-gefundenen Knoten



Erforsche alle folgenden nicht-gefundenen Knoten



Fertig!

Breitensuche – Implementierung

```
1 void bfsSearch(List adjList[n], int n, int start) {
2     int color[n];
3     Queue wait; // zu verarbeitende Knoten
4     for (int i = 0; i < n; i++) {
5         color[i] = WHITE; // noch nicht gefunden
6     }
7     color[start] = GRAY; // start ist noch zu verarbeiten
8     wait.enqueue(start);
9     while (!wait.isEmpty()) {
10        // nächster noch unverarbeiteter Knoten
11        int v = wait.dequeue();
12        foreach (w in adjList[v]) {
13            if (color[w] == WHITE) { // neuer ("ungefundener") Knoten
14                color[w] = GRAY; // w ist noch zu verarbeiten
15                wait.enqueue(w);
16            }
17        }
18        color[v] = BLACK; // v ist abgeschlossen
19    }
20 }
```

Eigenschaften der Breitensuche

- ▶ Knoten werden in der Reihenfolge mit **zunehmenden Abstand** vom Startknoten aus besucht.

Eigenschaften der Breitensuche

- ▶ Knoten werden in der Reihenfolge mit **zunehmenden Abstand** vom Startknoten aus besucht.
 - ▶ Nachdem alle Knoten mit Abstand d verarbeitet wurden, werden die mit $d + 1$ angegangen.

Eigenschaften der Breitensuche

- ▶ Knoten werden in der Reihenfolge mit **zunehmenden Abstand** vom Startknoten aus besucht.
 - ▶ Nachdem alle Knoten mit Abstand d verarbeitet wurden, werden die mit $d + 1$ angegangen.
 - ▶ Die Suche terminiert, wenn in Abstand d keine Knoten auftreten.

Eigenschaften der Breitensuche

- ▶ Knoten werden in der Reihenfolge mit **zunehmenden Abstand** vom Startknoten aus besucht.
 - ▶ Nachdem alle Knoten mit Abstand d verarbeitet wurden, werden die mit $d + 1$ angegangen.
 - ▶ Die Suche terminiert, wenn in Abstand d keine Knoten auftreten.
- ▶ Die Tiefe des Knotens v im Breitensuchbaum ist seine **kürzeste Kantendistanz** zum Startknoten.

Eigenschaften der Breitensuche

- ▶ Knoten werden in der Reihenfolge mit **zunehmenden Abstand** vom Startknoten aus besucht.
 - ▶ Nachdem alle Knoten mit Abstand d verarbeitet wurden, werden die mit $d + 1$ angegangen.
 - ▶ Die Suche terminiert, wenn in Abstand d keine Knoten auftreten.
- ▶ Die Tiefe des Knotens v im Breitensuchbaum ist seine **kürzeste Kantendistanz** zum Startknoten.
- ▶ Die zu verarbeitenden Knoten werden als **FIFO-Queue** (first-in first-out) organisiert.

Eigenschaften der Breitensuche

- ▶ Knoten werden in der Reihenfolge mit **zunehmenden Abstand** vom Startknoten aus besucht.
 - ▶ Nachdem alle Knoten mit Abstand d verarbeitet wurden, werden die mit $d + 1$ angegangen.
 - ▶ Die Suche terminiert, wenn in Abstand d keine Knoten auftreten.
- ▶ Die Tiefe des Knotens v im Breitensuchbaum ist seine **kürzeste Kantendistanz** zum Startknoten.
- ▶ Die zu verarbeitenden Knoten werden als **FIFO-Queue** (first-in first-out) organisiert.
- ▶ Es gibt **eine einzige** „Verarbeitungsmöglichkeit“ für v , nämlich, wenn es aus der Queue entnommen wird.

Eigenschaften der Breitensuche

- ▶ Knoten werden in der Reihenfolge mit **zunehmenden Abstand** vom Startknoten aus besucht.
 - ▶ Nachdem alle Knoten mit Abstand d verarbeitet wurden, werden die mit $d + 1$ angegangen.
 - ▶ Die Suche terminiert, wenn in Abstand d keine Knoten auftreten.
- ▶ Die Tiefe des Knotens v im Breitensuchbaum ist seine **kürzeste Kantendistanz** zum Startknoten.
- ▶ Die zu verarbeitenden Knoten werden als **FIFO-Queue** (first-in first-out) organisiert.
- ▶ Es gibt **eine einzige** „Verarbeitungsmöglichkeit“ für v , nämlich, wenn es aus der Queue entnommen wird.

Theorem (Komplexität der Breitensuche)

Die Zeitkomplexität ist $\Theta(|V| + |E|)$, der Platzbedarf $\Theta(|V|)$.

Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert.

Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert.
Die zugrundeliegende Strategie ist:

Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert.
Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).

Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert.
Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :

Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert.
Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche rekursiv von w aus, d. h.:

Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert.
Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche rekursiv von w aus, d. h.:
 - ▶ Erforsche Kante (v, w) , besuche w , forsche von dort aus, bis es nicht mehr weiter geht.

Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche rekursiv von w aus, d. h.:
 - ▶ Erforsche Kante (v, w) , besuche w , forsche von dort aus, bis es nicht mehr weiter geht.
 - ▶ Dann backtracke von w nach v .

Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche rekursiv von w aus, d. h.:
 - ▶ Erforsche Kante (v, w) , besuche w , forsche von dort aus, bis es nicht mehr weiter geht.
 - ▶ Dann backtracke von w nach v .
- ▶ Für jede Kante (v, w) in G mit gefundenem Nachfolger w :

Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche rekursiv von w aus, d. h.:
 - ▶ Erforsche Kante (v, w) , besuche w , forsche von dort aus, bis es nicht mehr weiter geht.
 - ▶ Dann backtracke von w nach v .
- ▶ Für jede Kante (v, w) in G mit gefundenem Nachfolger w :
 - ▶ „Überprüfe“ die Kante, ohne aber w zu besuchen.

Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert.
Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
- ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche rekursiv von w aus, d. h.:
 - ▶ Erforsche Kante (v, w) , besuche w , forsche von dort aus, bis es nicht mehr weiter geht.
 - ▶ Dann backtracke von w nach v .
- ▶ Für jede Kante (v, w) in G mit gefundenem Nachfolger w :
 - ▶ „Überprüfe“ die Kante, ohne aber w zu besuchen.
- ▶ Markiere Knoten v als „abgeschlossen“ (BLACK).

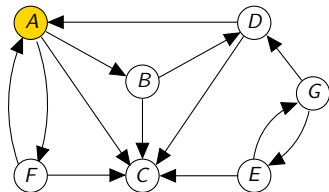
Tiefensuche

Tiefensuche (Depth-First Search, DFS)

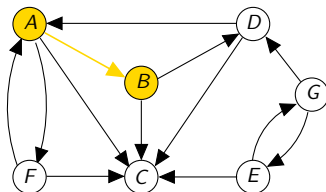
Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
 - ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche rekursiv von w aus, d. h.:
 - ▶ Erforsche Kante (v, w) , besuche w , forsche von dort aus, bis es nicht mehr weiter geht.
 - ▶ Dann backtracke von w nach v .
 - ▶ Für jede Kante (v, w) in G mit gefundenem Nachfolger w :
 - ▶ „Überprüfe“ die Kante, ohne aber w zu besuchen.
 - ▶ Markiere Knoten v als „abgeschlossen“ (BLACK).
- ▶ Man erhält wieder die Menge aller Knoten, die vom Startknoten aus erreichbar sind.

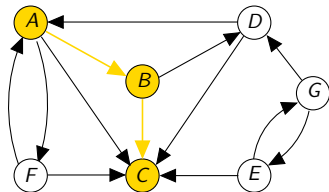
Tiefensuche – Beispiel (I)



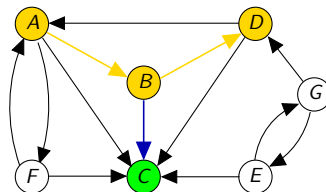
Beginn der Tiefensuche



Erforsche einen Knoten



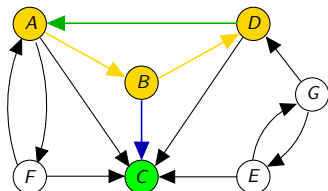
Erforsche einen Knoten



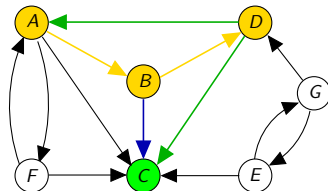
Sackgasse!

Backtracke und erforsche den nächsten Knoten

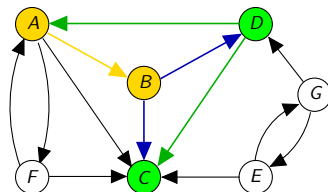
Tiefensuche – Beispiel (II)



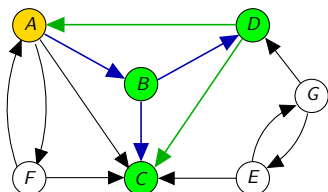
Nächster Zustand wurde bereits gefunden
Backtracke und erforsche den nächsten Knoten



Nächster Zustand wurde bereits gefunden
Backtracke und erforsche den nächsten Knoten

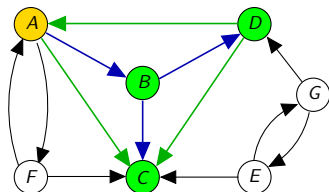


D ist eine Sackgasse
Backtracke und erforsche den nächsten Knoten

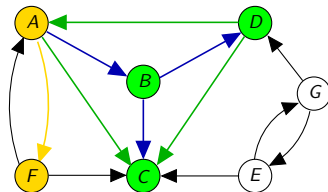


B ist eine Sackgasse
Backtracke und erforsche den nächsten Knoten

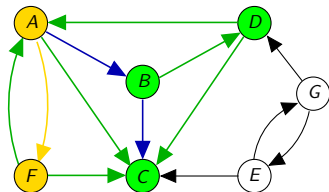
Tiefensuche – Beispiel (III)



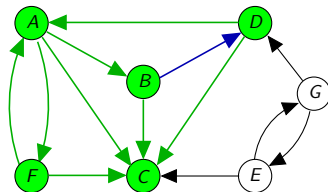
C wurde bereits gefunden
Backtrace und erforsche den nächsten Knoten



Erforsche den nächsten Knoten



Beide nächsten Knoten wurden bereits gefunden



Fertig!

Tiefensuche – Implementierung

```
1 void dfsRec(List adjList[n], int n, int start, int &color[n]) {
2     color[start] = GRAY;
3     foreach (next in adjList[start]) {
4         if (color[next] == WHITE) {
5             dfsSRec(adjList, n, next, color);
6         }
7     }
8     color[start] = BLACK;
9 }

11 void dfsSearch(List adjList[n], int n, int start) {
12     int color[n];
13     for (int i = 0; i < n; i++) { // Initialisierung
14         color[i] = WHITE;
15     }
16     dfsRec(adjList, n, start, color);
17 }
```

Eigenschaften der Tiefensuche

- ▶ Erforsche einen Pfad **so weit wie möglich**, bevor man backtrackt.

Eigenschaften der Tiefensuche

- ▶ Erforsche einen Pfad **so weit wie möglich**, bevor man backtrackt.
- ▶ Das entspricht der Reihenfolge der rekursiven Aufrufe.

Eigenschaften der Tiefensuche

- ▶ Erforsche einen Pfad **so weit wie möglich**, bevor man backtrackt.
- ▶ Das entspricht der Reihenfolge der rekursiven Aufrufe.
- ▶ Die zu verarbeitenden Knoten werden in LIFO-Reihenfolge geprüft.

Eigenschaften der Tiefensuche

- ▶ Erforsche einen Pfad **so weit wie möglich**, bevor man backtrackt.
- ▶ Das entspricht der Reihenfolge der rekursiven Aufrufe.
- ▶ Die zu verarbeitenden Knoten werden in LIFO-Reihenfolge geprüft.
- ▶ Es gibt **zwei** „Verarbeitungsmöglichkeiten“ für einen Knoten:

Eigenschaften der Tiefensuche

- ▶ Erforsche einen Pfad **so weit wie möglich**, bevor man backtrackt.
- ▶ Das entspricht der Reihenfolge der rekursiven Aufrufe.
- ▶ Die zu verarbeitenden Knoten werden in LIFO-Reihenfolge geprüft.
- ▶ Es gibt **zwei** „Verarbeitungsmöglichkeiten“ für einen Knoten:
 1. Wenn der Knoten entdeckt wird.

Eigenschaften der Tiefensuche

- ▶ Erforsche einen Pfad **so weit wie möglich**, bevor man backtrackt.
- ▶ Das entspricht der Reihenfolge der rekursiven Aufrufe.
- ▶ Die zu verarbeitenden Knoten werden in LIFO-Reihenfolge geprüft.
- ▶ Es gibt **zwei** „Verarbeitungsmöglichkeiten“ für einen Knoten:
 1. Wenn der Knoten entdeckt wird.
 2. Wenn der Knoten als „abgearbeitet“ markiert wird (und alle seine Nachfolger entdeckt werden).

Eigenschaften der Tiefensuche

- ▶ Erforsche einen Pfad **so weit wie möglich**, bevor man backtrackt.
 - ▶ Das entspricht der Reihenfolge der rekursiven Aufrufe.
 - ▶ Die zu verarbeitenden Knoten werden in LIFO-Reihenfolge geprüft.
 - ▶ Es gibt **zwei** „Verarbeitungsmöglichkeiten“ für einen Knoten:
 1. Wenn der Knoten entdeckt wird.
 2. Wenn der Knoten als „abgearbeitet“ markiert wird (und alle seine Nachfolger entdeckt werden).
- ⇒ Diese letztgenannte Möglichkeit macht Tiefensuche beliebt.

Eigenschaften der Tiefensuche

- ▶ Erforsche einen Pfad **so weit wie möglich**, bevor man backtrackt.
 - ▶ Das entspricht der Reihenfolge der rekursiven Aufrufe.
 - ▶ Die zu verarbeitenden Knoten werden in LIFO-Reihenfolge geprüft.
 - ▶ Es gibt **zwei** „Verarbeitungsmöglichkeiten“ für einen Knoten:
 1. Wenn der Knoten entdeckt wird.
 2. Wenn der Knoten als „abgearbeitet“ markiert wird (und alle seine Nachfolger entdeckt werden).
- ⇒ Diese letztgenannte Möglichkeit macht Tiefensuche beliebt.

Theorem (Komplexität der Tiefensuche)

Zeitkomplexität: $\Theta(|V| + |E|)$, Platzkomplexität: $\Theta(|V|)$.

Finden von Zusammenhangskomponenten (I)

Problem

Finde die Zusammenhangskomponenten des *ungerichteten* Graphen G .

Finden von Zusammenhangskomponenten (I)

Problem

Finde die Zusammenhangskomponenten des *ungerichteten* Graphen G .

Lösung

- ▶ *Konstruiere den zugehörigen symmetrischen Digraph (mit $2 \cdot |E|$ Kanten).*

Finden von Zusammenhangskomponenten (I)

Problem

Finde die Zusammenhangskomponenten des *ungerichteten* Graphen G .

Lösung

- ▶ Konstruiere den zugehörigen symmetrischen Digraph (mit $2 \cdot |E|$ Kanten).
- ▶ Verwende Tiefensuche:

Finden von Zusammenhangskomponenten (I)

Problem

Finde die Zusammenhangskomponenten des *ungerichteten* Graphen G .

Lösung

- ▶ Konstruiere den zugehörigen symmetrischen Digraph (mit $2 \cdot |E|$ Kanten).
- ▶ Verwende Tiefensuche:
 - ▶ Beginne bei einem beliebigen Knoten.

Finden von Zusammenhangskomponenten (I)

Problem

Finde die Zusammenhangskomponenten des *ungerichteten* Graphen G .

Lösung

- ▶ Konstruiere den zugehörigen symmetrischen Digraph (mit $2 \cdot |E|$ Kanten).
- ▶ Verwende Tiefensuche:
 - ▶ Beginne bei einem beliebigen Knoten.
 - ▶ Finde alle anderen Knoten (und Kanten) in der selben Komponente mit DFS.

Finden von Zusammenhangskomponenten (I)

Problem

Finde die Zusammenhangskomponenten des *ungerichteten* Graphen G .

Lösung

- ▶ Konstruiere den zugehörigen symmetrischen Digraph (mit $2 \cdot |E|$ Kanten).
- ▶ Verwende Tiefensuche:
 - ▶ Beginne bei einem beliebigen Knoten.
 - ▶ Finde alle anderen Knoten (und Kanten) in der selben Komponente mit DFS.
 - ▶ Wenn es weitere Knoten gibt, wähle einen und wiederhole das Verfahren.

Finden von Zusammenhangskomponenten (I)

Problem

Finde die Zusammenhangskomponenten des *ungerichteten* Graphen G .

Lösung

- ▶ Konstruiere den zugehörigen symmetrischen Digraph (mit $2 \cdot |E|$ Kanten).
- ▶ Verwende Tiefensuche:
 - ▶ Beginne bei einem beliebigen Knoten.
 - ▶ Finde alle anderen Knoten (und Kanten) in der selben Komponente mit DFS.
 - ▶ Wenn es weitere Knoten gibt, wähle einen und wiederhole das Verfahren.
- ▶ Man erhält einen **Tiefensuchswald**.
- ▶ Die Zeitkomplexität ist $\Theta(|V| + |E|)$.

Finden von Zusammenhangskomponenten (II)

```
1 // Ausgabe in cc: cc[v] = Komponente von Knoten v
2 void connComponents(List adjLst[n], int n, int &cc[n]) {
3     int color[n], ccNum = 0;
4     for (int v = 0; v < n; v++) { // Initialisierung
5         color[v] = WHITE;
6     }
7     for (int v = 0; v < n; v++) {
8         if (color[v] == WHITE) { // weitere Komponente
9             dfsSearch(adjLst, n, v, ccNum++, cc);
10        }
11    }
12 }
```

Finden von Zusammenhangskomponenten (III)

```
1 void dfsSearch(List adjLst[n], int n, int start, int &color[n],
2               int ccNum, int &cc[n]) {
3     color[start] = GRAY;
4     cc[start] = ccNum; // speichere Nummer der Komponente von v
5     foreach (next in adjLst[start]) {
6         if (color[next] == WHITE) {
7             dfsSearch(adjLst, n, next, color, ccNum, cc);
8         }
9     }
10    color[start] = BLACK;
11 }
```
