

Algorithmen und Datenstrukturen

C2. Graphenexploration: Anwendungen

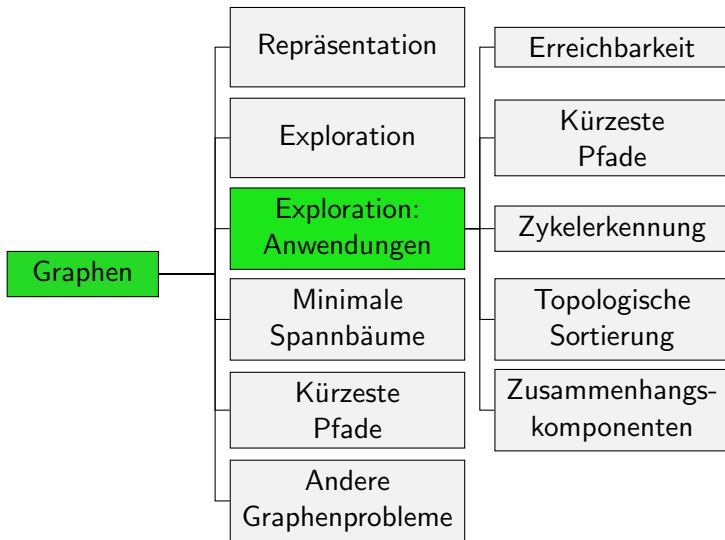
Gabriele Röger

Universität Basel

Erinnerung: Graphenexploration

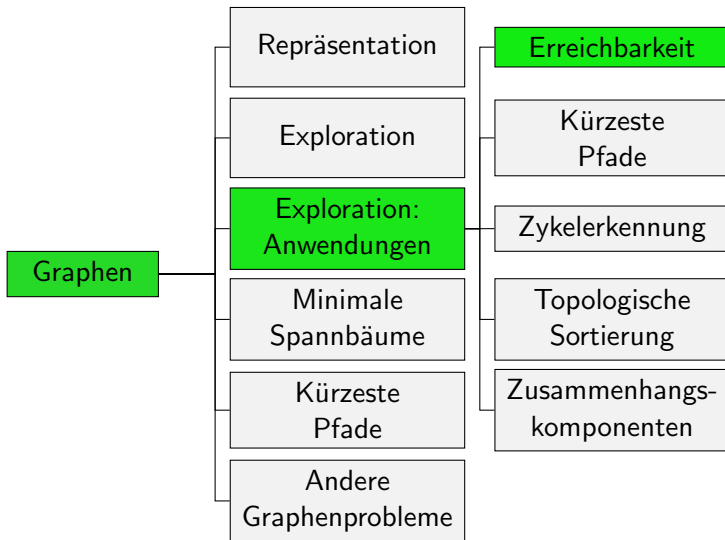
- **Aufgabe:** Gegeben einen Knoten v , besuche alle Knoten, die von v aus erreichbar sind.
- Wird oft als Teil anderer Graphenalgorithmen benötigt.
- **Tiefensuche:** erst einmal möglichst tief in den Graphen (weit weg von v)
- **Breitensuche:** erst alle Nachbarn, dann Nachbarn der Nachbarn, ...

Graphen: Übersicht



Erreichbarkeit

Graphen: Übersicht



Mark-and-Sweep-Speicherbereinigung

Ziel: Gib Speicherplatz frei, der von nicht mehr zugreifbaren Objekten belegt wird.

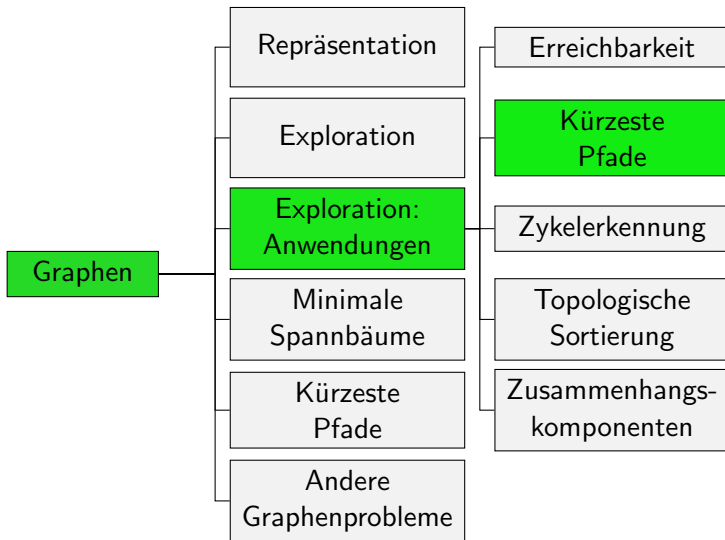
- Gerichteter Graph: **Objekte** als Knoten, **Referenzen auf Objekte** als Kanten
- Ein Bit pro Objekt für Markierung in Speicherbereinigung
- **Mark:** Markiere in regelmässigen Abständen alle erreichbaren Objekte.
- **Sweep:** Gib alle nicht markierten Objekte frei.

Zauberstab in Bildbearbeitung



Kürzeste Pfade

Graphen: Übersicht



Kürzeste Pfade: Idee

- Breitensuche besucht die Knoten mit aufsteigendem (minimalen) Abstand vom Startknoten.
- Erster Besuch eines Knoten passiert auf kürzestem Pfad.
- **Idee:** Verwende Pfad aus induzierten Suchbaum

Jupyter-Notebook



Jupyter-Notebook: `graph_exploration_applications.ipynb`

Kürzeste-Pfade-Problem

Kürzeste-Pfade-Problem mit einem Startknoten

- Gegeben: Graph und Startknoten s
- Anfrage für Knoten v
 - Gibt es Pfad von s nach v ?
 - Wenn ja, was ist der kürzeste Pfad?
- Engl. [single-source shortest paths](#), SSSP

Kürzeste Pfade: Algorithmus

```
1 class SingleSourceShortestPaths:
2     def __init__(self, graph, start_node):
3         self.predecessor = [None] * graph.no_nodes()
4         self.predecessor[start_node] = start_node
5
6         # precompute predecessors with breadth-first search with
7         # self.predecessors used for detecting visited nodes
8         queue = deque()
9         queue.append(start_node)
10        while queue:
11            v = queue.popleft()
12            for s in graph.successors(v):
13                if self.predecessor[s] is None:
14                    self.predecessor[s] = v
15                    queue.append(s)
16        ...
```

Im Prinzip wie gehabt
(nur als Klasse)

Kürzeste Pfade: Algorithmus (Fortsetzung)

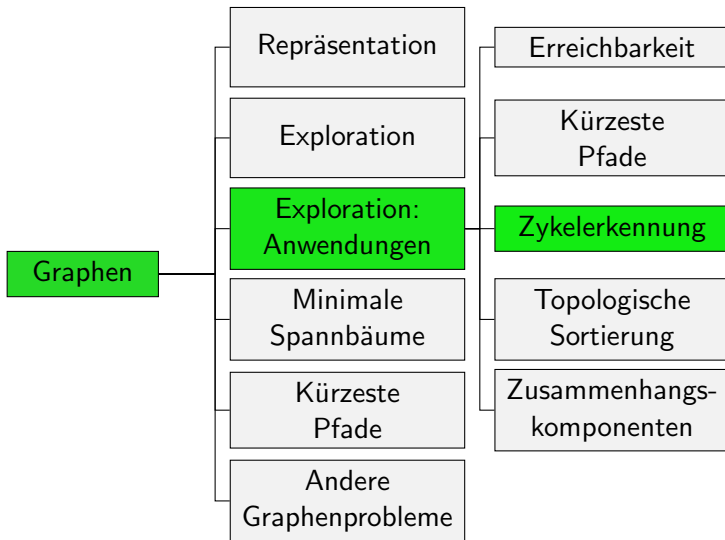
```
19     def has_path_to(self, node):
20         return self.predecessor[node] is not None
21
22     def get_path_to(self, node):
23         if not self.has_path_to(node):
24             return None
25         if self.predecessor[node] == node: # start node
26             return [node]
27         pre = self.predecessor[node]
28         path = self.get_path_to(pre)
29         path.append(node)
30         return path
```

Laufzeit?

Später: Kürzeste Pfade mit Kantengewichten

Azyklische Graphen

Graphen: Übersicht



Erkennung von azyklischen Graphen

Definition (Gerichteter, azyklischer Graph)

Ein **gerichteter, azyklischer Graph** (directed acyclic graph, DAG) ist ein gerichteter Graph, der keine gerichteten Zyklen enthält.

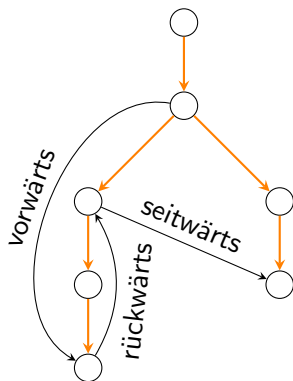
Erkennung von azyklischen Graphen

Definition (Gerichteter, azyklischer Graph)

Ein **gerichteter, azyklischer Graph** (directed acyclic graph, DAG) ist ein gerichteter Graph, der keine gerichteten Zyklen enthält.

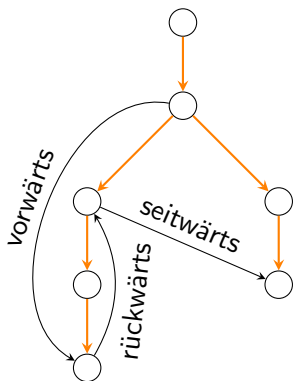
Aufgabe: Entscheide, ob ein gerichteter Graph einen Zyklus enthält. Falls ja, gib einen Zyklus aus.

Kriterium für Zyklfreiheit



Induzierter Suchbaum einer
Tiefensuche (orange) und
mögliche andere Kanten

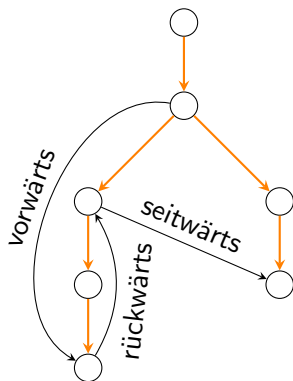
Kriterium für Zyklfreiheit



Induzierter Suchbaum einer
Tiefensuche (orange) und
mögliche andere Kanten

Der (erreichbare Teil-) Graph
ist genau dann azyklisch, wenn
keine Rückwärtskante existiert.

Kriterium für Zyklfreiheit



Induzierter Suchbaum einer
Tiefensuche (orange) und
mögliche andere Kanten

Der (erreichbare Teil-) Graph
ist genau dann azyklisch, wenn
keine Rückwärtskante existiert.

Idee: Merke dir Knoten auf aktuellem Pfad in Tiefensuche

Zykeltest: Algorithmus

```
1 class DirectedCycle:
2     def __init__(self, graph):
3         self.predecessor = [None] * graph.no_nodes()
4         self.on_current_path = [False] * graph.no_nodes()
5         self.cycle = None
6         for node in range(graph.no_nodes()):
7             if self.has_cycle():
8                 break
9             if self.predecessor[node] is None:
10                self.predecessor[node] = node
11                self.dfs(graph, node)
12
13     def has_cycle(self):
14         return self.cycle is not None
```

Wiederholte Tiefen-
suchen, so dass am
Ende alle Knoten
besucht wurden

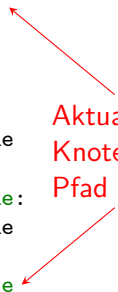
Zykeltest: Algorithmus (Fortsetzung)

```
16     def dfs(self, graph, node):
17         self.on_current_path[node] = True
18         for s in graph.successors(node):
19             if self.has_cycle():
20                 return
21             if self.on_current_path[s]:
22                 self.predecessor[s] = node
23                 self.extract_cycle(s)
24             if self.predecessor[s] is None:
25                 self.predecessor[s] = node
26                 self.dfs(graph, s)
27         self.on_current_path[node] = False
```

Zykeltest: Algorithmus (Fortsetzung)

```
16 def dfs(self, graph, node):
17     self.on_current_path[node] = True
18     for s in graph.successors(node):
19         if self.has_cycle():
20             return
21         if self.on_current_path[s]:
22             self.predecessor[s] = node
23             self.extract_cycle(s)
24         if self.predecessor[s] is None:
25             self.predecessor[s] = node
26             self.dfs(graph, s)
27     self.on_current_path[node] = False
```

Aktualisiere, ob
Knoten auf aktuellem
Pfad ist.



Zykeltest: Algorithmus (Fortsetzung)

```
16     def dfs(self, graph, node):
17         self.on_current_path[node] = True
18         for s in graph.successors(node):
19             if self.has_cycle():
20                 return
21             if self.on_current_path[s]:
22                 self.predecessor[s] = node
23                 self.extract_cycle(s)
24                 if self.predecessor[s] is None:
25                     self.predecessor[s] = node
26                     self.dfs(graph, s)
27         self.on_current_path[node] = False
```

Zyklus
gefunden

Aktualisiere, ob
Knoten auf aktuellem
Pfad ist.

Zykeltest: Algorithmus (Fortsetzung)

```
16 def dfs(self, graph, node):
17     self.on_current_path[node] = True
18     for s in graph.successors(node):
19         if self.has_cycle():
20             return
21         if self.on_current_path[s]:
22             self.predecessor[s] = node
23             self.extract_cycle(s)
24             if self.predecessor[s] is None:
25                 self.predecessor[s] = node
26                 self.dfs(graph, s)
27     self.on_current_path[node] = False
```

Brich ab, wenn
irgendwo Zyklus
gefunden.

Aktualisiere, ob
Knoten auf aktuellem
Pfad ist.

Zyklus
gefunden

Zykeltest: Algorithmus (Fortsetzung)

Bei Aufruf von `extract_cycle` liegt `node` auf einem Zyklus in `self.predecessor`.

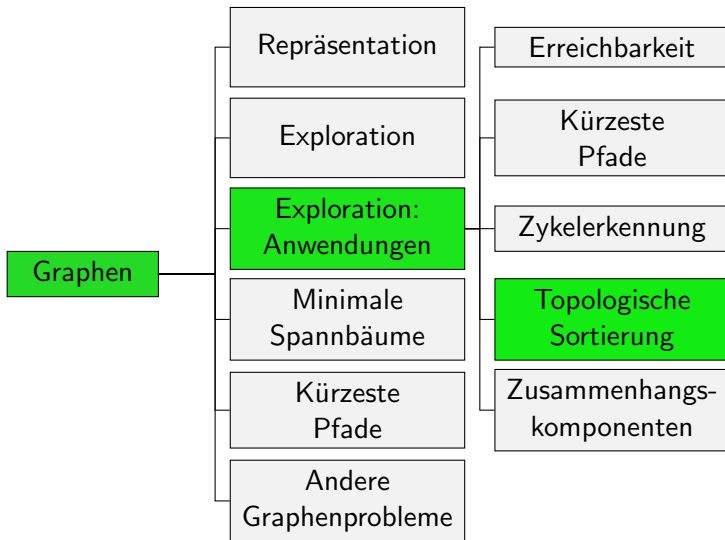
```
29     def extract_cycle(self, node):
30         self.cycle = deque()
31         current = node
32         self.cycle.appendleft(current)
33         while True:
34             current = self.predecessor[current]
35             self.cycle.appendleft(current)
36             if current == node:
37                 return
```

Jupyter-Notebook



Jupyter-Notebook: `graph_exploration_applications.ipynb`

Graphen: Übersicht



Topologische Sortierung

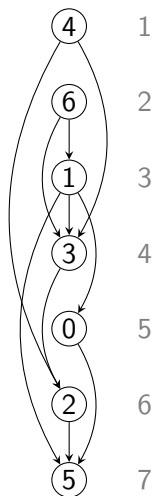
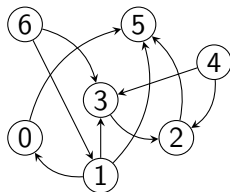
Definition

Eine **topologische Sortierung** eines **azyklischen**, gerichteten Graphen $G = (V, E)$, ist eine Nummerierung $no : V \rightarrow \mathbb{N}$ der Knoten, so dass für jede Kante (u, v) gilt, dass $no(u) < no(v)$.

Zum Beispiel relevant für **Ablaufplanung**:

Kante (u, v) drückt aus, dass u vor v „erledigt“ werden muss.

Topologische Sortierung: Illustration



Topologische Sortierung: Algorithmus

Theorem

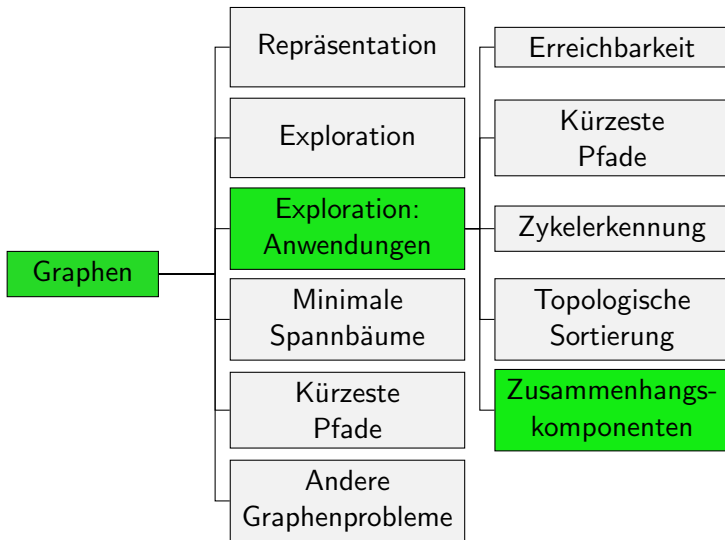
*Für den erreichbaren Teilgraphen eines azyklischen Graphen ist die **umgekehrte Depth-First-Postorder-Knotenreihenfolge** eine topologische Sortierung.*

Algorithmus:

- Folge von Tiefensuchen-Aufrufen (für bisher unbesuchte Knoten) bis alle Knoten besucht.
- Speichere jeweils umgekehrte Postorderreihenfolge P_i für i -te Suche
- Sei k Anzahl der Suchen. Dann ergibt die Aneinanderreihung von $P_k, \dots, P_1$ eine topologische Sortierung.

Zusammenhang

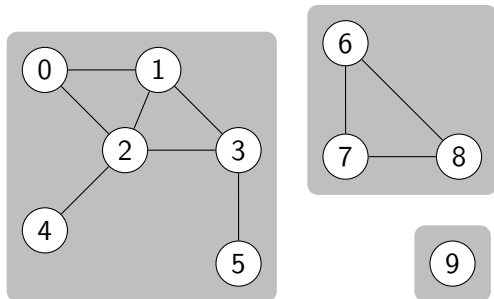
Graphen: Übersicht



Zusammenhangskomponenten ungerichteter Graphen

Ungerichteter Graph

- Zwei Knoten u und v sind in der gleichen **Zusammenhangskomponente**, wenn es einen Pfad zwischen u und v gibt.



Zusammenhangskomponenten: Interface

Wir möchten folgendes Interface implementieren:

```
1  class ConnectedComponents:
2      # Vorverarbeitender Konstruktor
3      def __init__(graph: UndirectedGraph) -> None
4
5      # Sind Knoten node1 und node2 verbunden?
6      def connected(node1: int, node2: int) -> bool
7
8      # Anzahl der Zusammenhangskomponenten
9      def count() -> int
10
11     # Komponentenbezeichner für node
12     # (zwischen 0 und count()-1)
13     def id(node: int) -> int
```

Zusammenhangskomponenten: Interface

Wir möchten folgendes Interface implementieren:

```
1  class ConnectedComponents:
2      # Vorverarbeitender Konstruktor
3      def __init__(graph: UndirectedGraph) -> None
4
5      # Sind Knoten node1 und node2 verbunden?
6      def connected(node1: int, node2: int) -> bool
7
8      # Anzahl der Zusammenhangskomponenten
9      def count() -> int
10
11     # Komponentenbezeichner für node
12     # (zwischen 0 und count()-1)
13     def id(node: int) -> int
```

Idee: Folge von Graphexplorationen bis alle Knoten besucht sind.
ID eines Knoten entspricht Iteration, in der er besucht wurde

Zusammenhangskomponenten: Algorithmus

```
1 class ConnectedComponents:
2     def __init__(self, graph):
3         self.id = [None] * graph.no_nodes()
4         self.curr_id = 0
5         visited = [False] * graph.no_nodes()
6         for node in range(graph.no_nodes()):
7             if not visited[node]:
8                 self.dfs(graph, node, visited)
9                 self.curr_id += 1
10
11     def dfs(self, graph, node, visited):
12         if visited[node]:
13             return
14         visited[node] = True
15         self.id[node] = self.curr_id
16         for n in graph.neighbours(node):
17             self.dfs(graph, n, visited)
```

Wie sehen connected, count und id aus?

Jupyter-Notebook



Jupyter-Notebook: `graph_exploration_applications.ipynb`

Zusammenhangskomponenten gerichteter Graphen

Gerichteter Graph G

- Ignoriert man die Richtung der Kanten, ist jede Zusammenhangskomponente des resultierenden ungerichteten Graphen eine **schwache Zusammenhangskomponente** von G .

Zusammenhangskomponenten gerichteter Graphen

Gerichteter Graph G

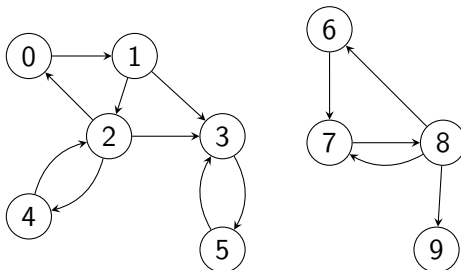
- Ignoriert man die Richtung der Kanten, ist jede Zusammenhangskomponente des resultierenden ungerichteten Graphen eine **schwache Zusammenhangskomponente** von G .
- G ist **stark zusammenhängend**, wenn von jedem Knoten zu jedem anderen Knoten ein gerichteter Pfad existiert.

Zusammenhangskomponenten gerichteter Graphen

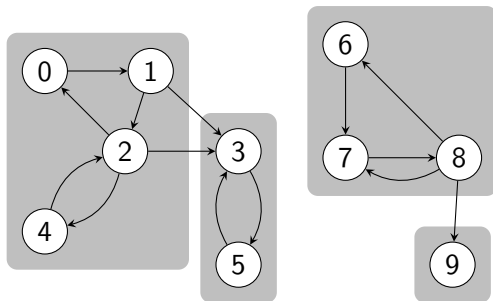
Gerichteter Graph G

- Ignoriert man die Richtung der Kanten, ist jede Zusammenhangskomponente des resultierenden ungerichteten Graphen eine **schwache Zusammenhangskomponente** von G .
- G ist **stark zusammenhängend**, wenn von jedem Knoten zu jedem anderen Knoten ein gerichteter Pfad existiert.
- Eine **starke Zusammenhangskomponente** von G ist ein maximal grosser Teilgraph, der stark zusammenhängend ist.

Starke Zusammenhangskomponenten



Starke Zusammenhangskomponenten



Starke Zusammenhangskomponenten

Kosaraju-Algorithmus

- Gegeben Graph $G = (V, E)$, berechne zunächst ein umgekehrte Postorderreihenfolge P (für alle Knoten) des Graphen $G^R = (V, \{(v, u) \mid (u, v) \in E\})$ (alle Kanten umgedreht).
- Führe eine Folge von Explorationen in G aus. Wähle dabei als nächsten Startknoten jeweils den ersten noch unbesuchten Knoten in P .
- Alle Knoten, die innerhalb einer Exploration erreicht werden, sind in der gleichen starken Zusammenhangskomponente.

Jupyter-Notebook



Jupyter-Notebook: `graph_exploration_applications.ipynb`

Zusammenfassung

Zusammenfassung

Wir haben eine Reihe von Anwendungen der Graphenexploration betrachtet:

- Erreichbarkeit
- Kürzeste Pfade
- Zykelerkennung
- Topologische Sortierung
- Zusammenhangskomponenten