Algorithmen und Datenstrukturen

C2. Graphenexploration: Anwendungen

Gabriele Röger

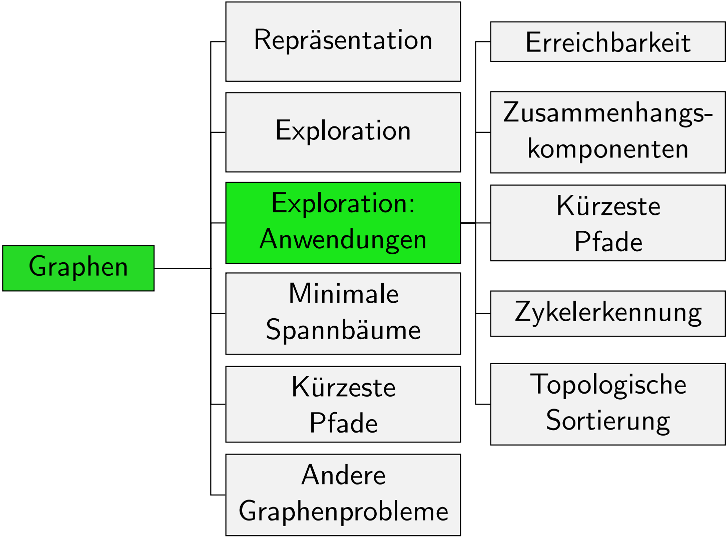
Universität Basel

25. April 2019

Erinnerung: Graphenexploration

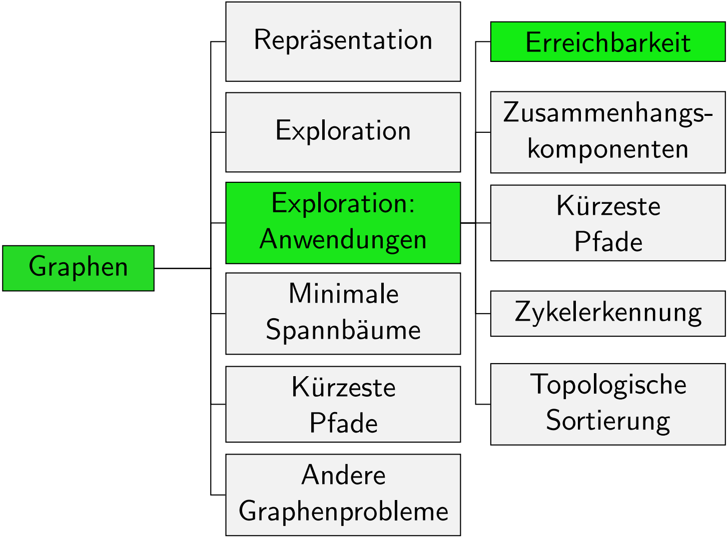
- Aufgabe: Gegeben einen Knoten v, besuche alle Knoten, die von v aus erreichbar sind.
- Wird oft als Teil anderer Graphenalgorithmen benötigt.
- Tiefensuche: erst einmal möglichst tief in den Graphen (weit weg von v)
- Breitensuche: erst alle Nachbarn, dann Nachbarn der Nachbarn, . . .

Graphen: Übersicht



Erreichbarkeit

Graphen: Übersicht



Mark-and-Sweep-Speicherbereinigung

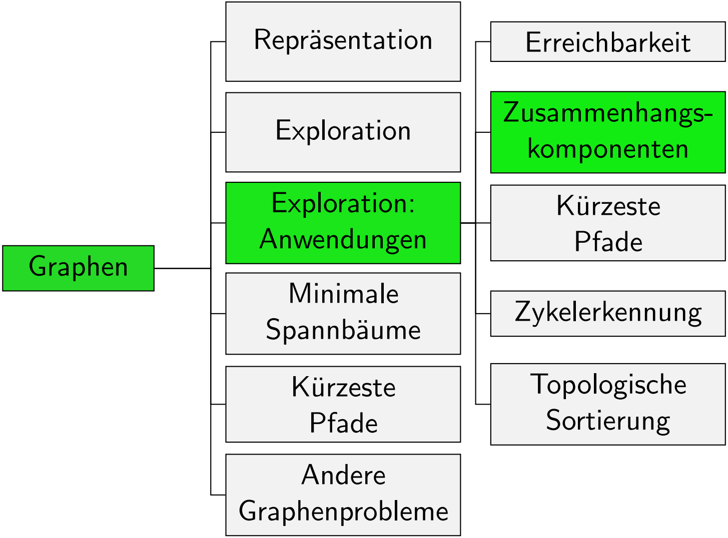
Ziel: Gib Speicherplatz frei, der von nicht mehr zugreifbaren Objekten belegt wird.

- 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

Zusammenhang

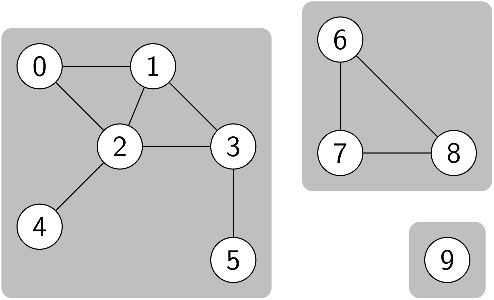
Graphen: Übersicht



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:

```
class ConnectedComponents:
  # Vorverarbeitender Konstruktor
  def __init__(graph: UndirectedGraph) -> None

  # Sind Knoten node1 und node2 verbunden?
  def connected(node1: int, node2: int) -> bool

  # Anzahl der Zusammenhangskomponenten
  def count() -> int

  # Komponentenbezeichner für node
  # (zwischen 0 und count()-1)
  def id(node: int) -> int
```

Zusammenhangskomponenten: Interface

Wir möchten folgendes Interface implementieren:

```
class ConnectedComponents:
  # Vorverarbeitender Konstruktor
  def __init__(graph: UndirectedGraph) -> None

  # Sind Knoten node1 und node2 verbunden?
  def connected(node1: int, node2: int) -> bool

  # Anzahl der Zusammenhangskomponenten
  def count() -> int

  # Komponentenbezeichner für node
  # (zwischen 0 und count()-1)
  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

```
class ConnectedComponents:
    def __init__(self, graph):
        self.id = [None] * graph.no_nodes()
        self.curr_id = 0
        visited = [False] * graph.no_nodes()
        for node in range(graph.no_nodes()):
            if not visited[node]:
                self.dfs(graph, node, visited)
                self.curr_id += 1

    def dfs(self, graph, node, visited):
        if visited[node]:
            return
        visited[node] = True
        self.id[node] = self.curr_id
        for n in graph.neighbours(node):
            self.dfs(graph, n, visited)
```

Wie sehen connected, count und id aus?

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

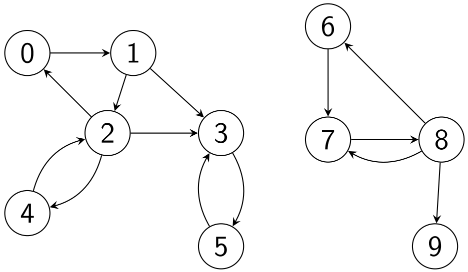
- Ignoriert man die Richtung der Kanten, ist jede Zusammenhangskomponente des resultierenden ungerichteten Graphen eine schwache Zusammenhangskomponente von G.
- G ist stark zusammenhängend, wenn von jedem Knoten zu jedem anderen Knoten ein gerichteter Pfad existiert.

Zusammenhangskomponenten gerichteter Graphen

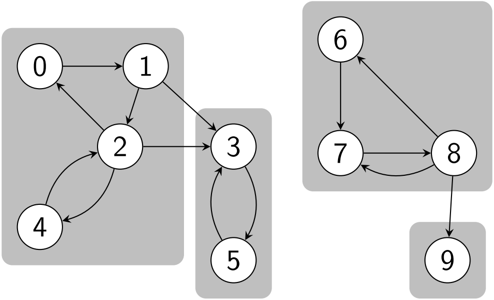
Gerichteter Graph G

- 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



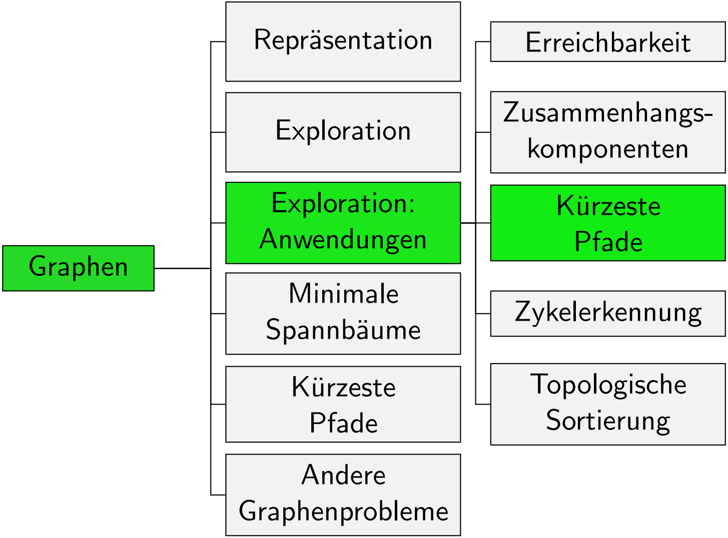
Starke Zusammenhangskomponenten

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.

Kürzeste Pfade

Graphen: Übersicht



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: 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

Kürzeste Pfade: Algorithmus

```
class SingleSourceShortestPaths:
    def __init__(self, graph, start_node):
        self.graph = graph
        self.predecessor = [None] * graph.no_nodes()
        self.predecessor[start_node] = start_node

        # precompute predecessors with beadth-first search with
        # self.predecessors used for detecting visited nodes
        queue = deque()
        queue.append(start_node)
        while queue:
            v = queue.popleft()
            for s in graph.successors(v):
                if self.predecessor[s] is None:
                    self.predecessor[s] = v
                    queue.append(s)
    ...
```

Im Prinzip wie gehabt (nur als Klasse)

Kürzeste Pfade: Algorithmus (Fortsetzung)

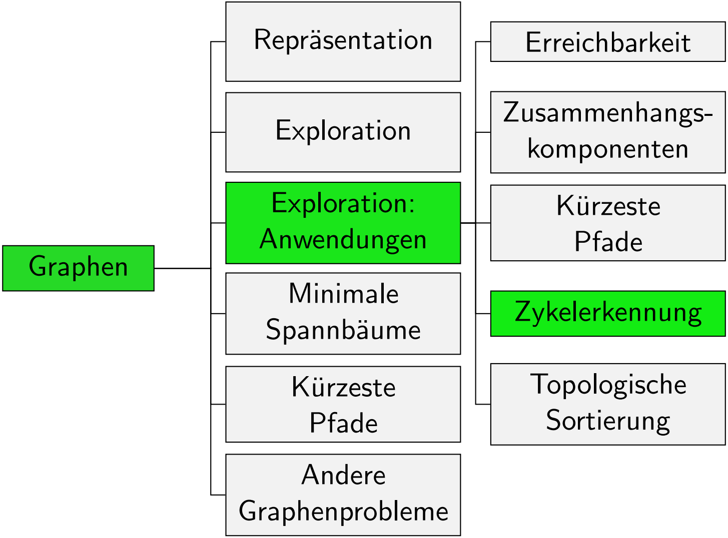
```python
    def has_path_to(self, node):
        return self.predecessor[node] is not None

    def get_path_to(self, node):
        if not self.has_path_to(node):
            return None
        if self.predecessor[node] == node:  # start node
            return [node]
        pre = self.predecessor[node]
        path = self.get_path_to(pre)
        path.append(node)
        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.

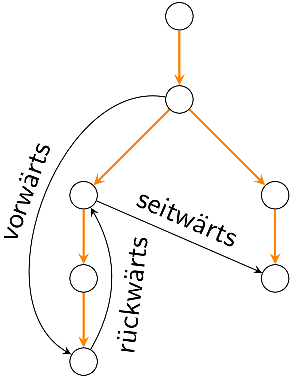
Erkennung von azyklischen Graphen

Definition (Gerichteter, azyklischer Graph)

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

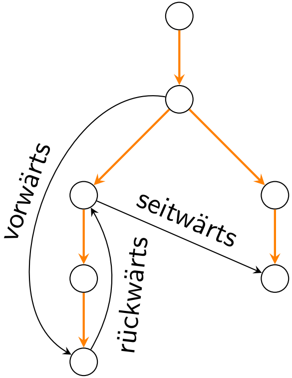
Aufgabe: Entscheide, ob ein gerichteter Graph einen Zyklus enthält. Falls ja, gib einen Zyklus aus.

Kriterium für Zykelfreiheit



Induzierter Suchbaum einer Tiefensuche (orange) und mögliche andere Kanten

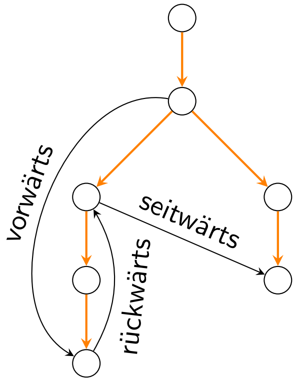
Kriterium für Zykelfreiheit



Induzierter Suchbaum einer Tiefensuche (orange) und mögliche andere Kanten

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

Kriterium für Zykelfreiheit



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

```
class DirectedCycle:
    def __init__(self, graph):
        self.predecessor = [None] * graph.no_nodes()
        self.on_current_path = [False] * graph.no_nodes()
        self.cycle = None
        for node in range(graph.no_nodes()):
            if self.has_cycle():
                break
            if self.predecessor[node] is None:
                self.predecessor[node] = node
                self.dfs(graph, node)

    def has_cycle(self):
        return self.cycle is not None
```

Wiederholte Tiefensuchen, so dass am Ende alle Knoten besucht wurden

Zykeltest: Algorithmus (Fortsetzung)

```
    def dfs(self, graph, node):
        self.on_current_path[node] = True
        for s in graph.successors(node):
            if self.has_cycle():
                return
            if self.on_current_path[s]:
                self.predecessor[s] = node
                self.extract_cycle(s)
            if self.predecessor[s] is None:
                self.predecessor[s] = node
                self.dfs(graph, s)
        self.on_current_path[node] = False
```

Zykeltest: Algorithmus (Fortsetzung)

```
    def dfs(self, graph, node):
        self.on_current_path[node] = True
        for s in graph.successors(node):
            if self.has_cycle():
                return
            if self.on_current_path[s]:
                self.predecessor[s] = node
                self.extract_cycle(s)
            if self.predecessor[s] is None:
                self.predecessor[s] = node
                self.dfs(graph, s)
        self.on_current_path[node] = False
```

Aktualisiere, ob Knoten auf aktuellem Pfad ist.

Zykeltest: Algorithmus (Fortsetzung)

```python
    def dfs(self, graph, node):
        self.on_current_path[node] = True
        for s in graph.successors(node):
            if self.has_cycle():
                return
            if self.on_current_path[s]:
                self.predecessor[s] = node
                self.extract_cycle(s)
            if self.predecessor[s] is None:
                self.predecessor[s] = node
                self.dfs(graph, s)
        self.on_current_path[node] = False
```

Zyklus gefunden

Aktualisiere, ob Knoten auf aktuellem Pfad ist.

Zykeltest: Algorithmus (Fortsetzung)

```
    def dfs(self, graph, node):
        self.on_current_path[node] = True
        for s in graph.successors(node):
            if self.has_cycle():
                return
            if self.on_current_path[s]:
                self.predecessor[s] = node
                self.extract_cycle(s)
            if self.predecessor[s] is None:
                self.predecessor[s] = node
                self.dfs(graph, s)
        self.on_current_path[node] = False
```

Brich ab, wenn irgendwo Zyklus gefunden.

Zyklus gefunden

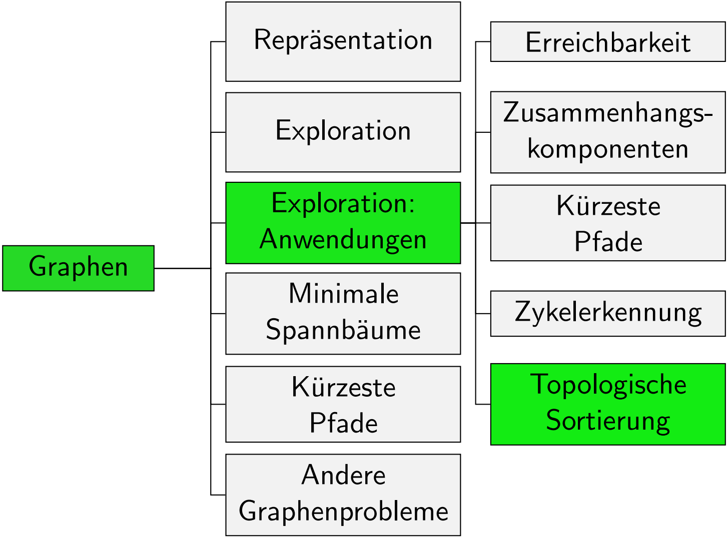
Aktualisiere, ob Knoten auf aktuellem Pfad ist.

Zykeltest: Algorithmus (Fortsetzung)

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

```
    def extract_cycle(self, node):
        self.cycle = deque()
        current = node
        self.cycle.appendleft(current)
        while True:
            current = self.predecessor[current]
            self.cycle.appendleft(current)
            if current == node:
                return
```

Graphen: Übersicht



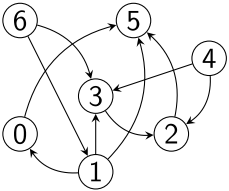
Topologische Sortierung

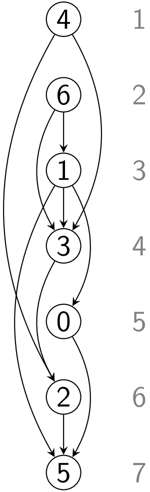
Definition

Eine topologische Sortierung eines azyklischen, gerichteten Graphen $G = (V, E)$, ist eine Nummerierung $no : V \to \mathbb{N}$ 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 azyklischenen 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 $P_k, \ldots, P_1$ eine topologische Sortierung.

Zusammenfassung

Zusammenfassung

Wir haben eine Reihe von Anwendungen der Graphenexploration betrachtet:

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