Algorithmen und Datenstrukturen

C1. Graphen: Grundlagen und Exploration

Gabriele Röger

Universität Basel

19. April 2018

C1.1 Motivation

C1.2 Grundlegende Definition

C1.3 Repräsentation

C1.4 Graphenexploration

C1.5 Zusammenfassung

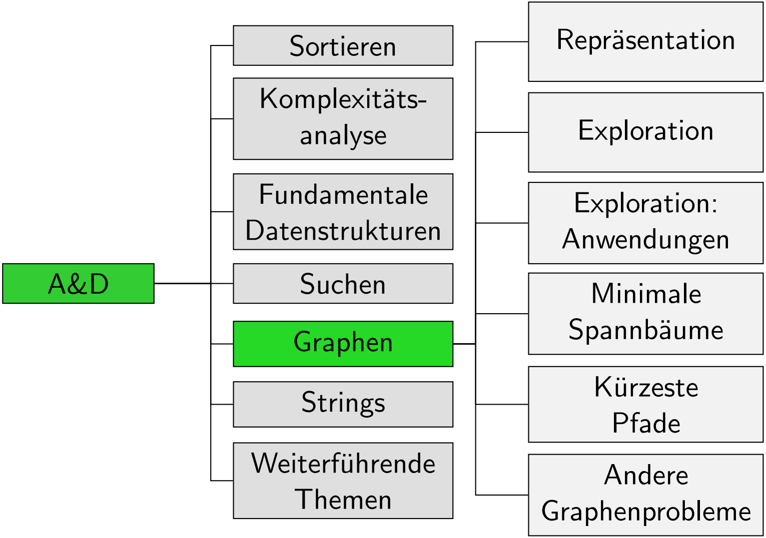
Informatikerin des Tages: Grace Hopper

Grace Hopper

- US-amerikanische Informatikerin (1906-1992)
- Revolutionäre Idee: Computerprogramme in verständlicher Sprache statt nur mit Einsen und Nullen
- „Grandma COBOL“

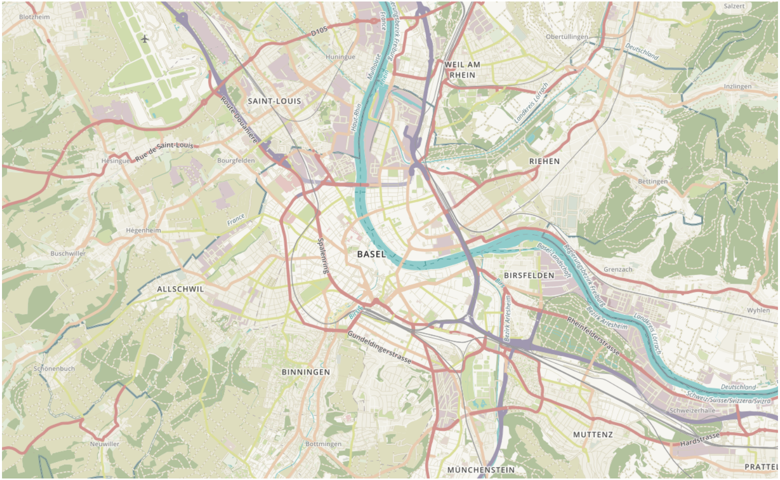
„It’s always easier to ask forgiveness than it is to get permission“

Inhalt dieser Veranstaltung



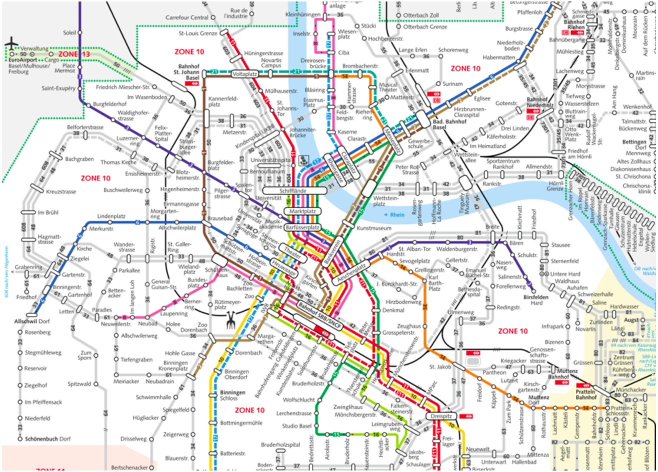
C1.1 Motivation

Strassenkarten



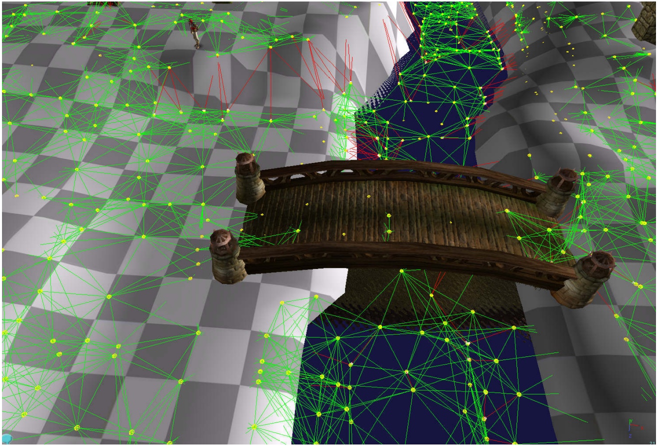
openstreetmap.org

Liniennetz



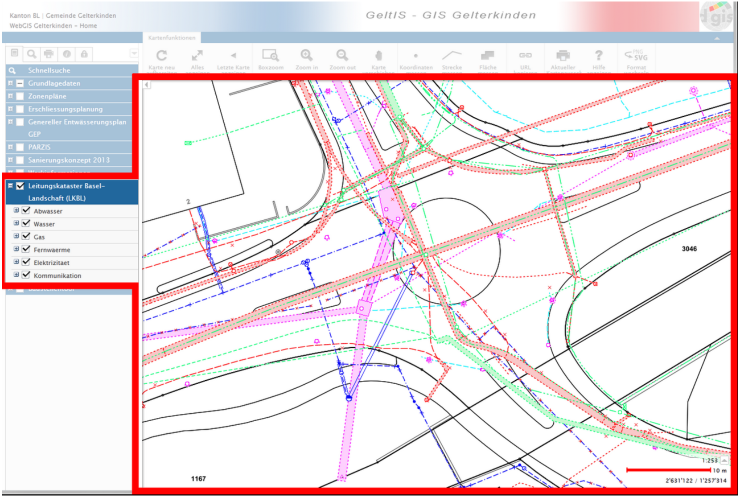
tnw.ch

Navigationsnetz in Spielen



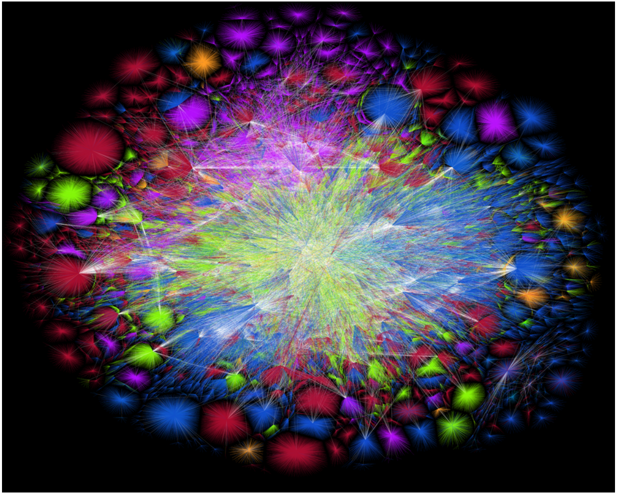
heroengine.com

Versorgungssystem



dgis.info

Internet

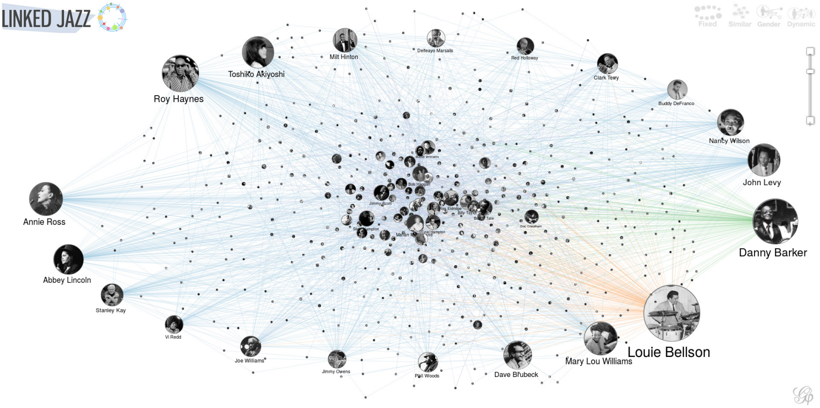


Barrett Lyon / The Opte Project
Visualization of the routing paths of the Internet.

Soziale Netzwerke

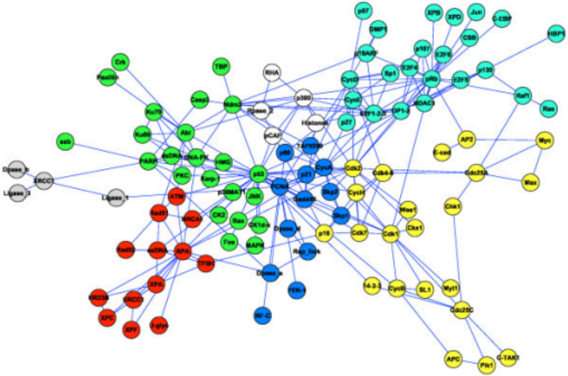
„Visualizing Friendships“ von Paul Butler

Zusammenarbeit



linkedjazz.org

Protein-Interaktion



Network representation of the p53 protein interactions
Module detection in complex networks using integer optimisation,
Xu G, Bennett L, Papageorgiou LG, Tsoka S - Algorithms Mol Biol (2010)

Mögliche Fragestellungen

- Sind A und B verbunden?
- Was ist der kürzeste Weg zwischen A und B?
- Wie weit sind zwei Elemente höchstens voneinander entfernt?
- Wieviel Wasser kann die Kanalisation abführen?

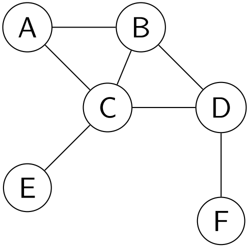
Abstrakte Graphen

Ein Graph besteht aus Knoten und Kanten zwischen Knoten.

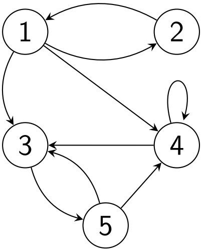
	Knoten	Kanten
Strassen	Kreuzung	Strassenabschnitt
Internet	AS ($\approx$ Provider)	Route
Facebook	Person	Freundschaft
Proteine	Protein	Interaktion

C1.2 Grundlegende Definition

Ungerichtete und gerichtete Graphen



ungerichteter Graph



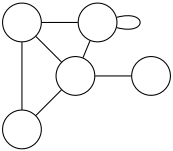
gerichteter Graph

Graphen

- Ein Graph besteht aus zwei Mengen V und E
  - V: Menge der Knoten (engl. vertices)
  - E: Menge der Kanten (engl. edges)
- Jede Kante verbindet zwei Knoten u und v
  - ungerichteter Graph: Menge $\{u, v\}$
  - gerichteter Graph: Paar (u, v)
- Bei Multigraphen kann es mehrere, parallele Kanten zwischen den gleichen Knoten geben.
- Bei gewichteten Graphen hat jede Kante ein Gewicht (Zahl).

Ungerichtete Graphen: Terminologie

- Nachbarn eines Knotens u: alle Knoten v mit $\{u, v\} \in E$.
- degree(v): Grad eines Knotens = Anzahl der Nachbarn.
  - Ausnahme: Schleife erhöht den Grad um 2.
    Schleife = Kante, die einen Knoten mit sich selbst verbindet.

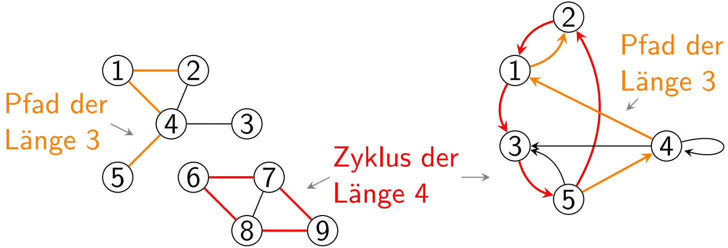


Gerichtete Graphen: Terminologie

- Nachfolger eines Knotens u: alle Knoten v mit $(u, v) \in E$.
- Vorgänger eines Knotens u: alle Knoten v mit $(v, u) \in E$.
- outdegree(v): Ausgangsgrad = Anzahl der Nachfolger
- indegree(v): Eingangsgrad = Anzahl der Vorgänger

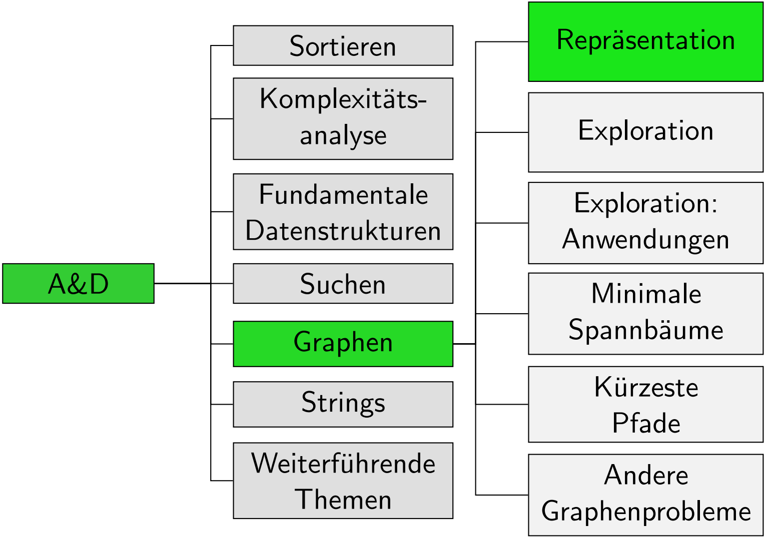
Pfade und Zyklen

- Pfad der Länge n: Sequenz $(v_0, \dots, v_n)$ von Knoten mit
  - $\{v_i, v_{i+1}\} \in E$ für $i = 0, \dots, n-1$ (ungerichteter Graph)
  - $(v_i, v_{i+1}) \in E$ für $i = 0, \dots, n-1$ (gerichteter Graph)
  - Beispiel: (5,4,1,2)
- Zyklus: Pfad mit gleichem Start- und Endknoten
  - (6,7,9,8,6) im ungerichteten und (5,2,1,3,5) im gerichteten Beispielgraphen
  - existiert kein Zyklus, ist der Graph azyklisch



C1.3 Repräsentation

Inhalt dieser Veranstaltung



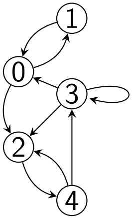
Repräsentation der Knoten

- Wir verwenden Zahlen von 0 bis $|V| - 1$ für die Knoten.
- Falls in Anwendung nicht gegeben: Verwende Symboltabellen, um zwischen Namen und Zahlen zu konvertieren.

Graphenrepräsentation mit Adjazenzmatrix

Graph $G = (\{0, \dots, |V| - 1\}, E)$ repräsentiert durch
$|V| \times |V|$-Matrix mit Einträgen a_{ik} (in Zeile i, Spalte k):

$$a_{ik} = \begin{cases} 1 & \text{falls } (v_i, v_k) \in E \text{ (gerichteter Graph) bzw.} \\ & \{v_i, v_k\} \in E \text{ (ungerichteter Graph)} \\ 0 & \text{sonst} \end{cases}$$

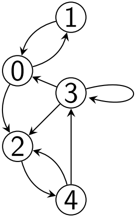


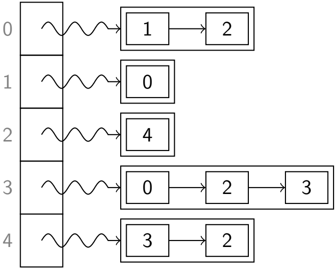
$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Bei ungerichteten Graphen symmetrisch

Graphenrepräsentation mit Adjazenzliste

Speichere für jeden Knoten die Liste aller Nachfolger / Nachbarn





Repräsentation: Komplexität

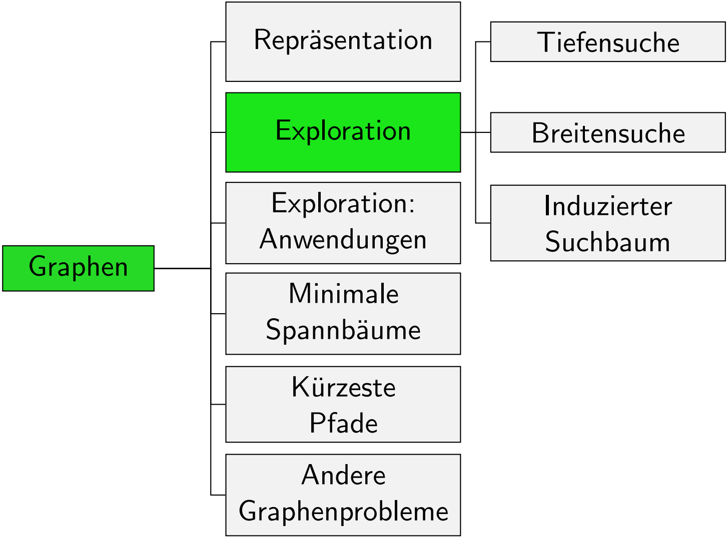
	Adj.matrix	Adj.liste
Platzbedarf	$\lvert V\rvert^2$	$\lvert E\rvert + \lvert V\rvert$
Kante hinzufügen	1	1
Kante zwischen u und v?	1	(out)degree(v)
Iterieren über ausgeh. Kanten	$\lvert V\rvert$	(out)degree(v)

Praxis: oft dünne Graphen (geringer durchschnittlicher Grad)

Welche Repräsentation?

C1.4 Graphenexploration

Graphen: Übersicht



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, ...

Tiefensuche

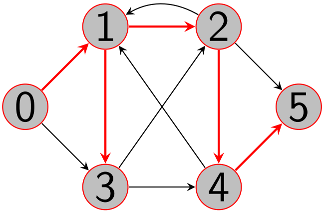
Markiere erreichte Knoten

- Markiere v
- Iteriere über die Nachfolger/Nachbarn w von v.
  - Falls w nicht markiert, starte rekursiv von w.

Englisch: Depth-first search, DFS

Tiefensuche: Beispiel

Hier: Besuche Nachfolger mit aufsteigender Knotennummer



Tiefensuche mit Startknoten 0
markiert Knoten in Reihenfolge
0 - 1 - 2 - 4 - 5 - 3

Tiefensuche: Algorithmus (rekursiv)

```
def depth_first_exploration(graph, node, visited=None):
    if visited is None:
        visited = set()
    if node in visited:
        return
    visited.add(node)
    for s in graph.successors(node):
        depth_first_exploration(graph, s, visited)
```

Falls zu erwarten ist, dass ein Grossteil der Knoten besucht wird: bool-Array statt Menge für `visited`

Depth-First-Knotenreihenfolgen

- Preorder: Knoten wird erfasst, bevor seine Kinder betrachtet werden.
- Postorder: Knoten wird erfasst, wenn die (rekursive) Tiefensuche mit allen seinen Kindern fertig ist.
- Umgekehrte Postorder: Wie Postorder, aber in umgekehrter Reihenfolge (spätere Knoten vorne)

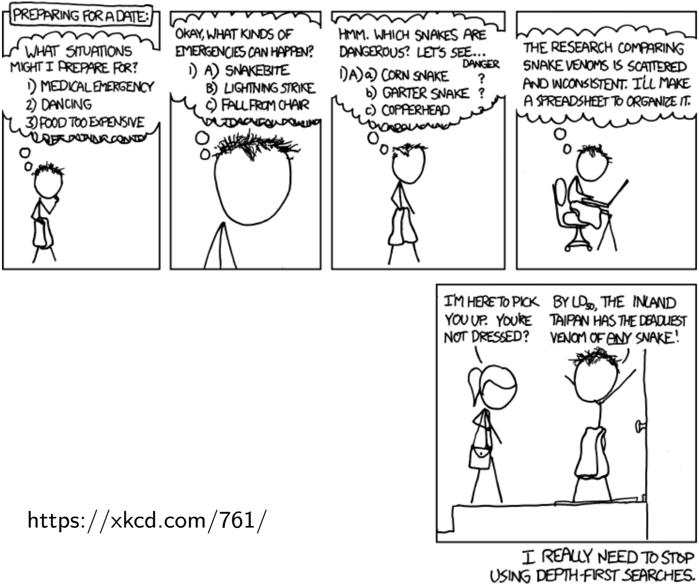
```
def depth_first_exploration(graph, node):
    if node in visited:
        return
    preorder.append(node)
    visited.add(node)
    for s in graph.successors(node):
        depth_first_exploration(graph, s, visited)
    postorder.append(node)
    reverse_postorder.appendleft(node)
```

(Repräsentation der Knotenreihenfolgen als Deque)

Tiefensuche: Algorithmus (iterativ)

```
def depth_first_exploration(graph, node):
    visited = set()
    stack = deque()
    stack.append(node)
    while stack:
        v = stack.pop()  # LIFO
        if v not in visited:
            visited.add(v)
            for s in graph.successors(v):
                stack.append(s)
```

Tiefensuche in der Praxis



https://xkcd.com/761/

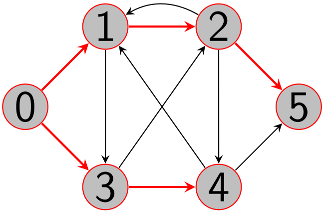
Breitensuche

- Markiere v
  $\rightarrow$ Abstand 0
- Markiere alle unmarkierten Nachfolger/Nachbarn von v
  $\rightarrow$ Abstand 1
- Markiere alle unmarkierten Nachfolger/Nachbarn von Abstand-1-Knoten
- Markiere alle unmarkierten Nachfolger/Nachbarn von Abstand-2-Knoten
- ...
- Bis Abstand-i-Knoten keine unmarkierten Nachfolger/Nachbarn haben

Englisch: Breadth-first search, BFS

Breitensuche: Beispiel

Hier: Besuche Nachfolger mit aufsteigender Knotennummer



Tiefensuche mit Startknoten 0 markiert Knoten in Reihenfolge 0 - 1 - 3 - 2 - 4 - 5

Breitensuche: Algorithmus (konzeptionell)

Einziger Unterschied zu iterativem Tiefensuchalgorithmus:
First-in-first-out-Behandlung der Knoten (statt last-in-first-out)

```
def breadth_first_exploration(graph, node):
    visited = set()
    queue = deque()
    queue.append(node)
    while queue:
        v = queue.popleft()  # FIFO
        if v not in visited:
            visited.add(v)
            for s in graph.successors(v):
                queue.append(s)
```

Breitensuche: Algorithmus (etwas effizienter)

Nur erstes Antreffen eines Knotens wird weiterbetrachtet.
Wir können den Knoten direkt markieren und ihn bei einem weiteren Antreffen sofort verwerfen.

```python
def breadth_first_exploration(graph, node):
    visited = set()
    queue = deque()
    visited.add(node)
    queue.append(node)
    while queue:
        v = queue.popleft()
        for s in graph.successors(v):
            if s not in visited:
                visited.add(s)
                queue.append(s)
```

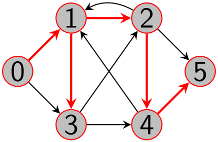
Laufzeit

Bei allen Algorithmenvarianten:

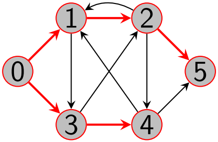
- Jeder erreichbare Knoten wird markiert.
- Man folgt jeder erreichbaren Kante einmal.
- Laufzeit $O(|V| + |E|)$
  - kann man auf erreichbare Knoten und Kanten einschränken

Induzierter Suchbaum

Der induzierte Suchbaum einer Graphenexploration enthält zu jedem besuchten Knoten (ausser dem Startknoten) eine Kante von dessen Vorgänger in der Exploration.



Tiefensuche



Breitensuche

(induzierter Suchbaum $\neq$ binärer Suchbaum)

Induzierter Suchbaum: Beispiel Breitensuche

- Jeder Knoten hat höchstens einen Vorgänger im Baum.
- Repräsentiere induzierten Suchbaum durch Vorgängerrelation
- Besuchte Knoten sind genau die, für die Vorgänger gesetzt ist: Verzichte auf `visited`.

```
def bfs_with_predecessors(graph, node):
    predecessor = [None] * graph.no_nodes()
    queue = deque()
    # use self-loop for start node
    predecessor[node] = node
    queue.append(node)
    while queue:
        v = queue.popleft()
        for s in graph.successors(v):
            if predecessor[s] is None:
                predecessor[s] = v
                queue.append(s)
```

C1.5 Zusammenfassung

- Graphen bestehen aus Knoten und Kanten
- Kanten können gerichtet oder ungerichtet sein.
- Graphenexploration besucht systematisch alle Knoten, die von einem bestimmten Knoten erreichbar sind.
  - Tiefensuche geht zuerst in die „Tiefe“.
  - Breitensuche besucht zuerst die Knoten, die näher am Startknoten sind.