Algorithmische Graphentheorie

Kapitel 3: Suchalgorithmen I

Babeş-Bolyai Universität, Fachbereich Informatik,
Klausenburg

Wiederholung

- Bislang:
  - Definitionen: Was ist ein Graph, was kann man damit untersuchen?
  - Repräsentation von Graphen
- Wichtig ist die Algorithmik und die möglichen Anwendungen!
  - Floyd-Warshall-Algorithmus
  - Algorithmus von Kruskal

Suchalgorithmen: Breitensuche (Breadth-first search)

- Breitensuche (breadth-first search, BFS) durchläuft die Knoten eines Graphen.
- Im Gegensatz zur Tiefensuche werden zunächst alle Knoten beschritten, die vom Ausgangsknoten direkt erreichbar sind. Erst danach werden Folgeknoten beschritten.
- Wichtiges Anwendungsbeispiel in der Graphentheorie: Finde alle Knoten innerhalb einer Zusammenhangskomponente.

- BFS und seine Anwendung zum Auffinden zusammenhängender Komponenten von Graphen wurde 1945 von Konrad Zuse in seiner (abgelehnten) Doktorarbeit über die Programmiersprache Plankalkül erfunden – diese Resultate erschienen erst 1972.
- Wiederentdeckt in 1959 von Moore im Rahmen der Durchquerung von Labyrinthen.
- Lee entdeckte denselben Algorithmus unabhängig von früheren Arbeiten in 1961 in seiner Arbeit über die Leiterbahnführung auf Leiterplatten.

Suchalgorithmen: Breitensuche (Breadth-first search)

BFS(G, s)

Input: Ein (un-) gerichteter Graph G in Adjazenzlistendarstellung; ein Knoten $s \in V(G)$.

Output: Für jeden Knoten $v \in V$ der Abstand dist(s, v) von s zu v.

```
1  for all v ∈ V do
2      Setze d[v] := +∞                    { Alle Knoten sind unentdeckt }
3  d[s] := 0
4  Q := { s }
5  while Q ≠ ∅ do
6      Entferne das Element u aus Q mit kleinstem Schlüsselwert d[u]
7      for all v ∈ ADJ[u] do
8          if d[v] = +∞ then
9              d[v] := d[u] + 1
10             Q := Q ∪ { v }
11 return d[]
```

- Im Algorithmus wird im gerichteten Fall jede gerichtete Kante höchstens einmal in Schritt 7 betrachtet.
- Im ungerichteten Fall wird jede Kante höchstens zweimal betrachtet, einmal für jeden Endknoten.
- Damit haben wir Zeit $O(n + m)$ plus die Zeit für die Verwaltung der Menge Q.
- Wir verwalten Q als lineare Liste, wobei in Schritt 6 das erste Element aus Q entfernt und in Schritt 10 die neuen Elemente hinten an die Liste angefügt werden.
- Wenn wir zeigen können, dass die Liste stets aufsteigend sortiert ist, folgt die Korrektheit der Implementierung und zudem die lineare Gesamtlaufzeit, da die Listenoperationen jeweils in konstanter Zeit durchführbar sind.

- Wir zeigen dazu, dass zu jedem Zeitpunkt die Liste sortiert ist und zudem höchstens Knoten mit zwei verschiedenen Markierungen t und $t+1$ für ein $t \in \mathbb{N}$ enthält.
- Dies ist zu Beginn ($Q = \{s\}$) sicherlich korrekt.
- Wird nun in Schritt 6 ein Knoten u mit Schlüssel $d[u] = t$ vom Anfang der Liste entfernt, so werden in Schritt 10 höchstens Knoten v mit $d[v] = d[u] + 1 = t + 1$ hinten an die Liste angefügt.
- Daher folgt die Sortierung der Liste.

- Durch Induktion nach der Anzahl der Markenänderungen in Schritt 9 zeigen wir zunächst, dass für alle $v \in V$ stets $d[v] \geq \text{dist}(s, v)$ gilt. Nenne diese Eigenschaft $(\star)$.
- Vor der ersten Markenänderung gilt diese Eigenschaft wegen $d[s] = 0 = \text{dist}(s, s)$ und $d[v] = +\infty$ für $v \neq s$.
- Wird nun in Schritt 9 die Marke eines Knotens v von $+\infty$ auf $d[u] + 1$ gesetzt, so gibt es eine (gerichtete) Kante von u nach v.
- Jeder Weg von s nach u lässt sich zu einem Weg von s nach v verlängern, daher gilt $\text{dist}(s, v) \leq \text{dist}(s, u) + 1$.
- Nach Induktionsvoraussetzung gilt $d[u] \geq \text{dist}(s, u)$, womit

$$d[v] = d[u] + 1 \geq \text{dist}(s, u) + 1 \geq \text{dist}(s, v)$$

folgt und $(\star)$ gezeigt ist.

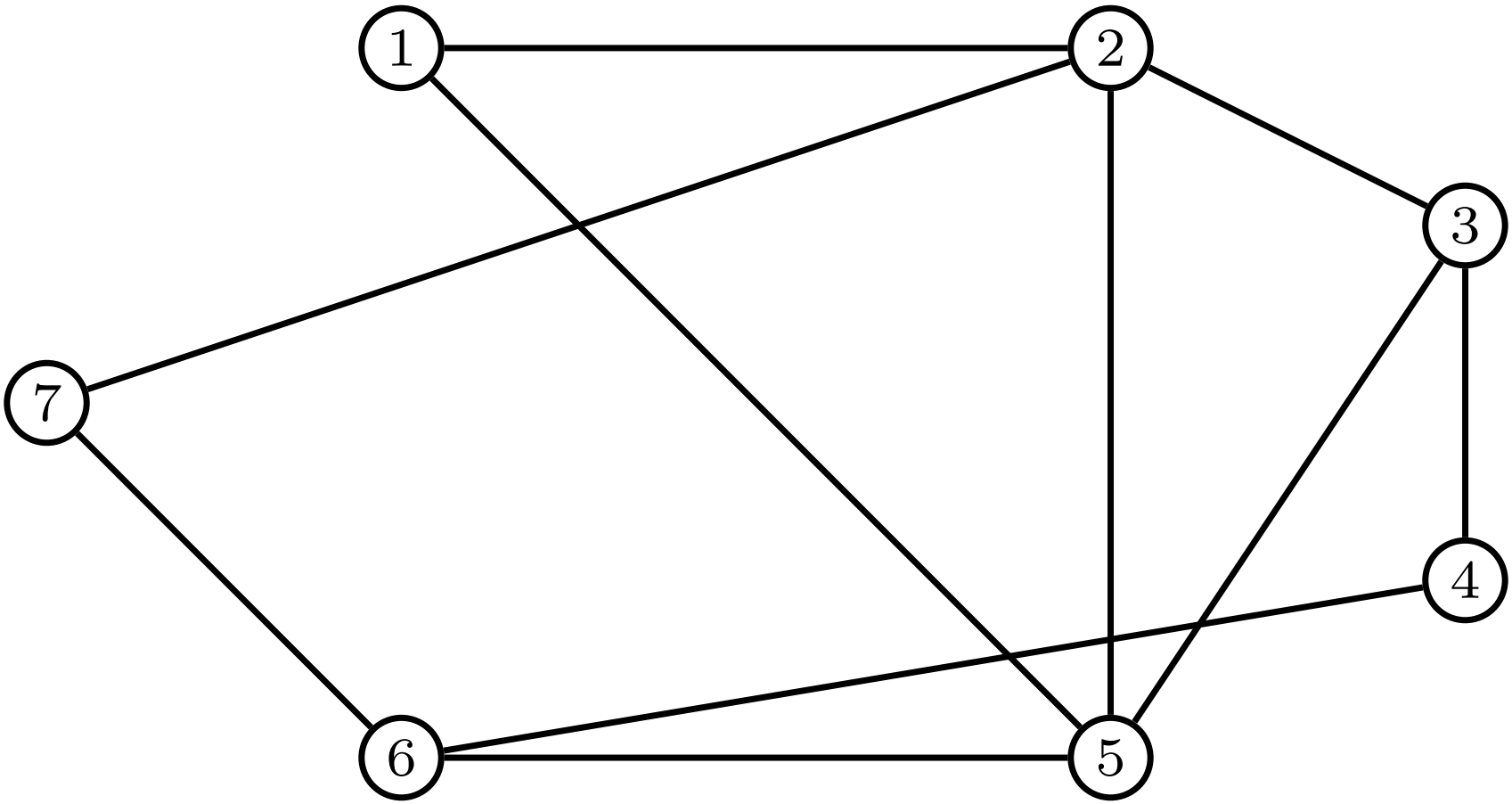
- Wir zeigen nun, wieder durch Induktion (nach k), dass bei Abbruch für alle $v \in V$ mit $\text{dist}(s, v) = k$ auch gilt $d[v] = k$.
- Für $k = 0$ ist nichts zu zeigen.
- Falls $\text{dist}(s, v) = k \geq 1$, so sei P ein kürzester Weg von s nach v mit Länge k.
- Wir schreiben $P = v_0 v_1 \dots v_k$ mit $s = v_0$ und $v_k = v$.
- Dann ist $v_0 v_1 \dots v_{k-1}$ ein kürzester Weg von s nach v_{k-1} (ein kürzerer Weg würde einen Weg von s nach v_k mit Länge $< k$ implizieren, Widerspruch).

- Nach Induktionsvoraussetzung gilt bei Abbruch $d[v_{k-1}] = \text{dist}(s, v_{k-1}) = k - 1$.
- Insbesondere wird v_{k-1} der Menge Q in einer Iteration hinzugefügt.
- Wenn dann $u = v_{k-1}$ in Schritt 6 wieder aus Q entfernt wird, gilt nach Induktionsvoraussetzung $d[u] = \text{dist}(s, u) = k - 1$ und es wird in der anschließenden Iteration über ADJ[u] entweder

$$d[v_k] = d[v_{k-1}] + 1 = (k-1) + 1 = k = \text{dist}(s, v_k)$$

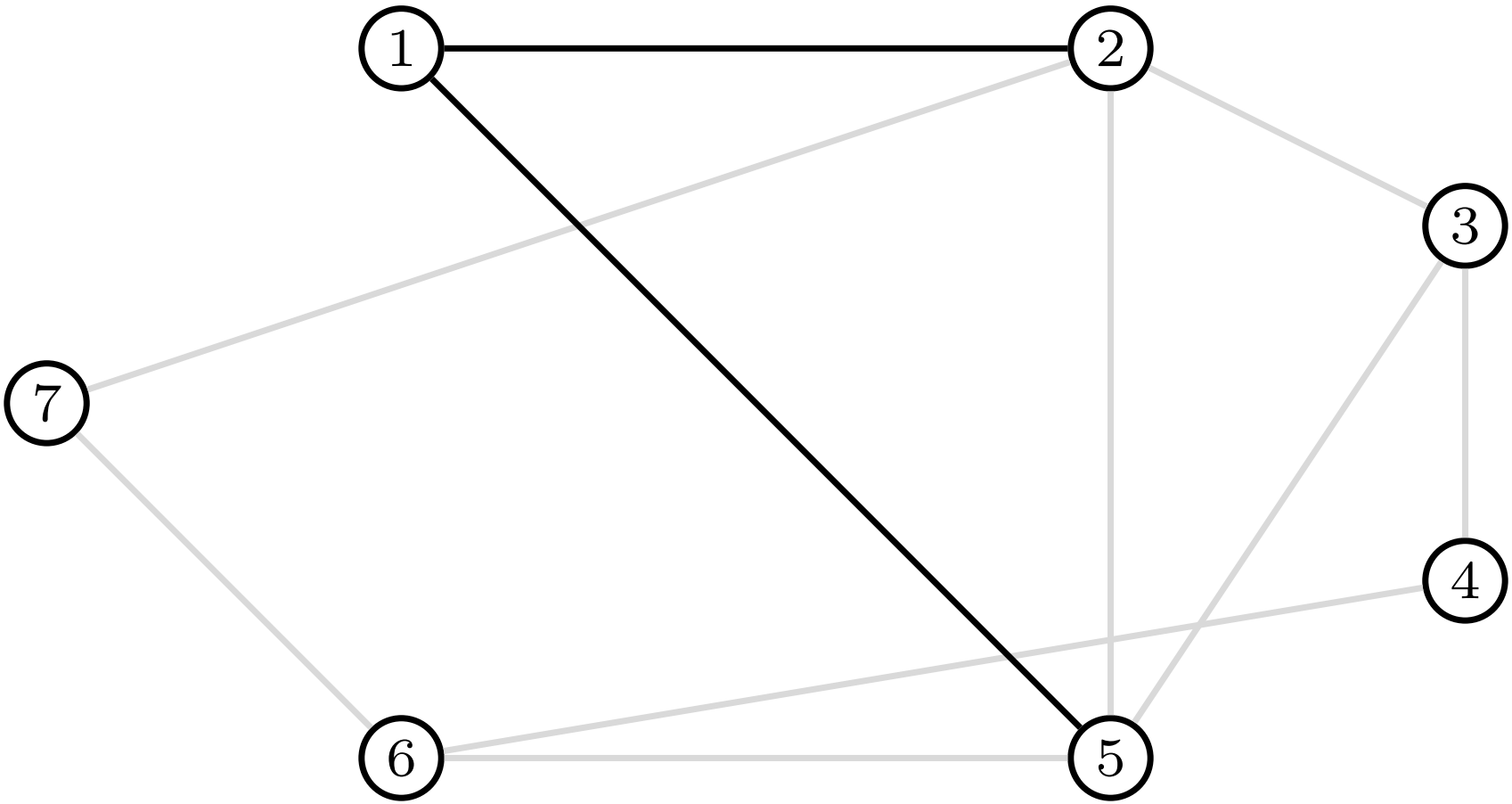
  gesetzt,
- oder v_k wurde bereits zu einem früheren Zeitpunkt Q hinzugefügt. Dann gilt aber $d[v_k] \leq d[u] = k - 1$ im Widerspr. zu $(\star)$, wonach $d[v_k] \geq \text{dist}(s, v_k) = |E(P)| = k$.
- Der Algorithmus bestimmt also korrekt die Abstände von s in einem (un)gerichteten Graphen und er kann so implementiert werden, dass seine Laufzeit $O(n + m)$ beträgt.

BFS: Beispiel für ungerichtete Graphen (1/6)



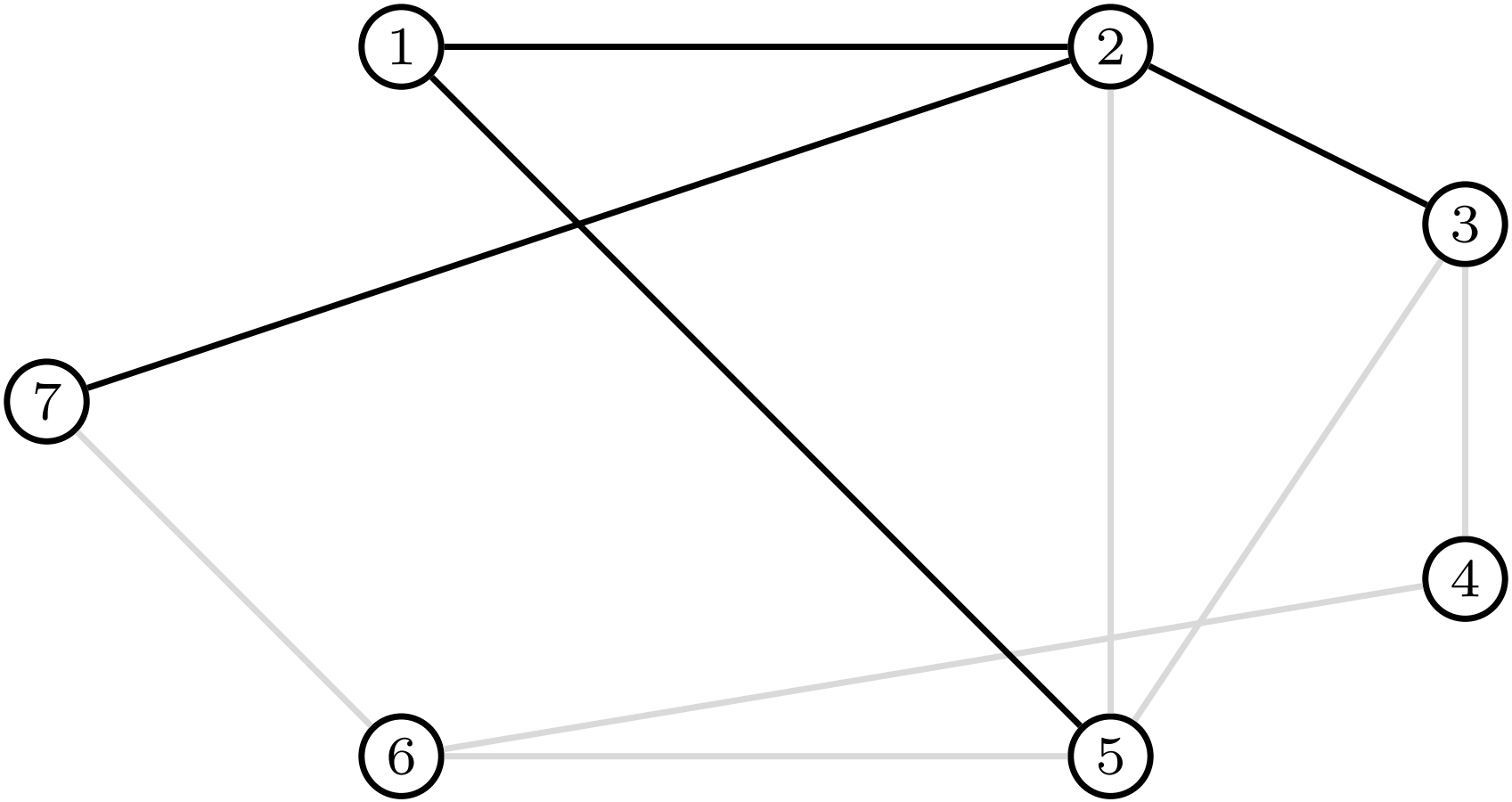
Ein ungerichteter Graph, s = Knoten 1

BFS: Beispiel für ungerichtete Graphen (2/6)



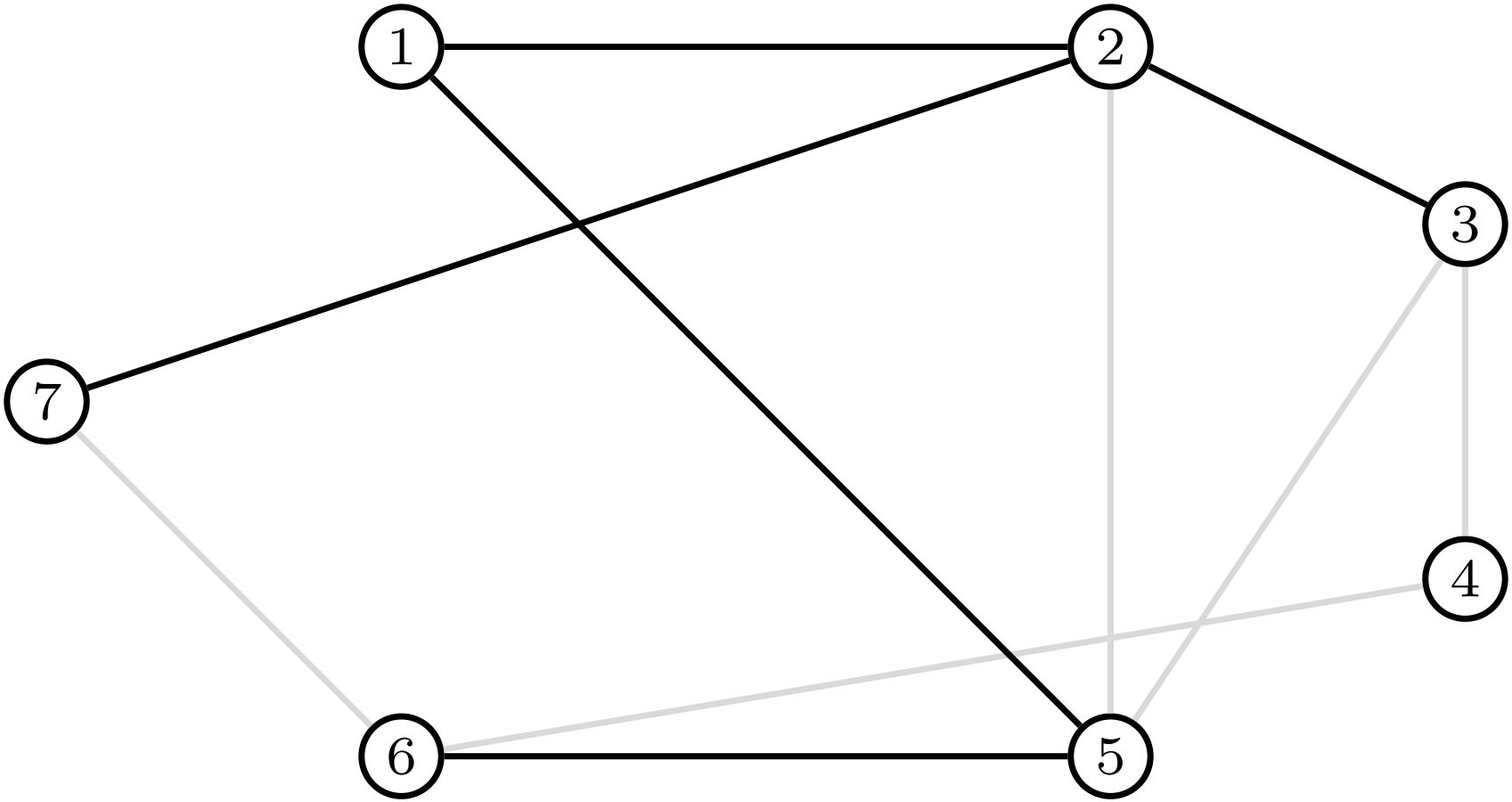
Erste Iteration

BFS: Beispiel für ungerichtete Graphen (3/6)



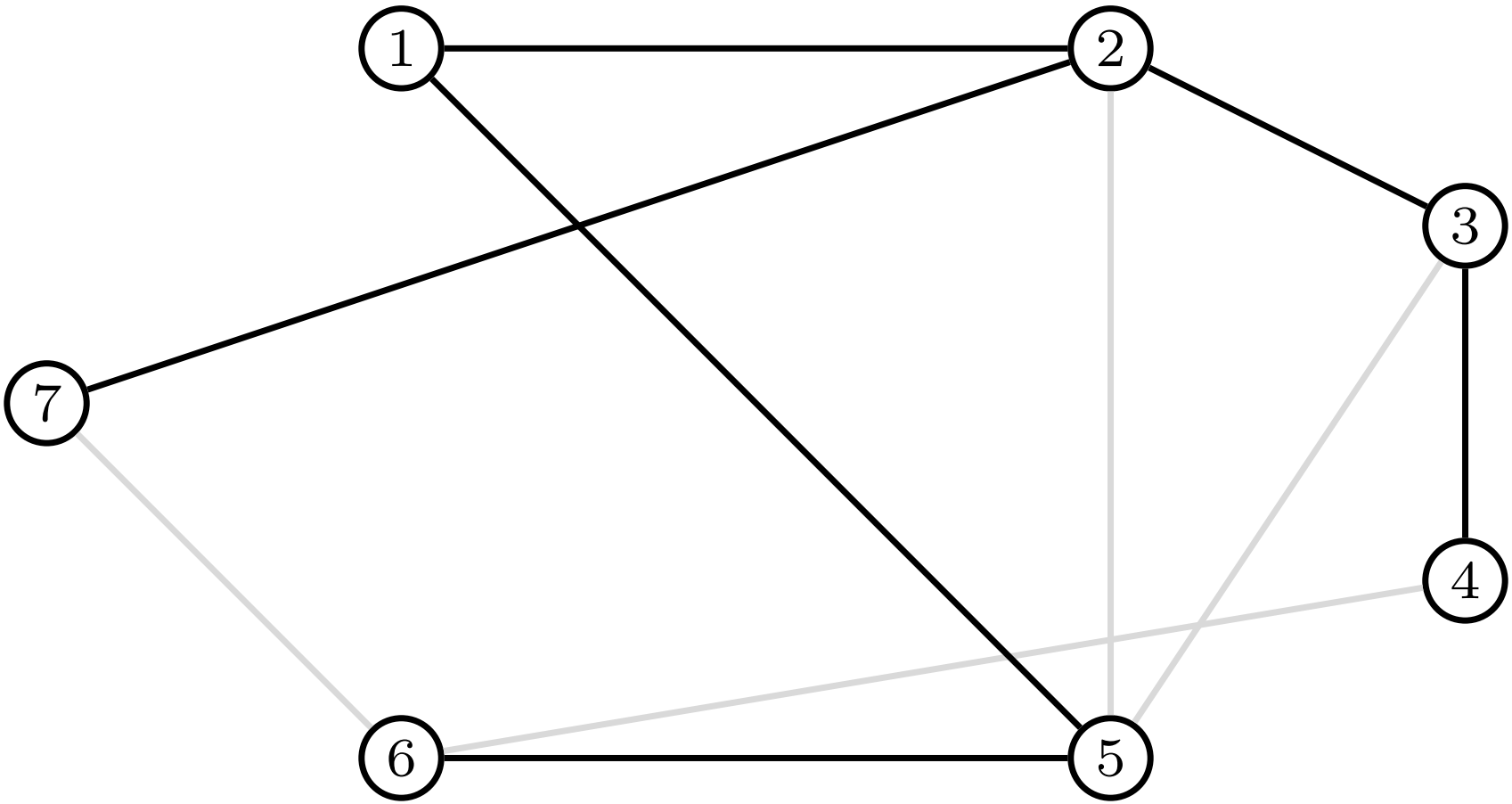
Zweite Iteration, 1. Nachbar (Knoten 2)

BFS: Beispiel für ungerichtete Graphen (4/6)



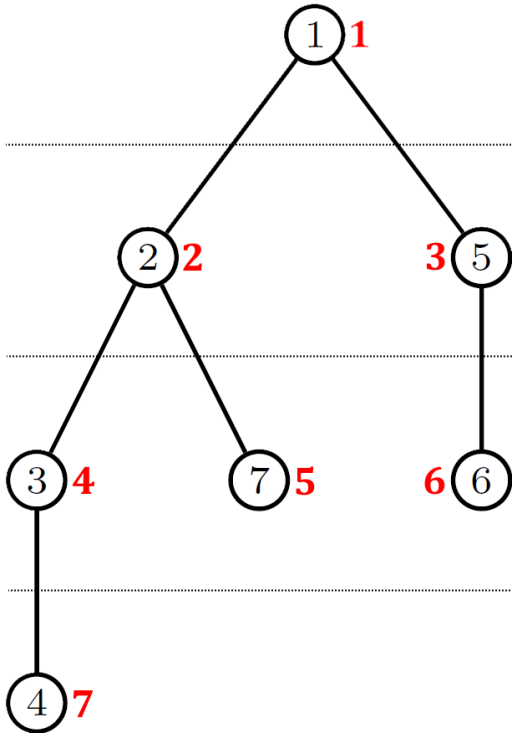
Zweite Iteration, 2. Nachbar (Knoten 5)

BFS: Beispiel für ungerichtete Graphen (5/6)



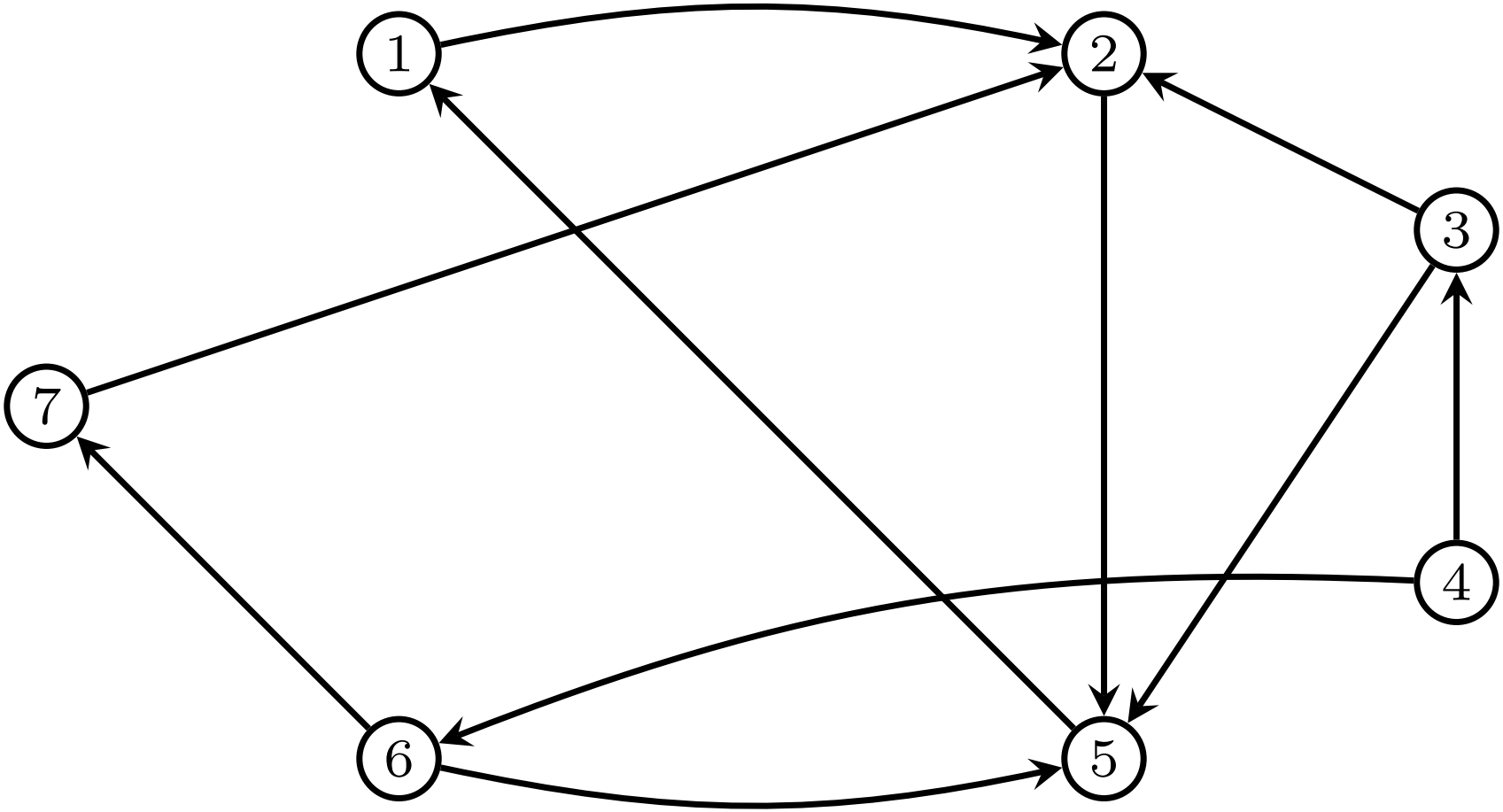
Dritte Iteration

BFS: Beispiel für ungerichtete Graphen (6/6)



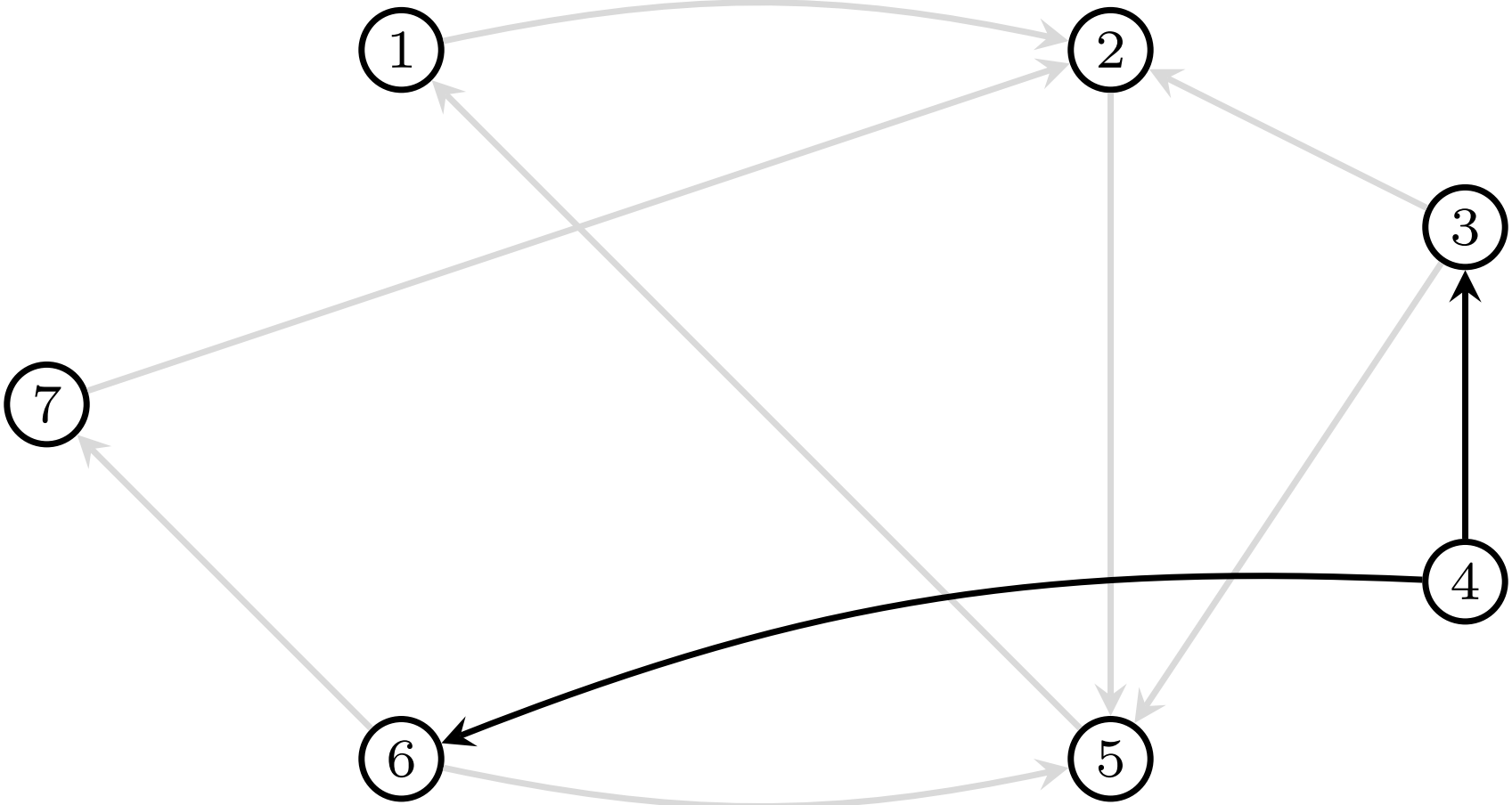
Baum der Breitensuche. In rot die Reihenfolge, in der Knoten besucht wurden. Dies ist ein Spannbaum des ursprünglichen Graphen. An diesem Baum können Sie auch unmittelbar den Abstand zwischen $s = 1$ und einem beliebigen Knoten v ablesen: er ist gleich dem Niveau, auf dem sich v befindet (man zählt abwärts und beginnt bei 0).

BFS: Beispiel für gerichtete Graphen (1/7)



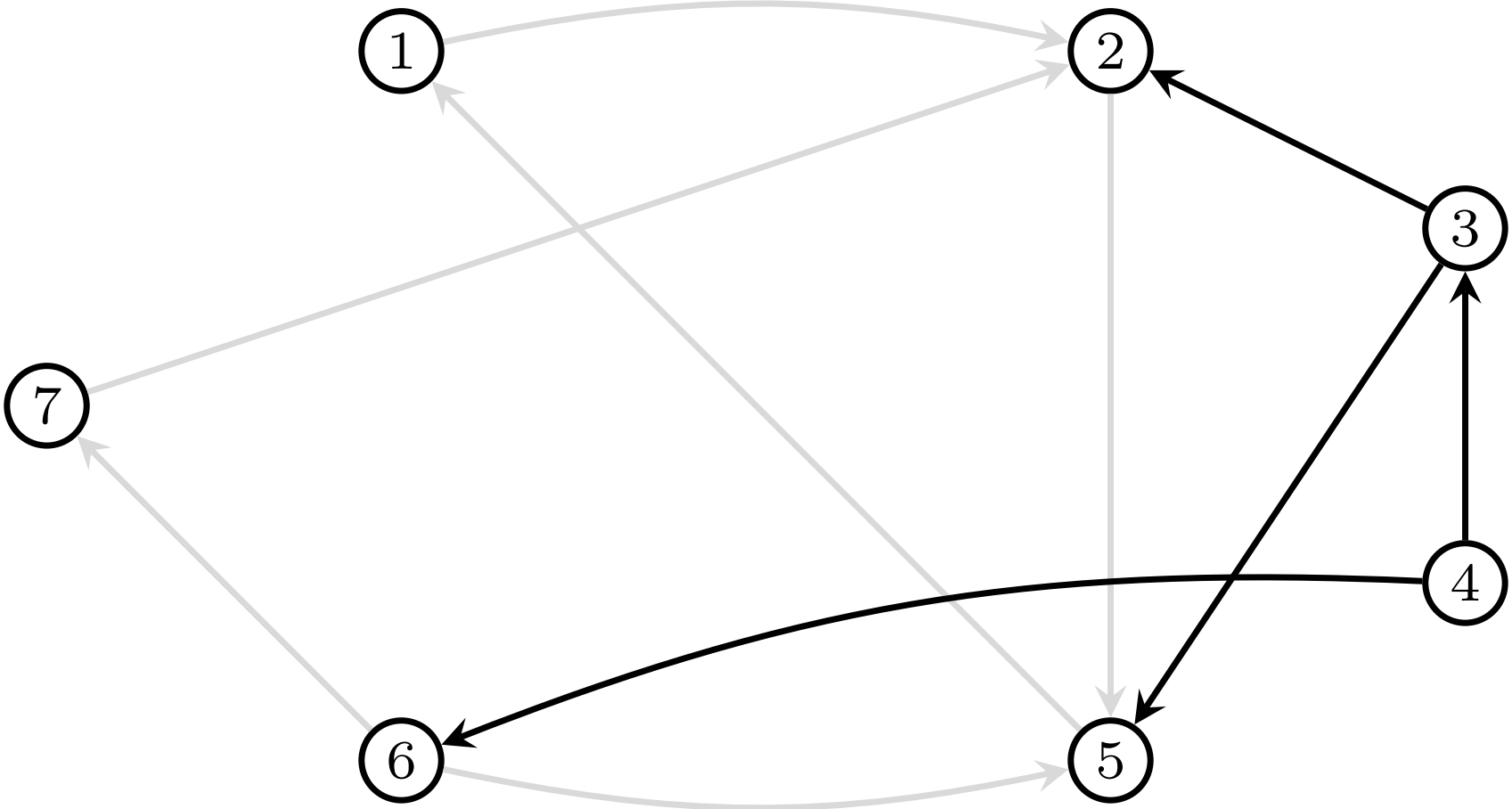
Ein gerichteter Graph, Startknoten s = Knoten 4

BFS: Beispiel für gerichtete Graphen (2/7)



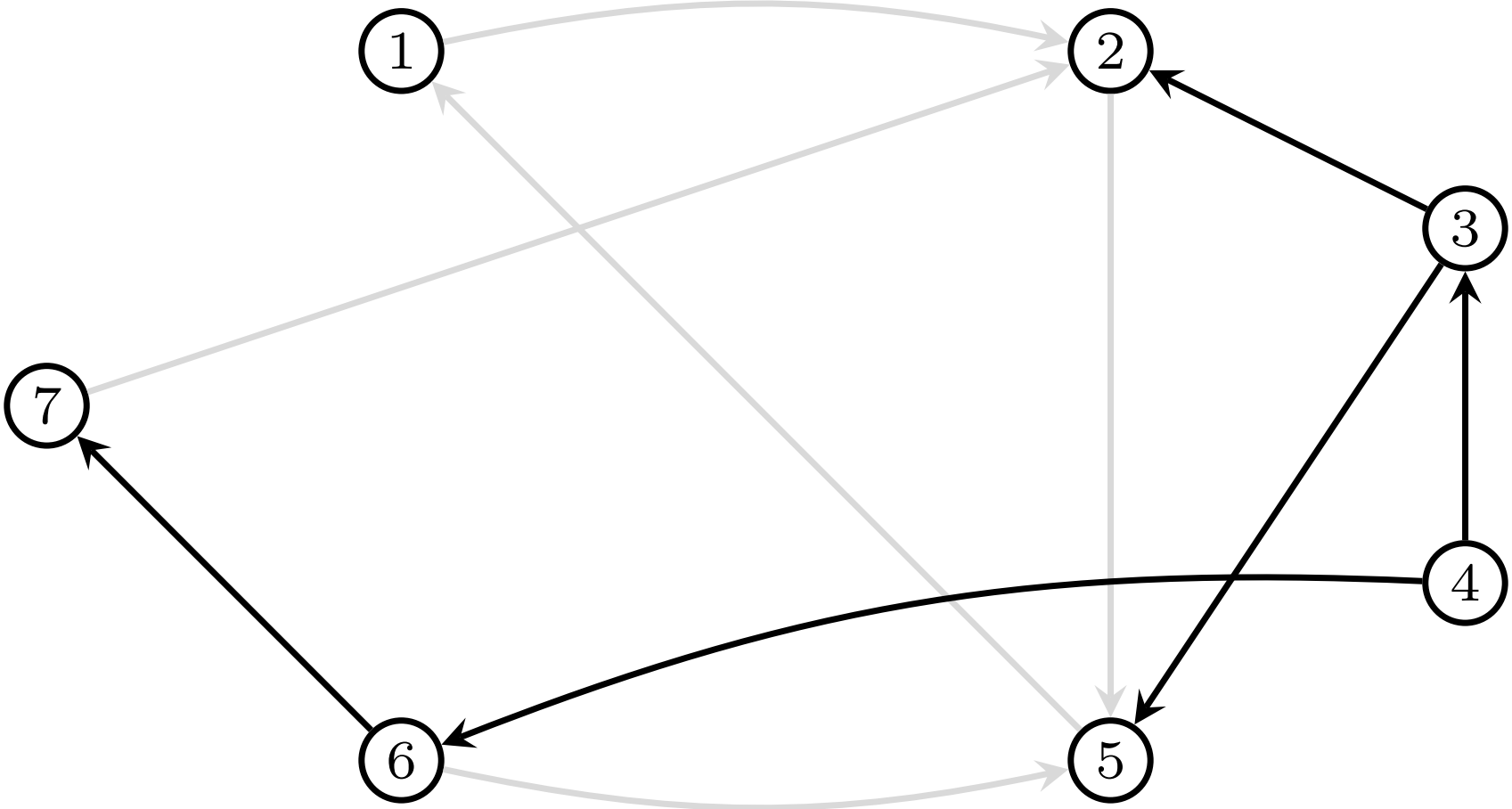
Erste Iteration

BFS: Beispiel für gerichtete Graphen (3/7)



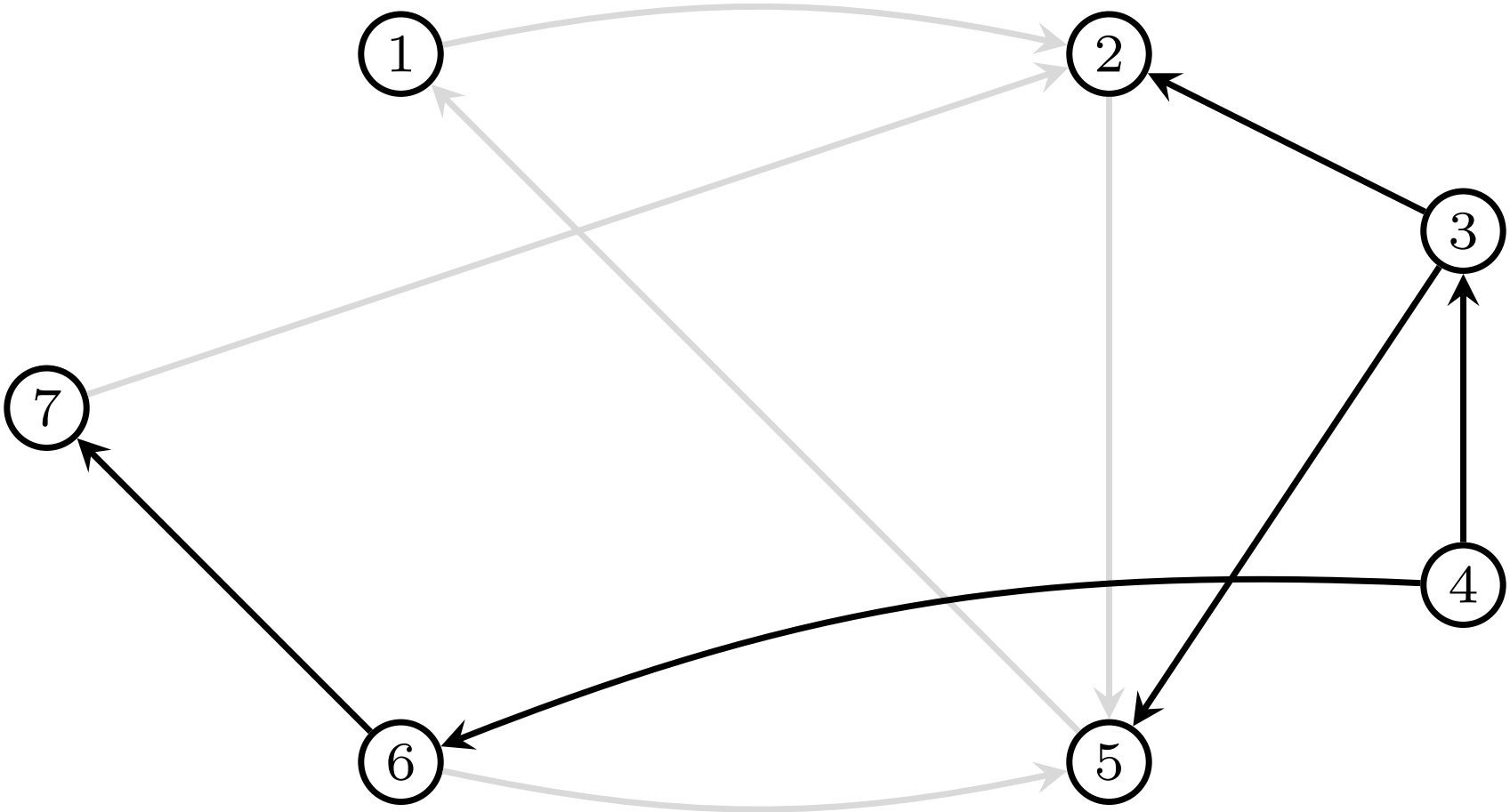
Zweite Iteration, 1. Nachbar (Knoten 3)

BFS: Beispiel für gerichtete Graphen (4/7)



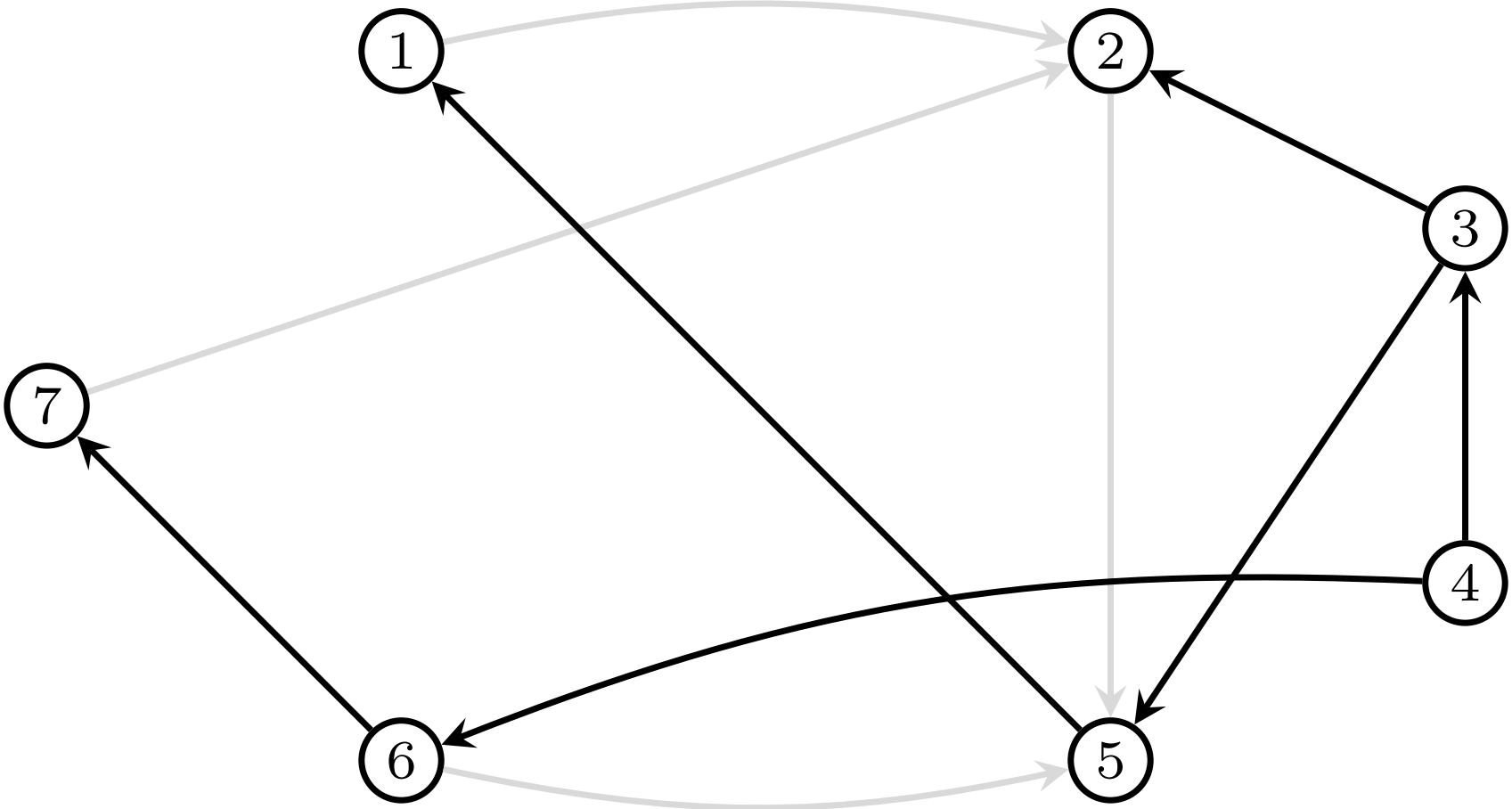
Zweite Iteration, 2. Nachbar (Knoten 6)

BFS: Beispiel für gerichtete Graphen (5/7)



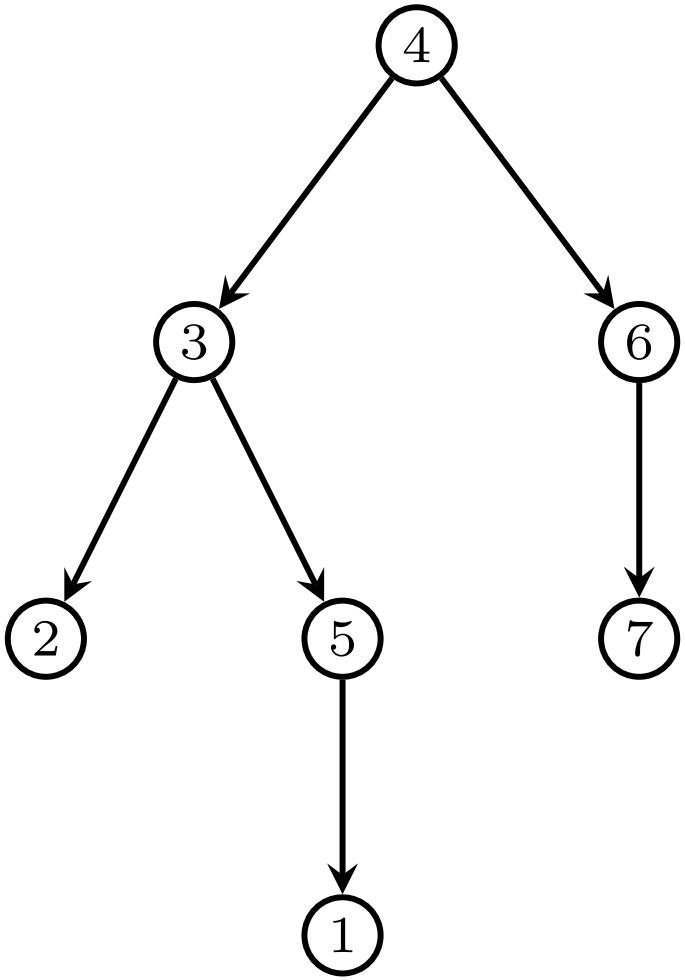
Dritte Iteration, 1. Nachbar (Knoten 2) – nichts geschieht

BFS: Beispiel für gerichtete Graphen (6/7)



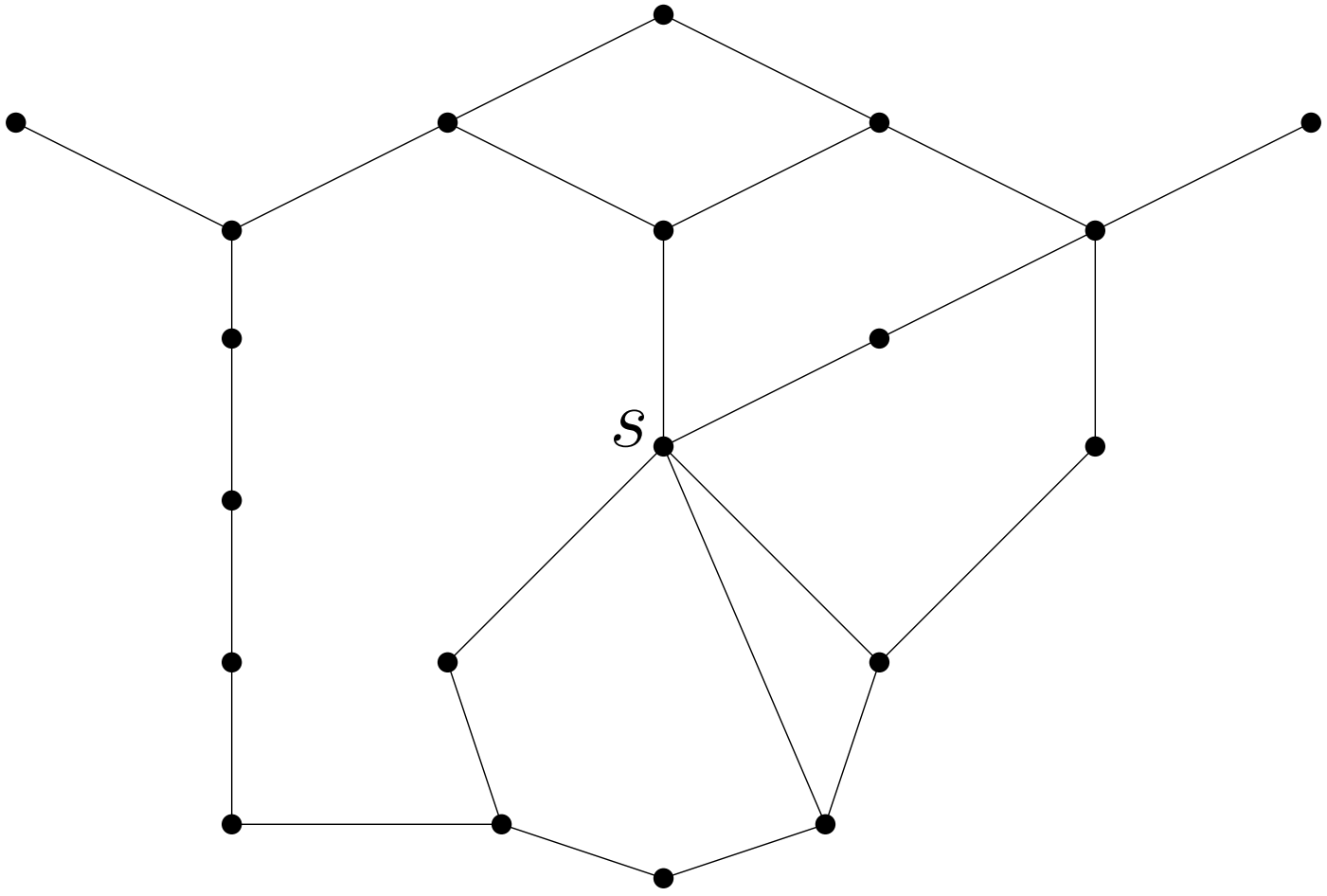
Dritte Iteration, 2. Nachbar (Knoten 5)

BFS: Beispiel für gerichtete Graphen (7/7)



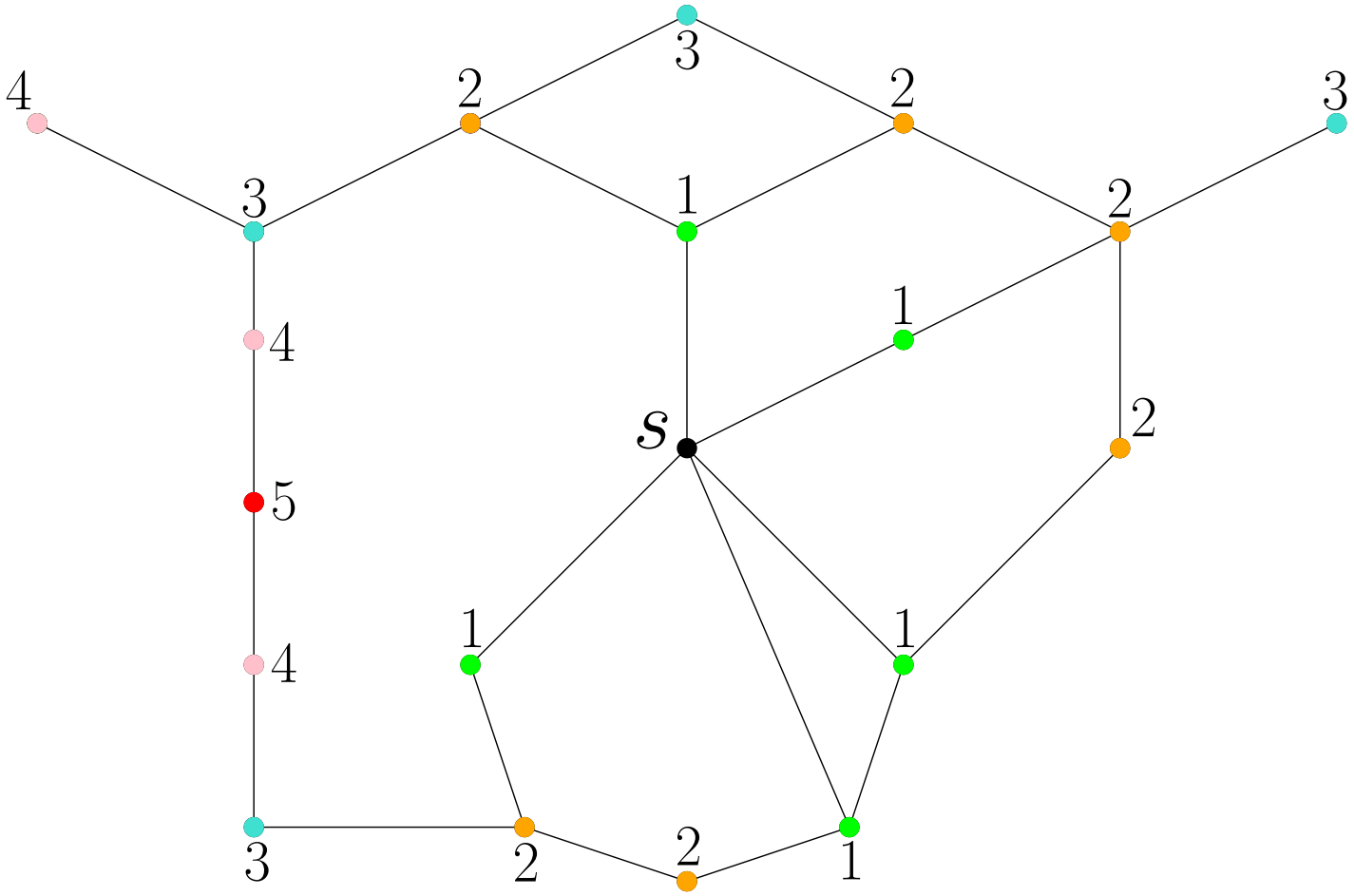
Baum der Breitensuche.

BFS: Beispiel



Aufgabe. Bestimmen Sie mithilfe einer Breitensuche jene(n) Knoten in obigem Graphen, die/der am weitesten von s entfernt ist.

BFS: Beispiel



Der rote Knoten ist der eindeutige, am weitesten entfernte Knoten von s (Entfernung 5).

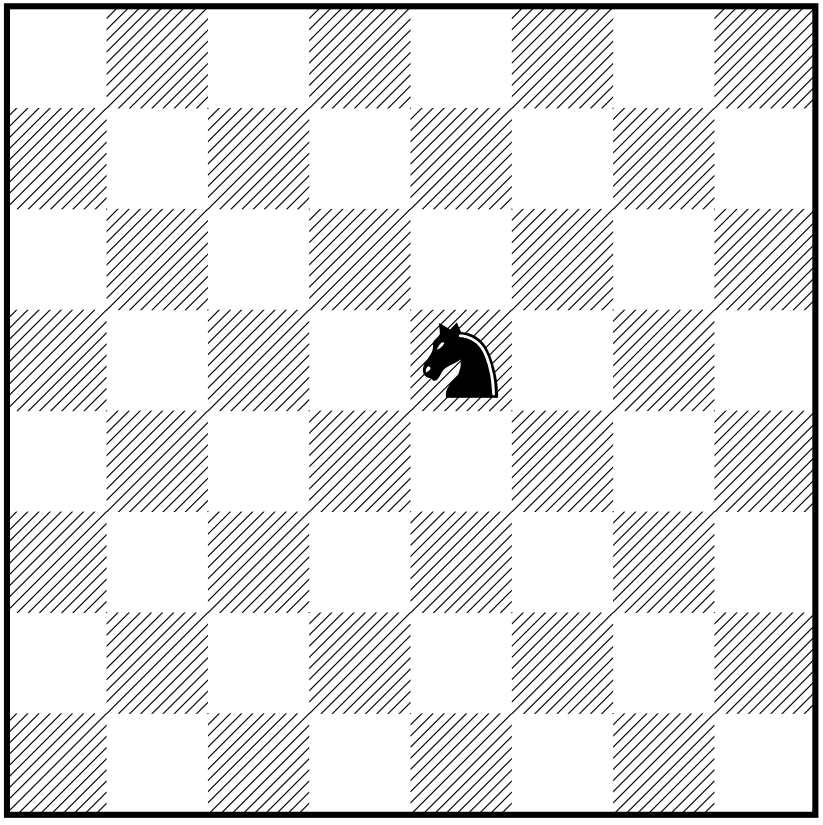
Tiefensuche: Springertour (1/6)

- 8×8 Schachbrett
- Platziere einen Springer und finde eine **Springertour**: eine Folge von Zügen, sodass der Springer genau einmal jedes einzelne Feld besucht (Anfang und Ende brauchen nicht identisch sein).
- Dieses sogenannte Springerproblem ist über 1000 Jahr alt: es findet sich in einem Sanskrit-Gedicht aus dem 9. Jhd. Im Westen wird es in einem Codex des 14. Jahrhunderts der Pariser Nationalbibliothek erwähnt. Leonhard Euler behandelte das Springerproblem 1759 mathematisch.
- Wie findet man eine solche Springertour?
- Backtracking!

Tiefensuche: Springertour (2/6)

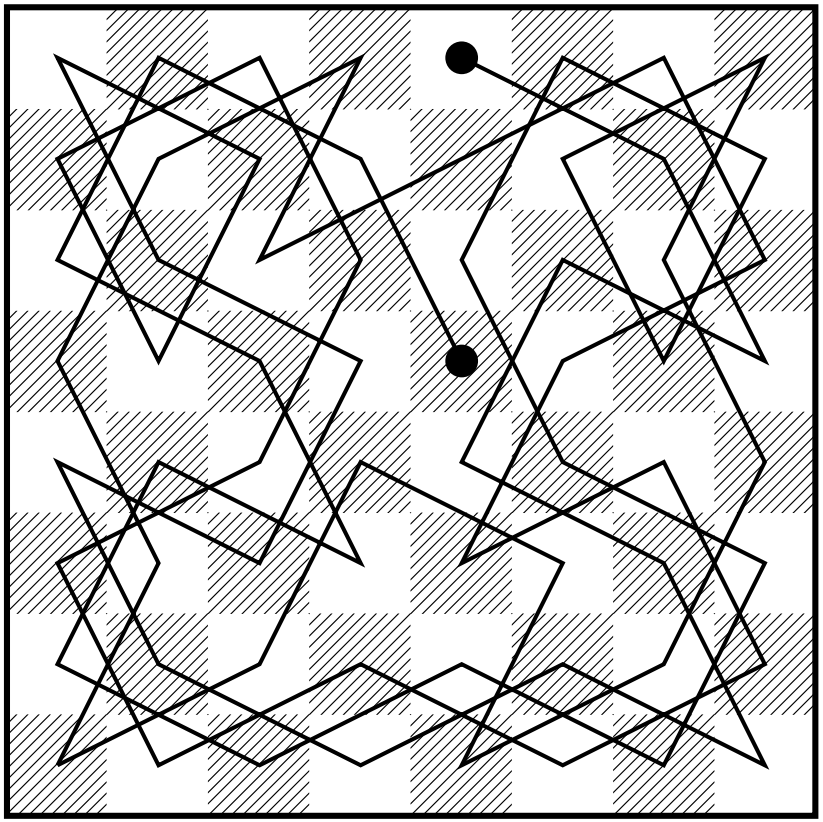
- Während BFS den Suchbaum eine Ebene nach der anderen aufbaut, wird bei der Tiefensuche ein Suchbaum erstellt, indem ein Zweig des Baums so weit wie möglich erforscht wird.
- Diese Tiefenerkundung des Graphen ist genau das, was wir brauchen, um einen Weg zu finden, der genau 63 Kanten hat.
- Wir werden sehen, dass der Tiefensuche-Algorithmus, wenn er eine Sackgasse findet (eine Stelle im Graphen, an der keine Bewegungen des Springers mehr möglich sind), den Baum bis zum nächsttieferen Knoten zurückführt, der es ihm erlaubt, eine legale Bewegung durchzuführen – dies bezeichnet man mit **backtracking**.

Tiefensuche: Springertour (3/6)



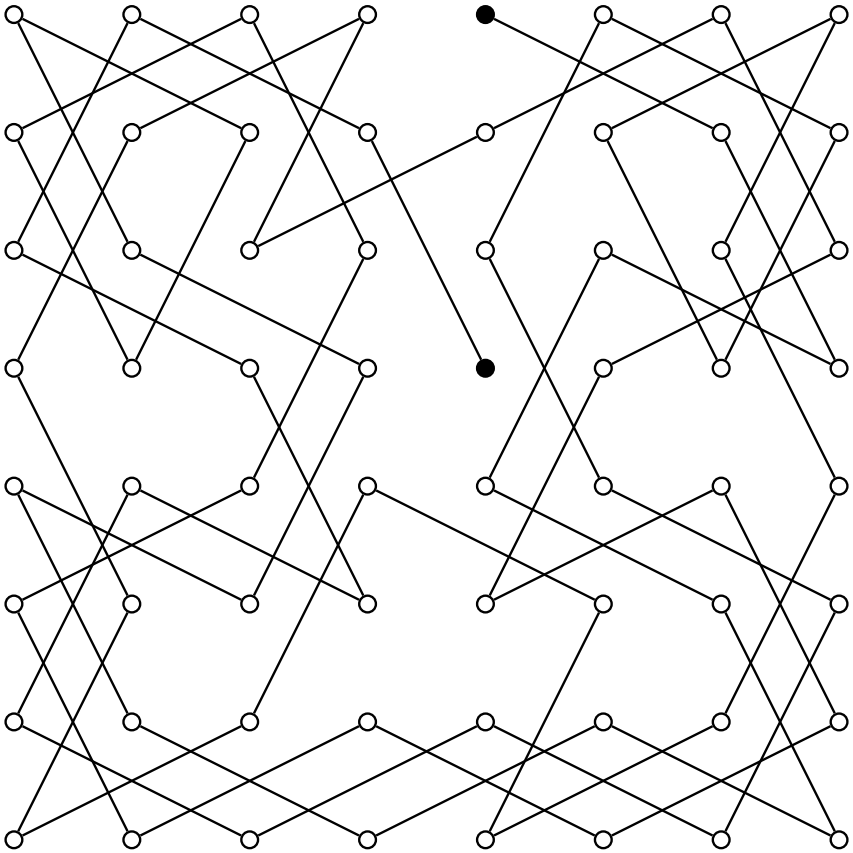
Startposition des Springers

Tiefensuche: Springertour (4/6)



Eine Springertour

Tiefensuche: Springertour (5/6)



Darstellung der Tour als Graph (spezifischer: Weg)

Tiefensuche: Springertour (6/6)

- Wie viele **geschlossene** (d.h. Startknoten = Endknoten) Springertouren existieren auf einem 8×8 Schachbrett? Schätzen Sie mal...

Tiefensuche: Springertour (6/6)

- Wie viele **geschlossene** (d.h. Startknoten = Endknoten) Springertouren existieren auf einem 8×8 Schachbrett? Schätzen Sie mal...
- 13.267.364.410.532 (OEIS A001230)
- Vorsicht ist geboten: das Resultat aus dem Artikel von M. Loebbing und I. Wegener, The Number of Knight's Tours Equals 33,439,123,484,294 – Counting with Binary Decision Diagrams (*Electronic Journal of Combinatorics* **3** (1996), R5) ist nicht korrekt (von den Autoren bestätigt).

Tiefensuche

- Kanten werden ausgehend von dem zuletzt entdeckten Knoten v, der mit noch unerforschten Kanten inzident ist, erforscht.
- Erreicht man von v aus einen noch nicht erforschten Knoten w, so verfährt man mit w genauso wie mit v.
- Wenn alle mit w inzidenten Kanten erforscht sind, erfolgt ein Backtracking zu v.
- Die Tiefensuche durchläuft den Graphen in die **Tiefe**: Ausgehend vom gegebenen Knoten besucht man dessen ersten, noch unbesuchten Nachfolger, danach dessen ersten noch unbesuchten Nachfolger, usw.
- Wenn ein besuchter Knoten keinen Nachfolger mehr hat, kehrt man zu seinem Vorgänger zurück und sucht bei dessen nächsten unbesuchten Nachfolger weiter.

- Ein Vorteil von BFS ist, dass es *iterativ* und nicht *rekursiv* ist.
- Obwohl rekursive Programme schön zu schreiben und auch sehr kurz sind, bedeutet dies häufig für den Computer viele Funktionsaufrufe und das kann durchaus teurer sein als ein iteratives Programm.

Die Tiefensuche (depth-first search, DFS) ist ein typisches Beispiel für ein rekursives Programm:

Gegeben sei ein Graph $G = (V, E)$ und ein beliebiger Startknoten $s \in V$. Alle Knoten in G sind zu Beginn unmarkiert. Führe die Funktion DFS mit Parameter s aus:

DFS(v) {

markiere v

für alle Nachbarn b von v führe aus:

falls b noch nicht markiert ist:

DFS(b)

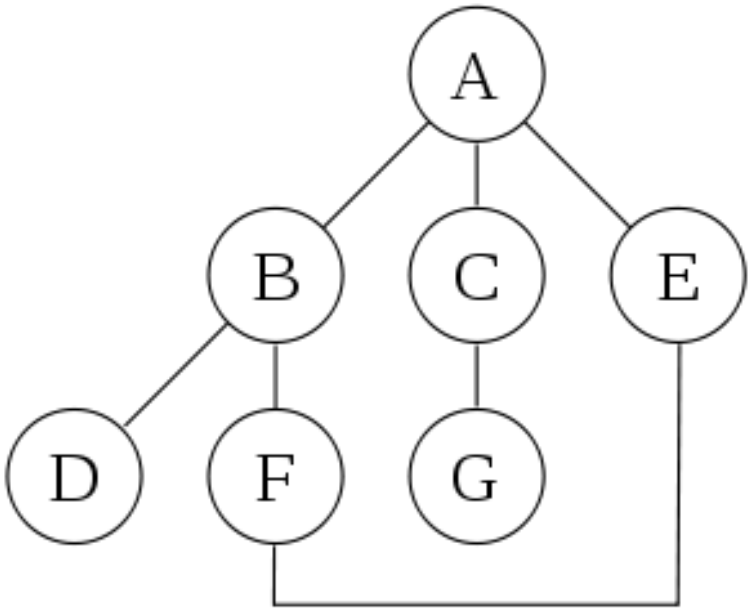
}

Der Zeitaufwand beträgt $O(|V| + |E|)$. Es wird jede Kante höchstens zweimal geprüft.

- Eine Version der Tiefensuche wurde bereits im 19. Jhd. von Charles Pierre Trémaux als Strategie zur Lösung von Labyrinthen untersucht.
- Der Link unten zeigt eine Animation der Erstellung eines Labyrinths mit Hilfe eines Algorithmus zur Generierung von Suchlabyrinthen in der Tiefe – eine der einfachsten Methoden zur Generierung eines Labyrinths mit Hilfe eines Computers.
- Auf diese Weise erzeugte Labyrinthe haben einen relativ geringen Verzweigungsfaktor und enthalten viele lange Korridore, was sie für die Erzeugung von Labyrinthen in Videospielen gut geeignet macht.

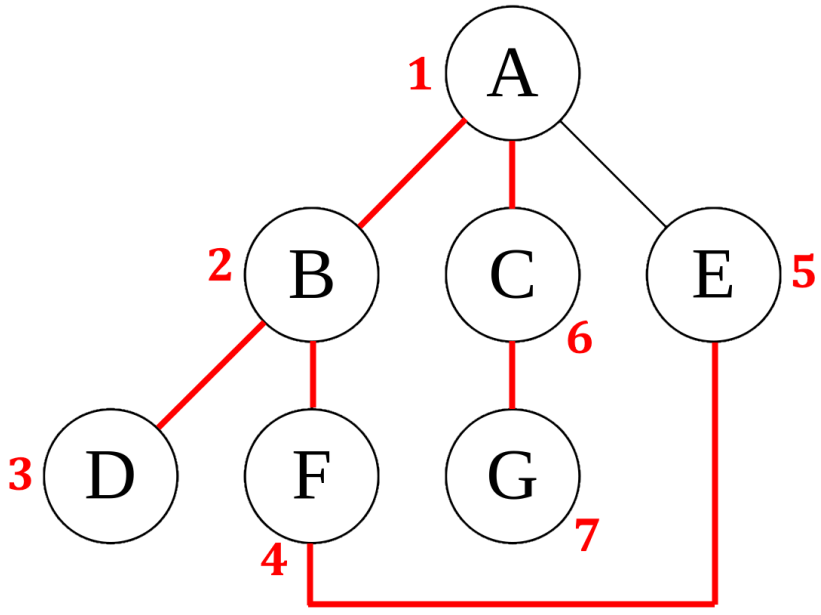
`https://upload.wikimedia.org/wikipedia/commons/4/45/MAZE_30x20_DFS.ogv`

DFS: Beispiel (1/2)



Aufgabe. Geben Sie in obigem Graphen das Resultat einer Tiefensuche an, die im Knoten A startet. Man wähle hier nach A den Knoten B und danach den Knoten D – man könnte diesen Anfang der Tiefensuche natürlich auch anders gestalten. Geben Sie auch den entstehenden Spannbaum und die Reihenfolge an, in der Knoten durchsucht werden.

DFS: Beispiel (2/2)



Graph mit Tiefensuche (Startknoten A). Rote Zahlen geben die Reihenfolge – die sogenannte *DFS-Nummerierung* – an, in der Knoten besucht wurden. Rote Kanten geben die Kanten des Spannbaums an, der durch die Tiefensuche erzeugt wurde.

Tiefensuche

Analog zu BFS kann auch der DFS-Algorithmus für die Tiefensuche gerichteter Graphen angewandt werden:

DFS(v) {

markiere v

für alle Knoten b, für die es eine gerichtete Kante vb gibt, führe aus:

falls b noch nicht markiert ist:

DFS(b)

}

Tiefensuche

Proposition. *Es sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph mit Startknoten s oder gerichteter Graph mit Wurzel s. Dann ist der DFS-Algorithmus wohldefiniert und findet für alle Knoten $v \in V$ einen Weg von s nach v.*

Tiefensuche: Anwendung (1/2)

Gegeben sei ein Graph mit n Knoten. Frage: Ist dieser Graph zusammenhängend? Hierfür verwenden wir folgende Prozedur, die auf DFS beruht:

- Angenommen, die Knoten von G sind $v_1, v_2, \ldots, v_n$.
- Starte die Suche bei einem beliebigen Knoten und benenne ihn in Knoten 1 um.
- Falls Knoten 1 keine adjazenten Knoten besitzt, so ist der Graph nicht zusammenhängend.
- Falls doch, dann bezeichne einen der zu Knoten 1 adjazenten Knoten als Knoten 2 und markiere die Kante $\{1, 2\}$ als gelesen.
- Falls Knoten 2 einen anderen adjazenten Knoten als 1 besitzt, so benenne diesen in Knoten 3 um.

Tiefensuche: Anwendung (2/2)

- Falls Knoten 2 keinen anderen adjazenten Knoten besitzt, so gehe zu Knoten 1 zurück und überprüfe, ob Knoten 1 einen anderen adjazenten Knoten besitzt.
- Falls die Antwort NEIN ist und $n > 2$, so ist der Graph nicht zusammenhängend.
- Falls die Antwort JA ist, so bezeichne einen dieser Knoten als Knoten 3 und markiere die Kante $\{1, 3\}$ als gelesen, usw.

Die Prozedur läuft, bis wir alle Knoten umbenannt / nummeriert haben. In diesem Fall ist der Graph zusammenhängend. Falls wir früher zurück zum Knoten 1 gelangen (also weniger als n Knoten wurden umbenannt), dann ist der Graph nicht zusammenhängend.

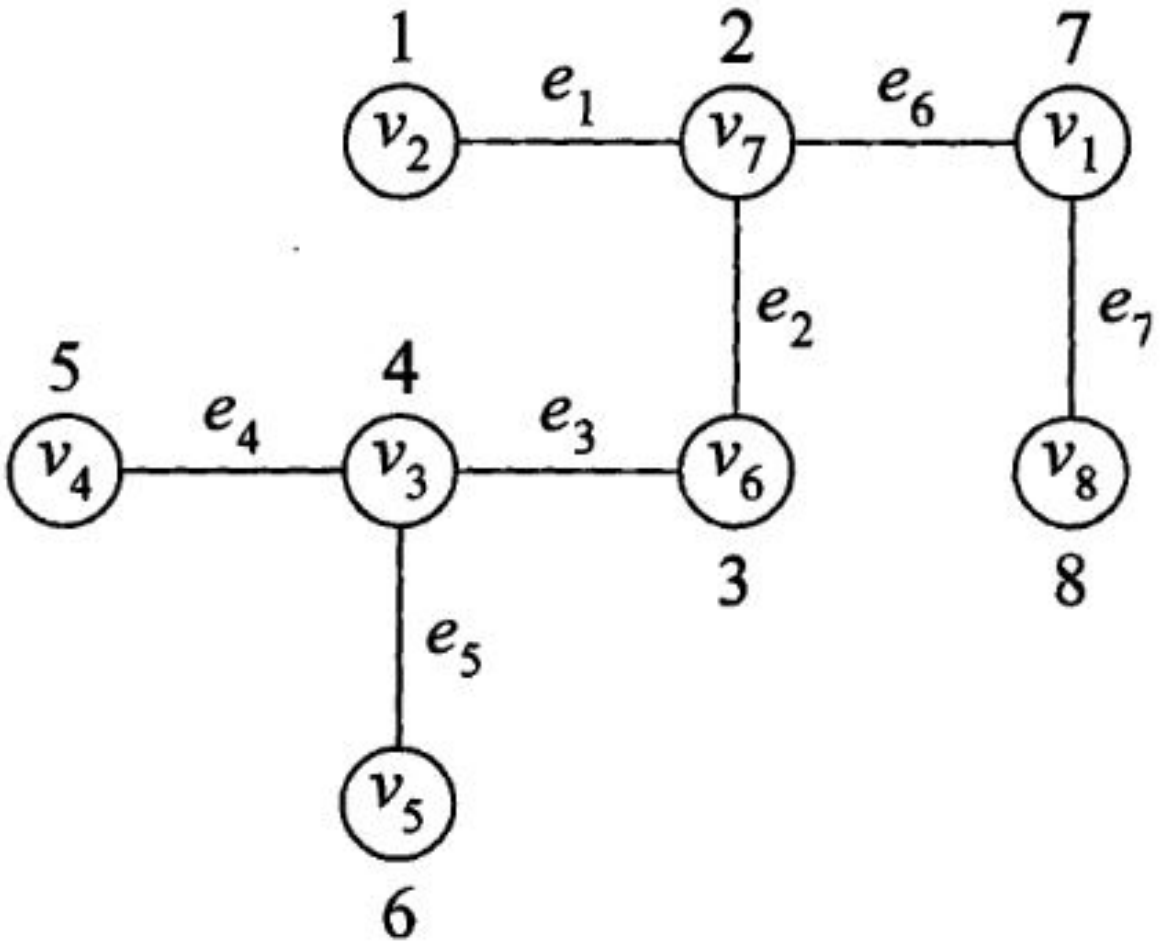
Beispiel (1/3)

Es seien $v_1, \ldots, v_8$ Knoten eines Graphen mit Adjazenzmatrix

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & \underline{1} & 0 \\ 0 & 1 & 0 & \underline{1} & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \underline{1} & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & \underline{1} & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

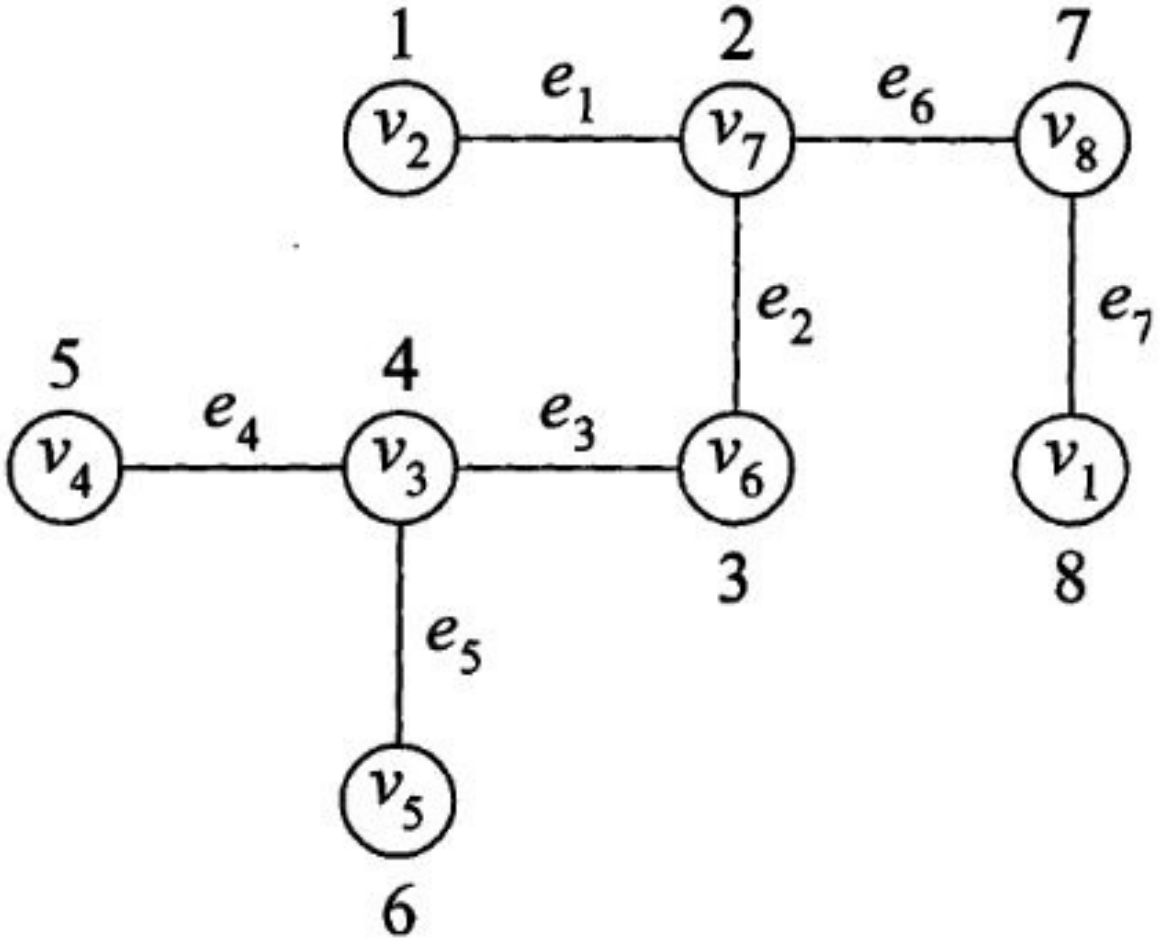
- $v_2 \to 1$
- Ein adjazenter Knoten ist v_7 (s. Matrix). $v_7 \to 2$.
- $v_6 \to 3$, $v_3 \to 4$, $v_4 \to 5$ hat keine andere adjazente Knoten, sodass wir die Prozedur ab Knoten 4 (d.h. v_3) weiterverfolgen.
- $v_5 \to 6$, usw.

BEISPIEL (2/3)



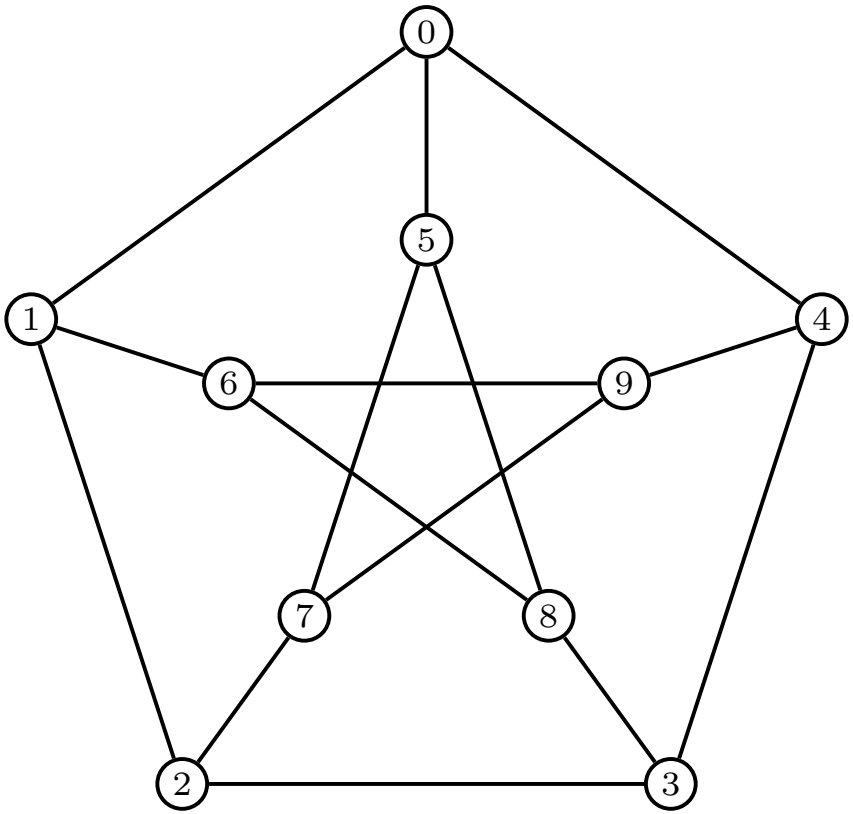
Dies ist die „offizielle“ Lösung, sie ist aber nicht korrekt, denn es gibt keine Kante zwischen v_1 und v_7.

Beispiel (3/3)



Die korrekte Lösung (v_1 und v_8 vertauscht).

DFS und BFS mit Sage



Der Petersen-Graph G (Julius Petersen, 1898)

Wir führen nun, in Sage, auf dem Petersen-Graphen BFS und DFS aus – standardmäßig mit Startknoten 0. Befehle: G.breadth_first_search(0) und G.depth_first_search(0).

DFS und BFS mit Sage

Wir starten mit Knoten 0.

```
G = graphs.PetersenGraph()
print list(G.breadth_first_search(0))
list(G.depth_first_search(0))
```

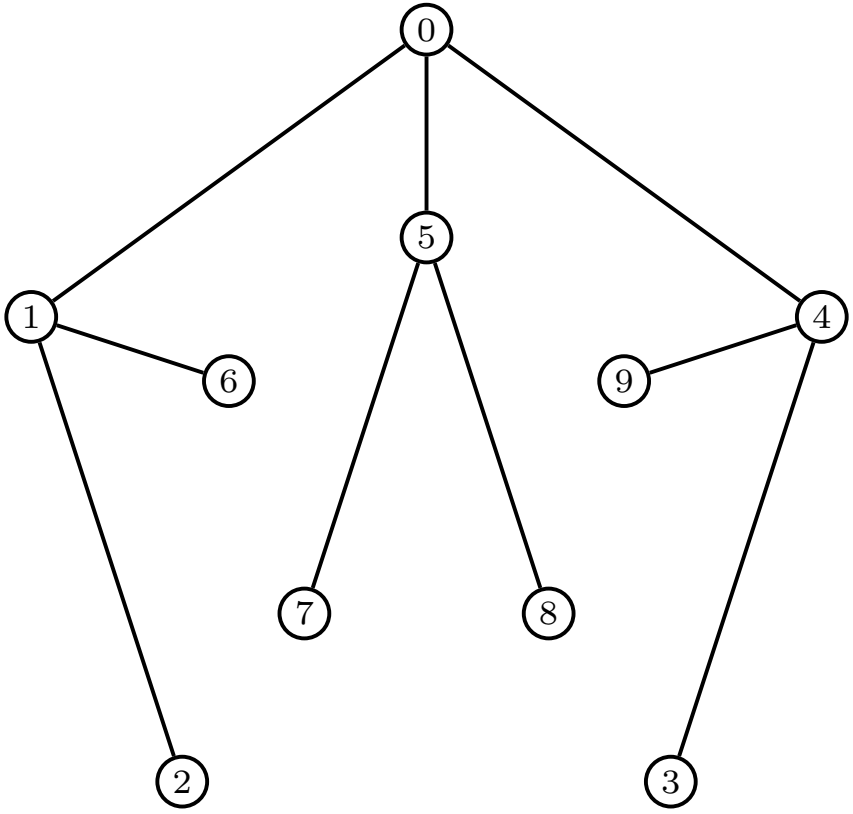
- BFS ergibt die Kantenliste

$$[01, 04, 05, 12, 16, 43, 49, 57, 58]$$

- DFS ergibt die Kantenliste

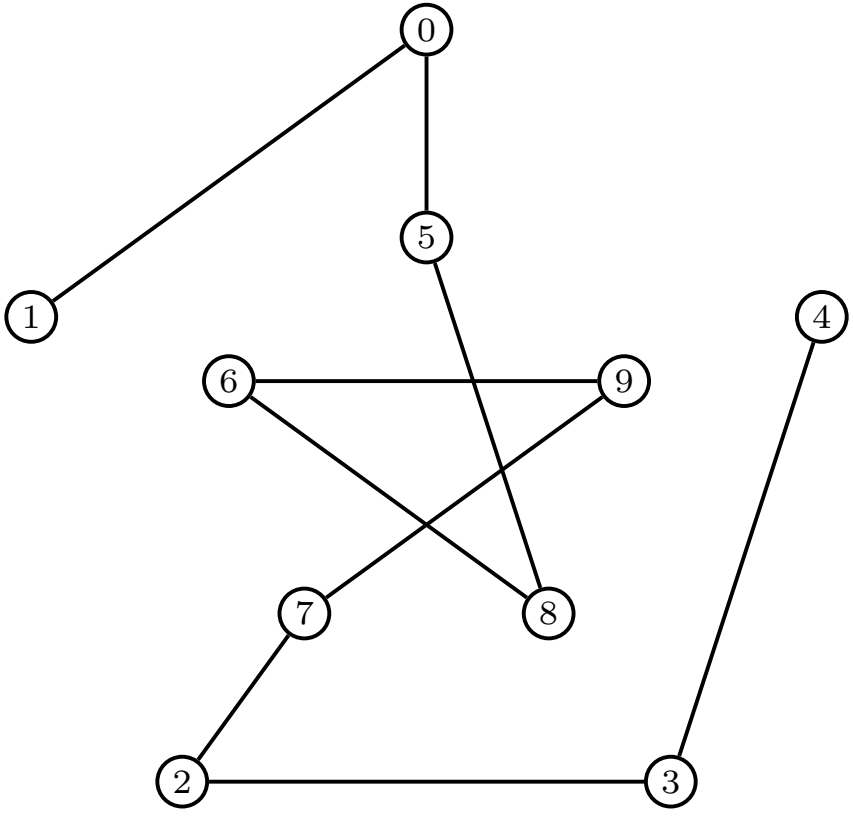
$$[05, 58, 86, 69, 97, 72, 23, 34, 01]$$

DFS und BFS mit Sage



BFS Petersen-Graph. Wir sehen sofort, dass es keinen Knoten gibt, der mehr als 2 Kanten vom Knoten 0 entfernt ist, d.h. man kann von 0 aus jeden anderen Knoten im Graphen mit einem Weg der Länge ≤ 2 erreichen.

DFS und BFS mit Sage



DFS Petersen-Graph. Hier stellen wir unmittelbar fest, dass der Petersen-Graph einen besonderen Spannbaum hat, nämlich einen mit nur 2 Blättern. Diese heißen *Hamiltonwege* und werden in Kapitel 7 behandelt.

Topologisches Sortieren

Es sei V eine Menge. Eine binäre Relation $R \subseteq V \times V$ heißt *Ordnung* auf V gdw. folgende Eigenschaften gelten:

1. *Reflexivität*: $\forall v \in V : (v, v) \in R$
2. *Antisymmetrie*: $\forall v, w \in V : (v, w) \in R \wedge (w, v) \in R$ gilt $v = w$
3. *Transitivität*:
   $\forall u, v, w \in V : (u, v) \in R \wedge (v, w) \in R \Rightarrow (u, w) \in R.$

Topologisches Sortieren: Topologische Ordnung

Wiederholung: Ein gerichteter Graph $G = (V, A)$ heißt *DAG* (*directed acyclic graph*) gdw. G keinen gerichteten Kreis der Länge ≥ 2 enthält.

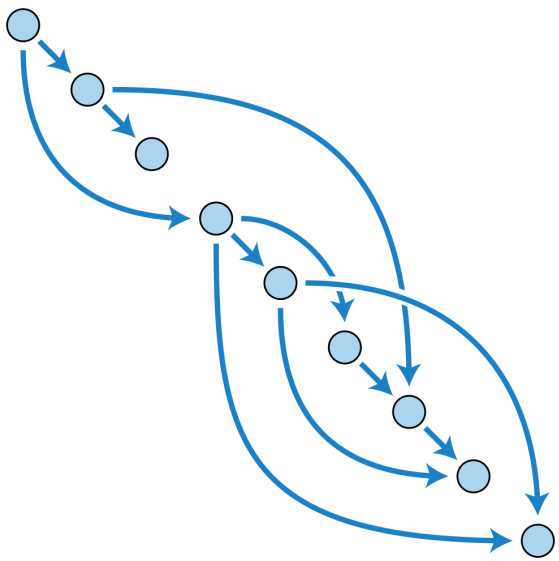
Lemma. *Es sei $G = (V, A)$ ein DAG. Die Relation $R \subseteq V \times V$ mit*

$$(v, w) \in R \Leftrightarrow \text{ es existiert ein gerichteter Weg von } v \text{ nach } w$$

definiert eine Ordnung auf V.

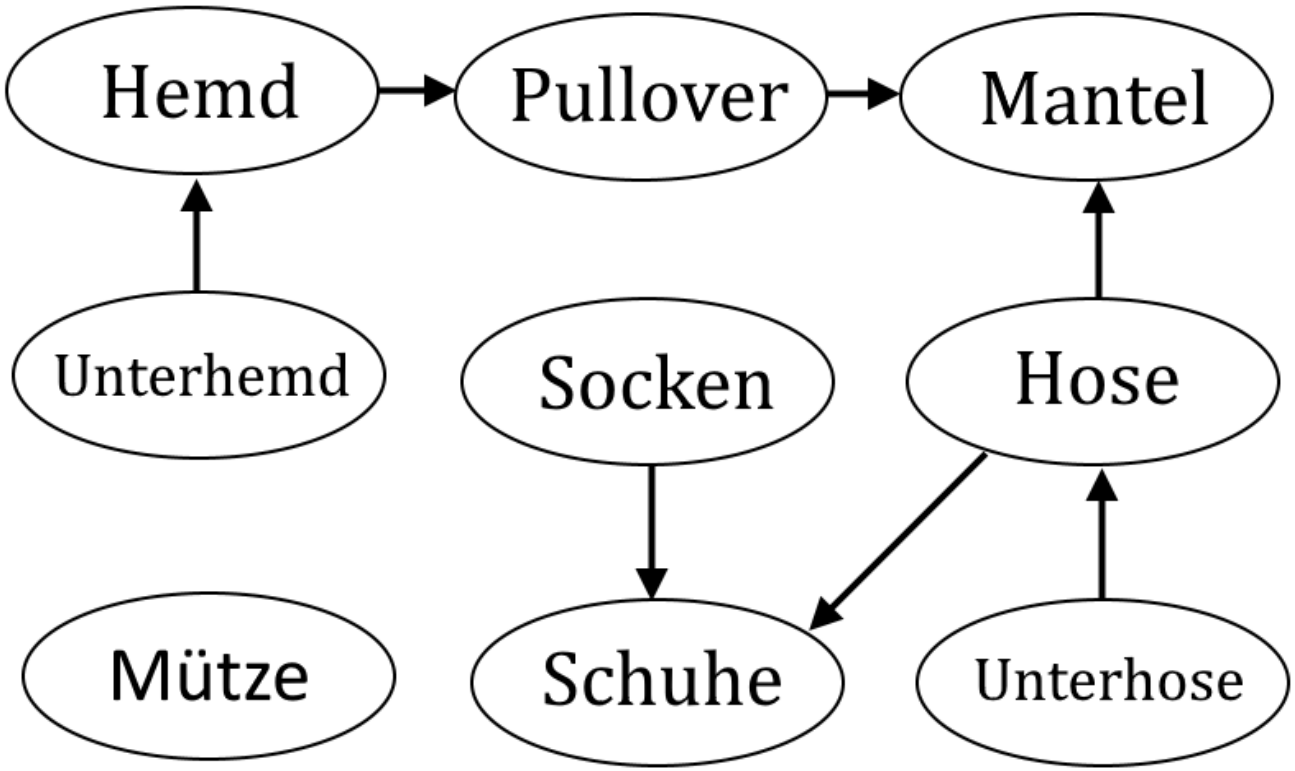
Es sei $G = (V, A)$ ein DAG mit $V = \{v_1, \ldots, v_n\}$. Für eine Permutation π heißt eine Knotenreihenfolge $v_{\pi(1)}, \ldots, v_{\pi(n)}$ aller Knoten aus V *topologische Ordnung* von G gdw.

$$(v_{\pi(i)}, v_{\pi(j)}) \in A \Rightarrow \pi(i) < \pi(j).$$

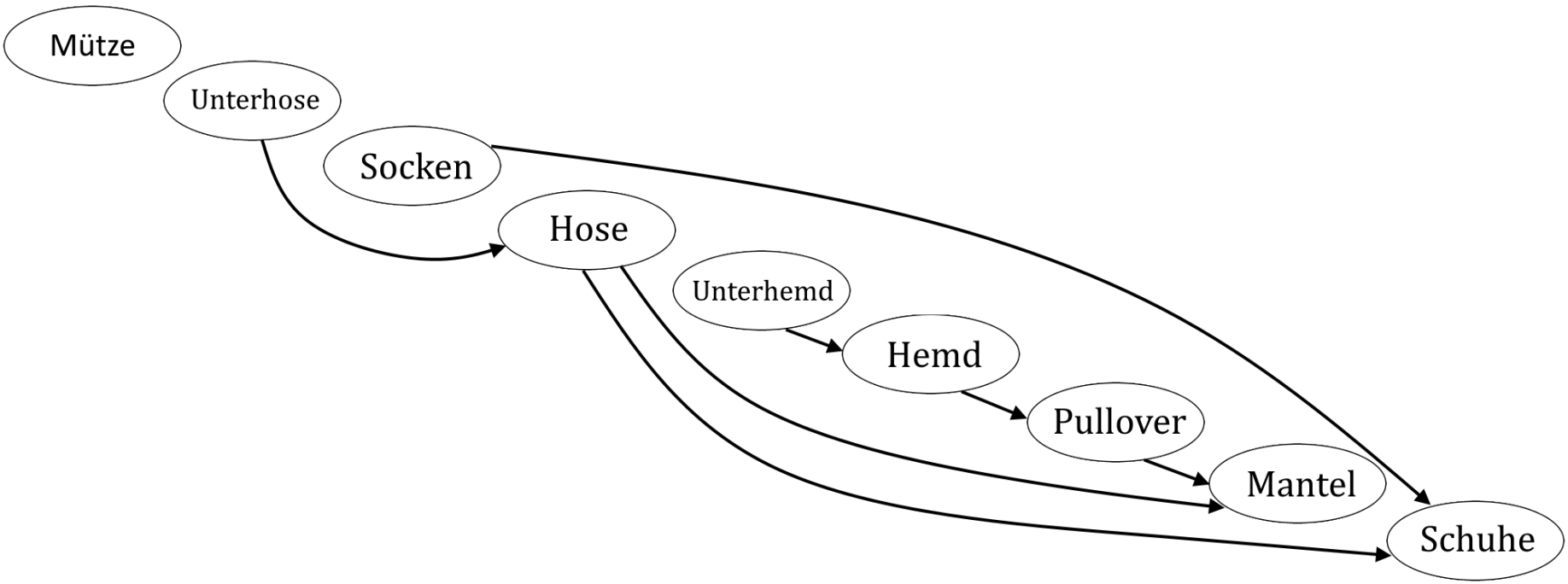


Beispiel (1/2)

Frage. Erlaubt folgender Graph eine topologische Sortierung? Wenn ja, so geben Sie diese an.



Beispiel (2/2)


Mütze
Unterhose
Socken
Hose
Unterhemd
Hemd
Pullover
Mantel
Schuhe

Topol. Sortieren: Algorithmus von Kahn, 1962

Wir wählen Knoten in der gleichen Reihenfolge wie in einer potentiell existierenden topologischen Sortierung.

Zunächst finden wir eine Liste von *Startknoten*, die keine eingehenden Kanten haben, und fügen diese zu einer Menge S hinzu – es muss mindestens einen solchen Knoten geben in einem nicht leeren, kreisfreien Graphen.

L = Leere Liste, die die sortierten Elemente beinhalten wird
S = Menge aller Knoten ohne eingehende Kante

while $S \neq \emptyset$ führe aus:
    entferne einen Knoten v aus S
    füge v an das Ende von L hinzu
    für jeden Knoten w mit einer Kante $e = (v, w)$ führe aus:
        entferne Kante e aus dem Graphen
        if w keine anderen eingehenden Kanten hat, **then**
            füge w zu S hinzu

if Graph hat Kanten
    return error (Graph enthält gerichteten Kreis)
else
    return L (eine topologische Sortierung)

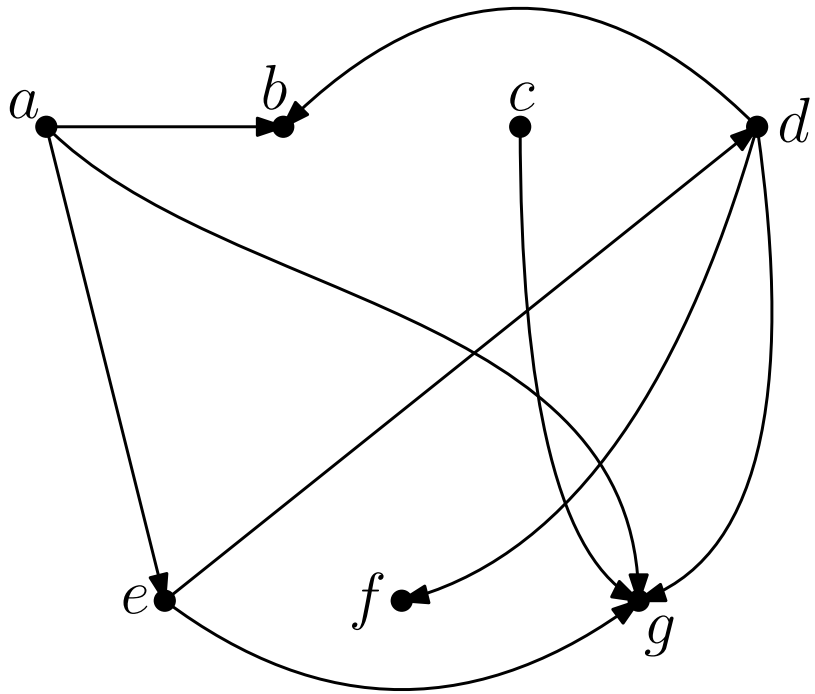
- Aufwand ist $O(|V| + |E|)$.
- Wenn es sich bei dem Graphen um einen DAG handelt, ist eine Lösung in der Liste L enthalten (die Lösung ist nicht unbedingt eindeutig).
- Andernfalls muss der Graph mindestens einen gerichteten Kreis haben, womit eine topologische Sortierung unmöglich ist.

- Wenn eine topologische Sortierung eines DAG G die Eigenschaft hat, dass alle Paare von aufeinanderfolgenden Knoten in der sortierten Folge durch Kanten verbunden sind, dann bilden diese Kanten einen gerichteten *Hamiltonweg* (dies ist ein Weg, der alle Knoten des Graphen besucht) in G.
- Wenn ein Hamiltonweg existiert, ist die topologische Sortierreihenfolge eindeutig – keine andere Reihenfolge ist kompatibel mit den Kanten des Weges.

- Wenn umgekehrt eine topologische Sortierung keinen Hamiltonweg bildet, dann hat der DAG G zwei oder mehr gültige topologische Sortierungen: man vertausche zwei aufeinanderfolgende Knoten, die nicht durch eine Kante miteinander verbunden sind.
- Daher ist es möglich, in linearer Zeit zu testen, ob eine eindeutige topologische Sortierung existiert und ob ein Hamiltonweg existiert, trotz der NP-Härte des Hamiltonweg-Problems für allgemeine gerichtete Graphen.

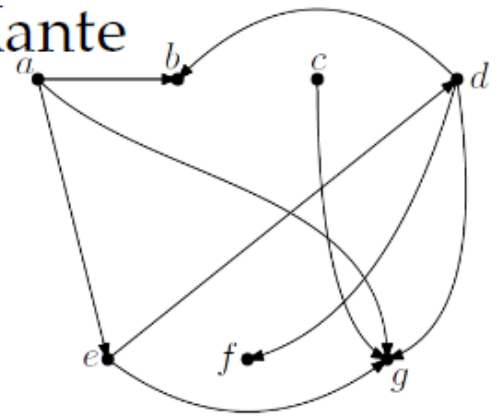
Beispiel (1/3)

Aufgabe. Nutzen Sie den Algorithmus von Kahn, um festzustellen, ob der folgenden Digraph eine topologische Sortierung zulässt. Wenn ja, so geben Sie diese an.



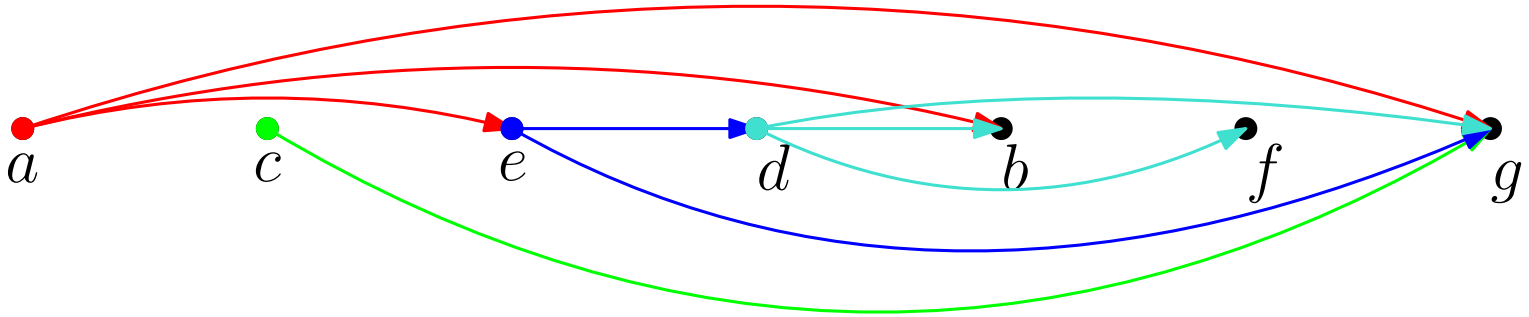
BEISPIEL (2/3)

L = Leere Liste, die die sortierten Elemente beinhalten wird
S = Menge aller Knoten ohne eingehende Kante



while $S \neq \emptyset$ führe aus:
    entferne einen Knoten v aus S
    füge v an das Ende von L hinzu
    für jeden Knoten w mit einer Kante $e = (v, w)$ führe aus:
        entferne Kante e aus dem Graphen
        if w keine anderen eingehenden Kanten hat, **then**
            füge w zu S hinzu
if Graph hat Kanten
    return error (Graph enthält gerichteten Kreis)
else
    return L (eine topologische Sortierung)

Beispiel (3/3)



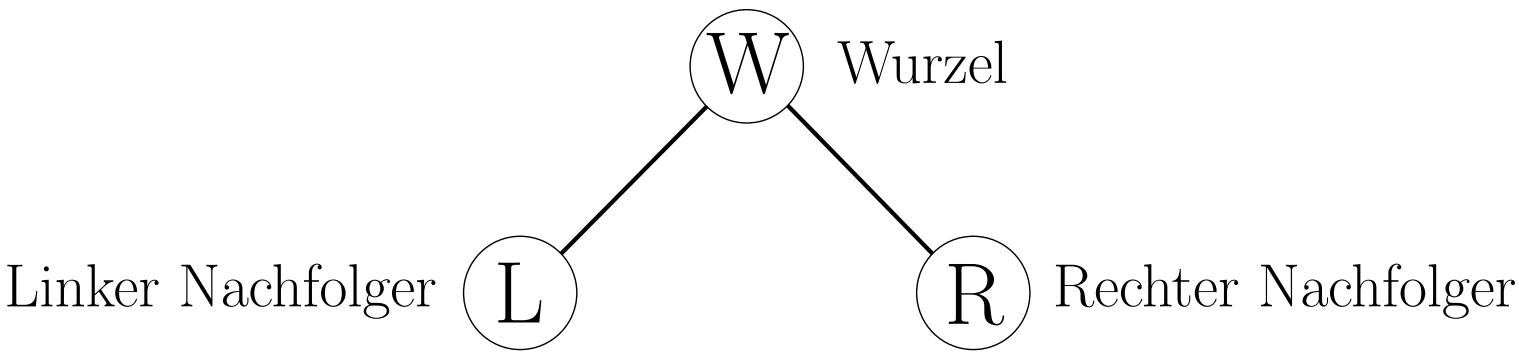
Wir stellen fest, dass a und c Eingangsgrad 0 haben (alle anderen Knoten nicht), diese fügen wir also der Liste S hinzu. Speichere a auf Position 0 und c auf Position 1. Entferne alle Kanten der Form (a, v). Nun hat e keine eingehenden Kanten mehr. Füge also e zu S hinzu (Position 2). Entferne alle Kanten der Form (c, v). Betrachte e und entferne alle Kanten der Form (e, v). Durch das Entfernen von (e, d) hat d keine eingehenden Kanten mehr, füge also d zu S hinzu (Position 3). Löschen wir alle von d ausgehenden Kanten (türkis), so haben danach b und f und g keine eingehenden Kanten mehr – diese fügen wir also zu S hinzu (Positionen 4 und 5 und 6) und erhalten obige topol. Sortierung a, c, e, d, b, f, g.

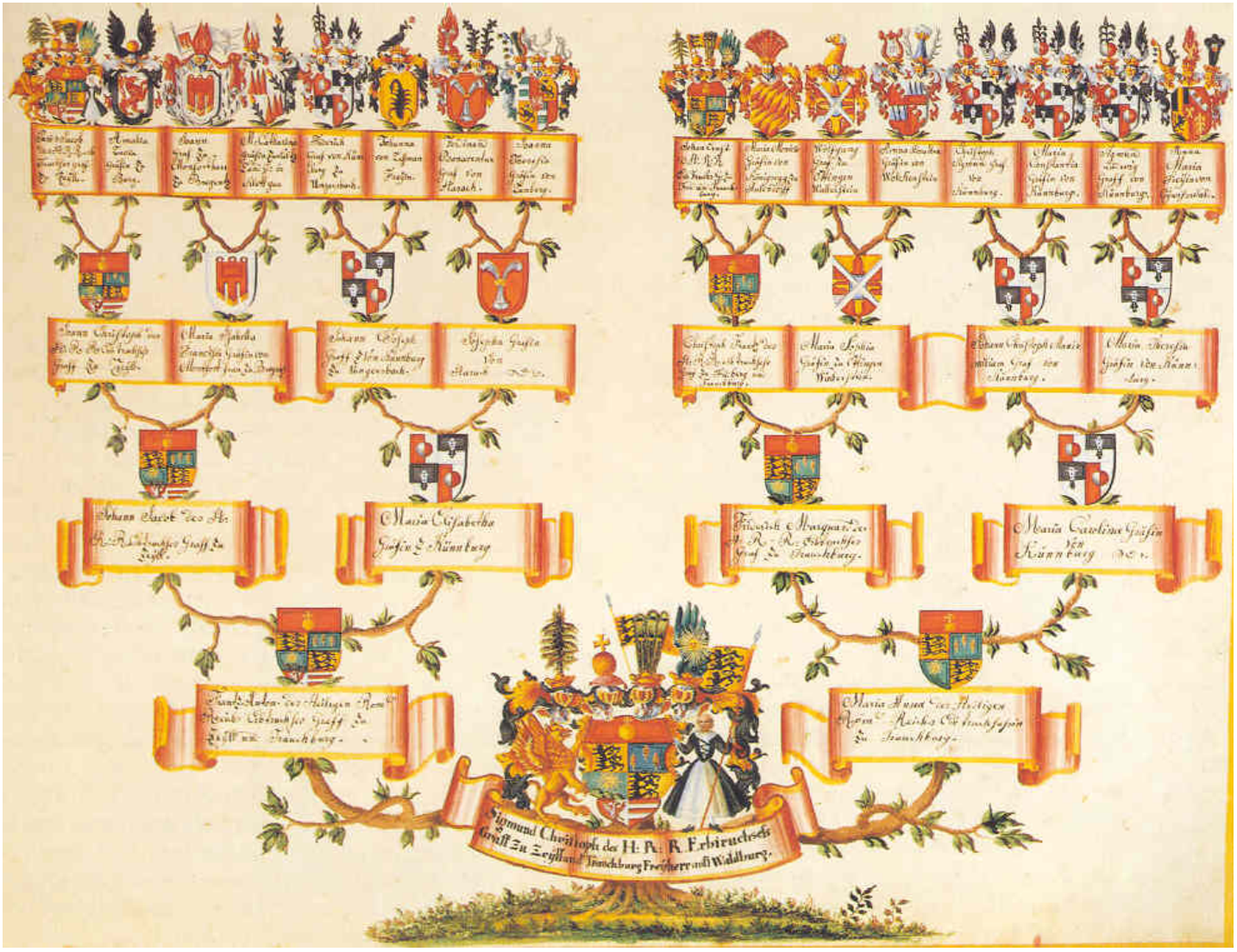
Zusammenfassung

- Tiefensuche: Gehe in die Tiefe solange wie möglich.
- Knotennummerierung auf aufspannendem, kreisfreien Teilgraphen.
- Breitensuche: Prinzip der Warteschlange
- Anwendungen:
  - systematisches Durchsuchen von Graphen, z.B. Labyrinth
  - Ermittlung von Eigenschaften eines Graphen
  - Uninformierte Suche in der KI
- Berechnung von zulässigen Reihenfolgen durch topologische Sortierung der Knoten.

Durchlaufen binärer Bäume

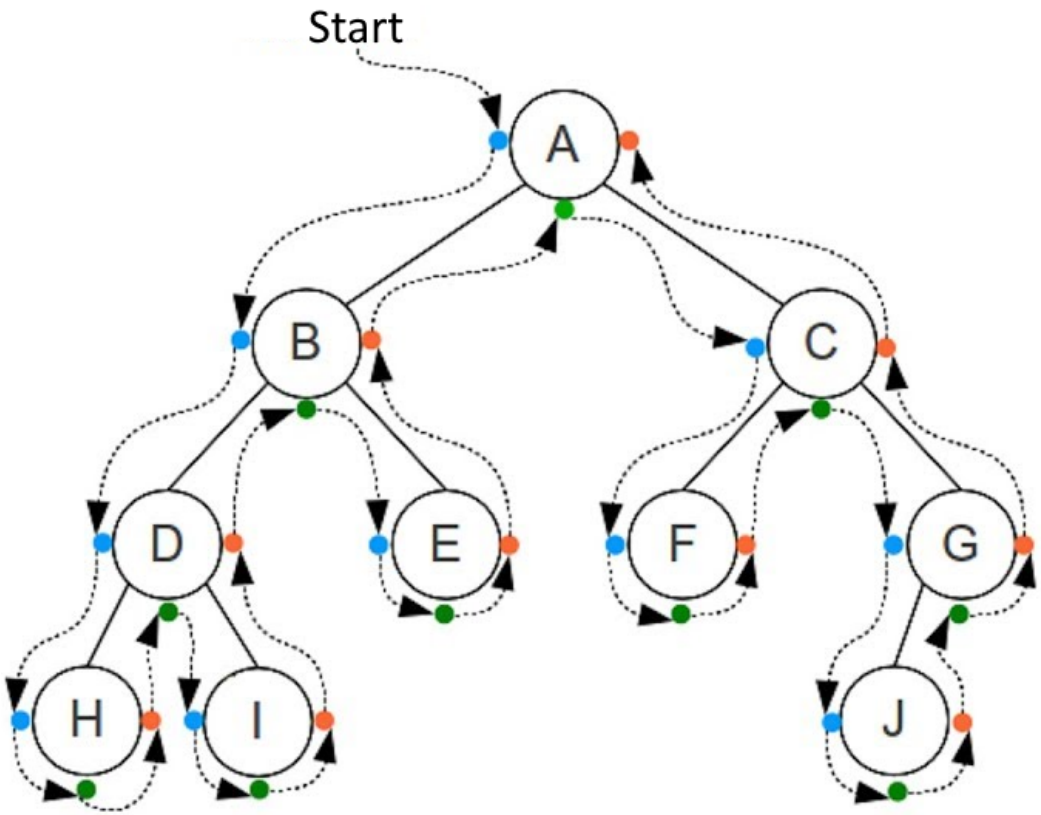
In *binären* Bäumen hat jeder Knoten **höchstens zwei** Nachfolger. Die Knoten müssen häufig in einer gewissen Reihenfolge abgearbeitet (durchlaufen) werden.





Ahnentafel des Sigmund Christoph Graf von Waldburg-Zeil-Trauchburg, Domherr in Konstanz, 1776.

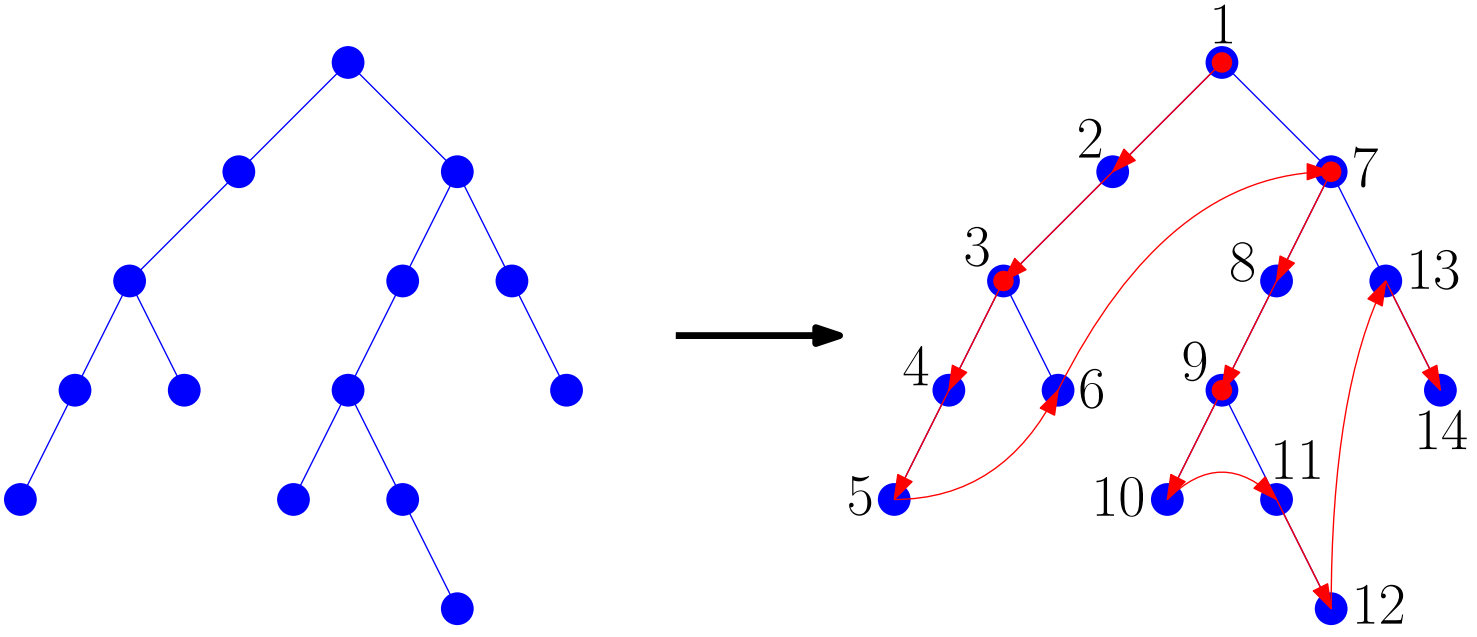
Es gibt im Wesentlichen drei verschiedene Möglichkeiten (sog. Ordnungen), die sich aus der Baumstruktur ergeben, um einen Baum zu durchlaufen:



Pre-Order	ABDHIECFGJ
In-Order	HDIBEAFCJG
Post-Order	HIDEBFJGCA

Durchlaufen binärer Bäume: Pre-order

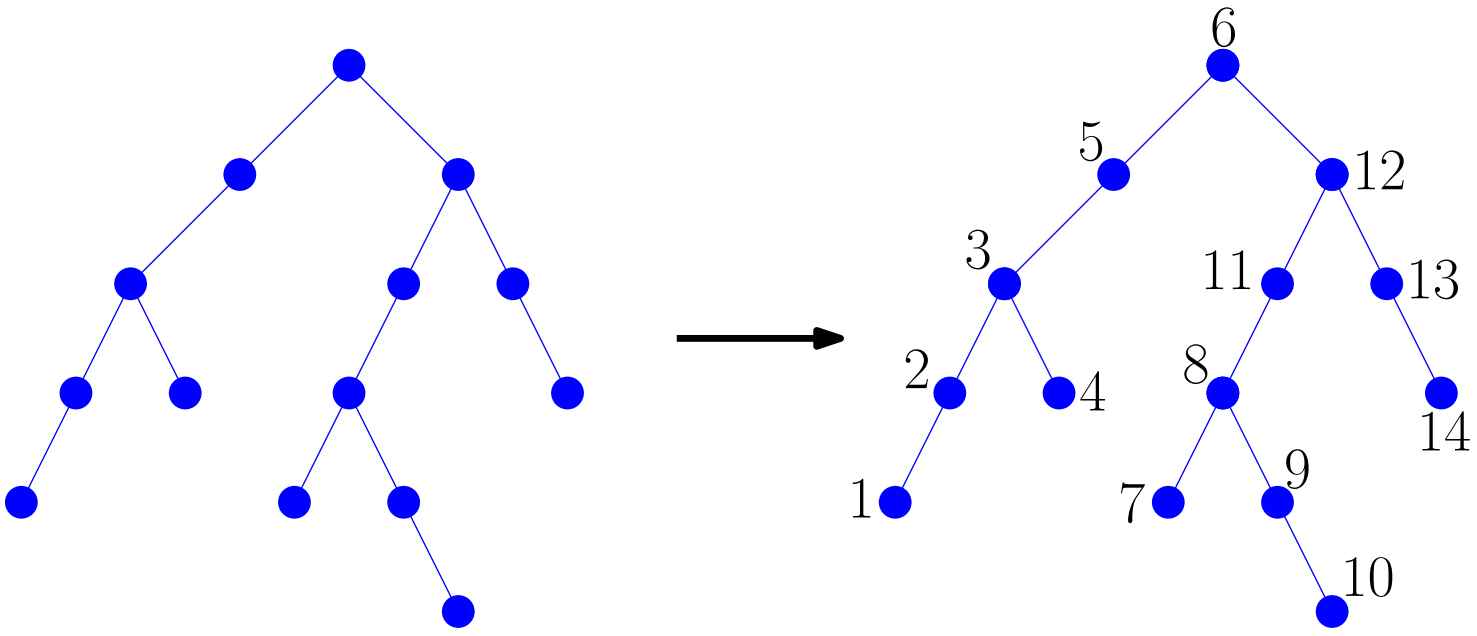
- Pre-order Durchlauf W-L-R:
- Es wird immer zuerst die Wurzel, dann der linke Teilbaum, dann der rechte Teilbaum abgearbeitet.



Diese Reihenfolge wird auch als Prefix-Notation bezeichnet.

Durchlaufen binärer Bäume: In-order

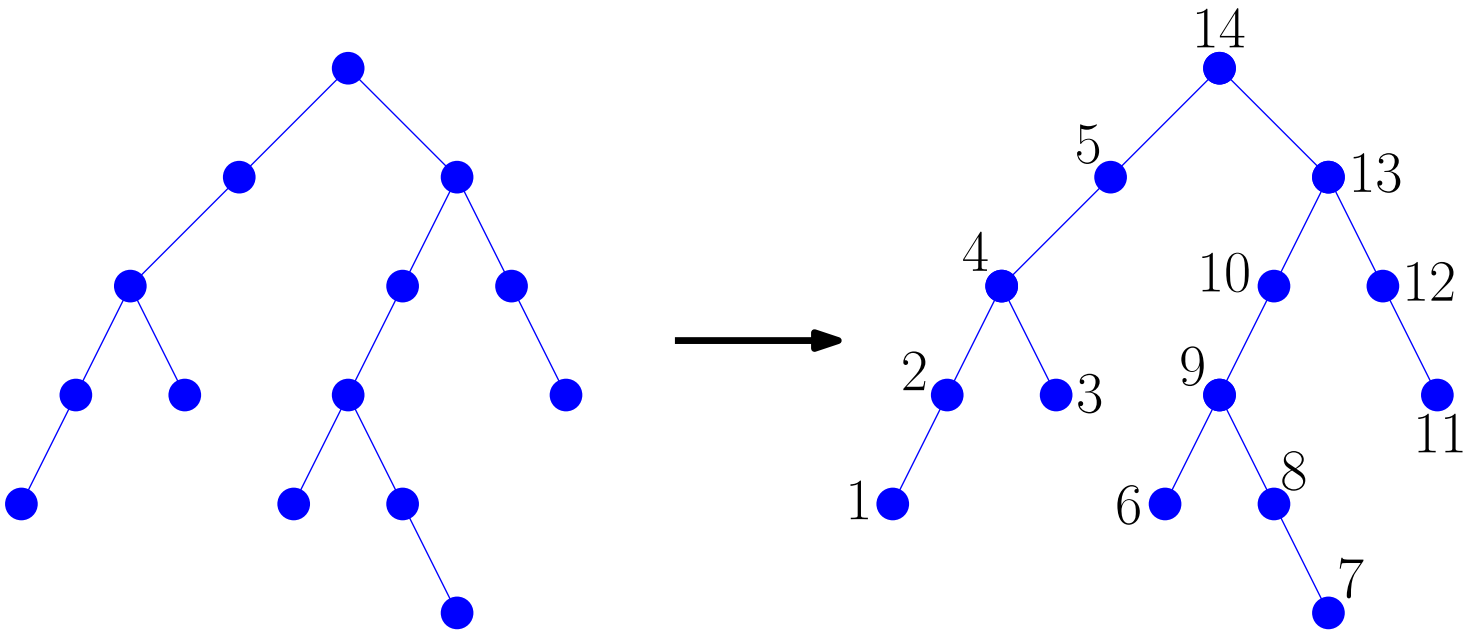
- In-order Durchlauf L-W-R:
- Zunächst der linke Teilbaum, dann die Wurzel, dann der rechte Teilbaum.



Diese Durchlaufreihenfolge wird auch als Infix-Notation bezeichnet.

Durchlaufen binärer Bäume: Post-order

- Post-order Durchlauf L-R-W:
- Bei diesem Verfahren wird immer zuerst der linke Teilbaum, dann der rechte Teilbaum und erst am Schluss die Wurzel abgearbeitet.



Dieses Verfahren wird auch als Postfix-Notation bezeichnet.