

MATHEMATIK IT-1

DISKRETE MATHEMATIK FÜR INFORMATIK-STUDIENGÄNGE

29. November 2021

Prof. G. Averkov

Institut für Mathematik, Fakultät 1

Fachgebiet Algorithmische Mathematik

Brandenburgische Technische Universität Cottbus-Senftenberg

Inhaltsverzeichnis

IV	Algorithmische Graphentheorie	4
1	Grundbegriffe der Graphentheorie	4
2	Speicherung von Graphen	30
3	Breitensuche	43
4	Tiefensuche	95
4.1	Anwendung I – Topologisches Sortieren	150

4.2	Anwendung II – Starke Zusammenhangskomponenten	157
5	Exkurs in Heap-basierte Prioritätsschlangen	177
6	Der Prim-Algorithmus	178
7	Färbung	200

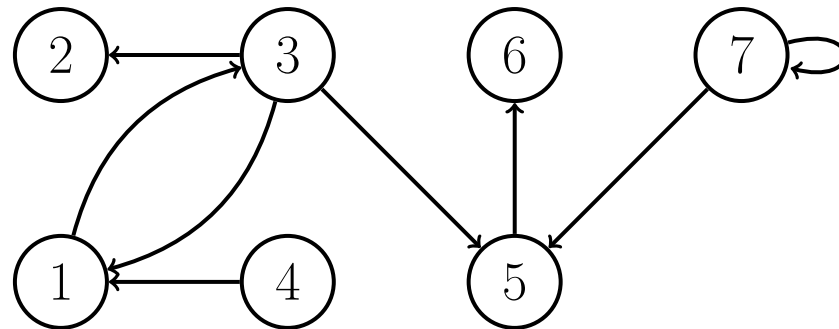
Kapitel IV

Algorithmische Graphentheorie

1 Grundbegriffe der Graphentheorie

1.1 Def. Sei V eine endliche Menge und sei A eine Teilmenge von $V \times V$. Das Paar $D = (V, A)$ heißt *Digraph* (engl. *directed graph*) mit *Knotenmenge* V und *Kantenmenge* A . Die Elemente von V heißen *Knoten* und die Elemente von A heißen (*gerichtete*) *Kanten* (engl. *arcs*) von D . Ist $(a, b) \in A$ eine *Kante*, so nennen wir a ihren *Startknoten* und b ihren *Endknoten*. Man sagt auch, dass die Knoten a und b *benachbart* und die Kante (a, b) *inzident* zu a und b ist. Eine Kante der Form (a, a) heißt *Schlinge*.

1.2 Bsp. Digraphen lassen sich auf vielfache Weise anschaulich darstellen. Eine verbreitete und nützliche Möglichkeit ist das „Zeichnen“ in die Ebene. Ein Digraph mit Doppelkante zwischen den Knoten 1 und 3 und Schlinge um den Knoten 7 sei zum Beispiel wie folgt dargestellt:



1.3 Def. Für einen gegebenen Knoten $a \in V$ heißt die Anzahl der Knoten $b \in V$ mit $(a, b) \in A$ der *Ausgangsgrad* von a . Analog heißt für $b \in V$ die Anzahl der Knoten $a \in V$ mit $(a, b) \in A$ der *Eingangsgrad* von b .

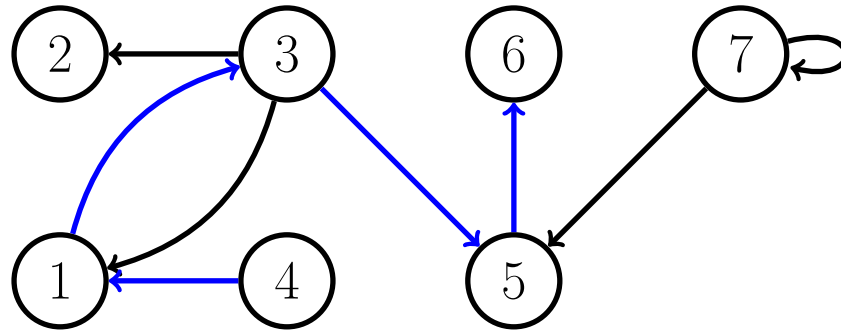
1.4 Def. Sind $D = (V, A)$ und $D' = (V', A')$ zwei Digraphen, so dass $V' \subseteq V$ und $A' \subseteq A$ gelten, so nennt man D' einen *Teilgraphen* von D . Ist $W \subseteq V$ so heit

$$D_W := (W, \{(a, b) : a, b \in W \text{ und } (a, b) \in A\})$$

der durch W *induzierte* Teilgraph von D .

1.5 Def. Für einen Digraphen $D = (V, A)$ und Knoten $a, b \in V$ heißt ein Tupel (v_0, \dots, v_k) mit $(v_i, v_{i+1}) \in A$, für $0 \leq i \leq k-1$ und $v_0 = a$ und $v_k = b$ ein (a, b) -*Pfad* der Länge k . Ein Pfad heißt *Weg*, wenn die Knoten des Pfades paarweise verschieden sind. Der Pfad (v_i, \dots, v_j) mit $0 \leq i \leq j \leq k$ heißt *Teilpfad* von (v_0, \dots, v_k) . Ein (a, b) -Pfad mit $a = b$ heißt *Zyklus*. Ein Zyklus (v_0, \dots, v_k) heißt *Kreis*, falls die Knoten v_1, \dots, v_{k-1} paarweise verschieden sind. Die Zyklen (v_0, \dots, v_k) und $(v_i, \dots, v_k, v_0, \dots, v_i)$ werden als gleich angesehen. Man sagt eine Kante (u, v) *gehört* zu einem Pfad (v_0, \dots, v_k) , falls $u = v_i$ und $v = v_{i+1}$ für ein i mit $0 \leq i \leq k-1$ gilt. Digraphen ohne Zyklen heißen *azyklisch*.

1.6 Bsp. Im Beispieldigraphen von oben markieren wir den Weg $(4, 1, 3, 5, 6)$ in blau:



1.7. Ein wichtiger Aspekt von Digraphen ist die Orientierung der Kanten, die es ermöglicht zwischen Start- und Endknoten zu unterscheiden. Wird eine solche Orientierung nicht benötigt, so vereinfachen sich manche Begriffe und wir sprechen anstelle von Digraphen dann von Graphen.

1.8 Def. Sei dazu V wieder eine endliche Menge und sei E nun eine Teilmenge der Menge $\binom{V}{2} = \{\{a, b\} : a, b \in V, a \neq b\}$.

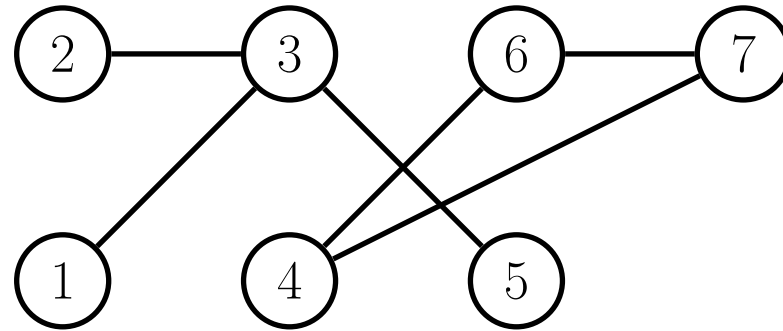
Das Paar $G = (V, E)$ heißt *Graph* mit *Knotenmenge* V und *Kantenmenge* E . Die Elemente von V heißen *Knoten* und die Elemente von E heißen *Kanten* (engl. *edges*) von G . Ist $\{a, b\} \in E$ eine Kante, so nennen wir a und b ihre *Endknoten*. Wiederum sagen wir, dass die Knoten a und b *benachbart* und die Kante $\{a, b\}$ *inzident* zu a und b ist.

1.9. Im Gegensatz zu Digraphen erlaubt die Definition eines Graphen keine Schlingen oder parallele Kanten. In manchen Literaturquellen lässt man mitunter Mehrfachkanten oder „ungerichtete Schlingen“ zu und spricht dann von einem *einfachen Graphen* wenn man beides verbietet.

1.10 Def. Für einen Knoten $a \in V$ heißt die Anzahl der Knoten $b \in V$ mit $\{a, b\} \in E$ der *Grad* des Knotens a . Die Begriffe (*induzierter*) *Teilgraph*, *Pfad*, *Weg*, *Teilpfad*, *Zyklus* und *Kreis* werden für Graphen analog zu denen für Digraphen definiert. Ein Graph ohne Zyklen heißt *Wald*.

1.11 Def. Ein Graph $G = (V, E)$ heißt *zusammenhängend*, falls für alle Knoten $a, b \in V$ ein (a, b) -Pfad existiert. Ein zusammenhängender Wald heißt *Baum*. Die Relation „es existiert ein (a, b) -Pfad“ ist eine Äquivalenzrelation auf den Knoten des Graphen G . Die Äquivalenzklassen bezüglich dieser Relation heißen *Zusammenhangskomponenten* des Graphen. Ein Wald ist ein Graph bei dem jede Zusammenhangskomponente ein Baum ist.

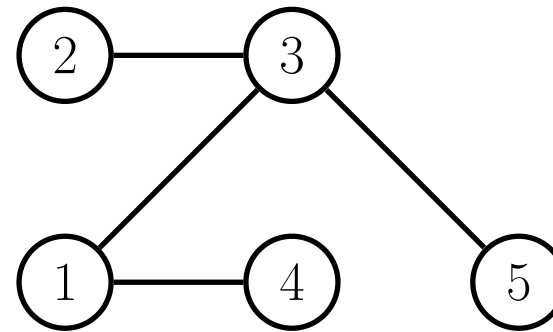
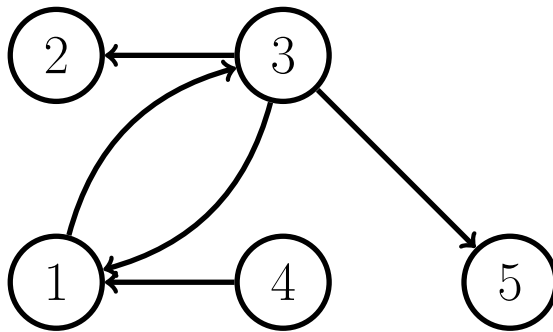
1.12 Bsp. Die folgende Abbildung zeigt einen Graphen mit zwei Zusammenhangskomponenten:



1.13. Wir benutzen gelegentlich die Kurzbezeichnung $ab := (a, b)$ für Kanten von Digraphen und $ab := \{a, b\}$ für Kanten von Graphen. Beachten Sie, dass es im Fall einer gerichteten Kante auf die Reihenfolge der Knoten a und b ankommt.

1.14 Def. Ein schleifenfreier Digraph $D = (V, A)$ kann auch als Orientierung eines ungerichteten Graphen $G = (V, E)$ mit $E = \{\{u, v\} : (u, v) \in A\}$ verstanden werden. Der Graph G wird dann als der *zugrundeliegende Graph* von D bezeichnet. Sobald E nicht leer ist, gibt es stets mindestens zwei Digraphen, denen G zugrundeliegt.

1.15 Bsp. Die folgende Abbildung zeigt einen (schleifenfreien) Digraphen und den dazu-gehörigen zugrundeliegenden Graphen:



Beachten Sie, dass die beiden in entgegengesetzten Richtungen orientierten Kanten zwischen den Knoten 1 und 3 zu *einer* Kante im ungerichteten Graphen verschmelzen.

1.16 Def. Halten wir noch ein paar wichtige spezielle Graphenklassen und deren Bezeichnungen fest: Sei dazu im Folgenden $G = (V, E)$ ein Graph mit n Knoten und m Kanten.

- Ist $E = \binom{V}{2}$, so heißt G **vollständig**. Ein vollständiger Graph mit n Knoten wird oft mit K_n bezeichnet.
- Ist $E = \emptyset$, so heißt G **leer**. Ein leerer Graph mit n Knoten wird oft mit \bar{K}_n bezeichnet.
- Ist V eine disjunkte Vereinigung zweier Mengen A und B und hat jede Kante von G die Form $\{a, b\}$ mit $a \in A$ und $b \in B$, so heißt G **bipartit**. Die Mengen A und B heißen dann **Partitionsklassen** von G .
- Ist G bipartit und gilt darüber hinaus, dass jedes $\{a, b\}$ mit $a \in A$ und $b \in B$ eine Kante von G ist, so heißt G **vollständig bipartit**. Ein vollständig bipartiter Graph,

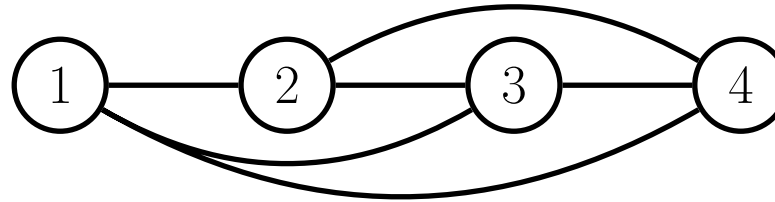
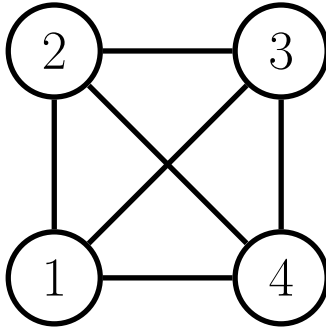
deren Partitionsklassen die Größe r und s haben, wird oft mit $K_{r,s}$ bezeichnet. Natürlich gilt $n = r + s$.

- Eine *Einbettung* von G in die Ebene \mathbb{R}^2 ist eine Menge $P = \{p_1, \dots, p_n\} \subseteq \mathbb{R}^2$ von Punkten zusammen mit einer Menge $S = \{S_1, \dots, S_m\}$ von Streckensegmenten, so dass es eine Bijektion $f : V \rightarrow P$ gibt mit der Eigenschaft, dass $\{u, v\}$ genau dann eine Kante von G ist, wenn die Punkte $f(u)$ und $f(v)$ Eckpunkte desselben Streckensegmentes S_i sind. Besitzt ein Graph eine Einbettung bei der je zwei verschiedene Streckensegmente sich höchstens in Eckpunkten schneiden, so heißt dieser Graph **planar**.

1.17 Bsp.

- K_n mit $n \leq 4$ sind planar.
- $K_{n,m}$ mit $n \leq 2$ und $m \leq 3$ sind auch planar.
- $K_{3,3}$ ist nicht planar. (Beweis durch Widerspruch)
- K_5 ist nicht planar. (Beweis durch Widerspruch)

1.18 Bsp. Die nachfolgende Illustration zeigt zwei Darstellungen des K_4 :



Keine der beiden Darstellungen zeigt, dass K_4 planar ist. Warum nicht? Finden Sie eine Darstellung, die dies illustriert!

1.19 Def. Ein Digraph $D = (V, A)$ heißt **gerichteter bzgl. gewurzelter Baum** (engl., out-tree oder arborescence) mit Wurzel $w \in V$, wenn der unterrichtete Graph zu D ein Baum ist, der Eingangsgrad von w gleich 0 ist und der Eingangsgrad aller anderen Knoten $v \in V \setminus \{w\}$ gleich 1 ist.

1.20 Prop. Zu jedem ungerichteten Baum $G = (V, E)$ und jedem $w \in V$ gibt es genau einen gerichteten Baum zu G mit der Wurzel w .

Beweis. Aufgabe.



1.21 Def. Ein Digraph $D = (V, A)$ heißt **gerichteter Wald**, wenn V disjunkte Vereinigung von Mengen V_1, \dots, V_t ist, bei denen D_{V_i} für jedes $i = 1, \dots, t$ ein gerichteter Baum ist.

1.22 Def. In einem gerichteten Baum/Wald $D = (V, A)$ nennen wir $v \in V$ **Nachfahre** von $u \in V$ und u **Vorfahre** von v , wenn ein (u, v) -Pfad existiert. Wenn $(u, v) \in A$ ist, so nennen wir v **Kind** oder **direkter Nachfahre** von u und u **Vater/Mutter/Elter** von v .

1.23. In der Graphentheorie ist die Bezeichnung der Hauptbegriffe nicht einheitlich. Von Quelle zu Quelle unterscheiden sich die Definitionen geringfügig und insbesondere in Bezug auf die Namensgebung gibt es teilweise große Unterschiede. Zu diesem Zweck listen wir hier die geläufigsten Synonyme um mögliche Verwirrungen beim Lesen unterschiedlicher Quellen zu ver-

meiden:

gerichteter Graph = Digraph

Knoten = Ecke

gerichtete Kante = Bogen

adjazent = benachbart

Grad = Valenz

Weg = einfacher Pfad

Kreis = einfacher Zyklus

Kantenzug = Pfad

geschlossener Kantenzug = Zyklus

Schleife = Schlinge

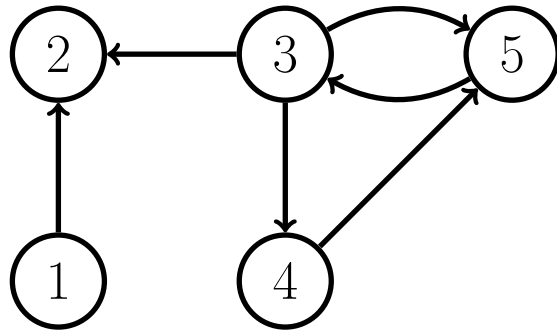
2 Speicherung von Graphen

2.1. Zum Abschluss dieser Einführung in die Grundbegriffe schauen wir uns noch verschiedene Möglichkeiten an, wie man einen gegebenen (Di)Graphen im Rechner darstellt, das heißt, in welcher Form man die Beziehungen zwischen Knoten und (gerichteten) Kanten verwalten kann.

Sei dazu ein Graph oder ein Digraph gegeben und zur Vereinheitlichung im Folgenden mit $D = (V, A)$ bezeichnet.

2.2 Def. Die **Kantenliste** von $D = (V, A)$ ist die Liste der Kanten von D .

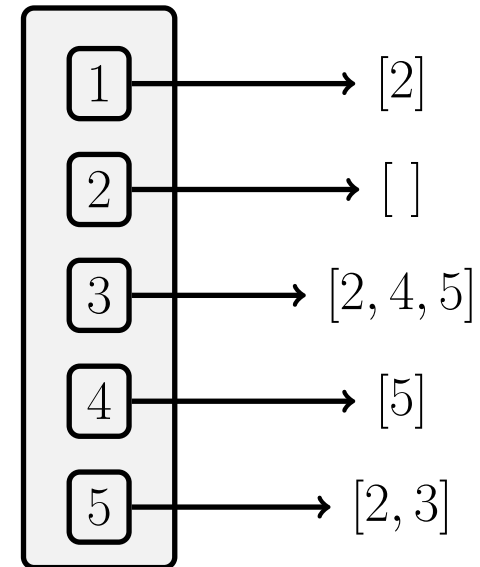
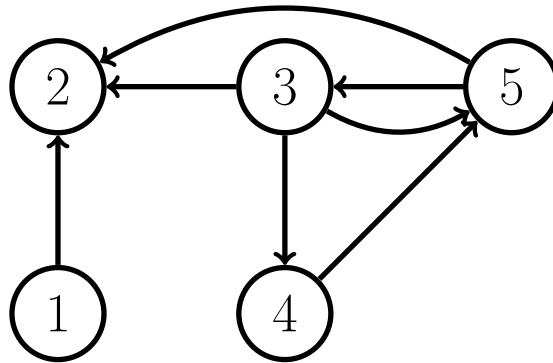
2.3 Bsp.



$[(1, 2), (3, 2), (3, 4), (3, 5), (4, 5), (5, 3)]$

2.4 Def. In der **Adjazenzliste** von G speichert man die Knotenmenge V als Liste und jedem Element $u \in V$ der Liste wird eine Liste aller Knoten $v \in V$ mit $uv \in E$ zugeordnet. (Der Speicherplatzbedarf für diese Darstellung ist offensichtlich $\Theta(|V| + |E|)$.)

2.5 Bsp.



2.6. Typische Fragen auf Graphen, deren algorithmische Behandlung grundlegend für viele Anwendungen ist, sind zum Beispiel die folgenden:

- Ist der Graph zusammenhängend?
- Gibt es einen Pfad im Graphen von einem Knoten s zu einem Knoten z ?
- Wieviele Zusammenhangskomponenten hat der Graph?
- Gibt es Kreise oder Zyklen im Graphen?
- Was ist der Grad eines gegebenen Knotens?

Ein allgemeines Prinzip um diese Art Fragen zu beantworten ist die sogenannte *Graphen-durchmusterung*, deren Ziel es ist alle Knoten und Kanten des gegebenen Graphen zu „erkunden“ und dabei nützliche Informationen über den Graphen zu sammeln. In Graphendurchmusterungen wird der (Di)Graph in der Regel durch seine Adjazenzliste gegeben.

2.7 Def. Matrizen mit $m \in \mathbb{N}$ Zeilen und $n \in \mathbb{N}$ Spalten und mit Komponenten in einer Menge K entsprechen den Abbildungen $A : \{1, \dots, m\} \times \{1, \dots, n\} \rightarrow K$. Das bedeutet, dass für die Angabe einer Matrix A für alle $i \in \{1, \dots, m\}$ und $j \in \{1, \dots, n\}$ die Komponente $A(i, j)$ der Matrix in der Zeile i und Spalte j festgelegt werden muss.

An der Stelle von $A(i, j)$ benutzt man traditionell eine Bezeichnung mit einem kleinen Buchstaben und untergestellten Indizes i und j .

So schreibt man zum Beispiel

$$A = (a_{i,j})_{\substack{i=1,\dots,m \\ j=1,\dots,n}} \in K^{m \times n}$$

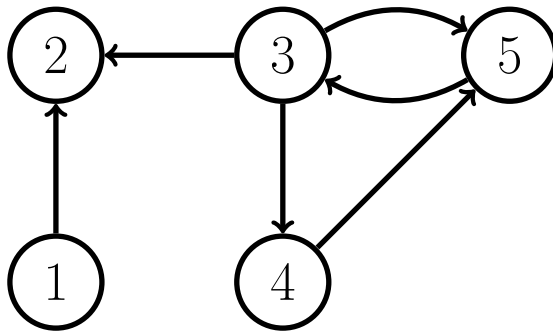
wenn man eine Matrix A der Größe (m, n) führen möchte, deren Komponente in der Position (i, j) als $a_{i,j}$ bezeichnet wird.

2.8 Def. Sei $|V| = n$ und $V = \{v_1, \dots, v_n\}$. Die **Adjazenzmatrix** von G ist die Matrix $A = (a_{i,j}) \in \mathbb{R}^{n \times n}$ mit Einträgen

$$a_{i,j} = \begin{cases} 1 & , \text{ falls } \{v_i, v_j\} \in E \text{ bzw. falls } (v_i, v_j) \in E, \\ 0 & , \text{ sonst.} \end{cases}$$

Der Speicherplatzbedarf (bei direkter Implementierung von Matrizen) ist $\Theta(|V|^2)$.

2.9 Bsp.



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

2.10 Def. Sei $G = (V, E)$ Digraph mit $m = |E|$ und $n = |V|$ und sei $V = \{v_1, \dots, v_n\}$ und $E = \{e_1, \dots, e_m\}$. Sei weiterhin G zunächst als gerichteter Graph ohne Schleifen angenommen. Die **Inzidenzmatrix** von G ist die Matrix $B = (b_{i,j}) \in \mathbb{R}^{n \times m}$ mit Einträgen

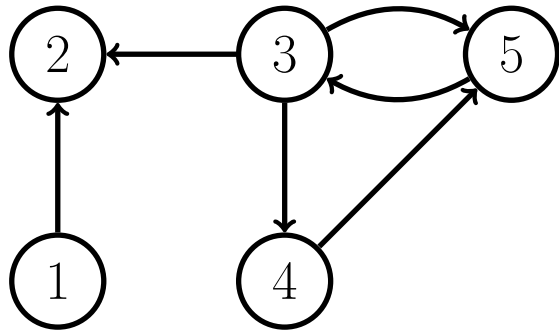
$$b_{i,j} = \begin{cases} 1 & , \text{ falls } v_i \text{ Startknoten von } e_j \text{ ist,} \\ -1 & , \text{ falls } v_i \text{ Endknoten von } e_j \text{ ist,} \\ 0 & , \text{ sonst.} \end{cases}$$

Ist G ein ungerichteter Graph, so lassen wir lediglich die Vorzeichen weg, das heißt, die

Einträge der Inzidenzmatrix sind

$$b_{i,j} = \begin{cases} 1 & , \text{ falls } v_i \text{ Endknoten von } e_j \text{ ist,} \\ 0 & , \text{ sonst.} \end{cases}$$

Der Speicherplatzbedarf (bei direkter Implementierung von Matrizen) ist $\Theta(|V| \cdot |E|)$.

2.11 Bsp.

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & -1 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 1 \end{pmatrix}$$

Die Kanten in der vorigen Abbildung sind in der Reihenfolge 12, 32, 34, 35, 45, 53 nummeriert.

3 Breitensuche

3.1. Die Eingabe für die Breitensuche ist ein Digraph $D = (V, A)$ in Form einer Adjazenzliste. Auf ungerichteten Graphen ist die Vorgehensweise analog und wir diskutieren (fast ausschließlich) die gerichtete Variante.

3.2 Def. Für zwei Knoten $u, v \in V$ in D definieren wir den *Abstand* von u nach v als die minimale Länge eines (u, v) -Pfades und bezeichnen ihn mit $\delta(u, v)$. Wenn kein solcher Pfad existiert, setzen wir $\delta(u, v) := \infty$. Für ungerichtete Graphen ist $\delta(u, v)$ entsprechend definiert und es gilt $\delta(u, v) = \delta(v, u)$ für alle Knoten $u, v \in V$. Wie oben erwähnt erlaubt uns die Breitensuche das Bestimmen der Abstände $\delta(s, v)$ von einem festen *Startknoten* $s \in V$ aus zu jedem anderen Knoten v . Ist $\delta(s, v) < \infty$, so sagen wir, dass v von s aus *erreichbar* ist.

3.3. Während der Durchmusterung des Digraphen D wird ein Array d der Länge $|V|$ verwaltet, in dem nach Abschluss der Breitensuche die Abstände $\delta(s, v)$, für alle Knoten $v \in V$, gespeichert sind.

3.4 Def. Für die Umsetzung der Breitensuche wird die folgende Hilfsdatenstruktur benutzt. Eine *Warteschlange* Q ist eine Liste $Q = [q_1, \dots, q_k]$, die mit zwei Grundoperationen ausgestattet ist:

- **DEQUEUE(Q):** Das *erste* Element q_1 von Q wird zurückgegeben und aus der Warteschlange entfernt. Man nennt das Element q_1 den *Kopf* der Warteschlange und nach der Operation gilt $Q = [q_2, \dots, q_k]$.
- **ENQUEUE(Q, x):** Das Element x wird am *Ende* von Q hinzugefügt. Nach der Ausführung dieser Operation gilt also $Q = [q_1, \dots, q_k, x]$.

3.5. Aufgrund der Einfüge- und Rückgabereihenfolge sagt man auch, dass Warteschlangen nach dem FIFO-Prinzip (First In - First Out) arbeiten. Warteschlangen mit höchstens $n \in \mathbb{N}$ Elementen können auf der Basis von Arrays der Länge n umgesetzt werden, sodass die Laufzeit der beiden Grundoperationen $\Theta(1)$ Zeiteinheiten beträgt.

3.6. Bevor wir mit der eigentlichen Breitensuche beginnen können, müssen eine Reihe von Attributen und natürlich auch die Warteschlange korrekt initialisiert werden: Zu Beginn setzen wir $d[v] := \infty$, für alle $v \in V \setminus \{s\}$, und $d[s] := 0$. Besteht die Gleichheit $d[v] = \infty$ zu einem Moment der Breitensuche, so bedeutet dies, dass der Knoten v bisher noch nicht entdeckt wurde. Die erste Veränderung des Wertes $d[v]$ entspricht der Entdeckung von v , und wir nennen $d[v]$ die *aktuelle Distanz* von s zu v . Wir verwalten weiterhin wie bei der Tiefensuche das Farbattribut $\text{FARBE}[u]$ eines Knotens $u \in V$ um den Bearbeitungszustand von u zu jedem Zeitpunkt der Breitensuche zu kennen. Auch die Vorgängerabbildung π kommt wieder zum Einsatz.

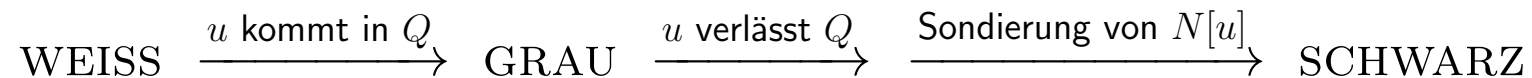
3.7 Def. Die **Vorgängerabbildung** ist ein Array π mit $|V|$ Komponenten. Durch $\pi[v]$ wird notiert von welchem Knoten aus der Knoten v entdeckt wurde.

3.8. Wir fassen die Initialisierungen zu einer eigenen Hilfsprozedur zusammen:

BREITENSUCHE-INITIALISIEREN(D, s)

- 1: **for** $u \in V \setminus \{s\}$:
 - 2: FARBE[u] := WEISS
 - 3: $d[u] := \infty$ ▷ *Aktueller Abstand zu s*
 - 4: $\pi[u] := \text{NIL}$ ▷ *Vorgänger von u*
 - 5: **end**
 - 6: FARBE[s] := GRAU
 - 7: $d[s] := 0$
 - 8: $\pi[s] := \text{NIL}$
 - 9: Deklariere eine leere Warteschlange Q
 - 10: ENQUEUE(Q, s)
-

3.9. Die Breitensuche ist so umgesetzt, dass das “Leben” eines jeden Knotens u als der folgender Ablauf zusammengefasst werden kann:



Beim Sondieren von $N[u]$ werden weiße Knoten entdeckt: sie kommen in Q und werden grau gefärbt. Die Breitensuche ist durch die Wahl des Containers Q als Warteschlange ausgezeichnet. Durch diese Wahl “simuliert” die Breitensuche die Ausbreitung einer Welle in einem Medium. Stellen Sie sich vor: in s kommt es zu einer Explosion; die Schallwelle breitet sich in G entlang der Kanten aus und erreicht mit der Zeit immer mehr Knoten. Wie sich die Front der Schallwelle durch die Knoten ausbreitet wird durch die Warteschlange Q “modelliert”.

BREITENSUCHE(s)

```
1: BREITENSUCHE-INITIALISIEREN( $s$ )
2: while  $Q$  enthält Knoten :
3:    $u := \text{DEQUEUE}(Q)$      $\triangleright$  Die Bearbeitung des Knotens  $u$  beginnt.
4:    $\triangleright$  Aus  $u$  ausgehende Kanten werden sondiert
5:   for  $v \in N[u]$  :
6:     if  $\text{FARBE}[v] = \text{WEISS}$  :
7:        $\text{FARBE}[v] := \text{GRAU}$ 
8:        $\text{ENQUEUE}(Q, v)$ 
9:        $\pi[v] := u$ 
10:       $d[v] := d[u] + 1$ 
11:    end
12:  end
13:   $\text{FARBE}[u] := \text{SCHWARZ}$ 
14: end
```

3.10. Hier eine Umsetzung der BFS in Sage/Python:

```
class BFS_Queue:
    def __init__(self, maxlength):
        self.contents=[None for i in range(maxlength)]
        self.head=0
        self.tail=0

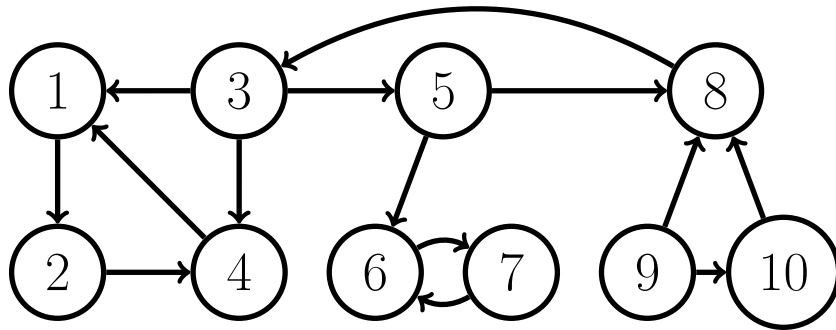
    def length(self):
        return self.tail-self.head

    def enqueue(self, x):
        self.contents[self.tail]=x
        self.tail+=1

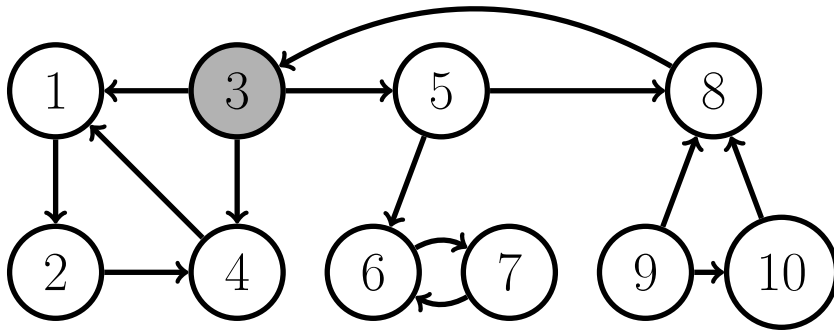
    def dequeue(self):
        self.head+=1
        return self.contents[self.head-1]
```

```
def bfs(G,s):  
    color={u: 'white' for u in G.vertices()}  
    pred={u: None for u in G.vertices()}  
    Q=BFS_Queue(maxlength=len(G.vertices()))  
    Q.enqueue(s)  
    color[s]='grey'  
    d={s:0}  
    while Q.length()>0:  
        u=Q.dequeue()  
        for v in G.neighbors(u):  
            if color[v]=='white':  
                color[v]='grey'  
                d[v]=d[u]+1  
                pred[v]=u  
                Q.enqueue(v)  
        color[u]='black'  
    return d,pred
```

3.11 Bsp. Zum Vergleich mit der Tiefensuche sehen wir uns die Arbeitsweise der Breitensuche wieder auf dem Digraphen aus Beispiel 4.8 an:

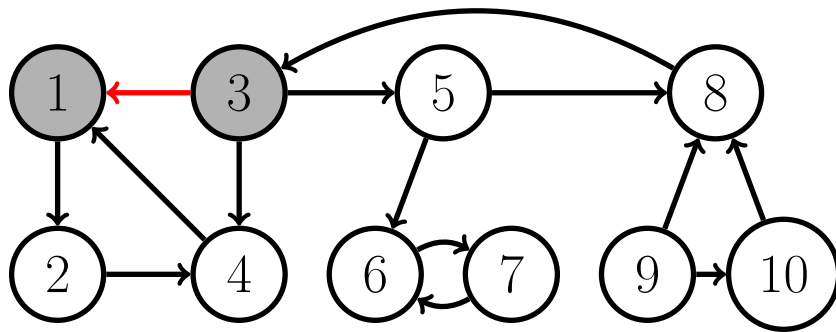


Als Startknoten wählen wir wieder $s = 3$ und treffen ansonsten jede Wahl aufsteigend in der Reihenfolge der Knotenindizes. Wir zeigen die ersten 8 Schritte und benutzen dieselbe Farbkodierung wie in Beispiel 4.8:

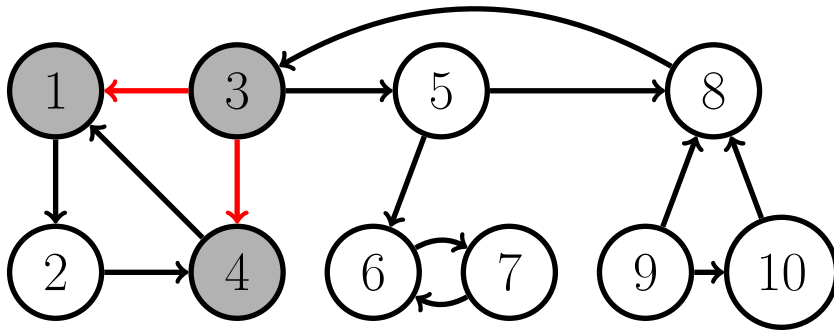
 $Q : \beta$

3. BREITENSUCHE

59

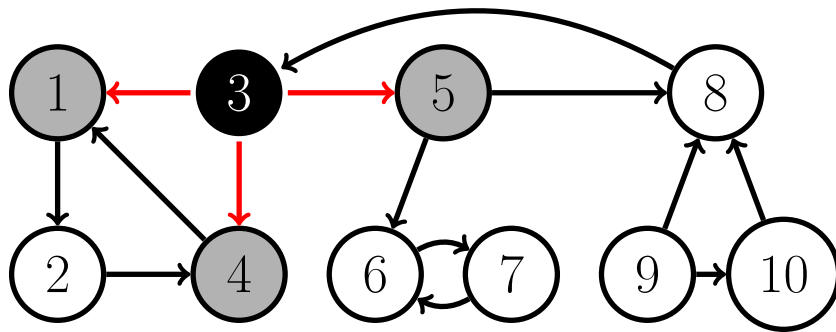


$Q : \emptyset$

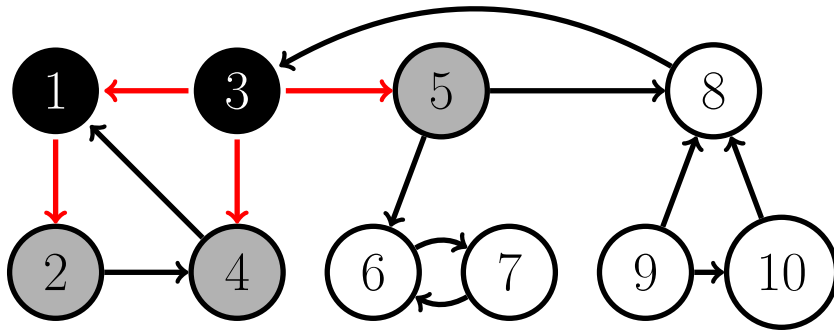
 $Q : \beta_{14}$

3. BREITENSUCHE

61



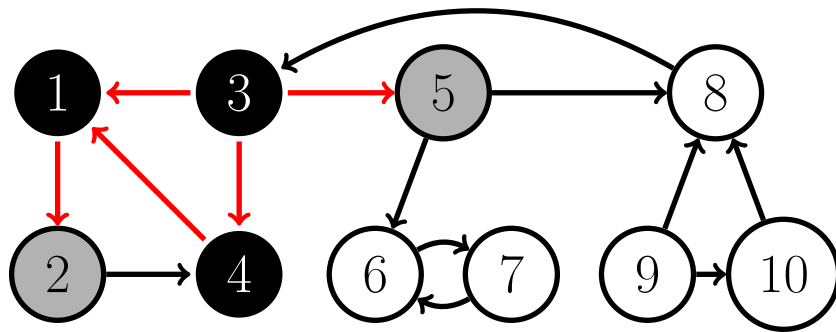
$Q : \beta 145$



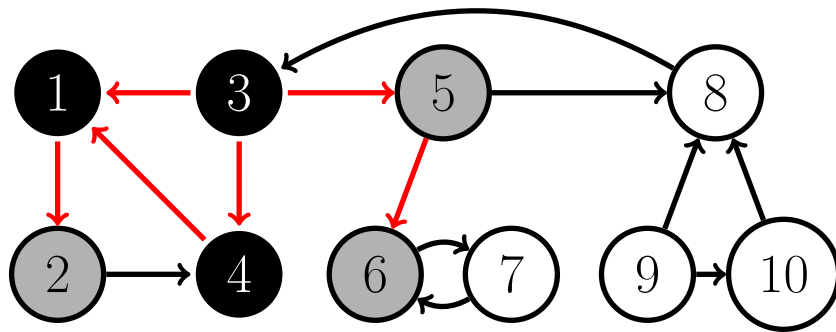
$Q : \beta \not\vdash 1452$

3. BREITENSUCHE

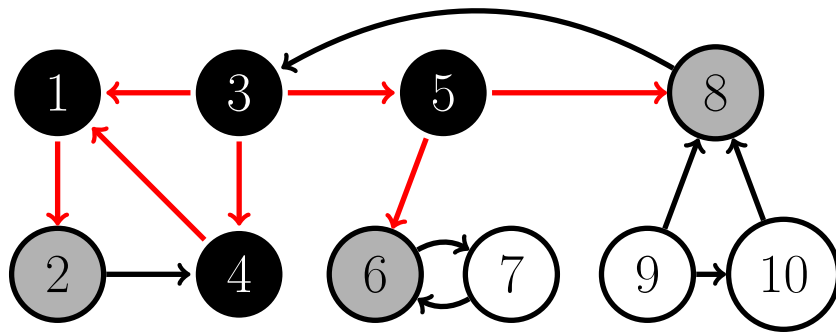
63



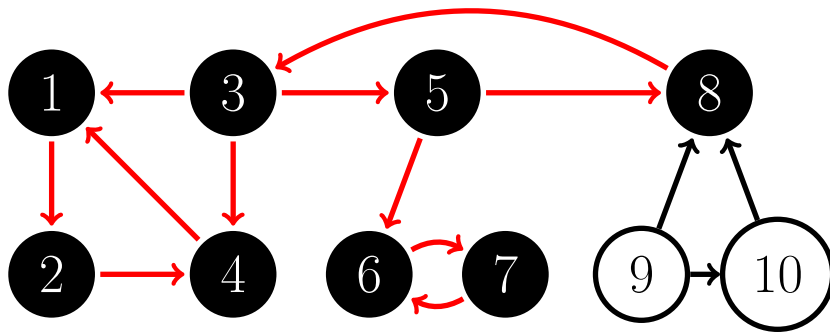
$Q : \beta \neq A52$



$Q : \beta \ 1 \ 4 \ \beta 26$


 $Q : \emptyset \neq A \neq \emptyset$

Da die Knoten 9 und 10 nicht vom Startknoten $s = 3$ aus erreichbar sind, ist der Endzustand nach dem Abschluss von $\text{BREITENSUCHE}(3)$ der folgende:



$Q : \beta \ 1 \ A \ \beta \ 2 \ \beta \ \beta$

Weitere Beispiele können mit `bfs_demo.sage` generiert werden.

3.12. Der Beweis der Korrektheit der Breitensuche inklusive der korrekten Bestimmung der Abstände $\delta(s, v)$, für $v \in V$, erfordert noch etwas Vorbereitung. Die Laufzeitanalyse können wir allerdings bereits durchführen, mit dem Ergebnis, dass auch die Breitensuche einen optimalen linearen Aufwand erfordert:

3.13 Thm. *Es sei ein Digraph $D = (V, A)$ durch eine Adjazenzliste gegeben und es sei $s \in V$ ein beliebiger Startknoten. Die Laufzeit von $\text{BREITENSUCHE}(s)$ ist $\Theta(|V| + |A|)$.*

Beweis. Der Aufwand für die Initialisierung $\text{BREITENSUCHE-INITIALISIEREN}(D, s)$ ist direkt als $\Theta(|V|)$ festzustellen. Der Aufwand von $\text{BREITENSUCHE}(s)$ kann aus dem Diagramm zur Bearbeitung von Knoten $u \in V$ abgelesen werden:

$$\text{WEISS} \xrightarrow{u \text{ kommt in } Q} \text{GRAU} \xrightarrow{u \text{ verlässt } Q} \xrightarrow{\text{Sondierung von } N[u]} \text{SCHWARZ}$$

Der Aufwand pro Knoten, der entdeckt wird ist konstant: der Aufwand besteht aus der Aufnahme in Q , der Färbung von Weiß zu Grau und von Grau zu Schwarz. Der Aufwand pro Kante (u, v) , die sondiert wird, ist ebenfalls konstant: jede Kante (u, v) wird höchstens ein mal sondiert, denn die Sondierung erfolgt nach der Entfernung von u aus der Warteschlange. Nach der Entfernung von u wird u schwarz gefärbt und daher nicht mehr in Q aufgenommen.

Zusammenfassend erhalten wir also wie behauptet einen Zeitaufwand von $\Theta(|V|) + \Theta(|V|) + \Theta(|A|) = \Theta(|V| + |A|)$. □

3.14 Def. Wir nennen einen (s, v) -Pfad der Länge $\delta(s, v)$ einen *kürzesten Pfad* von s nach v in D .

3.15 Lem. Sei $D = (V, A)$ ein Digraph und sei $s \in V$ ein beliebiger Startknoten. Dann gilt für jede Kante $(u, v) \in A$, dass

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Beweis. Falls u von s aus erreichbar ist, dann ist es auch v . In diesem Fall ist der kürzeste (s, v) -Pfad nicht länger als der kürzeste (s, u) -Pfad, verlängert um die Kante (u, v) , und die Ungleichung gilt. Ist u nicht von s aus erreichbar, so ist $\delta(s, u) = \infty$, und die Ungleichung ist trivialerweise erfüllt. □



3.16 Lem. Sei $D = (V, A)$ ein Digraph und sei $s \in V$ ein beliebiger Knoten. Nach Abschluss der Prozedur $\text{BREITENSUCHE}(s)$ gilt $d[v] \geq \delta(s, v)$, für alle $v \in V$.

Beweis. Wir argumentieren mittels vollständiger Induktion nach der Anzahl der Einfügeoperationen in die Warteschlange Q .

Der Induktionsanfang ist der Zeitpunkt nach der Initialisierung zu dem $d[s] = 0 = \delta(s, s)$ und, für alle $v \in V \setminus \{s\}$, die Ungleichung $d[v] = \infty \geq \delta(s, v)$ gilt.

Für den Induktionsschritt sei v ein weißer Knoten, der bei der Suche vom Knoten u aus entdeckt wird. Nach Induktionsvoraussetzung gilt $d[u] \geq \delta(s, u)$. Aufgrund der Zuweisung in Zeile 10 in $\text{BREITENSUCHE}(s)$ und Lemma 3.15 gilt

$$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v).$$

Der Knoten v wird in die Warteschlange Q eingefügt, und da er zu diesem Zeitpunkt grau gefärbt

wurde, ändert sich der Wert $d[v]$ nicht mehr und die Ungleichung bleibt bis zur Terminierung des Algorithmus gültig. □

3.17. Wir fassen dafür die Warteschlange als Liste oder Array $[v_1, \dots, v_r]$ auf, wobei v_1 der Kopf und v_r das Ende von Q ist.

3.18 Lem. Zu jedem Zeitpunkt der Ausführung von $\text{BREITENSUCHE}(s)$ erfüllt die Warteschlange $Q = [v_1, \dots, v_r]$ die Bedingung

$$d[v_1] \leq d[v_2] \leq \dots \leq d[v_r] \leq d[v_1] + 1.$$

Mit anderen Worten: die Liste

$$Q_d := [d[v_1], \dots, d[v_r]]$$

der Distanzen der Knoten von Q hat die Form $[t, \dots, t]$ oder die Form $[t, \dots, t, t+1, \dots, t+1]$ mit $t \in \mathbb{N}_0$ (die Liste Q_d enthält einen Wert oder zwei aufeinanderfolgende ganzzahlige Werte und sie ist aufsteigend sortiert).

Beweis. Am Anfang der Breitensuche enthält Q nur den Startknoten s und die Aussage gilt trivialerweise.

Wir zeigen, dass wenn die behauptete Bedingung zu einem Zeitpunkt gilt, dass sie auch dann gültig bleibt, wenn eine Warteschlangenoperation ausgeführt wird. (Das heißt, wir argumentieren mittels vollständiger Induktion über die Anzahl der ausgeführten Warteschlangenoperationen.)

Sei zunächst wie angenommen $Q = [v_1, \dots, v_r]$ und führen wir $\text{DEQUEUE}(Q)$ aus. Hat die Liste Q_d die Form $[t, \dots, t]$ so, hat sie nach der Entfernung von v_1 immer noch die Form $[t, \dots, t]$ (und Länge um eins kleiner). Hat die Liste Q_d die Form $[t, \dots, t, t+1, \dots, t+1]$ so, hat sie nach der Entfernung von v_1 die Form $[t, \dots, t, t+1, \dots, t+1]$ mit dem Block t, \dots, t , dessen Länge um eins geringer geworden ist, oder die Form $[t+1, \dots, t+1]$.

Schauen wir uns an, was passiert, wenn in Zeile 8 von $\text{BREITENSUCHE}(s)$ die Operation $\text{ENQUEUE}(Q, v)$ ausgeführt wird. Der Knoten v wird zum Ende der neuen Warteschlange $Q' = [v_1, \dots, v_r, v]$ und zum Zeitpunkt dieser Operation durchsucht der Algorithmus die Nachbarschaft des Knotens u , der zuvor aus der Warteschlange entfernt wurde. Daher gilt nach Annahme die Ungleichung $d[u] \leq d[v_i]$ für alle $1 \leq i \leq r$ und nach Zeile 10 weiterhin $d[v] = d[u] + 1 \leq d[v_1] + 1$. Es gilt außerdem, dass $d[v_r] \leq d[u] + 1 = d[v]$, da wie bereits erwähnt, u zuvor aus der Schlange entfernt wurde, und nach Annahme die Aussage des Lemmas vor der Operation $\text{ENQUEUE}(Q, v)$ gilt. Alle anderen Ungleichungen bleiben unangetastet und wir erhalten zusammenfassend

$$d[v_1] \leq d[v_2] \leq \dots d[v_r] \leq d[v] \leq d[v_1] + 1,$$

wie gewünscht. □

3.19 Kor. Seien v' und v'' Knoten die während der Ausführung von $\text{BREITENSUCHE}(s)$ in Q eingefügt werden, und nehmen wir weiterhin an, dass v' vor v'' eingefügt wird. Dann gilt $d[v'] \leq d[v'']$ zum Zeitpunkt des Einfügens von v'' .

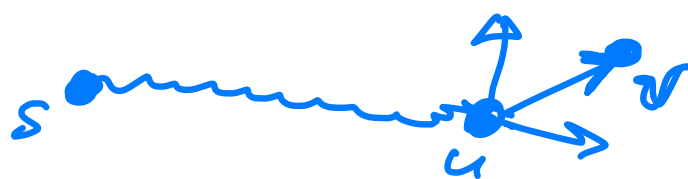
Beweis. Aus Lemma 3.18 folgt: solange Q nicht leer ist, wächst das Maximum von Q_d monoton während der Ausführung der Breitensuche (monotones Wachstum im nicht-strikten Sinn). Die Entfernung des Kopfs verändert Q_d nicht, solange die Schlange Q nicht leer ist. Beim Hinzufügen eines neuen Elements zu Q bleibt das Maximum im Fall $Q_d = [t, \dots, t, t + 1, \dots, t + 1]$ unverändert und es wächst um eine Einheit bei $Q_d = [t, \dots, t]$, weil nach dem Hinzufügen $Q_d = [t, \dots, t, t + 1]$ ist. \square

3.20 Thm. Sei $D = (V, A)$ ein Digraph und sei $s \in V$ ein beliebiger Knoten. Dann entdeckt $\text{BREITENSUCHE}(s)$ während ihrer Ausführung jeden Knoten, der von s aus erreichbar ist.


Bei der Terminierung gilt $d[v] = \delta(s, v)$, für alle $v \in V$.

Außerdem besteht für jeden von s aus erreichbaren Knoten $v \in V \setminus \{s\}$, einer der kürzesten (s, v) -Pfade aus einem kürzesten $(s, \pi[v])$ -Pfad und der Kante $(\pi[v], v)$.

Beweis. Wir führen einen Widerspruchsbeweis und nehmen dazu an, dass es einen Knoten v gibt, so dass $d[v] \neq \delta(s, v)$ gilt. Sei weiterhin angenommen, dass v ein solcher Knoten mit kleinstem Abstand $\delta(s, v)$ zu s ist. Da $d[s] = 0 = \delta(s, s)$ gilt, ist demnach in jedem Fall $v \neq s$. Nach Lemma 3.16 ist $d[v] \geq \delta(s, v)$ und daher $d[v] > \delta(s, v)$ für diesen ausgezeichneten Knoten v . Weiterhin muss v von s aus erreichbar sein, da ansonsten $\delta(s, v) = \infty \geq d[v]$ gelte. Sei nun u ein Knoten auf einem kürzesten (s, v) -Pfad, der unmittelbar vor v liegt, so dass $\delta(s, v) = \delta(s, u) + 1$ gilt. Die Wahl des Knotens v zusammen mit $\delta(s, u) < \delta(s, v)$ ergibt die



$$d[v] > \delta(s, v) \\ \delta(s, u) < \delta(s, v)$$

$$Q = [u, \dots] \rightarrow Q = [\dots]$$


81

3. BREITENSUCHE

Identität $d[u] = \delta(s, u)$, und wir fassen unsere Beobachtungen in folgender Ungleichungskette zusammen

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1. \quad (\text{IV.1})$$

Sehen wir uns den Zeitpunkt an, zu dem der Algorithmus $\text{BREITENSUCHE}(s)$ in Zeile 3 den Knoten u aus der Warteschlange entfernt. Wir unterscheiden danach, welche Farbe $\text{FARBE}[v]$ der Knoten v zu diesem Zeitpunkt trägt:

Ist $\text{FARBE}[v] = \text{WEISS}$, so wird in Zeile 10 $d[v] = d[u] + 1$ gesetzt und danach im Algorithmus nicht mehr verändert. Dies steht im Widerspruch zu (IV.1).

Ist $\text{FARBE}[v] = \text{GRAU}$, so ist v beim Sondieren eines anderen Knotens w grau gefärbt worden. Dieser Knoten wurde früher als u aus der Warteschlange entfernt und es wurde $d[v] = d[w] + 1$ gesetzt. Nach Korollar 3.19 gilt daher $d[w] \leq d[u]$ und damit $d[v] = d[w] + 1 \leq d[u] + 1$, im Widerspruch zu (IV.1).

Ist $\text{FARBE}[v] = \text{SCHWARZ}$, so wurde v bereits aus der Warteschlange entfernt und nach Korollar 3.19 gilt $d[v] \leq d[u]$, im Widerspruch zu (IV.1).

Zusammenfassend erhalten wir in jedem Fall einen Widerspruch zu (IV.1) und damit gilt also $d[v] = \delta(s, v)$, für alle Knoten $v \in V$.

Alle von s aus erreichbaren Knoten werden bei der Breitensuche entdeckt, da für einen ansonsten unentdeckten Knoten v der Initialwert $d[v] = \infty$ größer als $\delta(s, v)$ wäre. Abschließend bemerken wir noch, dass die Verwaltung der Vorgängerabbildung π und des Abstandsattributs d in den Zeilen 9 und 10 von $\text{BREITENSUCHE}(s)$ impliziert, dass $\delta(s, v) = d[v] = d[\pi[v]] + 1 = \delta(s, \pi[v]) + 1$ gilt und somit jeder kürzeste $(s, \pi[v])$ -Pfad über die Kante $(\pi[v], v)$ zu einem kürzesten (s, v) -Pfad erweitert werden kann. \square

$$d[v] = d[\pi[v]] + 1$$

$$\begin{array}{l} \begin{array}{c} u \xrightarrow{\quad} v \\ \left\{ \begin{array}{l} d[v] = d[u] + 1 \\ \pi[v] = u \end{array} \right. \end{array} \end{array}$$

3.21 Def. Die Vorgängerabbildung π aus der Breitensuche erzeugt auf einem Digraphen $D = (V, A)$, die von einem Startknoten $s \in V$ ausgeht, den **Vorgängerteilgraphen** $D_\pi = (V_\pi, A_\pi)$, der durch

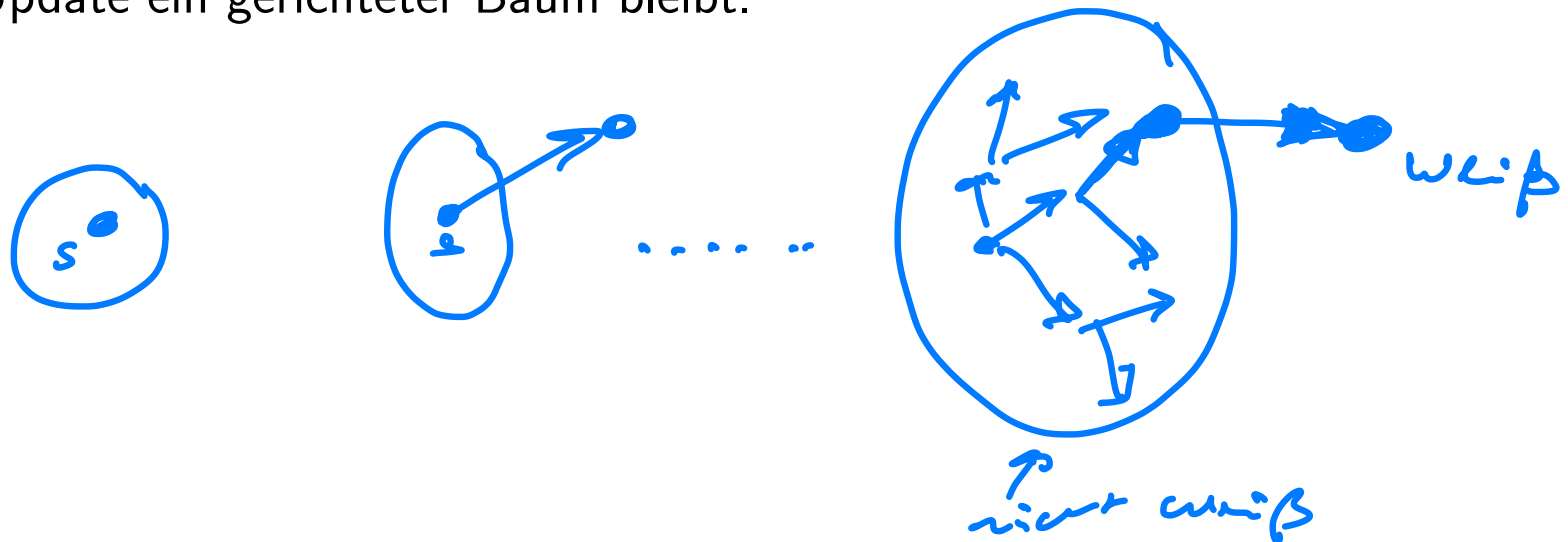
$$V_\pi := \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\} \quad \text{und} \quad A_\pi := \{(\pi[v], v) : v \in V_\pi \setminus \{s\}\}$$

definiert ist.

Es kann gezeigt, werden dass D_π ein gerichteter Baum mit Wurzel s ist. Dieser Baum nennt man den **Breitensuchbaum** der Breitensuche. die Kanten von D_π werden die **Baumkanten** der Breitensuche genannt.

3.22 Prop. Zu jeder Zeit der Ausführung der Breitensuche auf einem Digraphen $D = (V, A)$ ist D_π ein gerichteter Baum.

Beweis. Die Aussage kann durch Induktion über die Anzahl der Färbungen von weißen Knoten mit der grauen Farbe bewiesen werden. Nach der ersten Färbung ist D_π der Baum mit einem einzigen Knoten s . Nach jeder neuen Färbung von einem weißen Knoten v wird ein Knoten v , der außerhalb von D_π liegt D_π hinzugefügt und es entsteht eine neue Kante (u, v) zwischen einem Knoten u , der bereits in D_π liegt, und dem neu hinzugefügten Knoten v . Man sieht, dass D_π nach diesem Update ein gerichteter Baum bleibt. □



3.23. Zur Berechnung der kürzesten Pfade kann die folgende Prozedur benutzt werden:

PFAD-AUSGEBEN(v)

1: **if** $v = s$:

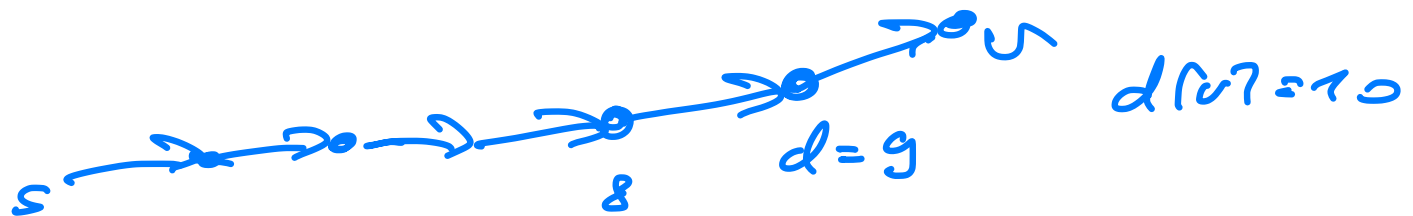
2: print s

3: **else:**

4: PFAD-AUSGEBEN($s, \pi[v]$)

5: print v

6: **end**



3.24 Prop. Ist π die Vorgängerabbildung zur Breitensuche auf $G = (V, A)$ mit dem Startknoten s , so gibt die Prozedur $\text{PFAD-AUSGEBEN}(v)$ folgende Algorithmus für jedes $v \in V$ mit $\delta(s, v) < \infty$ einen kürzesten (s, v) -Pfad aus.

Beweis. Dass die Prozedur $\text{PFAD-AUSGEBEN}(v)$ terminiert und einen kürzesten (s, v) -Pfad ausgibt kann durch Induktion über $\delta(s, v)$ gezeigt werden. Ist $\delta(s, v) = 0$ so ist $v = s$ und die Behauptung ist klar. Angenommen, die Behauptung gelte für alle $v \in V$ mit $\delta(s, v) = t - 1$ für ein $t \in \mathbb{N}$.

Wir betrachten ein $v \in V$ mit $\delta(s, v) = t$. In der Breitensuche setzt man $\pi[v] := u$ gleichzeitig mit $d[v] = d[u] + 1$ für $v \in N[u]$. Daher hat man $t = d[v] = d[\pi[v]] + 1$, sodass $d[\pi[v]] = t - 1$ gilt. Nach der Induktionsvoraussetzung gibt $\text{PFAD-AUSGEBEN}(s, \pi[v])$ den $(s, \pi[v])$ -Pfad der Länge $\delta(s, \pi[v]) = d[\pi[v]] = t - 1$ aus. Also gibt $\text{PFAD-AUSGEBEN}(v)$ den (s, v) -Pfad der Länge $t = d[v] = \delta(s, v)$ aus, was die Korrektheit und die Terminierung der Ausführung

3. *BREITENSUCHE*

87

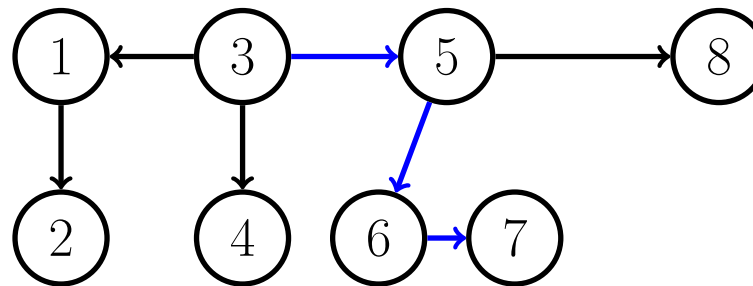
Prozedur PFAD-AUSGEBEN auf dem Knoten v bestätigt.

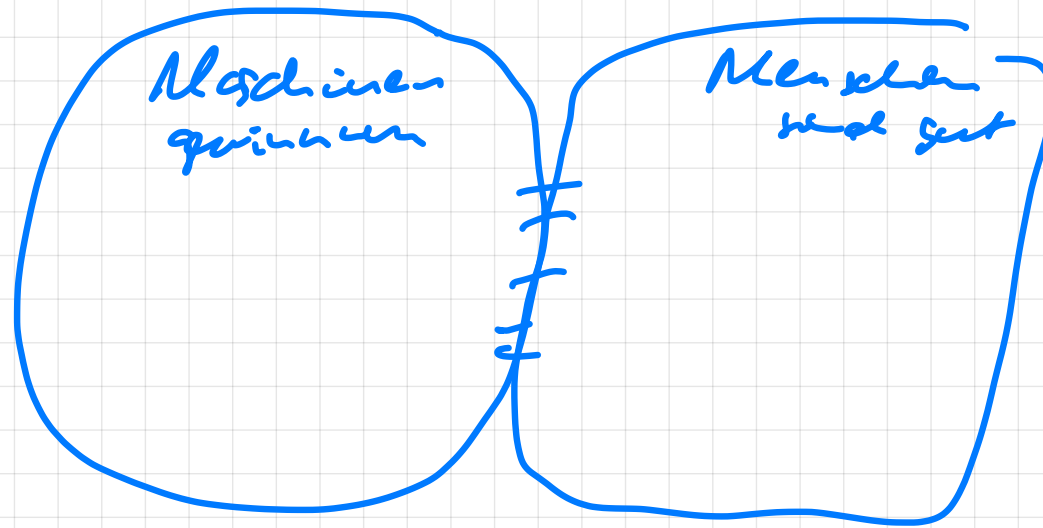


3.25 Bsp. Sammeln wir die Daten zur Vorgängerabbildung und zu den Distanzen während der Breitensuche in Beispiel 3.11, so erhalten wir nach Abschluss von BREITENSUCHE(3) die folgenden Werte:

Knoten u	1	2	3	4	5	6	7	8	9	10
$\pi[u]$	3	1	NIL	3	3	5	6	5	NIL	NIL
$d[u]$	1	2	0	1	1	2	3	2	∞	∞

Daraus ergibt sich der nachfolgende Breitensuchbaum, in dem der kürzeste $(3, 7)$ -Pfad in blau markiert ist:





$$y = f_p(x)$$

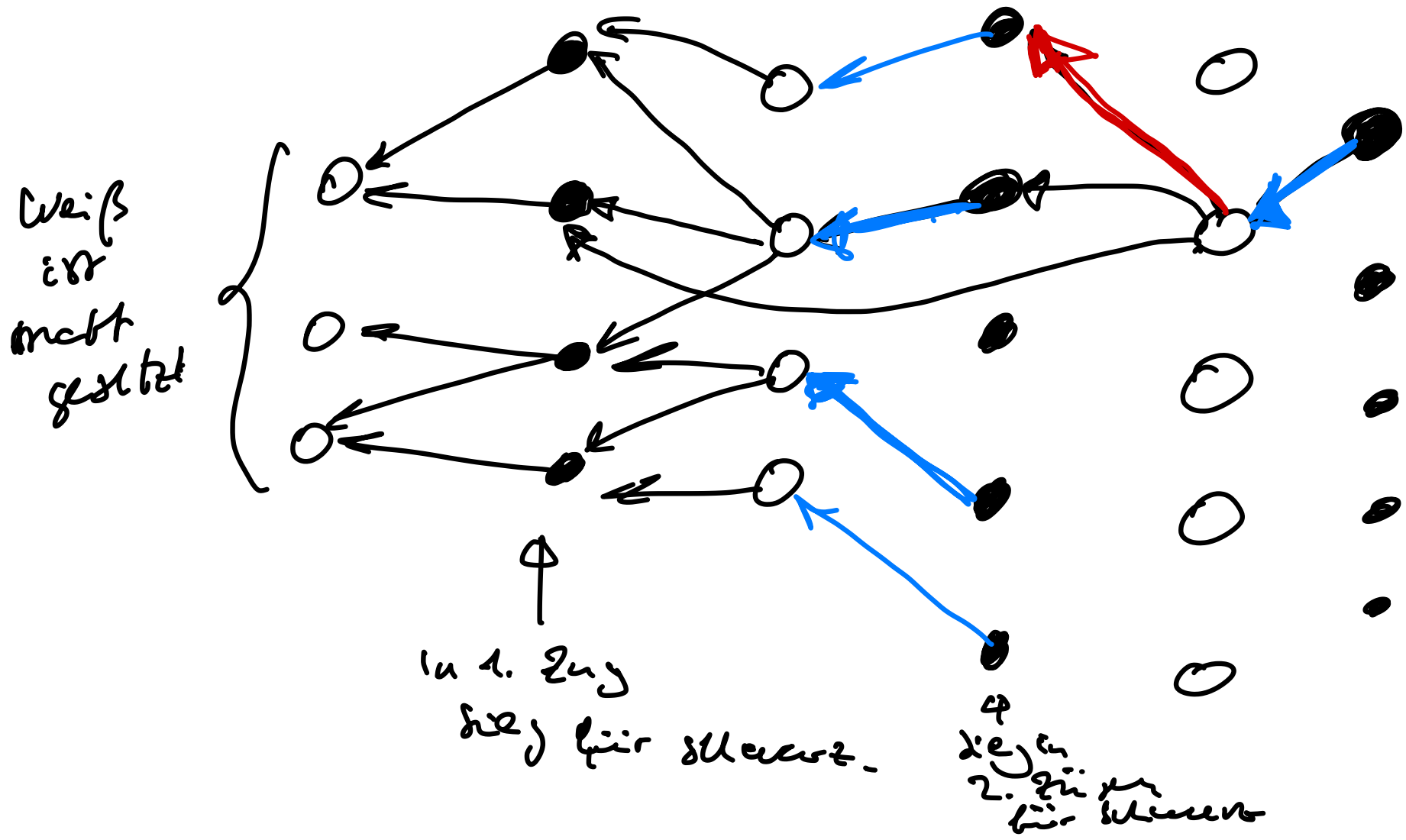
3.26. Einige Anwendungen der Breitensuche:

- Bestimmung der Zusammenhangskomponente des Knotens s bei einem ungerichteten Graphen G .
- Umsetzung von Web-Crawlern, die das Umfeld einer Webseite s bestimmen, als die Menge von Webseiten, die innerhalb von gegebener Anzahl von Klicks k von s aus erreichbar sind.
- Bei verschiedenen Arten von Spielen lässt sich die Frage stellen: kann man den Gewinn von einer gegebenen Position aus erzwingen und ggf. wie? Dafür benutzt man etwa im Schach die sogenannten Endspieldatenbanken. Die Generierung von Endspieldatenbanken basiert im Wesentlichen auf der Breitensuche. Vgl. den Abschnitt Generating Tablebases in https://en.wikipedia.org/wiki/Endgame_tablebase. Endspieldatenbanken kann man für verschiedene Spiele generieren (Treblecross, Hex, Go usw.), die Anzahl

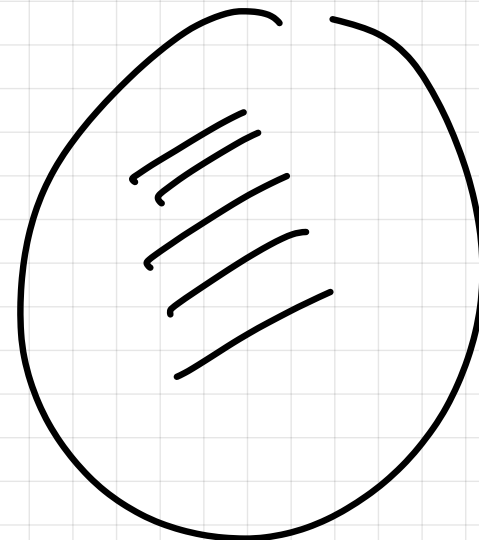
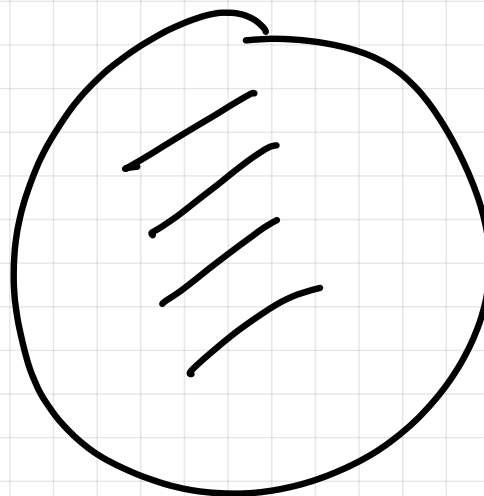
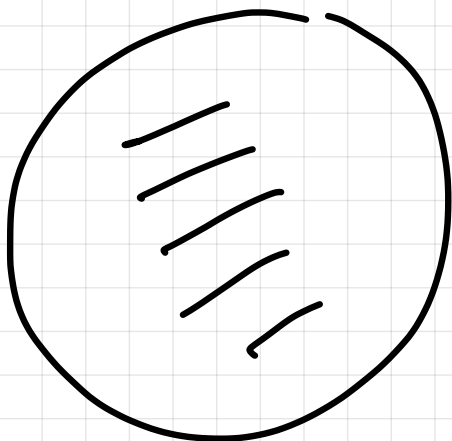
Nim

der Positionen ist aber ein mögliches praktisches Hindernis, da der Graph der Endspiele, den man kann erzeugen, potenziell extrem groß sein kann.

KT \mathbb{R}



A
→
B



3.27 Aufg. Wie kann man auf der Basis der Breitensuche testen, ob ein gegebener Graph bipartit ist?

3.28 Aufg. Implementieren Sie die Breitensuche. Testen Sie die Laufzeit Ihrer Implementierung auf den Graphen den realen Straßennetzwerken, die in der 9th DIMACS Implementation Challenge benutzt wurden. Die Graphen sind unter <http://www.diag.uniroma1.it/challenge9/download.shtml> verfügbar. Diese Graphen sind gewichtet: für den Test Ihrer Implementierung können Sie die Gewichte ignorieren.

3.29. Die hier gegebene Präsentation der Breitensuche basiert auf [CLRS17].

4 Tiefensuche

4.1. Die Eingabe für die hier diskutierte Tiefensuche ist ein Digraph $D = (V, A)$. Ist der Graph ungerichtet, so ist die Vorgehensweise vollkommen analog und daher diskutieren wir hier (fast ausschließlich) die gerichtete Variante. Der Digraph D ist durch seine Adjazenzliste N gegeben, das heißt, für jeden Knoten $u \in V$ ist $N[u]$ eine Liste aller $v \in V$ mit $(u, v) \in A$. Die Darstellung von D als Adjazenzliste eine Durchmusterung mit der optimalen Laufzeit $\Theta(|V| + |A|)$ erlaubt.

V als array

$u \in V$ $N(v)$ als Liste.

4.2. Die Tiefensuche ist ein “reiselustiger” Algorithmus zur Durchmusterung von Graphen. Wenn wir Knoten als Orte auffassen und $N[u]$ als die Umgebung des Ortes u , so können wir das Grundgerüst der Tiefensuche so beschreiben. Man will alle noch nicht gesehenen Orte erkunden, die man von einem gegebenen Ort u aus erreichen kann. Man kann dazu das folgende Vorgehen benutzen:

- Man sucht die Umgebung von u durch.
- Sobald man einen noch nicht besuchten Ort v sieht, erkundet man alle noch nicht gesehene Orte, die man von v aus erreichen kann.

Der Grundgedanke der Tiefensuche rekursiv: man erkundet neue Orte von u aus, indem man die Orte erkunden, die man aus noch nicht gesehenen Nachbarschaft von u erreichen kann. Die “Reiselust” dieser Durchmusterung besteht darin, dass man sich die Erkundung der Orte *sofort* einen neuen Ort versetzt, wenn ein solcher Ort gefunden wird.

Bei rekursiven Algorithmen soll man sich bei der Umsetzung in der Regel als Erstes darum kümmern, dass der Algorithmus terminiert. Daher soll man bei der Durchmusterung die Orte, an die man kommt, gleich als “gesehen” markieren. Stellen Sie sich einfach vor, dass Sie ein Stück Kreide nehmen und da, wo Sie angekommen sind, gleich schreiben “hier war ich schon”. Sollten Sie bei Ihrer Wanderung durch den Graphen noch ein mal an diesen Ort kommen, so werden Sie merken, dass Sie an diesem Ort bereits gewesen sind. Auf diese Weise werden Sie nicht endlos im Kreis herumlaufen, wenn etwa Ihr Graph ein Kreis ist oder einen Kreis enthält.

Die Umsetzung der oben beschriebenen Strategie basiert auf Farbattributen der Knoten, die man im Laufe der Durchmusterung sukzessiv zuerst als weiß (= noch nicht gesehen), dann als grau (= gesehen aber noch nicht abgearbeitet) und schließlich als schwarz (= abgearbeitet) setzt.

4.3 Def. Wenn die Tiefensuche entlang einer Kante verläuft, die in einen weißen Knoten endet, dann sagen wir, dass der entsprechende Knoten **entdeckt** wird.

Hier eine Zusammenfassung der Bedeutung der Farben der Knoten.

weiß: Der Knoten wurde noch nicht entdeckt.

grau: Der Knoten wurde entdeckt und die Tiefensuche für den Knoten läuft gerade.

schwarz: Der Knoten wurde entdeckt und die Tiefensuche für den Knoten ist bereits beendet.

Wird ein Knoten schwarz gefärbt, so sagen wir auch das er **abgearbeitet** wurde.

Durch die Tiefensuche können verschiedene Aufgaben erledigt werden. Nicht alle diese Aufgaben benötigen alle drei Farbattributen der Knoten. Bei manchen Aufgaben reicht auch

die Unterscheidung zwischen “entdeckt” (weiß) und “nicht entdeckt” (grau oder schwarz) aus.

4.4. Auf diese Weise können wir die Tiefensuche initialisiert:

TIEFENSUCHE-INITIALISIEREN(D)

1: **for** $u \in V$:

2: FARBE[u] = WEISS

3: $\pi[u]$ = NIL

4: **end**

4.5. Sobald die Initialisierung erfolgt ist, können wir die Tiefensuche von einem gegebenen Knoten ausführen. Hier die rekursive Umsetzung:

TIEFENSUCHE(u)

- 1: FARBE[u] := GRAU ▷ *u als entdeckt notiert*
- 2: **for** $v \in N[u]$:
- 3: ▷ *Die Kante (u, v) wird sondiert*
- 4: **if** FARBE[v] = WEISS :
- 5: $\pi[v] := u$ ▷ *Knoten v wurde entdeckt*
- 6: TIEFENSUCHE(v)
- 7: **end**
- 8: **end**
- 9: FARBE[u] := SCHWARZ

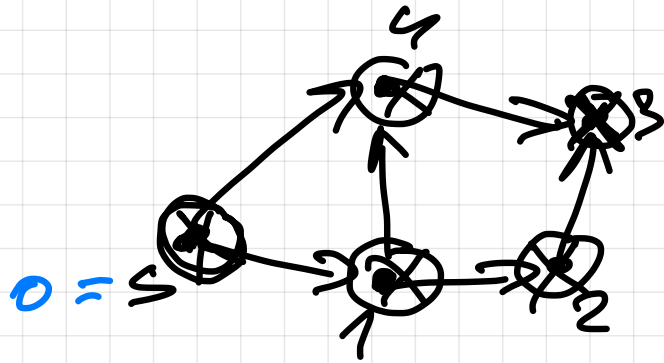
4.6. Die Prozedur Tiefensuche können wir in den folgenden Rahmen einbauen.

VOLLSTÄNDIGE-TIEFENSUCHE(D)

```
1: TIEFENSUCHE-INITIALISIEREN( $D$ )  
2: for  $u \in V$  :  
3:   if FARBE[ $u$ ] = WEISS :  
4:     TIEFENSUCHE( $u$ )  
5:   end  
6: end
```

Durch diese Prozedur entdecken wir jeden jeden Knoten genau ein mal und sondieren dabei jede Kante von $D = (V, A)$.

Bsp



0: 1 4

1: 2 4

2: 3

3:

4: 3

$TS(0) \rightarrow TS(1) \rightarrow TS(2) \rightarrow TS(3)$

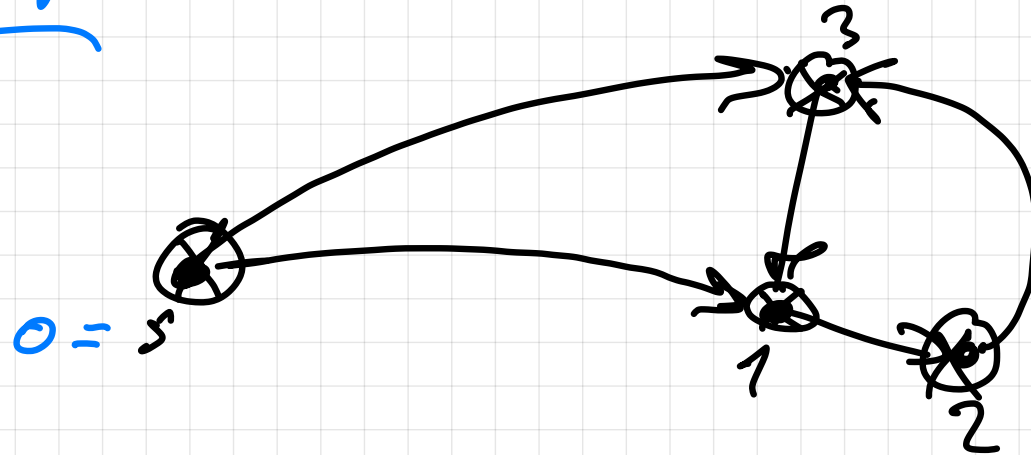
$TS(0) \rightarrow TS(1) \rightarrow TS(2)$

$TS(0) \rightarrow TS(1) \rightarrow TS(4)$

$TS(0) \rightarrow TS(1)$

$TS(0)$

Bsp



0 : 1 3
1 : 2
2 : 3
3 : 1

$TS(0) \rightarrow TS(1) \rightarrow TS(2) \rightarrow TS(3)$

$TS(0) \rightarrow TS(1) \rightarrow TS(2)$

$TS(0) \rightarrow TS(1)$

$TS(0)$

4.7 (Variante mit Zeitstempeln). Jeder Knoten $v \in V$ kann während der Tiefensuche mit zwei *Zeitstempeln* versehen werden: Der erste Zeitstempel $\text{GRAU}[v]$ zeichnet auf, wann der Knoten grau gefärbt wird, das heißt, wann er das erste Mal entdeckt wird. Der zweite Zeitstempel $\text{SCHWARZ}[v]$ hingegen, speichert den Zeitpunkt der Schwarzfärbung von v , das heißt, den Moment in dem der Knoten abgearbeitet ist.

Im Algorithmus werden beide Zeitstempel ganze Zahlen zwischen 1 und $2|V|$ sein, da es für jeden Knoten $v \in V$ genau einen Zeitpunkt der Entdeckung (Graufärbung) und einen Zeitpunkt der Abarbeitung (Schwarzfärbung) gibt.

Für jedes $v \in V$ gilt $\text{GRAU}[v] < \text{SCHWARZ}[v]$. Weiterhin ist v vor dem Zeitpunkt $\text{GRAU}[v]$ weiß, während der Zeitpunkte $\text{GRAU}[v], \dots, \text{SCHWARZ}[v] - 1$ grau, und ab dem Zeitpunkt $\text{SCHWARZ}[v]$ schwarz.

Für die Variante mit den Zeitstempeln sollen die Initialisierung und die Tiefensuche geringfügig ergänzt werden.

Die Initialisierung:

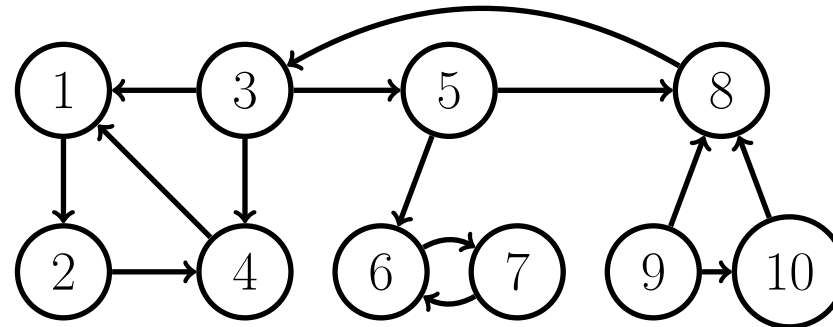
TIEFENSUCHE-INITIALISIEREN(D)

- 1: **for** $u \in V$:
 - 2: FARBE[u] = WEISS
 - 3: $\pi[u]$ = NIL
 - 4: **end**
 - 5: $t := 0$ ▷ *Initialisierung der Zeit-Variablen*
-

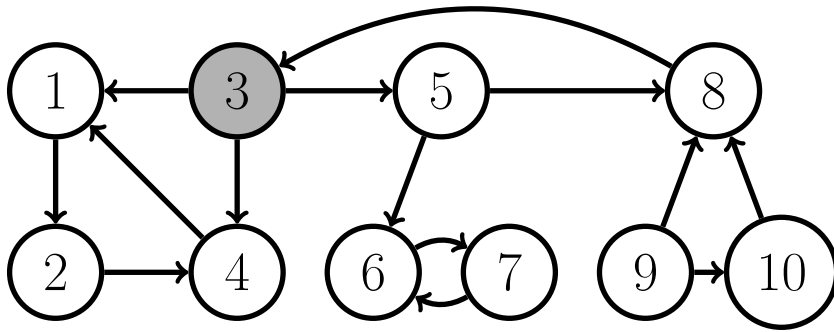
Die Tiefensuche:

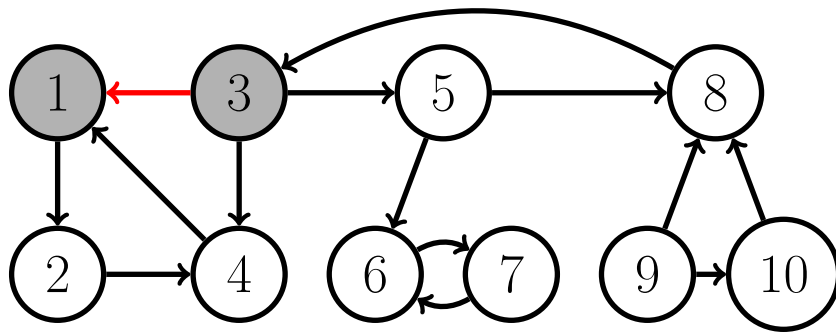
TIEFENSUCHE(u)

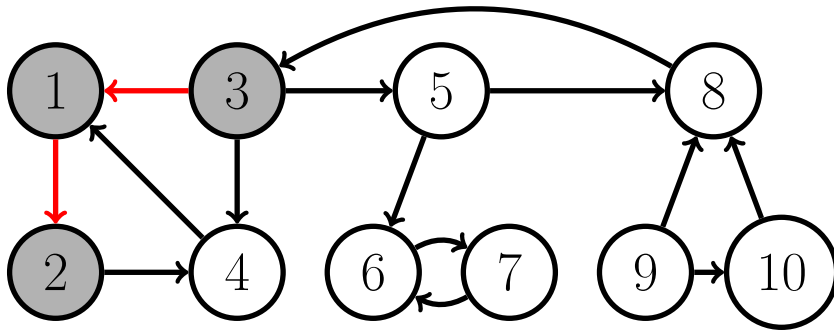
- 1: $t := t + 1$ \triangleright *Die Uhr tickt vor jeder Färbung*
 - 2: **GRAU**[u] := t \triangleright *Der Knoten u wurde gerade entdeckt*
 - 3: **FARBE**[u] := **GRAU**
 - 4: **for** $v \in N[u]$:
 - 5: **if** **FARBE**[v] = **WEISS** :
 - 6: $\pi[v] := u$
 - 7: **TIEFENSUCHE**(v)
 - 8: **end**
 - 9: **end**
 - 10: $t := t + 1$ \triangleright *Die Uhr tickt vor jeder Färbung*
 - 11: **SCHWARZ**[u] := t \triangleright *Der Knoten u wurde gerade abgearbeitet*
 - 12: **FARBE**[u] := **SCHWARZ**
-

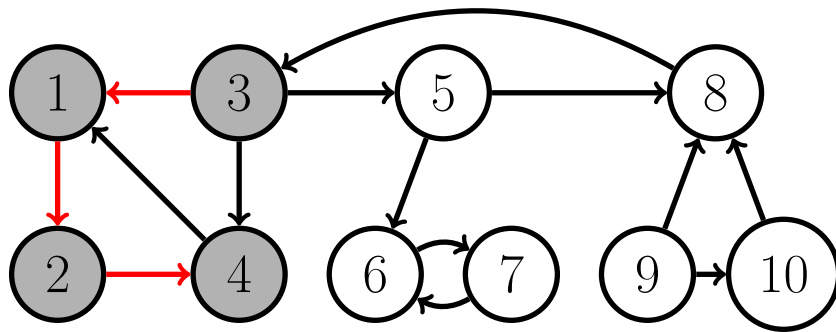
4.8 Bsp. Wir illustrieren die Tiefensuche am Digraphen

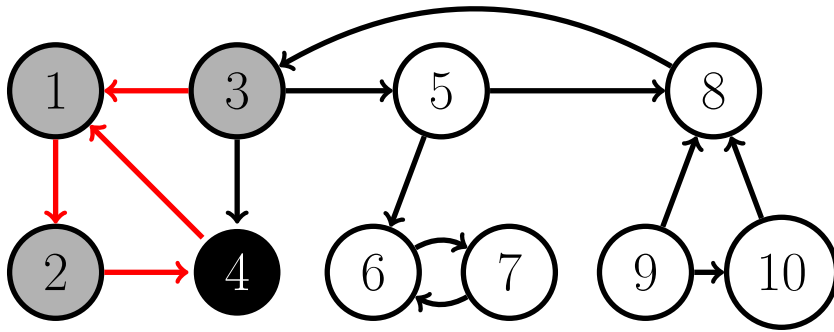
Wir wählen als Startknoten den Knoten 3 und treffen ansonsten jede Wahl aufsteigend in der Reihenfolge der Knotenindizes. Die ersten sechs Zeitschritte sind durch die folgende Sequenz von gefärbten Digraphen gegeben, wobei die Knotenfarbe dem aktuellen Farbattribut entspricht und die bereits sondierten Kanten rot markiert sind:

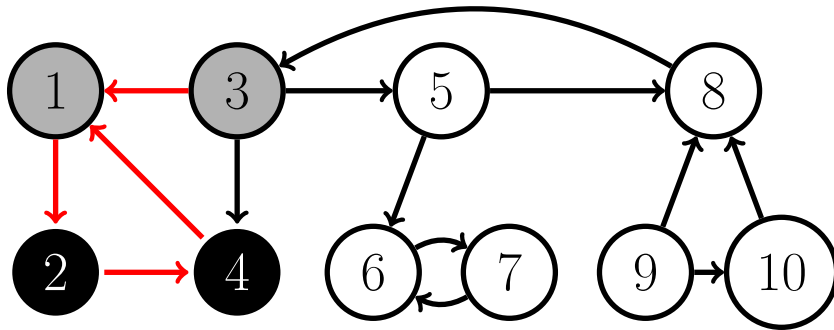
 $TS(3)$

 $TS(3) \rightarrow TS(1)$


$$TS(3) \rightarrow TS(1) \rightarrow TS(2)$$

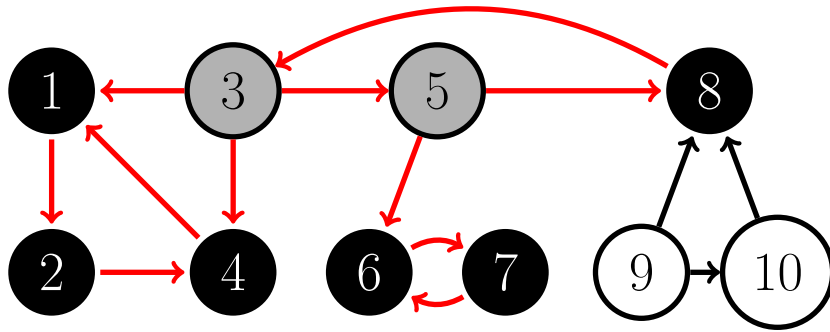

$$TS(3) \rightarrow TS(1) \rightarrow TS(2) \rightarrow TS(4)$$


$$TS(3) \rightarrow TS(1) \rightarrow TS(2)$$



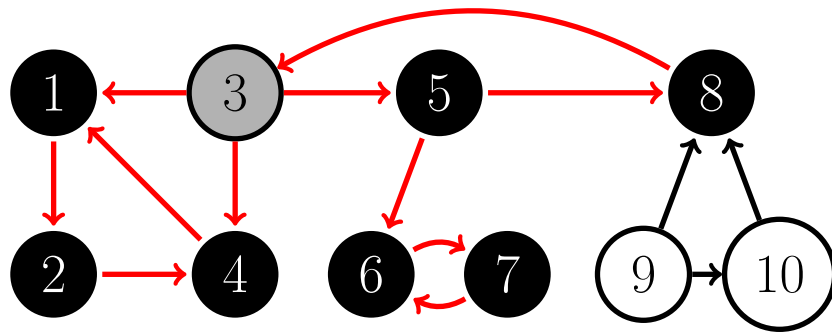
$TS(3) \rightarrow TS(1)$

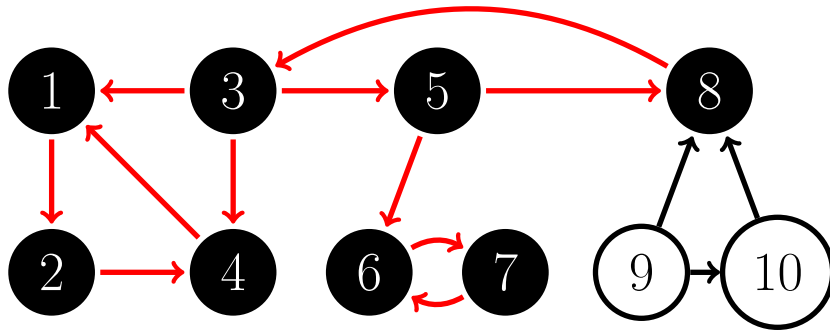
Wir steigen zum Zeitpunkt $SCHWARZ[8] = 14$ mit der Illustration wieder ein:


$$TS(3) \rightarrow TS(5)$$

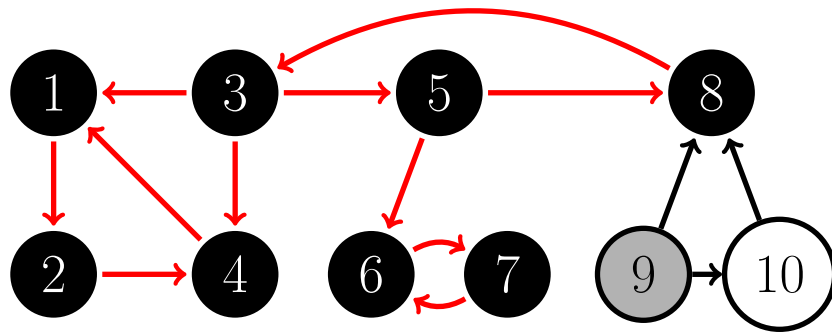
4. TIEFENSUCHE

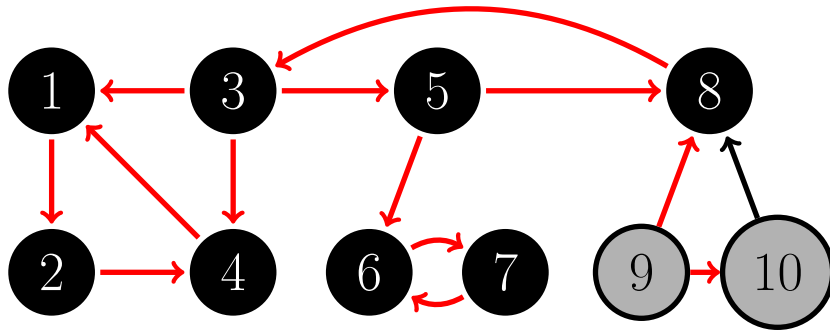
115





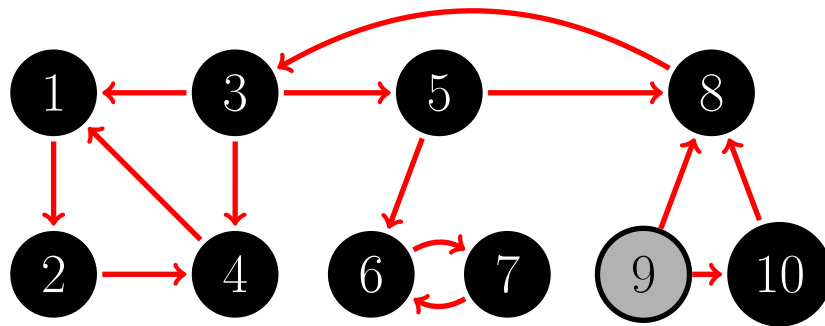
Nun muss ein neuer Knoten gewählt werden (Knoten 9) um die Tiefensuche für den ganzen Digraphen zu beenden:

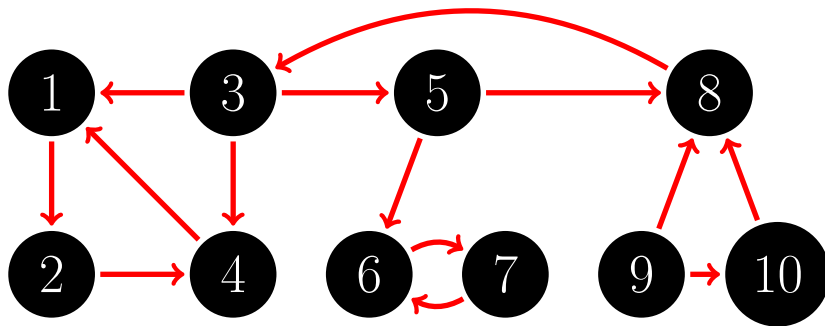




4. TIEFENSUCHE

119





Nun sind alle Knoten schwarz und die Tiefensuche ist beendet.

4.9. Bevor wir die Anwendungen der Tiefensuche diskutieren, analysieren wir Ihre Laufzeit.

4.10 Thm. *Es sei ein Digraph $D = (V, A)$ durch eine Adjazenzliste gegeben. Dann hat die Laufzeit von $\text{TIEFENSUCHE}(s)$ die Ordnung $O(|V| + |A|)$ und die Laufzeit von $\text{VOLLSTÄNDIGE-TIEFENSUCHE}(s)$ die Ordnung $\Theta(|V| + |A|)$.*

Beweis. Der Aufwand setzt sich aus den folgenden Operationen zusammen, die man den Knoten und Kanten zuordnet:

$$\text{WEISS} \xrightarrow{TS(u) \text{ Start}} \text{GRAU} \xrightarrow{\text{Sondierung von } N[u]} \text{SCHWARZ} \xrightarrow{TS(u) \text{ Ende}}$$

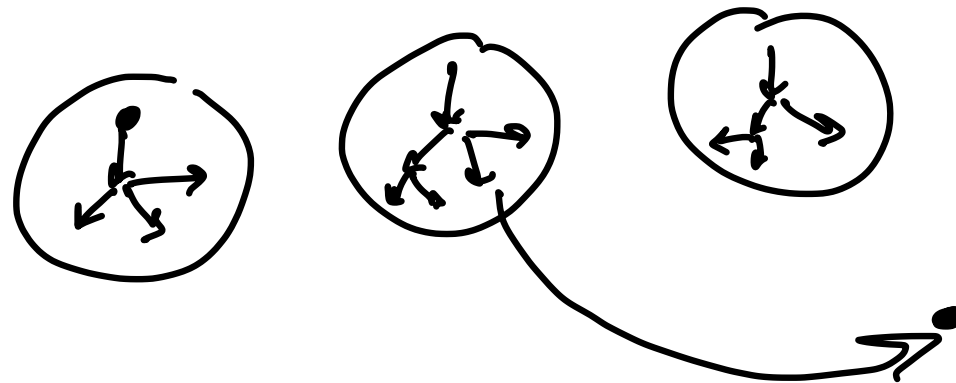
Hier wird TS als Abkürzung für TIEFENSUCHE benutzt. Für die Tiefensuche und vollständige Tiefensuche ist der Aufwand somit höchstens $O(|V| + |E|)$, denn der Aufwand pro Knoten $O(1)$, da kein Knoten u wegen der Änderung der Farben mehr als ein mal sondiert werden kann. Genau so ist der Aufwand pro jede Kante $O(1)$, da eine Kante (u, v) genau dann sondiert wird, wenn man u entdeckt, der Knoten u wird aber höchstens ein mal entdeckt.

Es ist klar, dass der Aufwand bei der vollständigen Tiefensuche $\Omega(|V| + |A|)$, da die For-Schleife in der vollständige Tiefensuche dafür sorgt, dass jeder Knoten ein mal entdeckt wird. \square

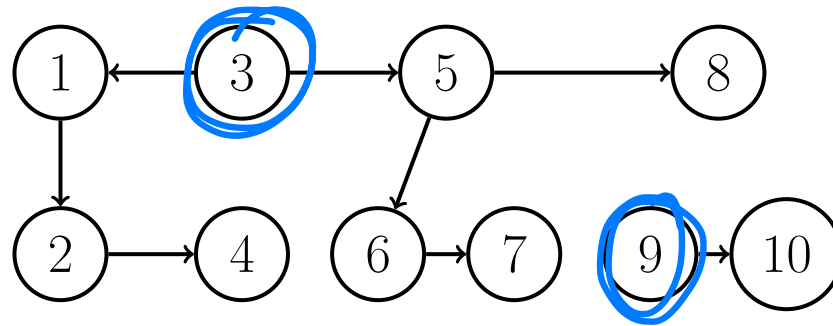
4.11 Def. Die Vorgängerabbildung π erzeugt den sogenannten **Vorgängerteilgraphen** eines Digraphen $D = (V, A)$, der formal durch $D_\pi = (V, A_\pi)$ mit

$$A_\pi = \{(\pi[v], v) : v \in V \text{ und } \pi[v] \neq \text{NIL}\}$$

definiert ist. Für jede Tiefensuche ist der Vorgängerteilgraph ein Wald, und wird daher im Folgenden als **Tiefensuchwald** bezeichnet. Er ist aus einem oder mehreren **Tiefensuchbäumen** zusammengesetzt.



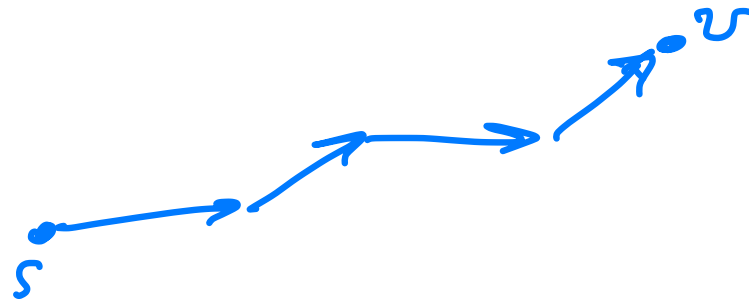
4.12 Bsp. In Beispiel 4.8 ist die Vorgängerabbildung nach abgeschlossener Tiefensuche durch $\pi = [3, 1, \text{NIL}, 2, 3, 5, 6, 5, \text{NIL}, 9]$ gegeben. Der zugehörige Tiefensuchwald ist also



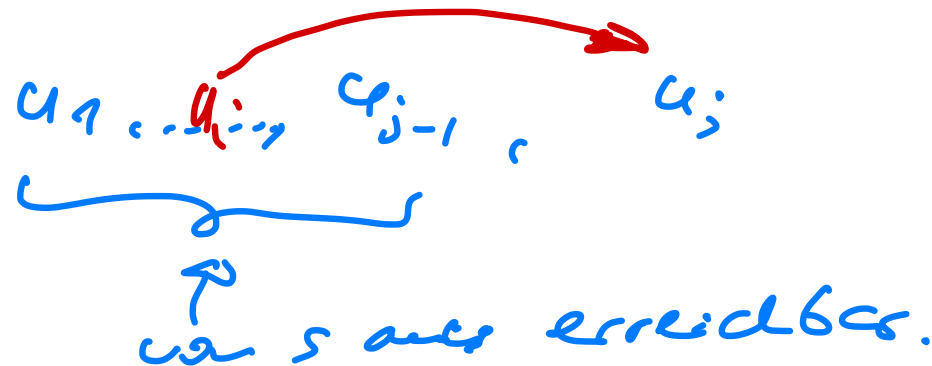
4.13 Thm. Sei $D = (V, A)$ Digraph, der durch eine Adjazenzliste gegeben ist. Sei $s \in V$. Dann gilt für die Tiefensuche auf D mit dem Startknoten s :

- (a) Die Menge aller Knoten von D , die von s aus durch einen Pfad erreichbar sind ist genau die Menge der Knoten, die während der Ausführung entdeckt werden.
- (b) D enthält genau dann einen von s aus erreichbaren Zyklus, wenn während der Ausführung beim Sondieren einer der Kanten (u, v) die Farbe von v grau ist.

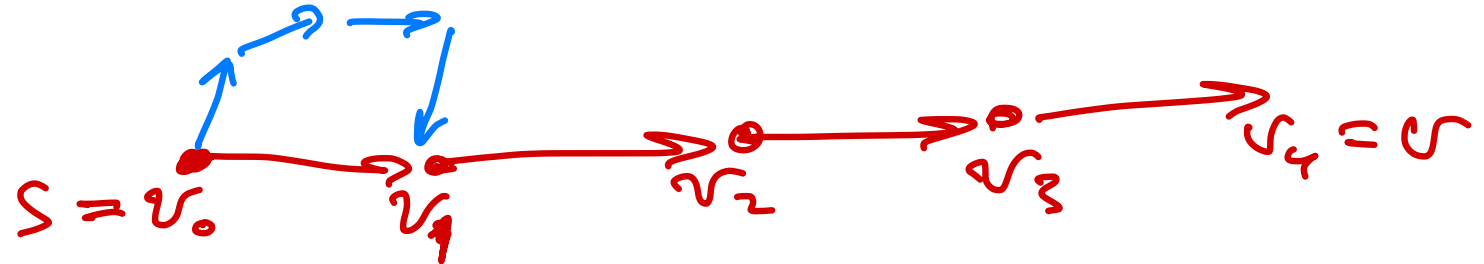
Beweis. Wir bezeichnen die Tiefensuche kurz als TS.



(a): Seien u_1, \dots, u_k alle Knoten, die während der Ausführung entdeckt werden und seien u_1, \dots, u_k in dieser Reihenfolge entdeckt. Dann ist u_1 von s aus erreichbar, denn $u_1 = s$. Die TS für einen Knoten u_j mit $j > 1$ wird aus einer TS für einen Knoten u_i aufgerufen, der im Moment des Aufrufs von $TS(u_j)$ bereits entdeckt ist. Man hat also $j < i$. Wenn u_j von s aus erreichbar ist, so ist auch u_i von s aus erreichbar, da (u_i, u_j) eine Kante von G ist. Somit folgt durch Induktion über j , dass jeder Knoten u_j von s aus erreichbar ist.

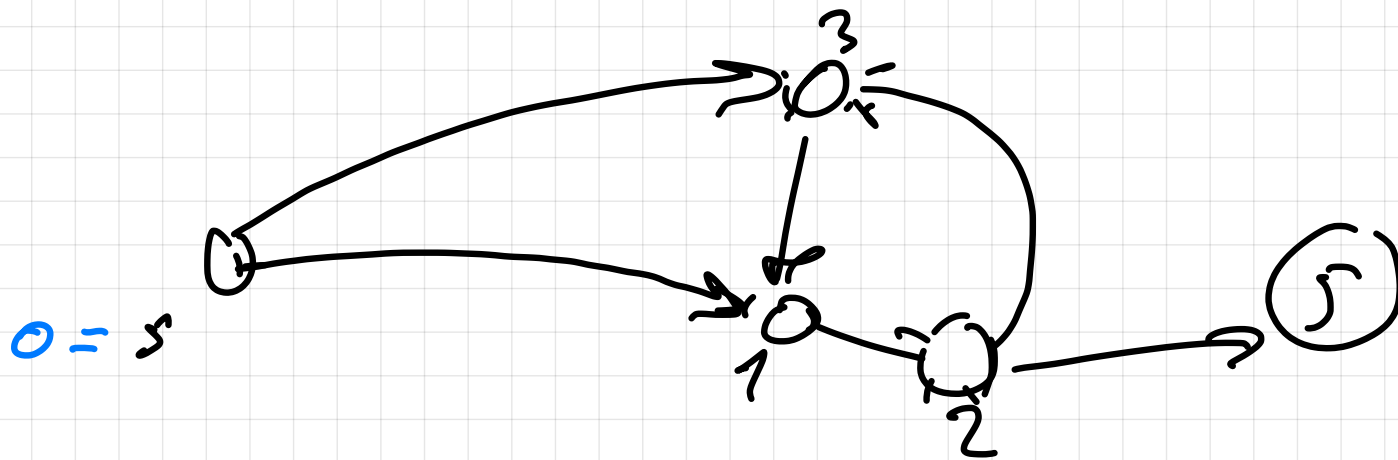


Umgekehrt zeigen wir, dass jeder Knoten $v \in V$, der von s aus erreichbar ist, während der Ausführung entdeckt wird. Sei (v_0, \dots, v_k) ein Pfad von s nach v . Wir zeigen, dass jeder Knoten v_j dieses Pfades entdeckt wird. Für $v_0 = s$ gilt die Aussage offensichtlich. Wird ein Knoten v_j mit $j < k$ entdeckt, so entdeckt man den Knoten v_{j+1} spätestens beim Sondieren der Kante (v_j, v_{j+1}) innerhalb der TS für v_j . Es kann als durch Induktion über j gezeigt werden, dass alle v_j und insbesondere auch $v_k = v$ entdeckt werden.



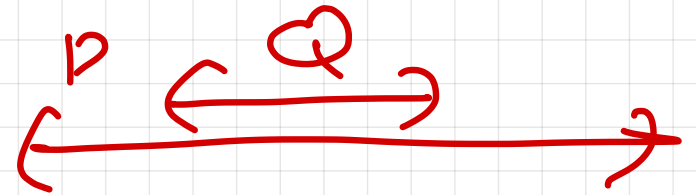
(b): Der Beweis von (b) ist analog zum Beweis von (a) und wird hier nicht angeführt.
(Aufgabe) □

4.14. Eine erste wichtige Eigenschaft der Tiefensuche ist der Zusammenhang von Entdeckungszeit und Abarbeitungszeit eines Knotens. Es stellt sich heraus, dass diese Zeitpunkte im folgenden Sinne eine *Klammerstruktur* aufweisen: Stellen wir die Entdeckungszeit eines Knotens u durch den Ausdruck „(u “ und die Abarbeitungszeit von u durch den Ausdruck „ u)“ dar, dann ergibt die gesamte Historie, über alle Knoten des (Di)Graphen hinweg gesehen, einen korrekt geklammerten Ausdruck.



$$\begin{array}{ccccccc}
 TS(0) \checkmark & TS(1) \checkmark & TS(2) \checkmark & TS(3) \checkmark & TS(3) \times & TS(5) \checkmark & TS(5) \times \\
 \hline
 TS(2) \times & TS(1) \times & TS(0) \times & & & &
 \end{array}$$

(0 (1 (2 (3 3) (5 5) 2) 1) 0)



4.15 Bsp. Zur Illustration dieses Zusammenhangs sehen wir uns wieder die Tiefensuche aus Beispiel 4.8 an. Die Daten der Zeitstempel sind in folgender Tabelle zusammengefasst:

Knoten u	1	2	3	4	5	6	7	8	9	10
Grau $[u]$	2	3	1	4	8	9	10	13	17	18
Schwarz $[u]$	7	6	16	5	15	12	11	14	20	19

Nach obiger Vorschrift können wir damit den zugehörigen Klammerausdruck

$$(3 (1 (2 (4 4) 2) 1) (5 (6 (7 7) 6) (8 8) 5) 3) (9 (10 10) 9)$$

ablesen.

Eine andere Möglichkeit diese Klammerstruktur auszudrücken ist im folgenden Satz festgehalten:

4.16 Def. Bzgl. einer vollständigen Tiefensuche auf $D = (V, A)$ mit Startknoten s definieren die **Lebenszeitintervall** eines Knoten $v \in V$ als die Menge

$$I_u := \{t \in \mathbb{Z} : \text{GRAU}[u] \leq t \leq \text{SCHWARZ}[u]\}.$$

4.17 Thm (Klammerungstheorem). *Bei der vollständigen Tiefensuche auf einem Digraphen $D = (V, A)$ ist für jedes Paar von Knoten u und v genau eine der drei folgenden Bedingungen erfüllt:*

- (a) $I_u \cap I_v = \emptyset$ und weder u noch v ist im Tiefensuchwald ein Nachfahre des anderen.*
- (b) $I_u \subseteq I_v$ und u ist im Tiefensuchwald ein Nachfahre von v .*
- (c) $I_v \subseteq I_u$ und v ist im Tiefensuchwald ein Nachfahre von u .*

Beweis. Die Aussage ist symmetrisch bzgl. u und v . Wir nehmen also ohne Beschränkung der Allgemeinheit an, dass während der Ausführung der vollständigen Tiefensuche der Knoten u als erster entdeckt wurde. Ist die Tiefensuche für v nach der Terminierung der Tiefensuche für u aufgerufen worden, so gilt $I_u \cap I_v = \emptyset$. Ansonsten ist die Tiefensuche für v vor der Terminie-

zung der Tiefensuche für u aufgerufen worden. Das bedeutet, dass der Knoten v während der Ausführung von u entdeckt worden ist. Der Aufruf von $TS(v)$ erfolgt also durch eine Folge der geschachtelten rekursiven Aufrufen

$$TS(u) \rightarrow \cdots \rightarrow TS(v)$$

Somit terminiert $TS(v)$ vor $TS(u)$. Es gilt also $I_v \subseteq I_u$ ist v ist Nachfahre von u . □

4.18. Als direkte Konsequenz des Klammerungstheorems erhalten wir eine Charakterisierung der Nachfahren im Tiefensuchwald eines gegebenen Knotens.

4.19 Kor. Der Knoten v ist in einem Tiefensuchwald eines Digraphen genau dann ein echter Nachfahre eines Knotens u , wenn

$$\text{GRAU}[u] < \text{GRAU}[v] < \text{SCHWARZ}[v] < \text{SCHWARZ}[u].$$

4.20. Eine alternative Charakterisierung der Nachfahreneigenschaft in Tiefensuchwäldern, die allerdings über die Farben der Knoten während der Suche geht, wird uns später bei den Anwendungen der Tiefensuche nützlich sein.

4.21 Thm (Theorem der weißen Pfade). *Der Knoten v ist in einem Tiefensuchwald eines Digraphen $D = (V, A)$ genau dann ein Nachfahre eines Knotens u , wenn es zum Zeitpunkt $\text{GRAU}[u]$, zu dem die Durchmusterung den Knoten u entdeckt hat, einen Pfad von u nach v gibt, der, bis auf den Knoten u , nur aus weißen Knoten besteht.*

Beweis. Komplett analog zum Beweis von Theorem 4.13(b). □

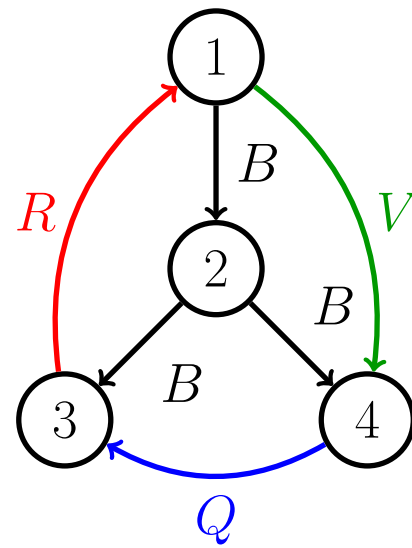
4.22 Def. Die Kanten $(u, v) \in A$ die bei einer Tiefensuche auf $D = (V, A)$ sondiert werden, werden in Abhängigkeit davon, welche Farbe v beim Sondieren von (u, v) hat und in welchem Zusammenhang $\text{GRAU}[u]$ und $\text{GRAU}[v]$ stehen, in die folgenden Arten unterteilt:

Art von $(u, v) \in A$	Bedingung beim Sondieren
Baumkante	$\text{FARBE}[v] = \text{WEISS}$
Rückwärtskante	$\text{FARBE}[v] = \text{GRAU}$
Vorwärtskante	$\text{FARBE}[v] = \text{SCHWARZ}, \text{GRAU}[u] < \text{GRAU}[v]$
Querkante	$\text{FARBE}[v] = \text{SCHWARZ}, \text{GRAU}[u] > \text{GRAU}[v]$

Baumkanten sind die Kanten des Tiefensuchwals D_π .

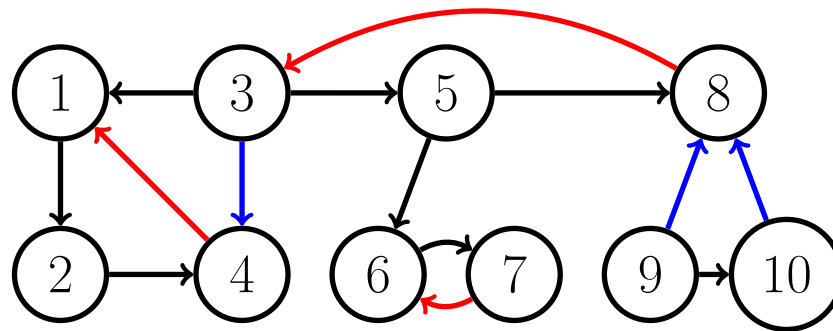
4.23 Bsp. Im Fall der Adjazensliste $1 : [2, 4]$ $2 : [3, 4]$ $3 : [1]$ $4 : [3]$

wird während der Ausführung von `TIEFENSUCHE(1)` die folgende Unterteilung der Kanten in die vier Arten festgelegt:



4.24 Prop. Wird die Unterteilung der Kanten in die vier Arten Baum-, Rückwärts- Vorwärts- und Querkante im Rahmen der vollständigen Tiefensuche durchgeführt, so ist jede Kante, welche zwei Bäume des Tiefensuchbaums verbindet eine Querkante.

4.25 Bsp. Wir schauen uns wiederum die Tiefensuche aus Beispiel 4.8 an und, basierend auf den bereits erhobenen Daten, stellen wir die Klassifikation der Kanten des bearbeiteten Digraphen farbkodiert in folgender Abbildung dar. Dabei sind Baumkanten schwarz, Rückwärtskanten rot, Vorwärtskanten blau und Querkanten grün markiert.



4.26 Thm. *Bei einer Tiefensuche auf einem ungerichteten Graphen G ist jede Kante entweder eine Baumkante oder eine Rückwärtskante.*

Beweis. Sei (u, v) eine beliebige Kante von G und seien die Knoten so bezeichnet, dass $\text{GRAU}[u] < \text{GRAU}[v]$ gilt. Die Tiefensuche muss daher den Knoten v entdeckt und fertig abgearbeitet haben, bevor u fertig abgearbeitet ist. In der Zwischenzeit ist der Knoten u stets grau. Falls die Durchmusterung die Kante zuerst in der Richtung von u nach v sondiert, so ist v bis dahin unentdeckt (also weiß) gewesen, da die Kante sonst bereits in Gegenrichtung sondiert worden wäre. Mit anderen Worten, die Kante (u, v) ist eine Baumkante.

Falls nun die Durchmusterung die Kante zuerst in der Richtung von v nach u sondiert, so ist die Kante (u, v) eine Rückwärtskante, da u zu dem Zeitpunkt der Sondierung noch grau ist. □

4.27. Analog zur diskutierten Breitensuche kann man eine *Tiefensuche* auch ohne Rekursion umsetzen. Dies hat praktische Vorteile, weil der Programmstack nicht belastet wird. Dazu ersetzt man die Warteschlange Q durch einen sogenannten *Stack* (= *Stapel*). Für die Tiefensuche kann ein Stack auf der Basis von einem Array umgesetzt werden.

Hier eine ziemlich direkte Konvertierung der rekursiven Umsetzung. Wir gehen von einem zugrundeliegenden Stack S aus. Als $\text{TOP}(S)$ bezeichnen wir das oberste Element des Stacks. Durch $\text{POP}(S)$ erfolgt die Entfernung und Rückgabe des obersten Elements. Durch $\text{PUSH}(S, u)$ wird ein neues Element u auf den Stack gelegt. Die Elemente $N[u]$ indexieren wir mit Zahlen 0 bis $\deg(u) - 1$, wobei hier $\deg(u)$ der Grad des Knotens u ist. Wir führen ein Array IND ein, in dem durch $\text{IND}[u]$ notiert wird, dass beim sondieren der Nachbarn von $u \in V$ der Knoten $v = N[u][\text{IND}[u]]$ als nächster dran ist. Ist $v = \deg(u)$, so hat man alle Nachbarn von u sondiert. Wir setzen am Anfang $\text{IND}[u] = 0$ für alle $u \in V$, färben den Startknoten s grau und legen s auf S . Auf diese Weise lassen sich mit Hilfe von S und IND die gerade laufenden rekursiven

Aufrufe der Tiefensuche simulieren:

TIEFENSUCHE-MIT-STACK(s)

- 1: Stack S für höchstens $|V|$ Elemente anlegen
- 2: FARBE[s] = GRAU
- 3: PUSH(S, s)
- 4: Liste IND mit IND[u] = 0 für alle $u \in V$ anlegen
- 5: **while** S nicht leer :
- 6: $u := \text{TOP}(S)$ ▷ *Suche für u läuft weiter*
- 7: **if** IND[u] = deg(u) :
- 8: POP(S) ▷ *Suche für u wird beendet*
- 9: FARBE[u] := SCHWARZ
- 10: **else:**
- 11: $v = N[u][\text{IND}[u]]$ ▷ *Sondierung der Nachbarn von u wird fortgesetzt*
- 12: IND[u] := IND[u] + 1
- 13: **if** FARBE[v] = WEISS :
- 14: FARBE[v] = GRAU ▷ *Neuer Knoten ist entdeckt*
- 15: PUSH(S, v) ▷ *Suche für v wird gestartet*

Die weiteren Daten, die man im Rahmen der rekursiven Tiefensuche berechnen kann, kann man auch in der obigen iterativen Version an den entsprechenden Stellen berechnen.

4.28. Umsetzung:

```
def dfs_init(G):  
    color=['white' for u in G.vertices()]  
    pred=[None for u in G.vertices()]  
    return color, pred  
  
def dfs(G, s, color, pred):  
    color[s]='grey'  
    for v in G.neighbors():  
        if color[v]=='white':  
            pred[v]=s  
            dfs(G, v, color, pred)  
    color[s]='black'  
  
def dfs_with_stack(G, s, color, pred):  
    S=[s]  
    color[s]='grey'  
    ind={s:0}
```

```
while len(S)>0:
    u=S[-1]
    if ind[u]==G.degree(u):
        S.pop()
        color[u]='black'
    else:
        v=G.neighbors(u)[ind[u]]
        ind[u]+=1
        if color[v]=='white':
            ind[v]=0
            color[v]='grey'
            S.append(v)
```

4.1 Anwendung I – Topologisches Sortieren

4.29 Def. Sei $D = (V, A)$ ein Digraph mit n Knoten. Eine *topologische Sortierung* von D ist eine Anordnung v_1, \dots, v_n seiner Knoten, so dass $i < j$ gilt, falls $(v_i, v_j) \in A$ eine Kante in D ist. Das heißt, der Startknoten einer jeden Kante kommt in der Anordnung vor dem Endknoten.

4.30. Man kann die topologische Sortierung auch als horizontale Anordnung der Knoten von D auffassen, so dass jede Kante von links nach rechts zeigt.

Diese Veranschaulichung zeigt, dass es nicht möglich ist, einen Digraphen topologisch zu sortieren, wenn er einen Zyklus enthält. Im Folgenden werden wir sehen, wie man mit einer Tiefensuche jeden *azyklischen* Digraphen topologisch sortieren kann. Als Konsequenz ergibt sich:

4.31 Prop. Ein Digraph besitzt genau dann eine topologische Sortierung, wenn er azyklisch ist.

4.32. Eine Anwendung des topologischen Sortierens ist **makefile**. Die Kanten des Diagraphen sind durch die Paare target-prerequisite gegeben. Die targets sollen in einer topologisch sortierten Reihenfolge abgearbeitet werden.

4.33. Das topologische Sortieren kann also als Methode verstanden werden, um zu testen ob es Zyklen im Eingabegraphen gibt. Eine andere Anwendung ist das Sequenzieren von Aufträgen (engl. *scheduling*). Zum Beispiel hilft die topologische Sortierung zu entscheiden, in welcher Reihenfolge einzelne Teilprojekte in einem großen Programmierprojekt kompiliert werden sollten.

4.34 Thm. *Nach der Ausführung der vollständigen Tiefensuche auf einem azyklischen Digraphen $D = (V, A)$ ist die Anordnung der Knoten $u \in V$ in der absteigenden Reihenfolge nach $\text{SCHWARZ}[u]$ eine topologische Sortierung. Diese Anordnung kann während der Tiefensuche in der Zeit $\Theta(|V| + |A|)$ berechnet werden.*

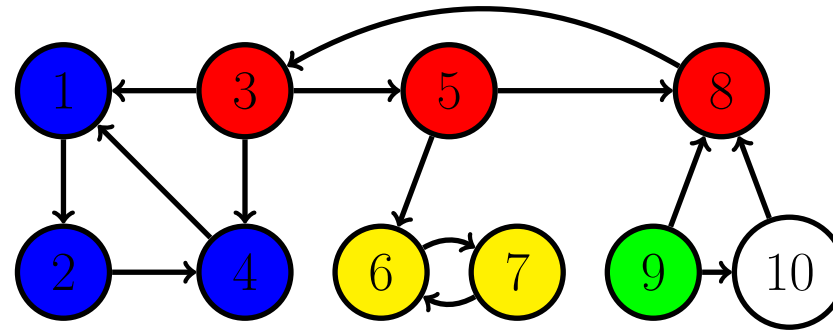
Beweis. Wir betrachten eine Kante (u, v) . Ist v vor u entdeckt worden, so wird während der Ausführung von $TS(u)$ der Knoten u nicht entdeckt, denn sonst gäbe es einen (v, u) -Pfad und somit auch einen Zyklus in D . Das bedeutet, dass in diesem Fall $TS(v)$ vor $TS(u)$ terminiert. Es gilt also $\text{SCHWARZ}[v] \leq \text{SCHWARZ}[u]$.

Wird u vor v entdeckt, so wird v während der Ausführung von $TS(u)$ spätestens beim Sondieren von (u, v) entdeckt. In diesem Fall terminiert $TS(v)$ ebenfalls vor $TS(u)$. Somit gilt auch in diesem Fall $\text{SCHWARZ}[v] \leq \text{SCHWARZ}[u]$. □

4.2 Anwendung II – Starke Zusammenhangskomponenten

4.35 Def. Eine inklusionsmaximale Teilmenge $K \subseteq V$ der Knoten eines gegebenen Digraphen $D = (V, A)$ heißt *starke Zusammenhangskomponente*, wenn es für jedes Paar von Knoten $u, v \in K$ sowohl einen (u, v) -Pfad als auch einen (v, u) -Pfad in D gibt. Das heißt, die Knoten u und v sind vom jeweils anderen aus erreichbar.

4.36 Bsp. Die starken Zusammenhangskomponenten des Digraphen aus Beispiel 4.8 bestehen aus den Knoten gleicher Farbe in folgender Abbildung:



4.37. Als weitere wichtige Anwendung der Tiefensuche zeigen wir hier, wie man mit ihrer Hilfe einen Digraphen in dessen starken Zusammenhangskomponenten zerlegen kann, das heißt, wir suchen eine Partition $V = K_1 \cup K_2 \cup \dots \cup K_r$ der Knotenmenge von D , so dass jede Teilmenge $K_i \subseteq V$ eine starke Zusammenhangskomponente ist. Eine solche Zerlegung liegt vielen Algorithmen auf Digraphen zugrunde, da sie einen Ansatz mittels des Schemas Teile-und-Beherrsche erlaubt: Nach der erfolgten Zerlegung des Digraphen arbeitet der jeweilige Algorithmus auf den einzelnen Komponenten separat und vereinigt dann die Lösungen entsprechend der Verbindungsstruktur der Komponenten untereinander.

Der Algorithmus, den wir hier untersuchen wollen, basiert darauf eine Tiefensuche auf dem gegebenen Digraphen $D = (V, A)$ und danach eine weitere auf seinem *transponierten Graphen* $D^T = (V, A^T)$ zu machen, dessen Kantenmenge durch $A^T = \{(u, v) \in V \times V : (v, u) \in A\}$ definiert ist. Wir erhalten also D^T aus D indem wir die Orientierung aller Kanten von D umdrehen.

Ein Knoten v ist in D genau dann von einem anderen Knoten u aus erreichbar, wenn u in D^T von v aus erreichbar ist. Es gilt weiterhin die folgende wichtige Eigenschaft:

4.38 Prop. Der zu einem Digraphen $D = (V, A)$ transponierte Graph D^T hat dieselben starken Zusammenhangskomponenten wie D .

4.39. Der konkrete Algorithmus zum Bestimmen der starken Zusammenhangskomponenten ist im folgenden Pseudocode angegeben. Beachten Sie die Modifikation der Reihenfolge, in der die Knoten bei der zweiten Tiefensuche durchlaufen werden.

STARKE-ZUSAMMENHANGSKOMPONENTEN(D)

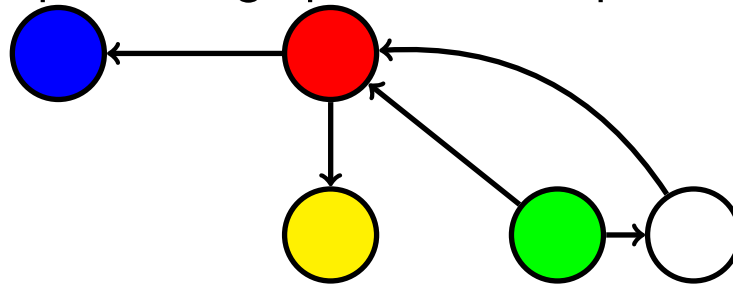
- 1: VOLLSTÄNDIGE-TIEFENSUCHE(D)
 - 2: berechne D^T
 - 3: VOLLSTÄNDIGE-TIEFENSUCHE(D^T) mit folgender Modifikation:
 - 4: Durchlaufe die Hauptschleife (Zeilen 2–6) der Tiefensuche auf D^T in absteigender Reihenfolge des Arrays SCHWARZ von D .
 - 5: gib die Knoten eines jeden Baumes im Tiefensuchwald $(D^T)_\pi$ als starke Zusammenhangskomponenten aus
-

4.40 Thm. *Ist ein Digraph $D = (V, A)$ als Adjazenzliste gegeben, so hat der Algorithmus $\text{STARKE-ZUSAMMENHANGSKOMPONENTEN}(D)$ lineare Laufzeit, das heißt, er benötigt $\Theta(|V| + |A|)$ Zeiteinheiten.*

Beweis. Die Tiefensuche in Zeile 1 hat nach Theorem 4.10 lineare Laufzeit. Die Berechnung des transponierten Graphen D^T kann auch in linearer Laufzeit geschehen, da D als Adjazenzliste gegeben ist (siehe Übungsblatt 7, Aufgabe (3)). Die modifizierte Tiefensuche auf D^T in den Zeilen 3-4 läuft wieder in Zeit $\Theta(|V| + |A|)$. Ebenso ist die Ausgabe der Bäume im Tiefensuchwald $(D^T)_\pi$ in linearer Laufzeit möglich. \square

4.41. Die Idee hinter dem Algorithmus STARKE-ZUSAMMENHANGSKOMPONENTEN(D) beruht auf dem Konzept des *Komponentengraphen* $D^K = (V^K, A^K)$ von $D = (V, A)$, der wie folgt definiert ist. Seien K_1, \dots, K_r die starken Zusammenhangskomponenten von D . Dann setzen wir $V^K := \{v_1, \dots, v_r\}$, also je ein Knoten v_i für je eine starke Zusammenhangskomponente K_i . Es gibt eine Kante $(v_i, v_j) \in A^K$ genau dann, wenn es ein $u \in K_i$ und ein $w \in K_j$ gibt, so dass $(u, w) \in A$ eine Kante in D ist. Der Komponentengraph D^K entsteht also aus D indem wir alle Kanten kontrahieren, deren Start- und Endknoten zur selben starken Zusammenhangskomponente gehören.

4.42 Bsp. Der Komponentengraph des Digraphen aus Beispiel 4.36 in derselben Farbkodierung:



4.43. Die wesentliche Eigenschaft des Komponentengraphen ist, dass er einen azyklischen Digraphen darstellt:

4.44 Lem. Sei $D = (V, A)$ ein Digraph und seien K, K' zwei verschiedene starke Zusammenhangskomponenten von D . Seien weiterhin $u, v \in K$ und $u', v' \in K'$ Knoten der jeweiligen Komponente. Falls D einen (u, u') -Pfad enthält, so kann er nicht gleichzeitig auch einen (v', v) -Pfad enthalten.

Beweis. Angenommen, D würde neben einem (u, u') -Pfad auch einen (v', v) -Pfad enthalten. Da K und K' starke Zusammenhangskomponenten sind, könnten wir den (u, u') -Pfad zu einem (u, v') -Pfad und ebenso den (v', v) -Pfad zu einem (v', u) -Pfad erweitern. Demnach wären die Knoten u und v' voneinander jeweils andersherum erreichbar, was der Annahme widerspricht, dass K und K' verschieden sind. \square

4.45. Für den Korrektheitsbeweis des angegebenen Algorithmus benötigen wir etwas mehr Verständnis der Beziehungen zwischen den während der Laufzeit erzeugten Zeitstempeln. Wenn wir nachfolgend von Zeitpunkten der Entdeckung und Abarbeitung sprechen, dann beziehen wir uns stets auf die entsprechenden Zeitstempel der ersten Tiefensuche in Zeile 1 von STARKE-ZUSAMM

Für die Formulierung der Aussage erweitern wir zunächst die Begriffe der Entdeckungs- und Abarbeitungszeitpunkte von einzelnen Knoten auf Teilmengen von Knoten: Für $U \subseteq V$ seien dazu

$$\text{GRAU}(U) := \min_{u \in U} \text{GRAU}[u] \quad \text{und} \quad \text{SCHWARZ}(U) := \max_{u \in U} \text{SCHWARZ}[u]$$

der früheste Entdeckungszeitpunkt beziehungsweise der späteste Abarbeitungszeitpunkt eines Knotens aus U .

4.46 Lem. Sei $D = (V, A)$ ein Digraph und seien K, K' zwei verschiedene starke Zusammenhangskomponenten von D .

(i) Falls eine Kante $(u, v) \in A$ mit $u \in K$ und $v \in K'$ existiert, so gilt

$$\text{SCHWARZ}(K) > \text{SCHWARZ}(K').$$

(ii) Falls eine Kante $(u, v) \in A^T$ mit $u \in K$ und $v \in K'$ existiert, so gilt

$$\text{SCHWARZ}(K) < \text{SCHWARZ}(K').$$

Beweis. i): Wir unterscheiden Fälle danach welche der beiden starken Zusammenhangskomponenten zuerst entdeckt wird.

Sei zuerst angenommen, dass $\text{GRAU}(K) < \text{GRAU}(K')$ gilt und sei x der erste entdeckte

Knoten in K . Zum Zeitpunkt $\text{GRAU}[x]$ sind alle anderen Knoten in K und alle Knoten in K' weiß. In diesem Moment enthält D also Pfade von x zu jedem anderen Knoten in K , die, bis auf x selbst, nur aus weißen Knoten bestehen. Die Existenz der Kante (u, v) von der Komponente K in die Komponente K' impliziert damit, dass ebenso Pfade von x zu jedem Knoten in K' bestehen, die, bis auf x selbst, nur aus weißen Knoten bestehen. Nach dem Theorem 4.21 der weißen Pfade sind also alle Knoten in $K \cup K'$ Nachfahren von x im Tiefensuchwald und nach Korollar 4.19 erhalten wir $\text{SCHWARZ}(K) = \text{SCHWARZ}[x] > \text{SCHWARZ}(K')$.

Sei nun angenommen, dass $\text{GRAU}(K) > \text{GRAU}(K')$ gilt und sei y der erste entdeckte Knoten in K' . Analog zu oben sind zum Zeitpunkt $\text{GRAU}[y]$ alle Knoten in K' mit einem Pfad von y aus erreichbar, der, bis auf y selbst, nur aus weißen Knoten besteht. Wegen Theorem 4.21 sind damit alle Knoten aus K' Nachfahren von y im Tiefensuchwald und mit Korollar 4.19 gilt $\text{SCHWARZ}[y] = \text{SCHWARZ}(K')$. Nach Annahme sind zum Zeitpunkt $\text{GRAU}[y]$ alle Knoten in K weiß. Da die Kante (u, v) von K nach K' existiert, folgt aus Lemma 4.44, dass es keinen

Pfad von einem Knoten in K' zu einem Knoten in K geben kann. Da damit kein Knoten in K von y aus erreichbar ist, ist die ganze Komponente K zum Zeitpunkt $\text{SCHWARZ}[y]$ noch immer weiß. Als Konsequenz erhalten wir $\text{SCHWARZ}[w] > \text{SCHWARZ}[y]$, für jeden Knoten $w \in K$, und damit $\text{SCHWARZ}(K) > \text{SCHWARZ}(K')$.

ii): Nach Definition von A^T ist (v, u) Kante von D . Da nach Beobachtung 4.38 die Teilmen-
gen K und K' starke Zusammenhangskomponenten von D^T sind, können wir Teil i) anwenden
und erhalten $\text{SCHWARZ}(K) < \text{SCHWARZ}(K')$. □

4.47. In Worten ausgedrückt besagt das vorige Lemma, dass jede Kante von D (bzw. D^T), die zwei verschiedene starke Zusammenhangskomponenten verbindet, von der Komponente mit dem späteren (bzw. früheren) Abarbeitungszeitpunkt ausgeht.

Auf der Grundlage dieser Beobachtung können wir nun die Grundidee des Korrektheitsbeweises erläutern: Die Tiefensuche auf D^T startet in der starken Zusammenhangskomponente K_1 , die den zuletzt abgearbeiteten Knoten u_1 der Tiefensuche auf D enthält. Der transponierte Graph D^T enthält nach Lemma 4.46 (ii) keine Kanten, die von K_1 zu einer anderen starken Zusammenhangskomponente verlaufen. Daher werden bei der Tiefensuche die von u_1 aus startet genau die Knoten aus der Komponente K_1 besucht. Der Algorithmus wählt danach den Knoten u_2 außerhalb der Komponente K_1 , der als letztes unter den verbleibenden Knoten bei der Tiefensuche auf D abgearbeitet wurde und besucht wie zuvor, alle Knoten der starken Zusammenhangskomponente K_2 , die u_2 enthält. Dies geht iterativ weiter bis alle Tiefensuchbäume erstellt sind, und wir sehen, dass diese den starken Zusammenhangskomponenten entsprechen.

4.48 Thm. $\text{STARKE-ZUSAMMENHANGSKOMPONENTEN}(D)$ *bestimmt die starken Zusammenhangskomponenten eines Digraphen $D = (V, A)$ korrekt.*

Beweis. Wir argumentieren über vollständige Induktion nach der Anzahl der Tiefensuchbäume, die bei der Tiefensuche auf D^T in den Zeilen 3-4 des Algorithmus $\text{STARKE-ZUSAMMENHANGSKOMPONENTEN}$ gefunden werden und zeigen, dass jeder solche Baum eine starke Zusammenhangskomponente bildet. Die konkrete Aussage, die wir per Induktion beweisen, ist: „Die ersten k Tiefensuchbäume, die in den Zeilen 3-4 erzeugt werden, sind starke Zusammenhangskomponenten von D .“

Der Induktionsanfang ist mit $k = 0$ klar. Für den Induktionsschritt, sei $B \subseteq V$ der $(k+1)$ -te erzeugte Baum bei der Tiefensuche auf D^T . Sei weiterhin $u \in B$ die Wurzel dieses Baumes und $K \subseteq V$ die starke Zusammenhangskomponente, die u enthält. Für jede starke Zusammenhangskomponente $K' \neq K$, die bereits besucht wurde, gilt $\text{SCHWARZ}[u] = \text{SCHWARZ}(K) > \text{SCHWARZ}(K')$, wegen der Reihenfolge in der die Knoten in Zeile 4 durchlaufen werden. Wegen

der Induktionsannahme sind zum Zeitpunkt zu dem die Suche den Knoten u besucht, alle anderen Knoten von K weiß. Nach dem Theorem 4.21 der weißen Pfade sind daher alle anderen Knoten in K Nachfahren von u in dessen Tiefensuchbaum.

Weiterhin müssen nach Induktionsannahme und Lemma 4.46 (ii) alle Kanten von D^T , die aus K herausführen, zu starken Zusammenhangskomponenten verlaufen, die bereits entdeckt wurden. Damit wird kein Knoten außerhalb von K ein Nachfahre von u bei der Tiefensuche auf D^T . Das heißt, dass die Knoten des Tiefensuchbaumes von D^T , der von u ausgeht, eine starke Zusammenhangskomponente bilden, und der Induktionsschritt ist gezeigt. \square

4.49. Die Präsentation der Tiefensuche basiert auf [CLRS17].

5 Exkurs in Heap-basierte Prioritätsschlangen

IM AUFBAU

6 Der Prim-Algorithmus

6.1. In diesem Abschnitt sei $G = (V, E, w)$, mit Gewichtsfunktion $w : E \rightarrow \mathbb{R}$, stets ein gewichteter aber ungerichteter Graph, der weiterhin als *zusammenhängend* angenommen wird.

6.2 Def. Ein Baum $T = (V', E')$ heißt *Spannbaum* von G , falls $V = V'$ und $E' \subseteq E$ gelten. Für jeden Teilgraphen $G' = (V', E')$ von G definieren wir sein *Gewicht* als

$$w(G') := \sum_{e \in E'} w(e).$$

Ein Spannbaum $T = (V, E')$ heißt *minimaler Spannbaum* von G , wenn T unter allen Spannbäumen von G minimales Gewicht hat.

6.3. Unser Ziel ist es, das *Problem des minimalen Spannbaums* möglichst effizient zu lösen. Das heißt, gegeben einen zusammenhängenden gewichteten Graphen G , finde einen minimalen Spannbaum von G .

Das Bestimmen eines *minimalen* Spannbaums ist in verschiedenen Problemen in der Praxis relevant:

- Bei elektronischen Schaltkreisen will man eine Menge von Kontakten verdrahten, um diese auf das gleiche Potenzial zu legen.
- *Spanning tree protocol* (Übersendung von Paketen in einem Netzwerk).
- Approximationslösung für das Problem des *Handlungsreisenden*.

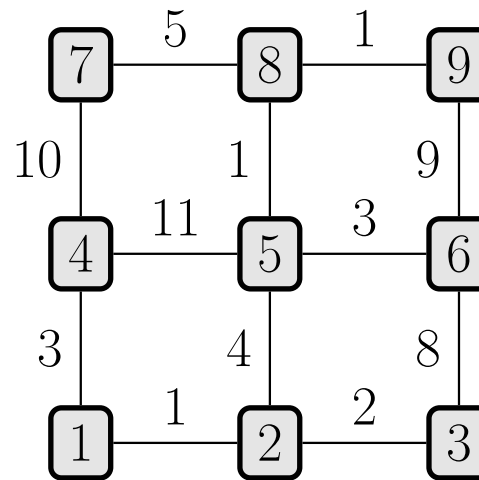
Wir legen Ihnen nahe, sich mit diesen Anwendungen selbst etwas vertraut zu machen und nachzuvollziehen, inwiefern das Finden eines minimalen Spannbaums in jedem dieser Beispiele von Nutzen ist.

6.4. Sind alle Kantengewichte positiv, so ist das Problem des minimalen Spannbaums äquivalent zum Problem der Bestimmung eines zusammenhängenden Teilgraphen $G' = (V', E')$ von G mit $V = V'$ und dem kleinsten Gewicht.

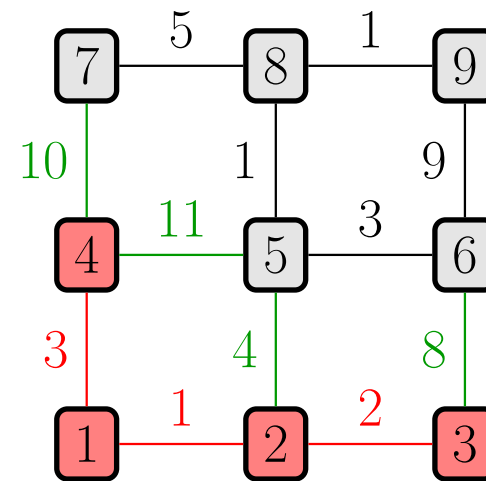
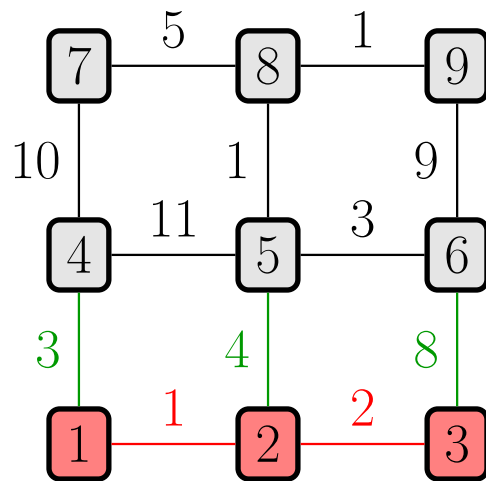
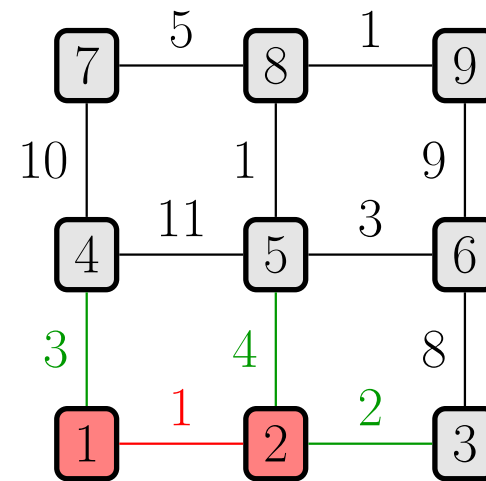
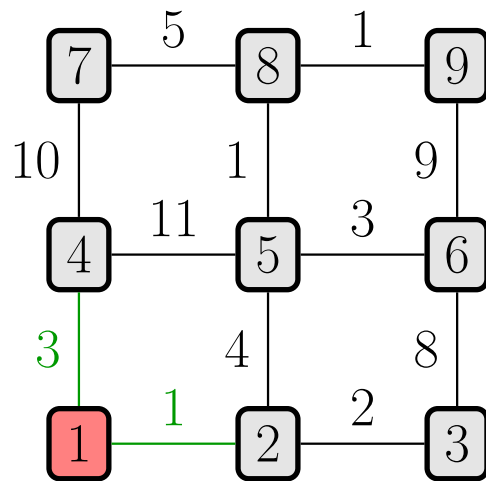
6.5. In der Literatur gibt es eine ganze Reihe von Algorithmen zur Lösung des Problems des minimalen Spannbaums. Wir diskutieren hier den sogenannten *Prim-Algorithmus* der im Jahr 1957 in einer Arbeit von Robert C. Prim erschien und zwei Jahre später von Dijkstra neu beschrieben wurde. Beiden Autoren (und allgemein der westlichen Forschungsgemeinschaft) war anscheinend die Arbeit von Vojtěch Jarník aus dem Jahr 1930 nicht bekannt, die die Vorgehensweise bereits lange zuvor diskutierte.

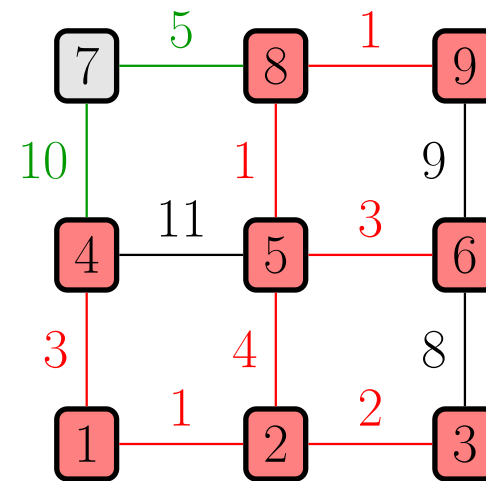
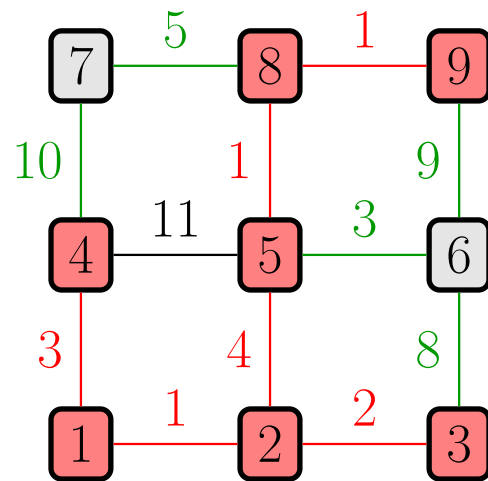
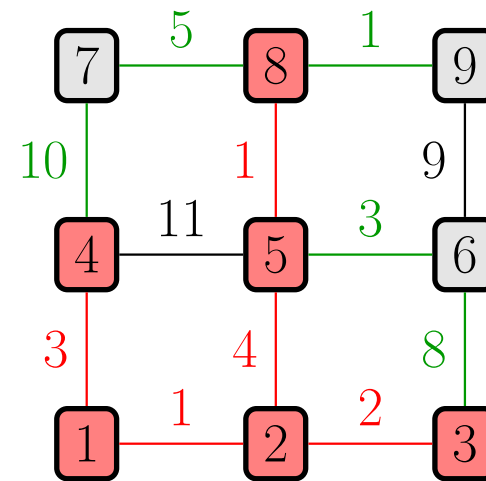
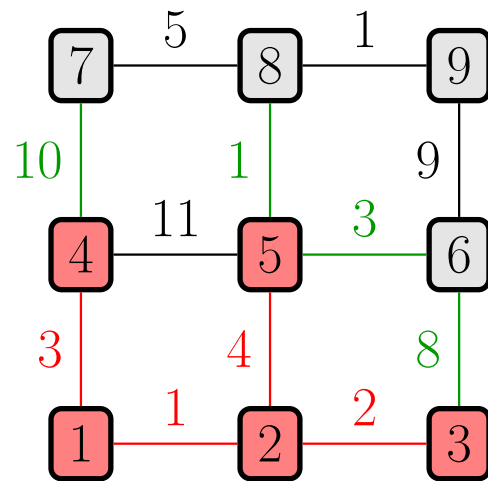
6.6. Der Prim-Algorithmus basiert auf der einfachen Idee einen Spannbaum sukzessive „von Null auf“ durch das Hinzufügen von Kanten kleinstmöglichen Gewichtes wachsen zu lassen. Genauer gesagt starten wir mit einem Startbaum T_0 , der aus einem beliebigen Knoten s von G besteht. Diesen erweitern wir dann durch eine zu s adjazente Kante kleinsten Gewichtes zu dem Teilbaum T_1 . Im nächsten Schritt wählen wir wieder eine Kante kleinsten Gewichtes unter den verbleibenden Kanten in G aus, die zu T_1 hinzugefügt werden kann und dabei die Baumeigenschaft erhält. Wir bekommen dadurch den Teilbaum T_2 , der nun aus zwei Kanten besteht. Führen wir dies Schritt für Schritt weiter gelangen wir zu einem Teilbaum $T_{|V|-1}$ der aus $|V| - 1$ Kanten besteht und ein aufspannender Baum von G ist. Der Clou ist nun, dass dieser aufspannende Baum minimales Gewicht unter allen Spannbäumen von G hat.

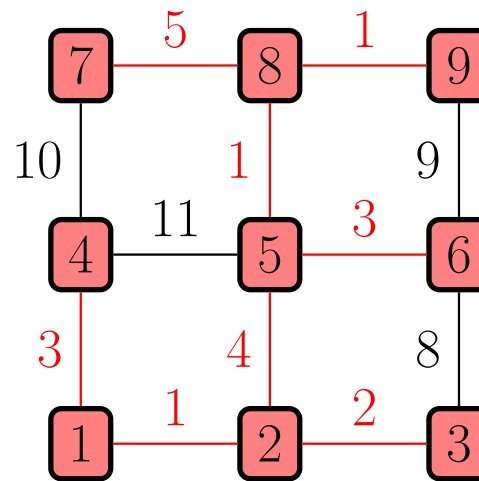
6.7 Bsp. Gegeben sei der folgende gewichtete zusammenhängende Graph:



Beginnend vom Startknoten $s = 1$ illustrieren wir die Vorgehensweise des Prim-Algorithmus. In jedem Graphen der folgenden Sequenz markieren wir den aktuellen Teilbaum T_i in rot und die in Frage kommenden Kanten für die Erweiterung zum Teilbaum T_{i+1} in grün:







6.8. Die Umsetzung dieser simplen Idee erfolgt nun über den im nachfolgenden Pseudocode gegebenen Algorithmus. Darin ist die Vorgängerabbildung wieder durch das Attribut $\pi[v]$ eines jeden Knotens $v \in V$ beschrieben. Der aktuelle Teilbaum, der nach obiger Idee sukzessive zu einem minimalen Spannbaum erweitert wird, hat zu jedem Zeitpunkt der Ausführung die Knotenmenge $V' = V \setminus Q$ und die Kantenmenge $E' = \{\{\pi[v], v\} : v \in V' \setminus \{s\}\}$. Für alle Knoten $v \in Q$ mit $\pi[v] \neq \text{NIL}$ ist das Attribut $\ell[v] < \infty$ und entspricht dem Gewicht der Kante $\{\pi[v], v\}$. Diese Kante ist eine Kante mit aktuell kleinstem Gewicht, die den Knoten v mit dem aktuellen Teilbaum (V', E') verbindet.

PRIM()

- 1: Ein beliebiges $s \in V$ fixieren.
 - 2: \triangleright *Initialisierung*
 - 3: **for** $v \in V$:
 - 4: $\ell[v] := \infty$
 - 5: $\pi[v] := \text{NIL}$
 - 6: **end**
 - 7: $\ell[s] := 0$
 - 8: $Q :=$ Prioritätswarteschlange mit ℓ als Schlüssel, die alle Knoten $v \in V$ enthält.
 - 9: \triangleright *Iteratives Aktualisieren der Kanten*
 - 10: **while** $Q \neq \emptyset$:
 - 11: $u := \text{MINIMUM-ENTFERNEN}(Q)$
 - 12: **for** $v \in N[u]$:
 - 13: PRIM-UPDATE(u, v)
 - 14: **end**
 - 15: **end**
-

Die Hilfsprozedur $\text{PRIM-UPDATE}(u, v)$ ist wie folgt umgesetzt:

 $\text{PRIM-UPDATE}(u, v)$

- 1: **if** $v \in Q$ und $w(u, v) < \ell[v]$:
 - 2: $\ell[v] := w(u, v)$
 - 3: $\pi[v] := u$
 - 4: **end**
-

Damit in dieser Hilfsprozedur der Test „Ist $v \in Q$?“ schnell, das heißt, in konstanter Zeit, durchgeführt werden kann, verwalten wir ein Array, das für jeden Knoten $v \in V$ notiert, ob der Knoten aktuell in Q enthalten ist oder nicht. So ein Array entspricht dem Array mit Farbattributen in der Breiten- und der Tiefensuche.

6.9 Aufg. Verfolgen Sie die Arbeitsweise von $\text{PRIM}()$ anhand der Dynamiken der involvierten Objekte Q , ℓ und π auf dem gewichteten Graphen in Beispiel 6.7.

6.10. Nach erfolgter Ausführung des Prim-Algorithmus entspricht der letzte aktuelle Teilbaum (V', E') dem *Vorgängerteilgraphen* $G_\pi := (V, E_\pi)$ von G mit Kantenmenge $E_\pi := \{\{\pi(v), v\} : v \in V'\}$. Wie oben bereits erwähnt stellt sich heraus, dass dieser Vorgängerteilgraph tatsächlich ein minimaler Spannbaum von G ist:

6.11 Thm. *Sei $G = (V, E, w)$ ein zusammenhängender gewichteter Graph, der durch eine Adjazenzliste gegeben ist. Der Algorithmus $\text{PRIM}()$ löst das Problem des minimalen Spannbau-
baums korrekt, das heißt, der Vorgängerteilgraph G_π ist ein minimaler Spannbaum von G .*

*Ist die Prioritätswarteschlange auf der Basis von Heaps umgesetzt, so beträgt die Laufzeit
des Verfahrens $O(|E| \log |V|)$.*

Beweis. Schauen wir zuerst auf die Laufzeit von $\text{PRIM}()$. Die Initialisierung verläuft in Zeit $\Theta(|V|)$. Beachten Sie dabei lediglich, dass der Min-Heap für die Prioritätswarteschlange Q in linearer Zeit initialisiert werden kann, da die Schlüssel für die Knoten $v \neq s$ alle gleich ∞ sind. Die Initialisierung kann also durch ein Array $[s, v_1, \dots, v_k]$ erfolgen, wobei v_1, \dots, v_k eine beliebige Anordnung der Knoten in $V \setminus \{s\}$ ist.

Nach erfolgter Initialisierung wird die `while`-Schleife exakt $|V|$ Mal aufgerufen, ein Mal für jeden Knoten von G . Die Prozedur $\text{MINIMUM-ENTFERNEN}(Q)$ benötigt bei der Um-

setzung mittels Heaps, höchstens $O(\log |V|)$ Zeiteinheiten, und damit zusammengefasst über die ganze Laufzeit von $\text{PRIM}()$ höchstens $O(|V| \log |V|)$ Zeit. Die `for`-Schleife wird über den gesamten Algorithmus $O(|E|)$ oft ausgeführt, da jede Kante genau 2 Knoten hat. Die Hilfsprozedur $\text{PRIM-UPDATE}(u, v)$ hat den Anschein in konstanter Zeit zu arbeiten. Beachten Sie aber, dass nach der Ausführung der Zuweisung $\ell[v] := w(u, v)$ der Schlüssel von v in der Prioritätswarteschlange Q mittels $\text{SCHLÜSSEL-VERKLEINERN}(Q, v, \ell[v])$ verkleinert werden muss, was $O(\log |V|)$ Zeiteinheiten erfordert. Insgesamt werden für die Rechenschritte in der `for`-Schleife damit $O(|E| \log |V|)$ Zeiteinheiten benötigt.

Die Gesamtlaufzeit von $\text{PRIM}()$ ergibt sich nun durch Addition der oben bestimmten Teillaufzeiten und damit zu $\Theta(|V|) + O(|V| \log |V|) + O(|E| \log |V|) = O(|E| \log |V|)$.

Wir beweisen nun die Korrektheit des Algorithmus. Man überzeugt sich leicht davon, dass die beiden folgenden Bedingungen eine Schleifeninvariante für die `while`-Schleife bilden:

- Der Wert $\ell[v]$ für $v \in Q$ ist das kleinste Gewicht einer Kante die den Knoten v mit einem

Knoten aus $V \setminus Q$ verbindet. (Die Existenz einer verbindenden Kante ist durch $\ell[v] < \infty$ angezeigt.)

- Der Teilgraph $G' = (V', E')$ mit $V' = V \setminus Q$ und $E' = \{\{\pi[v], v\} : v \in V' \setminus \{s\}\}$ ist ein Baum.

Nach der Terminierung des Algorithmus ist der obige Teilgraph gleich dem Vorgängerteilgraphen $G' = G_\pi$ und nach der zweiten Eigenschaft oben ein Spannbaum von G .

Wir zeigen nun, dass G_π minimales Gewicht unter allen Spannbäumen von G hat. Für einen beliebigen Spannbaum S_1 von G wollen wir also $w(G_\pi) \leq w(S_1)$ zeigen. Ist $S_1 = G_\pi$, so ist die Behauptung trivial. Ansonsten, existiert eine Kante die zu G_π aber nicht zu S_1 gehört. Wir betrachten G_π vor dem ersten Moment der Ausführung, in dem eine solche Kante $e \notin S_1$ zu G_π hinzugefügt wurde.

In S_1 existiert ein eindeutiger Pfad p , der einen Knoten im aktuellen Teilgraphen G' mit einem

Knoten außerhalb von G' verbindet (es gibt mindestens einen Pfad, weil S_1 zusammenhängend und aufspannend ist, und es gibt nicht mehr als einen Pfad, weil S_1 ansonsten Zyklen hätte). Dieser Pfad p enthält eine Kante e_1 von S_1 mit einem Endknoten in V' und einem Endknoten außerhalb von V' . Durch das Entfernen der Kante e_1 aus S_1 und das Hinzufügen der Kante e entsteht ein Spannbaum S_2 von G . Nach der Beschreibung des Prim-Algorithmus hat die Kante e das kleinste Gewicht unter allen Kanten, die einen Knoten aus V' mit einem Knoten außerhalb von V' verbinden. Daher gilt $w(e_1) \geq w(e)$ und somit $w(S_2) \leq w(S_1)$. Weiterhin haben am Ende der Ausführung des Algorithmus S_2 und G_π mehr Kanten gemeinsam als S_1 und G_π .

Wiederholen wir dieses Argument iterativ so entsteht eine endliche Folge von Spannbäumen S_1, S_2, \dots, S_k mit $w(S_1) \geq w(S_2) \geq \dots \geq w(S_k)$, in der das letzte Element S_k ein Spannbaum ist, dessen alle Kanten zu G_π gehören. Somit gilt $S_k = G_\pi$ und wir haben $w(S_1) \geq w(G_\pi)$ wie gewünscht bewiesen. □

6.12.

- (a) Benutzt man anstelle eines Min-Heaps einen sogenannten *Fibonacci-Heap* als Grundlage für die Prioritätswarteschlange Q , so verbessert sich die Laufzeit auf $O(|E| + |V| \log |V|)$.
- (b) Ein alternativer Algorithmus zum Bestimmen eines minimalen Spannbaums ist Kruskal's Algorithmus. Dieser verfolgt die Strategie einen Wald sukzessive mit Kanten zu einem Baum zu füllen, und dabei wie bei $\text{PRIM}()$ in jedem Schritt eine zulässige Kante mit kleinstem Gewicht zu wählen. Die Laufzeit ist auch hier $O(|E| \log |V|)$ bei geeigneter Umsetzung der involvierten Datenstrukturen.

Die Korrektheit des Kruskal-Algorithmus ist etwas weniger direkt zu beweisen als die des Prim-Algorithmus, da man sicherstellen muss, dass zum Wald hinzugefügte Kanten keine Kreise erzeugen.

6.13. Die Präsentation des Primalgorithmus hier basiert auf [CLRS17].

7 Färbung

7.1 Def. Für einen Graphen $G = (V, E)$ und eine k -elementige Menge C heißt eine Abbildung $f : V \rightarrow C$ eine **k -Färbung** wenn $f(u) \neq f(v)$ für alle $\{u, v\} \in E$ gilt.

Das minimale k , für welches G eine k -Färbung nennt man die **chromatische Zahl** von G und bezeichnet diesen Wert als $\chi(G)$.

7.2 Prop (Brute-Force-Färbung). Es gibt einen Algorithmus, der für einen (als Adjazenz- oder Kantenliste) gegebenen Graphen $G = (V, E)$ und ein $k \in \mathbb{N}$ in der Zeit T mit $O(k^{|V|} \cdot |V| \cdot |E|)$ entscheidet, ob G eine k -Färbung besitzt.

Beweis. Sei $k \geq 2$, denn der Fall $k = 1$ ist trivial. Es gibt $k^{|V|}$ Abbildungen $V \rightarrow \{0, \dots, k-1\}$, die man alle algorithmisch aufzählen kann. Zur Aufzählung kann man solche Abbildung als Darstellungen der Zahlen aus $\{0, \dots, k^{|V|} - 1\}$ im Stellenwertsystem zur Basis k auffassen und all diese Darstellungen von der Darstellung von 0 beginnend durch das sukzessive Inkrementieren aufzählen. Eine Inkrementierung benötigt $O(|V|)$ Elementaroperationen, sodass man beim Aufzählen auf insgesamt $O(k^{|V|}|V|)$ Elementaroperationen kommt. Für jede Abbildung kann durch das Iterieren über alle Kanten überprüft werden, ob die Abbildung eine k -Färbung ist. \square

7.3. Der Algorithmus aus Proposition 7.2 ist kein Polynomialzeit-Algorithmus: denn im Fall $k \geq 2$ hat die Laufzeit des Algorithmus die Ordnung mindestens $k^{|V|} \geq 2^{|V|}$. Das heißt, es ist kein effizienter Algorithmus.

Im Spezialfall $k = 2$, gibt es aber einen effizienten Algorithmus.

7.4 Prop. Es gibt einen Algorithmus der für einen als Adjazenzliste gegebenen Graphen $G = (V, E)$ in der Zeit $\Theta(|V| + |E|)$, ob G eine 2-Färbung besitzt.

Beweis. Graphen mit 2-Färbung nennt man auch bipartit. Ob ein Graph bipartit ist, kann mit Hilfe der Tiefensuche oder Breitesuche entschieden werden (Aufgabe). □

7.5. Es ist kein effizienter Test der 3-Färbbarkeit bekannt. Man vermutet, es gibt keinen solchen Test. In der Theorie und Anwendungen gibt es Tausende von Rechenproblemen, die zur Entscheidung der 3-Färbbarkeit äquivalent sind, wobei man in diesem Zusammenhang die Äquivalenz in einem bestimmten genau definierbaren Sinn einführt. Das bedeutet: Die 3-Färbbarkeit gehört zu den sogenannten NP -vollständigen Problemen.

Literaturverzeichnis

- [AZ02] Aigner, Ziegler. Das Buch der Beweise. Springer 2002

- [Ber17] Berghammer: Mathematik für Informatiker. Grundlegende Begriffe und Strukturen. Springer Vieweg 2017

- [Ber19] Berghammer: Mathematik für Informatiker. Grundlegende Begriffe und Strukturen und ihre Anwendung. Springer Vieweg 2019

- [Big05] Biggs. Discrete mathematics. Oxford University Press 2005

- [Bri01] Mathematik für Informatiker. Einführung an praktischen Beispielen aus der Welt der Computer. München: Hanser 2001

- [CLRS17] Cormen, T. H., Leiserson, C. E., Rivest, R., Stein, C. Algorithmen-Eine Einführung. De Gruyter Oldenbourg. 2017.

- [GR14] Goebbels, Rethmann. Mathematik für Informatiker: eine aus der Informatik motivierte Einführung mit zahlreichen Anwendungs- und Programmbeispielen. Springer Vieweg 2014

- [GS11] Heinz Peter Gumm, Manfred Sommer. Einführung in die Informatik, Oldenbourg Verlag 2011

- [KK15] Knauer, Knauer. Diskrete und algebraische Strukturen - kurz gefasst. Springer Spektrum 2015.
- [KP09] Kreußler, Pfister. Mathematik für Informatiker: Algebra, Analysis, Diskrete Strukturen. Springer 2009.
- [LLM21] Lehman, Leighton, Meyer. Mathematics for Computer Science. Lecture notes at MIT. <https://courses.csail.mit.edu/6.042/spring18/mcs.pdf>
- [Lov20] Lovász, László. Complexity of Algorithms. Lecture Notes. 2020. <https://web.cs.elte.hu/~kiralym/complexity.pdf>
- [Sch12] Schubert. Mathematik für Informatiker: ausführlich erklärt mit vielen Programmbeispielen und Aufgaben.
- [Ste01] Steger. Diskrete Strukturen 1. Kombinatorik, Graphentheorie, Algebra. Springer 2001

[Tit19] Peter Tittmann. Einführung in die Kombinatorik, 3. Auflage, Springer 2019