Spezifikation und Verifikation
Kapitel 2
CTL Model Checking

Frank Heitmann
heitmann@informatik.uni-hamburg.de

16. Mai 2014

CTL

In der Computation Tree Logic (CTL) ist es möglich über die Pfade in einem Transistionssystem zu argumentieren. Hierzu wird die Logik um Pfadquantoren 'A' und 'E' erweitert. Die Semantik wird dann über unendliche, gerichtete Bäume definiert, die man durch ein "unfolding" des Transitionssystems in einen Erreichbarkeitsbaum erhält.

Transitionssystem

Definition (Transitionssystem)

Ein *labelled transition system* (LTS) ist ein Tupel $TS = (S, s_0, R, L)$mit

- einer endlichen Menge von Zuständen S,
- einem Startzustand $s_0 \in S$,
- einer links-totalen Übergangsrelation $R \subseteq S \times S$ und
- einer labelling function $L : S \to \mathcal{P}(V)$, die jedem Zustand s die Menge der atomaren Formeln $L(s) \subseteq V$ zuweist, die in s gelten.

Linkstotal bedeutet, dass es zu jedem $s \in S$ stets ein s' mit $(s, s') \in R$ gibt.

Transitionssystem

Definition (Pfad im LTS)

Ein *Pfad* π in einem LTS $TS = (S, s_0, R, L)$ ist eine unendliche Sequenz von Zuständen

$$\pi = s_1 s_2 s_3 \dots$$

derart, dass $(s_i, s_{i+1}) \in R$ für alle $i \geq 1$.

- Mit π^i, $i \geq 1$ bezeichnen wir den Suffix, der an s_i startet, d.h. den Pfad $\pi^i = s_i s_{i+1} \dots$
- Mit $\pi(i)$, $i \geq 1$, bezeichnen wir den i-ten Zustand in π, d.h. $\pi(i) = s_i$.
- Wenn s_1 der Startzustand s_0 von TS ist, wird π auch als Rechnung bezeichnet.

CTL: Syntax

Definition (Syntax von CTL)

Die (wohlgeformten) Formeln der Computation Tree Logic (CTL) werden durch die folgende Grammatik definiert:

$$\begin{aligned}\phi \quad ::= \quad & v \mid \neg\phi \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \\ & EX\phi \mid EF\phi \mid EG\phi \mid E[\phi U \phi] \mid \\ & AX\phi \mid AF\phi \mid AG\phi \mid A[\phi U \phi]\end{aligned}$$

wobei $v \in V$ ein aussagenlogisches Atom ist.

Kann man auch mit einer induktiven Definition machen.

CTL: Semantik

Definition (Semantik von CTL (I))

Sei $M = (S, s_0, R, L)$ ein LTS und $s \in S$ ein Zustand. Eine CTL Formel ϕ ist erfüllt in s (in M), wenn $M, s \models \phi$ gilt, wobei die Relation $\models$ induktiv definiert ist:

$M, s \models v$	gdw.	$v \in L(s)$ für $v \in V$
$M, s \models \neg\phi$	gdw.	$M, s \not\models \phi$
$M, s \models \phi_1 \wedge \phi_2$	gdw.	$M, s \models \phi_1$ *und* $M, s \models \phi_2$
$M, s \models \phi_1 \vee \phi_2$	gdw.	$M, s \models \phi_1$ *oder* $M, s \models \phi_2$

CTL: Semantik

Definition (Semantik von CTL (II))

$M, s \models EX\phi$	gdw.	ein Zustand $s' \in S$ existiert mit $(s, s') \in R$ und $M, s' \models \phi$
$M, s \models EF\phi$	gdw.	ein Pfad $\pi = s_1 s_2 \ldots$ beginnend bei s $(s_1 = s)$ existiert und ein $i \geq 1$, so dass $M, s_i \models \phi$ gilt.
$M, s \models EG\phi$	gdw.	ein Pfad $\pi = s_1 s_2 \ldots$ beginnend bei s $(s_1 = s)$ existiert und für alle $i \geq 1$ $M, s_i \models \phi$ gilt.
$M, s \models E[\phi_1 U \phi_2]$	gdw.	ein Pfad $\pi = s_1 s_2 \ldots$ beginnend bei s existiert und ein $j \geq 1$, so dass $M, s_j \models \phi_2$ und $M, s_i \models \phi_1$ für alle $i < j$.

CTL: Semantik

Definition (Semantik von CTL (III))

$M, s \models AX\phi$	gdw.	$M, s' \models \phi$ für alle $s' \in S$ mit $(s, s') \in R$.
$M, s \models AF\phi$	gdw.	für alle Pfade $\pi = s_1 s_2 \ldots$ beginnend bei s ein $i \geq 1$ existiert mit $M, s_i \models \phi$.
$M, s \models AG\phi$	gdw.	für alle Pfade $\pi = s_1 s_2 \ldots$ beginnend bei s $M, s_i \models \phi$ für alle $i \geq 1$ gilt.
$M, s \models A[\phi_1 U \phi_2]$	gdw.	für alle Pfade $\pi = s_1 s_2 \ldots$ beginnend bei s ein $j \geq 1$ existiert derart, dass $M, s_j \models \phi_2$ und $M, s_i \models \phi_1$ für alle $i < j$ gilt

CTL: Semantik

Definition (Semantik von CTL (IV))

Sei $M = (S, s_0, R, L)$ ein LTS und ϕ eine CTL Formel.

- Wenn $M, s_0 \models \phi$ gilt, schreiben wir auch $M \models \phi$ und sagen, dass M ein *Model* für ϕ ist oder dass ϕ *erfüllt ist* in M.
- Zwei CTL formulas ϕ und ψ sind *äquivalent*, $\phi \equiv \psi$, wenn für alle Modele M und alle Zustände s in M auch $M, s \models \phi$ gdw. $M, s \models \psi$ gilt.

CTL: Äquivalenzen

Gängige Abkürzungen bzw. Äquivalenzen:

$$\begin{aligned}
\phi_1 \vee \phi_2 &\equiv \neg(\neg\phi_1 \wedge \neg\phi_2) \\
\top &\equiv \phi \vee \neg\phi \\
EF\phi &\equiv E[\top U \phi] \\
AG\phi &\equiv \neg EF \neg\phi \\
AF\phi &\equiv \neg EG \neg\phi \\
AX\phi &\equiv \neg EX \neg\phi \\
A[\phi_1 U \phi_2] &\equiv \neg(E[\neg\phi_2 U \neg(\phi_1 \vee \phi_2)] \vee EG\neg\phi_2)
\end{aligned}$$

CTL: Äquivalenzen

Oft - und besonders beim Model Checking - benutzt man ein Set von "adequate connectives", d.h. ein Set von Junktoren, dass Ausdrucksstark genug ist, um jede Formel der Logik auszudrücken.

Für CTL sind solche Sets z.B.

$$\{\neg, \wedge, EX, AF, EU\}$$

oder

$$\{\neg, \wedge, EX, EG, EU\}$$

Alle Formeln, die andere Junktoren benutzen, können stets durch eine äquivalente Formeln ersetzt werden, die nur Junktoren obiger Mengen benutzen.

Das Model-Checking-Problem

Das Problem

Das *model checking problem* für LTL oder CTL fragt, gegeben ein LTS M und eine Formel ϕ, ob $M \models \phi$ gilt, d.h. ob M ein Model für ϕ ist.

Eingabe: Ein LTS M und eine LTL oder CTL Formel ϕ.
Frage: Gilt $M \models \phi$?

Model Checking. Ergebnisse

Satz

Sei M ein LTS.

1. *Sei ϕ eine LTL Formel. Das model checking problem für LTL, d.h. die Frage, ob $M \models \phi$ gilt, ist* PSPACE*-vollständig und kann in $O(|M| \cdot 2^{|\phi|})$ Zeit entschieden werden.*
2. *Sei ϕ eine CTL Formel. Das model checking problem für CTL, d.h. die Frage, ob $M \models \phi$ gilt, kann in $O(|M| \cdot |\phi|)$ Zeit entschieden werden.*

Wichtige Anmerkung

Das Modell M wird allerdings i.A. sehr schnell sehr groß. Daher ist $|M|$ der dominante Faktor, was zu dem berühmten Problem der Zustandsraumexplosion führt.

Die Idee

Sei M ein LTS und ϕ eine CTL Formel.

- Wähle eine Menge von "adequate connectives".
- Behandle die dortigen Operatoren, dazu:
  - Markiere jene Zustände von M mit den Teilformeln von ϕ, die in ihnen gelten.
  - Beginne dazu mit den kleinsten Teilformeln und arbeite "nach oben" bis zu ϕ.

Fallunterscheidung je nach Teilformel ψ...

Der Algorithmus (1)

Markiere $s \in S$ mit

- $\psi = p$, wenn $p \in L(S)$.
- $\psi = \psi_1 \wedge \psi_2$, wenn s bereits mit ψ_1 und ψ_2 markiert ist.
- $\psi = \neg\psi_1$, wenn s nicht bereits mit ψ_1 markiert ist.

Der Algorithmus (2)

Markiere $s \in S$ mit

- $\psi = EX\psi_1$, wenn ein Nachfolger von s mit ψ_1 markiert ist.

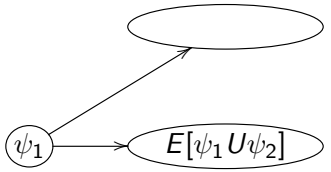
Für EU und AF brauchen wir Unterroutinen...

Der Algorithmus (3)

Ist $\psi = E[\psi_1 U \psi_2]$, dann

- markiere zunächst jeden Zustand mit ψ, der bereits mit ψ_2 markiert ist.
- Wiederhole bis keine Veränderung mehr Eintritt:
  - Markiere jeden Zustand mit ψ, der
    mit ψ_1 markiert ist und
    mindestens einen mit ψ markierten Nachfolger hat.

EU - bildlich



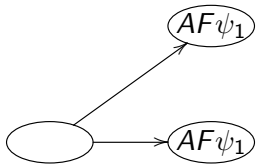
Hier wird der Zustand ganz links mit $E[\psi_1 U \psi_2]$ markiert.

Der Algorithmus (4)

Ist $\psi = AF\psi_1$, dann

- markiere zunächst jeden Zustand mit ψ, der bereits mit ψ_1 markiert ist.
- Wiederhole bis keine Veränderung mehr Eintritt:
  - Markiere jeden Zustand mit ψ, bei dem
    alle Nachfolger mit ψ markiert sind.

AF - bildlich



Hier wird der Zustand ganz links mit $AF\psi_1$ markiert.

Der Algorithmus - Laufzeit

Laufzeit?

- $O(\phi \cdot V \cdot (V + E))$

Das "Problem" ist AF. EX und EU kriegt man mit einem *backwards breadth-first search* effizient hin. (Kein Knoten muss mehr als einmal besucht werden!)

Der Algorithmus (4a)

Ist $\psi = EG\psi_1$, dann

- markiere zunächst jeden Zustand mit ψ, der bereits mit ψ_1 markiert ist
- Wiederhole bis keine Veränderung mehr Eintritt:
  - Entferne die Markierung ψ bei jedem Zustand, bei dem kein Nachfolger mit ψ markiert ist.

Das macht's noch nicht schneller ...

Der Algorithmus (4a)

Ist $\psi = EG\psi_1$, dann

- Schränke das Transitionssystem auf die Zustände ein, in denen ψ_1 gilt.
- Bestimme die maximalen strengen Zusammenhangskomponenten (maximal strongly connected components, SCCs.
- Benutze *backwards breadth-first search* auf dem eingeschränkten Transitionssystem, um alle Zustände zu bestimmen, von denen aus eine (nicht-triviale) SCC zu erreichen ist.

Das macht's schneller – wenn man die SCCs schnell bestimmen kann!

Breiten- und Tiefensuche

Wir brauchen zur Bestimmung der SCCs die Tiefensuche.

Außerdem brauchten wir im Algorithmus oben noch die Breitensuche.

Nachfolgend erst die Breiten- dann die Tiefensuche und dann die SCCs ...

(Im Anhang sind zudem Grundlagen zu Graphen zu finden.)

Breitensuche - Die Idee

Gegeben ein Graph G und ein Startknoten s 'entdeckt' die Breitensuche alle Knoten, die von s aus erreichbar sind. Zudem wird der Abstand (in Kanten) von s aus berechnet (tatsächlich wird sogar der *kürzeste Abstand* ermittelt).

Man kann auch zusätzlich einen 'Breitensuchbaum' (mit Wurzel s) und damit die kürzesten Pfade von s zu den anderen Knoten ermitteln.

Der Algorithmus entdeckt zunächst die Knoten mit Entfernung k und dann die mit Entfernung $k+1$, daher der Name. Er geht erst in die Breite...

Breitensuche - Die Idee

- Starte mit Knoten s in einer Queue Q.
- Wiederhole solange Q nicht leer...
  - Nimm vordersten Knoten v aus Q.
  - (Bearbeite diesen und Färbe diesen so, dass er nicht wieder besucht wird.)
  - Tue alle Nachbarn von v, die bisher nicht besucht wurden in die Queue.

Anmerkung

Durch die Queue wird sichergestellt, dass die Knoten 'in der Breite' besucht werden.

Breitensuche - Kernidee als Algorithmus

Algorithmus 1 BFS(G, s)

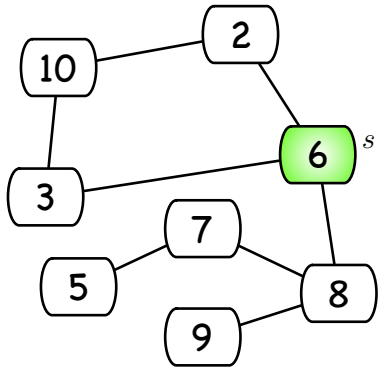
```
1:  farbe[s] = pink
2:  Q = ∅, enqueue(Q, s)
3:  while Q ≠ ∅ do
4:      u = dequeue(Q)
5:      for each v ∈ Adj[u] do
6:          if farbe[v] == weiss then
7:              farbe[v] = pink
8:              enqueue(Q, v)
9:          end if
10:     end for
11:     farbe[u] = schwarz
12: end while
```


Zur Nachbereitung

Anmerkung (zur Nachbereitung)

Wir verzichten für das nächste Beispiel auf die Angabe der einzelnen Adjazenzlisten. Man kann sie aus dem Beispiel ablesen.
Man beachte noch, dass je nachdem wie die Knoten in der Adjazenzliste angeordnet sind, der Algorithmus verschiedene Ergebnisse liefern kann. Wären die Knoten 7 und 9 bspw. in der Adjazenzliste der 8 in umgekehrter Reihenfolge, so würde zuerst die 9 abgearbeitet werden, bevor man bei 7 und 5 weiter im Baum hinabsteigt. (Bei der Tiefensuche später ist dieser Unterschied noch merklicher.)

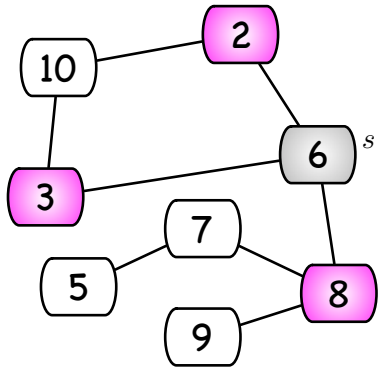
Breitensuche - im Graphen



A:

Q:

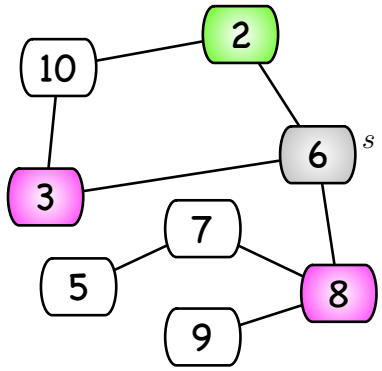
Breitensuche - im Graphen



A: 6

Q: 2 3 8

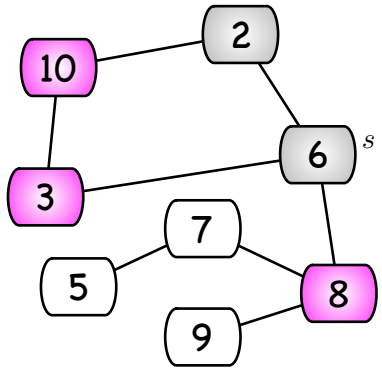
Breitensuche - im Graphen



A: 6

Q: 3 8

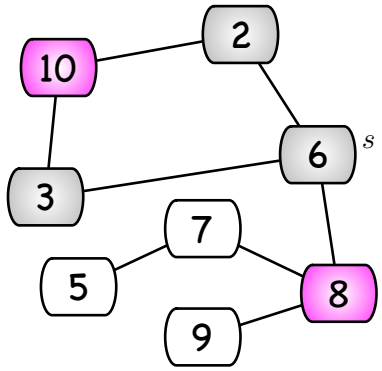
Breitensuche - im Graphen



A: 6 2

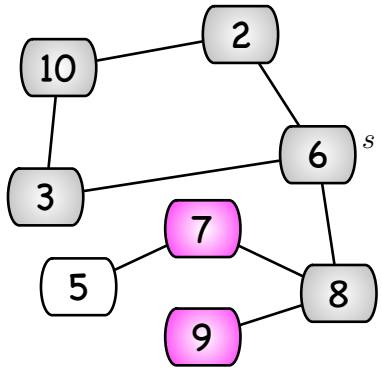
Q: 3 8 10

Breitensuche - im Graphen



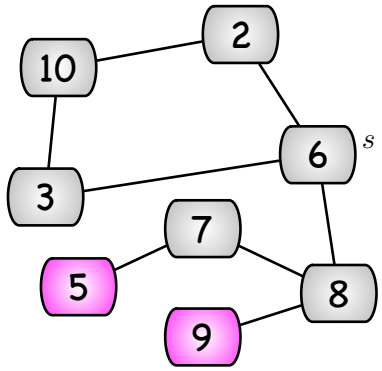
A:	6	2	3					
Q:	8	10						

Breitensuche - im Graphen



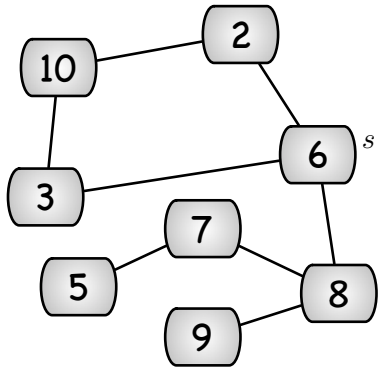
A:	6	2	3	8	10			
Q:	7	9						

Breitensuche - im Graphen



A:	6	2	3	8	10	7		
Q:	9	5						

Breitensuche - im Graphen



A:	6	2	3	8	10	7	9	5
Q:								

Breitensuche - mehr Informationen

Reichern wir die Breitensuche etwas an, so können wir mehr Informationen gewinnen, z.B. den Abstand der einzelnen Knoten von s und den Breitensuchbaum.

Der folgende Algorithmus ist wie im [Cormen].

Breitensuche - Initialisierung

Algorithmus 2 BFS(G, s) - Teil 1, Initphase

1: **for** each $u \in V(G) \setminus \{s\}$ **do**
2: farbe$[u]$ = weiss, $d[u] = \infty, \pi[u] = \texttt{nil}$
3: **end for**
4: farbe$[s]$ = grau
5: $d[s] = 0, \pi[s] = \texttt{nil}$
6: $Q = \emptyset$, enqueue(Q, s)

Anmerkung

Farben: weiss heißt 'noch nicht besucht', grau 'besucht, aber noch unbesuchte Nachbarn', schwarz 'besucht, alle Nachbarn entdeckt'. $d[u]$ ist die Anzahl der Schritt von s zu u, $\pi[u]$ der Vorgänger von u auf diesem Pfad.

Breitensuche - Hauptschleife

Algorithmus 3 BFS(G, s) - Teil 2, Hauptteil

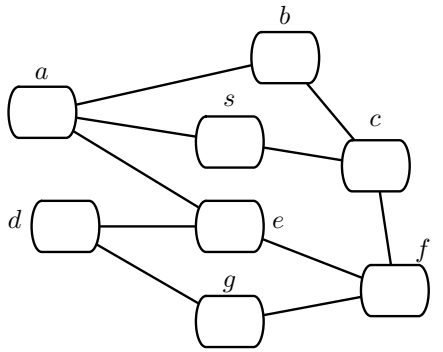
1: **while** $Q \neq \emptyset$ **do**
2: u = dequeue(Q)
3: **for** each $v \in Adj[u]$ **do**
4: **if** farbe$[v]$ == weiss **then**
5: farbe$[v]$ = grau
6: $d[v] = d[u] + 1, \pi[v] = u$
7: enqueue(Q, v)
8: **end if**
9: **end for**
10: farbe$[u]$ = schwarz
11: **end while**

Zur Nachbereitung

Anmerkung (zur Nachbereitung)

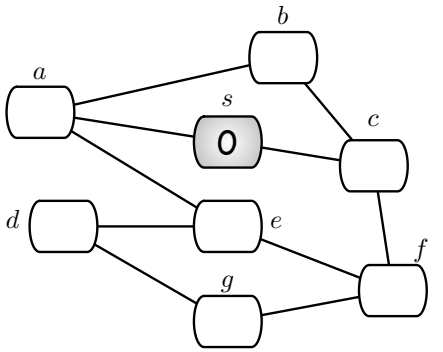
Für das nachfolgende Beispiel verzichten wir erneut auf die Angabe der Adjazenzlisten.
Die Zahl in einem Knoten v ist nun nicht mehr der Schlüssel o.ä., sondern der Wert von $d[v]$. Die Farbe 'grün' in den Bildern entspricht der Farbe 'schwarz' im Quellcode (d.h. ein grüner Knoten ist vollständig abgearbeitet und alle seine Kinder sind 'entdeckt' (also mindestens 'grau')).
Die dicken Kanten stellen die Baumkanten dar, die mittels der π Funktion gewonnen werden können, die jeden Knoten auf seinen Vorgänger im Baum abbildet.

Breitensuche - Beispiel



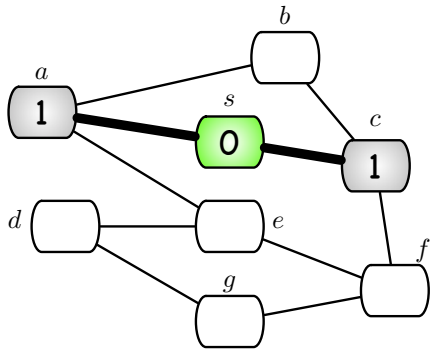
Q:

Breitensuche - Beispiel



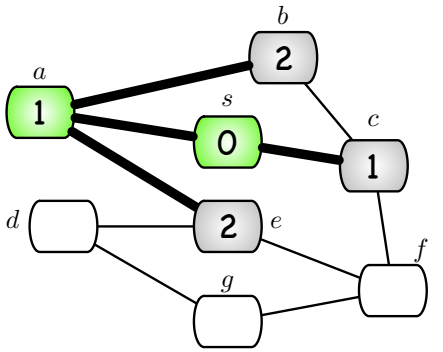
Q: s

Breitensuche - Beispiel



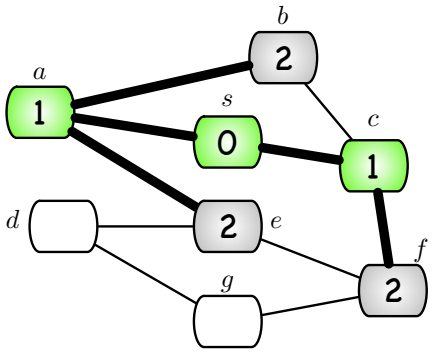
Q: a c

Breitensuche - Beispiel



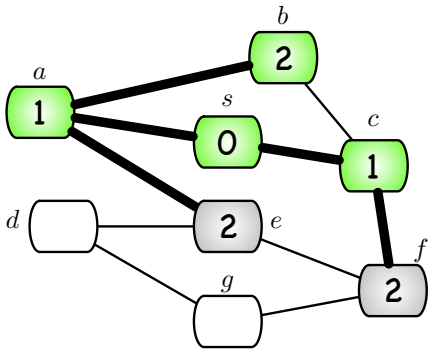
Q: c b e

Breitensuche - Beispiel



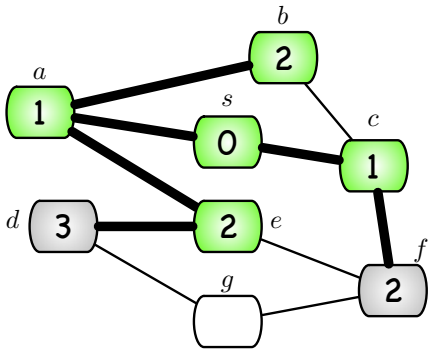
Q: b e f

Breitensuche - Beispiel



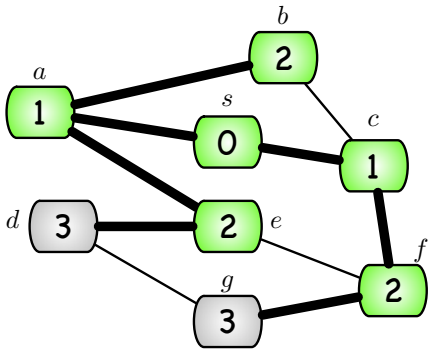
Q: e f

Breitensuche - Beispiel



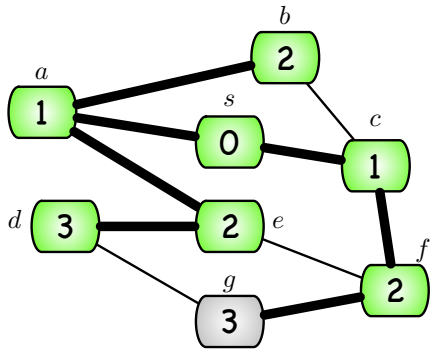
Q: f d

Breitensuche - Beispiel

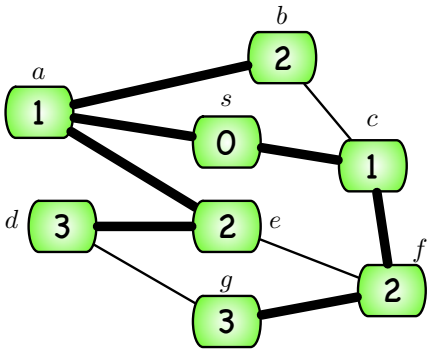


Q: d g

Breitensuche - Beispiel



Breitensuche - Beispiel



Analyse (Laufzeit)

- Zu Anfang ist jeder Knoten weiß. Der Test in der Schleife stellt daher sicher, dass jeder Knoten nur einmal in die Queue eingefügt und aus ihr entnommen wird.
- Für die Warteschlangenoperationen brauchen wir also $O(V)$.
- Die Adjazenzliste jedes Knotens wird nur einmal geprüft (wenn der Knoten aus der Queue entnommen wird).
- Die Summe aller Adjazenzlisten ist in $\Theta(E)$, folglich benötigt das Prüfen der Adjazenzlisten $O(E)$.
- Initialisierung geht auch in $O(V)$.
- Insgesamt ergibt sich so: $O(V + E)$, was linear in der Größe der Adjazenzliste ist.

Anmerkung

Diese Laufzeitanalyse stimmt nur, wenn der Graph durch Adjazenzlisten gegeben ist!

Analyse (Korrektheit)

Die Korrektheit kann man mittels einer Schleifeninvarianten zeigen, die besagt, dass die Queue aus der Menge der grauen Knoten besteht.

Alternativ kann man mit mehreren Lemmata zeigen, dass die ermittelten Abstände tatsächlich die kürzesten sind und ein Baum entsteht. Man erhält so (siehe [Cormen]):

Satz

Gegeben $G = (V, E)$ (gerichtet oder ungerichtet) und $s \in V$. BFS ermittelt jeden von s aus erreichbaren Knoten v. Bei Terminierung gilt $d[v] = \delta(s, v)$ für alle v. Zudem ist für jeden von s aus erreichbaren Knoten v $(v \neq s)$ einer der kürzesten Pfade von s nach v ein kürzester Pfad von s nach $\pi[v]$ gefolgt von der Kante $(\pi[v], v)$.

Breitensuche - Ergebnis

Zur Betonung:

Satz

In einem ungewichteten Graphen (bzw. einem Graph in dem jede Kante das Gewicht 1 hat) ermittelt der BFS-Algorithmus für jeden Knoten v den kürzesten Abstand zu s sowie einen kürzesten Pfad von s zu v (den man erhält indem man den $\pi[v]$ rückwärts folgt). Der Algorithmus läuft in $O(V + E)$, wenn der Graph als Adjazenzliste gegeben ist.

Beweis.

Beweise kann man im [Cormen] nachlesen (ca. 3 Seiten). □

Tiefensuche - Die Idee

Bei der Tiefensuche geht man einfach immer weiter noch ungeprüfte Kanten entlang, so lange dies möglich ist. Mein folgt also 'einem Pfad' so lange es möglich ist. Erst wenn man einen Knoten erreicht von dem aus man nur bereits besuchte Knoten erreichen kann, geht man einen Schritt zurück.

Bleiben unentdeckte Knoten übrig wählt man diese als neue Startknoten. So kann sich (bei einem unzusammenhängenden Graphen) ein Tiefensuch*wald* ergeben. (Die Breitensuche kann man auch so anpassen, aber typischerweise werden die Algorithmen so benutzt.)

Tiefensuche - Die Idee

- Starte mit Knoten s in einem Stack S.
- Wiederhole solange S nicht leer...
  - Nimm obersten Knoten v aus S.
  - (Bearbeite diesen und Färbe diesen so, dass er nicht wieder besucht wird.)
  - Tue alle Nachbarn von v, die bisher nicht besucht wurden auf den Stack.

Anmerkung

Durch den Stack wird sichergestellt, dass die Knoten 'in der Tiefe' besucht werden.

Tiefensuche - Idee

Algorithmus 4 DFS(G, s)

```
1:  farbe[s] = pink
2:  S = ∅, push(S, s)
3:  while S ≠ ∅ do
4:      u = pop(S)
5:      for each v ∈ Adj[u] do
6:          if farbe[v] == weiss then
7:              farbe[v] = pink
8:              push(S, v)
9:          end if
10:     end for
11:     farbe[u] = schwarz
12: end while
```

Tiefen- vs. Breitensuche

Beobachtung

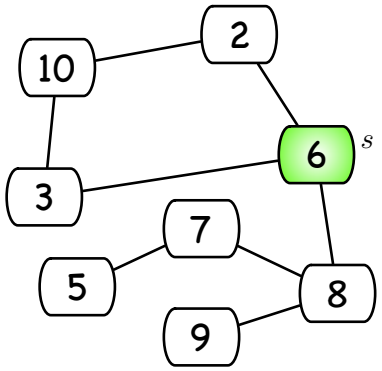
Der einzige Unterschied zur Breitensuche ist die Benutzung eines Stacks statt einer Queue!

Zur Nachbereitung

Anmerkung (zur Nachbereitung)

Wie bei der Breitensuche verzichten wir auch im nachfolgenden Beispiel auf die Angabe der Adjazenzlisten.

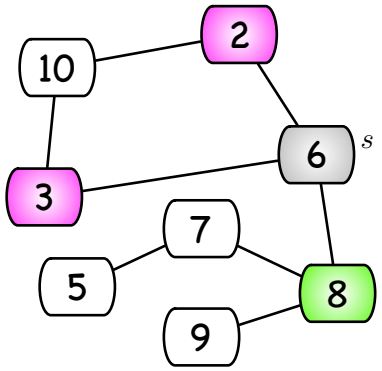
Tiefensuche - im Graphen



A: | | | | | | | | |

S: | | | | | | | | |

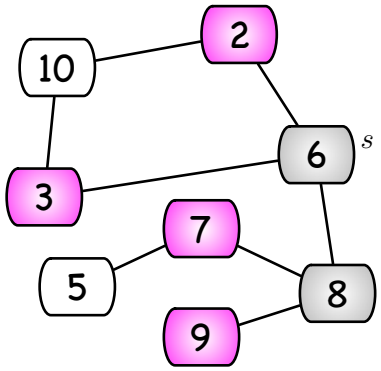
Tiefensuche - im Graphen



A: | 6 | | | | | | | |

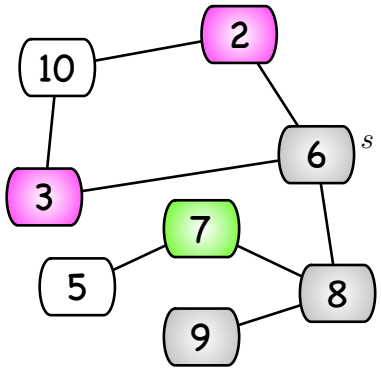
S: | 3 | 2 | | | | | | |

Tiefensuche - im Graphen



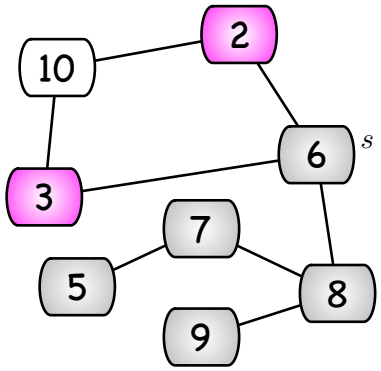
A:	6	8						
S:	9	7	3	2				

Tiefensuche - im Graphen



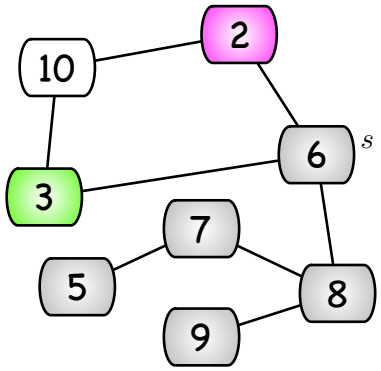
A:	6	8	9					
S:	3	2						

Tiefensuche - im Graphen



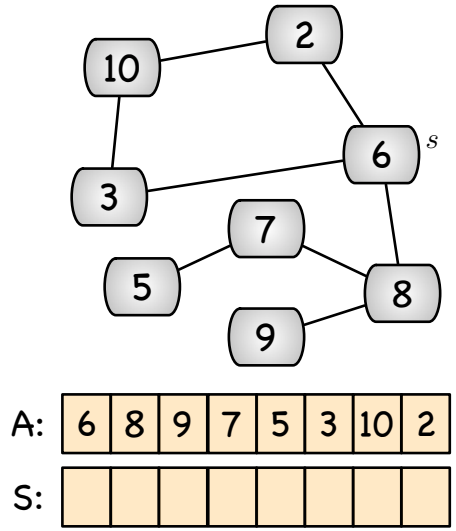
A:	6	8	9	7	5			
S:	3	2						

Tiefensuche - im Graphen



A:	6	8	9	7	5			
S:	2							

Tiefensuche - im Graphen



Tiefensuche - mehr Informationen

Reichern wir nun die Tiefensuche etwas an, so können wir mehr Informationen gewinnen, was insb. später nützlich wird. (Der folgende Algorithmus ist wieder wie im [Cormen].)

Anmerkung

- Farben: weiss heißt 'noch nicht besucht', grau 'besucht, aber noch unbesuchte Nachbarn', schwarz 'besucht, alle Nachbarn verarbeitet'.
- $d[u]$ ist 'discovery' Zeit, $f[u]$ ist 'finished' Zeit. (Beachte: Immer wenn $d[u]$ oder $f[u]$ gesetzt werden, wird vorher die Zeit erhöht!)
- $\pi[u]$ ist der Vorgänger von u in diesem Tiefensuchbaum.

Tiefenensuche - Initialisierung

Algorithmus 5 DFS(G) - Initphase und Aufruf

```
1: for each u ∈ V(G) do
2:     farbe[u] = weiss, π[u] = nil
3: end for
4: zeit = 0
5: for each u ∈ V(G) do
6:     if farbe[u] == weiss then
7:         DFS_VISIT(u)
8:     end if
9: end for
```


Tiefensuche - Hauptroutine

Algorithmus 6 DFS_VISIT(u) - Teil 2, Hauptteil

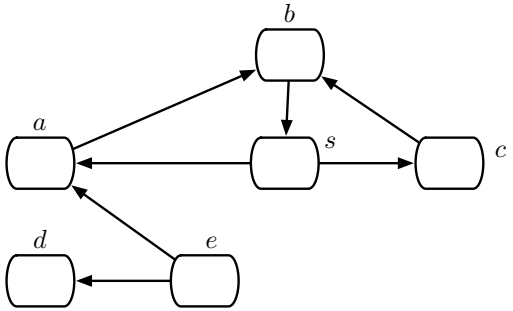
```
1: farbe[u] = grau
2: zeit = zeit + 1
3: d[u] = zeit
4: for each v ∈ Adj[u] do
5:     if farbe[v] == weiss then
6:         π[v] = u
7:         DFS_VISIT(v)
8:     end if
9: end for
10: farbe[u] = schwarz
11: zeit = zeit + 1
12: f[u] = zeit
```

Zur Nachbereitung

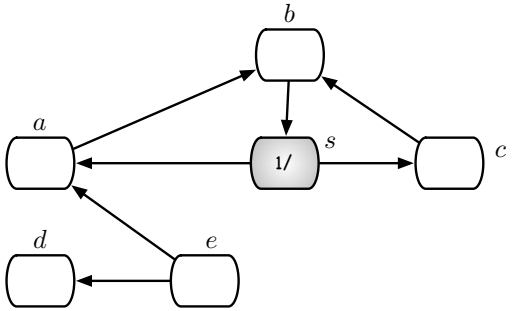
Anmerkung (zur Nachbereitung)

Für das nachfolgende Beispiel verzichten wir erneut auf die Angabe der Adjazenzlisten.
Die Zahlen x/y in einem Knoten v sind die Werte $d[v](= x)$ und $f[v](= y)$. Die Farbe 'grün' in den Bildern entspricht wieder der Farbe 'schwarz' im Quellcode, dicke Kanten stellen wieder Baumkanten dar. Die gestrichelten Kanten, sind entdecke Kanten, die als Rückwärtskanten (B) oder Querkanten (C) klassifiziert worden. Ferner gibt es Vorwärtskanten, die im Beispiel aber nicht auftreten. Eine Kante von s zu b wäre bei ansonsten gleichen Ablauf (d.h. wenn zuerst die Kante (s, a) gewählt wird) eine Vorwärtskante.

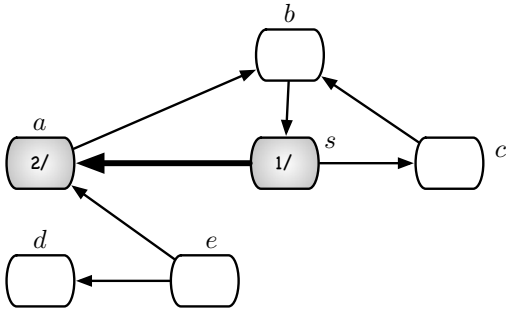
Tiefensuche - Beispiel



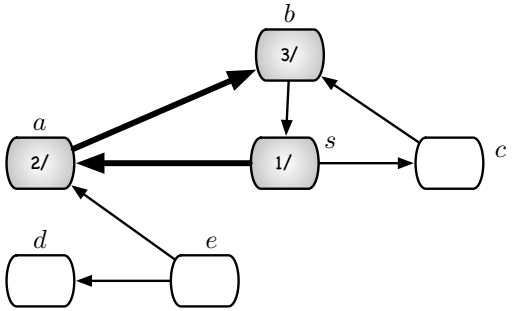
Tiefensuche - Beispiel



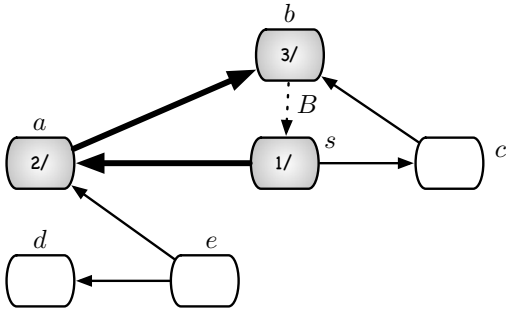
Tiefensuche - Beispiel



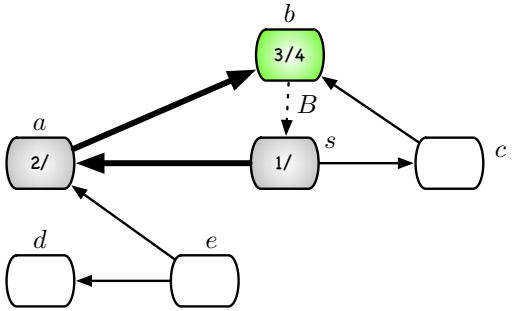
Tiefensuche - Beispiel



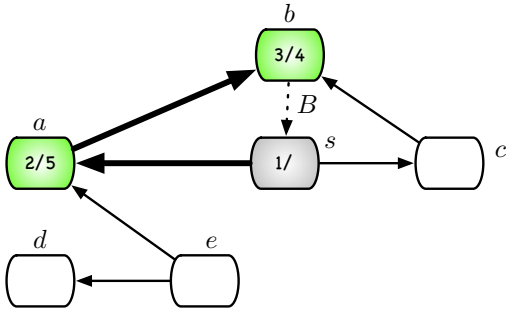
Tiefensuche - Beispiel



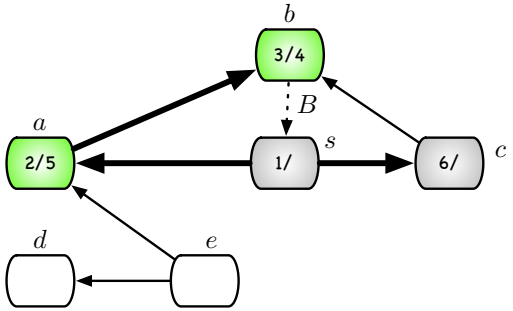
Tiefensuche - Beispiel



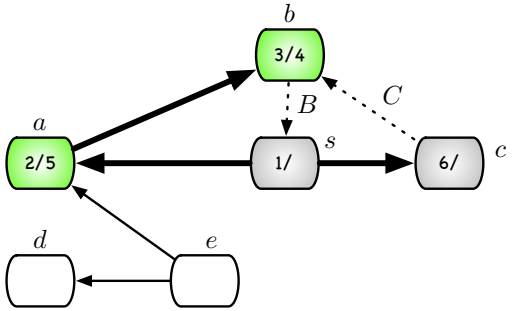
Tiefensuche - Beispiel



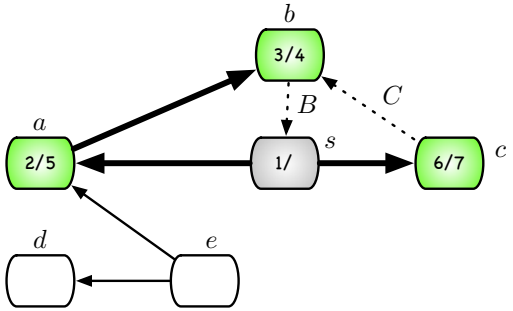
Tiefensuche - Beispiel



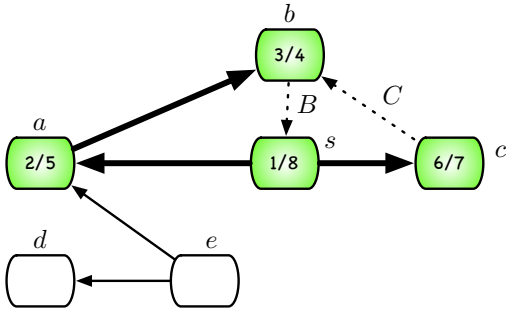
Tiefensuche - Beispiel



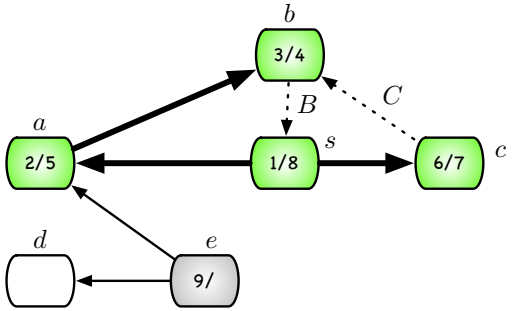
Tiefensuche - Beispiel



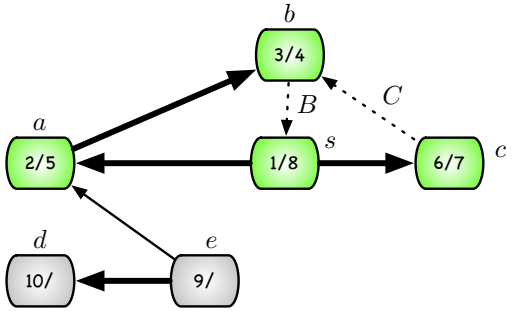
Tiefensuche - Beispiel



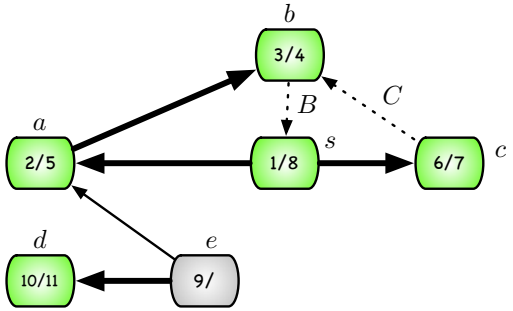
Tiefensuche - Beispiel



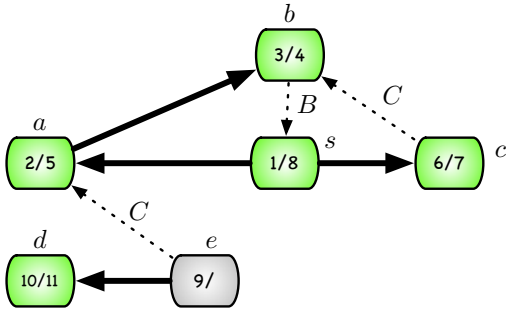
Tiefensuche - Beispiel



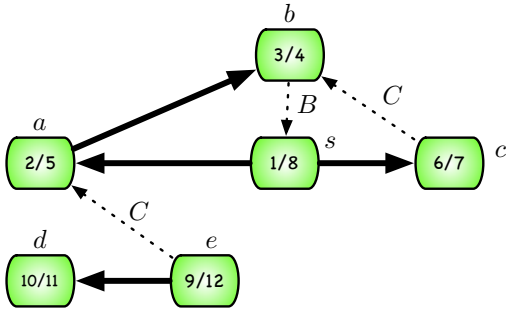
Tiefensuche - Beispiel



Tiefensuche - Beispiel



Tiefensuche - Beispiel



Tiefensuche - Besondere Kanten

Man kann bei der Tiefensuche die Kanten klassifizieren je nachdem, welche Farbe der Knoten v bei einer Kanten (u, v) hat (u ist der Knoten der gerade bearbeitet wird, v ist in $Adj[u]$):

- farbe$[v]$ = weiss: Baumkante.
- farbe$[v]$ = grau: Rückwärtskante.
- farbe$[v]$ = schwarz: Vorwärts- oder Querkante.
  - Vorwärtskante, falls $d[u] < d[v]$.
  - Querkante, falls $d[u] > d[v]$.

Anmerkung (zur Nachbereitung)

Im Beispiel eben ist dies bereits geschehen!

Tiefensuche - Besondere Kanten

Dabei ist

- Baumkante. Kanten, die zum Baum gehören. v wurde bei der Sondierung der Kante (u, v) entdeckt (erstmalig besucht).
- Rückwärtskante. Kante (u, v), die u mit einem Vorfahren v im Tiefensuchbaum verbindet.
- Vorwärtskante. (Nicht-Baum-)Kante (u, v), die u mit einem Nachfahre in v im Tiefensuchbaum verbindet.
- Querkanten. Die übrigen Kanten. Im gleichen Tiefensuchbaum, wenn der eine Knoten nicht Vorfahre des anderen ist oder zwischen zwei Knoten verschiedener Tiefensuchbäume (im Wald).

Anmerkung

Bei einem ungerichteten Graphen gibt es nur Baum- und Rückwärtskanten.

Analyse

Satz

Die Tiefensuche ist korrekt und ihre Laufzeit ist wieder in $O(V + E)$.

Breiten- und Tiefensuche - Zusammenfassung

Zusammenfassung

- Breitensuche nutzt eine Queue, Tiefensuche einen Stack. (Die Tiefensuche wie im [Cormen] kommt ohne Stack aus, arbeitet dafür aber rekursiv und mit Zeitstempeln.)
- Beide Operationen laufen in $\Theta(V + E)$, wenn der Graph mittels Adjazenzlisten gegeben ist.
- Die Breitensuche hat (meist) nur einen Startknoten. Sie wird oft verwendet, um kürzeste Abstände und den Vorgängerteilgraphen zu ermitteln.
- Die Tiefensuche ermittelt einen Tiefensuchwald (mehrere Startknoten). Sie wird oft als Unterroutine in anderen Algorithmen verwendet.

Breiten- und Tiefensuche - Erweiterungen

In der ganz grundlegenden Variante werden mit der Breiten- bzw. Tiefensuche nur Knoten in einer bestimmten Reihenfolge besucht (wobei die genaue Reihenfolge noch von der Reihenfolge, in der die Nachbarn eines Knotens gegeben sind, abhängt). Zudem kann

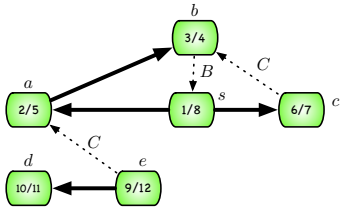
- die Breitensuche mit 'distance'-Stempeln angereichert werden, um so die kürzesten Abstände von s zu ermitteln und
- die Tiefensuche mit 'discovery' und 'finished' Zeit angereichert werden. So können die Kanten klassifiziert werden und man kann z.B. Kreise finden und noch andere Dinge tun. Wie oben kann man mit dem Vorgänger einen Tiefensuchbaum konstruieren.

Tiefensuche - Wichtige Eigenschaften

Es folgen einige wichtige - und nützliche - Eigenschaften der Tiefensuche...

(Beweise kann man im [Cormen] nachlesen. Nur ca. eine Seite)

Tiefensuche - Klammerungstheorem (1/2)

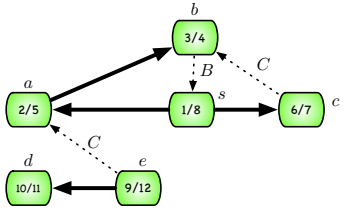


Satz

Bei der Tiefensuche erfüllt jedes Paar u, v von Knoten genau eine der folgenden Bedingungen:

1. *Die Intervalle $[d[u], f[u]]$ und $[d[v], f[v]]$ sind paarweise disjunkt und keiner der Knoten u und v ist Nachfahre des anderen.*

Tiefensuche - Klammerungstheorem (2/2)

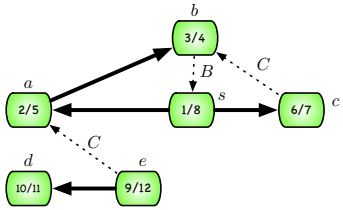


Satz

Bei der Tiefensuche erfüllt jedes Paar u, v von Knoten genau eine der folgenden Bedingungen:

2. *Das Intervall $[d[u], f[u]]$ ist vollständig im Intervall $[d[v], f[v]]$ enthalten und u ist im Tiefensuchwald ein Nachfahre von v.*
3. *Wie 2. nur mit u und v vertauscht.*

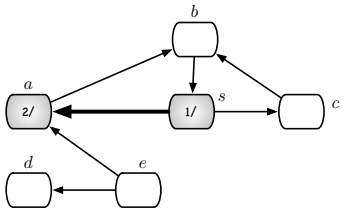
Tiefensuche - Intervalle der Nachfahren



Satz

Im Tiefensuchwald von G ist v genau dann ein echter Nachfahre von u, wenn $d[u] < d[v] < f[v] < f[u]$ gilt.

Tiefensuche - Theorem der weißen Pfade



Satz

Im Tiefensuchwald von G ist v genau dann ein Nachfahre von u, wenn v zur Zeit d[u] (u wird entdeckt) von u aus entlang eines nur aus weißen Knoten bestehenden Pfades erreichbar ist.

Problemstellung

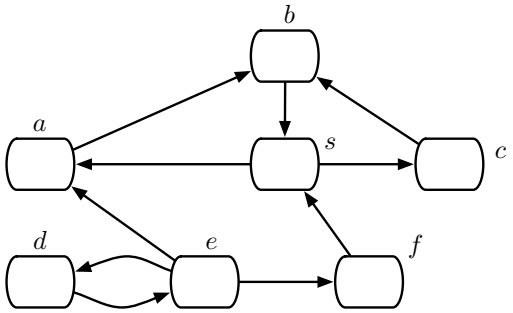
Definition (Bestimmung starker Zusammenhangskomponenten)

Eingabe: Gegeben ein gerichteter Graph $G = (V, E)$.
Gesucht: Die starken Zusammenhangskomponenten (SCCs) des Graphen, d.h. maximale Menge $C \subseteq V$ derart, dass für jedes Paar $u, v \in C$ sowohl $u \Rightarrow v$ als auch $v \Rightarrow u$ gilt (es gibt einen Pfad von u nach v und andersherum).

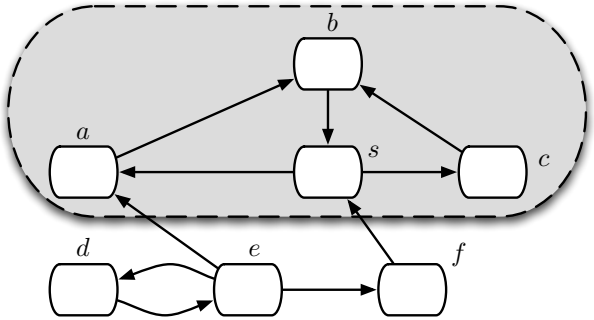
Anmerkung

Dieses Problem tritt oft bei gerichteten Graphen auf. Zunächst wird der Graph in seine starken Zusammenhangskomponenten zerlegt und dann werden diese separat betrachtet. (Oft werden die Lösungen anschließend noch entsprechend der Verbindungen zwischen den einzelnen Komponenten zusammengefügt.)

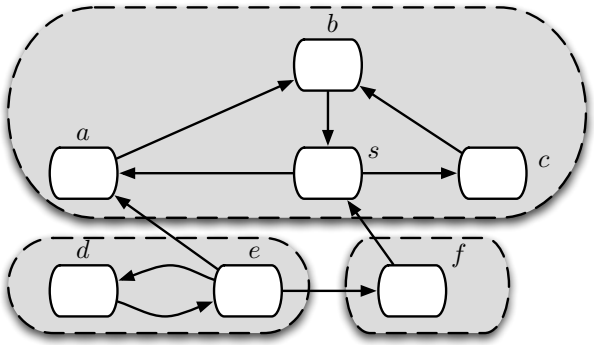
Ein Beispiel



Ein Beispiel



Ein Beispiel



Der Komponentengraph

Oft ist auch der Komponentengraph gesucht...

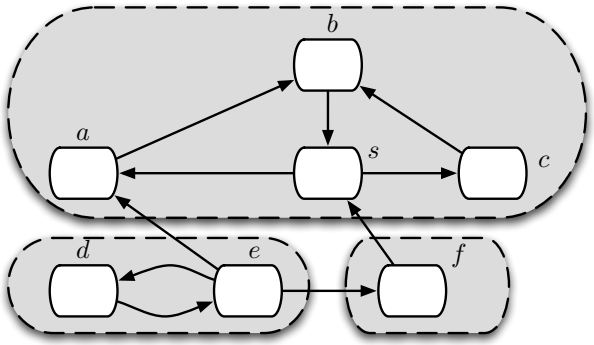
Definition

Sei G ein gerichteter Graph mit Zusammenhangskomponenten $C_1, \ldots, C_k$. Der Komponentengraph $G^{scc} = (V^{scc}, E^{scc})$ ist definiert durch $V^{scc} = \{v_1, \ldots, v_k\}$ (jeder Knoten v_i repräsentiert dabei eine Zusammenhangskomponente C_i) und (v_i, v_j) ist genau dann eine Kante in E^{scc}, wenn es in G einen Knoten $x \in C_i$, einen Knoten $y \in C_j$ und eine Kante (x, y) gibt.

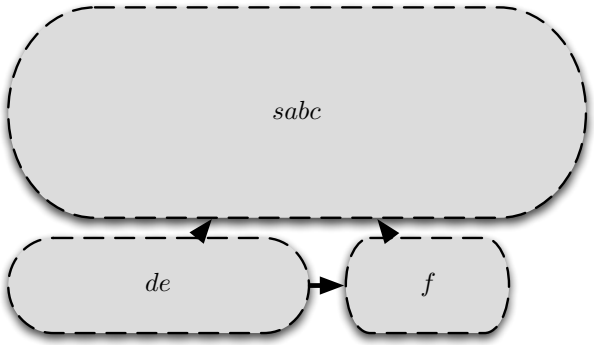
Anmerkung

Der Komponentengraph lässt sich aus den SCCs ermitteln.

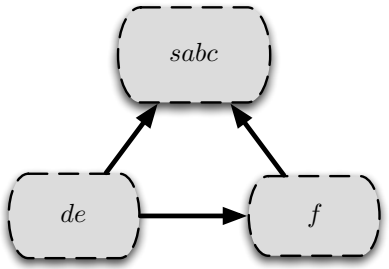
Der Komponentengraph - Beispiel



Der Komponentengraph - Beispiel



Der Komponentengraph - Beispiel



SCC - Algorithmus und Idee

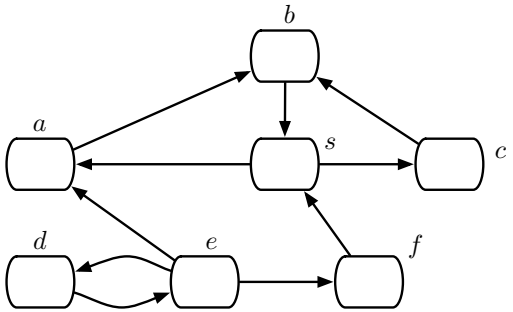
Algorithmus 7 SCC(G)

1: Aufruf von DFS(G) zur Berechnung der Endzeiten $f[v]$.
2: Berechne G^T (transponierter Graph).
3: Aufruf von DFS(G^T), betrachte in der Hauptschleife von DFS die Knoten jedoch in der Reihenfolge fallender $f[v]$ (in Zeile 1 berechnet).
4: Die Knoten jedes in Zeile 3 berechneten Baumes sind eine (separate) strenge Zusammenhangskomponente.

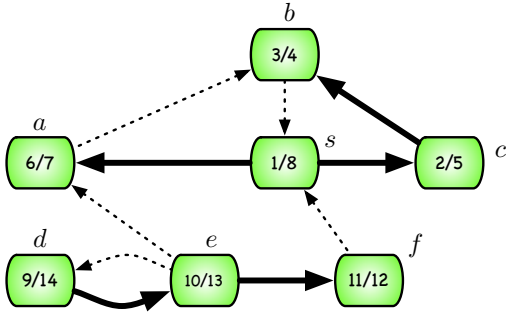
Anmerkung

Der zu einem Graphen $G = (V, E)$ transponierte Graph ist definiert durch $G^T = (V, E^T)$ mit $E^T = \{(u, v) \mid (v, u) \in E\}$. Ist G in Adjazenzlisten-Darstellung gegeben, kann G^T in Zeit $O(V + E)$ berechnet werden.

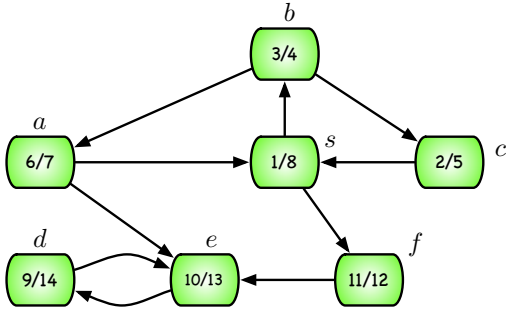
SCC - Beispiel



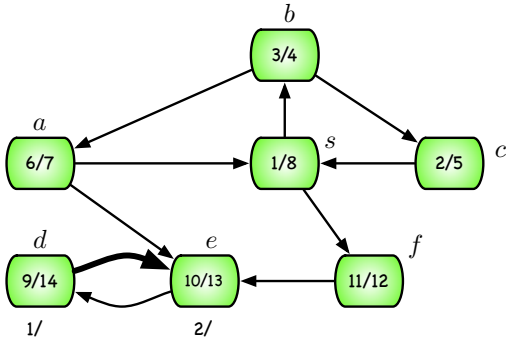
SCC - Beispiel



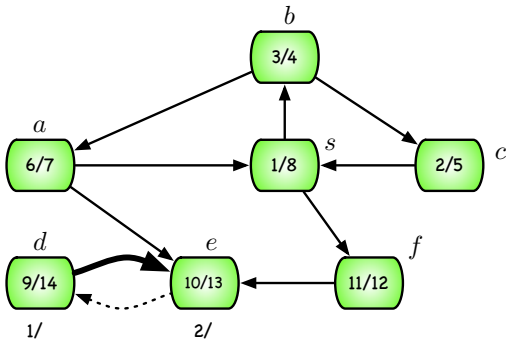
SCC - Beispiel



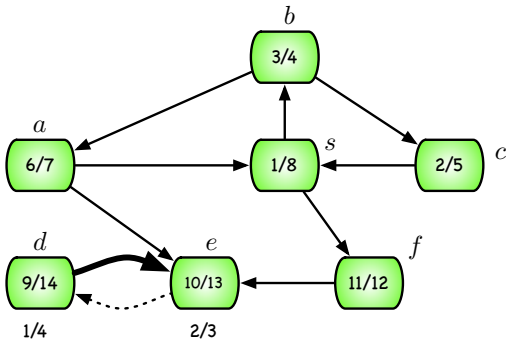
SCC - Beispiel



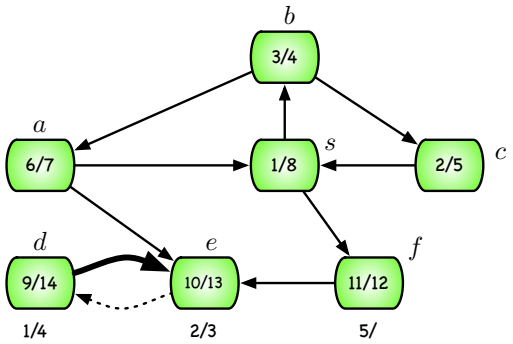
SCC - Beispiel



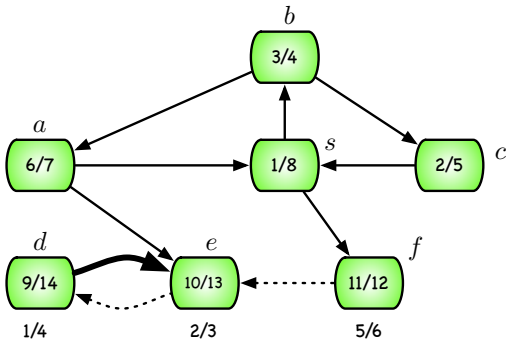
SCC - Beispiel



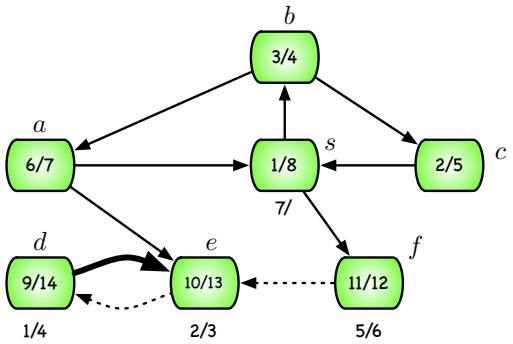
SCC - Beispiel



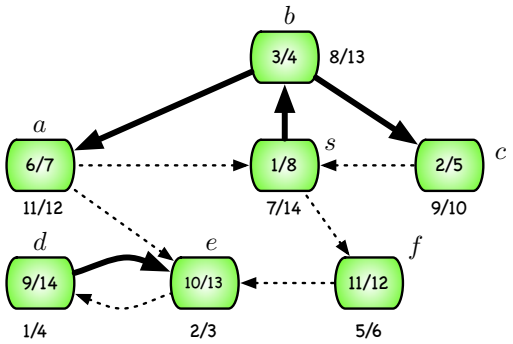
SCC - Beispiel



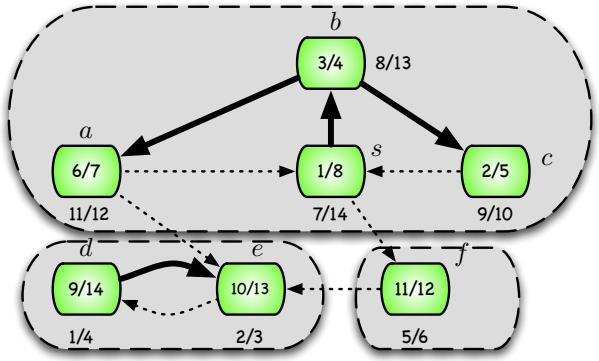
SCC - Beispiel



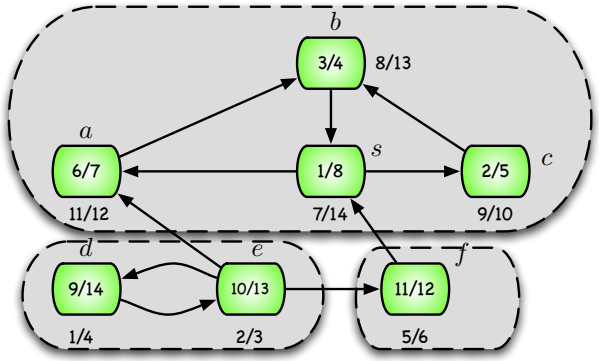
SCC - Beispiel



SCC - Beispiel



SCC - Beispiel



SCC - Analyse

Satz

- *G und G^T haben die gleichen SCCs.*
- *Seien C und C' zwei verschiedene SCCs eines DAG $G = (V, E)$.*
  - *Seien $u, v \in C$ und $u', v' \in C'$. Gibt es in G einen Pfad $u \Rightarrow u'$, so gibt es keinen Pfad $v' \Rightarrow v$ in G.*
  - *Gibt es $(u, v) \in E$ mit $u \in C$ und $v \in C'$, so gilt $f(C) > f(C')$.*
  - *Gibt es $(u, v) \in E^T$ mit $u \in C$ und $v \in C'$, so gilt $f(C) < f(C')$.*
- *$\mathtt{SCC}(G)$ ist korrekt und hat eine Laufzeit in $\Theta(V + E)$.*

Für Interessierte

Interessierte Leser finden die Beweise in Kapitel 22 im [Cormen] (ca. 3 Seiten)

Anhang

Anhang

Graphen

Einführung

Graphen sind eine grundlegende Datenstruktur, die in vielen Bereichen der Informatik (und auch in anderen Bereichen) Anwendung findet. Man kann ohne Einschränkung zwei Elemente einer Mengen (den Knoten) in Beziehung setzen (durch eine Kante).

Anmerkung

Erlaubt man verschiedene Kanten-'Typen', so kann man sogar verschiedene Beziehungen ausdrücken.

Definitionen

Definition

Ein *Graph* ist ein Tupel $G = (V, E)$ bestehend aus einer Menge V (auch $V(G)$) von *Knoten* oder Ecken und einer Menge E (auch $E(G)$) von *Kanten*.
Ist G ein *ungerichteter Graph*, so ist

$$E \subseteq \{\{v_1, v_2\} \mid v_1, v_2 \in V, v_1 \neq v_2\},$$

ist G ein *gerichteter Graph*, so ist

$$E \subseteq V^2.$$

Ist $|E|$ viel kleiner als $|V|^2$, so nennt man den Graphen *dünn besetzt*. Ist $|E|$ nahe an $|V|^2$, so spricht man von *dicht besetzten* Graphen.

Gewichteter Graph

Definition

Bei einem *gewichteten Graphen* ist neben dem Graph $G = (V, E)$ (gerichtete oder ungerichtet) noch eine *Gewichtsfunktion* $w : E \to \mathbb{R}^+$ gegeben, die jeder Kante $e \in E$ ihre *Kosten* $w(e)$ zuweist.

Definitionen

Definition

- Sind je zwei Knoten von G mit einer Kante verbunden, so ist G ein *vollständiger Graph*. Bei n Knoten: K^n.
- Eine Menge paarweise nicht benachbarter Knoten nennt man *unabhängig*.
- Der *Grad* $d(v)$ eines Knotens v ist die Anzahl mit v inzidenter Kanten.
- Die Menge der *Nachbarn* eines Knotens v bezeichnet man mit $N(v)$ (hier gilt $d(v) = |N(v)|$).
- $\delta(G)$ ist der *Minimalgrad* von G, $\Delta(G)$ der *Maximalgrad*.

Definitionen

Definition

- Ein *Weg* ist ein nicht leerer Graph $P = (V, E)$ mit $V = \{x_0, x_1, \ldots, x_k\}$, $E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\}$, wobei die x_i paarweise verschieden sind. x_0 und x_k sind die Enden von P, sie sind durch P verbunden. Die Anzahl der Kanten eines Weges ist seine *Länge*.
- Ist P wie oben ein Weg, so ist $P + x_kx_0$ ein Kreis (der Länge $k + 1$).
- Der Abstand zweier Knoten x und y voneinander wird mit $d(x, y)$ bezeichnet und ist die geringste Länge eines x-y-Weges.

Definitionen

Definition

- Seien $G = (V, E)$ und $G' = (V', E')$ Graphen. Gilt $V' \subseteq V$ und $E' \subseteq E$, so nennt man G' einen *Teilgraphen* von G.
- Ist $G = (V, E)$ ein Graph und $V' \subseteq V$, so nennt man den Graphen $G' = (V', E')$ mit $E' = \{\{v_1, v_2\} \in E \mid v_1, v_2 \in V'\}$ den von V' *induzierten Graphen*.

Darstellung von Graphen

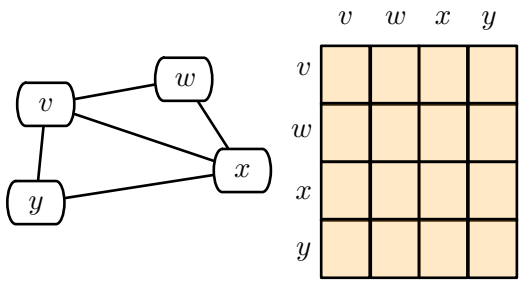
Ein Graph $G = (V, E)$ wird dargestellt indem man seine Knoten als Punkte und die Tupel oder Mengen aus E als (gerichtete) Kanten zwischen die Knoten einzeichnet.

Im Computer speichert man einen Graphen meist mittels einer *Adjazenzmatrix* oder einer *Adjazenzliste*. (Man kann die Mengen V und E aber auch direkt speichern.)

Anmerkung

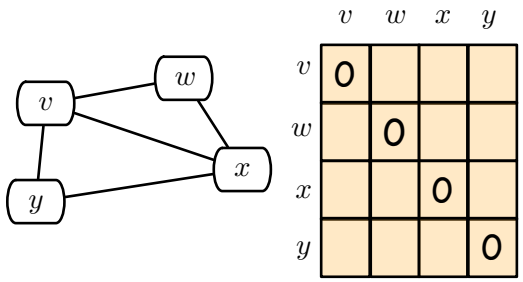
Bei Graphen schreibt man (und wir) oft $O(V + E)$ etc., wenn $O(|V| + |E|)$ gemeint ist. Man beachte zudem, dass dies die Komplexität bzgl. der Kenngrößen V und E ausdrückt und nicht umbedingt die Größe der Eingabe wiederspiegelt!

Darstellung von Graphen - Adjazenzmatrix



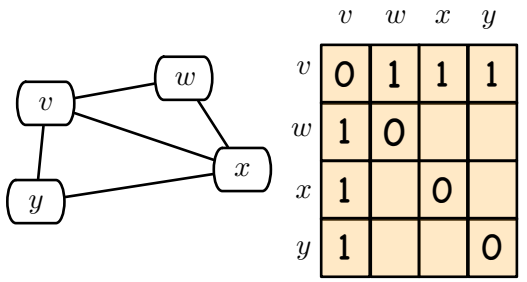
$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

Darstellung von Graphen - Adjazenzmatrix



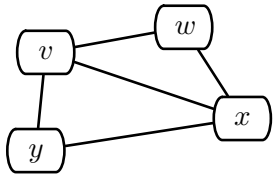
$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

Darstellung von Graphen - Adjazenzmatrix



$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

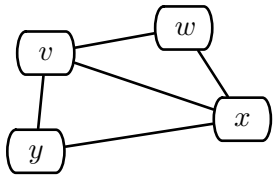
Darstellung von Graphen - Adjazenzmatrix



	v	w	x	y
v	0	1	1	1
w	1	0	1	0
x	1	1	0	1
y	1	0	1	0

$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

Darstellung von Graphen - Adjazenzmatrix

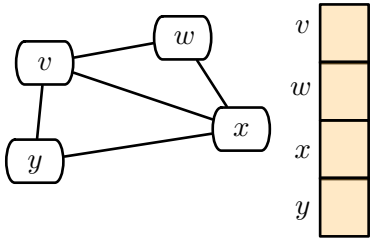


	v	w	x	y
v	0	1	1	1
w	1	0	1	0
x	1	1	0	1
y	1	0	1	0

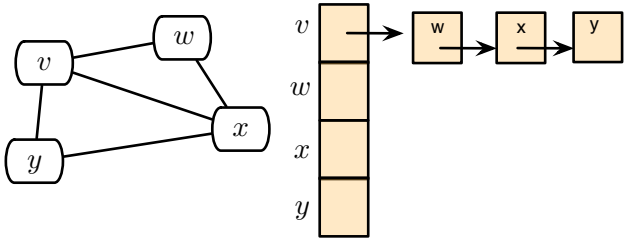
Bei einer Adjazenzmatrix hat man eine $n \times n$-Matrix, bei der an der Stelle (i, j) genau dann eine 1 steht, wenn v_i und v_j verbunden sind.

Der Speicherplatzbedarf ist in $\Theta(V^2)$ (unabhängig von der Kantenzahl).

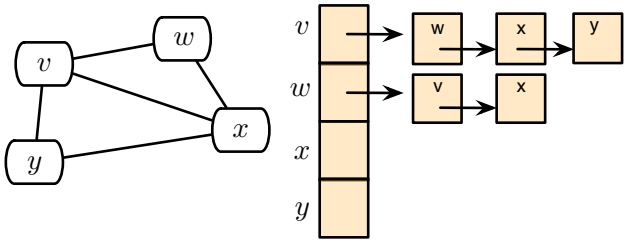
Darstellung von Graphen - Adjazenzlisten



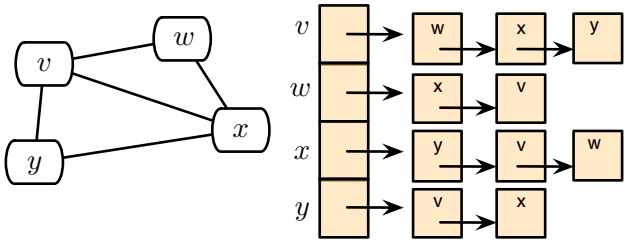
Darstellung von Graphen - Adjazenzlisten



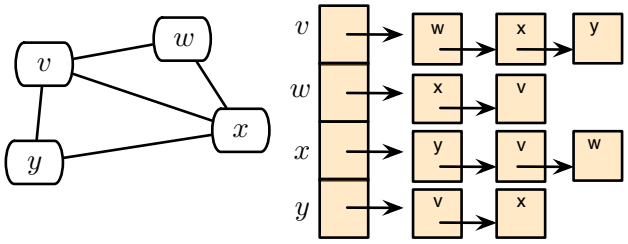
Darstellung von Graphen - Adjazenzlisten



Darstellung von Graphen - Adjazenzlisten



Darstellung von Graphen - Adjazenzlisten



Bei der Adjazenzlistendarstellung haben wir ein Array von $|V|$ Listen, für jeden Knoten eine. Die Adjazenzliste $Adj[v]$ zu einem Knoten v enthält alle Knoten, die mit v adjazent sind.

Bei einem gerichteten Graphen ist die Summe aller Adjazenzlisten $|E|$, bei einem ungerichteten Graphen $|2E|$. Der Speicherplatzbedarf ist folglich $\Theta(V + E)$.

Darstellung von Graphen - Zusammenfassung

- Adjazenzmatrix: $|V| \times |V|$-Matrix $A = (a_{ij})$ mit $a_{ij} = 1$ falls $(i, j) \in E$ und 0 sonst. Größe in $\Theta(V^2)$.
- Adjazenzliste: Liste $Adj[v]$ für jeden Knoten $v \in V$ in der die Knoten, die mit v adjazent sind gespeichert sind. Größe in $\Theta(V + E)$.
- Bei einer Adjazenzmatrix kann man schnell herausfinden, ob zwei Knoten benachbart sind oder nicht. Dafür ist es langsamer alle Knoten zu bestimmen, die mit einem Knoten benachbart sind. (Bei Adjazenzlisten genau andersherum.)
- Beide Darstellungen sind ineinander transformierbar.
- Beide Darstellungen sind leicht auf den Fall eines gewichteten Graphen anpassbar.

Die Struktur eines Graphen

Oft möchte man etwas grundlegendes über die *Struktur* eines gegebenen Graphen erfahren. Hierzu ist es zunächst praktisch Algorithmen zu haben, die den Graphen *durchwandern*, d.h. in systematischer Weise die Kanten entlangwandern und die Knoten besuchen. Oft ist es auch nützlich einen *Spannbaum* zu ermitteln.

Definition

Ein Spannbaum ist ein Teilgraph T eines Graphen G, wobei T ein Baum ist und alle Knoten von G enthält.

Breiten- und Tiefensuche

Die *Breiten-* und *Tiefensuche* in einem Graphen erreichen (im Prinzip) beide Ziele! Es wird zwar i.A. nicht der minimale Spannbaum ermittelt, doch dazu gibt es speziellere Algorithmen.

Breiten- und Tiefensuche sind oft 'Urtypen' für weitere Graphalgorithmen entweder

- als wichtige Subroutine oder
- als 'Ideengeber'

Beides wurde im Hauptteil dieser Vorlesung behandelt.