Breadth first search.

Algorithmus (Breitensuche, BFS)

BFS(G, s)

Eingabe: $G = (V, E)$ in Adj.-Listen-Dstg., $|V| = n$.
und $s \in V$ der Startknoten.

Datenstrukturen:

Schlange $Q = Q[1 \ldots n]$ mit head, tail.

Array $col[1 \ldots n]$, um zu merken, ob Knoten bereits entdeckt.

$col[u] =$ weiß

$\Leftrightarrow$ u noch nicht entdeckt

$col[u] =$ grau

$\Leftrightarrow$ u entdeckt, aber noch nicht abgearbeitet

$col[u] =$ schwarz

$\Leftrightarrow$ u entdeckt und abgearbeitet.

1. for each $u \in V$ {

   col[u] = weiß

   }

2. col[s] = grau; // s Startknoten

   $Q = \{s\}$

   // Initialisierung
   // zu Ende.

3. while $Q \neq \emptyset$ { // Testbar mit
   // tail ≠ head

4. u := Q[head]
   // u wird bearbeitet

5. for each $v \in Adj[u]$ { (expandiert)

6. if col[v] = weiß {
   // v wird entdeckt.

7. col[v] = grau;

8. v in Schlange tun

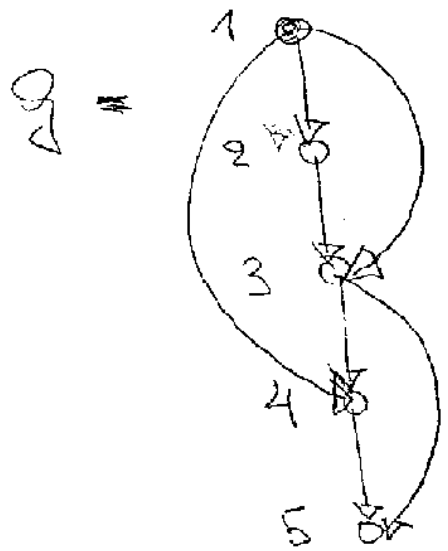
   }
   // Schlange wurde grau

   }

9. u aus Q raus; col[u] = schwarz □

1.24

Noch ein Beispiel:



BFS (G, 1)

Q

1

Q 2,3,4

3,4

4,5

5

Q

Entdeckkante 1

Entdeckkante 2

Zusätzlich: Array $d[1 \ldots n]$

mit $d[v]$ = Entdeckzeit von v.

Anfangs $d[s] = 0$, $d[v] = \infty$

für $v \neq s$. Wird v über u

entdeckt, so setzen wir

$d[v] = d[u] + 1$.

Bei Abarbeitung von u.

Zusätzlich: Breitensuchbaum

durch Array $\pi[1 \ldots n]$.

$\pi[s]$ = nil, da s Wurzel.

Wird v über u entdeckt,

dann

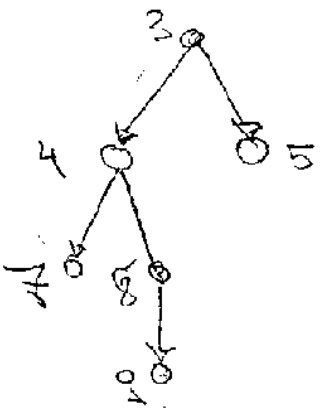
$\pi[v] := u$

1.26

Interessante Darstellung

des Baums durch π : Vaterarray

$\pi[u]$ = Vater von u!



$\pi[2] = 8$

$\pi[8] = 4$

$\pi[4] = 3$

$\pi[5] = 3$

$\pi[7] = 4$

$\pi[u] = v \Rightarrow$ Kante (v, u)

$v \to u$ im Graph.

Verifikation: Wollen etwa haben: Am Ende von BFS(G, s) ist

$\{v \mid col[v] = schwarz\}$ = Menge der von s erreichbaren Knoten.

v von s erreichbar :⇔ Es gibt Weg $(v_0, v_1, \ldots, v_k)$, $v_0 = s$, $v_k = v$

Invariante:

$grau_i = Q_i$

"$schwarz_i + grau_i$ von $grau_i$ erreichbar" (aber nicht in $grau_i$)

= Menge der von s erreichbaren Knoten

Beweis der Invariante: (induktiv s!)

Gilt vor Beginn der Schleife:

$schwarz = \emptyset$, $grau = \{s\}$.

gilt Invariante vor dem i'ten Lauf,
dann auch vor dem $i+1$'ten:

Von s erreichbar (Das ist noch vor dem i'ten.)

$=$ Variable noch i

sch_i + q_i + von q_i erreichbar, (nicht in q_i)

(u; Adj[u] ∩ weiß)

$=$

sch_{i+1} + q_{i+1} + von q_{i+1} erreichbar (nicht in q_{i+1})

$= sch_i$ ∘ u falls ein $u \in q_i$ $= q_i \setminus \{u\} \cup$ (Adj[u] ∩ weiß)

$\subseteq$ von q_i erreichbar

Quintessenz: $Q_i = \emptyset$, dann $q_{i+1} = \emptyset$, damit „Von s erreichbar $= sch_i$"
neg. Invariante.

1.23

Definition (Distanz)

Sei $G = (V, E)$ ges. Graph. Für

$u, v \in V$ sei

Dist(u,v) ← Distanz von u nach v.

= minimale Weglänge von u nach v

= $\min\{k \mid$ Es gibt Weg der Länge k von u nach v in $G\}$

= $\min\{k \mid$ Es gibt $(v_0, \dots, v_k)$ mit $v_0 = u$ und $v_k = v$ in $G\}$.

□

Nach BFS(G, s) ist $d[u] = \text{Dist}(s,u)$.

Invariante

Kante $u \to v$, wobei u von v verschieden.

Beobachtung

In $\mathcal{BTG}(\mathcal{L}, \Lambda)$ wird jede von Λ verschiedene "Kante" genau einmal entworfen.

Denn $u \to v$, dann u grau, v ungefärbt. Danach u schwarz und nie mehr grau. $\square$

1.31

wenn $x \to y$ gegangen wird

bei $BTZ(9,5)$, d.h. wenn

u expandiert wird, ist

u grün (d.h. color[u] = grün)

und v kann schwarz, grün

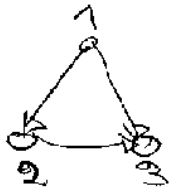
oder weiß sein.

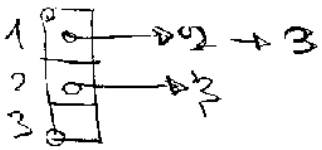
Die Farbe von x bei dem Entfernen

hängt nicht nur von y selbst,

sondern auch von den Adj.-Listen

ab. □

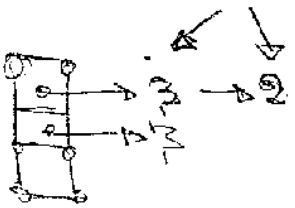




BFS(2,1)

2
1
2 3
3
ϕ

Beim Pfeil $2 \to 3$ ist 3 grau.



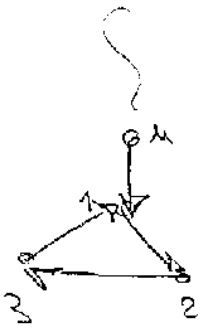
Beim Pfeil $2 \to 3$ ist 3 schwarz

Andere Farben: Übung 2.

1.23

Weitere Netzen der Breitensuche

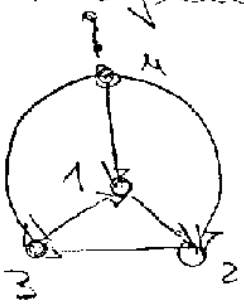
Kreise finden



in G.

1 erste Knoten, der auf Kante entdeckt wird. Wenn 3 expandiert wird ist 1 schwarz

Alle möglichen



in G

Alle 3 Knoten 1,2,3 werden bei Expansion von 4 das erste Mal entdeckt. Trotzdem: Es gibt

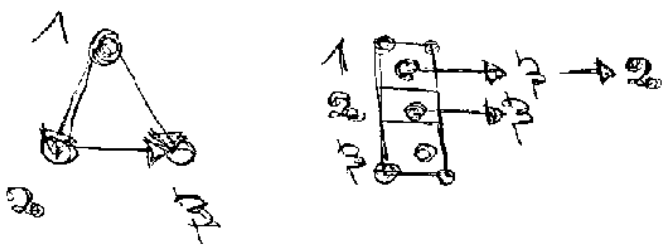
Kante, die bei Beabeitung
auf schwarzen Knoten führt.

Folgerung

Kreis ⇒ es gibt ○→○ Kante,
so daß ● schwarz, wenn
○→● gezogen wird. □

Gibt ein Kreis davon abhängen?

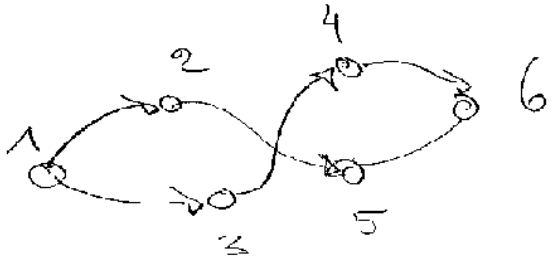
Leider nein:



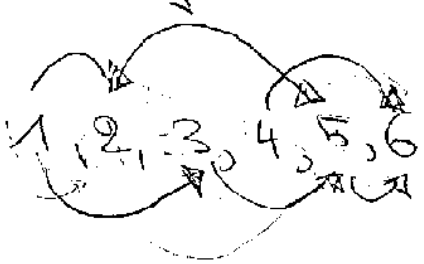
Dann 3 schwarz bei ○→○,
trotzdem kein Kreis.

1.36

Wie stelle ich kreisfreie gerichtete Graphen?

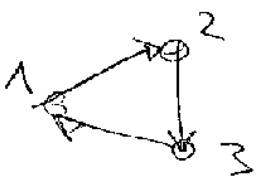


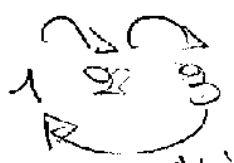
Anordnung der Knoten



Alle Kanten gemäß der Ordnung.

Aber bei Kreis





nicht in Ordnung

nicht in Ordnung.

1.37

Definition (Topologische Sortierung)

Sei $G = (V, E)$ ein gerichteter Graph.
Eine topologische Sortierung ist eine Anordnung von V: also

$$(v_1, v_2, v_3, \dots, v_n),$$

so daß gilt: Ist $u \to v \in E$,
so ist

$$u = v_i, \; v = v_j \text{ und } i < j.$$

Alle Kanten gehen von links nach rechts in der Ordnung. □

Satz

G hat topologische Ordnung

$\Leftrightarrow$

G hat keinen Kreis.

Beweis.

„$\Rightarrow$" Sei also

$(v_1, v_2, \ldots, v_n)$

top. sort von G. Falls

Kreis

$(v_{i_0}, v_{i_1}, \ldots, v_{i_0})$,

dann existiert eine

Kante der Art $v_{i_j} \to v_{i_k}$ mit $i < j$.

„⇐" Habe also $\mathcal{G}$ keinen Knoten.

Dann gibt es Knoten u mit $Egrad(u) = 0$. Also .

Widerspruch! Nehme irgendeinen Knoten, u_1. $E_{gr}(u_1) = 0$, dann ✓.

Sonst , nehme u_2 her.

$E\text{-}grad(u_2) = 0$ ✓, sonst an u_3

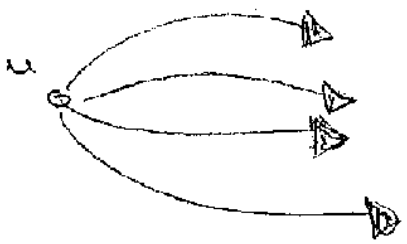
Falls $E\text{-}grad(u_3) = 0$, dann ✓.

Sonst

Können so weiter. Spätestens u_n hat $E\text{-}grad(u_n) = 0$, da sonst Kreis.

Also: Haben u mit $E\text{-}\deg(u) = 0$.

Also im $\mathcal{G}$ gibt es so aus:



$v_1 = u$ erste Knoten der Lösung.

Lösche u aus $\mathcal{G}$. Hier wieder

Knoten u' mit $E\text{-}\deg(u') = 0$

im neuen Graphen, wegen Kantenfreiheit.

$v_2 = u'$ zweite Knoten der Lösung.

Immer so weiter geht top Fort.

Formal: Induktion über $|V|$. □

Algorithmus (Top-Sort)

Eingabe: $G = (V, E)$ beliebiger gerichteter Graph, $V = \{1, \ldots, n\}$.

Ausgabe: Array $v[1 \ldots n]$, so daß $(v[1], v[2], \ldots, v[n])$ eine top. Sort. darstellt, falls G kreisfrei. Andernfalls Meldung, daß G Kreis hat.

Vorgehensweise:

1. Suche Knoten u mit $E\text{-}Grad(u) = 0$.

   Falls nicht existent $\Rightarrow$ Kreis.

   Falls u existiert $\Rightarrow$ u aus G löschen.
   Dazu: Kanten $u \to v$ löschen.

2. Suche Knoten u in neuem Graphen mit $E\text{-}Grad(u) = 0$

   $\vdots$ wie oben

Wie findet man u gute

1. Fall: Falls Adj-Matrix,

$A = (a_{v_1, v_2})$; dann alle

$a_{v_1, u} = 0$ für $v_1 \in V$.

2. Fall: 2 in Adj-Listen, gegeben.

Jedesmal Adj-Listen (!)

durchsuchen und zählen

ob ein Knoten u nicht vorkommt.

Ein solches u hat E-Grad(u) = 0.

Dann Adj[u] = nil setzen

Dazu ein array A mit neuen A[u] = 1 ⇔ u kommt vor!

Schleife mit n Durchläufen,

wenn kein u gefunden wird

Ausgabe Kreis, sonst beim i-ten

Lauf v[i] = u. □

Verbessere die Suche nach u mit

$E\text{-}Grad(u) = 0$:

0. Ermittle array (E)-Grad$[1 \ldots m]$,
   mit der Eingangsgrade.

1. Suche u mit $E\text{-}Grad[u] = 0$
   $v[1] = u$, Kreis falls nicht existent.
   Für alle $v \in Adj[u]$
   $E\text{-}Grad[v] = E\text{-}Grad[v] - 1$.

   ⋮

1. Weitere Verbesserung: Knoten u
   mit $E\text{-}Grad[u] = 0$ gleich
   in Datenstrukturen (etwa
   Schlange) speichern.

Algorithmus (Verbesserte Top Sort)

Eingabe: $G = (V, E)$ beliebiger gerichteter Graph, in Adj-Listen Dstg. bei $V = \{1, \ldots, n\}$.

Ausgabe: Wie bei Top Sort.

Weitere Datenstrukturen:

Array E-Grad[1...n] // Aktuelle
// Eingangsgrade

Schlange Q // Knoten mit
// E-Grad 0.
// Q kann auch Liste sein.

1\. E-Grad[v] auf 0, Q = $\emptyset$ initialisieren.

for each $v \in V$ {

Gehe Adj[v] durch,

zähle für jedes gefundene

u E-Grad[u] := E-Grad[u] + 1.

⋮

2. for each $u \in V$ {
   if E-grad[u] = 0 {
      Q = Q ∪ {u}
   }
}

3. for i = 1 to n {
   if Q = ∅ {
      output "Kreis"; exit
   }

4. v[i] = Q[head];
   Lösche Q[head] aus Q;

5. For each u ∈ Adj[v[i]] {
      E-grad[u] = E-grad[u] - 1;
      if E-grad[u] = 0 {
         Q = Q ∪ {u}
      }
   }
}

□