RUB

Vorlesung Datenstrukturen

Graphdurchläufe

Maike Buchin

22. und 27.6.2017

Graphexploration

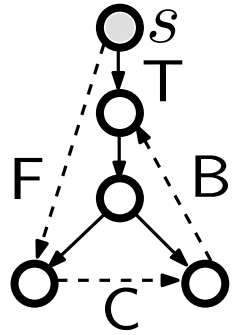
Motivation: Für viele Zwecke will man den gesamten Graphen durchlaufen, zB. um festzustellen ob er (stark) zusammenhängt.

Wir betrachten zwei grundlegende **Erkundungsstrategien**, bei denen jede Kante genau einmal angesehen wird:

- Breitensuche
- Tiefensuche

Idee beider Suchen: starte an einem beliebigen Knoten s (der Wurzel) und besuche von dort alle Kanten. Gehe dabei zuerst in die Breite bzw. Tiefe.

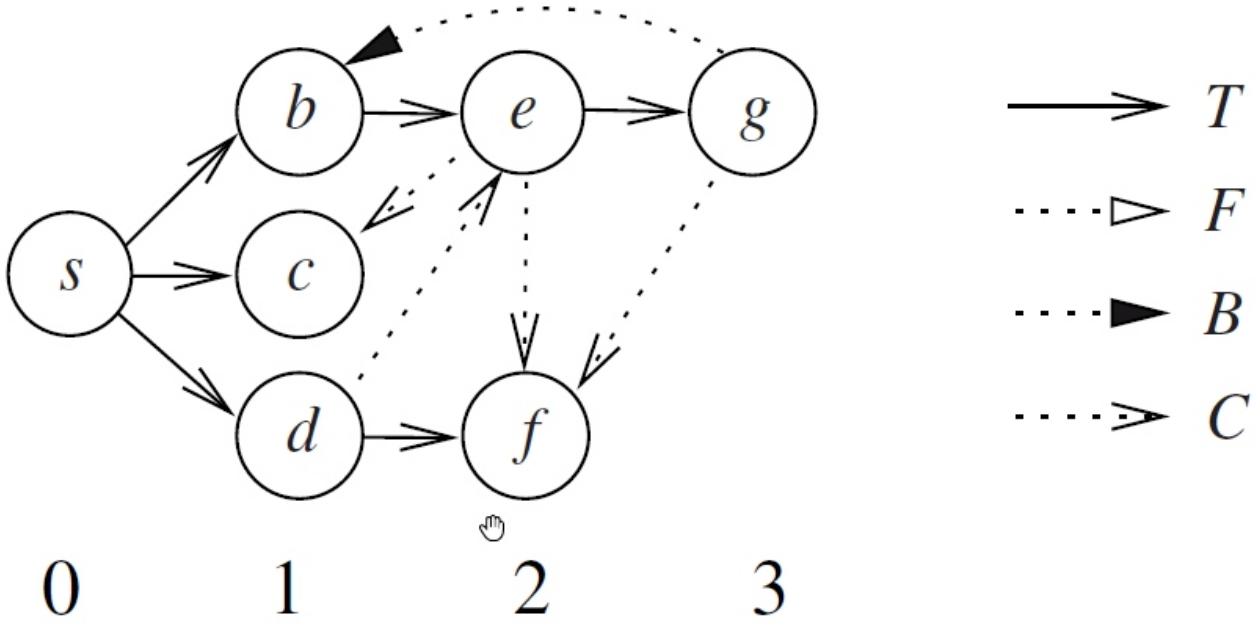
Beide Suchen konstruieren **gerichtete Wälder** (bzw Baum), und unterteilen die Kanten bzgl. diesem in 4 Klassen: Baum-(T), Forwärts-(F), Rückwärts-(B), und Querkanten (C).



9.1 Breitensuche

Breitensuche (BFS, breadth first search) erkundet den Graphen **Schicht-für-Schicht**. Startknoten s bildet Schicht 0, Nachbarn von s bilden Schicht 1 usw. D.h. Schicht $i+1$ wird gebildet von Knoten mit Nachbarn in Schicht i, aber keiner früheren Schicht. Ist Knoten v in Schicht i, sagen wir auch er hat *Tiefe* i bzw. *Abstand* i zu s.

Beispiel:



9.1 Breitensuche

RUB

Pseudocode:

Function $bfs(s : NodeId) : (NodeArray\ \textbf{of}\ 1..n) \times (NodeArray\ \textbf{of}\ NodeId)$
- $d = \langle \infty, \ldots, \infty \rangle : NodeArray\ \textbf{of}\ NodeId$ // Abstand von der Wurzel
- $parent = \langle \bot, \ldots, \bot \rangle : NodeArray\ \textbf{of}\ NodeId$
- $d[s] := 0$
- $parent[s] := s$ // Schleife: signalisiert „Wurzel“
- $Q = \langle s \rangle : Set\ \textbf{of}\ NodeId$ // aktuelle Schicht des BFS-Baums
- $Q' = \langle \rangle : Set\ \textbf{of}\ NodeId$ // nächste Schicht des BFS-Baums
- **for** $\ell := 0$ **to** ∞ **while** $Q \neq \langle \rangle$ **do** // erkunde Schicht für Schicht
  - **invariant** Q enthält alle Knoten mit Abstand ℓ von s
  - **foreach** $u \in Q$ **do**
    - **foreach** $(u,v) \in E$ **do** // durchlaufe Ausgangskanten von u
      - **if** $parent[v] = \bot$ **then** // bislang nicht erreichter Knoten gefunden
        - $Q' := Q' \cup \{v\}$ // für nächste Schicht merken
        - $d[v] := \ell + 1$
        - $parent[v] := u$ // aktualisiere BFS-Baum
  - $(Q, Q') := (Q', \langle \rangle)$ // schalte auf nächste Schicht um
- **return** $(d, parent)$ // der BFS-Baum ist jetzt $\{(v,w) : w \in V, v = parent[w]\}$

Konstruierter **BFS-Baum**: $\{(v,w) : w \in V, v = p[w]\}$

Laufzeit: $O(m+n)$ verwendet man Adjanzenzarray/-listen.

9.1 Breitensuche

RUB

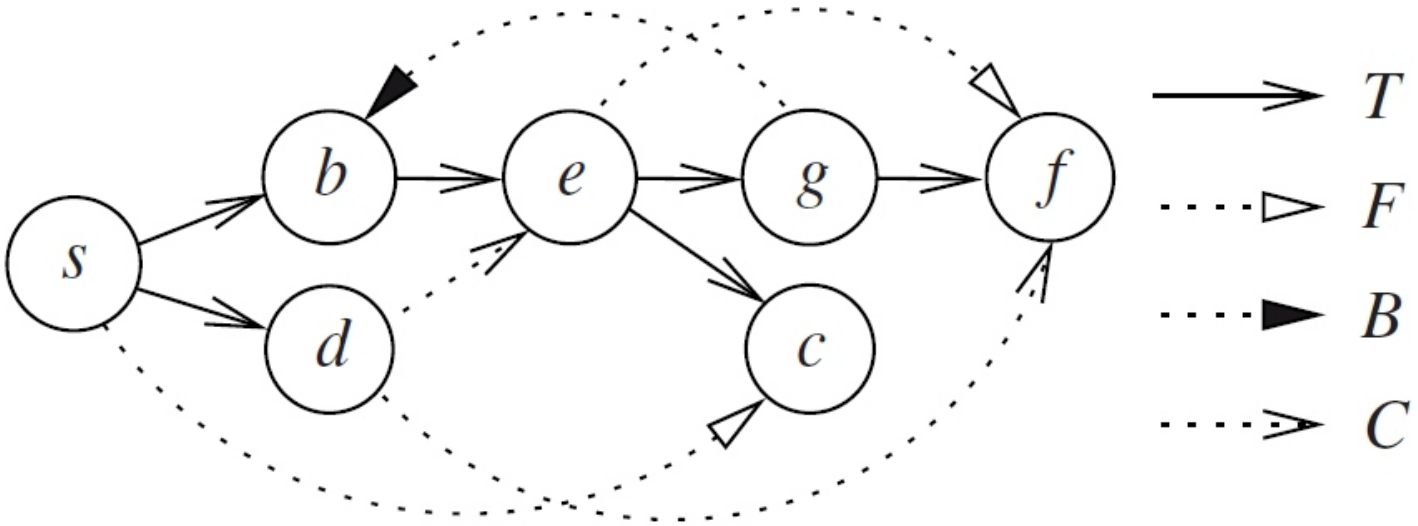
Eigenschaften:

- Kanten (Knotentiefe) im BFS-Baum entsprechen kürzesten Wegen (Abständen) bzgl. der Anzahl Kanten zum Startknoten s.
- BFS-Baum ist ein Spannbaum (Baum auf allen Knoten); Querkanten schließen Kreise

9.2 Tiefensuche

Tiefensuche (DFS, depth first search) geht zunächst in die Tiefe. Dabei entstehen zwar nicht die Schichten der BFS, aber es ergibt sich ebenfalls eine sehr nützliche Strategie.

Beispiel:



RUB

Algorithmenschema mit Unterprogrammen *init*, *root*, *traverseTreeEdge*, *traverseNonTreeEdge*, *backtrack*

Tiefensuche in einem gerichteten Graphen $G = (V, E)$
entferne alle Knotenmarkierungen
init
foreach $s \in V$ **do**
  if s ist nicht markiert **then**
    root(s) // Mache s zu einer Wurzel
    DFS(s, s) // Baue neuen DFS-Baum mit Wurzel s

Procedure *DFS*$(u, v : NodeId)$ // erkunde v, von u kommend.
  markiere v als *aktiv*
  foreach $(v, w) \in E$ **do**
    if w ist markiert **then** *traverseNonTreeEdge*(v, w) // w schon vorher erreicht
    else *traverseTreeEdge*(v, w) // w vorher nicht erreicht
      DFS(v, w)
  backtrack(u, v) // Aufräumen; Zusammenfassen und Rückgabe von Daten
  markiere v als *beendet*
  return // Rücksprung

DFS-Baum: TreeEdges. **Laufzeit:** $O(m + n)$

9.2 Tiefensuche

RUB

Tiefensuche arbeitet mit **Knotenmarkierungen**: *unmarked*, *active*, *finished*.

Zu jeder Zeit der Ausführung gibt es mehrere **aktive Knoten**.
Genauer: es gibt Knoten $v_1, \ldots, v_k$ mit aktiven Aufrufen DFS(v_1, v_1), DFS(v_1, v_2), ...,DFS(v_{k-1}, v_k).
Rekursionsstack enthält $\langle (v_1, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \rangle$.
Der aktuelle Knoten ist v_k.

9.2 Tiefensuche

9.2.1 DFS-Nummern, Endezeiten und topologisches Sortieren

Nummerierung der Knoten (Ankunfts- & Endezeit) erlaubt:

- Klassifizierung der Kanten
- Test auf Azyklizität
- Topologisches Sortieren

Berechne zwei Arrays: dfsNum, finNum.

Verwende dazu globale Zähler: dfsPos, finPos.

- initialisiere beide Arrays zu 0, und beide Zähler als 1.
- bei neuer Wurzel v oder Baumkante zu v, wird dfsNum$[v]$ auf dfsPos gesetzt und dfsPos um 1 erhöht.
- wenn ein Knoten v beendet wird, wird finNum[v] $=$ finPos gesetzt und finPos um 1 erhöht.

Also:

$$
\begin{array}{ll}
\mathit{init}: & \mathit{dfsPos} = 1 : 1..n; \quad \mathit{finPos} = 1 : 1..n \\
\mathit{root}(s): & \mathit{dfsNum}[s] := \mathit{dfsPos}{+}{+} \\
\mathit{traverseTreeEdge}(v,w): & \mathit{dfsNum}[w] := \mathit{dfsPos}{+}{+} \\
\mathit{backtrack}(u,v): & \mathit{finNum}[v] := \mathit{finPos}{+}{+}
\end{array}
$$

9.2 Tiefensuche

9.2.1 DFS-Nummern, Endezeiten und topologisches Sortieren

Wir nutzen dfsNum, um Knoten zu ordnen:

$$u \prec v \quad :\Leftrightarrow \quad \mathsf{dfsNum}[u] < \mathsf{dfsNum}[v]$$

dfsNum und finNum kodieren wichtige Eigenschaften, zB. die **Klassifizierung eines Knoten** v:

- unmarkiert: $\mathsf{dfsNum}[v] = 0$
- aktiv: $\mathsf{dfsNum}[v] > 0$ und $\mathsf{finNum}[v] = 0$
- beendet: $\mathsf{dfsNum}[v] > 0$ und $\mathsf{finNum}[v] > 0$

Lemma: Die Knoten des Rekursionsstack sind bzgl. $\prec$ geordnet.

Beweis: Nach jeder Zuweisung von $\mathsf{dfsNum}[v]$ wird der Zähler dfsPos um 1 erhöht. Also gilt bei jeder Aktivierung, dass der aktuelle Knoten die bislang größte dfsNum hat.

9.2 Tiefensuche

RUB

9.2.1 DFS-Nummern, Endezeiten und topologisches Sortieren

Klassifizierung einer Kante (v, w)

Typ	$dfsNum[v] < dfsNum[w]$	$finNum[w] < finNum[v]$	Markierung von w
Baumkante	ja	ja	unmarkiert
Vorwärtskante	ja	ja	*beendet*
Rückwärtskante	nein	nein	*aktiv*
Querkante	nein	ja	*beendet*

Lemma. Die folgenden Eigenschaften sind äquivalent:

- G ist ein azyklischer gerichteter Graph (DAG).
- Tiefensuche auf G erzeugt keine Rückwärtskanten.
- Für jede Kante (v, w) von G gilt finNum $[v] >$ finNum$[w]$.

Eine **Topologische Sortierung** eines DAG ist eine lineare Anordnung der Knoten, so dass alle Kanten von kleineren zu größeren Knoten gehen.

Folgerung: Aus obigem Lemma folgt, dass die Sortierung nach steigenden Endenummern eine topologische Sortierung ist.

9.2.2 Starke Zusammenhangskomponenten

RUB

Erinnerung: Zwei Knoten in einem gerichteten Graphen gehören zu einer starken Zusammenhangskomponente, wenn jeder von beiden vom jeweils anderen aus erreichbar ist.

In ungerichteten Graphen lassen sich Zusammenhangskomponenten einfach mit BFS oder DFS bestimmen.

Frage: Wie bestimmt man Zusammenhangskomponenten in gerichteten Graphen?

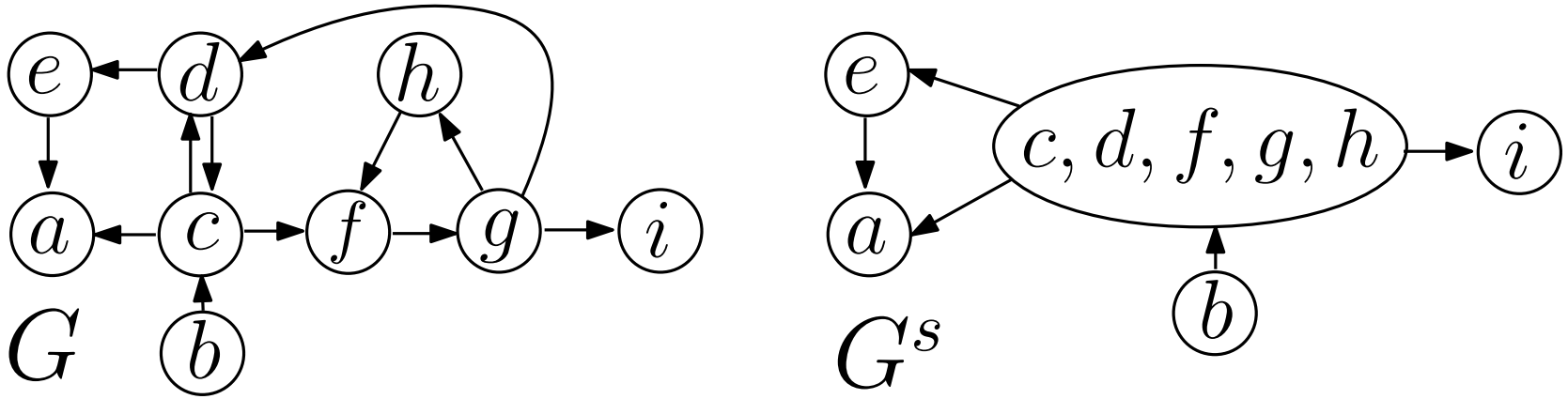
9.2.2 Starke Zusammenhangskomponenten

Vergröberter Graph G^s

Knoten V^s = (starke Zusammenhangs-)Komponenten von G

Kante $(C, D) \in E^s \Leftrightarrow \exists u \in C, v \in D : (u, v) \in E$

Beispiel:



Beobachtung: Knotenmengen verschiedener Komponenten sind disjunkt.

Lemma: Der vergröberte Graph G^s zu G ist azyklisch.

Beweis: Angenommen, es gäbe einen Kreis $C_0, \ldots, C_k$ mit $C_0 = C_k$ und $k \geq 2$ in G^s. Dann gibt es für $0 \leq i < k$ eine Kante von C_i nach C_{i+1}. Wegen des starken Zusammenhangs innerhalb von Komponenten lassen sich diese zu einem Kreis in G ergänzen. ↯

9.2.2 Starke Zusammenhangskomponenten

RUB

Ziel: Ein $O(m+n)$ Zeit Algorithmus zur Bestimmung der starken Zshgkomponenten sowie einer top. Sortierung dieser basierend auf DFS.

Idee: Füge nach und nach die Kanten ein und beobachte, wie sich der aktuelle Graph G_C und G_C^s verändern.

Wie verändern sich diese wenn eine Kante e hinzugefügt wird?

1. Fall: Beide Endknoten von e gehören zur gleichen Komponente von $G_C \to G_C, G_C^s$ ändern sich nicht

2. Fall: Kante e verbindet verschiedene Komponenten von G_C, schließt aber keinen Kreis $\to$ Komponenten ändern sich nicht

3. Fall: Kante e verbindet verschiedene Komponenten von G_C und schließt mind. einen Kreis $\to$ Komponenten auf geschlossenen Kreisen werden vereinigt

9.2.2 Starke Zusammenhangskomponenten

RUB

Ein effizienter Algorithmus ergibt sich bei Einfügen der Kanten in der Reihenfolge einer DFS.

Wir unterscheiden drei **Arten von Komponenten:**

- *unerreicht*: unmarkierte Knoten mit Ein- & Ausgangsgrad 0 bilden isolierte Komponente in G_C^s
- *offen*: Komponente aus mind. 1 aktiven und beendeten Knoten
- *geschlossen*: Komponente aus nur beendeten Knoten

Wir nennen Knoten ebenso wie die Komponente in der sie sind.

Für jede Komponente nennen wir den Knoten mit der kleinsten DFS-Nummer ihren Repräsentanten.

9.2.2 Starke Zusammenhangskomponenten

RUB

Invarianten

(1) Alle Kanten, die aus geschlossenen Knoten herausführen, führen zu geschlossenen Knoten.

(2) Sei P der Weg von der (aktuellen) Wurzel zum aktuellen Knoten, welcher die Repräsentanten aller offenen Knoten $S_1, \ldots, S_k$ enthält. Dann gilt:

- für $2 \leq i \leq k$: auf P gibt es eine Baumkante von S_{i-1} nach S_i, und dies ist die einzige Kante nach S_i in G_C^s
- für $1 \leq i \leq j \leq k$ gibt es keine Kante von S_j nach S_i
- für $1 \leq i \leq j \leq k$ alle Knoten in S_j sind vom Repräsentanten von S_i aus erreichbar

(3) Sortiert man die offenen Knoten entsprechend ihrer DFS-Nummern, dann teilen die Repräsentanten diese genau entsprechend der Komponenten auf

9.2.2 Starke Zusammenhangskomponenten

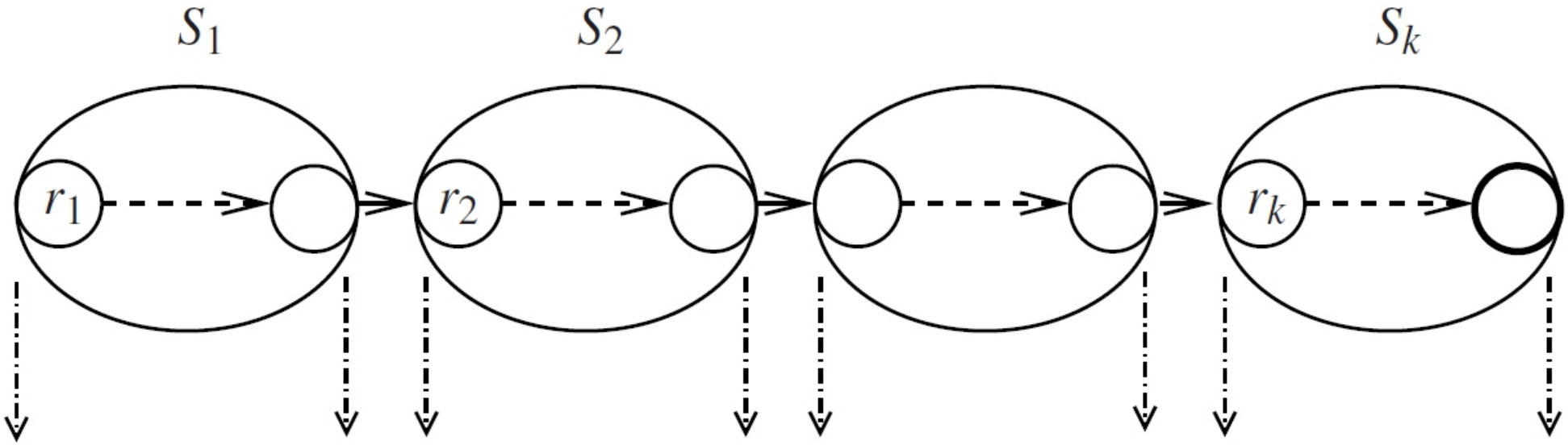
RUB

Folgerungen aus den Invarianten

Lemma: Jede geschlossene Komponente von G_c ist auch Komponente von G.
Beweis: folgt aus Invariante 1.

Darstellung der offenen Komponenten von G_C ergibt sich aus den Invarianten 2,3:

- Folge oNodes: offene Knoten mit aufsteigenden DFS-Nr
- Folge oReps: offene Repräsentanten mit aufsteig. DFS-Nr



RUB

Beweis der Invarianten und Vorgehen des Algorithmus

Wir überlegen uns, dass alle möglichen Aktionen während der DFS die Invarianten aufrecht erhalten

- vor Beginn der DFS ist kein Knoten markiert, keine Kante traversiert, G_C enthält nur unerreichte Komponenten; oNodes, oReps sind leer
- wird eine neue Wurzel s markiert, sind vorher alle markierten Knoten beendet; also keine offenen Komponenten und oNodes, oReps sind leer; offene Komponente $\{s\}$ wird erzeugt und s in oNodes, oReps eingefügt
- wird eine Baumkante (v, w) traversiert, wird w als aktiv markiert, die offene Komponente $\{w\}$ wird erzeugt und w in oNodes, oReps eingefügt
- wird eine Nicht-Baumkante (v, w) traversiert, so ist w bereits markiert. Ist $\{w\}$ geschlossen, ändern sich nach obigem Lemma die Komp. nicht. Ist $\{w\}$ offen, dann werden durch Einfügen von (v, w) Komp. $S_i, \ldots, S_k$ verschmolzen. Lösche alle oReps r mit $\text{dfsNum}[r] > \text{dfsNum}[w]$
- wird ein Knoten v durch backtrack (u, v) beendet, wird dadurch eine Komp. geschlossen falls v ihr letzter offener Knoten, also Repräsentant, war. Falls ja, lösche v aus oReps, und alle Knoten bis v aus oNodes.

9.2.2 Starke Zusammenhangskomponenten

RUB

Geeignete Unterprogramme

```
init:
    component : NodeArray of NodeId                    // Repräsentanten
    oReps = ⟨⟩ : Stack of NodeId                        // Repräsentanten offener Komponenten
    oNodes = ⟨⟩ : Stack of NodeId                       // alle Knoten in offenen Komponenten

root(w) oder traverseTreeEdge(v,w):
    oReps.push(w)                                       // neue offene
    oNodes.push(w)                                      // Komponente

traverseNonTreeEdge(v,w):
    if w ∈ oNodes then
        while w ≺ oReps.top do oReps.pop                // vereinige Komponenten auf Kreis

backtrack(u,v):
    if v = oReps.top then
        oReps.pop                                       // Komponente als
        repeat                                          // geschlossen erkannt,
            w := oNodes.pop                             // Behandlung abschließen
            component[w] := v
        until w = v
```

9.2.2 Starke Zusammenhangskomponenten

RUB

Satz: Die obigen Unterprogramme für das DFS-Algorithmenschema liefern einen Algorithmus, der in Zeit $O(m+n)$ die starken Zusammenhangskomponenten eines gerichteten Graphen berechnet.