1. Übung Algorithmentechnik

Lehrstuhl für Algorithmik I
Institut für Theoretische Informatik
Universität Karlsruhe (TH)

04.11.2008

Aufgabe 1

Aufgabe (Amortisierte Analyse)

- Modellieren Sie eine Queue mit zwei Stacks so, dass die amortisierten Kosten von DEQUEUE und ENQEUE jeweils in $\mathcal{O}(1)$ sind.
- Geben Sie die Operationen DEQUEUE und ENQUEUE in Pseudocode an und begründen Sie die amortisierten Kosten.

Aufgabe 1

- globale Datenstruktur: zwei Stacks E und D, die initial leer sind
- E wird für ENQUEUE benutzt
- D wird für DEQUEUE benutzt
- betrachten Folge von n ENQUEUE- und DEQUEUE-Operationen für Analyse

Aufgabe 1

Algorithmus 1 : ENQUEUE(x)

1 PUSH(E,x) $\mathcal{O}(1)$

Aufwand 1 $= \mathcal{O}(1)$

Algorithmus 2 : DEQUEUE()

```
1 wenn D = ∅ dann
2     solange E ≠ ∅ tue
3         Push(D,Pop(E))
4 pop(D)                                   O(1)
```

Aufwand 1 bzw. $2|E| + 1 = \mathcal{O}(|E|)$

Korrektheit

- betrachte Folge von INSERT und DEQUEUE $x_1 x_2 \cdots x_k D_1 x_{k+1} \cdots x_r D_k$
- zu Beginn seien D, E leer
- vor D_1 liegen $x_1 x_2 \cdots x_k$ in umgekehrter Reihenfolge des Einfügens auf E
- nach D_1 ist E leer und auf D liegen $x_k \cdots x_1$ in Reihenfolge des Einfügens
- $D_1, \ldots, D_k$ entfernen jeweils $x_1, \ldots, x_k$ in der Reihenfolge des Einfügens
- d.h. D_i entfernt x_i
- vor D_{k+1} liegen $x_{k+1} \cdots x_s$ auf E und D ist leer
- nach D_{k+1} liegen $x_s \cdots x_{k+1}$ auf D und $D_{k+1}, \ldots, D_s$ entfernen jeweils $x_{k+1}, \ldots, x_s$ ...

Ganzheitsmethode

- ≫ wenn DEQUEUE Aufwand von $|E|$ verursacht, dann müssen vorher $|E|$ ENQUEUE-Operationen stattgefunden haben
- ≫ Gesamtaufwand für DEQUEUE-Operationen ist höchstens so groß wie Gesamtaufwand für ENQUEUE-Operationen, d.h. $\mathcal{O}(n)$
- ⇒ Gesamtaufwand $T(n) = \mathcal{O}(n)$
- ≫ Aufwand pro Operation $T(n)/n = \mathcal{O}(1)$

Buchungsmethode

≫ definieren Kosten wie folgt

	tatsächliche Kosten	**amortisierte Kosten**
ENQUEUE	1	3
DEQUEUE	1, $\|E\|$	1

≫ Idee: beim Einfügen bezahlt jedes Element das Einfügen in E sofort und hinterlegt Kredit von 2 für Hinzufügen zu D (1 POP, 1 PUSH)

⇒ amortisierte Kosten in $\mathcal{O}(1)$

Potentialmethode

- sei D_i Datenstruktur nach i-ter Operation,
- Datenstruktur sei zu Beginn leer
- definiere Potential Φ : $D_i \mapsto \Phi(D_i) := 2|E|$
- $\Phi(D_O) = 0$, $\Phi(D_i) \geq \Phi(D_0)$
- im Folgenden $\hat{c}_i$: amortisierte Kosten, c_i reale Kosten der i-ten Operation
- dann gilt

$$\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Potentialmethode

- i-te Operation ist ENQUEUE mit $|E| := k$:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 2(k+1) - 2k = 3\end{aligned}$$

- i-te Operation ist DEQUEUE mit $|D| > 0$, $|E| := k$:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 2k - 2k = 1\end{aligned}$$

- i-te Operation ist DEQUEUE mit $|D| = 0$, $|E| := k$:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 2k + 1 + 0 - 2k = 1\end{aligned}$$

Aufgabe 2

Aufgabe (Untere Schranken)

- Gegeben sei eine Folge $(a_i)_{i=1}^n$ von n Elementen.
- Zeigen Sie, dass man mindestens $\Omega(\log n)$ Vergleiche benötigt, um den Index $i_{\max}$ des maximalen Elementes $a_{i_{\max}}$ zu bestimmen.

Untere Schranken

Problemstellung

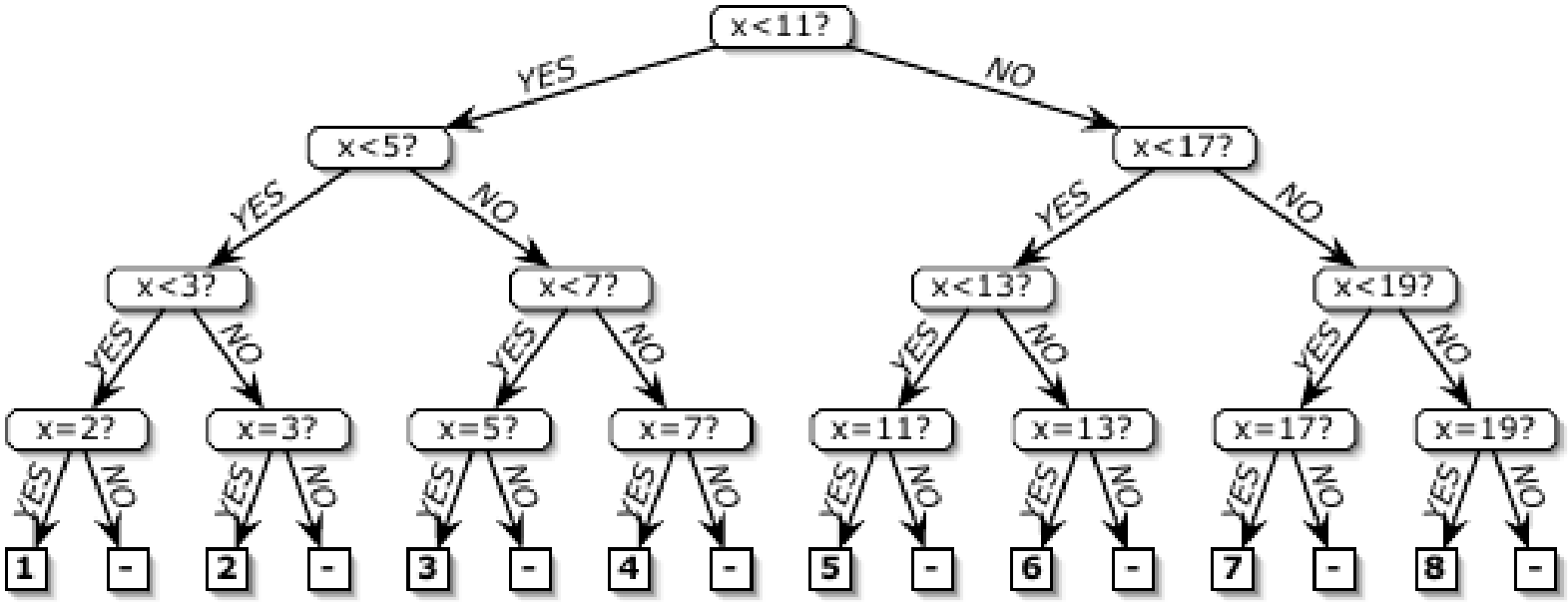
- sei Π ein Problem
- finde eine Funktion $\ell : \mathbb{N} \longrightarrow \mathbb{N}$ (untere Schranke für Π) mit folgender Eigenschaft:
- jeder Algorithmus $\mathcal{A}$, der Π löst, hat eine worst-case Komplexität $T_{\mathcal{A}}(n) \in \Omega(\ell(n))$,
- $\rightsquigarrow$ d.h. kein Algorithmus mit Komplexität $T_{\mathcal{A}}(n) \in o(\ell(n))$

- Problem: Aussage über alle Algorithmen schwierig
- Festlegung eines Algorithmen-Modells durch sog. Entscheidungsbäume
- Komplexitätsmaß ist Anzahl der Entscheidungen

Entscheidungsbäume als Algorithmen-Modell

- Algorithmus als Baum mit konstantem Maximalgrad
- nicht-uniformes Modell, d.h. unterschiedliche Bäume für Instanzen unterschiedlicher Größe
- Knoten ist "Orakel", das zu jeder Instanz einen Kindknoten ausgibt
- Blätter sind mit Ausgabe des Algorithmus annotiert
- Berechnung: Weg von der Wurzel zu einem Blatt (gemäß "Orakelspruch")
- Komplexität einer Berechnung: Länge des Weges
- Komplexität des Algorithmus: Höhe des Baumes

Entscheidungsbäume als Algorithmen-Modell



Entscheidungsbäume als Algorithmen-Modell

- sei h Höhe des Baumes und n die Anzahl der Blätter und k Maximalgrad
- dann gilt $n \leq k^h$, d.h. $h \geq \log_k n$
- Annahme: Algorithmus $\mathcal{A}$ habe n verschiedene Ausgaben

$\Rightarrow$ Baum hat mind. n Blätter

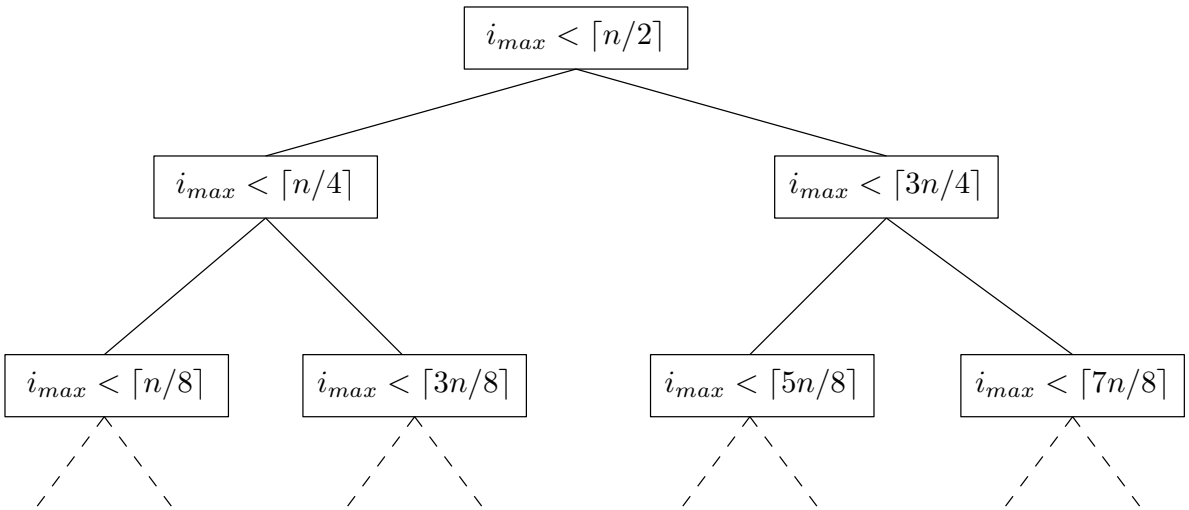
$\Rightarrow$ Höhe des Baumes ist mind. $\log_k n$

- Informationstheoretische Begründung: benutze Weg, um Ausgabe zu kodieren: Bei Höhe h und Maximalgrad k kann es höchstens $n = k^h$ verschiedene Ausgaben (Blätter) geben

$\Rightarrow$ $\log_k n$ ist (triviale) untere Schranke

Erster Ansatz

- ≫ Algorithmus für das Maximum-Problem hat n verschiedene Ausgaben
- ⇒ $\log_2 n$ ist untere Schranke, falls nur Vergleiche zugelassen sind
- ≫ diese Schranke ist scharf!
- ≫ verwende binären Suchbaum als Algorithmus



Anderer Ansatz

- Problem mit erstem Ansatz: ein Algorithmus, der Index des maximalen Elements nicht kennt, kann Orakelfrage nicht beantworten
- Modell ist zu mächtig
- Einschränkung: Jede Anfrage muss von der Form $x_i < x_j$ sein
- sei T Entscheidungsbaum in diesem Modell für das Maximum-Problem mit n Elementen
- wir konstruieren Eingabe, bei der der Algorithmus möglichst viele Vergleiche anstellen muss (Adversary-Argument)

Adversary Argument

- zunächst nehmen wir an, dass $x_i = i$ für alle i
- falls Algorithmus Anfrage $x_i < x_j$ stellt, markieren wir x_i
- Markierung bedeutet: Algorithmus weiß, dass x_i nicht das maximale Element ist
- x_n wird nicht markiert (da es das größte Element ist)
- Annahme: x_k ($k \neq n$) werde ebenfalls nicht markiert (d.h. Algorithmus macht weniger als $n-1$ Vergleiche)
- Algorithmus kann nicht unterscheiden, ob x_k oder x_n Maximum ist
- bei Ausgabe x_n wählen wir $x_k := n+1$
- sonst belassen wir die Folge (konsistent mit Anfragen)

$\Rightarrow$ mindestens $n-1$ Vergleiche nötig

Aufgabe 3

Aufgabe (Laufzeit rekursiver Funktionen)

Bestimmen Sie den größten ganzzahligen Wert a mit der Eigenschaft, dass ein Algorithmus mit der durch

$$T_B(n) = a \cdot T_B(n/4) + n^2$$

bestimmten Laufzeit im Θ-Kalkül asymptotisch schneller ist, als ein Algorithmus mit der durch

$$T_A(n) = 7 \cdot T_A(n/2) + n^2$$

bestimmten Laufzeit.

Master-Theorem

Theorem (Master-Theorem)

Seien $a \geq 1$ und $b > 1$ Konstanten, $f(n)$ eine Funktion in n und $T(n)$ über nichtnegative ganze Zahlen definiert durch

$$T(n) = a \cdot T(n/b) + f(n),^{a}$$

dann gelten

(i) $T(n) \in \Theta(f(n))$ *falls* $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ *und* $af(\frac{n}{b}) \leq cf(n)$ *für eine Konstante* $c < 1$ *und* $n \geq n_0$,

(ii) $T(n) \in \Theta(n^{\log_b a} \log n)$ *falls* $f(n) \in \Theta(n^{\log_b a})$,

(iii) $T(n) \in \Theta(n^{\log_b a})$ *falls* $f(n) \in O(n^{\log_b a - \varepsilon})$.

---

[a] Dabei stehen n/b für $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$.

Aufgabe 4

Aufgabe (Das Tiefen-Bestimmungs-Problem)

Entwerfen Sie eine Datenstruktur, die eine Menge $\mathcal{F}$ von disjunkten Bäumen verwaltet und folgende Operationen unterstützt.

- ≫ MAKE-TREE(v): erzeuge neuen Baum mit Wurzel v
- ≫ ATTACH(r, v): hänge Baum mit Wurzel r an Knoten v an
- ≫ FIND-DEPTH(v): bestimme Tiefe von v

Makeset

Algorithmus 3 : $\textsc{Makeset}(x)$ (#2)

Eingabe : Element x

1 $\textsc{Vor}[x] \leftarrow -1$

Die Laufzeit ist in $\mathcal{O}(1)$.

Weighted Union

Algorithmus 4 : $\textsc{Union}(i, j)$ (#3)

```
1 z ← VOR[i] + VOR[j]
2 wenn |VOR[i]| < |VOR[j]| dann
3     VOR[i] ← j und VOR[j] ← z
4 sonst
5     VOR[j] ← i und VOR[i] ← z
```

Die Laufzeit ist in $\mathcal{O}(1)$.

Find mit Pfadkompression

Algorithmus 5 : FIND(x) (#3)

Eingabe : Element x

Ausgabe : Menge, in der x enthalten ist

```
1 j ← x                                          Repräsentanten-Suche
2 solange VOR[j] > 0 tue
3     j ← VOR[j]
4 i ← x                                          Pfadkompression
5 solange VOR[i] > 0 tue
6     temp ← i
7     i ← VOR[i]
8     VOR[temp] ← j
9 Gib j aus
```

Die Laufzeit ist in $\mathcal{O}(\log n)$.

Modifikation von Union-Find

- Modifikation zur Verwaltung der Tiefe (Struktur der Bäume muss nicht durch Union-Find verwaltet werden)
- Annotation der Knoten mit Wert $d(v)$, der folgende Eigenschaft hat
- sei $v_k := \textsc{Find}(v_0)$
- sei $v_0, v_1, v_2, \ldots, v_k$ die Folge von Knoten, die $\textsc{Find}(v_0)$ von v_0 zur Wurzel v_k durchläuft
- dann soll gelten: Tiefe von v_0 ergibt sich als $\sum_{i=0}^{k} d(v_i)$
- $d(v_i)$ ist also Differenz zwischen Tiefe von v_i und v_{i+1} (direkter Vorgänger)

Korrektheit

- es gilt

$$d(v_i) := \textsc{Find-Depth}(v_i) - \textsc{Find-Depth}(v_{i+1}) \quad (i < k)$$
$$d(v_k) := \textsc{Find-Depth}(v_k)$$

- Summe ist Teleskop-Summe, d.h.

$$\sum_{i=0}^{k} d(v_i) = \textsc{Find-Depth}(v_0)$$

- stelle sicher, dass obige Gleichungen immer erfüllt sind

Makeset

Algorithmus 6 : MAKE-TREE(x)

Eingabe : Knoten x

Ausgabe : Neuer Baum mit Wurzel x

1 $d[x] \leftarrow 0$

2 $\text{VOR}[x] \leftarrow -1$

Korrektheit ist klar.

Algorithmus 7 : FIND(x)

Eingabe : Knoten x

Ausgabe : Das Paar (DEPTH, r) (Tiefe und Wurzel von x)

```
1  r ← x                                              Repräsentanten-Suche
2  DEPTH ← d[x]
3  solange VOR[r] > 0 tue
4      r ← VOR[r]
5      DEPTH ← DEPTH + d[r]
6  i ← x                                              Pfadkompression
7  PDEPTH ← 0
8  solange VOR[i] > 0 tue
9      t ← d[i]
10     d[i] ← DEPTH − PDEPTH − d[r]
11     PDEPTH ← PDEPTH + t
12     temp ← i
13     i ← VOR[i]
14     VOR[temp] ← r
```

Korrektheit

- sei $v_0, \ldots, v_k$ Folge, die Find durchläuft, d.h. $x = v_0$
- Zeile 9/10
  $d[v_i] \leftarrow \textsc{Depth} - \textsc{PDepth} - d[r]$
- $\textsc{Depth}$=$\textsc{Find-Depth}(x)$ und hat Wert $\sum_{s=0}^{k} d_{alt}[s]$
- $\textsc{PDepth}$ hat Wert $\sum_{s=0}^{i-1} d_{alt}[s]$ im i-ten Schritt

$$\begin{aligned} \Rightarrow d_{neu}[v_i] &= \sum_{s=i}^{k} d_{alt}[s] - d_{alt}[r] \\ &= \textsc{Find-Depth}(v_i) - d_{alt}[r] \\ &= \textsc{Find-Depth}(v_i) - \textsc{Find-Depth}(r) \end{aligned}$$

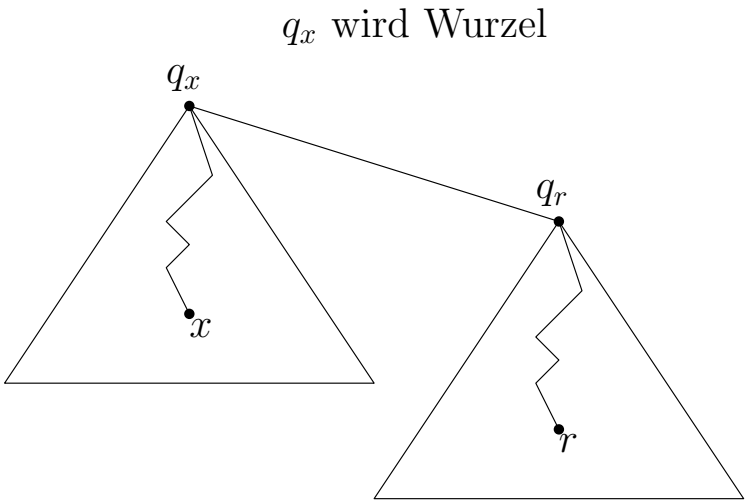
Algorithmus 8 : ATTACH(r, x) (modifiziertes Union)

Eingabe : Wurzel r eines Baumes und ein Knoten x

Ausgabe : Der Knoten r wird an x angehängt

```
1  (d_x, q_x) ← FIND(x)
2  (d_r, q_r) ← FIND(r)
3  z ← VOR[q_x] + VOR[q_r]
4  wenn |VOR[q_r]| < |VOR[q_x]| dann                    q_x wird Wurzel
5  |   VOR[q_r] ← q_x und VOR[q_x] ← z
6  |   d(q_r) ← d(q_r) + d_x + 1 − d(q_x)
7  sonst                                                q_r wird Wurzel
8  |   VOR[q_x] ← q_r und VOR[q_r] ← z
9  |   d(q_r) ← d(q_r) + d_x + 1
10 |   d(q_x) ← d(q_x) − d(q_r)
```

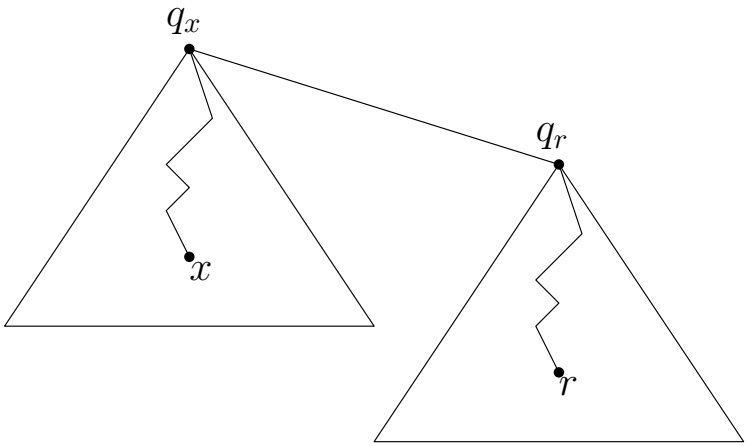
Korrektheit



$$d(q_r) \leftarrow d(q_r) + d_x + 1 - d(q_x)$$

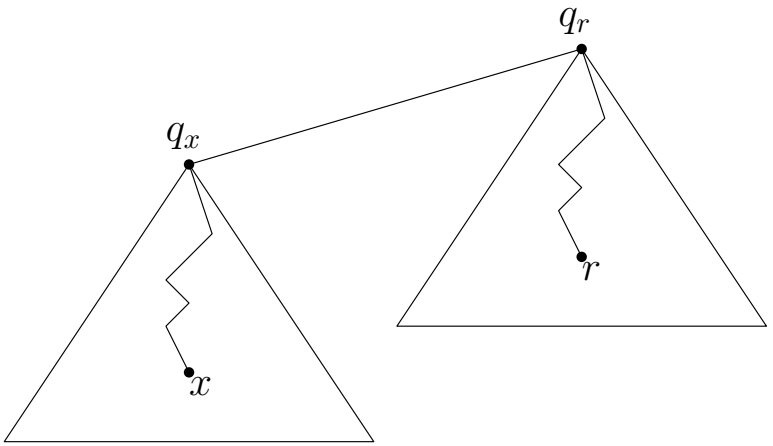
Korrektheit

q_x wird Wurzel



$$d(q_r) \leftarrow d(q_r) + d_x + 1 - d(q_x)$$

q_r wird Wurzel



$$d(q_r) \leftarrow d(q_r) + d_x + 1$$
$$d(q_x) \leftarrow d(q_x) - d(q_r)$$

Algorithmus 9 : FIND-DEPTH(x)

Eingabe : Knoten x

Ausgabe : Die Tiefe DEPTH(x) von x

1 $(d, r) \leftarrow \text{FIND}(i)$

2 $\text{DEPTH}(x) \leftarrow d$

Aufgabe 5

Aufgabe (Berechnung starker Zusammenhangskomponenten)

- Gegeben sei ein gerichteter Graph $D = (V, E)$.
- für $u, v \in V$ gelte $u \rightleftharpoons v$, wenn es einen gerichteten Weg von u nach v und einen gerichteten Weg von v nach u gibt
- Zeigen Sie, dass $\rightleftharpoons$ eine Äquivalenzrelation ist.
- Die Äquivalenzklassen zu $\rightleftharpoons$ heißen starke Zusammenhangskomponenten. Schreiben Sie einen Algorithmus in Pseudocode, der die starken Zusammenhangskomponenten findet.

Äquivalenzrelation

- $\gg$ Reflexivität: $u \rightleftharpoons u$, da Weg der Länge 0 existiert
- $\gg$ Symmetrie: $u \rightleftharpoons v$
- $\Rightarrow$ $v \rightleftharpoons u$, nach Definition
- $\gg$ Transitivität: $u \rightleftharpoons v$, $v \rightleftharpoons w$
- $\Rightarrow$ $u \rightleftharpoons w$: da Weg von u über v nach w und umgekehrt existiert

Bemerkung: $u \rightleftharpoons w$ gdw. u und v liegen auf einem gerichteten Kreis

Tiefensuche

Algorithmus 10 : DFS(G=(V,E))

```
1 für v ∈ V tue
2     π(w) ← Nil
3     Color[v] ← White
4 für v ∈ V tue
5     wenn Color[v] = White dann
6         DFS-Visit(v)
```

Algorithmus 11 : DFS-Visit(v, S)

```
1 Color[v] ← Gray
2 für (v, w) ∈ E tue
3     wenn Color[w] = White dann                    neuer Knoten
4         π(w) ← v
5         Mod-DFS-Visit(w, S)
6 Color[v] ← Black
```

Ineffizienter Algorithmus

Algorithmus 12 : STARKE-ZUSAMMENHANGSKOMPONENTEN-1

```
1 für v ∈ V tue
2     MAKESET(v)
3 für {v, w} ∈ 2^V tue
4     wenn FIND(v) ≠ FIND(w) dann
5         DFS-VISIT(v)
6         DFS-VISIT(w)
7         wenn w ∈ DFS-VISIT(v) und v ∈ DFS-VISIT(w) dann
8             UNION(FIND(v), FIND(w))
```

Aufwand $\mathcal{O}(n^3)$!

modifizierte Tiefensuche

Vorgehensweise

- verwalte Mengen von äquivalenten Knoten mit Union-Find
- falls Tiefensuche Kreis findet, können alle Knoten auf dem Kreis "kontrahiert" werden
- verwenden Stack S für Repräsentanten von Mengen äquivalenter Knoten mit der Eigenschaft, dass einer der Knoten auf dem aktuell offenen Pfad der Tiefensuche liegt

modifizierte Tiefensuche

Algorithmus 13 : STARKE-ZUSAMMENHANGSKOMPONENTEN

Daten : Stack S

1 **für** $v \in V$ **tue**
2 MAKESET(v)
3 $\pi(w) \leftarrow$ NIL
4 COLOR[FIND(v)] $\leftarrow$ WHITE
5 **für** $v \in V$ **tue**
6 **wenn** COLOR[FIND(v)] = WHITE **dann**
7 MOD-DFS-VISIT(v)

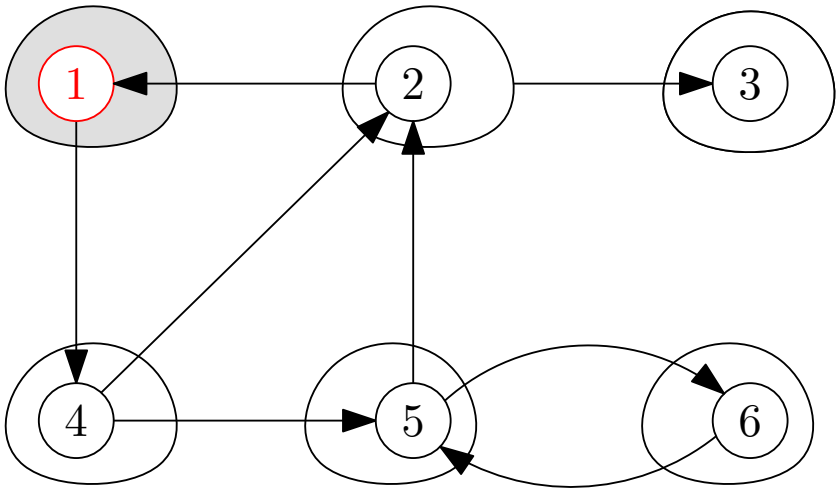
Algorithmus 14 : MOD-DFS-VISIT(v, S)

```
1  COLOR[FIND(v)] ← GRAY
2  wenn FIND(v) ≠ FIND(PEEK(S)) dann
3      PUSH(S, v)
4  für (v, w) ∈ E tue
5      wenn COLOR[FIND(w)] = WHITE dann                neuer Knoten
6          π(w) ← v
7          MOD-DFS-VISIT(w, S)
8      sonst wenn COLOR[FIND(w)] = GRAY dann   Kreis gefunden
9          KONTRAHIERE-KREIS(w, S)
10 wenn FIND(π(v)) ≠ FIND(PEEK(S)) dann        Stack aufräumen
11     COLOR[FIND(v)] ← BLACK
12     POP(S)
```

Algorithmus 15 : KONTRAHIERE-KREIS(w, S)

```
1 x ← POP(S)
2 solange FIND(w) ≠ FIND(x) tue
3 |   UNION(FIND(w), FIND(x))
4 |   x ← POP(S)
5 COLOR[FIND(w)] ← GREY
6 PUSH(S, w)
```

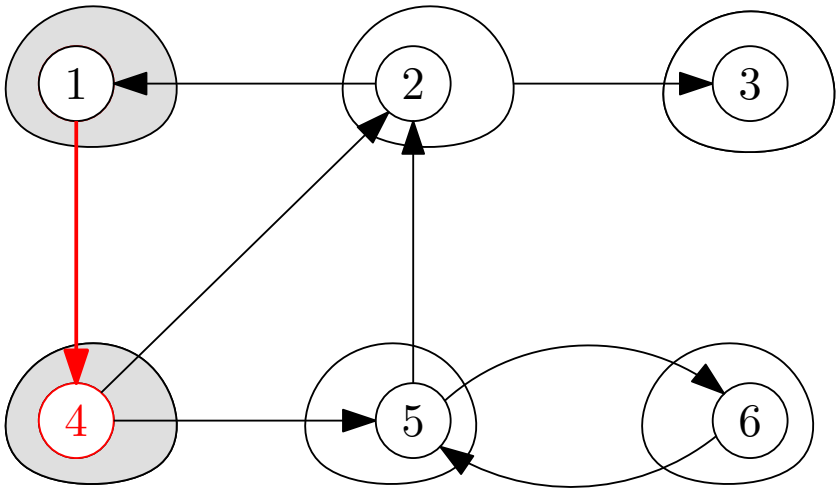
Beispiel



{1}

Stack

Beispiel

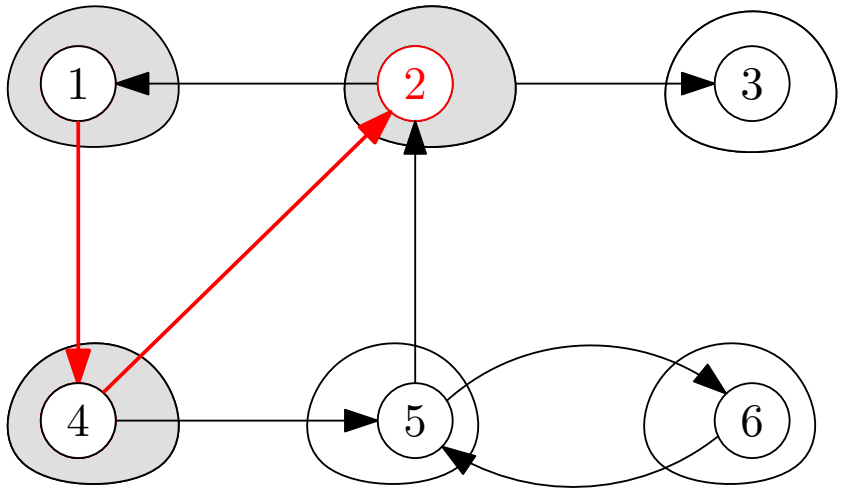


{4}

{1}

Stack

Beispiel



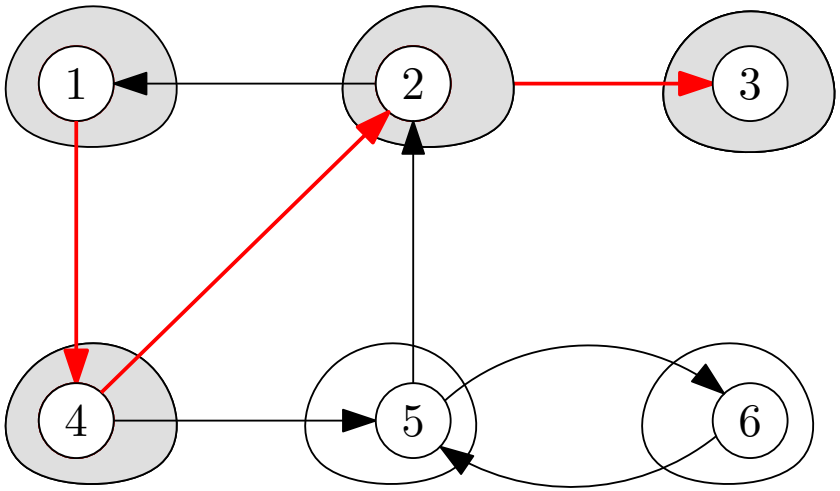
{2}

{4}

{1}

Stack

Beispiel



{3}

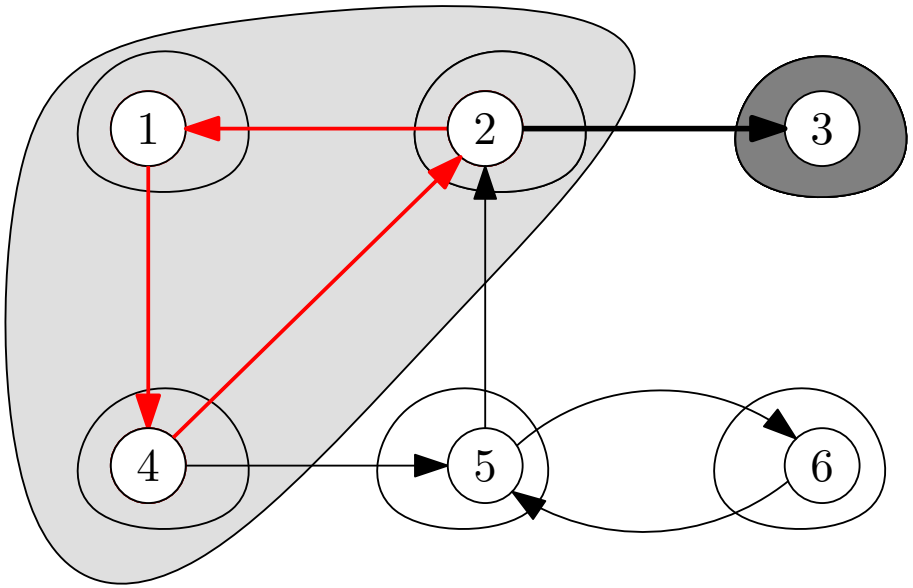
{2}

{4}

{1}

Stack

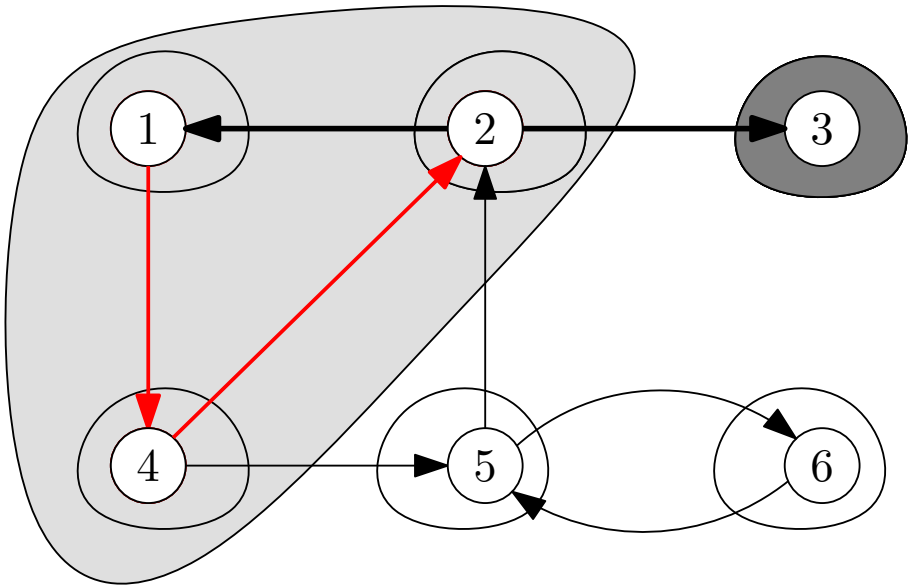
Beispiel



$\{1, 2, 4\}$

Stack

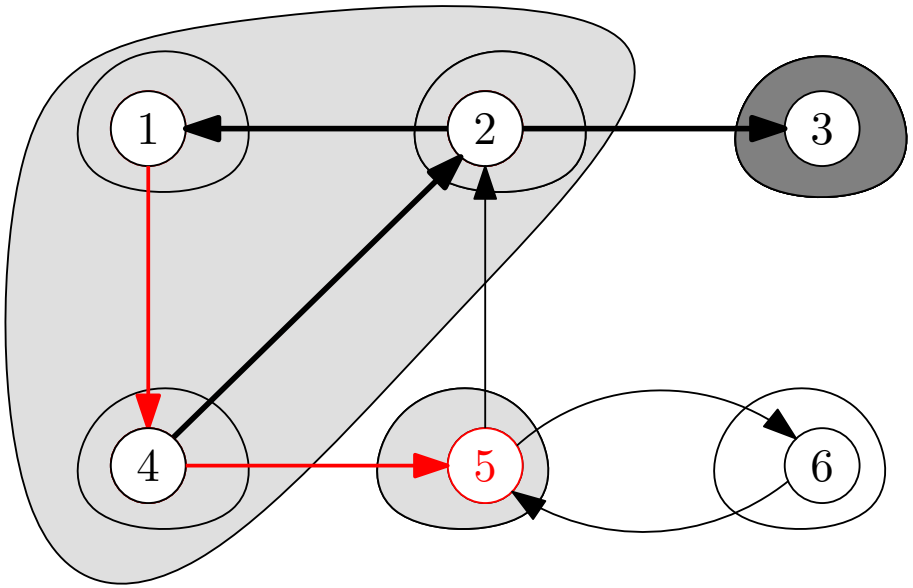
Beispiel



$\{1, 2, 4\}$

Stack

Beispiel

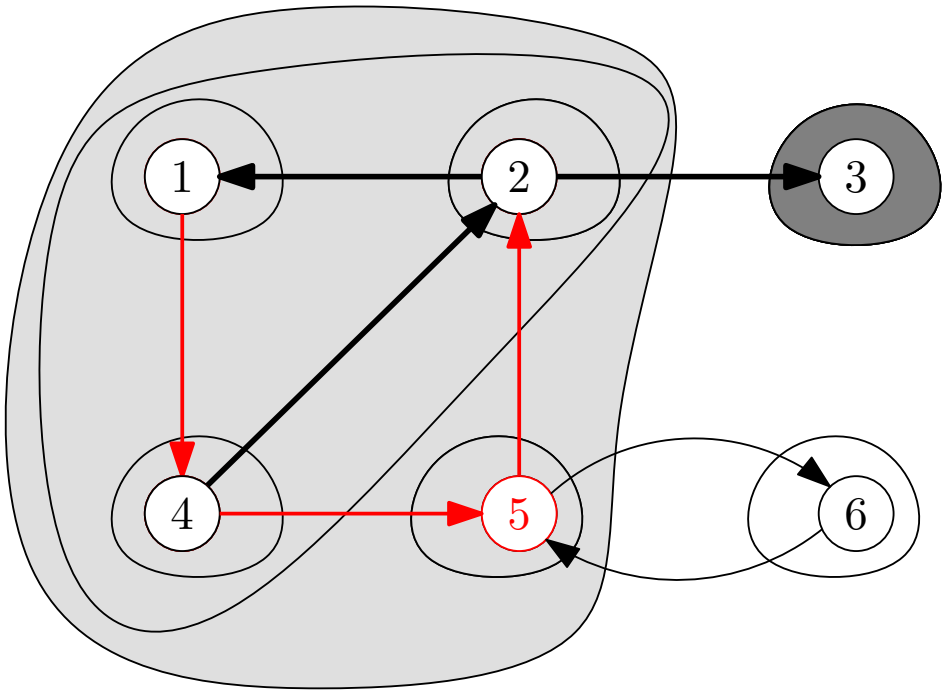


$\{5\}$

$\{1, 2, 4\}$

Stack

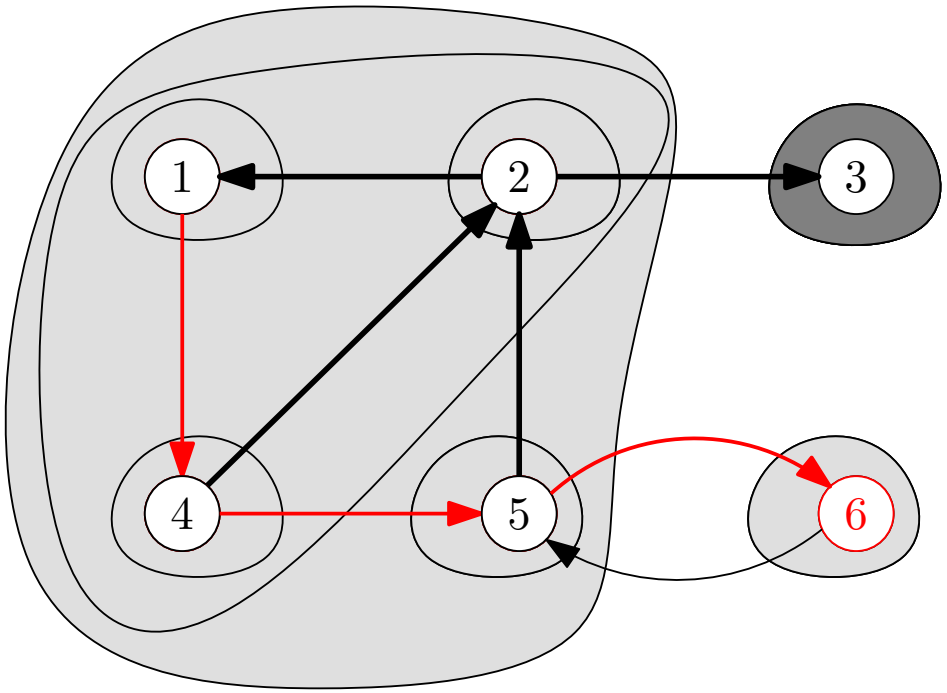
Beispiel



$\{1, 2, 4, 5\}$

Stack

Beispiel

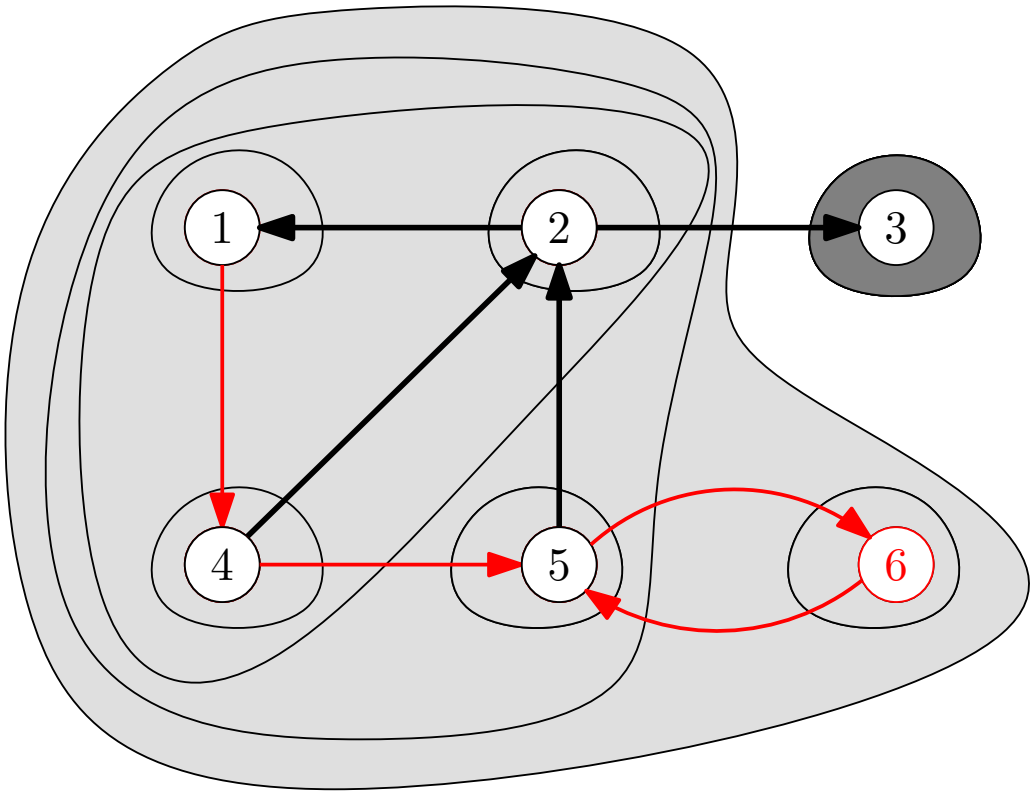


$\{6\}$

$\{1, 2, 4, 5\}$

Stack

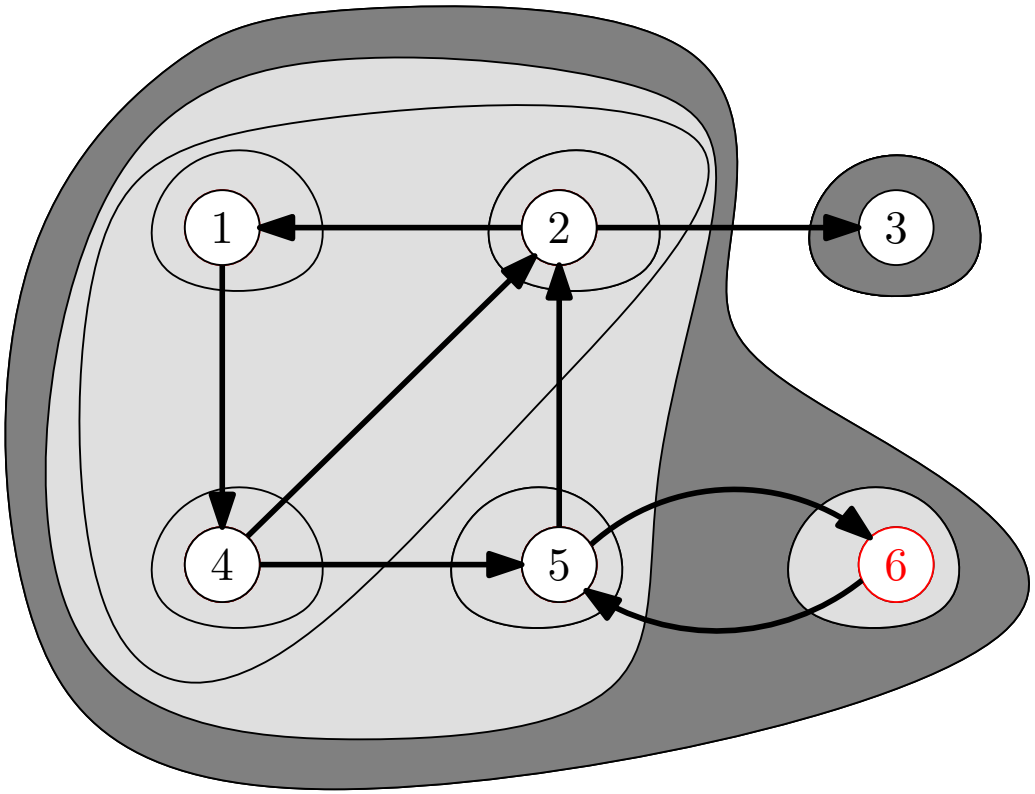
Beispiel



$\{1, 2, 4, 5, 6\}$

Stack

Beispiel



$\{1, 2, 4, 5, 6\}$

Stack

Korrektheit

- ≫ Algorithmus durchläuft Graph mit Tiefensuche
- ≫ sei $v_1, \ldots, v_k$ der aktuelle Tiefensuche-Pfad
- ⇒ Stackinhalt: $w_1, \ldots, w_k$ mit $\textsc{Find}(v_i) = \textsc{Find}(w_i)$
- ≫ d.h. Stack enthält zu jedem Zeitpunkt für jeden Knoten v auf dem aktuell offenen Pfad den Repräsentanten einer Menge, die v enthält
- ≫ sei C gerichteter Kreis in G
- ⇒ die Knoten auf C werden zu einer Menge zusammengefasst
- ≫ Menge wird erst schwarz gefärbt, wenn alle Knoten in der Menge abgesucht wurden