Karlsruhe Institute of Technology

Algorithmen zur Visualisierung von Graphen

Darstellung von Symmetrien und Inkrementelle Verfahren

INSTITUT FÜR THEORETISCHE INFORMATIK – LEHRSTUHL ALGORITHMIK I

MARCUS KRUG

bevor's losgeht

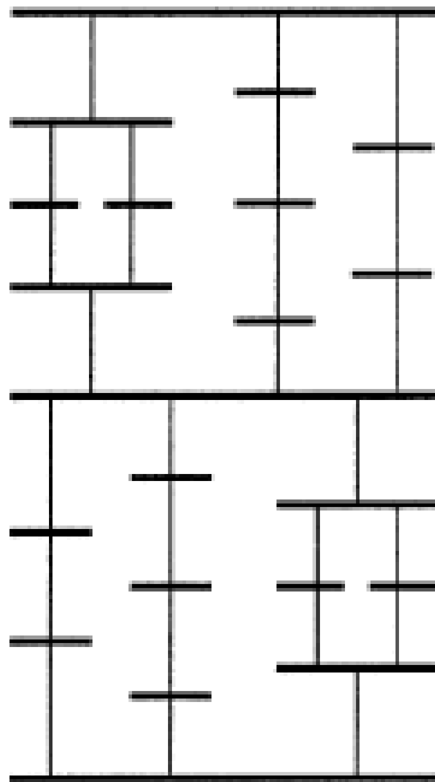
Vorschlag für Prüfungstermine

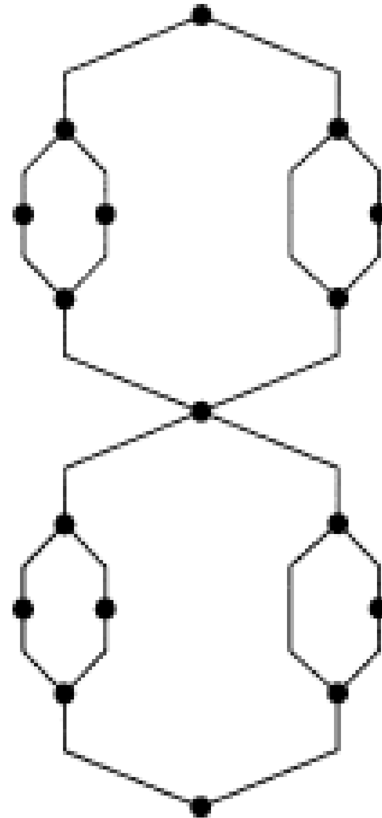
- Donnerstag, 1. März ab 9:00 Uhr
- Freitag, 30. März ab 9:00 Uhr

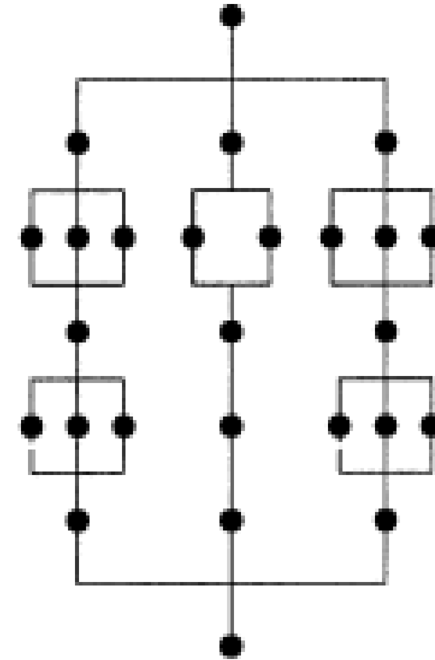
Evaluation

Institut für Theoretische Informatik
Lehrstuhl Algorithmik I

Darstellung von Symmetrien in Graphen







aus Hong, Eades, Lee ’00

Darstellung von Symmetrien in Graphen

KIT Karlsruhe Institute of Technology

Definition: Automorphismen eines DAG

Ein Automorphismus eines DAG $G = (V, E)$ ist eine Knotenpermutation $\pi : V \to V$, die Adjazenzen respektiert und alle Kantenrichtungen erhält oder alle Kantenrichtungen umdreht:

$$(u, v) \in E \Leftrightarrow (\pi(u), \pi(v)) \in E$$

oder

$$(u, v) \in E \Leftrightarrow (\pi(v), \pi(u)) \in E.$$

Die Automorphismen von G bilden mit der Hintereinanderausführung eine Gruppe.

Darstellung von Symmetrien in Graphen

Karlsruhe Institute of Technology

Definition: Automorphismen eines DAG

Ein Automorphismus eines DAG $G = (V, E)$ ist eine Knotenpermutation $\pi : V \to V$, die Adjazenzen respektiert und alle Kantenrichtungen erhält oder alle Kantenrichtungen umdreht:

$$(u, v) \in E \Leftrightarrow (\pi(u), \pi(v)) \in E$$

oder

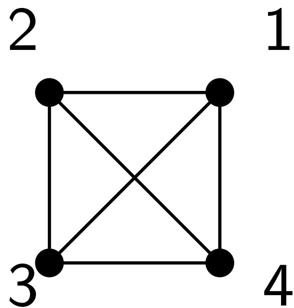
$$(u, v) \in E \Leftrightarrow (\pi(v), \pi(u)) \in E.$$

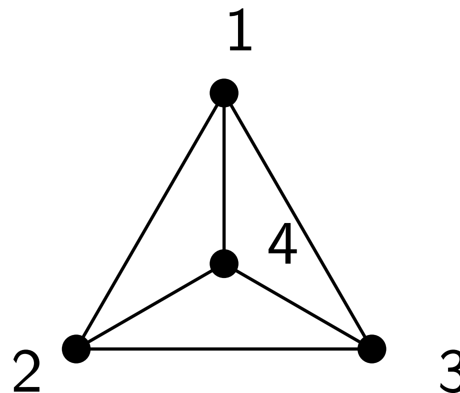
Die Automorphismen von G bilden mit der Hintereinanderausführung eine Gruppe.

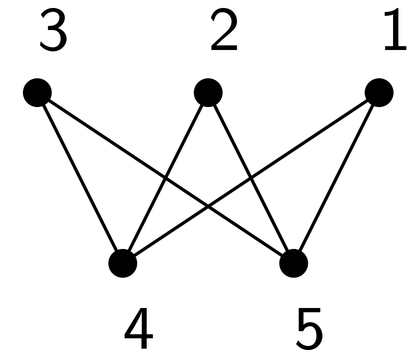
- die Automorphismengruppe eines Graphen zu bestimmen ist GI-vollständig
- für Graphen mit beschränktem Maximalgrad und planare Graphen geht das in Polynomialzeit

Darstellbare Symmetrien

- ein Automorphismus π eines Graphen ist darstellbar, wenn es eine Zeichnung gibt, die eine Symmetrie enthält, welche π induziert
- Lin charakterisiert darstellbare Automorphismen [Lin '92]
- Darstellbare Automorphismen eines Graphen zu finden ist NP-schwer
- für planare Graphen ist das wieder in Polynomialzeit möglich

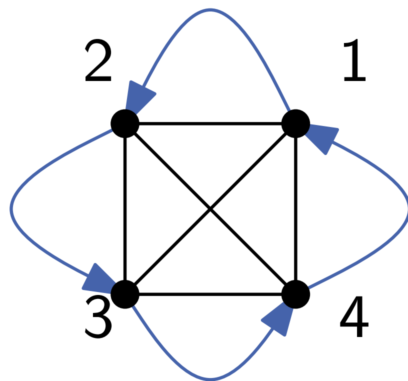




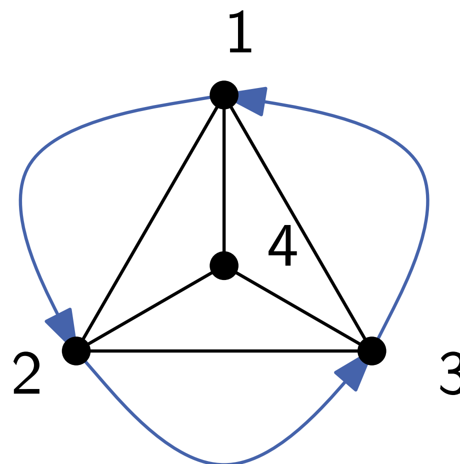


Darstellbare Symmetrien

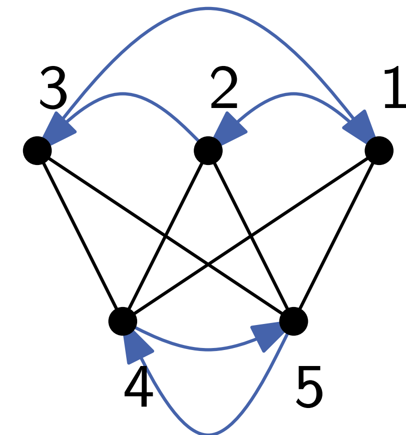
- ein Automorphismus π eines Graphen ist darstellbar, wenn es eine Zeichnung gibt, die eine Symmetrie enthält, welche π induziert
- Lin charakterisiert darstellbare Automorphismen [Lin '92]
- Darstellbare Automorphismen eines Graphen zu finden ist NP-schwer
- für planare Graphen ist das wieder in Polynomialzeit möglich



stellt $1 \to 2 \to 3 \to 4 \to 1$ dar, aber nicht $1 \to 2 \to 3 \to 1$

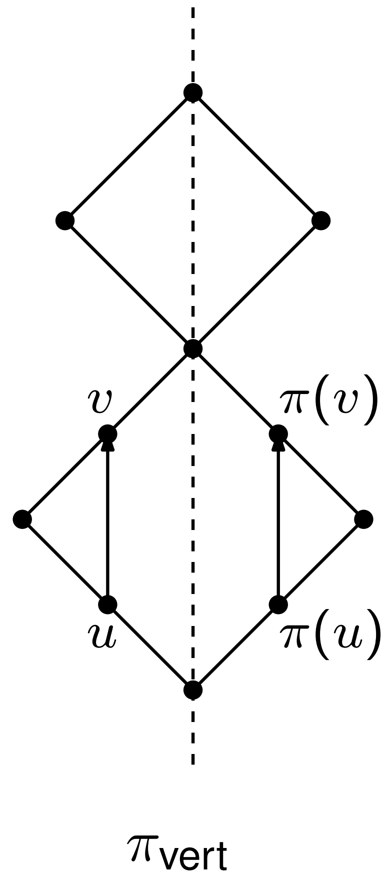


stellt $1 \to 2 \to 3 \to 1$ dar, aber nicht $1 \to 2 \to 3 \to 4 \to 1$

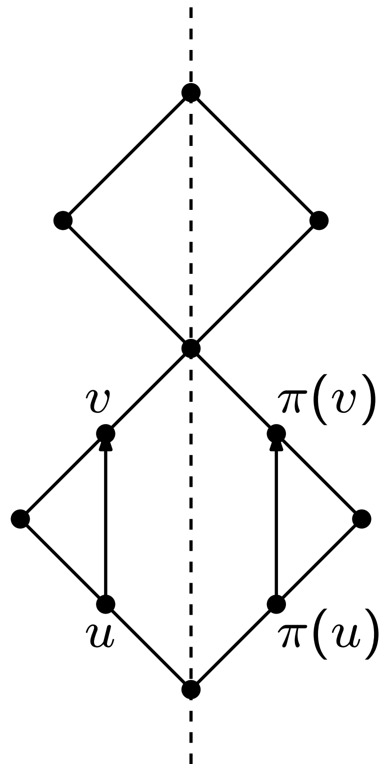


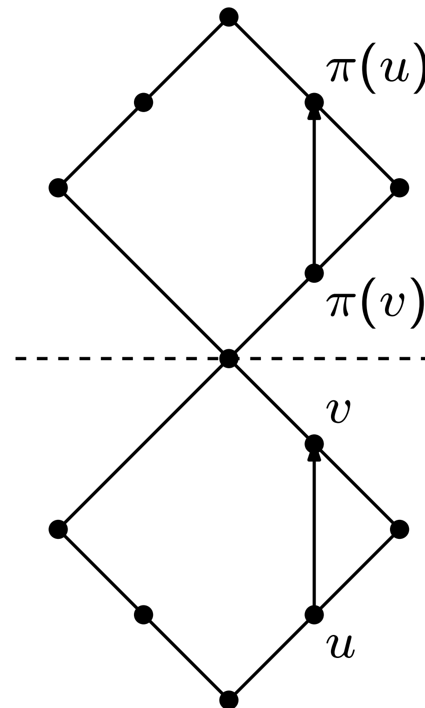
$1 \to 2 \to 3 \to 1$, $4 \to 5 \to 4$ nicht darstellbar

Symmetrien in SP-Graphen

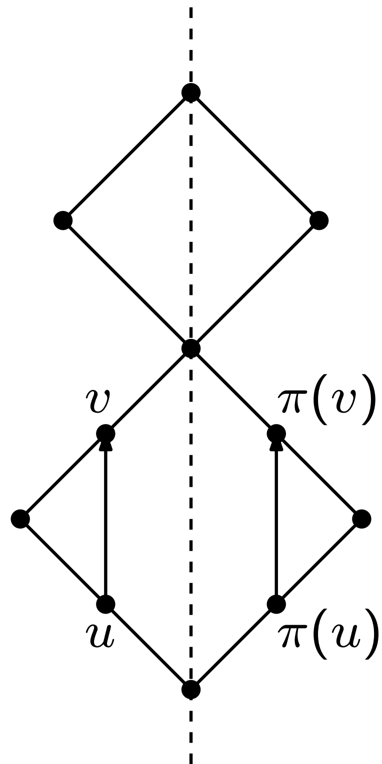


Symmetrien in SP-Graphen

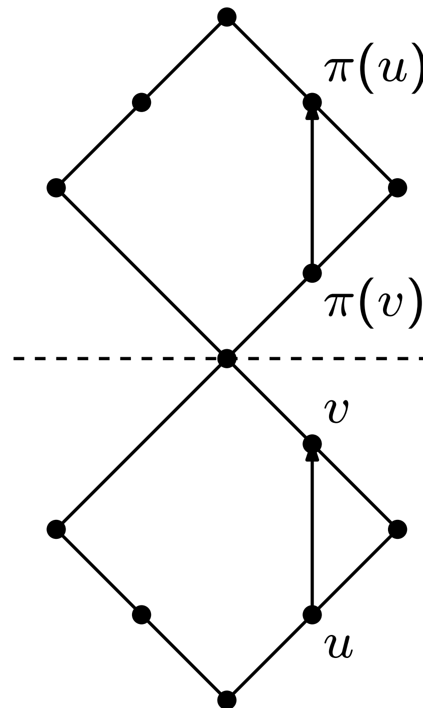




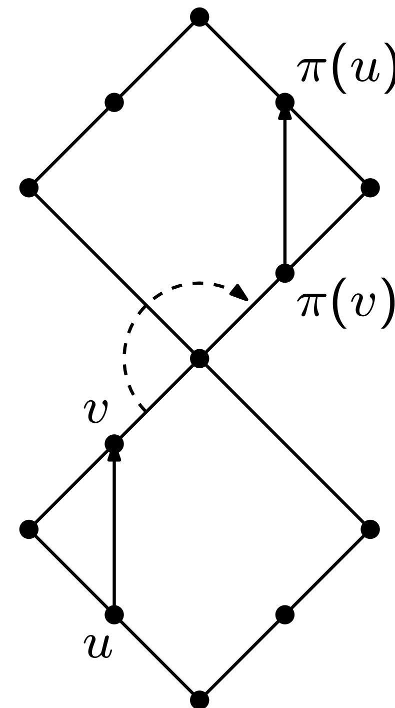
Symmetrien in SP-Graphen



π_{vert}



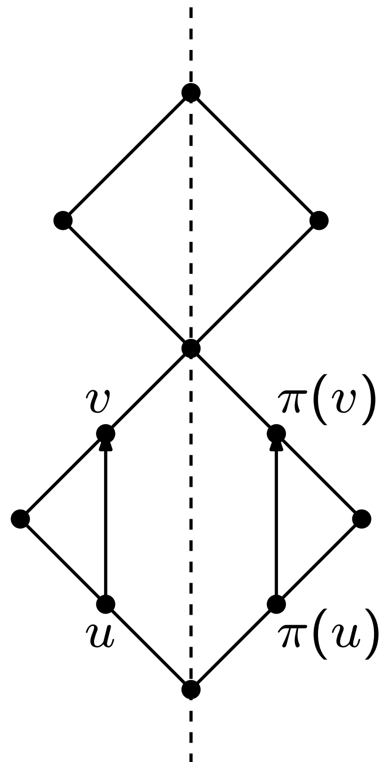
π_{hor}



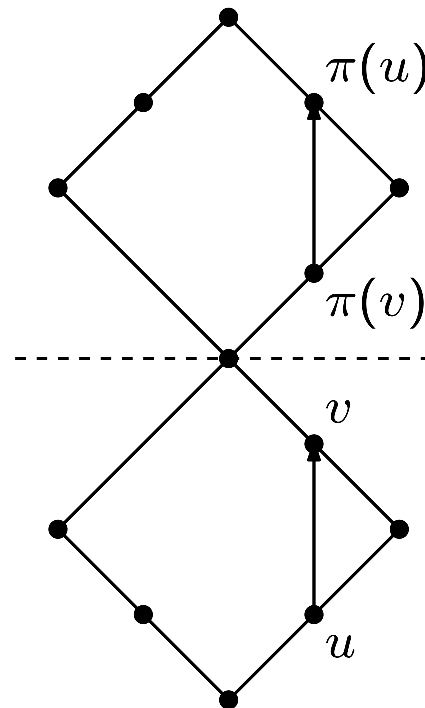
π_{rot}

Institut für Theoretische Informatik
Lehrstuhl Algorithmik I

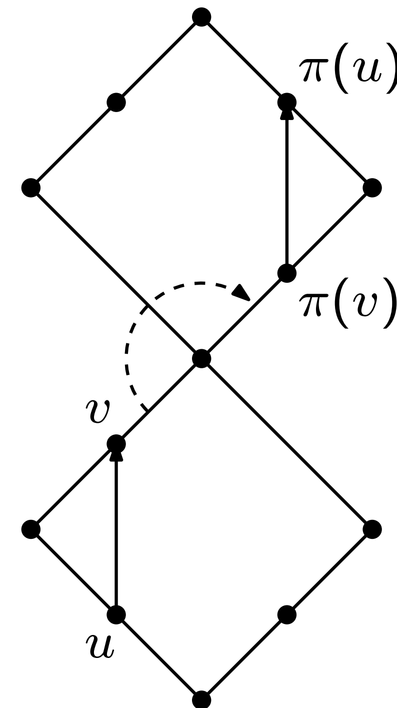
Symmetrien in SP-Graphen



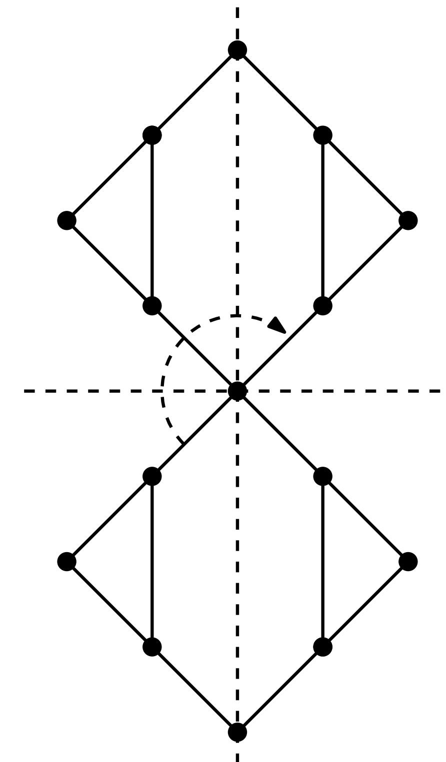
π_{vert}



π_{hor}



π_{rot}



$\{\pi_{\text{vert}}, \pi_{\text{hor}}, \pi_{\text{rot}}\}$

Symmetrien in SP-Zeichnungen

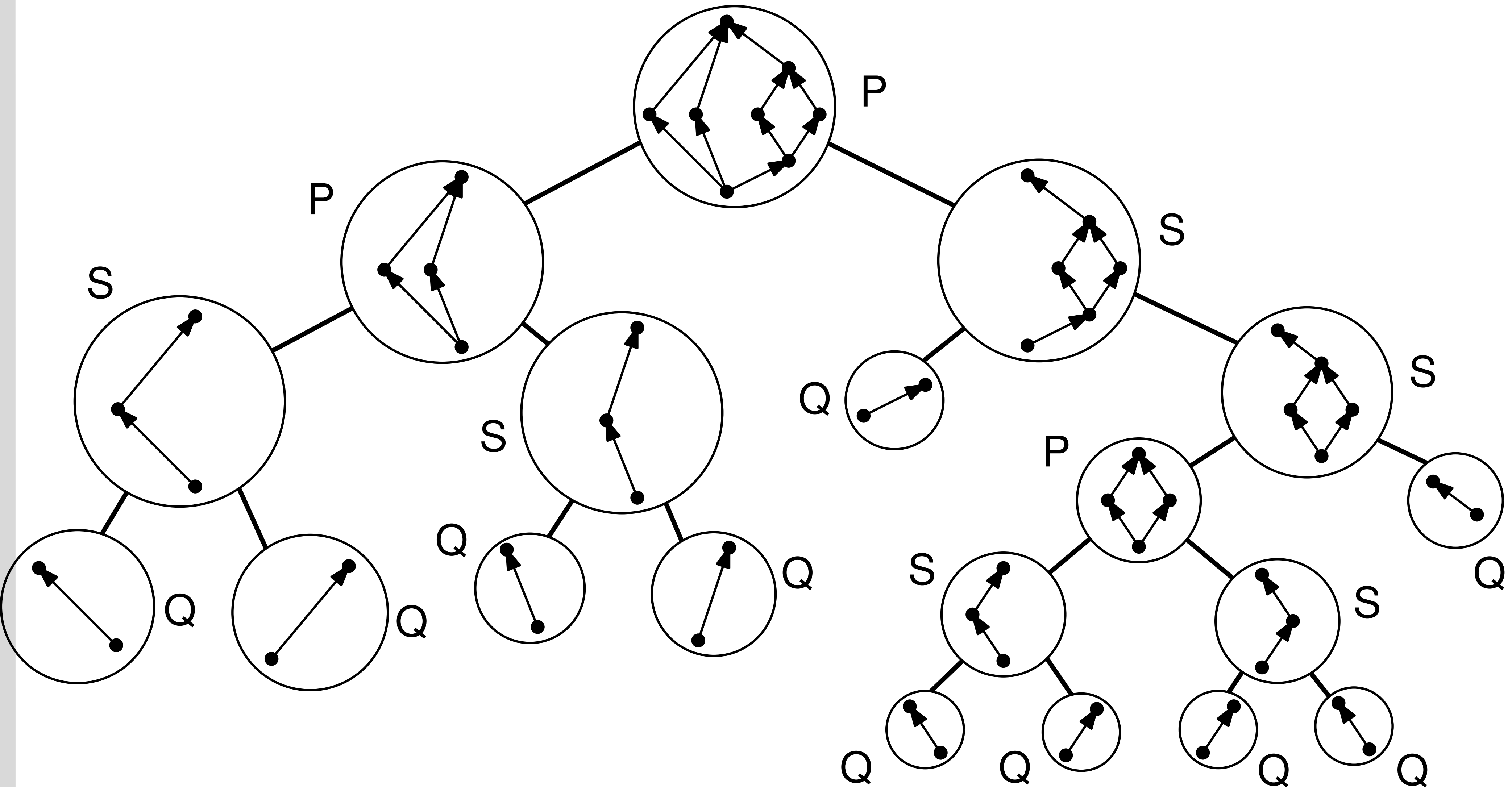
Karlsruhe Institute of Technology

Satz (Hong, Eades, Lee ’00)

Die in einem kreuzungsfreien Aufwärtslayout eines SP-Graphen darstellbaren Symmetrien sind entweder

- $\{\text{id}\}$
- $\{\text{id}, \pi\}$ mit $\pi \in \{\pi_{\text{vert}}, \pi_{\text{hor}}, \pi_{\text{rot}}\}$
- $\{\text{id}, \pi_{\text{vert}}, \pi_{\text{hor}}, \pi_{\text{rot}}\}$.

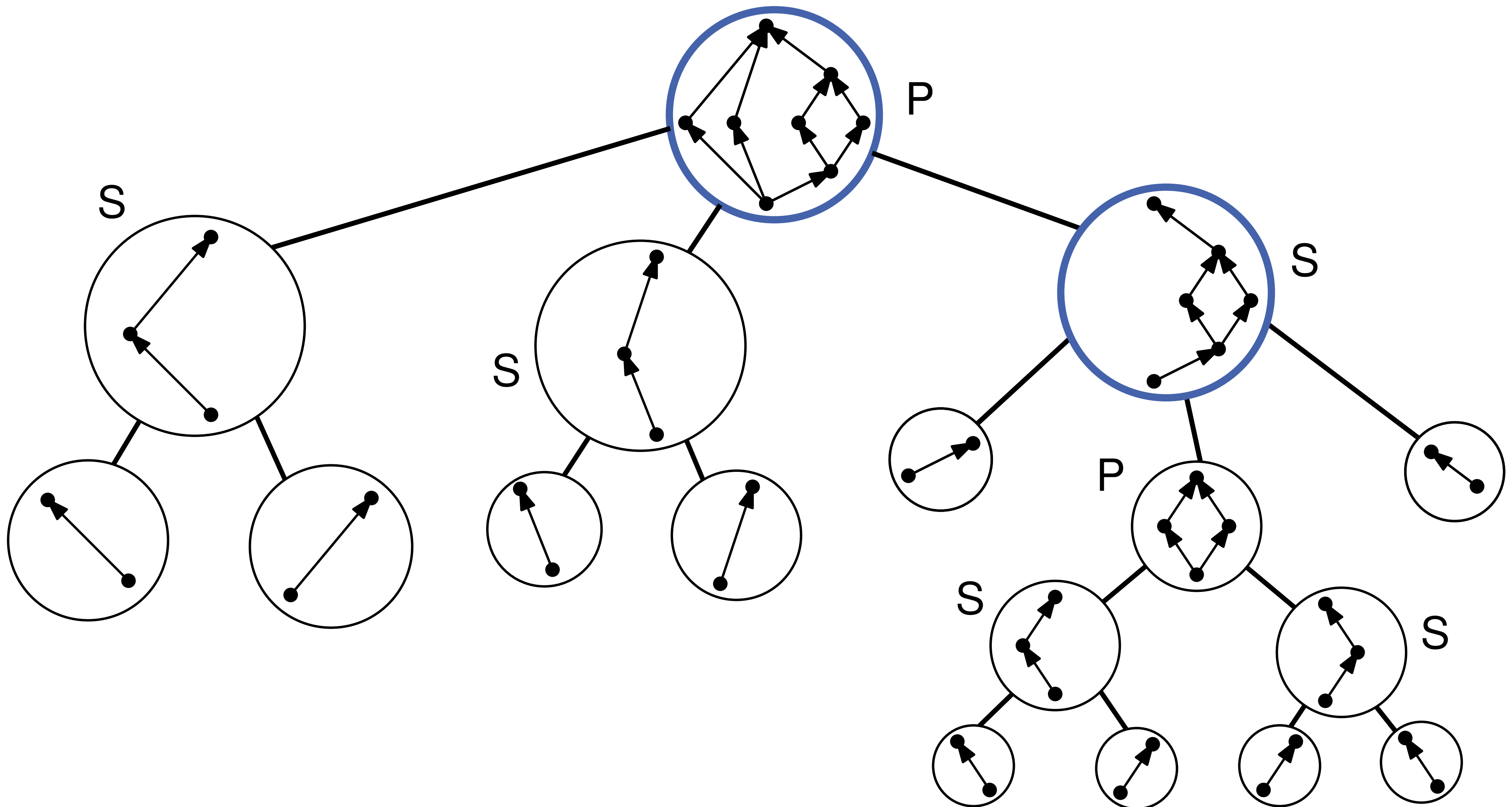
Knotenkodierung im Dekompositionsbaum



Knotenkodierung im Dekompositionsbaum

P P S S S S Q Q Q Q Q P S S Q Q Q Q Q

Knotenkodierung im Dekompositionsbaum



Institut für Theoretische Informatik
Lehrstuhl Algorithmik I

Knotenkodierung im Dekompositionsbaum

Karlsruhe Institute of Technology

Kanonische Kodierung

- Setze $C(G) = \langle 0 \rangle$ für alle Q-Knoten G.

Knotenkodierung im Dekompositionsbaum

Kanonische Kodierung

- Setze $C(G) = \langle 0 \rangle$ für alle Q-Knoten G.
- Für jede Tiefe $t = \max_G \text{tiefe}(G), \dots, 0$
  - Für jeden S- oder P-Knoten G der Tiefe t mit Nachfolgern $G_1, \dots, G_k$ setze $C(G) = \langle c(G_1), \dots, c(G_k) \rangle$ und sortiere $C(G)$ nichtabsteigend, falls G ein P-Knoten ist.

Knotenkodierung im Dekompositionsbaum

Karlsruhe Institute of Technology

Kanonische Kodierung

- Setze $C(G) = \langle 0 \rangle$ für alle Q-Knoten G.
- Für jede Tiefe $t = \max_G \text{tiefe}(G), \dots, 0$
  - Für jeden S- oder P-Knoten G der Tiefe t mit Nachfolgern $G_1, \dots, G_k$ setze $C(G) = \langle c(G_1), \dots, c(G_k) \rangle$ und sortiere $C(G)$ nichtabsteigend, falls G ein P-Knoten ist.
  - Sortiere die Menge der Tupel aller Knoten der Tiefe t lexikographisch.

Knotenkodierung im Dekompositionsbaum

Karlsruhe Institute of Technology

Kanonische Kodierung

- Setze $C(G) = \langle 0 \rangle$ für alle Q-Knoten G.
- Für jede Tiefe $t = \max_G \text{tiefe}(G), \ldots, 0$
  - Für jeden S- oder P-Knoten G der Tiefe t mit Nachfolgern $G_1, \ldots, G_k$ setze $C(G) = \langle c(G_1), \ldots, c(G_k) \rangle$ und sortiere $C(G)$ nichtabsteigend, falls G ein P-Knoten ist.
  - Sortiere die Menge der Tupel aller Knoten der Tiefe t lexikographisch.
  - Für jede Komponente G der Tiefe t setze ihre Kodierung auf c, falls ihr Tupel in der sortierten Tupelfolge als c-tes verschiedenes Tupel auftritt.

Knotenkodierung im Dekompositionsbaum

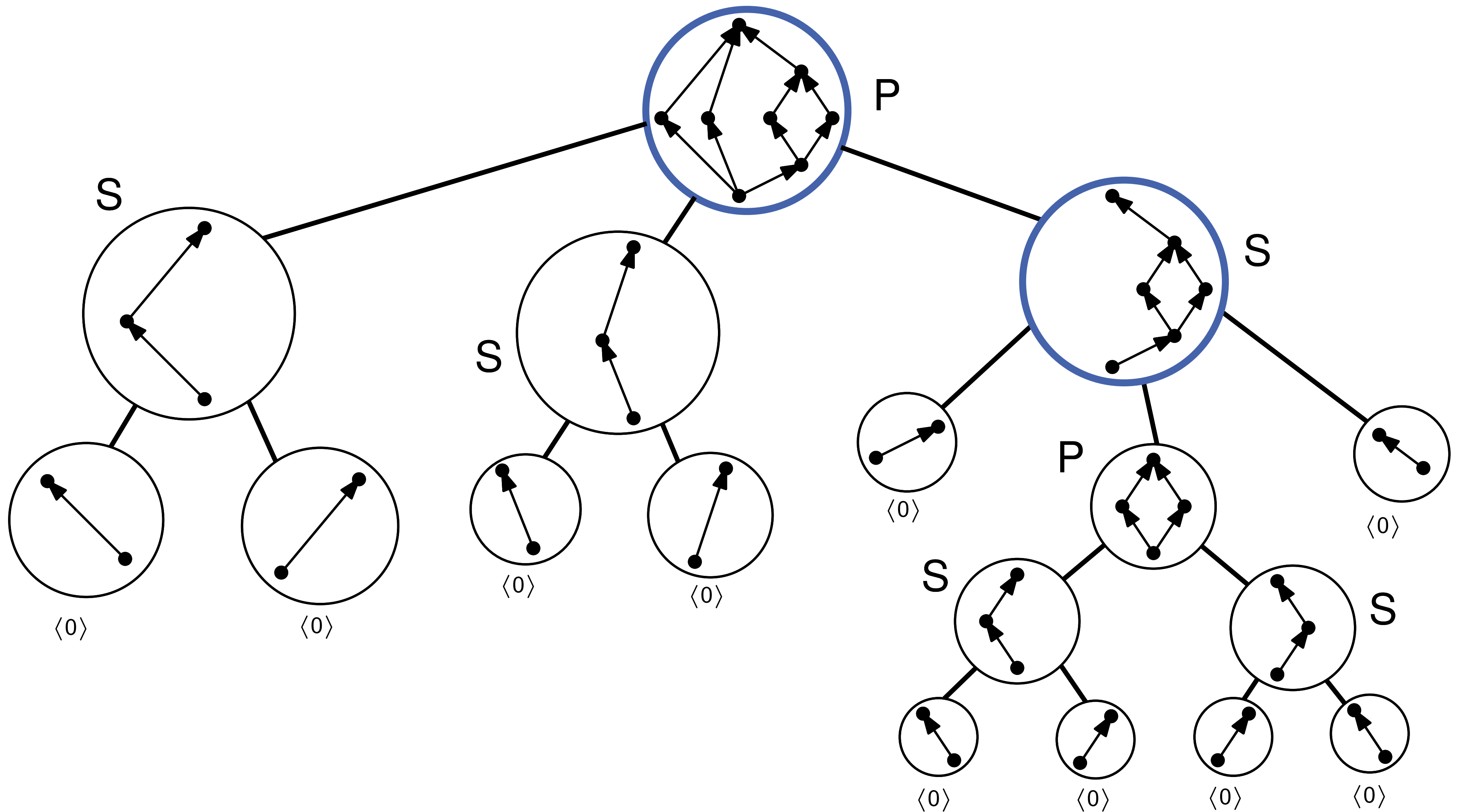
KIT
Karlsruhe Institute of Technology

Kanonische Kodierung

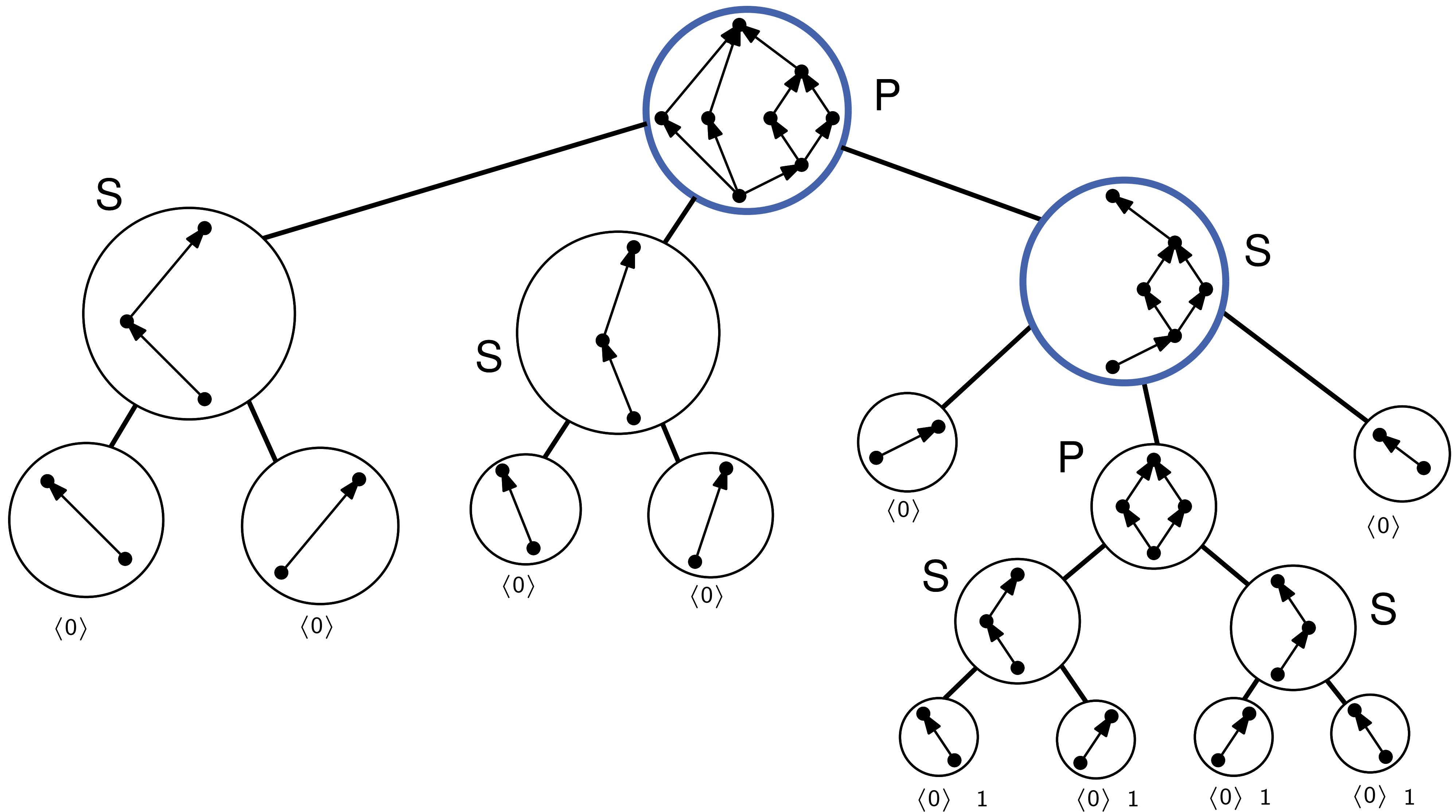
- Setze $C(G) = \langle 0 \rangle$ für alle Q-Knoten G.
- Für jede Tiefe $t = \max_G \text{tiefe}(G), \ldots, 0$
  - Für jeden S- oder P-Knoten G der Tiefe t mit Nachfolgern $G_1, \ldots, G_k$ setze $C(G) = \langle c(G_1), \ldots, c(G_k) \rangle$ und sortiere $C(G)$ nichtabsteigend, falls G ein P-Knoten ist.
  - Sortiere die Menge der Tupel aller Knoten der Tiefe t lexikographisch.
  - Für jede Komponente G der Tiefe t setze ihre Kodierung auf c, falls ihr Tupel in der sortierten Tupelfolge als c-tes verschiedenes Tupel auftritt.

Zwei Knoten u und v gleicher Tiefe sind genau dann isomorph, wenn sie den gleichen Code haben.

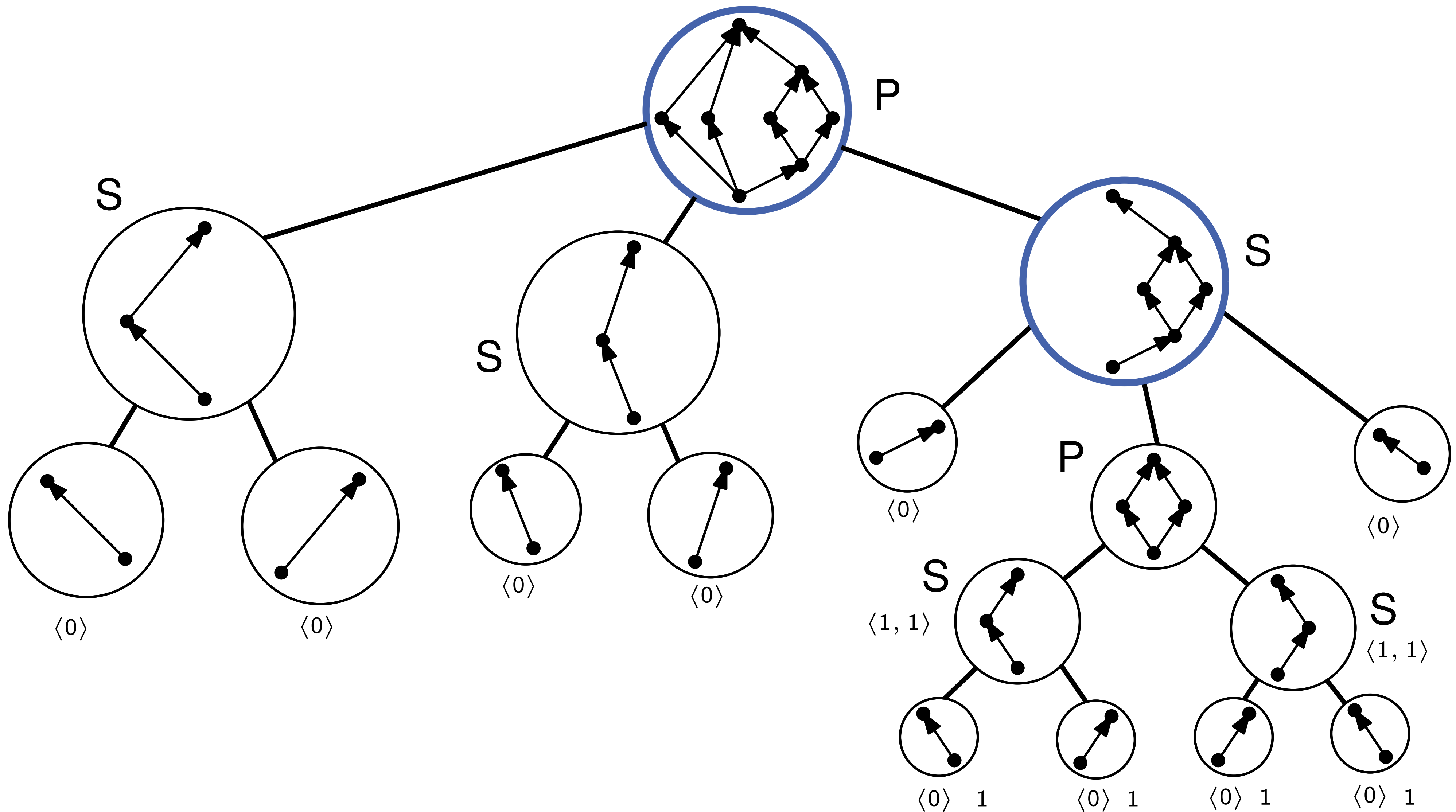
Knotenkodierung im Dekompositionsbaum



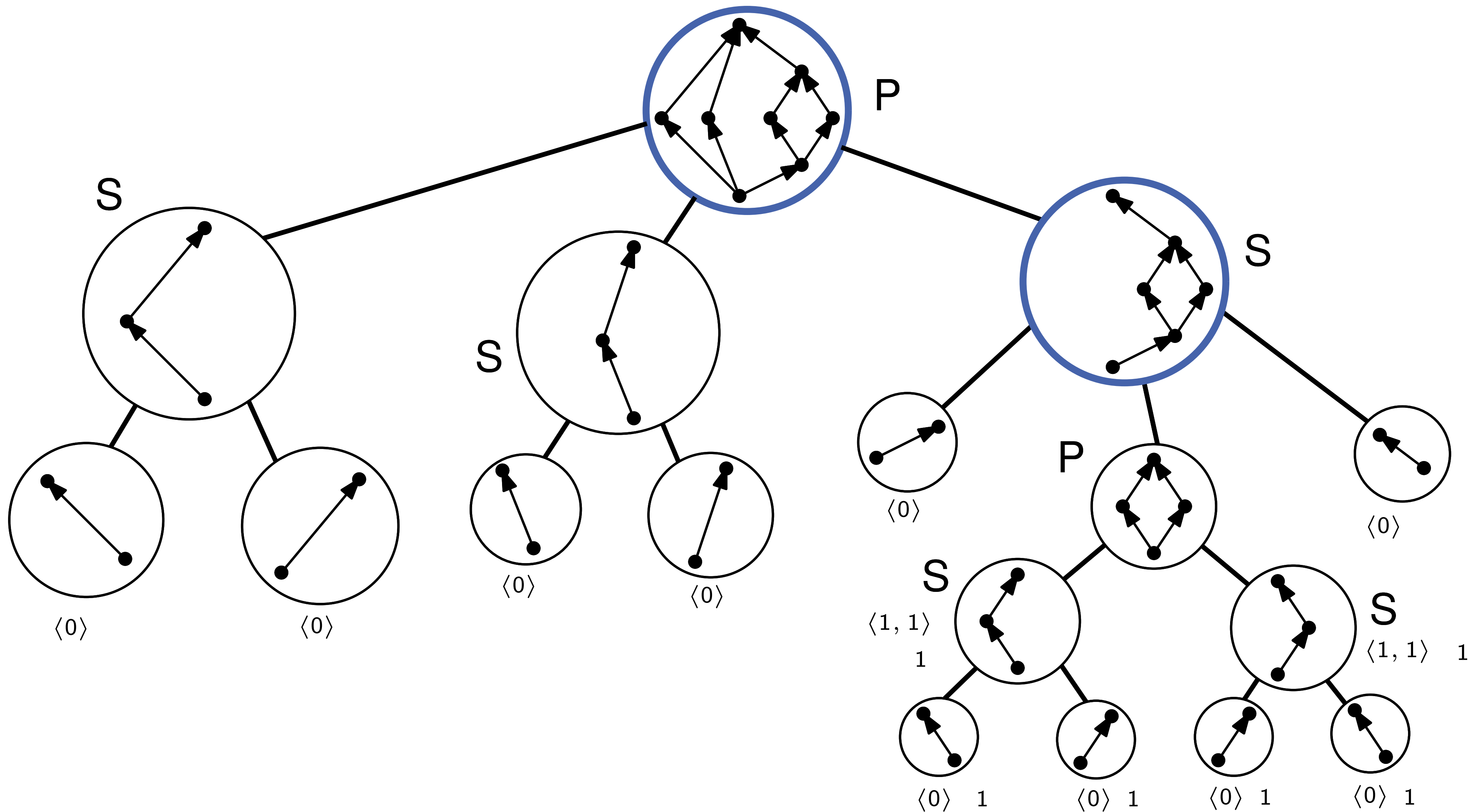
Knotenkodierung im Dekompositionsbaum



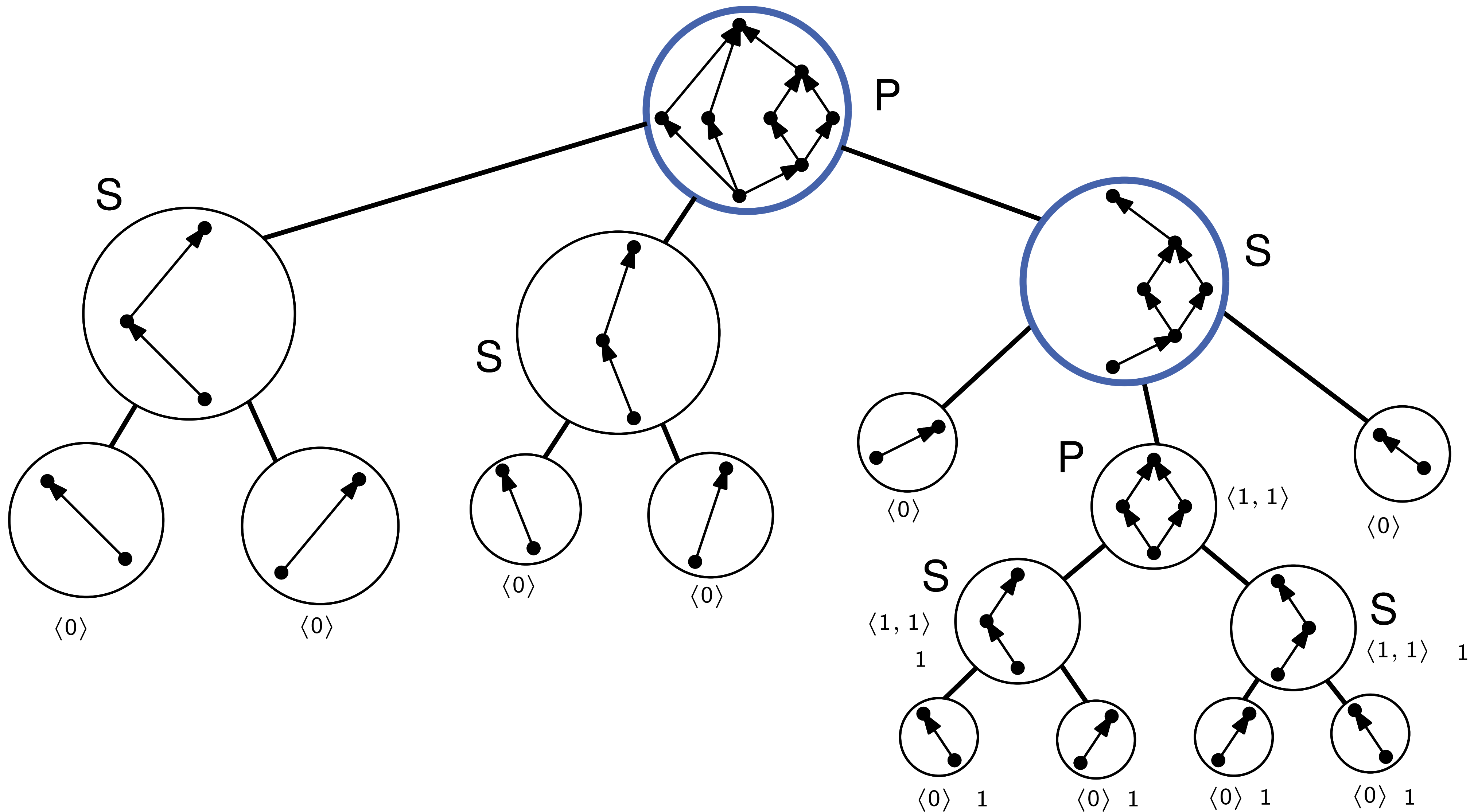
Knotenkodierung im Dekompositionsbaum



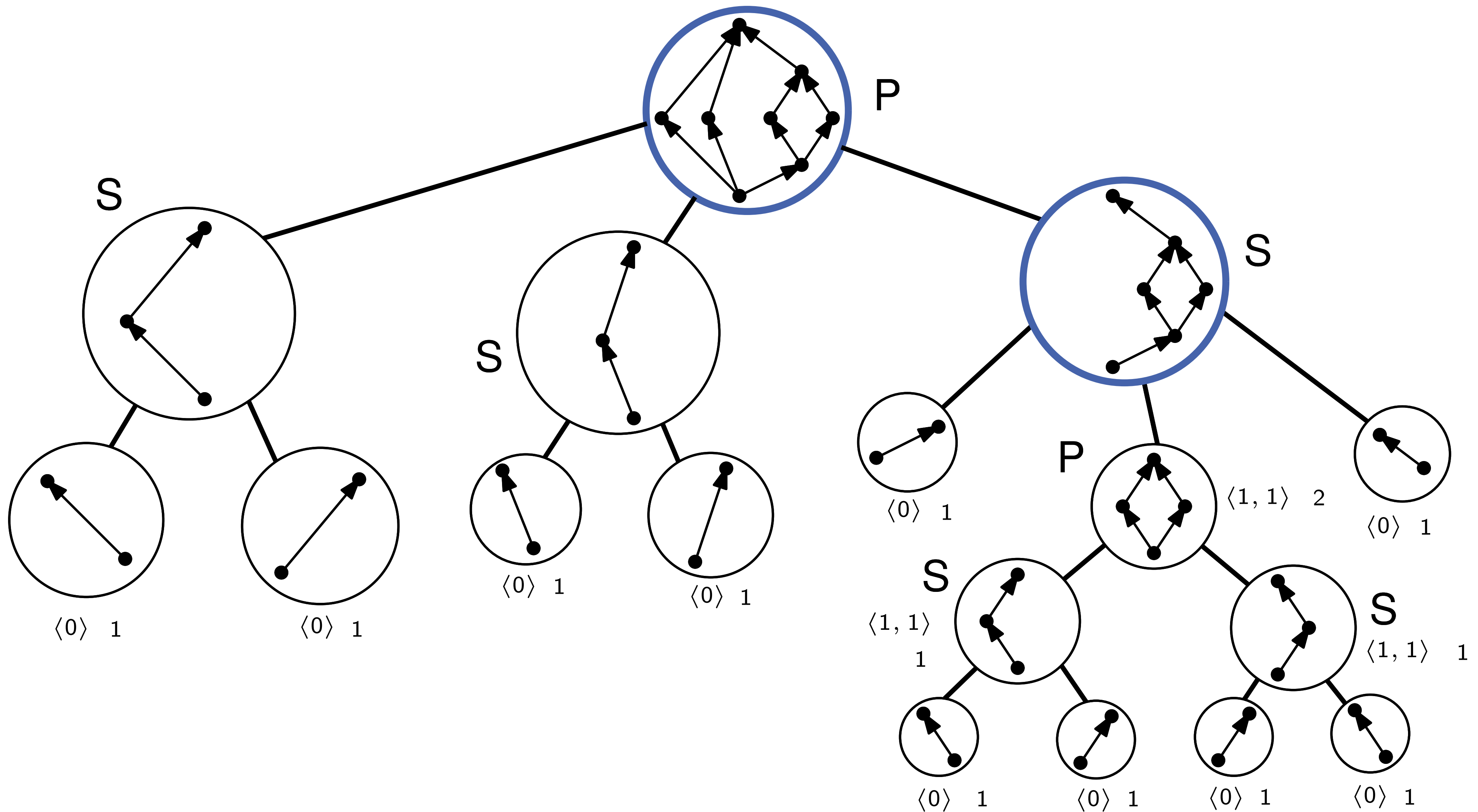
Knotenkodierung im Dekompositionsbaum



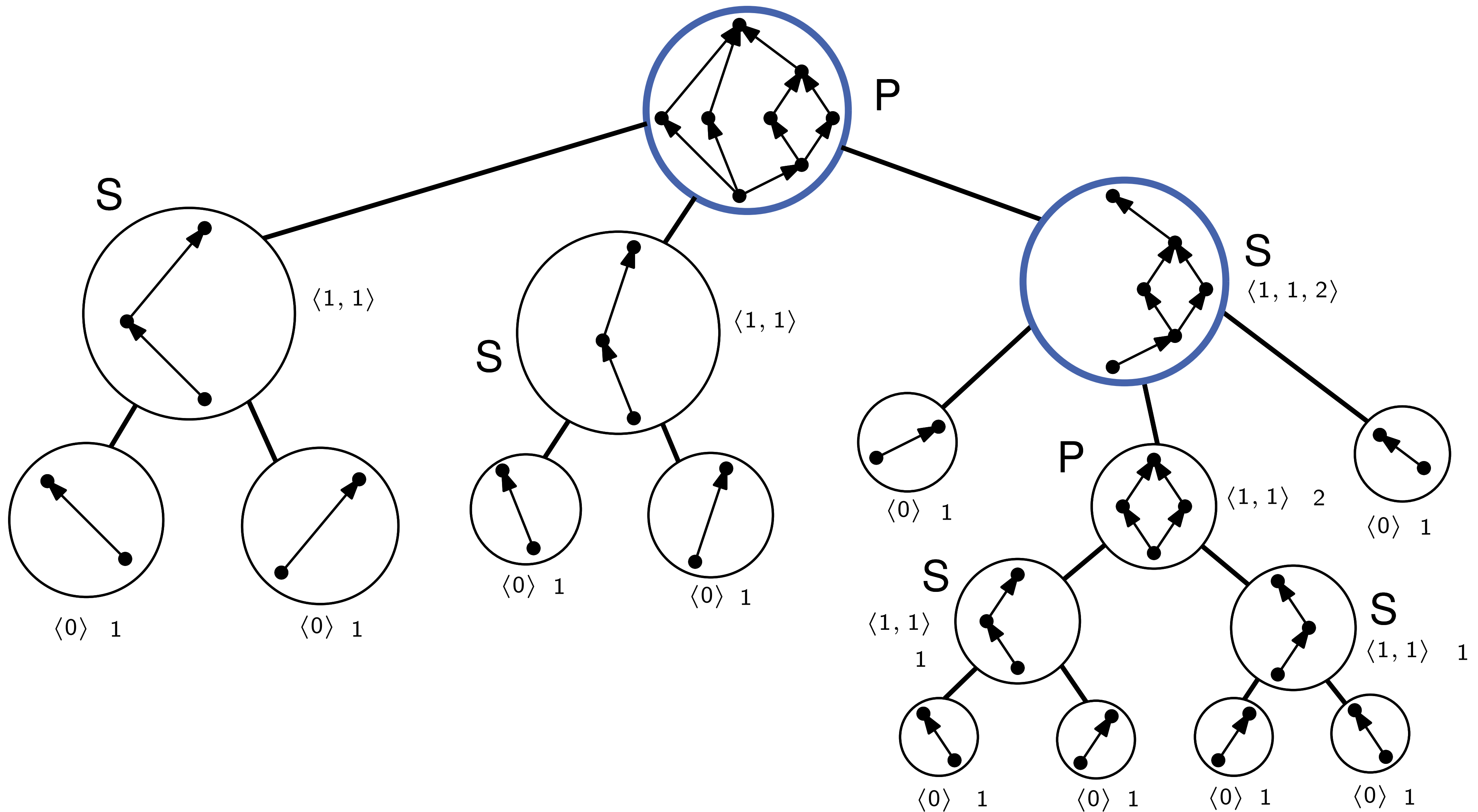
Knotenkodierung im Dekompositionsbaum



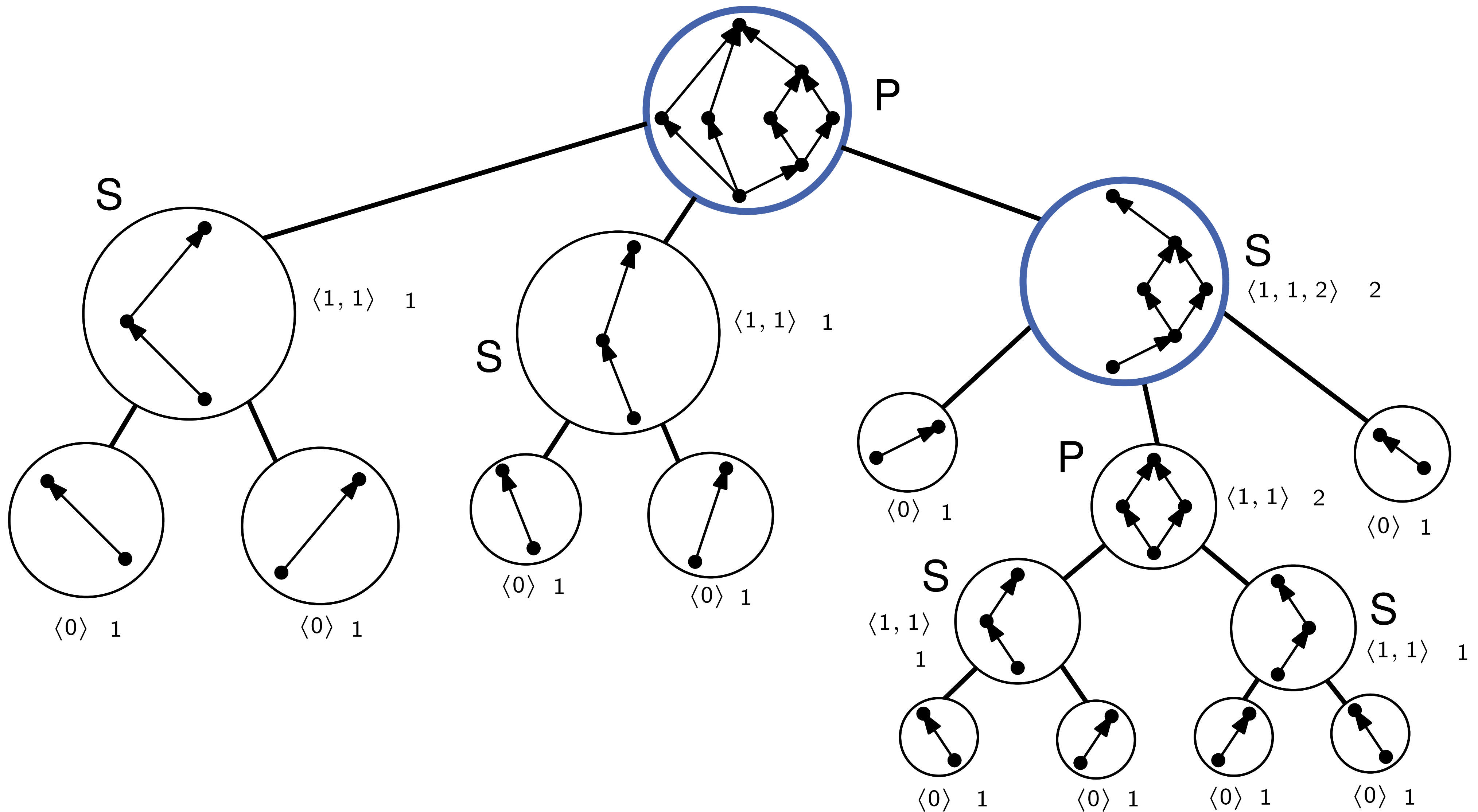
Knotenkodierung im Dekompositionsbaum



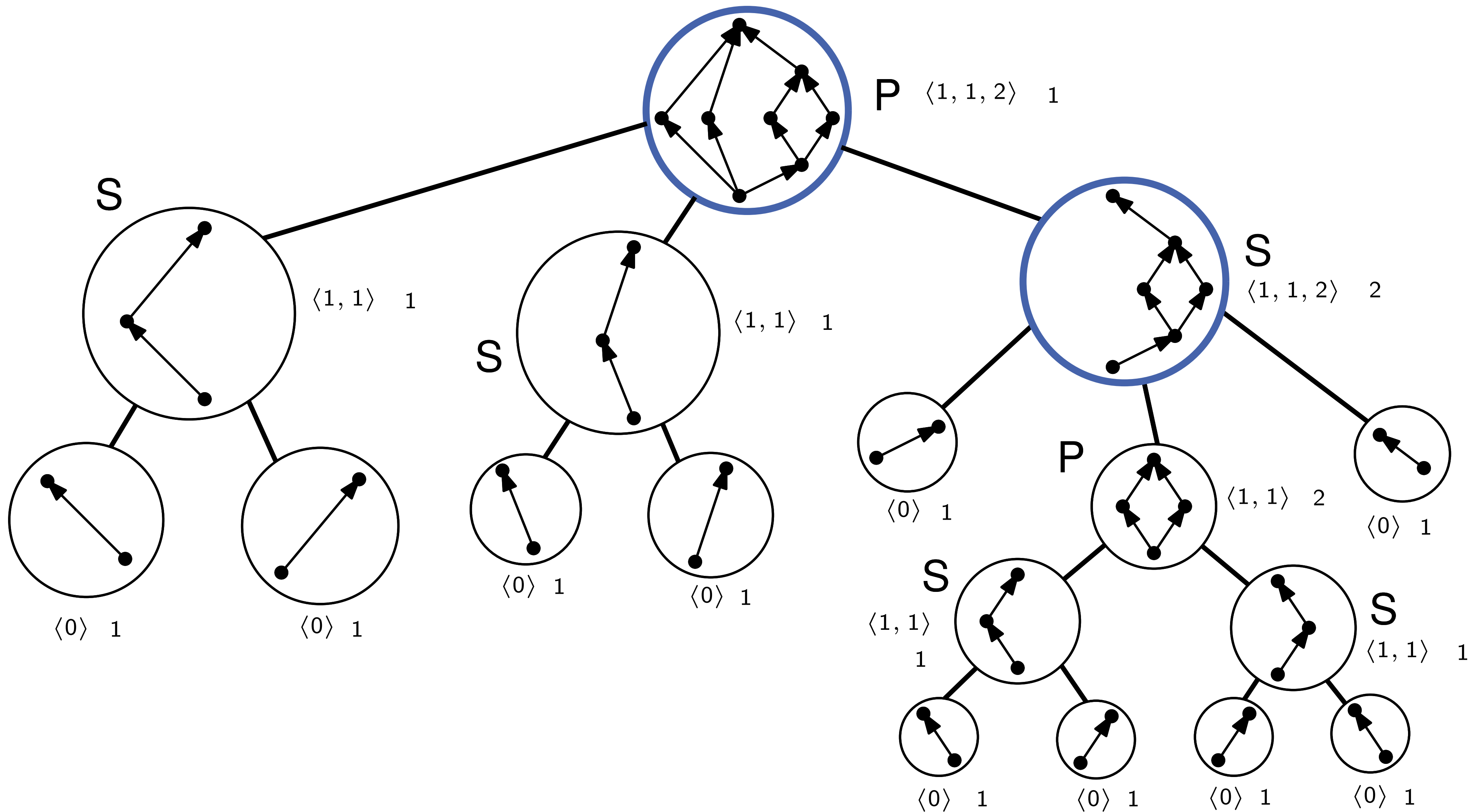
Knotenkodierung im Dekompositionsbaum



Knotenkodierung im Dekompositionsbaum



Knotenkodierung im Dekompositionsbaum



Vertikale Symmetrien

Karlsruhe Institute of Technology

Satz (Hong, Eades, Lee ’00)

Gegeben sei der kanonische Dekompositionsbaum eines serien-parallelen Graphen. Es sei G eine Komponente, die durch Komposition der Komponenten $G_1, \ldots, G_k$ entstehe.

- Ist G ein S-Knoten, dann ist G vertikal symmetrisch, wenn alle $G_1, \ldots, G_k$ vertikal symmetrisch sind.

Vertikale Symmetrien

Karlsruhe Institute of Technology

Satz (Hong, Eades, Lee ’00)

Gegeben sei der kanonische Dekompositionsbaum eines serien-parallelen Graphen. Es sei G eine Komponente, die durch Komposition der Komponenten $G_1, \ldots, G_k$ entstehe.

- Ist G ein S-Knoten, dann ist G vertikal symmetrisch, wenn alle $G_1, \ldots, G_k$ vertikal symmetrisch sind.
- Ist G ein P-Knoten, so betrachten wir die Klassen $\mathcal{G}_j = \{G_i : 1 \leq i \leq k, c(G_i) = j\}$, $j = 1, \ldots, k$, von isomorphen Teilgraphen.
  - $|\mathcal{G}_j|$ gerade $\forall j \Rightarrow$ vertikal symmetrisch

Vertikale Symmetrien

Karlsruhe Institute of Technology

Satz (Hong, Eades, Lee '00)

Gegeben sei der kanonische Dekompositionsbaum eines serien-parallelen Graphen. Es sei G eine Komponente, die durch Komposition der Komponenten $G_1, \dots, G_k$ entstehe.

- Ist G ein S-Knoten, dann ist G vertikal symmetrisch, wenn alle $G_1, \dots, G_k$ vertikal symmetrisch sind.
- Ist G ein P-Knoten, so betrachten wir die Klassen $\mathcal{G}_j = \{G_i : 1 \leq i \leq k, c(G_i) = j\}$, $j = 1, \dots, k$, von isomorphen Teilgraphen.
  - $|\mathcal{G}_j|$ gerade $\forall j \Rightarrow$ vertikal symmetrisch
  - $|\mathcal{G}_j|$ ungerade für genau ein $j \Rightarrow$ G vertikal symmetrisch g.d.w. Graphen in $\mathcal{G}_j$ vertikal symmetrisch

Vertikale Symmetrien

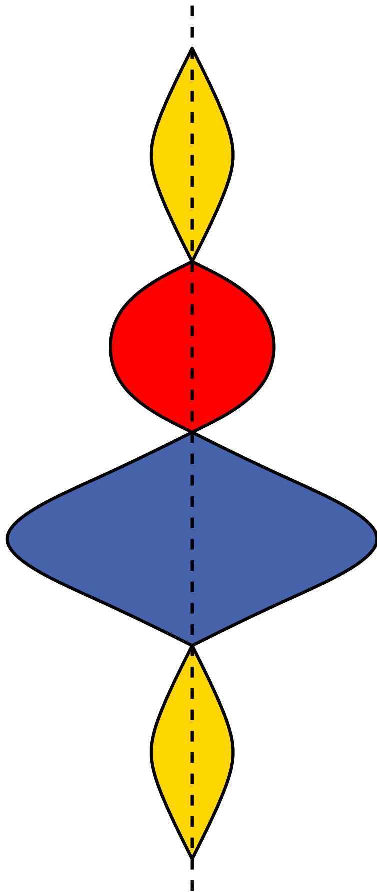
Karlsruhe Institute of Technology

Satz (Hong, Eades, Lee '00)

Gegeben sei der kanonische Dekompositionsbaum eines serien-parallelen Graphen. Es sei G eine Komponente, die durch Komposition der Komponenten $G_1, \dots, G_k$ entstehe.

- Ist G ein S-Knoten, dann ist G vertikal symmetrisch, wenn alle $G_1, \dots, G_k$ vertikal symmetrisch sind.
- Ist G ein P-Knoten, so betrachten wir die Klassen $\mathcal{G}_j = \{G_i : 1 \leq i \leq k, c(G_i) = j\}$, $j = 1, \dots, k$, von isomorphen Teilgraphen.
  - $|\mathcal{G}_j|$ gerade $\forall j \Rightarrow$ vertikal symmetrisch
  - $|\mathcal{G}_j|$ ungerade für genau ein $j \Rightarrow$ G vertikal symmetrisch g.d.w. Graphen in $\mathcal{G}_j$ vertikal symmetrisch
  - $|\mathcal{G}_i|, |\mathcal{G}_j|$ ungerade für $i \neq j \Rightarrow G$ nicht vertikal symmetrisch

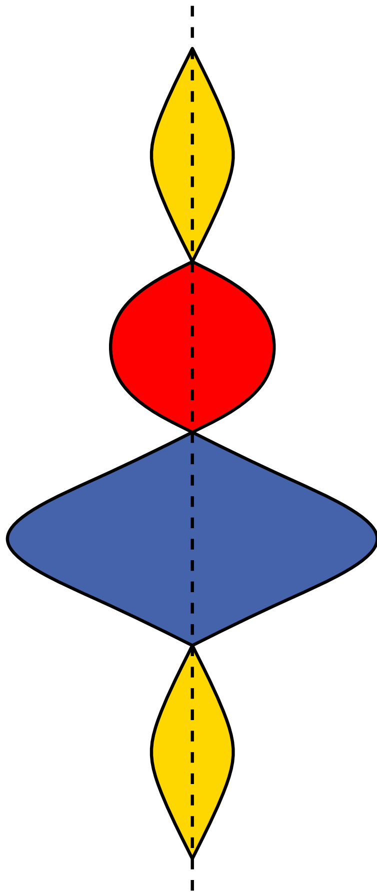
Beweisidee vertikale Symmetrie



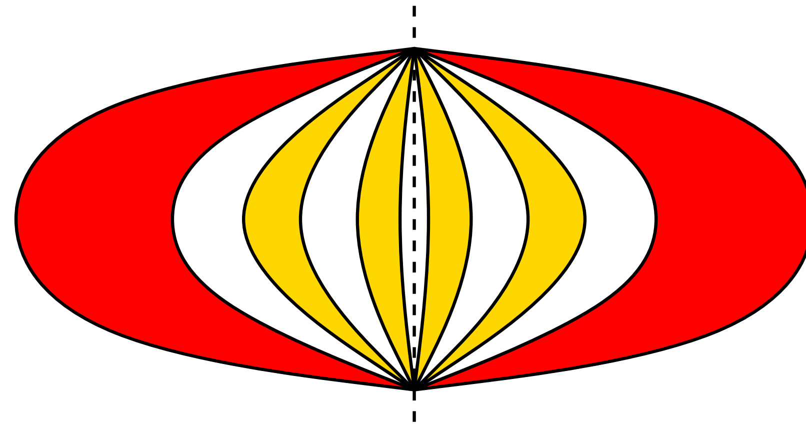
S-Knoten

Institut für Theoretische Informatik
Lehrstuhl Algorithmik I

Beweisidee vertikale Symmetrie



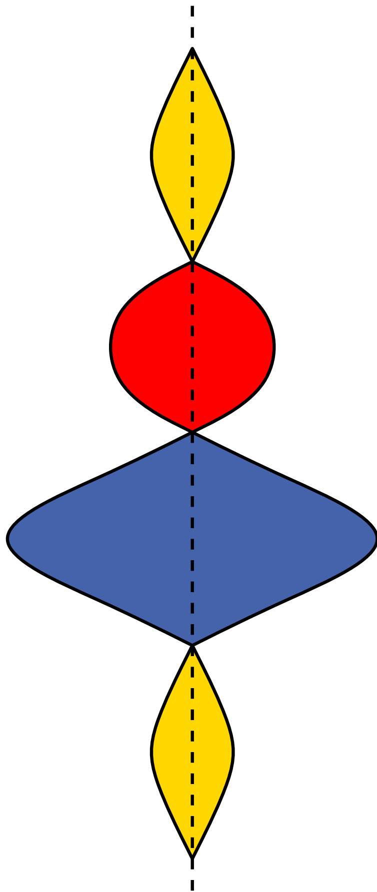
S-Knoten



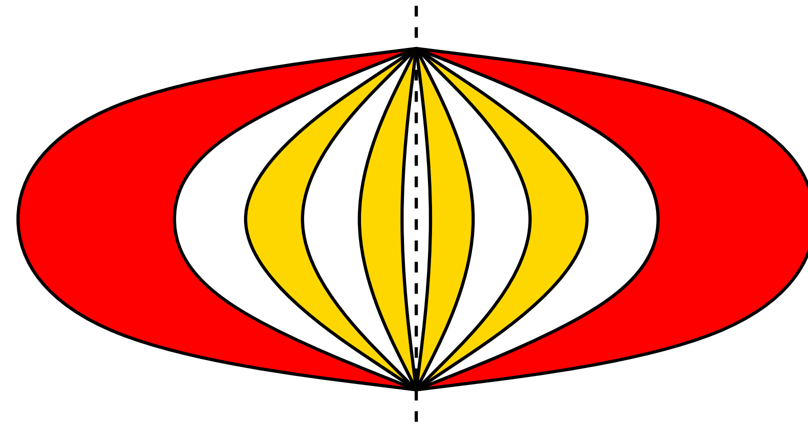
P-Knoten, alle Klassen gerade Anzahl

Beweisidee vertikale Symmetrie

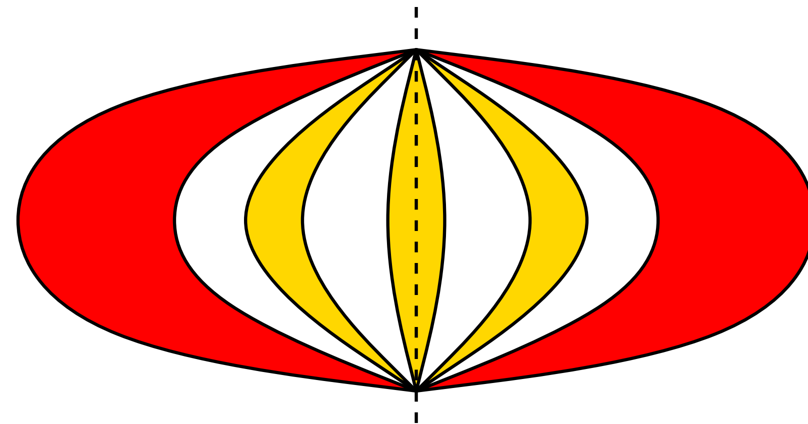
Karlsruhe Institute of Technology



S-Knoten
alle Knoten v-symm.

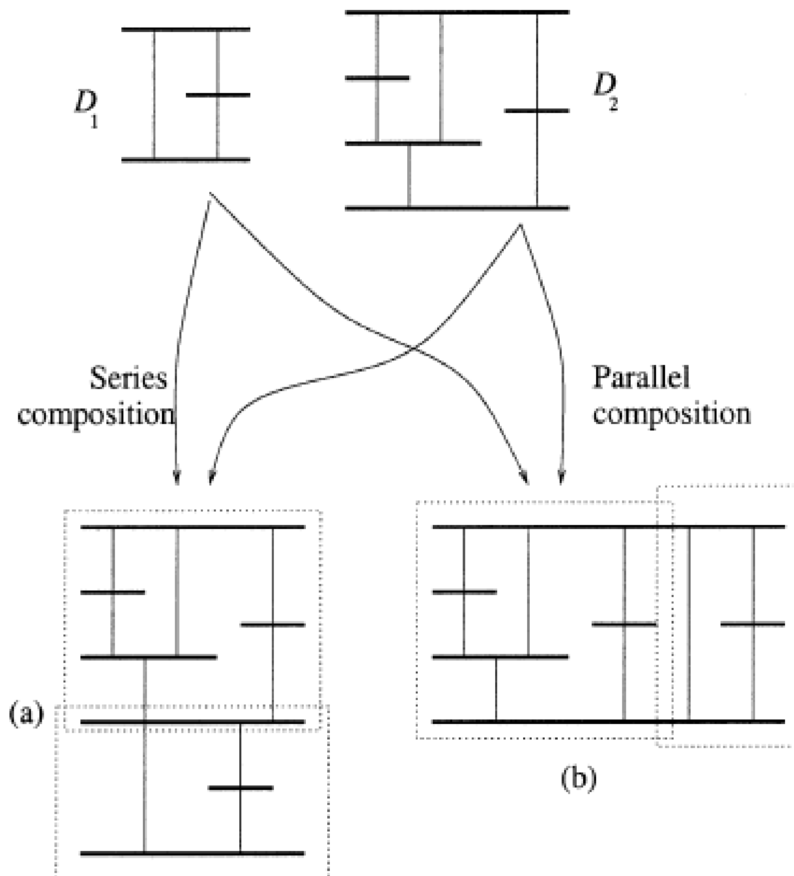


P-Knoten, alle Klassen gerade Anzahl



P-Knoten, eine Klasse ungerade Anzahl & v-symm.

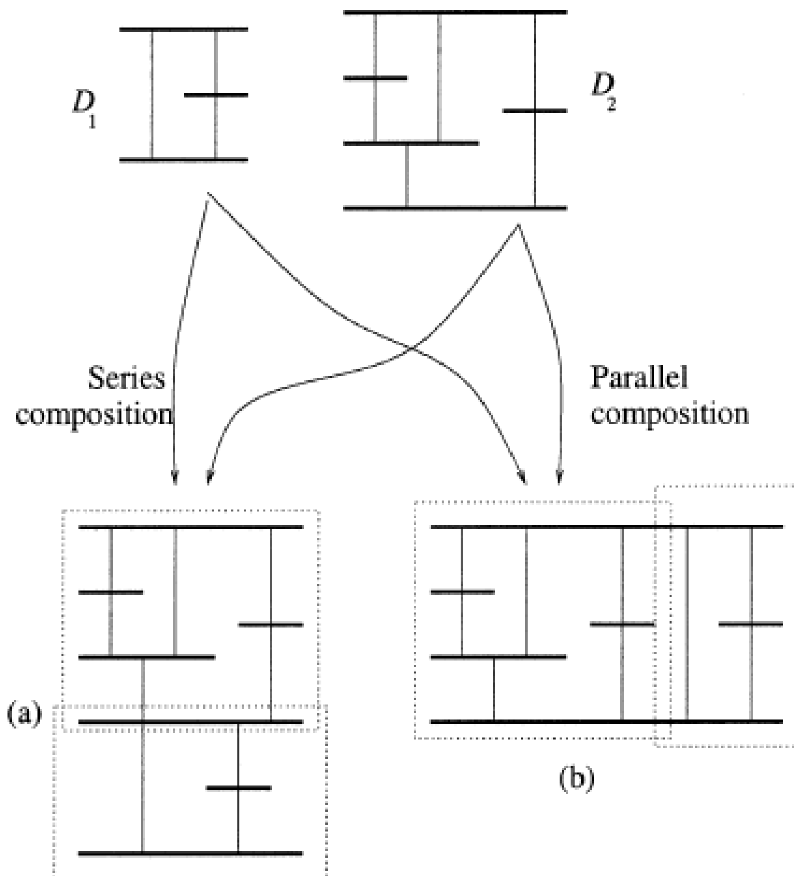
Symmetrien zeichnen

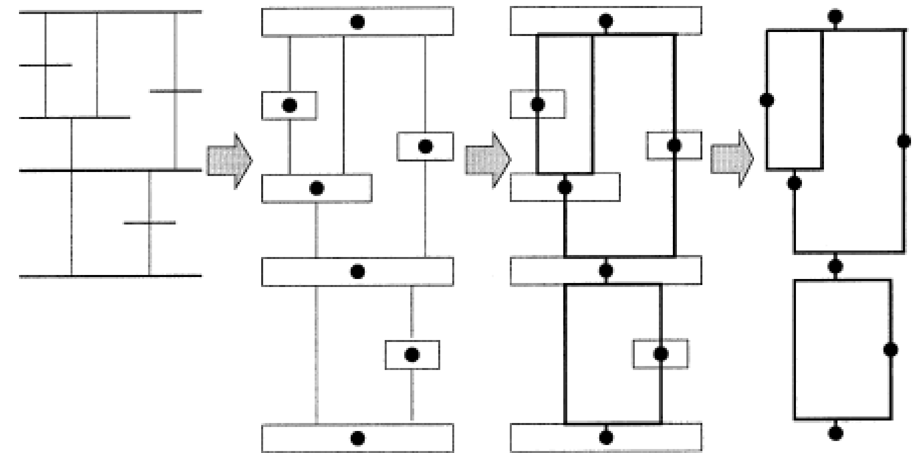


aus Hong, Eades, Lee ’00

Symmetrien zeichnen

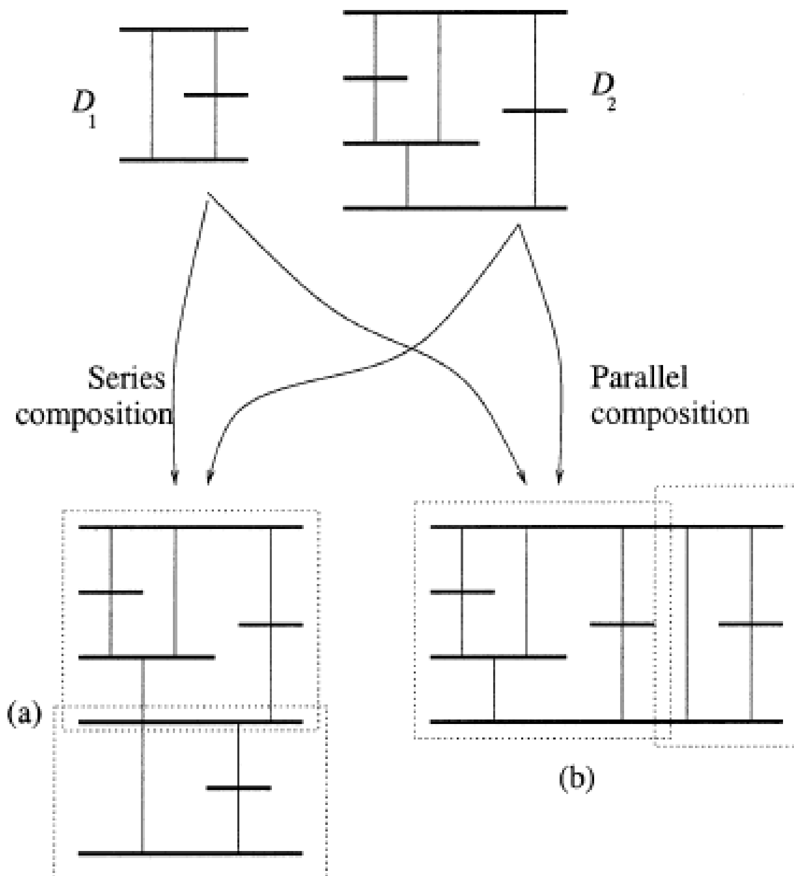
Karlsruhe Institute of Technology

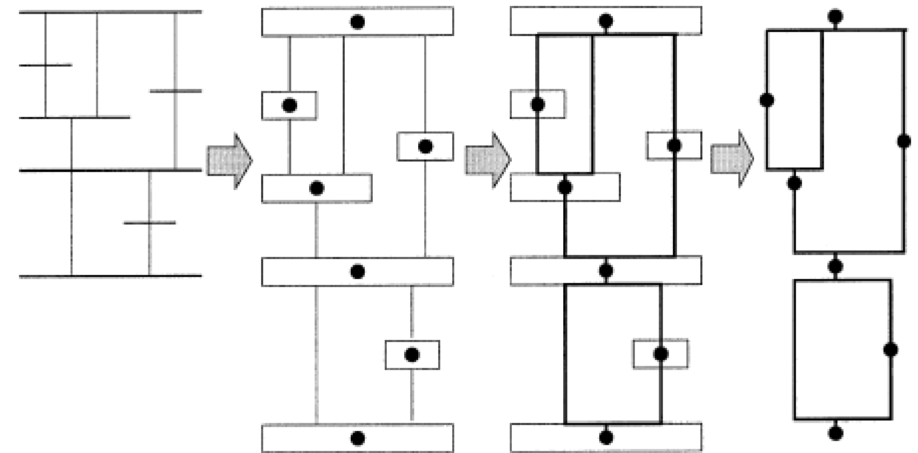


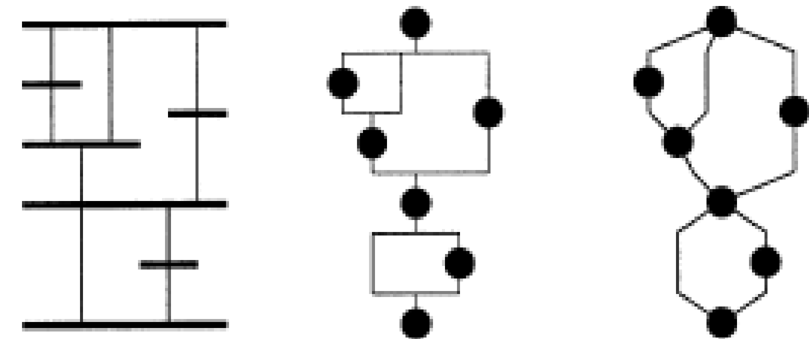


aus Hong, Eades, Lee ’00

Symmetrien zeichnen







aus Hong, Eades, Lee ’00

Inkrementelle Verfahren

Allgemeine Vorgehensweise

- bestimme geeignete Knoten-Ordnung
- berechne Layout inkrementell durch iteratives Hinzufügen der Knoten

Beispiel: Orthogonale Zeichnung von Graphen mit Maximalgrad ≤ 4

Vorgehensweise

- 2-fach-Zusammenhangskomponenten berechnen
- für jede 2-fach-Zusammenhangskomponente geeignete Knoten-Ordnung berechnen
- Inkrementelles Layout für Zusammenhangskomponente
- rekursiver Ansatz für Gesamt-Graph

Blöcke und Block-Baum

Karlsruhe Institute of Technology

Schnittknoten (Artikulation): Knoten, dessen Entfernen die Anzahl der Zusammenhangskomponenten erhöht; C Menge der Schnittknoten von G

Blöcke und Block-Baum

Karlsruhe Institute of Technology

Schnittknoten (Artikulation): Knoten, dessen Entfernen die Anzahl der Zusammenhangskomponenten erhöht; C Menge der Schnittknoten von G

Block maximaler 2-fach zusammenhängender Graph oder Brücke oder isolierter Knoten; Menge der Blöcke B

Blöcke und Block-Baum

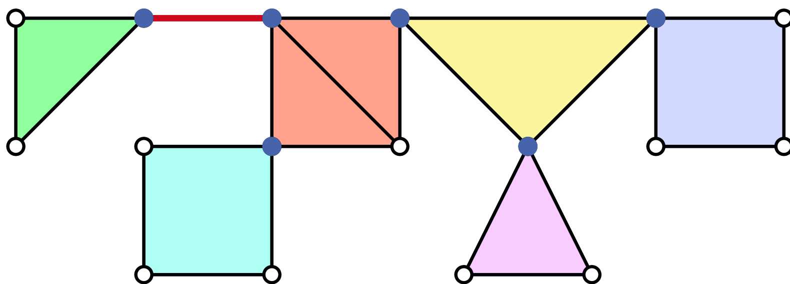
Karlsruhe Institute of Technology

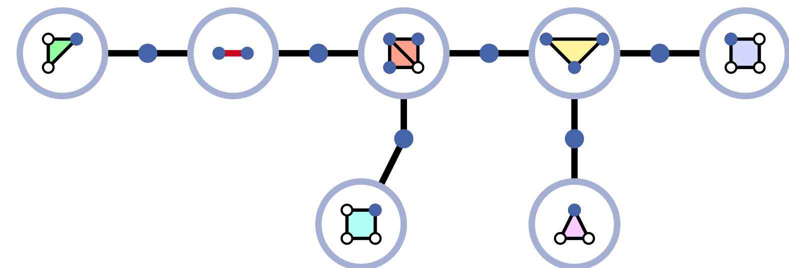
Schnittknoten (Artikulation): Knoten, dessen Entfernen die Anzahl der Zusammenhangskomponenten erhöht; C Menge der Schnittknoten von G

Block maximaler 2-fach zusammenhängender Graph oder Brücke oder isolierter Knoten; Menge der Blöcke B

Block-Baum

Baum $T = (\mathcal{B}, F)$ mit $\mathcal{B} = B \cup C$ und F enthält alle Kanten zwischen $b \in B$ und $c \in C$ genau dann, wenn $c \in b$ gilt.

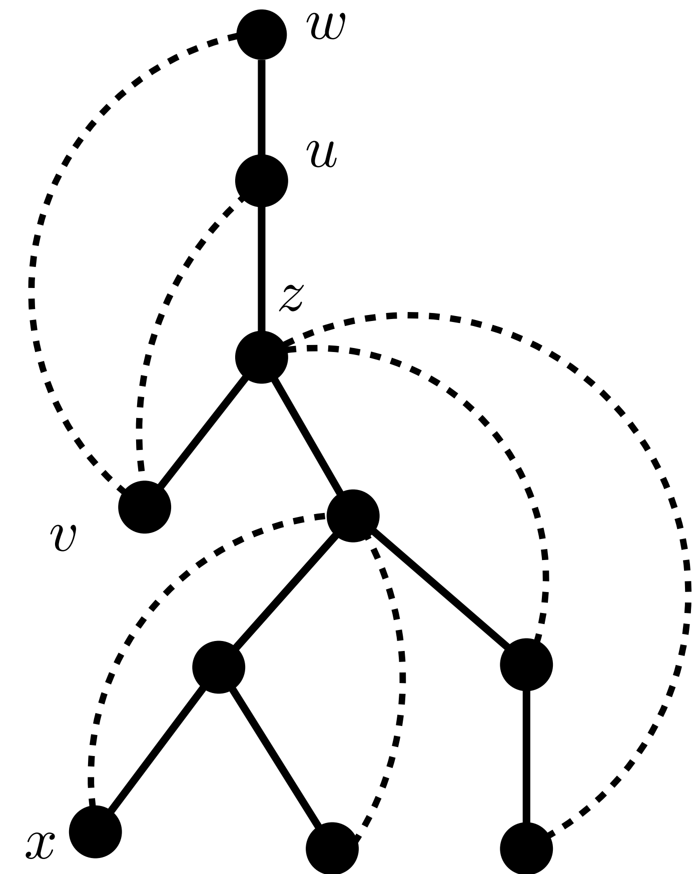




Terminologie

Spannbaum Baum der alle Knoten enthält

Spannbaum mit Wurzel Jeder Knoten (außer der Wurzel) hat eindeutigen Vorgänger

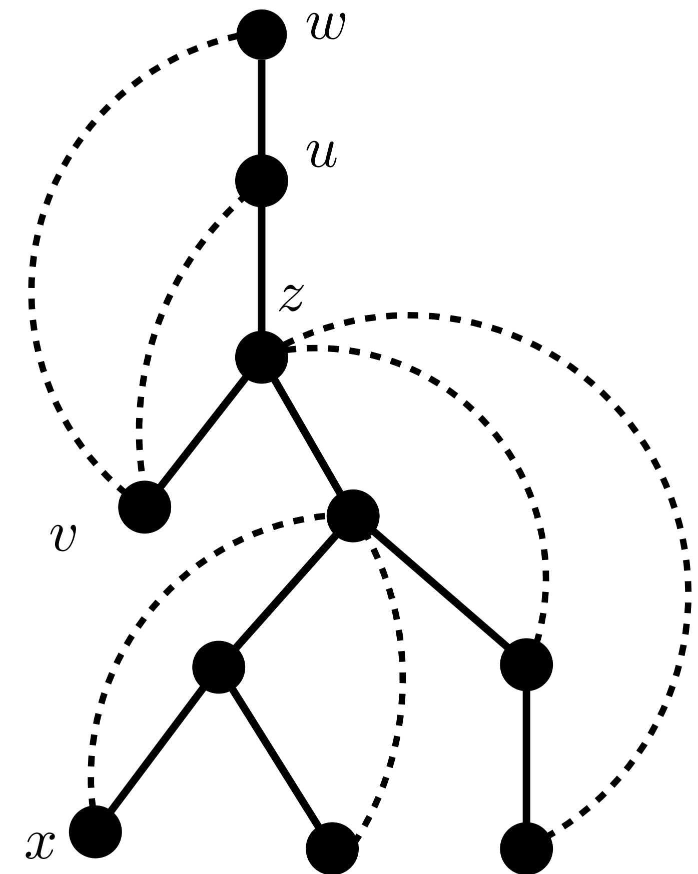


Terminologie

Spannbaum Baum der alle Knoten enthält

Spannbaum mit Wurzel Jeder Knoten (außer der Wurzel) hat eindeutigen Vorgänger

DFS-Baum Spannbaum mit Wurzel, der durch Tiefensuche in Graphen entsteht

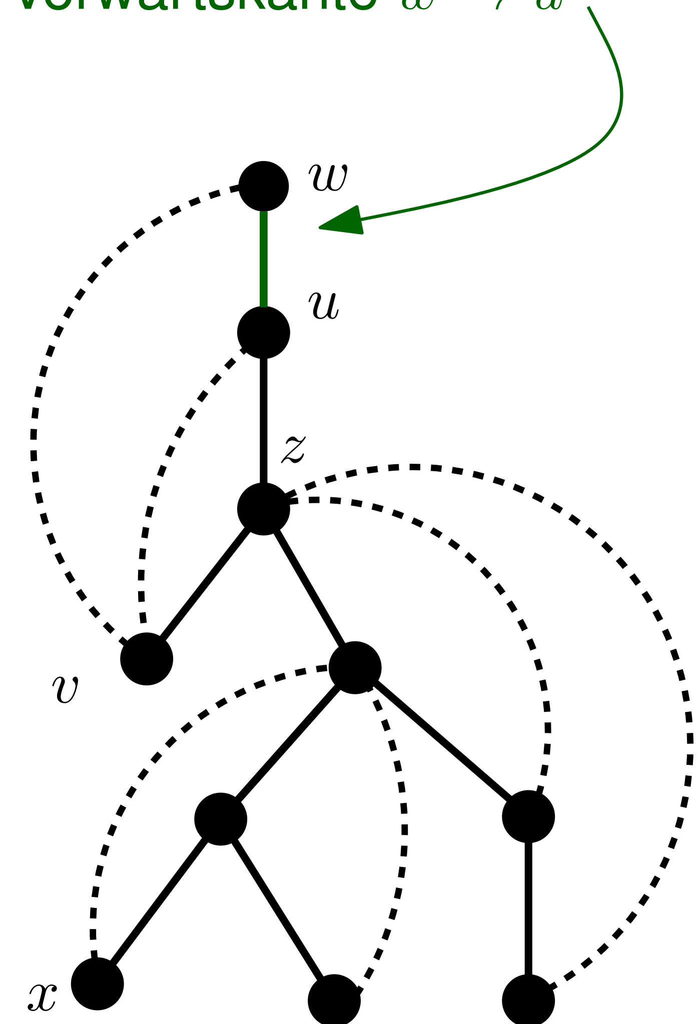


Terminologie

Spannbaum Baum der alle Knoten enthält

Spannbaum mit Wurzel Jeder Knoten (außer der Wurzel) hat eindeutigen Vorgänger

DFS-Baum Spannbaum mit Wurzel, der durch Tiefensuche in Graphen entsteht

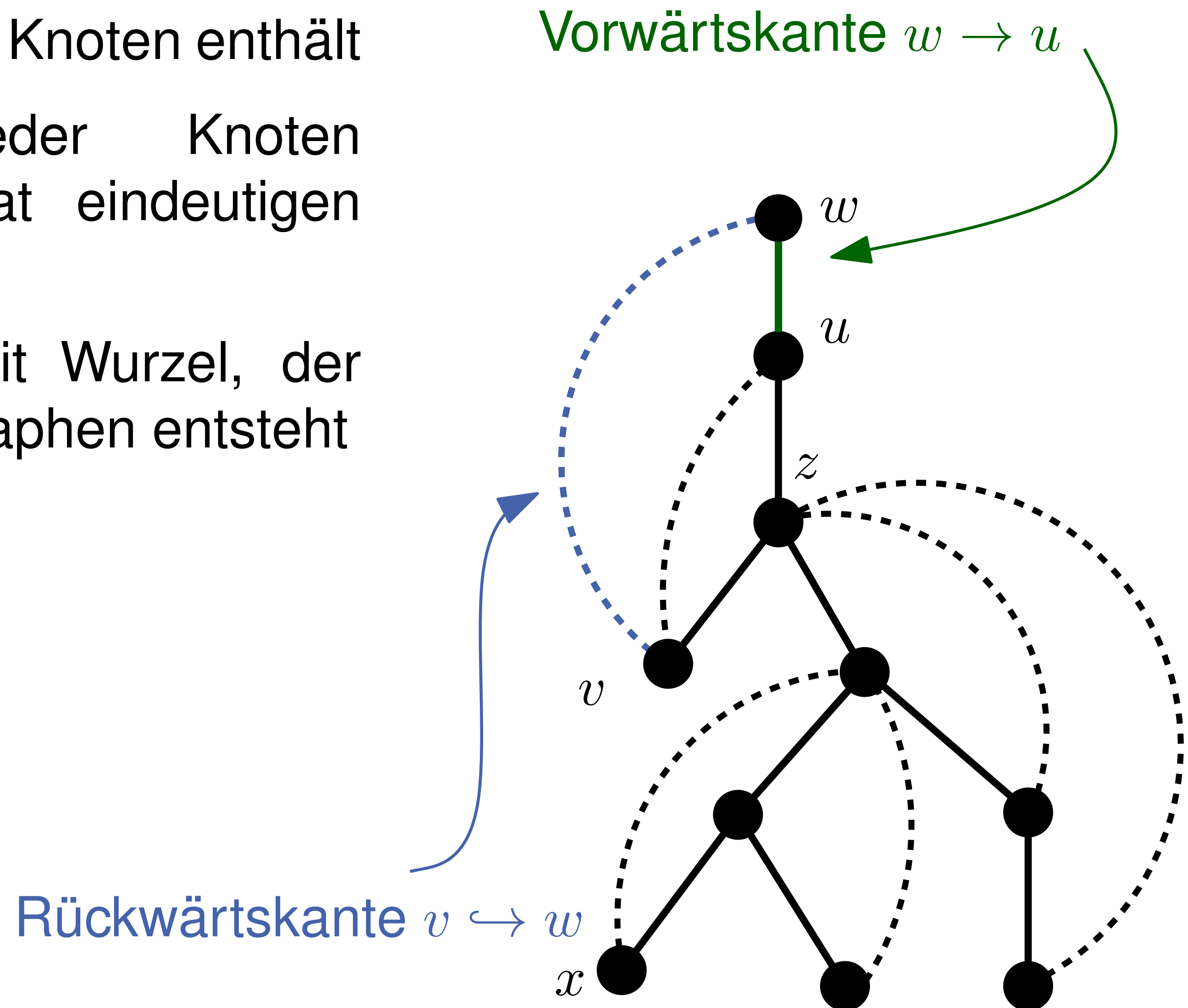


Terminologie

Spannbaum Baum der alle Knoten enthält

Spannbaum mit Wurzel Jeder Knoten (außer der Wurzel) hat eindeutigen Vorgänger

DFS-Baum Spannbaum mit Wurzel, der durch Tiefensuche in Graphen entsteht

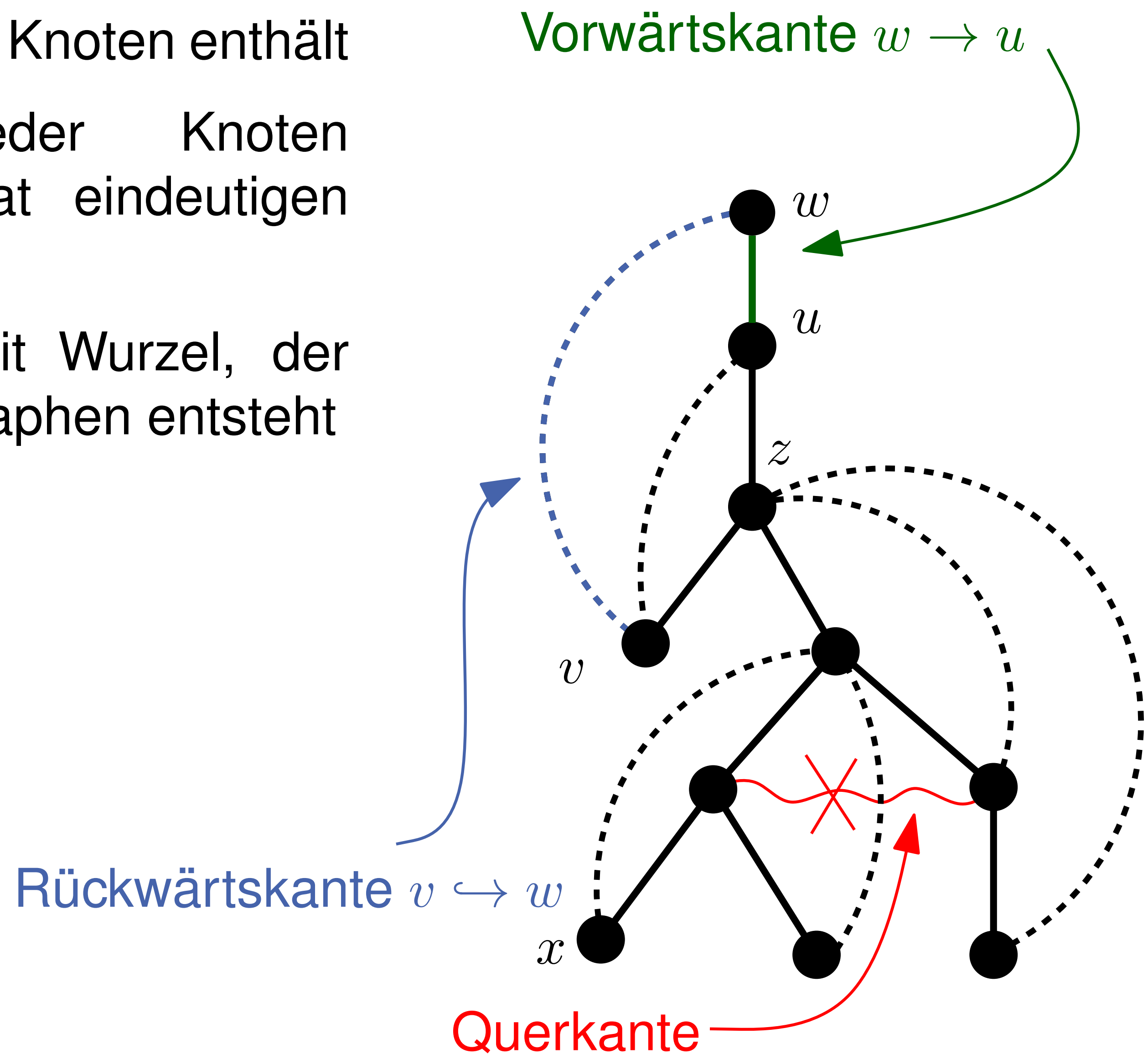


Terminologie

Spannbaum Baum der alle Knoten enthält

Spannbaum mit Wurzel Jeder Knoten (außer der Wurzel) hat eindeutigen Vorgänger

DFS-Baum Spannbaum mit Wurzel, der durch Tiefensuche in Graphen entsteht

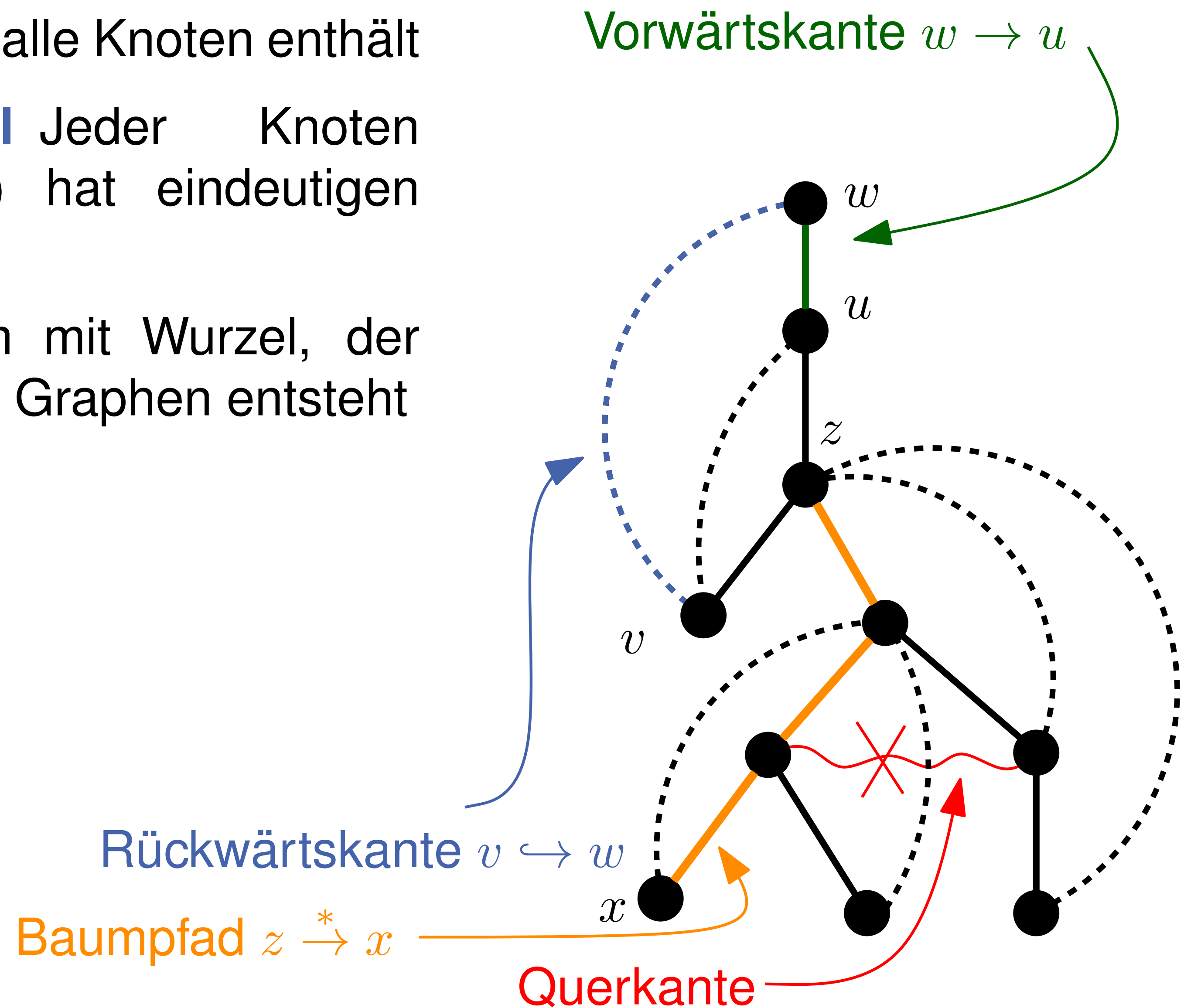


Terminologie

Spannbaum Baum der alle Knoten enthält

Spannbaum mit Wurzel Jeder Knoten (außer der Wurzel) hat eindeutigen Vorgänger

DFS-Baum Spannbaum mit Wurzel, der durch Tiefensuche in Graphen entsteht



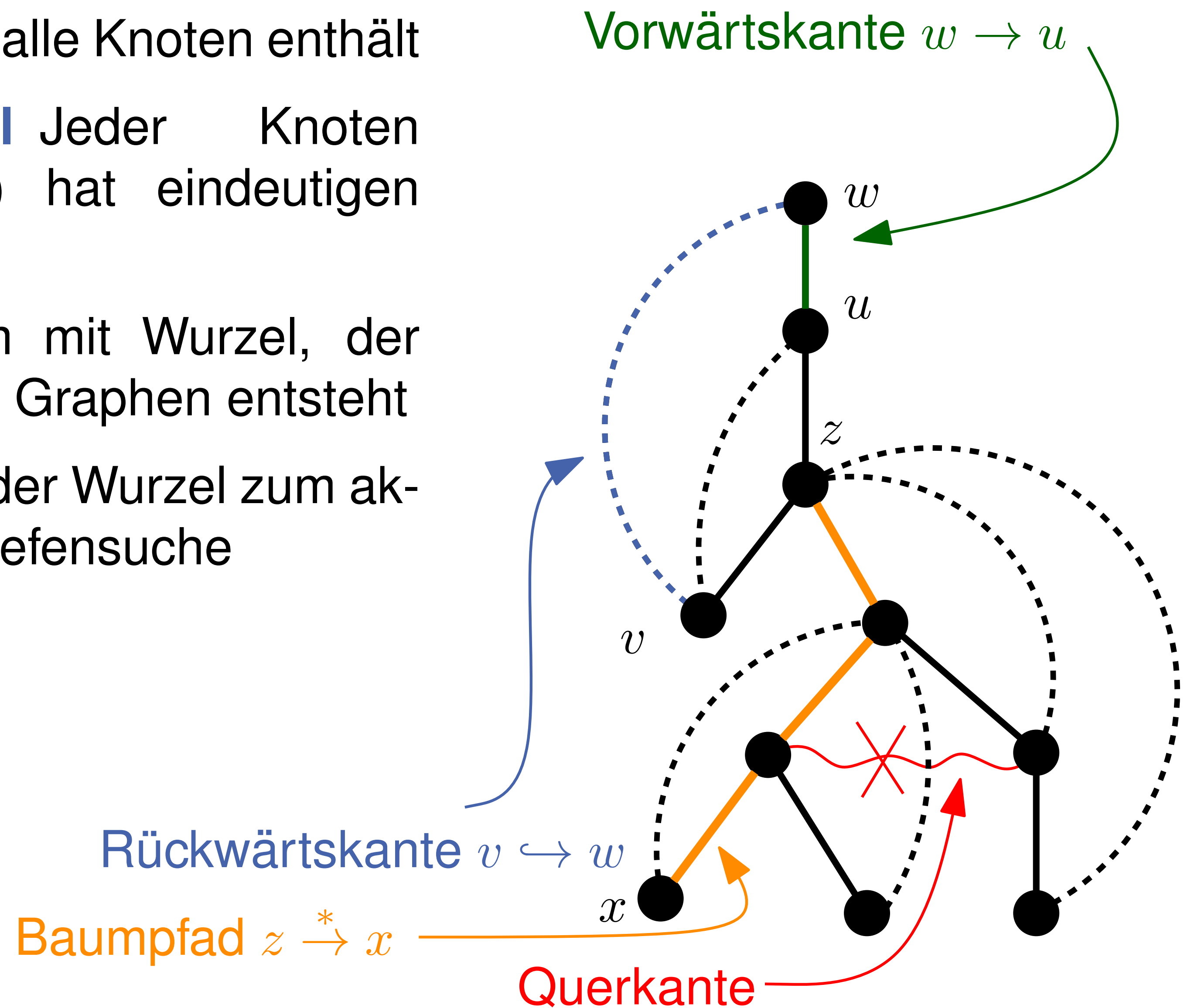
Terminologie

Spannbaum Baum der alle Knoten enthält

Spannbaum mit Wurzel Jeder Knoten (außer der Wurzel) hat eindeutigen Vorgänger

DFS-Baum Spannbaum mit Wurzel, der durch Tiefensuche in Graphen entsteht

offener Pfad Pfad von der Wurzel zum aktuellen Knoten der Tiefensuche



Terminologie

Karlsruhe Institute of Technology

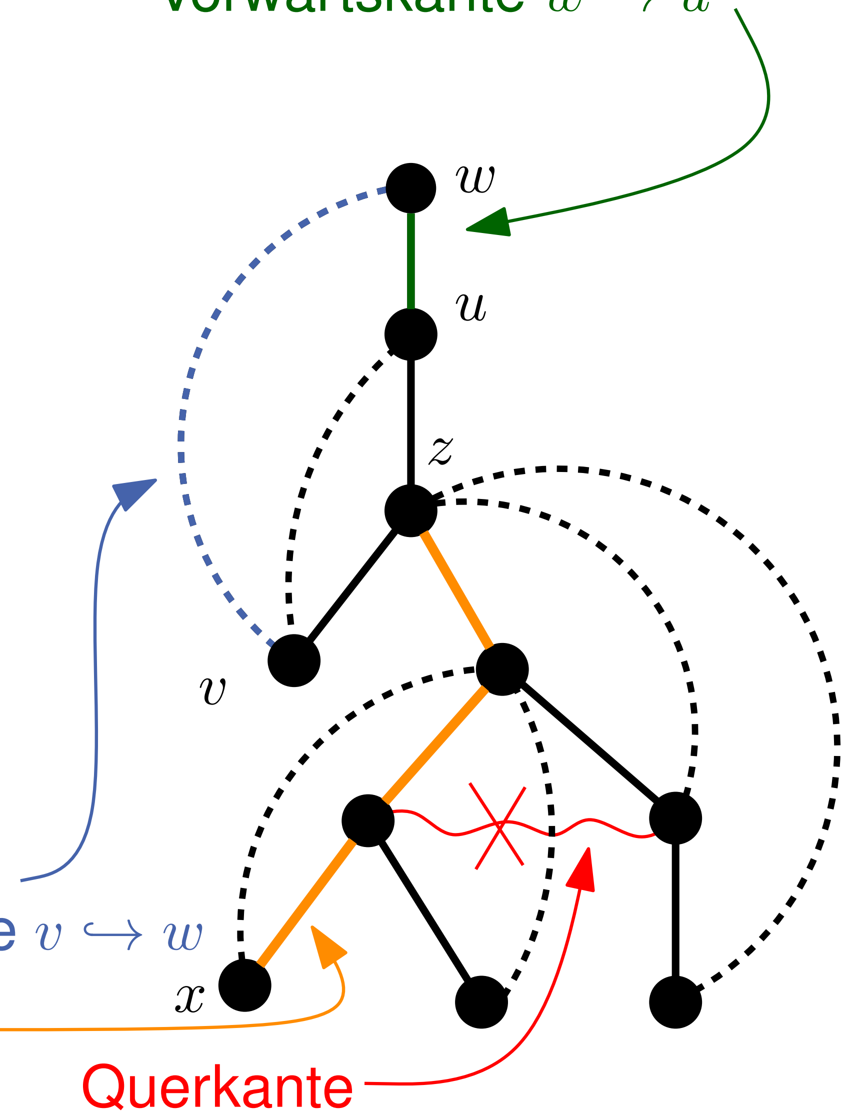
Spannbaum Baum der alle Knoten enthält

Spannbaum mit Wurzel Jeder Knoten (außer der Wurzel) hat eindeutigen Vorgänger

DFS-Baum Spannbaum mit Wurzel, der durch Tiefensuche in Graphen entsteht

offener Pfad Pfad von der Wurzel zum aktuellen Knoten der Tiefensuche

Lowpoint Der Lowpoint w von u ist der Knoten in $T(G)$, der s am nächsten ist, so dass ein Weg $u \xrightarrow{*} v \hookrightarrow w$ existiert.



Berechnung von Blöcken

Karlsruhe Institute of Technology

Beobachtung [Tarjan ’73]

Ein Graph $G = (V, E)$ ist genau dann zweifach zusammenhängend, wenn im DFS-Baum $T(G)$ nur die Wurzel s ihr eigener Lowpoint ist und wenn es höchstens eine Kante $v \to w$ gibt, so dass v der Lowpoint von w ist (dann ist v die Wurzel).

Berechnung von Blöcken

Karlsruhe Institute of Technology

Beobachtung [Tarjan '73]

Ein Graph $G = (V, E)$ ist genau dann zweifach zusammenhängend, wenn im DFS-Baum $T(G)$ nur die Wurzel s ihr eigener Lowpoint ist und wenn es höchstens eine Kante $v \to w$ gibt, so dass v der Lowpoint von w ist (dann ist v die Wurzel).

Algorithmus zur Berechnung von 2-fachen Zusammenhangskomponenten von Tarjan

- 1. Phase: berechne Lowpoints
- 2. Phase: benutze Lowpoint-Information, um Zusammenhangskomponenten zu berechnen
- Nachteil: Graph muss 2 mal durchsucht werden

Berechnung von Blöcken

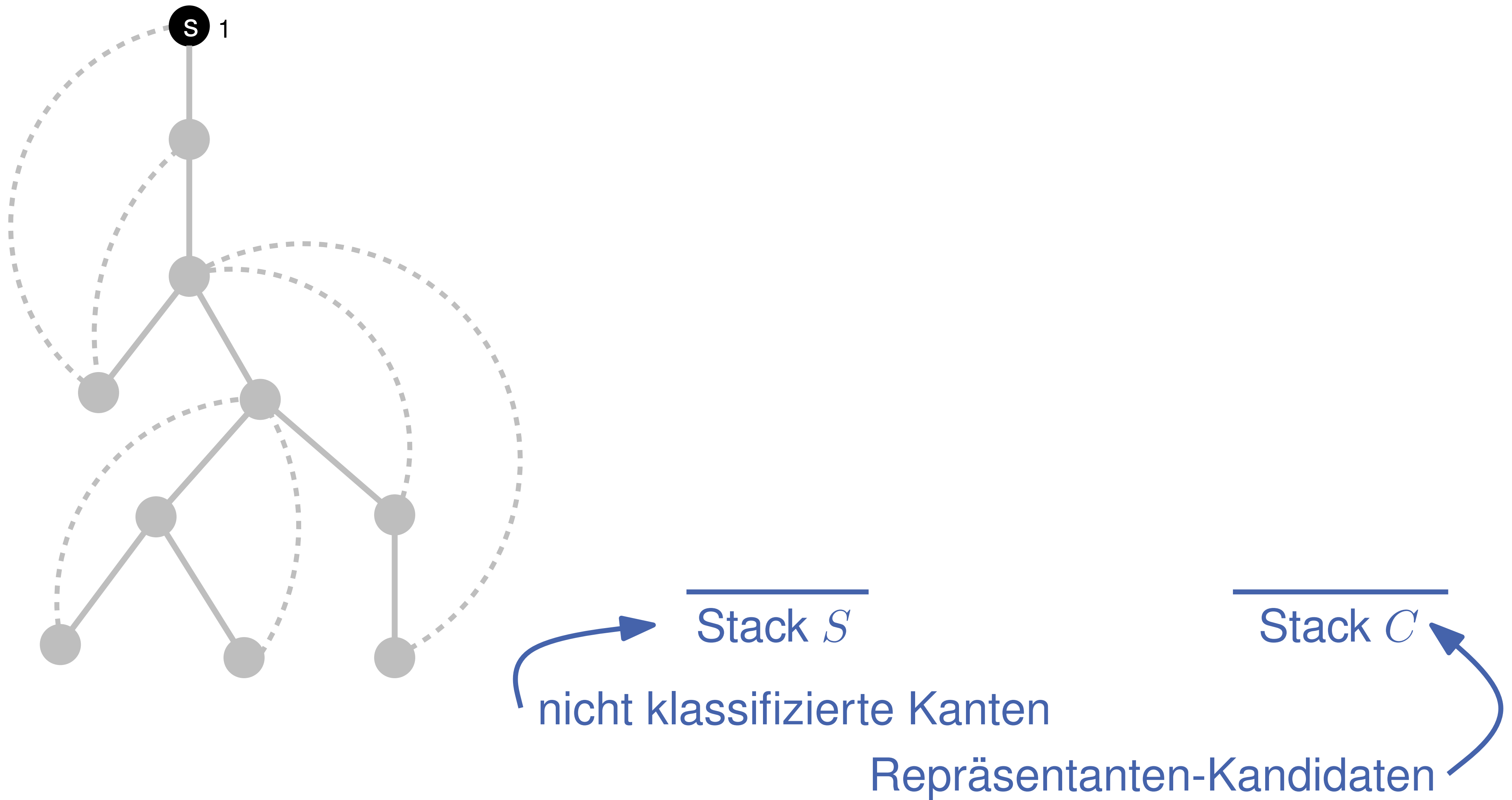
Andere Herangehensweise [Gabow '00]
Pfadbasierte Berechnung von 2-fachen Zusammenhangskomponenten in DFS-Baum

Beobachtung

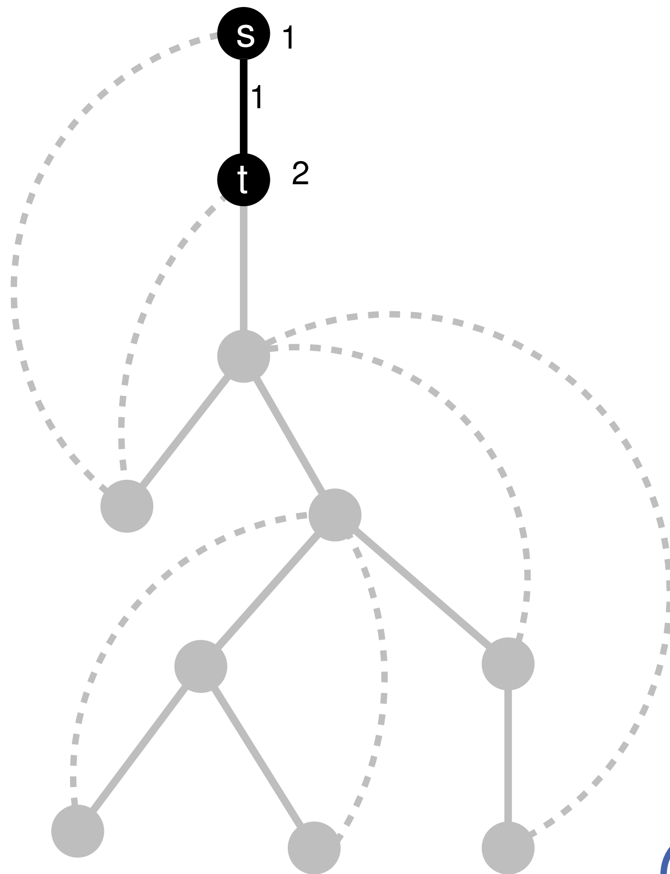
Zwei Kanten sind in der gleichen 2-fachen Zusammenhangskomponenten genau dann, wenn es einen einfachen Kreis gibt, der beide Kanten enthält.

$\rightarrow$ Äquivalenz-Relation auf der Menge der Kanten, benutze "erste" Kante einer Komponente als Repräsentant

Berechnung von Blöcken



Berechnung von Blöcken



$\frac{s \to t}{\text{Stack } S}$

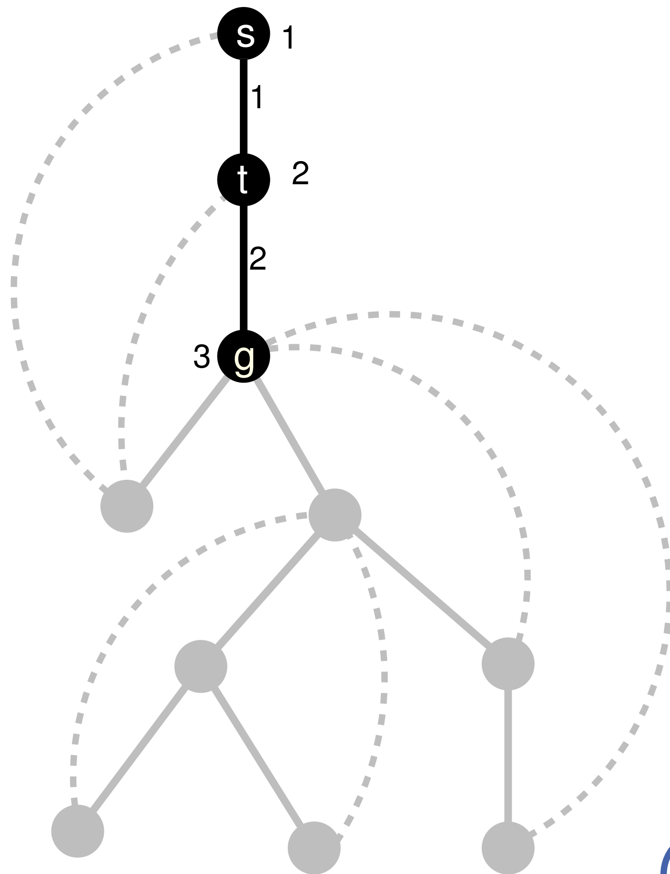
$\frac{s \to t\ (1)}{\text{Stack } C}$

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Institut für Theoretische Informatik
Lehrstuhl Algorithmik I

Berechnung von Blöcken



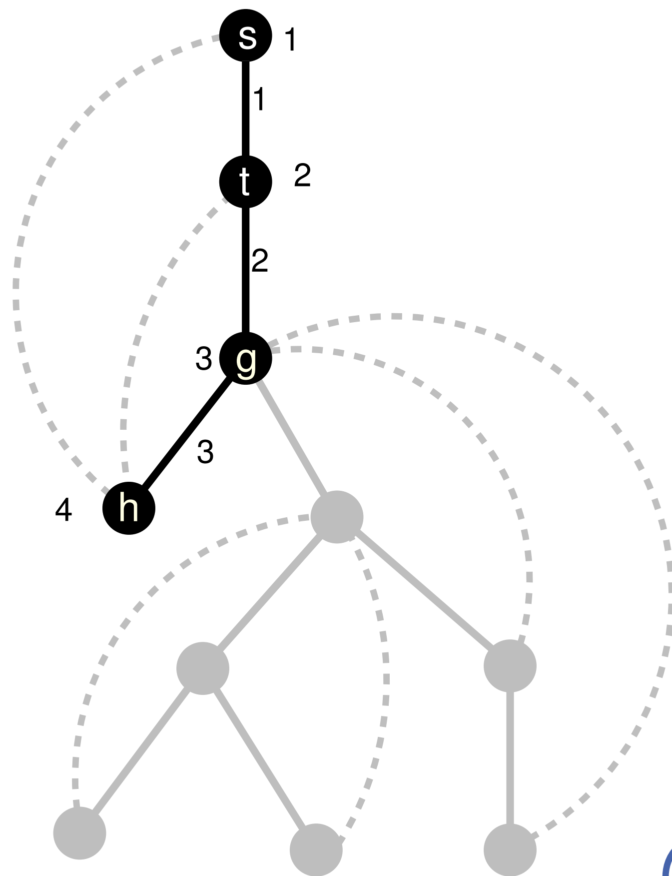
$$\frac{\begin{array}{c} t \to g \\ s \to t \end{array}}{\text{Stack } S}$$

$$\frac{\begin{array}{c} t \to g\ (2) \\ s \to t\ (1) \end{array}}{\text{Stack } C}$$

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



$$g \to h$$
$$t \to g$$
$$s \to t$$

Stack S

$$g \to h\ (3)$$
$$t \to g\ (2)$$
$$s \to t\ (1)$$

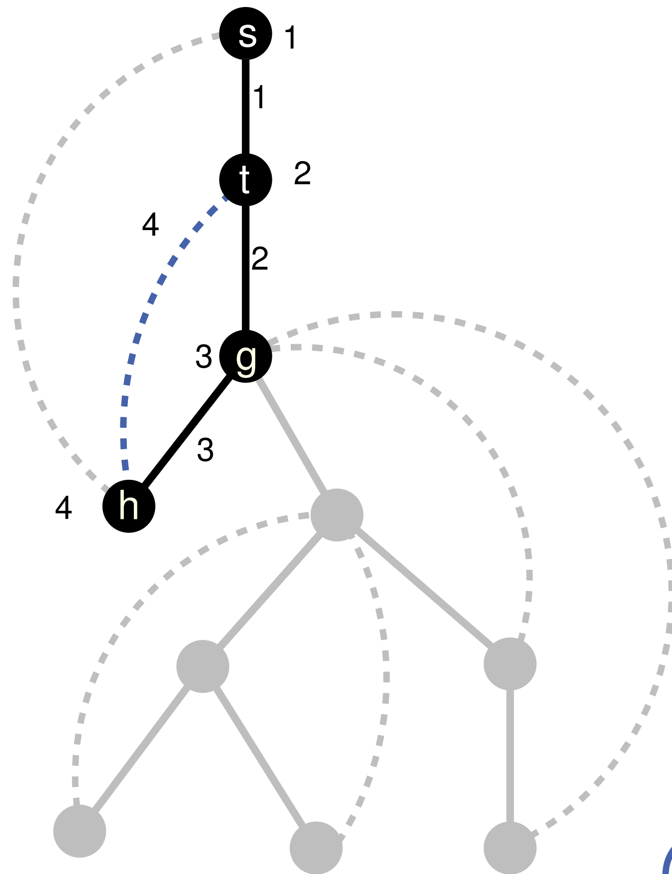
Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Institut für Theoretische Informatik
Lehrstuhl Algorithmik I

Berechnung von Blöcken



$$g \to h$$
$$t \to g$$
$$s \to t$$

Stack S

~~$g \to h$ (3)~~
$t \to g$ (2)
$s \to t$ (1)

Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken

Karlsruhe Institute of Technology

$g \to h$
$t \to g$
$s \to t$

Stack S

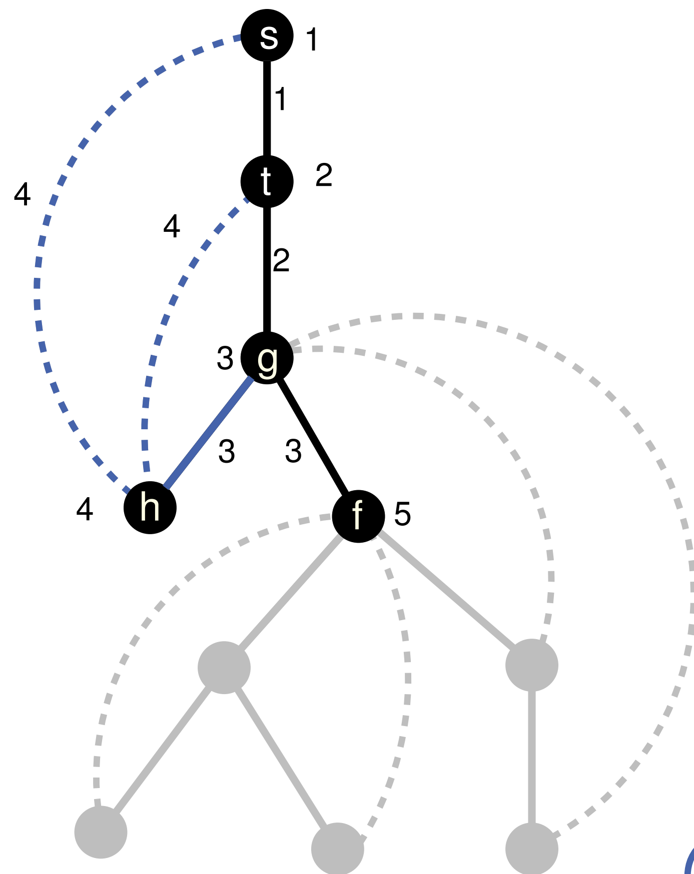
~~$g \to h$ (3)~~
~~$t \to g$ (2)~~
$s \to t$ (1)

Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



$g \to f$
$g \to h$
$t \to g$
$s \to t$

Stack S

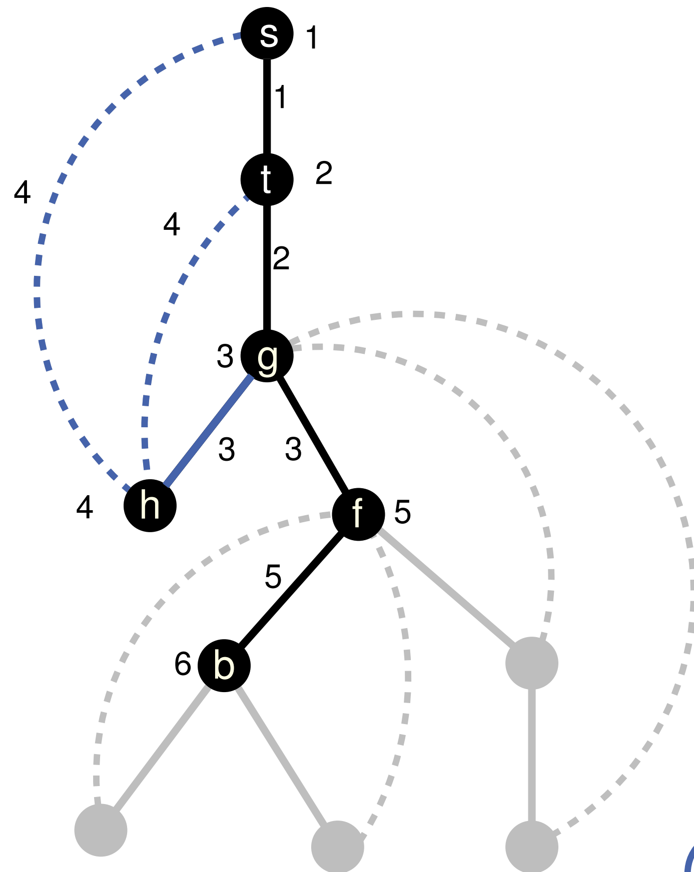
nicht klassifizierte Kanten

$g \to f$ (3)
~~$g \to h$ (3)~~
~~$t \to g$ (2)~~
$s \to t$ (1)

Stack C

Repräsentanten-Kandidaten

Berechnung von Blöcken



$f \to b$
$g \to f$
$g \to h$
$t \to g$
$s \to t$

Stack S

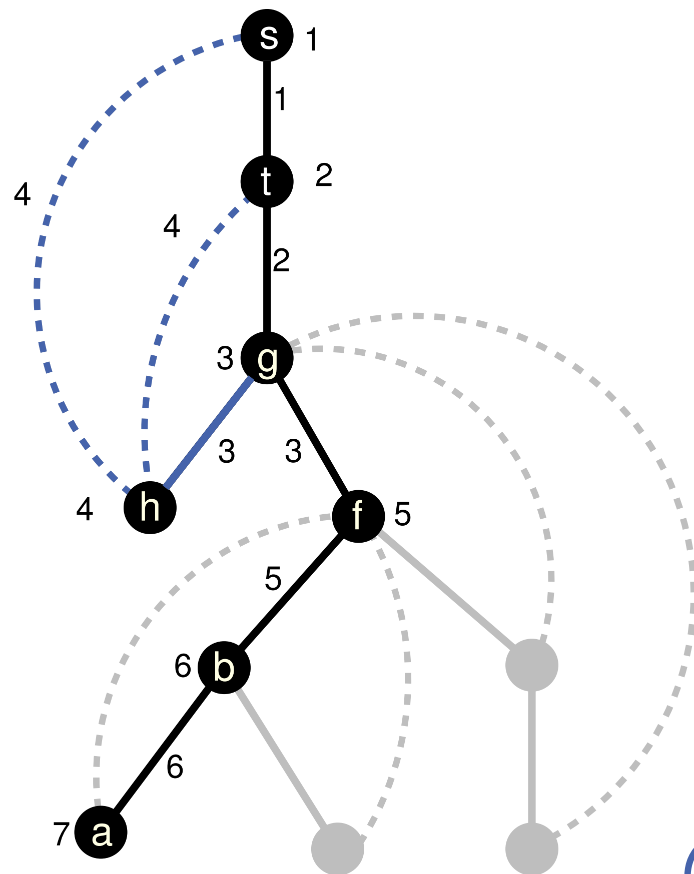
$f \to b$ (5)
$g \to f$ (3)
~~$g \to h$ (3)~~
~~$t \to g$ (2)~~
$s \to t$ (1)

Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



$b \to a$
$f \to b$
$g \to f$
$g \to h$
$t \to g$
$s \to t$

Stack S

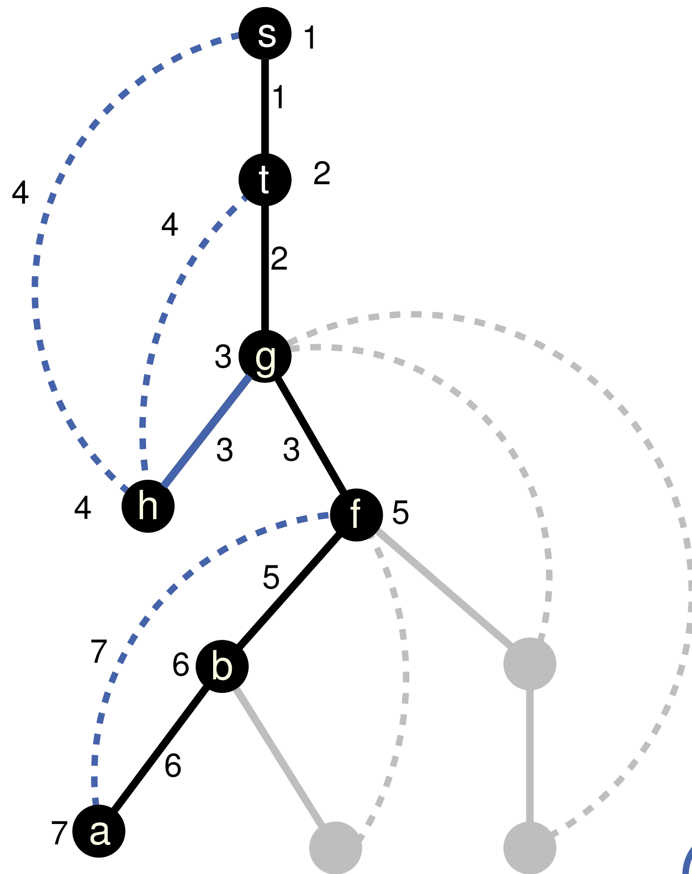
$b \to a$ (6)
$f \to b$ (5)
$g \to f$ (3)
~~$g \to h$ (3)~~
~~$t \to g$ (2)~~
$s \to t$ (1)

Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



$b \to a$
$f \to b$
$g \to f$
$g \to h$
$t \to g$
$s \to t$

Stack S

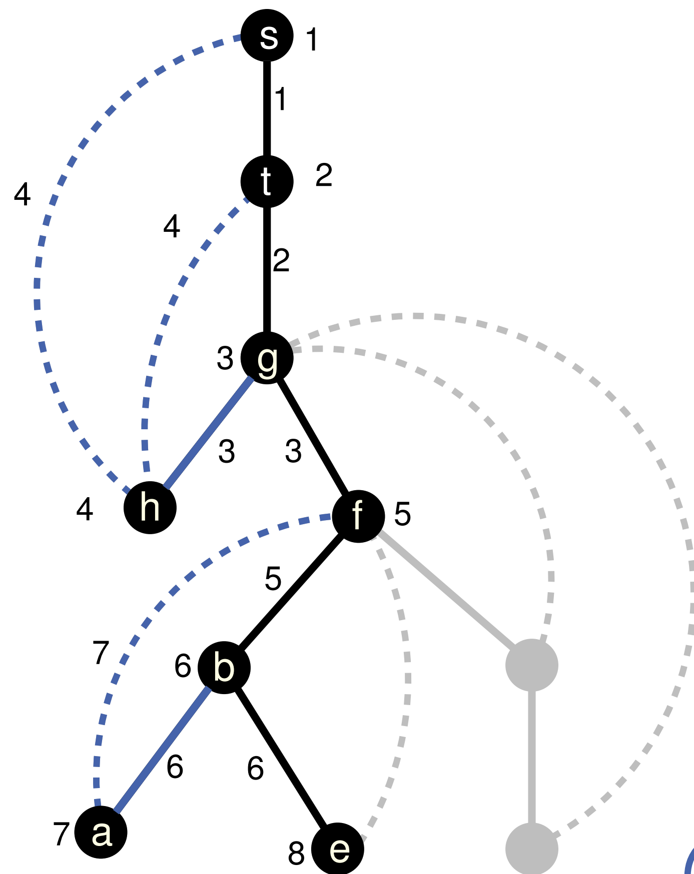
~~$b \to a$ (6)~~
$f \to b$ (5)
$g \to f$ (3)
~~$g \to h$ (3)~~
~~$t \to g$ (2)~~
$s \to t$ (1)

Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



$b \to e$
$b \to a$
$f \to b$
$g \to f$
$g \to h$
$t \to g$
$s \to t$

Stack S

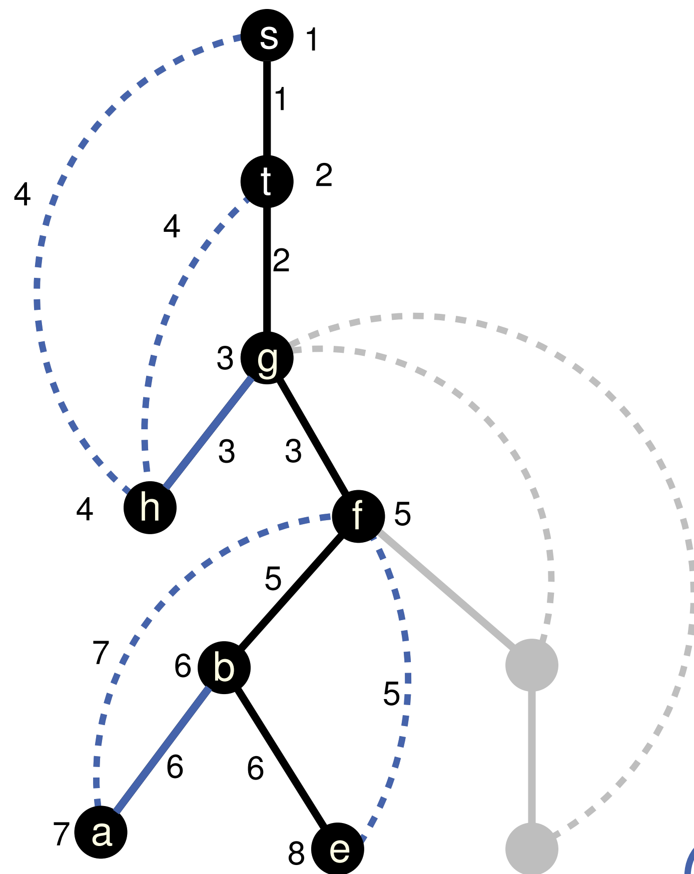
$b \to e$ (6)
~~$b \to a$ (6)~~
$f \to b$ (5)
$g \to f$ (3)
~~$g \to h$ (3)~~
~~$t \to g$ (2)~~
$s \to t$ (1)

Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



$b \to e$
$b \to a$
$f \to b$
$g \to f$
$g \to h$
$t \to g$
$s \to t$

Stack S

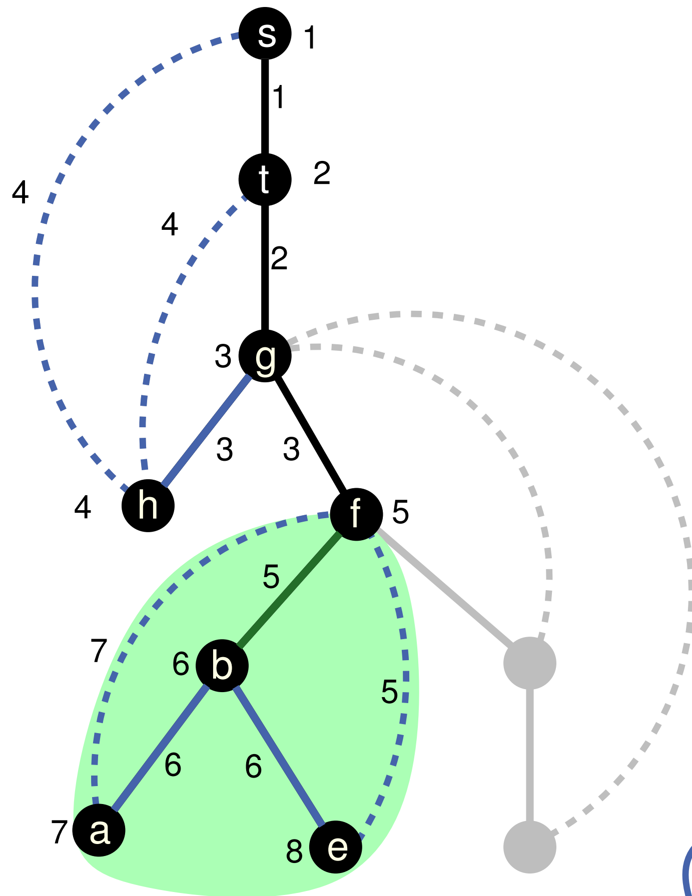
~~$b \to e$ (6)~~
~~$b \to a$ (6)~~
$f \to b$ (5)
$g \to f$ (3)
~~$g \to h$ (3)~~
~~$t \to g$ (2)~~
$s \to t$ (1)

Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



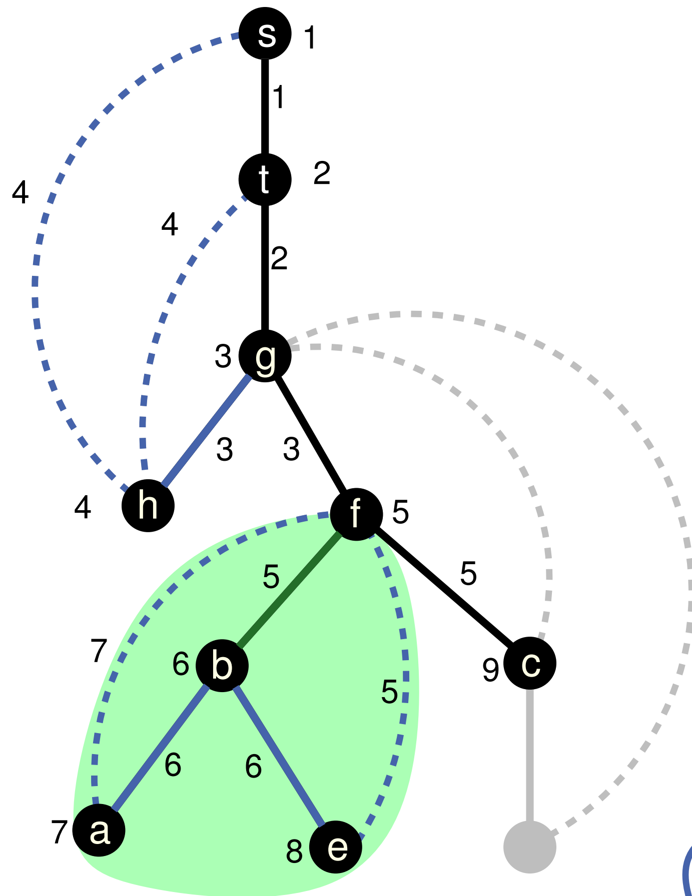
Stack S
~~$b \to c$~~
~~$b \to a$~~
~~$f \to b$~~
$g \to f$
$g \to h$
$t \to g$
$s \to t$

Stack C
~~$b \to e$ (6)~~
~~$b \to a$ (6)~~
~~$f \to b$ (5)~~
$g \to f$ (3)
~~$g \to h$ (3)~~
~~$t \to g$ (2)~~
$s \to t$ (1)

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



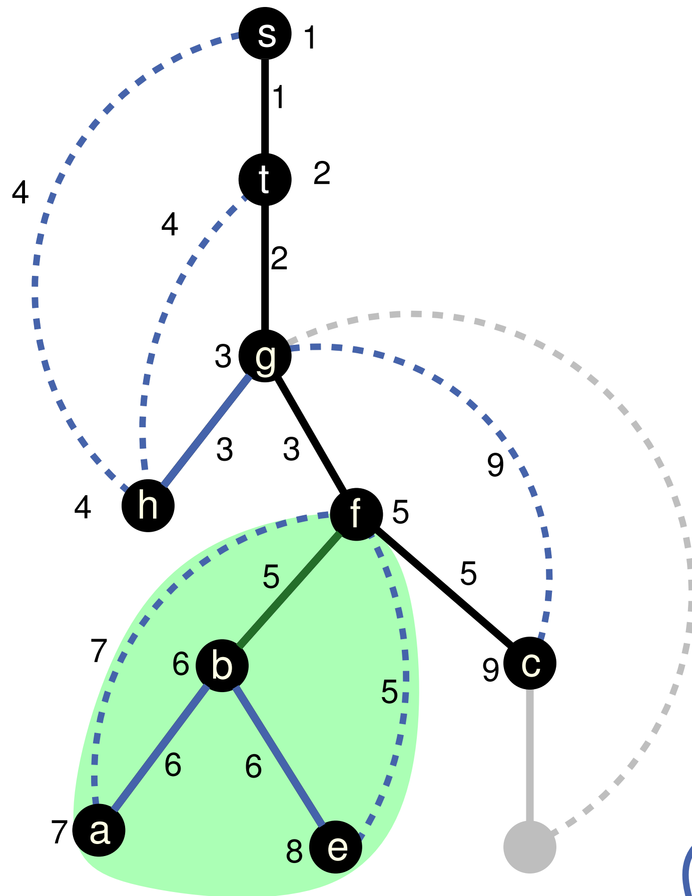
Stack S
$f \rightarrow c$
~~$b \rightarrow e$~~
~~$b \rightarrow a$~~
~~$f \rightarrow b$~~
$g \rightarrow f$
$g \rightarrow h$
$t \rightarrow g$
$s \rightarrow t$

Stack C
$f \rightarrow c$ (5)
~~$b \rightarrow e$ (6)~~
~~$b \rightarrow a$ (6)~~
~~$f \rightarrow b$ (5)~~
$g \rightarrow f$ (3)
~~$g \rightarrow h$ (3)~~
~~$t \rightarrow g$ (2)~~
$s \rightarrow t$ (1)

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



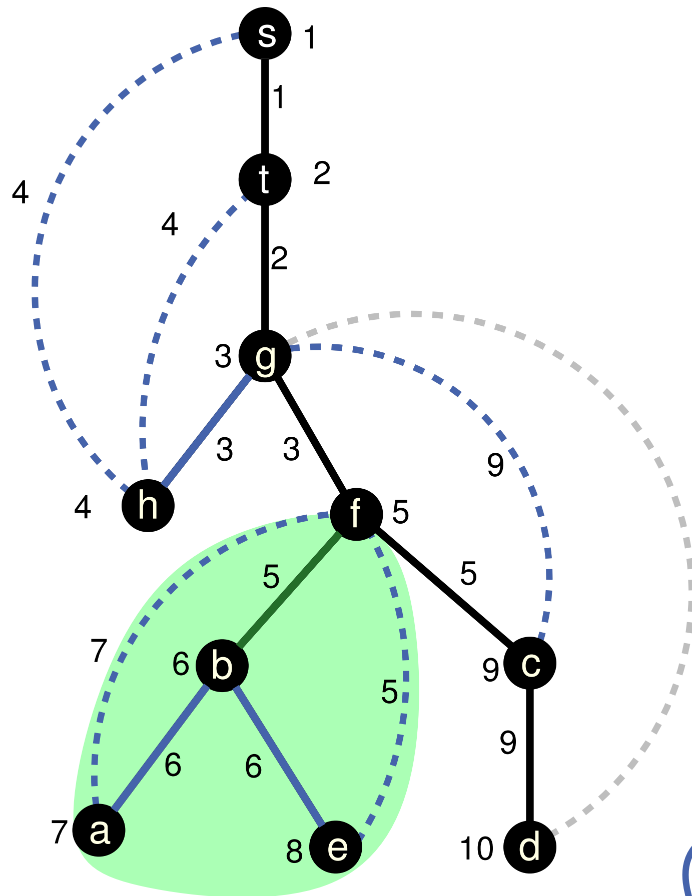
Stack S	Stack C
$f \to c$	~~$f \to c$ (5)~~
~~$b \to e$~~	~~$b \to e$ (6)~~
~~$b \to a$~~	~~$b \to a$ (6)~~
~~$f \to b$~~	~~$f \to b$ (5)~~
$g \to f$	$g \to f$ (3)
$g \to h$	~~$g \to h$ (3)~~
$t \to g$	~~$t \to g$ (2)~~
$s \to t$	$s \to t$ (1)

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken

KIT Karlsruhe Institute of Technology



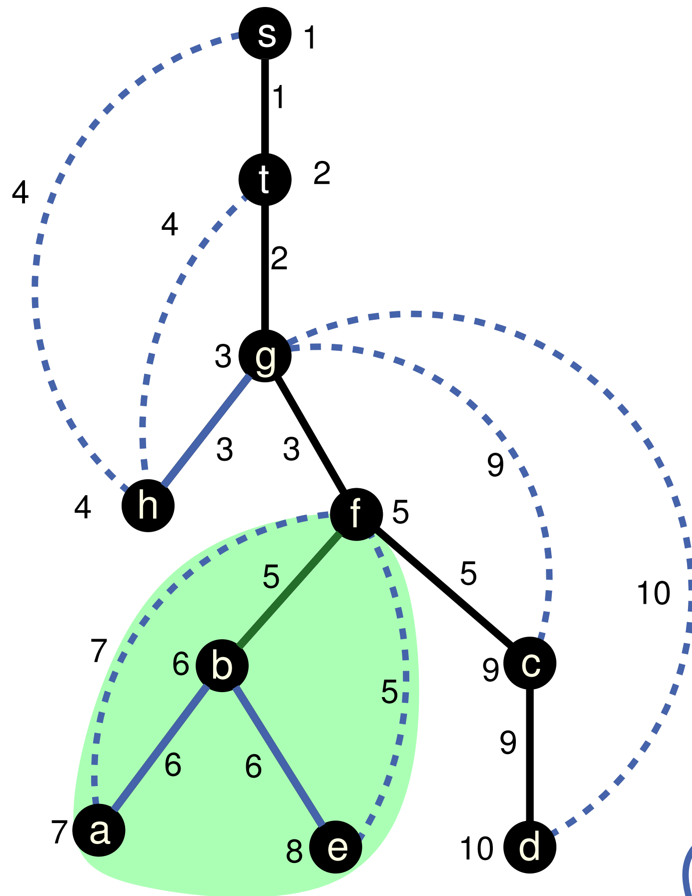
$c \to d$
~~$f \to c$~~ (not struck on Stack S: $f \to c$)

Stack S	Stack C
$c \to d$	$c \to d$ (9)
$f \to c$	~~$f \to c$ (5)~~
~~$b \to e$~~	~~$b \to e$ (6)~~
~~$b \to a$~~	~~$b \to a$ (6)~~
~~$f \to b$~~	~~$f \to b$ (5)~~
$g \to f$	$g \to f$ (3)
$g \to h$	~~$g \to h$ (3)~~
$t \to g$	~~$t \to g$ (2)~~
$s \to t$	$s \to t$ (1)

nicht klassifizierte Kanten → Stack S

Repräsentanten-Kandidaten → Stack C

Berechnung von Blöcken



$c \to d$
$f \to c$
~~$b \to e$~~
~~$b \to a$~~
~~$f \to b$~~
$g \to f$
$g \to h$
$t \to g$
$s \to t$

Stack S

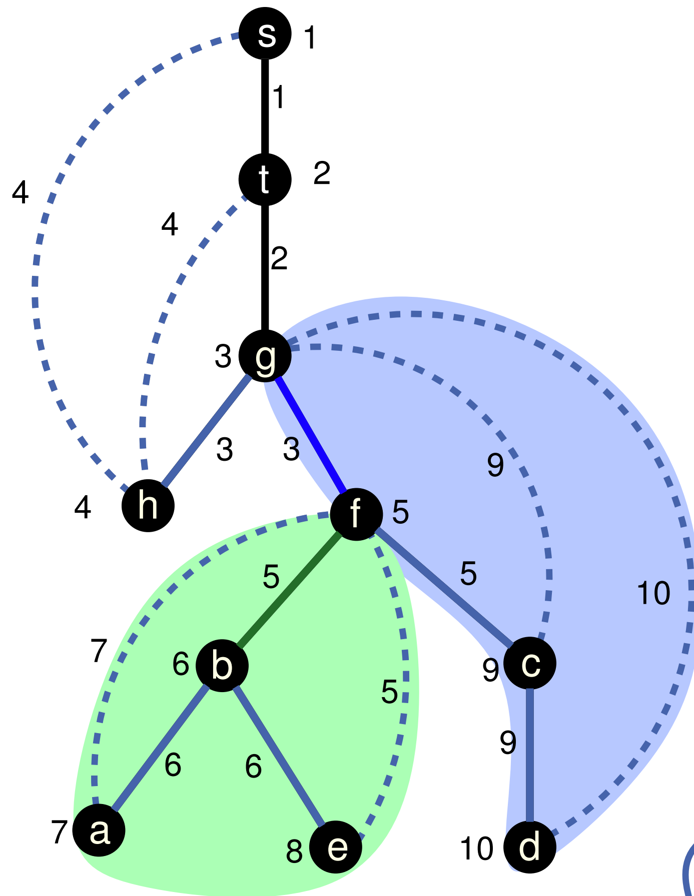
~~$c \to d$ (9)~~
~~$f \to c$ (5)~~
~~$b \to e$ (6)~~
~~$b \to a$ (6)~~
~~$f \to b$ (5)~~
$g \to f$ (3)
~~$g \to h$ (3)~~
~~$t \to g$ (2)~~
$s \to t$ (1)

Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



~~$c \to d$~~
~~$f \to c$~~
~~$b \to e$~~
~~$b \to a$~~
~~$f \to b$~~
~~$g \to f$~~
$g \to h$
$t \to g$
$s \to t$

Stack S

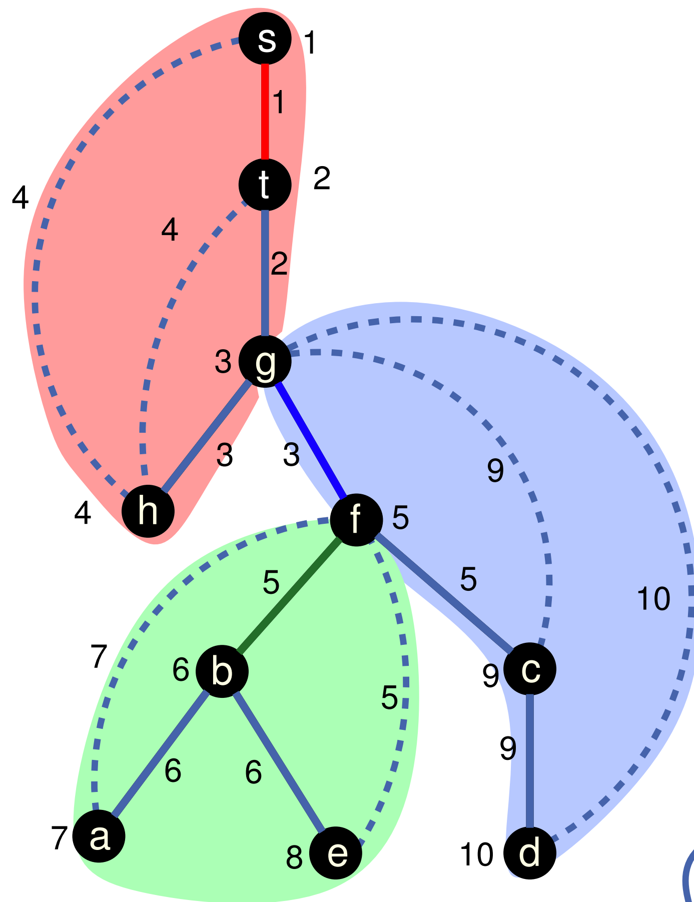
~~$c \to d$ (9)~~
~~$f \to c$ (5)~~
~~$b \to e$ (6)~~
~~$b \to a$ (6)~~
~~$f \to b$ (5)~~
~~$g \to f$ (3)~~
~~$g \to h$ (3)~~
~~$t \to g$ (2)~~
$s \to t$ (1)

Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

Berechnung von Blöcken



~~$c \rightarrow d$~~
~~$f \rightarrow c$~~
~~$b \rightarrow e$~~
~~$b \rightarrow a$~~
~~$f \rightarrow b$~~
~~$g \rightarrow f$~~
~~$g \rightarrow h$~~
~~$t \rightarrow g$~~
~~$s \rightarrow t$~~

Stack S

~~$c \rightarrow d$ (9)~~
~~$f \rightarrow c$ (5)~~
~~$b \rightarrow e$ (6)~~
~~$b \rightarrow a$ (6)~~
~~$f \rightarrow b$ (5)~~
~~$g \rightarrow f$ (3)~~
~~$g \rightarrow h$ (3)~~
~~$t \rightarrow g$ (2)~~
~~$s \rightarrow t$ (1)~~

Stack C

nicht klassifizierte Kanten

Repräsentanten-Kandidaten

st-Ordnung

Definition

Sei $G = (V, E)$ ein zweifach zusammenhängender Graph und $s \neq t \in V$. Eine Ordung $s = v_1, v_2, \ldots, v_n = t$ der Knoten von G heißt *s-t-Ordnung*, falls für alle Knoten v_j, $1 < j < n$, gilt:

$$\text{es ex.} \quad 1 \leq i < j < k \leq n \quad \text{mit} \quad \{v_i, v_j\}, \{v_j, v_k\} \in E \ .$$

st-Ordnung

Definition

Sei $G = (V, E)$ ein zweifach zusammenhängender Graph und $s \neq t \in V$. Eine Ordung $s = v_1, v_2, \ldots, v_n = t$ der Knoten von G heißt s-t-Ordnung, falls für alle Knoten v_j, $1 < j < n$, gilt:

$$\text{es ex.} \quad 1 \leq i < j < k \leq n \quad \text{mit} \quad \{v_i, v_j\}, \{v_j, v_k\} \in E\ .$$

Satz [Lempel, Even, Cederbaum, 1967]

Ist $G = (V, E)$ ein zweifach zusammenhängender Graph, dann existiert zu jeder Kante $(s, t) \in E$ eine s-t-Ordnung.

Berechnung einer st-Ordnung

Karlsruhe Institute of Technology

st-Orientierung

Sei $G = (V, E)$ ein Graph. Eine st-Orientierung (bipolar orientation) von G ist eine Orientierung der Kanten, so dass der resultierende Graph azyklisch ist und s die einzige Quelle sowie t die einzige Senke sind.

Berechnung einer st-Ordnung

Karlsruhe Institute of Technology

st-Orientierung

Sei $G = (V, E)$ ein Graph. Eine st-Orientierung (bipolar orientation) von G ist eine Orientierung der Kanten, so dass der resultierende Graph azyklisch ist und s die einzige Quelle sowie t die einzige Senke sind.

Satz

G hat eine st-Ordnung genau dann, wenn er eine st-Orientierung hat.

Berechnung einer st-Orientierung aus Ohrenzerlegung

Ohrenzerlegung

KIT Karlsruhe Institute of Technology

Definition

Sei $G = (V, E)$ ein 2-fach zusammenhängender Graph. Ein Ohr von G ist ein einfacher Pfad oder Kreis $O = (v_0, \{v_0, v_1\}, v_1, \ldots, v_{k-1}, \{v_{k-1}, v_k\}, v_k)$. Ein Ohr heißt offen, falls $v_0 \neq v_k$ ist, d.h. falls O ein einfacher Pfad ist.

Ohrenzerlegung

Definition

Sei $G = (V, E)$ ein 2-fach zusammenhängender Graph. Ein Ohr von G ist ein einfacher Pfad oder Kreis $O = (v_0, \{v_0, v_1\}, v_1, \ldots, v_{k-1}, \{v_{k-1}, v_k\}, v_k)$. Ein Ohr heißt offen, falls $v_0 \neq v_k$ ist, d.h. falls O ein einfacher Pfad ist.

Eine Folge $D = (P_0, \ldots, P_r)$ von (offenen) Pfaden heißt (offene) Ohrendekomposition, falls $E(P_0), \ldots, E(P_r)$ eine Partition von E ist und für alle $P_i = (v_0, e_1, v_1, \ldots, e_k, v_k)$, $1 \leq i \leq r$, gilt $\{v_0, v_k\} \subseteq V_{i-1}$ und $\{v_1, \ldots, v_{k-1}\} \cap V_{i-1} = \emptyset$.

Ohrenzerlegung

Karlsruhe Institute of Technology

Definition

Sei $G = (V, E)$ ein 2-fach zusammenhängender Graph. Ein Ohr von G ist ein einfacher Pfad oder Kreis $O = (v_0, \{v_0, v_1\}, v_1, \dots, v_{k-1}, \{v_{k-1}, v_k\}, v_k)$. Ein Ohr heißt offen, falls $v_0 \neq v_k$ ist, d.h. falls O ein einfacher Pfad ist.

Eine Folge $D = (P_0, \dots, P_r)$ von (offenen) Pfaden heißt (offene) Ohrendekomposition, falls $E(P_0), \dots, E(P_r)$ eine Partition von E ist und für alle $P_i = (v_0, e_1, v_1, \dots, e_k, v_k)$, $1 \leq i \leq r$, gilt $\{v_0, v_k\} \subseteq V_{i-1}$ und $\{v_1, \dots, v_{k-1}\} \cap V_{i-1} = \emptyset$.

Eine Ohrendekomposition beginnt mit $\{s, t\} \in E$, falls $P_0 = (s, \{s, t\}, t)$.

Ohrenzerlegung

Karlsruhe Institute of Technology

Definition

Sei $G = (V, E)$ ein 2-fach zusammenhängender Graph. Ein Ohr von G ist ein einfacher Pfad oder Kreis $O = (v_0, \{v_0, v_1\}, v_1, \dots, v_{k-1}, \{v_{k-1}, v_k\}, v_k)$. Ein Ohr heißt offen, falls $v_0 \neq v_k$ ist, d.h. falls O ein einfacher Pfad ist.

Eine Folge $D = (P_0, \dots, P_r)$ von (offenen) Pfaden heißt (offene) Ohrendekomposition, falls $E(P_0), \dots, E(P_r)$ eine Partition von E ist und für alle $P_i = (v_0, e_1, v_1, \dots, e_k, v_k)$, $1 \leq i \leq r$, gilt $\{v_0, v_k\} \subseteq V_{i-1}$ und $\{v_1, \dots, v_{k-1}\} \cap V_{i-1} = \emptyset$.

Eine Ohrendekomposition beginnt mit $\{s, t\} \in E$, falls $P_0 = (s, \{s, t\}, t)$.

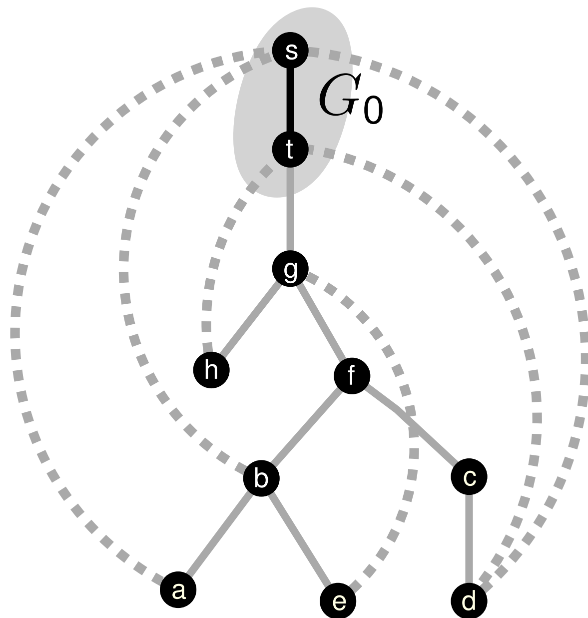
Eine Ohrendekomposition induziert Graphen $G_i = (V_i, E_i)$ mit $V_i = \bigcup_{j=0}^{i} V(P_j)$ und $E_i = \bigcup_{j=0}^{i} E(P_j)$, $0 \leq i \leq r$.

Ohrenzerlegung

Karlsruhe Institute of Technology

Satz [Whitney '32]

Ist G ein zweifach zusammenhängender Graph, dann existiert für jede Kante $\{s,t\} \in E$ eine offene Ohrendekomposition, die mit $\{s,t\}$ beginnt.



Iteratives Vorgehen [Brandes '02]

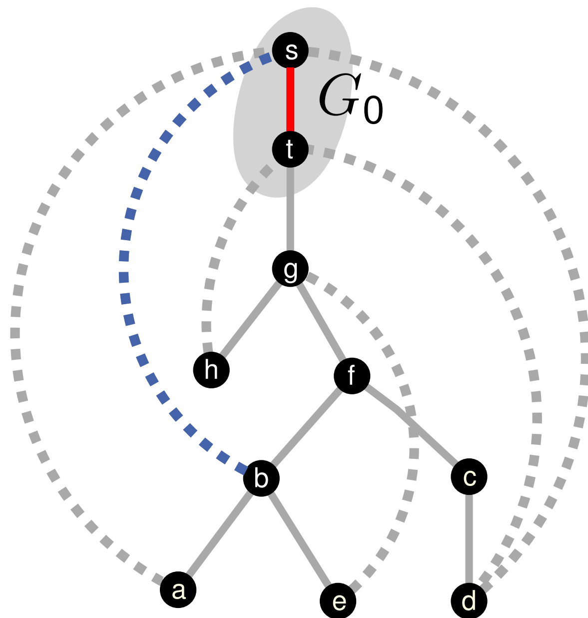
- $P_0 = (s\{s,t\},t)$
- solange $\exists\ w \rightarrow x \in E_i$, $v \hookrightarrow w \notin E_i$ mit $x \overset{*}{\rightarrow} v$ und $w, x \in V_i$
- u letzter Knoten in V_i auf $x \overset{*}{\rightarrow} v$

$\Rightarrow$ $P_{i+1} = u \overset{*}{\rightarrow} v \hookrightarrow w$ (Ohr zu $v \hookrightarrow w$)

Ohrenzerlegung

Satz [Whitney '32]

Ist G ein zweifach zusammenhängender Graph, dann existiert für jede Kante $\{s,t\} \in E$ eine offene Ohrendekomposition, die mit $\{s,t\}$ beginnt.



Iteratives Vorgehen [Brandes '02]

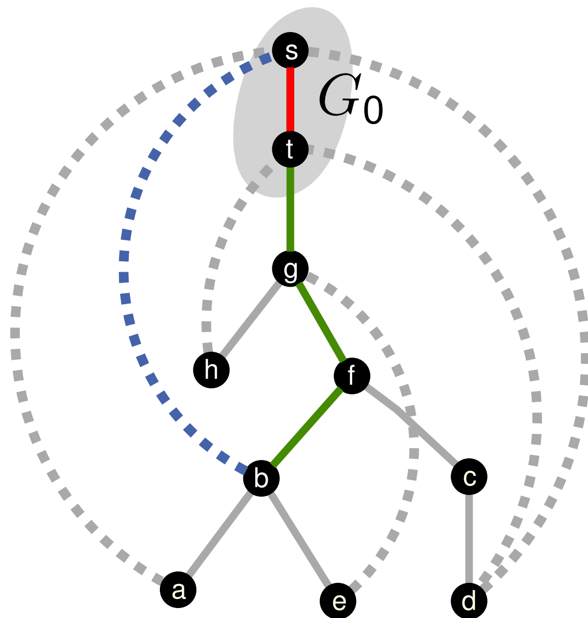
- $P_0 = (s\{s,t\},t)$
- solange $\exists\, w \to x \in E_i,\, v \hookrightarrow w \notin E_i$ mit $x \xrightarrow{*} v$ und $w, x \in V_i$
- u letzter Knoten in V_i auf $x \xrightarrow{*} v$

$\Rightarrow$ $P_{i+1} = u \xrightarrow{*} v \hookrightarrow w$ (Ohr zu $v \hookrightarrow w$)

Ohrenzerlegung

Satz [Whitney ’32]

Ist G ein zweifach zusammenhängender Graph, dann existiert für jede Kante $\{s,t\} \in E$ eine offene Ohrendekomposition, die mit $\{s,t\}$ beginnt.



Iteratives Vorgehen [Brandes ’02]

- $P_0 = (s\{s,t\},t)$
- solange $\exists\ w \to x \in E_i,\ v \hookrightarrow w \notin E_i$ mit $x \stackrel{*}{\to} v$ und $w, x \in V_i$
- u letzter Knoten in V_i auf $x \stackrel{*}{\to} v$

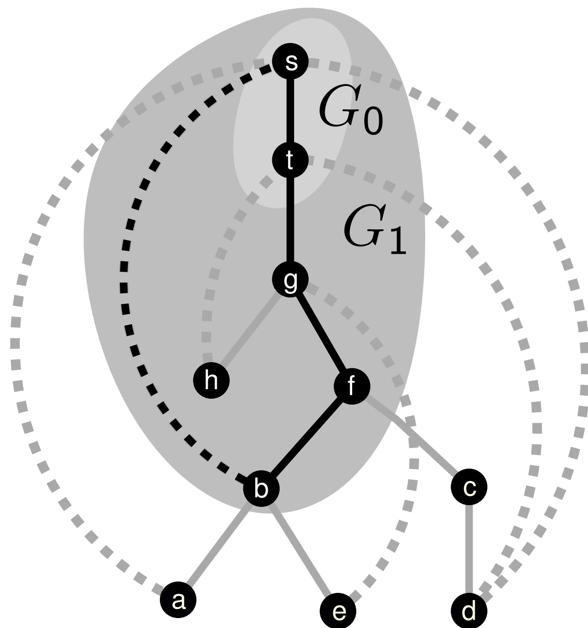
$\Rightarrow$ $P_{i+1} = u \stackrel{*}{\to} v \hookrightarrow w$ (Ohr zu $v \hookrightarrow w$)

Ohrenzerlegung

Karlsruhe Institute of Technology

Satz [Whitney '32]

Ist G ein zweifach zusammenhängender Graph, dann existiert für jede Kante $\{s,t\} \in E$ eine offene Ohrendekomposition, die mit $\{s,t\}$ beginnt.



Iteratives Vorgehen [Brandes '02]

- $P_0 = (s\{s,t\},t)$
- solange $\exists\, w \to x \in E_i$, $v \hookrightarrow w \notin E_i$ mit $x \overset{*}{\to} v$ und $w, x \in V_i$
- u letzter Knoten in V_i auf $x \overset{*}{\to} v$

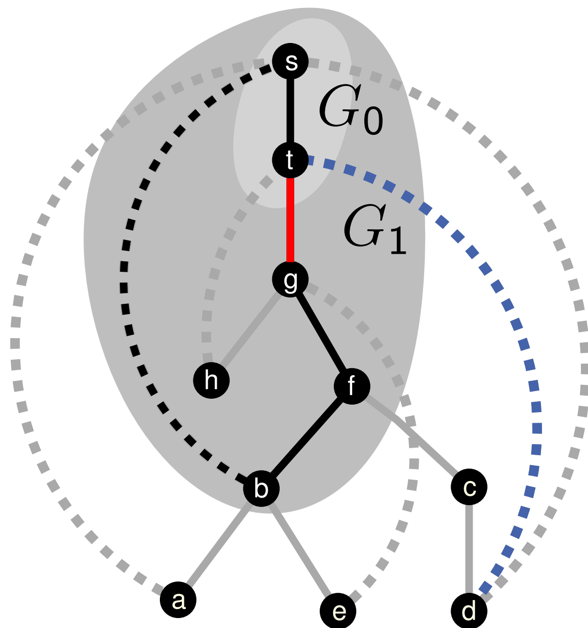
$\Rightarrow$ $P_{i+1} = u \overset{*}{\to} v \hookrightarrow w$ (Ohr zu $v \hookrightarrow w$)

Ohrenzerlegung

Karlsruhe Institute of Technology

Satz [Whitney '32]

Ist G ein zweifach zusammenhängender Graph, dann existiert für jede Kante $\{s,t\} \in E$ eine offene Ohrendekomposition, die mit $\{s,t\}$ beginnt.



Iteratives Vorgehen [Brandes '02]

- $P_0 = (s\{s,t\},t)$
- solange $\exists\, w \to x \in E_i$, $v \hookrightarrow w \notin E_i$ mit $x \xrightarrow{*} v$ und $w, x \in V_i$
- u letzter Knoten in V_i auf $x \xrightarrow{*} v$

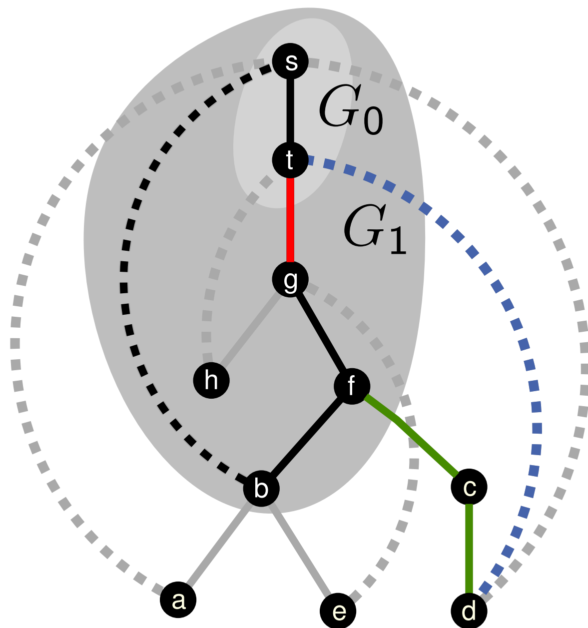
$\Rightarrow$ $P_{i+1} = u \xrightarrow{*} v \hookrightarrow w$ (Ohr zu $v \hookrightarrow w$)

Ohrenzerlegung

Karlsruhe Institute of Technology

Satz [Whitney ’32]

Ist G ein zweifach zusammenhängender Graph, dann existiert für jede Kante $\{s,t\} \in E$ eine offene Ohrendekomposition, die mit $\{s,t\}$ beginnt.



Iteratives Vorgehen [Brandes ’02]

- $P_0 = (s\{s,t\},t)$
- solange $\exists\, w \to x \in E_i$, $v \hookrightarrow w \notin E_i$ mit $x \overset{*}{\to} v$ und $w, x \in V_i$
- u letzter Knoten in V_i auf $x \overset{*}{\to} v$

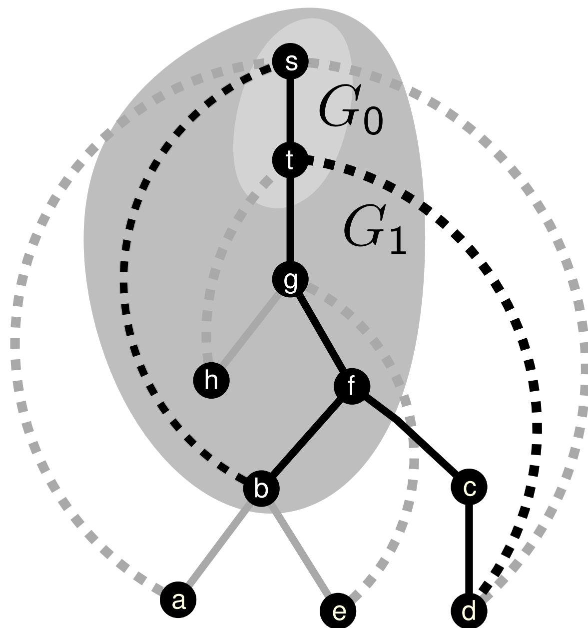
$\Rightarrow$ $P_{i+1} = u \overset{*}{\to} v \hookrightarrow w$ (Ohr zu $v \hookrightarrow w$)

Ohrenzerlegung

Karlsruhe Institute of Technology

Satz [Whitney '32]

Ist G ein zweifach zusammenhängender Graph, dann existiert für jede Kante $\{s,t\} \in E$ eine offene Ohrendekomposition, die mit $\{s,t\}$ beginnt.



Iteratives Vorgehen [Brandes '02]

- $P_0 = (s\{s,t\},t)$
- solange $\exists\, w \rightarrow x \in E_i,\, v \hookrightarrow w \notin E_i$ mit $x \overset{*}{\rightarrow} v$ und $w, x \in V_i$
- u letzter Knoten in V_i auf $x \overset{*}{\rightarrow} v$

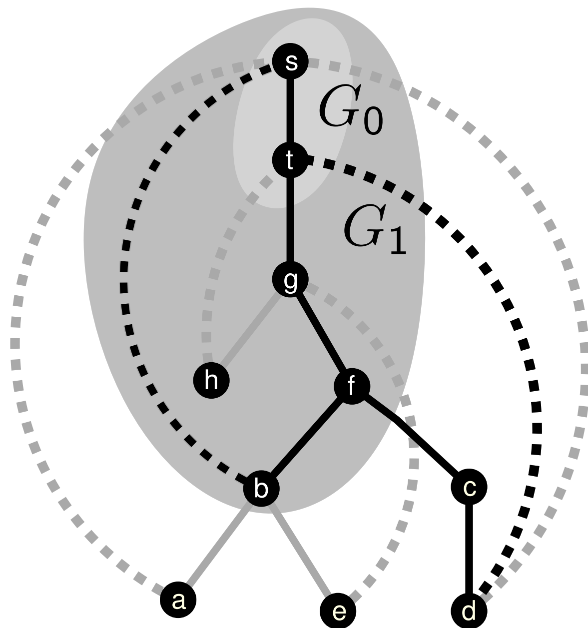
$\Rightarrow$ $P_{i+1} = u \overset{*}{\rightarrow} v \hookrightarrow w$ (Ohr zu $v \hookrightarrow w$)

Ohrenzerlegung

Karlsruhe Institute of Technology

Satz [Whitney '32]

Ist G ein zweifach zusammenhängender Graph, dann existiert für jede Kante $\{s,t\} \in E$ eine offene Ohrendekomposition, die mit $\{s,t\}$ beginnt.



Iteratives Vorgehen [Brandes '02]

- $P_0 = (s\{s,t\},t)$
- solange $\exists\ w \to x \in E_i,\ v \hookrightarrow w \notin E_i$ mit $x \overset{*}{\to} v$ und $w, x \in V_i$
- u letzter Knoten in V_i auf $x \overset{*}{\to} v$

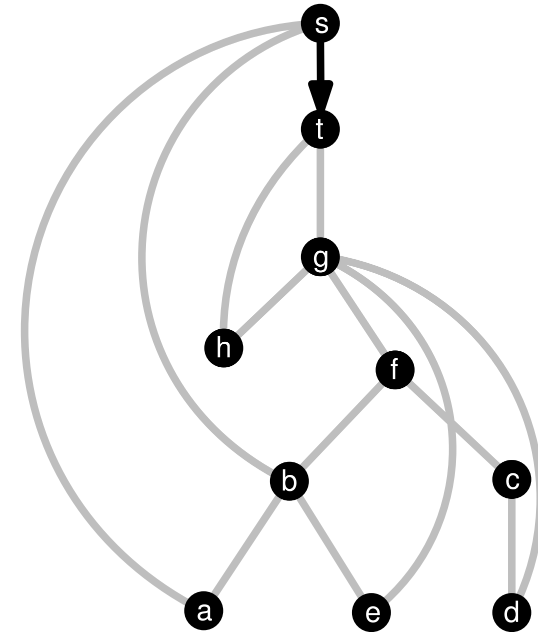
$\Rightarrow$ $P_{i+1} = u \overset{*}{\to} v \hookrightarrow w$ (Ohr zu $v \hookrightarrow w$)

Satz

$D(T)$ ist eine offene Ohrendekomposition, die mit $\{s,t\}$ beginnt.

Orientierung der Kanten

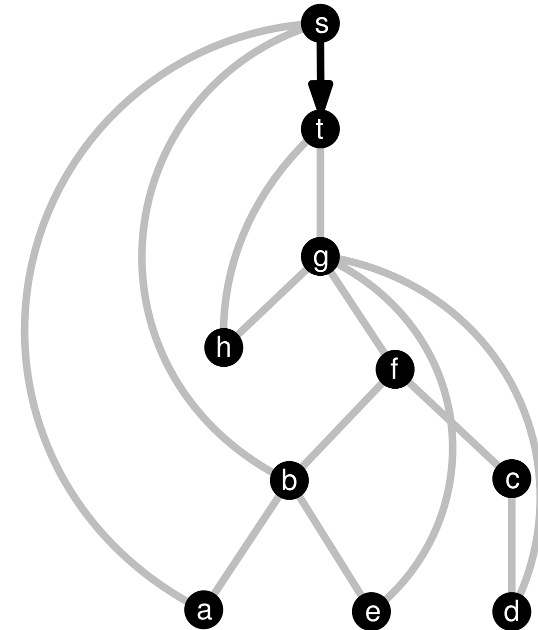
Definition: Rückwärtskante $v \hookrightarrow w$ hängt von Vorwärtskante $w \to x$ ab, für die $w \xrightarrow{*} v$ gilt.



Orientierung der Kanten

Definition: Rückwärtskante $v \hookrightarrow w$ hängt von Vorwärtskante $w \to x$ ab, für die $w \overset{*}{\to} v$ gilt.

Orientierung der Kanten: Betrachte Sequenz von Ohren $P_0, P_1, \ldots, P_k$ in $D(T)$. Orientiere P_0 von s nach t und anschließend iterativ alle weitere Ohren entsprechend der Orientierung ihrer abhängigen Kante.

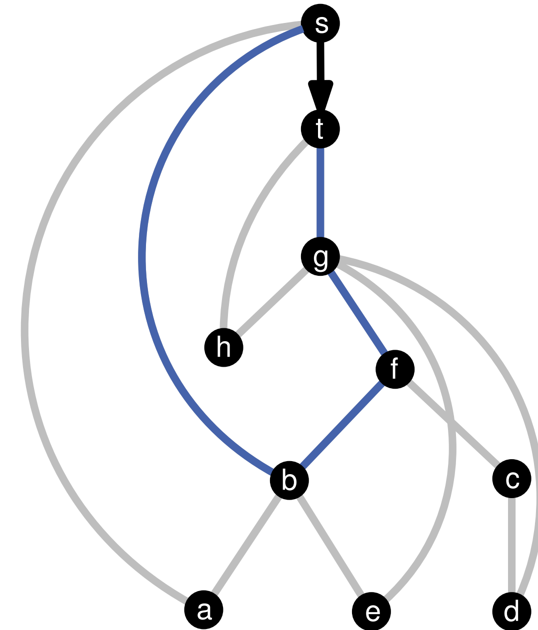


Orientierung der Kanten

Karlsruhe Institute of Technology

Definition: Rückwärtskante $v \hookrightarrow w$ hängt von Vorwärtskante $w \to x$ ab, für die $w \overset{*}{\to} v$ gilt.

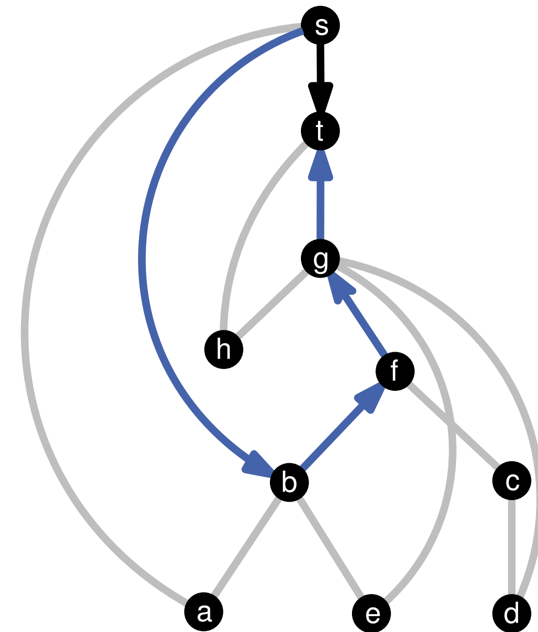
Orientierung der Kanten: Betrachte Sequenz von Ohren $P_0, P_1, \ldots, P_k$ in $D(T)$. Orientiere P_0 von s nach t und anschließend iterativ alle weitere Ohren entsprechend der Orientierung ihrer abhängigen Kante.



Orientierung der Kanten

Definition: Rückwärtskante $v \hookrightarrow w$ hängt von Vorwärtskante $w \to x$ ab, für die $w \overset{*}{\to} v$ gilt.

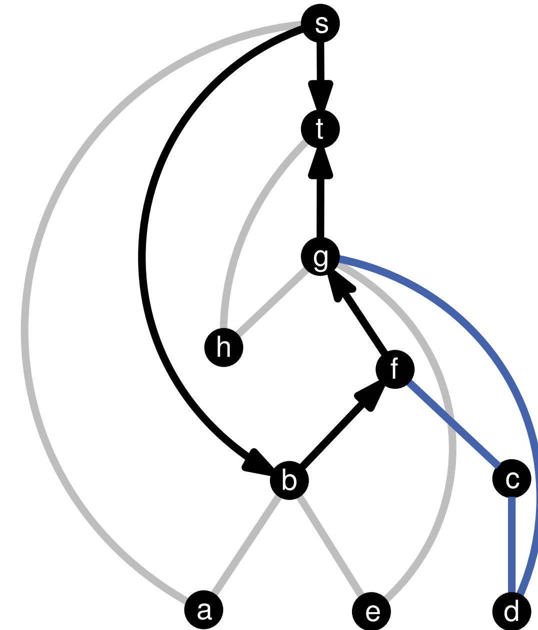
Orientierung der Kanten: Betrachte Sequenz von Ohren $P_0, P_1, \ldots, P_k$ in $D(T)$. Orientiere P_0 von s nach t und anschließend iterativ alle weitere Ohren entsprechend der Orientierung ihrer abhängigen Kante.



Orientierung der Kanten

Definition: Rückwärtskante $v \hookrightarrow w$ hängt von Vorwärtskante $w \to x$ ab, für die $w \overset{*}{\to} v$ gilt.

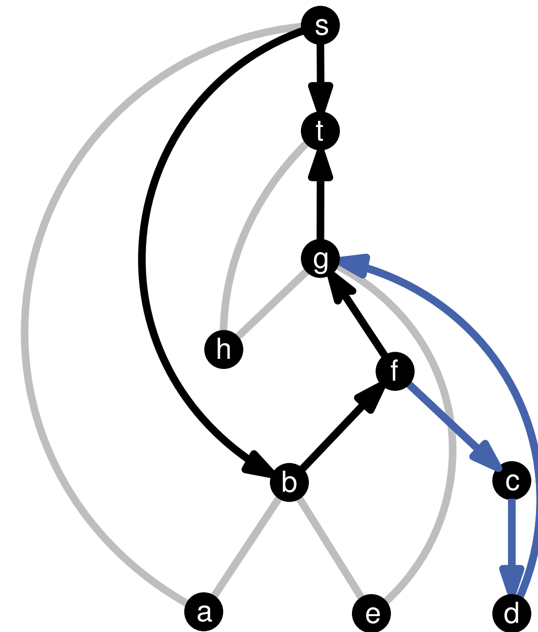
Orientierung der Kanten: Betrachte Sequenz von Ohren $P_0, P_1, \ldots, P_k$ in $D(T)$. Orientiere P_0 von s nach t und anschließend iterativ alle weitere Ohren entsprechend der Orientierung ihrer abhängigen Kante.



Orientierung der Kanten

Definition: Rückwärtskante $v \hookrightarrow w$ hängt von Vorwärtskante $w \to x$ ab, für die $w \stackrel{*}{\to} v$ gilt.

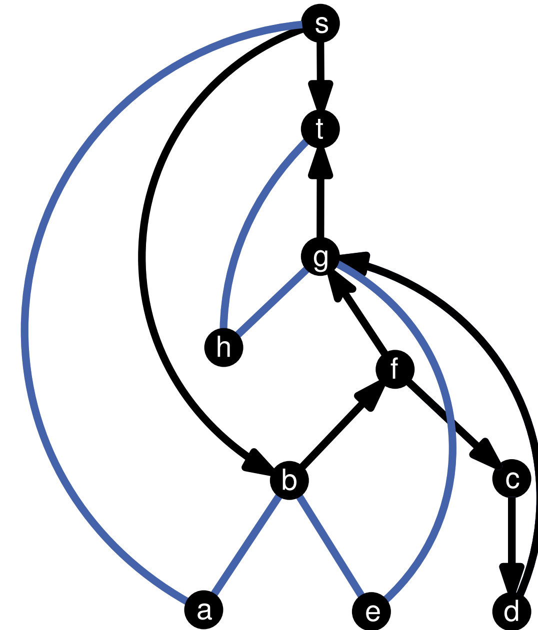
Orientierung der Kanten: Betrachte Sequenz von Ohren $P_0, P_1, \ldots, P_k$ in $D(T)$. Orientiere P_0 von s nach t und anschließend iterativ alle weitere Ohren entsprechend der Orientierung ihrer abhängigen Kante.



Orientierung der Kanten

Definition: Rückwärtskante $v \hookrightarrow w$ hängt von Vorwärtskante $w \to x$ ab, für die $w \overset{*}{\to} v$ gilt.

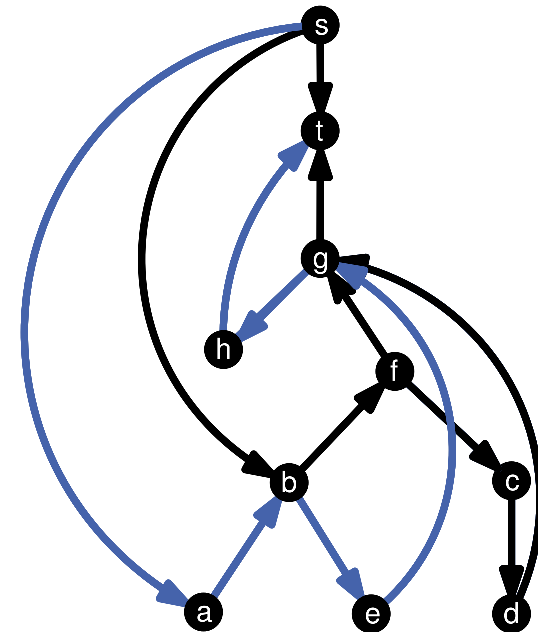
Orientierung der Kanten: Betrachte Sequenz von Ohren $P_0, P_1, \ldots, P_k$ in $D(T)$. Orientiere P_0 von s nach t und anschließend iterativ alle weitere Ohren entsprechend der Orientierung ihrer abhängigen Kante.



Orientierung der Kanten

Definition: Rückwärtskante $v \hookrightarrow w$ hängt von Vorwärtskante $w \to x$ ab, für die $w \overset{*}{\to} v$ gilt.

Orientierung der Kanten: Betrachte Sequenz von Ohren $P_0, P_1, \ldots, P_k$ in $D(T)$. Orientiere P_0 von s nach t und anschließend iterativ alle weitere Ohren entsprechend der Orientierung ihrer abhängigen Kante.



Satz

Die Orientierung von $P_0, \ldots, P_i$ liefert eine s-t-Orientierung von G_i für alle $0 \le i \le r$, und für die partiellen Ordnungen $\prec_i$ gilt:

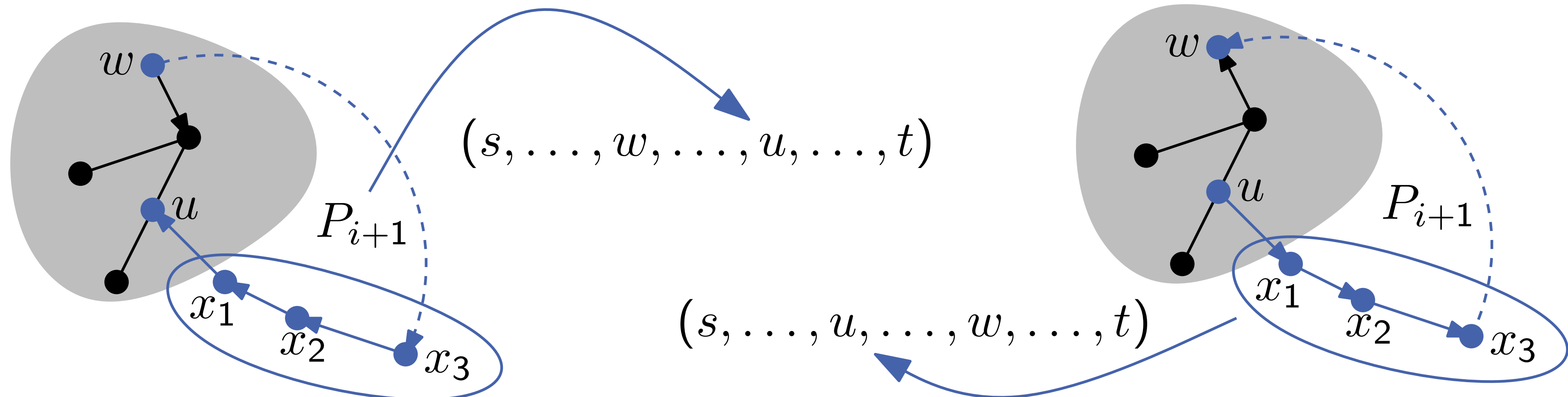
$$w \to x \in E_i \text{ und } w \prec_i x \text{ (bzw. } x \prec_i w) \implies w \prec_i v \text{ (bzw. } v \prec_i w)$$

für alle $v \in T(x) \cap V_i$.

st-Ordnung

Iterative Berechnung der st-Ordnung

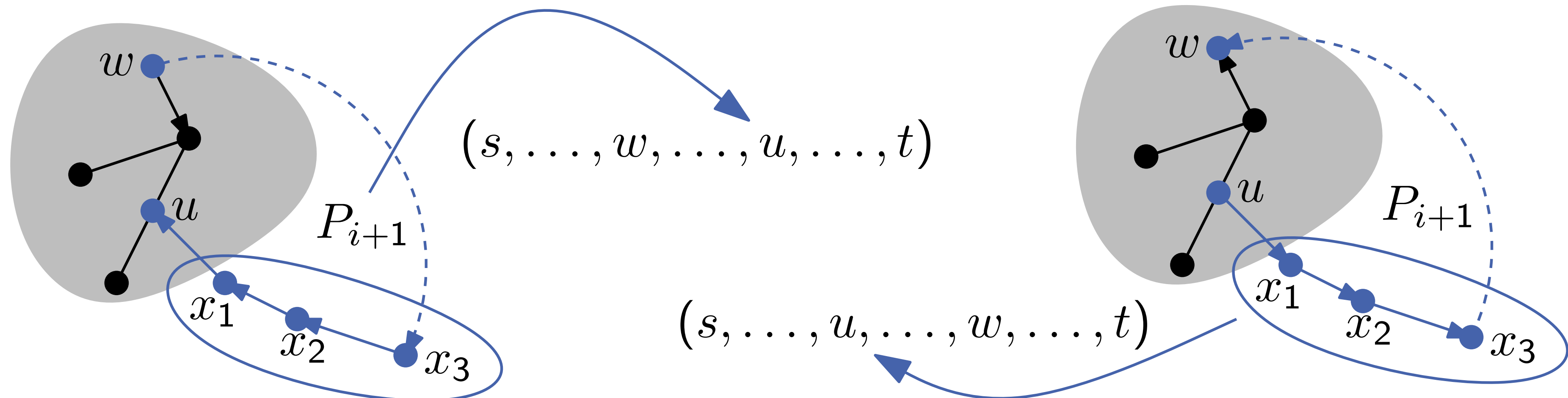
- starte mit Sequenz (s, t)
- betrachte Ohren $P_1, \ldots, P_r$ in dieser Ordnung
- sei $P_i = u \xrightarrow{*} v \hookrightarrow w$ das Ohr zu $v \hookrightarrow w$
- falls P_i von w nach u orientiert wird, füge Sequenz $V(P_i)$ unmittelbar vor u ein, sonst hinter u



st-Ordnung

Iterative Berechnung der st-Ordnung

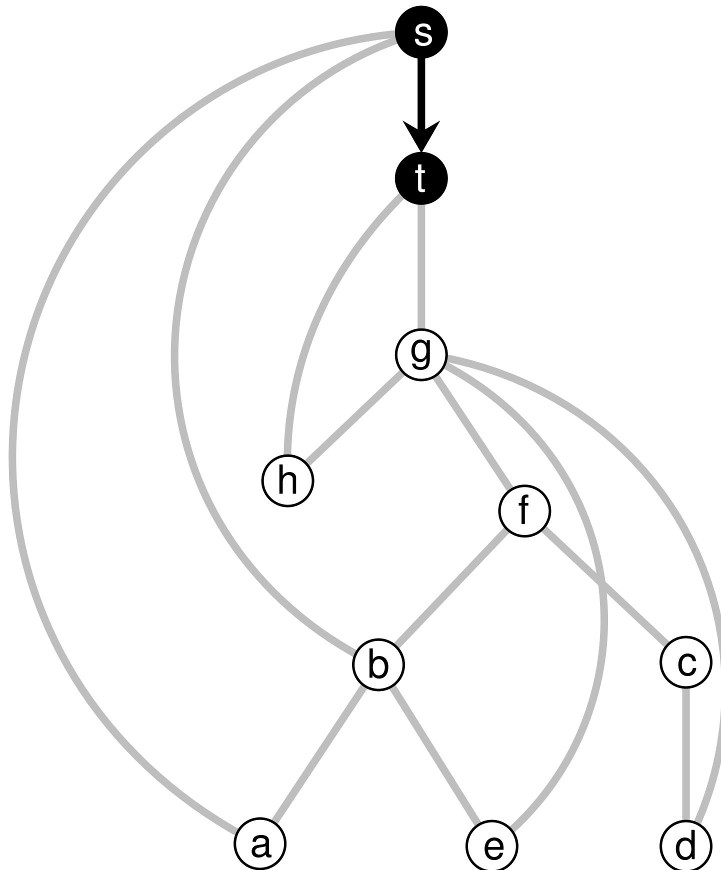
- starte mit Sequenz (s, t)
- betrachte Ohren $P_1, \ldots, P_r$ in dieser Ordnung
- sei $P_i = u \overset{*}{\rightarrow} v \hookrightarrow w$ das Ohr zu $v \hookrightarrow w$
- falls P_i von w nach u orientiert wird, füge Sequenz $V(P_i)$ unmittelbar vor u ein, sonst hinter u



Satz

Die erhaltene Ordnung von V_i ist eine lineare Erweiterung von $\prec_i$ für alle $0 \leq i \leq r$.

Eager st-Ordering



s → t

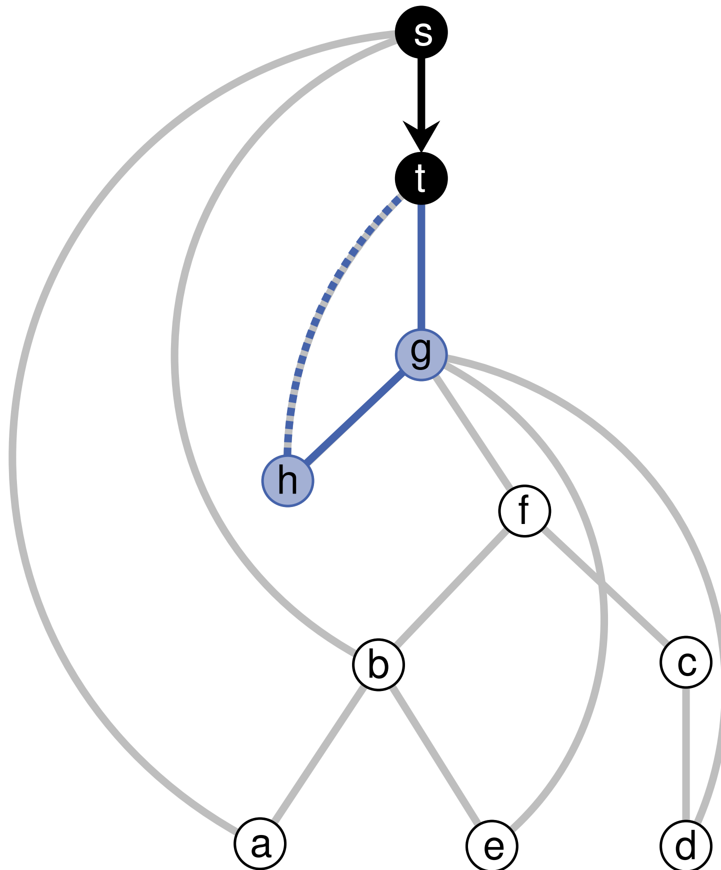
process_ears(tree edge $w \to x$) begin

- **foreach** $v \hookrightarrow w \in D[w \to x]$ **do**
  - $u \leftarrow v$;
  - **while** $u \notin L$ **do** $u \leftarrow PARENT[u]$;
  - $P \leftarrow (u \stackrel{*}{\to} v \hookrightarrow w)$;
  - **if** $w \to x$ *von* w *nach* x *(bzw.* x *nach* w*) orientiert* **then**
    - orientiere P von w nach u (bzw. u nach w);
    - füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;
  - **foreach** *Baumkante* $w' \to x'$ *von* P **do** $process_ears(w' \to x')$;
- $D[\{w, x\}] \leftarrow \emptyset$;

dfs(vertex v) begin

- $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;
- **while** *es ex. nicht nummerierte Kante* $e = \{v, w\}$ **do**
  - $DFS[e] \leftarrow DFS[v]$;
  - **if** w *nicht nummeriert* **then**
    - $CHILDEDGE[v] \leftarrow e$;
    - $PARENT[w] \leftarrow v$;
    - $dfs(w)$;
  - **else**
    - $\{w, x\} \leftarrow CHILDEDGE[w]$;
    - $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
    - **if** $x \in L$ **then** $process_ears(w \to x)$;

Eager st-Ordering



s → t

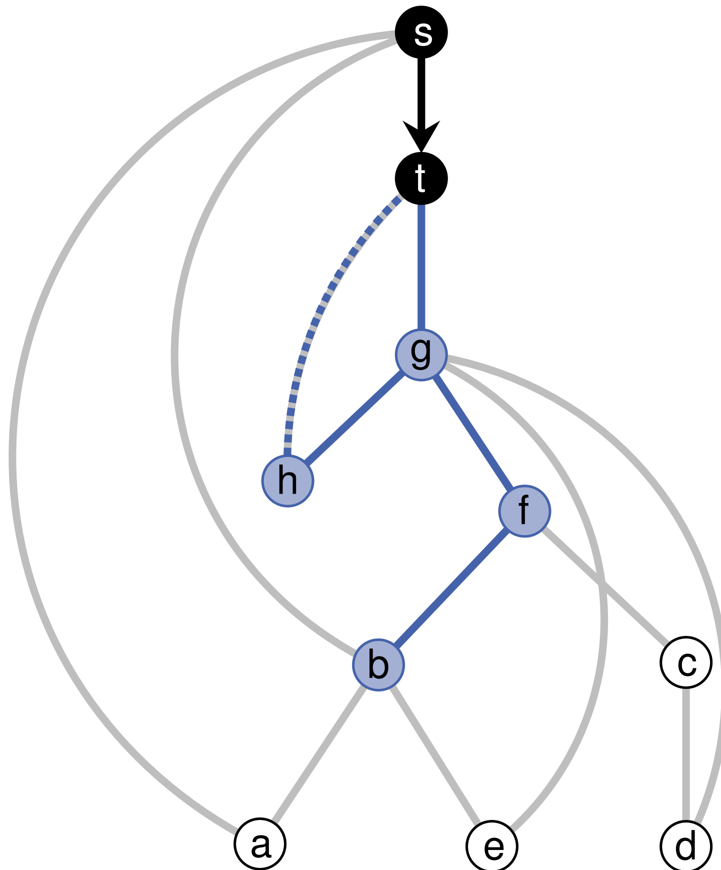
process_ears(tree edge $w \to x$) begin

- **foreach** $v \hookrightarrow w \in D[w \to x]$ **do**
  - $u \leftarrow v$;
  - **while** $u \notin L$ **do** $u \leftarrow PARENT[u]$;
  - $P \leftarrow (u \xrightarrow{*} v \hookrightarrow w)$;
  - **if** $w \to x$ *von* w *nach* x *(bzw.* x *nach* w*) orientiert* **then**
    - orientiere P von w nach u (bzw. u nach w);
    - füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;
  - **foreach** *Baumkante* $w' \to x'$ *von* P **do** $process_ears(w' \to x')$;
- $D[\{w, x\}] \leftarrow \emptyset$;

dfs(vertex v) begin

- $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;
- **while** *es ex. nicht nummerierte Kante* $e = \{v, w\}$ **do**
  - $DFS[e] \leftarrow DFS[v]$;
  - **if** w *nicht nummeriert* **then**
    - $CHILDEDGE[v] \leftarrow e$;
    - $PARENT[w] \leftarrow v$;
    - $dfs(w)$;
  - **else**
    - $\{w, x\} \leftarrow CHILDEDGE[w]$;
    - $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
    - **if** $x \in L$ **then** $process_ears(w \to x)$;

Eager st-Ordering



$s \rightarrow t$

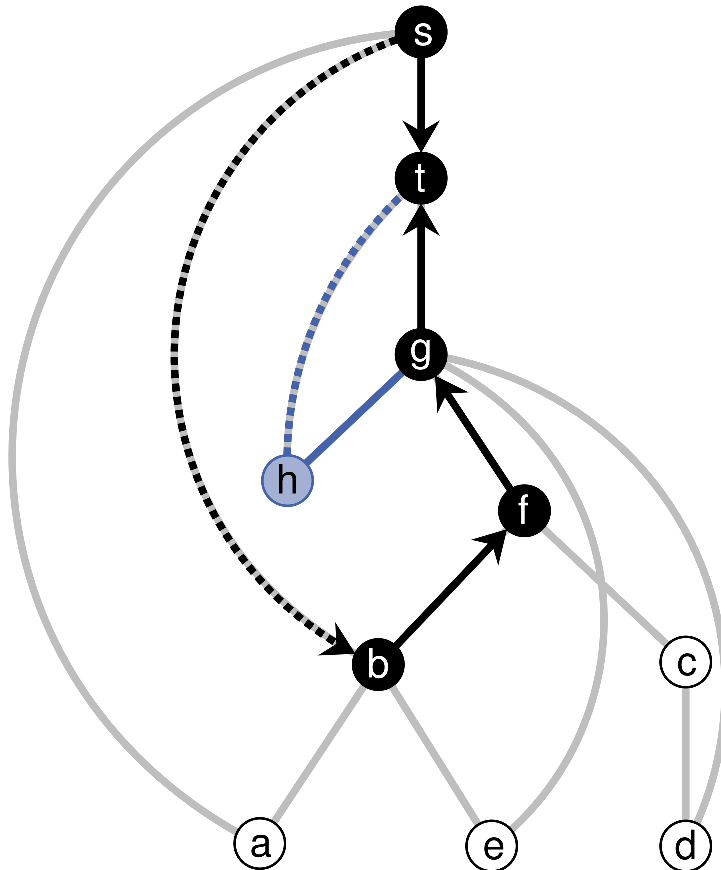
process_ears(tree edge $w \rightarrow x$) begin

- **foreach** $v \hookrightarrow w \in D[w \rightarrow x]$ **do**
  - $u \leftarrow v$;
  - **while** $u \notin L$ **do** $u \leftarrow PARENT[u]$;
  - $P \leftarrow (u \overset{*}{\rightarrow} v \hookrightarrow w)$;
  - **if** $w \rightarrow x$ *von* w *nach* x *(bzw.* x *nach* w*) orientiert* **then**
    - orientiere P von w nach u (bzw. u nach w);
    - füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;
  - **foreach** *Baumkante* $w' \rightarrow x'$ *von* P **do** $process_ears(w' \rightarrow x')$;
- $D[\{w, x\}] \leftarrow \emptyset$;

dfs(vertex v) begin

- $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;
- **while** *es ex. nicht nummerierte Kante* $e = \{v, w\}$ **do**
  - $DFS[e] \leftarrow DFS[v]$;
  - **if** w *nicht nummeriert* **then**
    - $CHILDEDGE[v] \leftarrow e$;
    - $PARENT[w] \leftarrow v$;
    - $dfs(w)$;
  - **else**
    - $\{w, x\} \leftarrow CHILDEDGE[w]$;
    - $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
    - **if** $x \in L$ **then** $process_ears(w \rightarrow x)$;

Eager st-Ordering

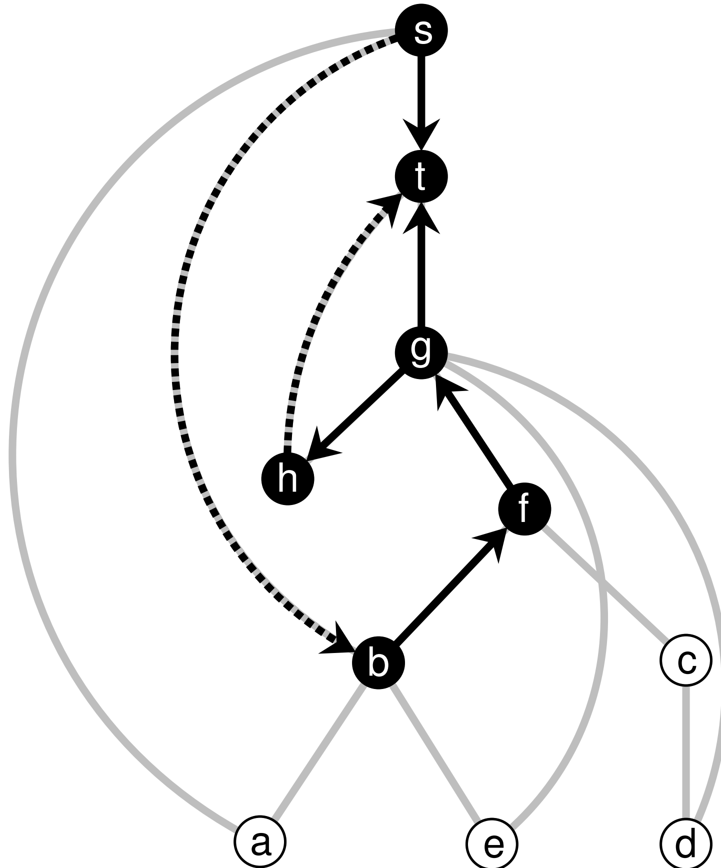


$s \rightarrow b \rightarrow f \rightarrow g \rightarrow t$

process_ears(tree edge $w \rightarrow x$) begin
- **foreach** $v \hookrightarrow w \in D[w \rightarrow x]$ **do**
  - $u \leftarrow v$;
  - **while** $u \notin L$ **do** $u \leftarrow PARENT[u]$;
  - $P \leftarrow (u \overset{*}{\rightarrow} v \hookrightarrow w)$;
  - **if** $w \rightarrow x$ *von* w *nach* x *(bzw.* x *nach* w*) orientiert* **then**
    - orientiere P von w nach u (bzw. u nach w);
    - füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;
  - **foreach** *Baumkante* $w' \rightarrow x'$ *von* P **do** $process_ears(w' \rightarrow x')$;
- $D[\{w, x\}] \leftarrow \emptyset$;

dfs(vertex v) begin
- $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;
- **while** *es ex. nicht nummerierte Kante* $e = \{v, w\}$ **do**
  - $DFS[e] \leftarrow DFS[v]$;
  - **if** w *nicht nummeriert* **then**
    - $CHILDEDGE[v] \leftarrow e$;
    - $PARENT[w] \leftarrow v$;
    - $dfs(w)$;
  - **else**
    - $\{w, x\} \leftarrow CHILDEDGE[w]$;
    - $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
    - **if** $x \in L$ **then** $process_ears(w \rightarrow x)$;

Eager st-Ordering



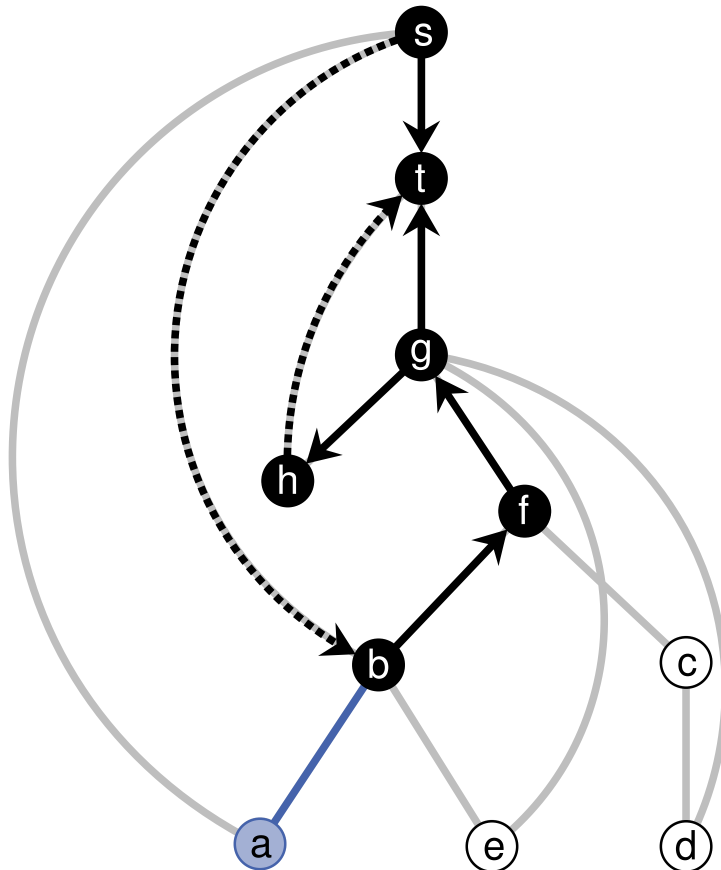
$s \rightarrow b \rightarrow f \rightarrow g \rightarrow t$

process_ears(tree edge $w \rightarrow x$) begin
- **foreach** $v \hookrightarrow w \in D[w \rightarrow x]$ **do**
  - $u \leftarrow v$;
  - **while** $u \notin L$ **do** $u \leftarrow PARENT[u]$;
  - $P \leftarrow (u \overset{*}{\rightarrow} v \hookrightarrow w)$;
  - **if** $w \rightarrow x$ *von* w *nach* x *(bzw.* x *nach* w*) orientiert* **then**
    - orientiere P von w nach u (bzw. u nach w);
    - füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;
  - **foreach** *Baumkante* $w' \rightarrow x'$ *von* P **do** $process_ears(w' \rightarrow x')$;
- $D[\{w, x\}] \leftarrow \emptyset$;

dfs(vertex v) begin
- $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;
- **while** *es ex. nicht nummerierte Kante* $e = \{v, w\}$ **do**
  - $DFS[e] \leftarrow DFS[v]$;
  - **if** w *nicht nummeriert* **then**
    - $CHILDEDGE[v] \leftarrow e$;
    - $PARENT[w] \leftarrow v$;
    - $dfs(w)$;
  - **else**
    - $\{w, x\} \leftarrow CHILDEDGE[w]$;
    - $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
    - **if** $x \in L$ **then** $process_ears(w \rightarrow x)$;

Eager st-Ordering

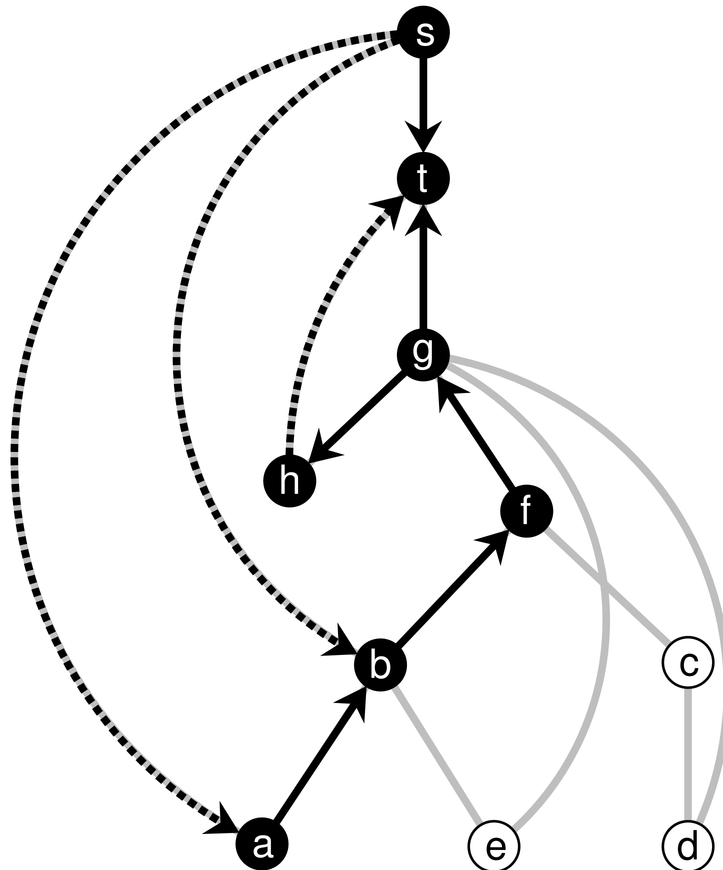
Karlsruhe Institute of Technology



process_ears(tree edge $w \to x$) begin
- **foreach** $v \hookrightarrow w \in D[w \to x]$ **do**
  - $u \leftarrow v$;
  - **while** $u \notin L$ **do** $u \leftarrow PARENT[u]$;
  - $P \leftarrow (u \overset{*}{\to} v \hookrightarrow w)$;
  - **if** $w \to x$ *von* w *nach* x *(bzw.* x *nach* w*) orientiert* **then**
    - orientiere P von w nach u (bzw. u nach w);
    - füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;
  - **foreach** *Baumkante* $w' \to x'$ *von* P **do** $process_ears(w' \to x')$;
- $D[\{w, x\}] \leftarrow \emptyset$;

dfs(vertex v) begin
- $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;
- **while** *es ex. nicht nummerierte Kante* $e = \{v, w\}$ **do**
  - $DFS[e] \leftarrow DFS[v]$;
  - **if** w *nicht nummeriert* **then**
    - $CHILDEDGE[v] \leftarrow e$;
    - $PARENT[w] \leftarrow v$;
    - $dfs(w)$;
  - **else**
    - $\{w, x\} \leftarrow CHILDEDGE[w]$;
    - $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
    - **if** $x \in L$ **then** $process_ears(w \to x)$;

Eager st-Ordering

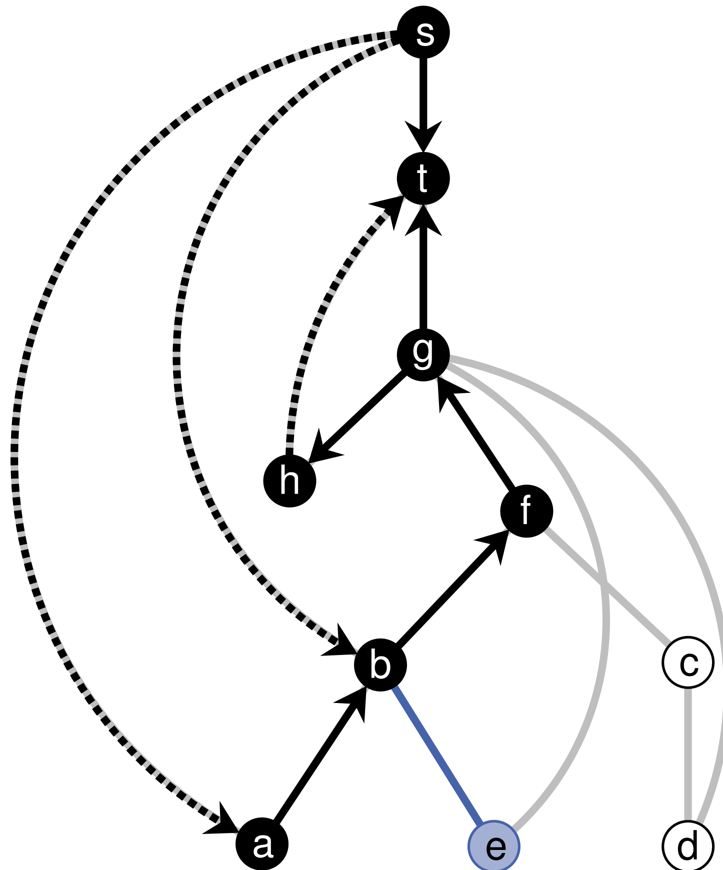


process_ears(tree edge $w \rightarrow x$) begin
- **foreach** $v \hookrightarrow w \in D[w \rightarrow x]$ **do**
  - $u \leftarrow v$;
  - **while** $u \notin L$ **do** $u \leftarrow PARENT[u]$;
  - $P \leftarrow (u \overset{*}{\rightarrow} v \hookrightarrow w)$;
  - **if** *$w \rightarrow x$ von w nach x (bzw. x nach w) orientiert* **then**
    - orientiere P von w nach u (bzw. u nach w);
    - füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;
  - **foreach** *Baumkante $w' \rightarrow x'$ von P* **do** $process_ears(w' \rightarrow x')$;
- $D[\{w, x\}] \leftarrow \emptyset$;

dfs(vertex v) begin
- $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;
- **while** *es ex. nicht nummerierte Kante $e = \{v, w\}$* **do**
  - $DFS[e] \leftarrow DFS[v]$;
  - **if** *w nicht nummeriert* **then**
    - $CHILDEDGE[v] \leftarrow e$;
    - $PARENT[w] \leftarrow v$;
    - $dfs(w)$;
  - **else**
    - $\{w, x\} \leftarrow CHILDEDGE[w]$;
    - $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
    - **if** $x \in L$ **then** $process_ears(w \rightarrow x)$;

s → a → b → f → g → h → t

Eager st-Ordering



process_ears(tree edge $w \to x$**) begin**

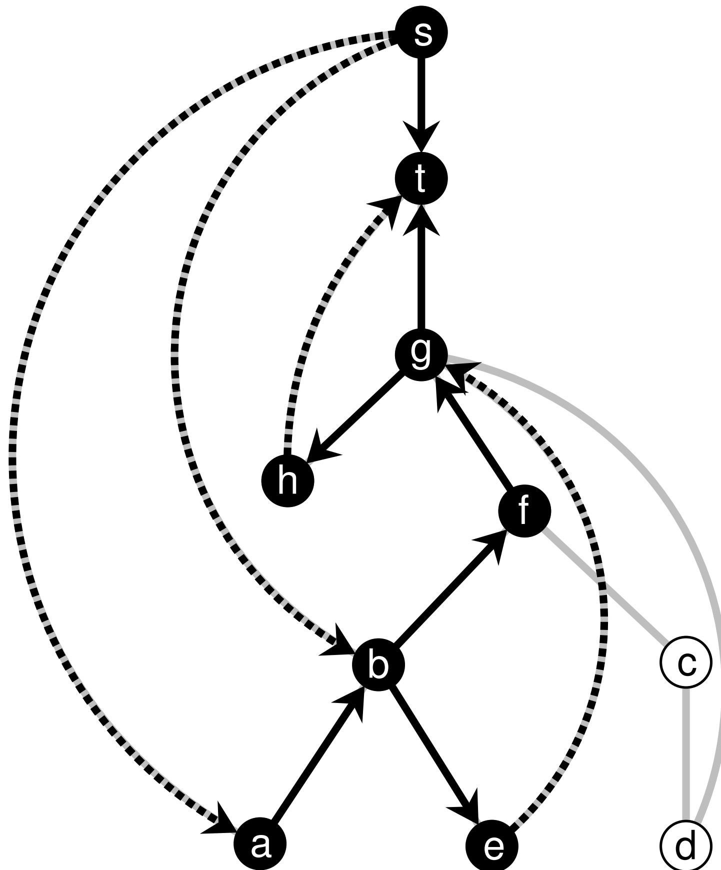
- **foreach** $v \hookrightarrow w \in D[w \to x]$ **do**
  - $u \leftarrow v$;
  - **while** $u \notin L$ **do** $u \leftarrow PARENT[u]$;
  - $P \leftarrow (u \xrightarrow{*} v \hookrightarrow w)$;
  - **if** $w \to x$ *von* w *nach* x *(bzw.* x *nach* w*) orientiert* **then**
    - orientiere P von w nach u (bzw. u nach w);
    - füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;
  - **foreach** *Baumkante* $w' \to x'$ *von* P **do** $process_ears(w' \to x')$;
- $D[\{w, x\}] \leftarrow \emptyset$;

dfs(vertex v**) begin**

- $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;
- **while** *es ex. nicht nummerierte Kante* $e = \{v, w\}$ **do**
  - $DFS[e] \leftarrow DFS[v]$;
  - **if** w *nicht nummeriert* **then**
    - $CHILDEDGE[v] \leftarrow e$;
    - $PARENT[w] \leftarrow v$;
    - $dfs(w)$;
  - **else**
    - $\{w, x\} \leftarrow CHILDEDGE[w]$;
    - $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
    - **if** $x \in L$ **then** $process_ears(w \to x)$;

s → a → b → f → g → h → t

Eager st-Ordering



process_ears(tree edge $w \to x$) **begin**

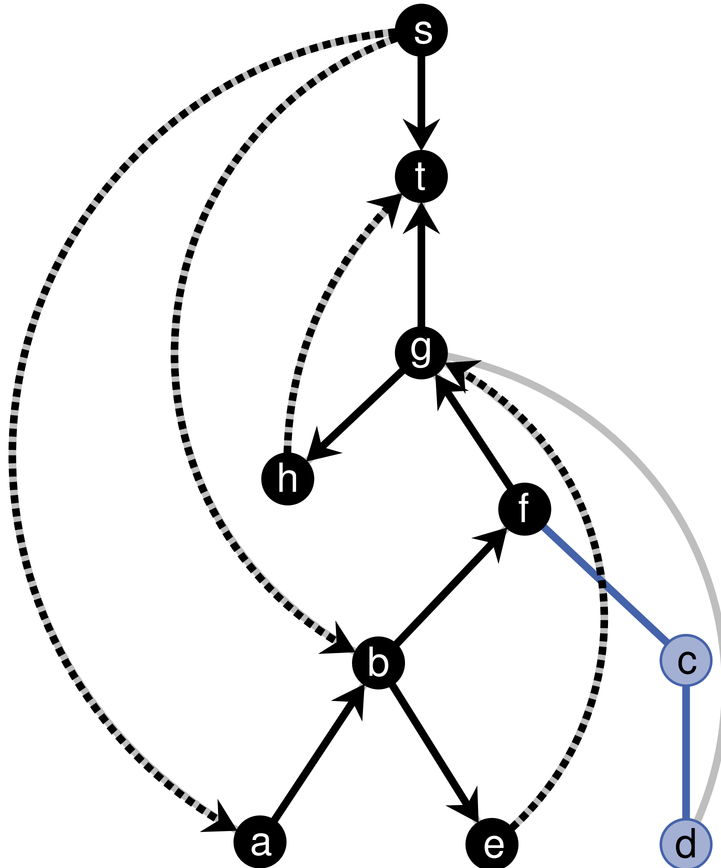
- **foreach** $v \hookrightarrow w \in D[w \to x]$ **do**
  - $u \leftarrow v$;
  - **while** $u \notin L$ **do** $u \leftarrow PARENT[u]$;
  - $P \leftarrow (u \overset{*}{\to} v \hookrightarrow w)$;
  - **if** $w \to x$ *von* w *nach* x *(bzw.* x *nach* w*) orientiert* **then**
    - orientiere P von w nach u (bzw. u nach w);
    - füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;
  - **foreach** *Baumkante* $w' \to x'$ *von* P **do** $process_ears(w' \to x')$;
- $D[\{w, x\}] \leftarrow \emptyset$;

dfs(vertex v) **begin**

- $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;
- **while** *es ex. nicht nummerierte Kante* $e = \{v, w\}$ **do**
  - $DFS[e] \leftarrow DFS[v]$;
  - **if** w *nicht nummeriert* **then**
    - $CHILDEDGE[v] \leftarrow e$;
    - $PARENT[w] \leftarrow v$;
    - $dfs(w)$;
  - **else**
    - $\{w, x\} \leftarrow CHILDEDGE[w]$;
    - $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
    - **if** $x \in L$ **then** $process_ears(w \to x)$;

$s \to a \to b \to e \to f \to g \to h \to t$

Eager st-Ordering



process_ears(tree edge $w \rightarrow x$**) begin**

  foreach $v \hookrightarrow w \in D[w \rightarrow x]$ **do**

    $u \leftarrow v$;

    while $u \notin L$ **do** $u \leftarrow PARENT[u]$;

    $P \leftarrow (u \stackrel{*}{\rightarrow} v \hookrightarrow w)$;

    if $w \rightarrow x$ *von* w *nach* x *(bzw.* x *nach* w*) orientiert* **then**

      orientiere P von w nach u (bzw. u nach w);
      füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;

    foreach *Baumkante* $w' \rightarrow x'$ *von* P **do** $process_ears(w' \rightarrow x')$;

  $D[\{w, x\}] \leftarrow \emptyset$;

dfs(vertex v**) begin**

  $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;

  while *es ex. nicht nummerierte Kante* $e = \{v, w\}$ **do**

    $DFS[e] \leftarrow DFS[v]$;

    if w *nicht nummeriert* **then**

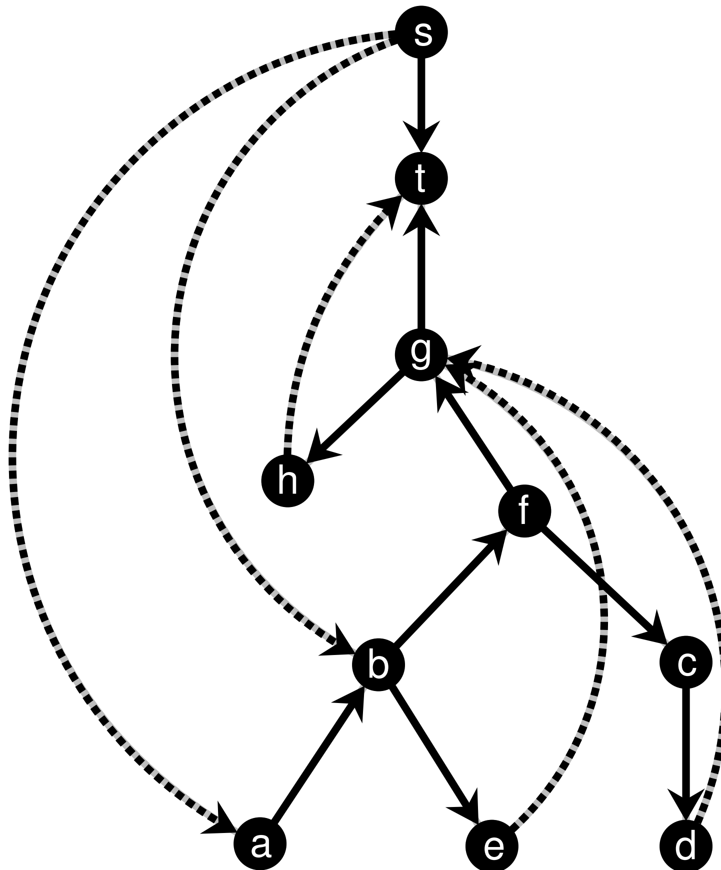
      $CHILDEDGE[v] \leftarrow e$;
      $PARENT[w] \leftarrow v$;
      $dfs(w)$;

    else

      $\{w, x\} \leftarrow CHILDEDGE[w]$;
      $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
      if $x \in L$ **then** $process_ears(w \rightarrow x)$;

s → a → b → e → f → g → h → t

Eager st-Ordering



process_ears(**tree edge** $w \to x$) **begin**

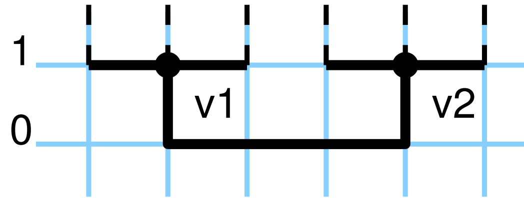
- **foreach** $v \hookrightarrow w \in D[w \to x]$ **do**
  - $u \leftarrow v$;
  - **while** $u \notin L$ **do** $u \leftarrow PARENT[u]$;
  - $P \leftarrow (u \overset{*}{\to} v \hookrightarrow w)$;
  - **if** $w \to x$ *von* w *nach* x *(bzw.* x *nach* w*) orientiert* **then**
    - orientiere P von w nach u (bzw. u nach w);
    - füge innere Knoten von P unmittelbar vor (bzw. hinter) u in L ein;
  - **foreach** *Baumkante* $w' \to x'$ *von* P **do** $process_ears(w' \to x')$;
- $D[\{w, x\}] \leftarrow \emptyset$;

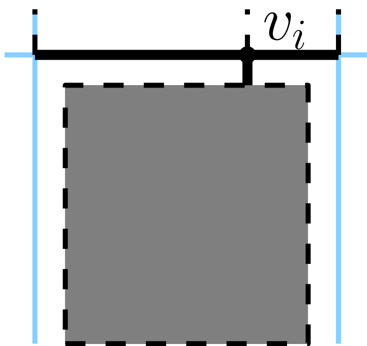
dfs(vertex v**) begin**

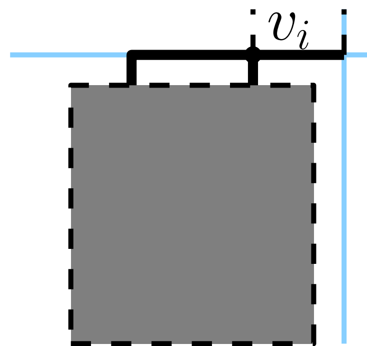
- $i \leftarrow i + 1$; $DFS[v] \leftarrow i$;
- **while** *es ex. nicht nummerierte Kante* $e = \{v, w\}$ **do**
  - $DFS[e] \leftarrow DFS[v]$;
  - **if** w *nicht nummeriert* **then**
    - $CHILDEDGE[v] \leftarrow e$;
    - $PARENT[w] \leftarrow v$;
    - $dfs(w)$;
  - **else**
    - $\{w, x\} \leftarrow CHILDEDGE[w]$;
    - $D[\{w, x\}] \leftarrow D[\{w, x\}] \cup \{e\}$;
    - **if** $x \in L$ **then** $process_ears(w \to x)$;

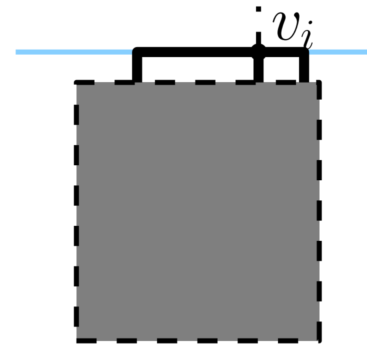
s → a → b → e → f → c → d → g → h → t

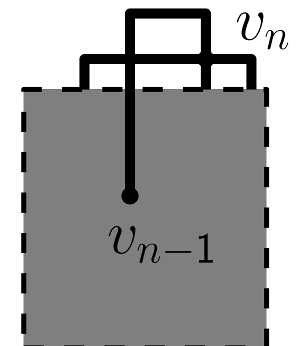
Algorithmus für Blöcke





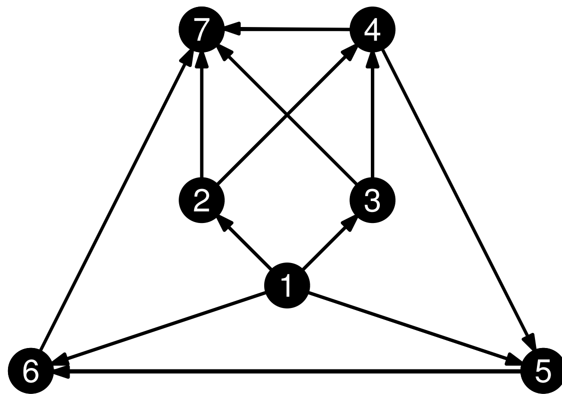




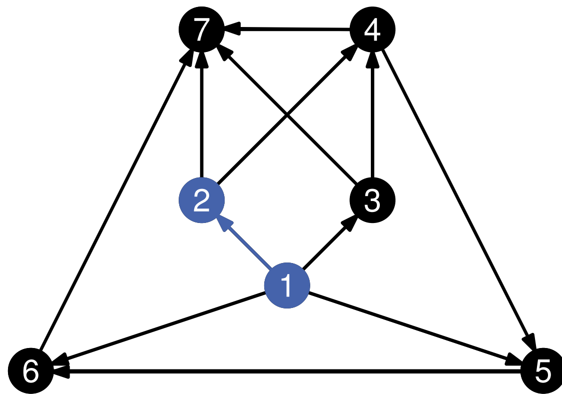


Beispiel

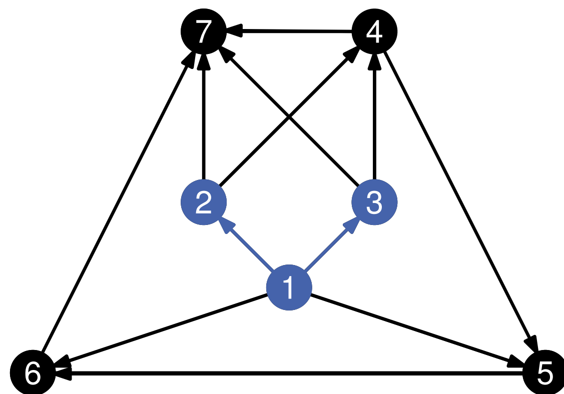
Karlsruhe Institute of Technology

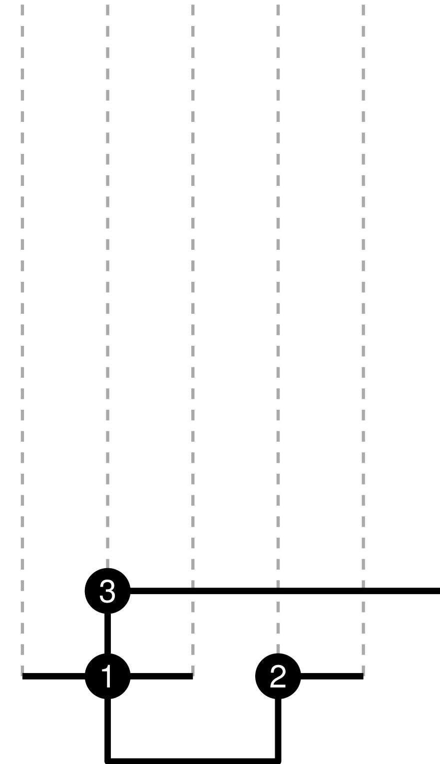


Beispiel

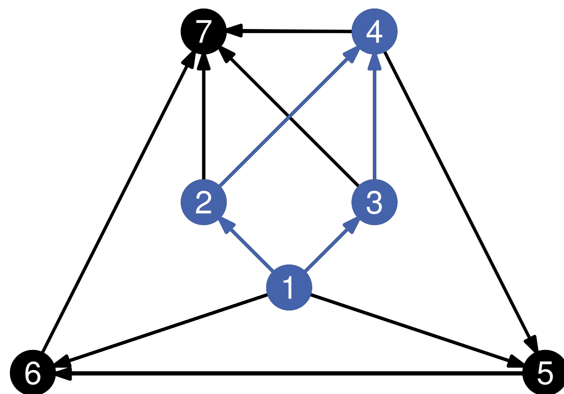


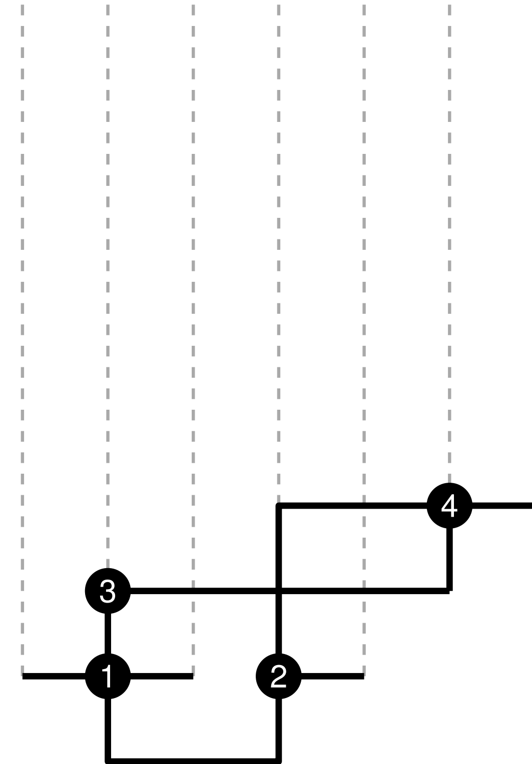
Beispiel



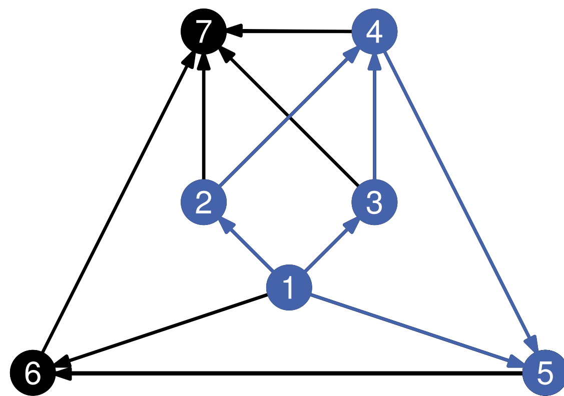


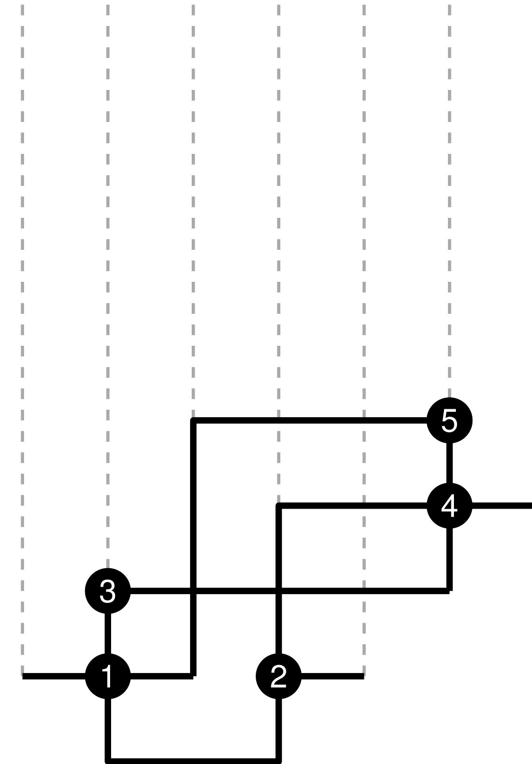
Beispiel



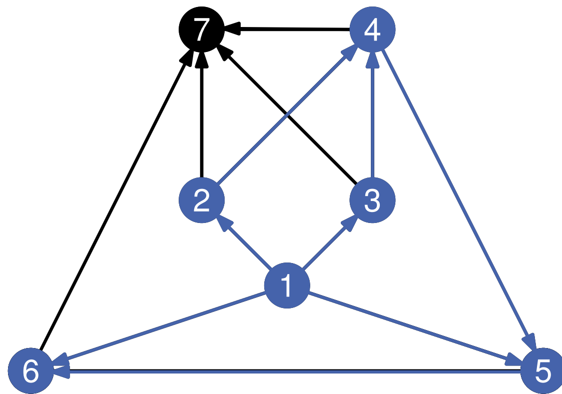


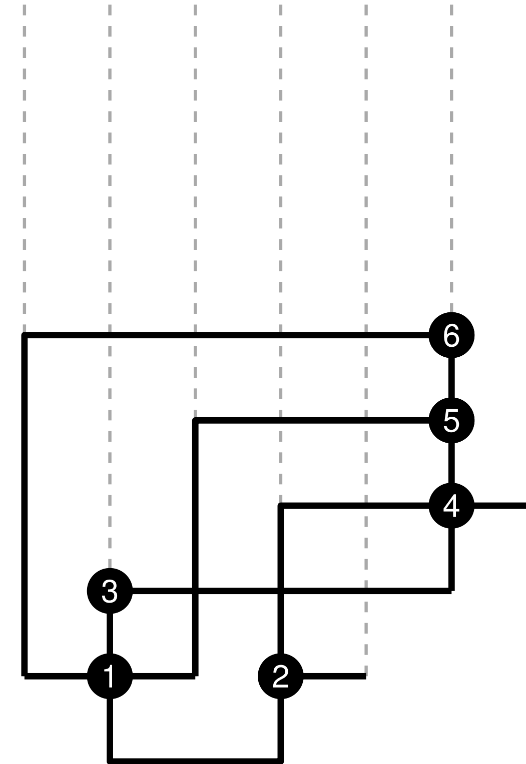
Beispiel



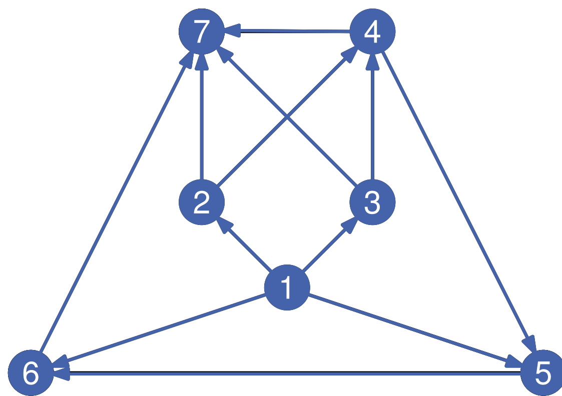


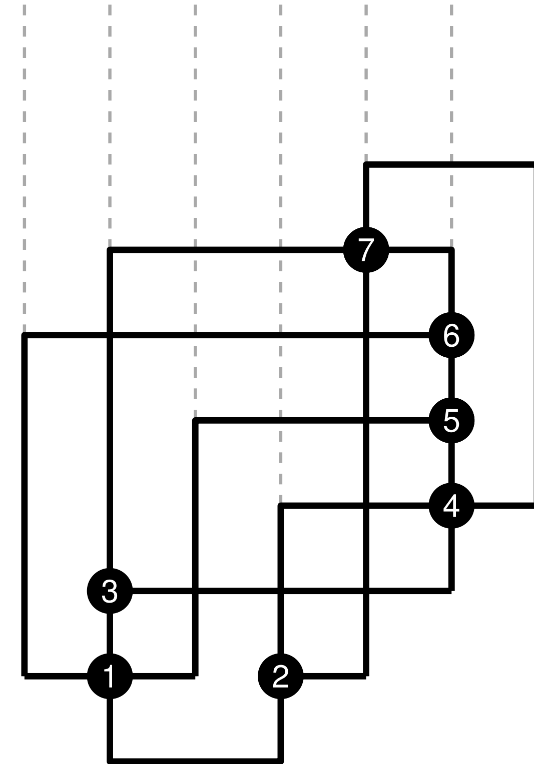
Beispiel





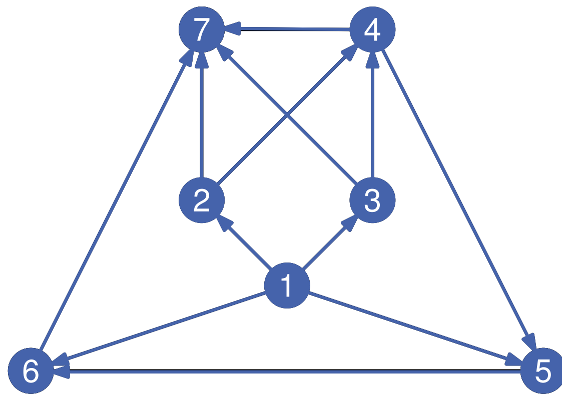
Beispiel

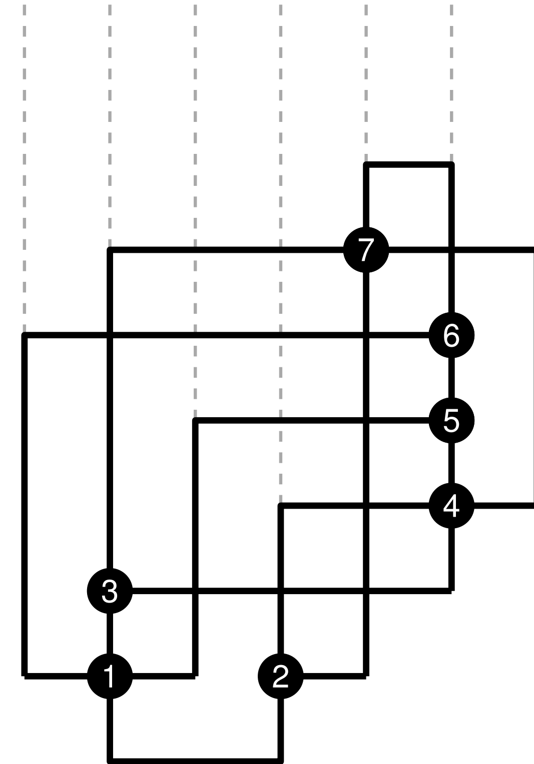




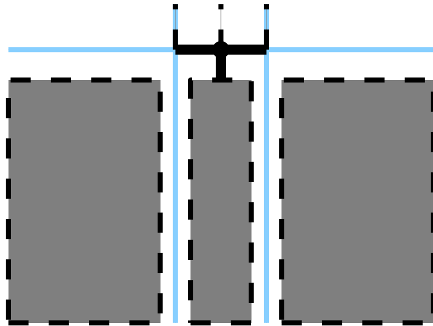
Institut für Theoretische Informatik
Lehrstuhl Algorithmik I

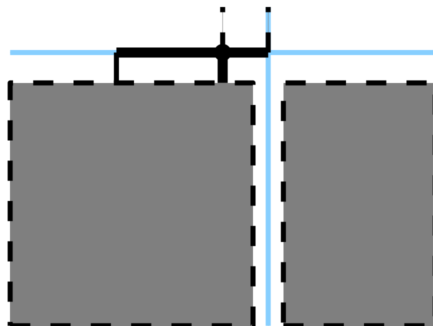
Beispiel

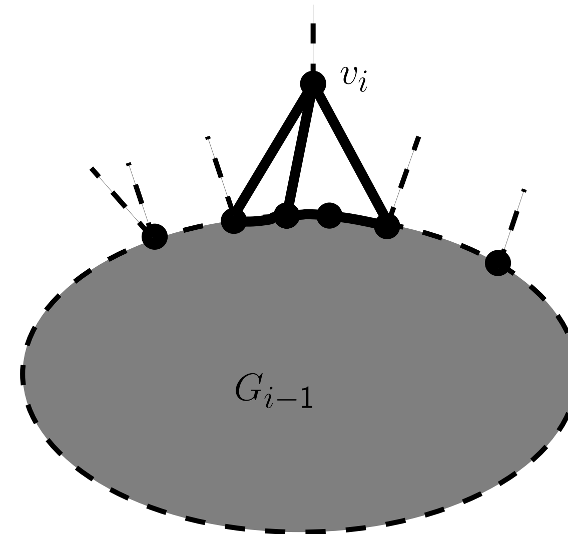




Planarität







Analyse

Satz [Biedl & Kant '94]

Die benötigte Gittergröße ist höchstens $(m - n + 1) \times n$.

Analyse

KIT Karlsruhe Institute of Technology

Satz [Biedl & Kant '94]

Die benötigte Gittergröße ist höchstens $(m - n + 1) \times n$.

Satz [Biedl & Kant '94]

Die Gesamtzahl der Knicke ist höchstens $2m - 2n + 4$, und keine Kante hat mehr als zwei Knicke, es sei denn G ist ein Oktaeder.

Algorithmus für allgemeine Graphen

Karlsruhe Institute of Technology

Satz [Biedl & Kant '94]

Sei G ein einfach zusammenhängender Graph mit Maximalgrad 4. Dann kann G auf einem Gitter der Größe $n \times n$ mit höchstens 2 Knicken pro Kante eingebettet werden.