

# Zusammenfassung für Zeichnen von Graphen

Sommersemester 2014

von Dagmar Sorg

# DIVIDE AND CONQUER

## 1 Binärbäume

Die Tiefe eines Knotens ( $depth(v)$ ) ist der Abstand von der Wurzel bis  $v$ .

Der *Divide and Conquer*-Ansatz wird verwendet, um ein geradliniges Gitterlayout für geradlinige Repräsentationen zu finden. Dies funktioniert für Binärbäume wie folgt:

1. Bestimmung des Teillayouts für  $T_l(v)$
2. Bestimmung des Teillayouts für  $T_r(v)$
3. Zusammenfügen zu Gesamtlayouts

### 1.1 Baumdurchläufe

Es gibt drei Arten von Baumdurchläufen:

1. Preorder
2. Inorder
3. Postorder

**Ein Gitterlayout** hat ausschließlich ganzzahlige Koordinaten

**Geradlinige Repräsentationen** sind Standardrepräsentationen mit geraden Kanten

**Ein Layout ist vollatändig bestimmt**, wenn für jeden Knoten  $v \in V$  eine  $x$ -Koordinate ( $x(v)$ ) und eine  $y$ -Koordinate ( $y(v)$ ) gegeben sind.

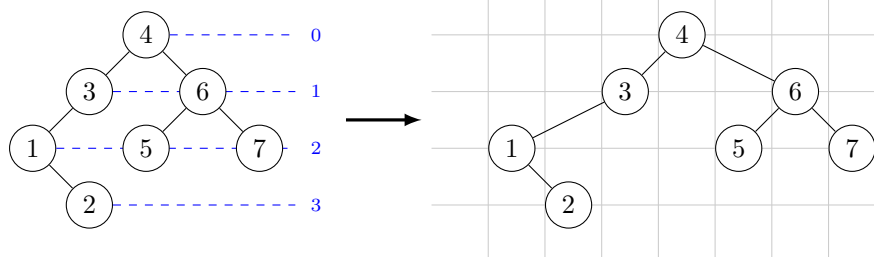
### 1.2 Definition eines Gitterlayouts

Für jeden Knoten ist der Punkt im Layout folgendermaßen definiert:

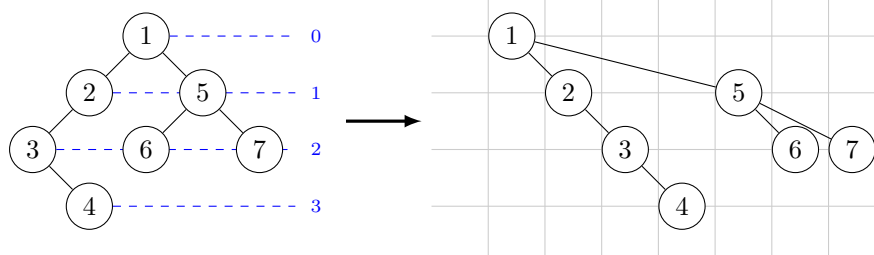
$$\begin{aligned}x(v_i) &= i \\y(v_i) &= -depth(v_i)\end{aligned}$$

Dieses Layout ist *abwärts*, *kreuzungsfrei* und kann in Linearzeit bestimmt werden.

**Beispiel (Inorder – Layout):**



**Beispiel (Preorder – Layout):**



### Nachteile:

1. Breite =  $n - 1$
2. Kantenlänge  $\mathcal{O}(n)$
3. Knoten sind nicht zentriert über Nachfolgern

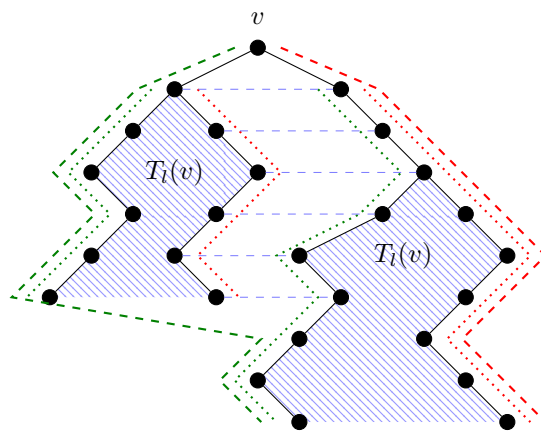
Problem 1 und 2 können durch Berechnung relativer Koordinaten behoben werden:

- Berechnung der Koordinaten der Teilbäume getrennt
  - Zusammenlegen der Layouts, sodass die *umgebenden Rechtecke* Abstand 2 oder 3 haben
  - Elterknoten wird zentriert über den Nachfolgern platziert (bzw. eins weiter nach rechts/links, wenn es wichtig ist, um welchen Nachfolger es sich handelt)
- ⇒ immer noch in Linearzeit bestimmbar, aber immer noch **zu breit**
- ⇒ verwenden von **Konturen** statt Rechtecken:  
Platzieren der Teilbäume so, dass der minimale horizontale Abstand zweier Knoten der gleichen Tiefe 2 (bzw. 3) ist

#### 1.2.1 Bestimmung eines Layouts mithilfe von Konturen

- Abstand zwischen zwei Knoten mit gleicher Tiefe ist zwei (drei, falls Teilbaumwurzeln sonst ungeraden Abstand haben)
- zur Bestimmung in Linearzeit (Bestimmung pro Knoten):
  1. Berechnung des  $x$ -Offsets (relative Position zum Vorgänger)
  2. Speichern der linken und rechten Kontur seines Teilbaumes als eine einfach verkettete Liste
- Algorithmus von *Reingold und Tilford* arbeitet in zwei Schritten:

1. *postorder*: zur Bestimmung von Konturen und  $x$ -Offsets



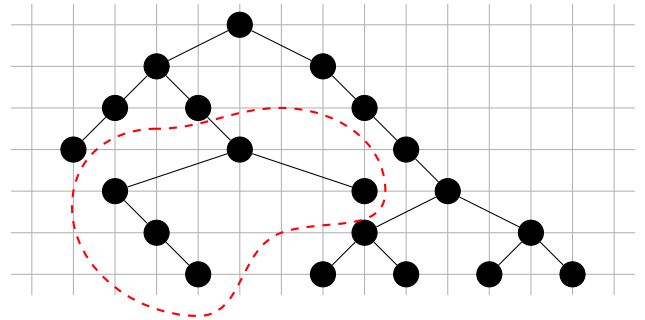
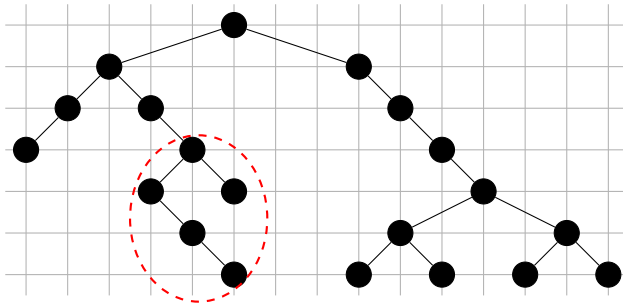
- Abarbeiten von  $T_l(v)$  und  $T_r(v)$
- Absteigen in den Konturen beider Teilbäume parallel, bis die Konturen des niedrigeren Teilbaums endet
- linke (rechte) Kontur von  $T(v)$  besteht aus  $v$ , der linken (rechten) Kontur von  $T_l(v)$  ( $T_r(v)$ ) und dem (falls vorhanden) linken (rechten) Kontur von  $T_r(v)$  ( $T_l(v)$ )
- Bestimmen des Mindestabstandes  $d \geq 2$  der Nachfolger von  $v$  aus den  $x$ -Offsets der rechten Kontur von  $T_l(v)$  und der linken Kontur von  $T_r(v)$
- Erhöhen von  $d_v$  um 1, falls ungerade
- Setzen des  $x$ -Offsets der Nachfolger von  $v$  (wenn vorhanden) auf  $-\frac{d_v}{2}$  bzw.  $+\frac{d_v}{2}$

2. *preorder*: Kalkulation der  $x$ -Koordinaten mithilfe des Zusammennehmens der  $x$ -Offsets

- Bearbeitung von  $T(v)$ :  $x$  ist die Koordinate des Vorgängers (oder  $x = 0$ , falls  $v$  die Wurzel ist)
- $x(v) = x + x - Offset(v)$
- Algorithmus von Reingold/Tilford (in Linearzeit) berechnet:
  - \* geradliniges Gitterlayout, das tiefengeschichtet und kreuzungsfrei ist
  - \* Knoten mit selber Tiefe haben Abstand  $\geq 2$
  - \* Knoten sind über Nachfolgern zentriert
  - \* linke/rechte Nachfolger liegen strikt links/rechts von ihrem Vorgänger
  - \* identische Teilbäume sind gleich ausgelegt

⇒ Binärbaumlayout\*

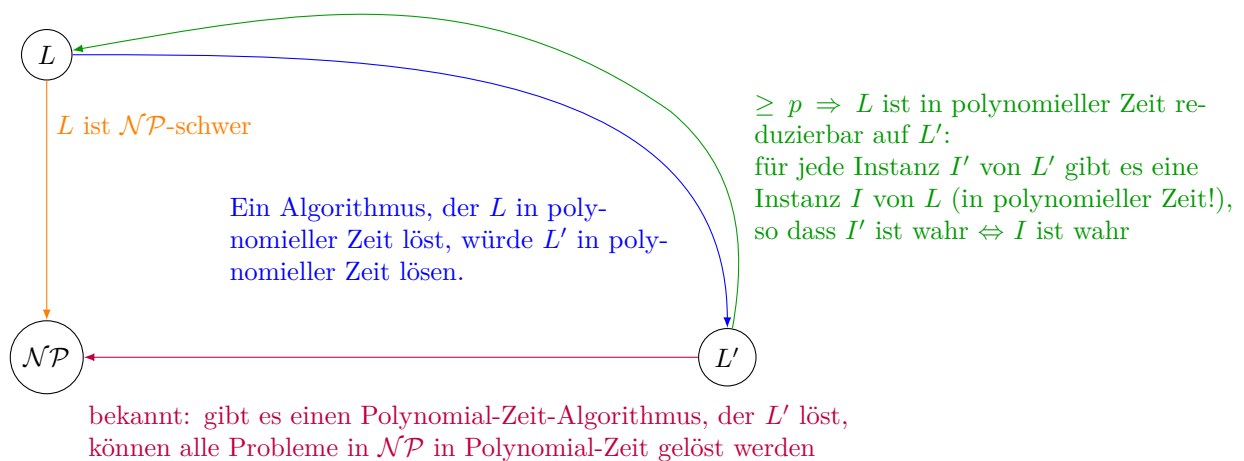
\* Das Layout ist nicht unbedingt Platzoptimal:



### 1.2.2 Die Breitenminimierung von Binärbäumen ist $\mathcal{NP}$ -schwer.

**Beweisidee:**

*Bemerkung:* Das Problem ist in  $\mathcal{P}$ , falls keine ganzzahligen Koordinaten (Gitterlayout) erforderlich sind.  
*Allgemeine Idee:*

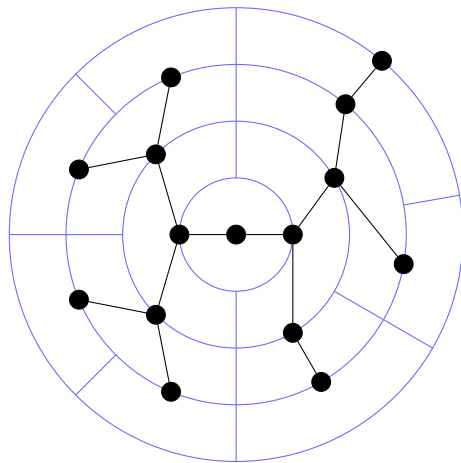


Hier wird gezeigt, dass sogar für feste Werte ( $\mathcal{W} = 24$ ) das Problem in  $\mathcal{NP}$  liegt. Das Problem ist sicher in  $\mathcal{NP}$ , da sich jedes gegebene Layout auf die Eigenschaften eines Binärbaumes und die Breite prüfen lässt.

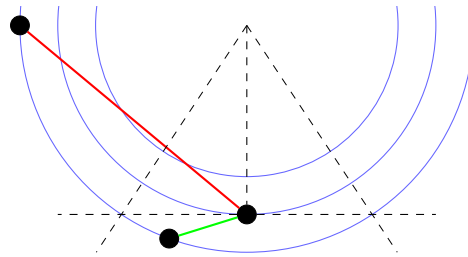
*Ablauf des Beweises:*

- Reduzierung von 3-SAT auf Binärbaum mit minimaler Breite mithilfe von  $F = C_1 \wedge \dots \wedge C_m$  als 3-SAT-Formel mit
  - Klauseln  $C_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$
  - Literalen  $y_{i,j} \in \{x_1, \dots, x_n, \overline{x_1}, \dots, \overline{x_n}\}$
- Konstruktion der Instanz  $T(F)$  für das Layoutproblem, die genau dann ein Layout mit Breite  $\mathcal{W} \leq 24$  hat, falls  $F$  erfüllbar ist
- Erzeugung von Teilbäumen für Variablen, Literale und Klauseln
- Teilbäume der drei in der Klausel  $C_i$  auftretenden Literale werden zu  $T(C_i)$  zusammengefügt (mithilfe von einer ausreichend langen Kette von Knoten, die beim mittleren Literal eingefügt wird und an der Wurzel des nächsten Baumes endet)
- Beweis, dass falls  $F$  gilt auch  $T(F)$  gilt und falls  $F$  nicht gilt auch  $T(F)$  nicht gilt

### 1.3 Radiales Layout



- der Radius entspricht der Tiefe des Knotens
- rekursive Zuweisung der Position jeden Knotens, mit dem ihm noch verbleibenden Kreisteil
- für ein kreuzungsfreies Layout wird der für die Rekursion verfügbare Platz durch die Tangente der Teilbaumwurzel beschränkt
- Vergleich Algorithmus 1. (berechnet in linearer Zeit ein kreuzungsfreies Layout)



## 2 Serien-parallele (SP) Graphen

Ein *SP*-Graph ist:

- gerichtet
- besteht entweder aus zwei Knoten  $s, t$  und der Kante  $\{s, t\}$  oder
- aus zwei *SP*-Graphen  $G_1, G_2$  mit  $s_1, s_2, t_1, t_2$ , entstanden aus

**serielle Komposition:**  $t_1, s_2$  werden verschmolzen,  $s_1 \rightarrow s, t_2 \rightarrow t$

**parallele Komposition:** jeweils  $s_1, s_2$  und  $t_1, t_2$  werden zu  $s, t$  verschmolzen

Fakten:

- Jeder *SP*-Graph ist *azyklisch* und *planar*.
- Jedes kreuzungsfreie Aufwärtslayout für geordnete *SP*-Graphen benötigt im schlechtesten Fall eine Gitter, das exponentiell groß in der Anzahl der Knoten des Graphen ist.

**Beweis:**

$$\pi = (x, s_n, p, t_n)$$

$$\Delta_1 = (a, x, t_n), \Delta_2 = (t_n, p, b)$$

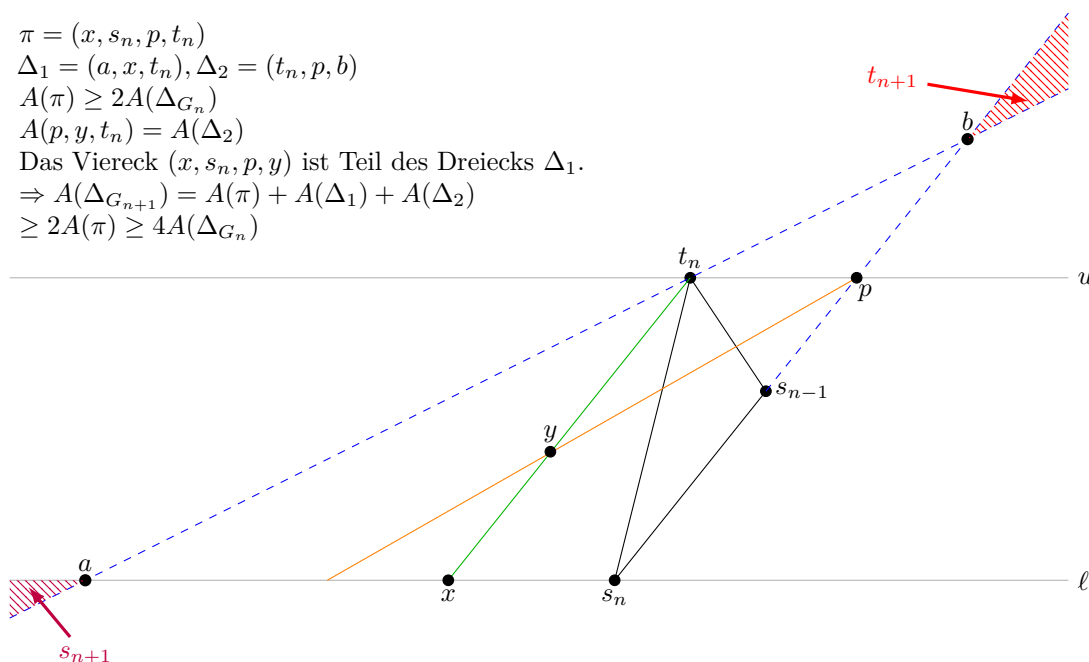
$$A(\pi) \geq 2A(\Delta_{G_n})$$

$$A(p, y, t_n) = A(\Delta_2)$$

Das Viereck  $(x, s_n, p, y)$  ist Teil des Dreiecks  $\Delta_1$ .

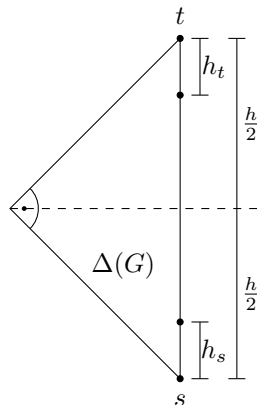
$$\Rightarrow A(\Delta_{G_{n+1}}) = A(\pi) + A(\Delta_1) + A(\Delta_2)$$

$$\geq 2A(\pi) \geq 4A(\Delta_{G_n})$$



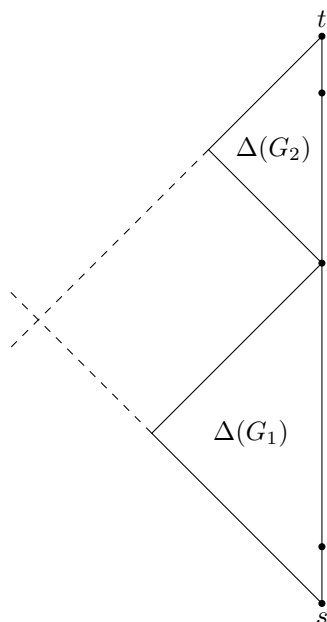
## 2.1 Divide-and-Conquer Ansatz zur Erstellung von *SP*-Graphen auf einem Gitter der Größe $\mathcal{O}(n^2)$

- im Dekompositionsbaum stehen Q-Knoten nur rechts von einem P-Knoten (*linkslastig*)
- Algorithmus:



1. Layout von  $G$  liegt in einem rechtwinkligen, gleichschenkligen Dreieck  $\Delta_G$ , mit vertikaler Basis und linksliegenden Scheitel
2. auf der unteren / oberen Ecke von  $\Delta(G)$  liegt die Quelle / Senke von  $G$  aber kein Knoten liegt auf den linken Ecken von  $\Delta(G)$
3. falls  $v$  Nachbar der Quelle / Senke von  $G$  ist, dann liegt kein anderer Knoten rechts der Senkrechten durch  $v$  und unterhalb der fallenden / oberhalb der steigenden Diagonalen durch  $v$

Serielle Komposition:



Parallele Komposition:

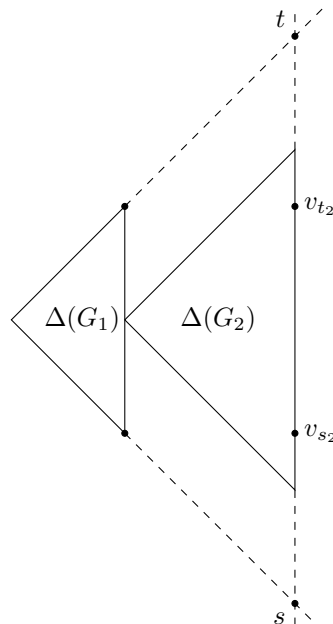
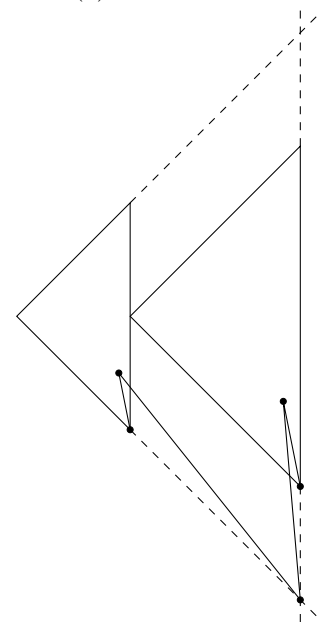


Bild (1):



Aus den Höhenunterschieden und Höhen zu den Komponenten können mittels eines *preorder* Durchlaufes durch den Dekompositionsbaum die absoluten Koordinaten ermittelt werden (ähnlich wie bei *x*-Offsets der Binärbäume).

Der Algorithmus erstellt in Linearzeit aus einem linkslastig geordneten Dekompositionsbaum eines einfachen *SP*-Graphes ein Gitterlayout, das

- kreuzungsfrei ist (Kreuzungen können nur bei paralleler Komposition entstehen): aus (3.) folgt das Bild (1)
- höchstens quadratische Fläche benötigt: per Induktion beweisbar (alle Teile des Graphen  $G$  liegen in seinem Dreieck  $\rightarrow$  bleibt zu zeigen, dass die Höhe von  $\Delta(G)$  linear in der Anzahl der Knoten ist).
- Schönere Darstellung mit *Sichtbarkeitsrepräsentation*
- Erweiterung auf die Darstellung mit *orthogonalen Buskanten*

# INKREMENTELLE KONSTRUKTION

Bei dieser Art der Konstruktion entsteht der Graph nach und nach, d.h. ein Anfangslayout wird Stück für Stück erweitert und nicht zum Schluss als ein Ganzes berechnet.

Jeder der Layoutalgorithmen besteht zusätzlich aus einem Vorverarbeitungsschritt, zum Festlegen einer Reihenfolge mit bestimmten Eigenschaften, die für den eigentlichen Algorithmus wichtig sind.

## 1 Orthogonale Gitterlayouts

- jeder Knoten hat höchstens Grad 4
- Algorithmus von Biedl und Kant (vier Schritte):
  1. Zerlegung des Graphen in Komponenten
  2. Bestimmung der Reihenfolge für die Knoten jeder Komponente
  3. inkrementelle Bestimmung des Layouts für jede Komponente: die Knoten werden in ihrer Reihenfolge in das Layout eingefügt
  4. Kombinierung der Layouts der einzelnen Komponenten
- der Graph wird zuerst in hinreichend zusammenhängende Teilgraphen zerlegt

### 1.1 Zweifache Zusammenhangskomponenten

- Zwei Kanten in einem ungerichteten Graphen heißen *biconnected*, wenn sie auf einem einfachen Kreis liegen.
- eine zweifache Zusammenhangskomponente wird auch als *Block* bezeichnet
- aus einem zweifach zusammenhängenden Graphen müssen mindestens zwei Knoten entfernt werden, damit der Graph nicht mehr zusammenhängend ist
- die zweifachen Zusammenhangskomponenten können in Linearzeit berechnet werden, Vergleich Algorithmus 2.

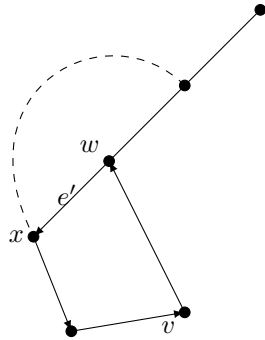
#### **Beweisidee:**

per Induktion über die Anzahl der Kantendurchläufe:

- eine Kante heißt *offen*, wenn noch kein Backtracking erfolgt ist
- eine Kante heißt *fertig*, falls Backtracking erfolgt ist
- eine Komponente heißt *offen* / *fertig*, falls ihre erste Kante *offen* / *fertig* ist
- $G_t$  ist der durch die nummerierten Kanten induzierte Teilgraph nach  $t$  Kantendurchläufen
- offene Zusammenhangskomponenten werden als  $G_t^{(i)}$ , seine ersten Kanten als  $E_t^{(i)}$  mit  $e_i^{(i)}$ ,  $1 \leq i \leq k_t$  in der Reihenfolge, wie sie markiert wurden
- zu Zeigen sind die folgenden Invarianten:
  1. alle Kanten einer fertigen Zusammenhangskomponenten zeigen auf die erste Kante der Komponente
  2. auf dem Stack  $C$  liegen (von unten nach oben)  $e_t^{(1)}, \dots, e_t^{(k_t)}$
  3. auf dem Stack  $S$  bilden die Kanten aus  $E_t^{(1)}, \dots, E_t^{(k_t)}$  Intervalle (in dieser Reihenfolge)

- alle Aussagen sind am Anfang richtig, bleibt zu zeigen dass sie auch für  $t > 0$ , nach  $t - 1$  Durchläufen gelten (zwei Fälle):

1. Vorwärtsdurchlauf:



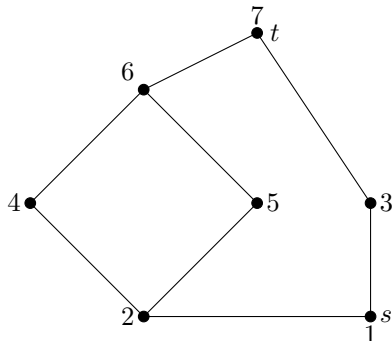
- $e$  ist Baumkante
  - $\Rightarrow w$  ist Knoten vom Grad 1 in  $G_t$
  - $\Rightarrow e$  ist einzige Kante einer neuen offenen Komponente
  - $\Rightarrow$  alle Invarianten gelten
- $e$  ist Rückwärtskante zum Knoten  $w$ 
  - $\Rightarrow e$  bildet zusammen mit dem letzten (bei  $w$  beginnenden) Teilstück des Weges der offenen Kanten einen einfachen Kreis
  - $\Rightarrow$  alle Kanten auf diesem Kreis gehören zur gleichen Zusammenhangskomponente von  $G_t$
  - $\Rightarrow$  alle nach  $e$  durchlaufenen Kanten werden aus  $C$  entfernt
  - $\Rightarrow$  alle Invarianten gelten

2. Backtracking:

- $e$  wird fertig
  - $\Rightarrow$  durch 2. Invariante:  $e$  erste Kante der offenen Komponente  $\Leftrightarrow$  sie liegt oben auf  $C$
- es gibt keine Quer- oder Vorwärtskanten in der ungerichteten Tiefensuche
  - $\Rightarrow$  die Zusammenhangskomponente kann nicht mehr größer werden
  - $\Rightarrow$  Zusammenhangskomponente ist fertig
- durch 3. Invariante werden durch die **repeat**-Schleife die richtigen Kanten von  $S$  entfernt
  - $\Rightarrow$  die Invarianten gelten

- Linearzeit: jede Kante wird höchstens einmal auf  $C$  und  $S$  gelegt

## 1.2 Knotenreihenfolge bestimmen (2. Schritt)

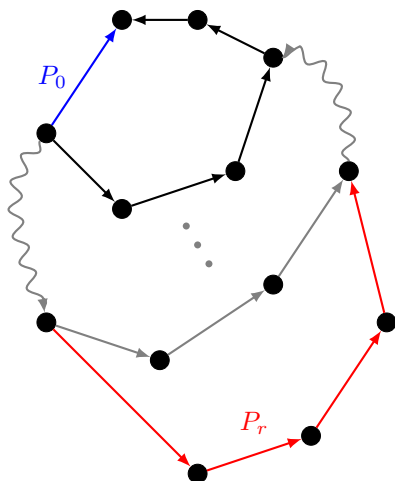


- der Algorithmus nutzt die *st-Ordnung* als Reihenfolge der einzufügenden Knoten
- eine *st-Ordnung* ist wie folgt definiert:
 
$$\exists 1 \leq i < j < k \leq n \text{ mit } \{v_i, v_j\}, \{v_j, v_k\} \in E$$
- $G = (V, E)$ , zweifach zusammenhängender Graph  $\Leftrightarrow \exists$  zu jeweils zwei Knoten  $s \neq t \in V$  eine *st-Ordnung*

- durch eine *Orientierung* erhält jede Kante in einem Graphen eine Richtung
- eine *st-Orientierung* ist azyklisch mit  $s, t$  als einzige Quelle/Senke
- eine *st-Ordnung* kann mithilfe von *topologischer Sortierung* aus einer *st-Orientierung* in Linearzeit berechnet werden



### 1.2.1 Ohrendekomposition



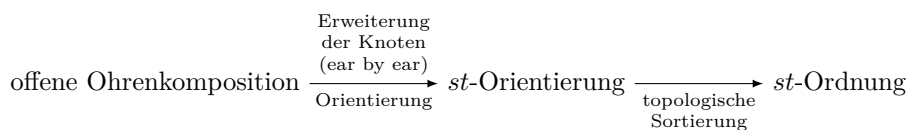
- die Folge  $D = (P_0, \dots, P_r)$  von (offenen) Pfaden heißt (*offene*) *Ohrendekomposition*, falls
  - für  $G_i = (V_i, E_i)$  gilt  $V_i = \bigcup_{j=0}^i V(P_j)$
  - und  $E_i = \bigcup_{j=0}^i E(P_j)$ ,  $0 \leq i \leq r$
  - $E(P_0), \dots, E(P_r)$  ist eine Partition von  $E$
  - für alle  $P_i = (v_0, e_1, v_1, \dots, e_k, v_k)$ ,  $1 \leq i \leq r$  gilt
    - $\{v_0, v_k\} \subseteq V_{i-1}$
    - $\{v_1, \dots, v_{k-1}\} \cap V_{i-1} = \emptyset$
- eine Ohrendekomposition beginnt mit der Kante  $\{s, t\} \in E$ , wenn  $P_0 = (s, \{s, t\}, t)$

- für jeden zweifach zusammenhängenden Graphen gibt es für jede Kante  $e = \{s, t\}$  eine Ohrendekomposition, die mit  $e$  beginnt
- es kann in Linearzeit eine  $st$ -Orientierung aus einer offenen Ohrendekomposition konstruiert werden, die mit  $\{s, t\}$  beginnt

**Beweis:**

Durch Konstruktion:

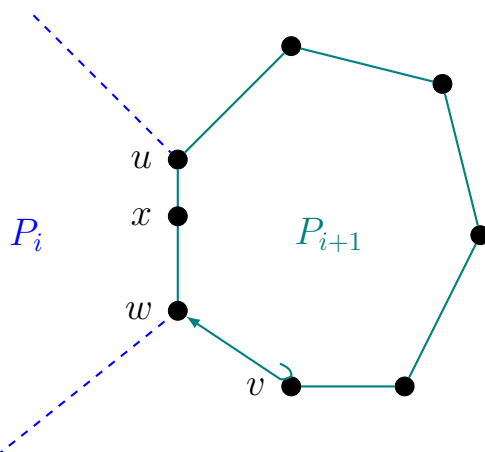
- aus  $P_0, \dots, P_r$ , beginnend mit  $\{s, t\}$ , wird eine  $st$ -Orientierung konstruiert, indem
  - $P_0$  von  $s$  nach  $t$  orientiert wird
  - $P_i = (u, \dots, w)$ ,  $1 \leq i \leq r$  von  $u$  nach  $w$  orientiert wird, falls  $u$  in der von  $P_0, \dots, P_{i-1}$  induzierten partiellen Ordnung vor  $w$  liegt, sonst umgekehrt
- ] Weg von der Ohrendekomposition zu einer  $st$ -Ordnung:



- Bestimmung einer Ohrendekomposition durch einen *Tiefensuchbaum*:

- Baumkanten  $\{v, w\}$  werden als  $v \rightarrow w$  bezeichnet
- Rückwärtskanten  $\{v, w\}$  werden als  $v \hookleftarrow w$  bezeichnet
- ein  $uv$ -Pfad mit höchstens Baumkanten wird als  $u \xrightarrow{*} v$  bezeichnet
- Konstruktion der Ohrendekomposition:

1.  $P_0$  ist das Anfangsohr
2.  $\exists v \hookleftarrow w \notin E_i$  nach Konstruktion von  $P_0, \dots, P_i, i \geq 0$   
 $\Rightarrow P_{i+1}$  ist wie folgt definiert ( $v, w, x \in V$ ):



- $w, x \in V_i$
- $v \hookleftarrow w \notin E$
- $w \rightarrow x$
- $x \xrightarrow{*} v$
- ist  $u$  der letzte Knoten auf  $x \xrightarrow{*} v$  mit  $u \in V_i$  ( $u = v$  ist möglich)  
 $\Rightarrow P_{i+1} = u \xrightarrow{*} v \hookleftarrow w$
- aus  $w \rightarrow x \xrightarrow{*} u$  folgt, dass  $P_{i+1}$  offen ist ( $P_{i+1}$  ist das Ohr zu  $v \hookleftarrow w$ , *trivial*, falls  $u = v$ )

- eine offene Ohrendekomposition  $D(T) = (P_0, \dots, P_r)$  überdeckt  $G = (V, E)$  vollständig  
 $\Rightarrow V = V_r, E = E_r$ :

**Beweisidee:**

Mit der zweifach Verbundenheit des Graphen kann man durch Konstruktion mithilfe eines Knotens  $u \notin V_r$  einen Widerspruch herbei führen.

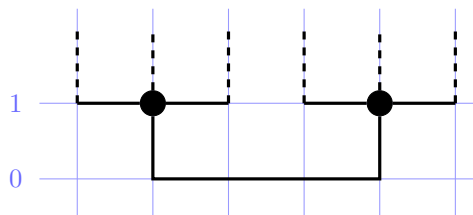
- die Ohren werden aufgrund der Orientierung der zugehörigen Baumkante orientiert, in der Reihenfolge von  $D(T)$
- die Dekomposition definiert für jedes  $i = 0, \dots, r$  eine partielle Ordnung (reflexiv, transitiv, antisymmetrisch, nicht für alle Paare definiert - sonst total)

$$\prec_i: \{v, w\} \in E_i \text{ von } v \text{ nach } w \text{ orientiert} \Rightarrow v \prec_i w$$

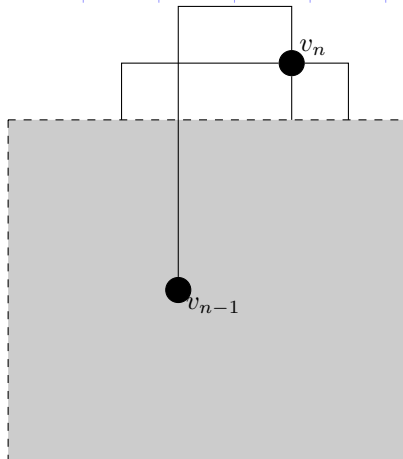
- die Orientierung von  $D(T)$  liefert eine  $st$ -Orientierung von  $G$
- die erhaltene Ordnung von  $V_i$  ist eine lineare Erweiterung von  $\prec_j$  für alle  $0 \leq j \leq i$  ( $\prec_j \subseteq \prec_i$ )  
 $\Rightarrow$  die Ordnung ist eine  $st$ -Ordnung von  $G$
- Vergleich Algorithmus 3.

### 1.3 Layout einer Komponente (3. Schritt)

- gegeben sind der zweifach zusammenhängende Graph  $G$  und die  $st$ -Ordnung von  $G$
- für jeden Knoten gilt  $d_g^{\leftarrow}(v) = \#$  adjazenter Vorgänger
- für jeden Knoten gilt  $d_g^{\rightarrow}(v) = \#$  adjazenter Nachfolger



- $d(v) = d_g^{\leftarrow}(v) + d_g^{\rightarrow}(v)$
- $1 \leq d_g^{\leftarrow}(v)$
- $3 \leq d_g^{\rightarrow}(v)$
- Plazierung der ersten beiden Knoten (**links**)

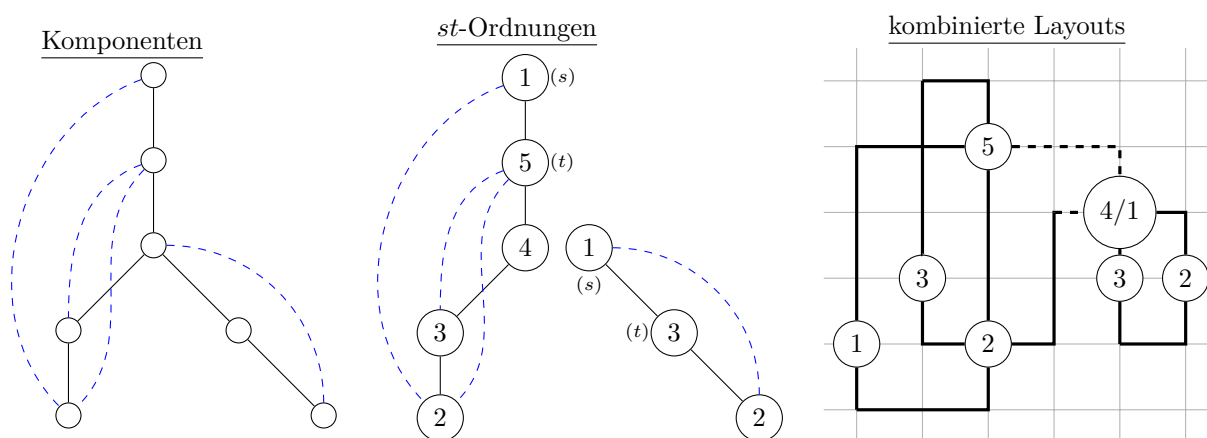


- für alle adjazenten Nachfolger werden jeweils die Gittervertikalen durch  $v$  sowie, falls nötig, zwei neue links und rechts vom Layout reserviert
- alle anderen Knoten werden auf der Gitterhorizontalen mit  $y(v_i) = i$  entsprechend der Anzahl der adjazenten Vorgänger platziert
- $v_n$  kommt auf die Gitterhorizontale  $y(t) = n$  (siehe links)

- die benötigte Gittergröße ist höchstens  $(m - n + 1) \times n$ 
  - erste beiden Knoten brauchen Höhe 1
  - die weiteren Knoten erhöhen die Höhe um 1
  - der letzte Knoten erhöht die Höhe höchstens um 2 $\Rightarrow$  Höhe  $n$ 
  - erste beiden Knoten brauchen Breite  $d_g^{\rightarrow}(v_1) + d_g^{\rightarrow}(v_2) - 2$
  - die weiteren Knoten vergrößern die Breite um  $d_g^{\rightarrow}(v_i) - 1$
  - der letzte Knoten erhöht die Breite nicht $\Rightarrow$  Breite  $= \sum_{v \in V} (d_g^{\rightarrow}(v) - 1) + 1 = m - n + 1$

- die Gesamtzahl der Knicke ist höchstens  $2m - 2n + 4$  und keine Kante hat mehr als 2 Knicke
  - für  $v_1, v_2$  werden  $d_g^{\rightarrow}(v_1) + d_g^{\rightarrow}(v_2) - 1$  Knicke erzeugt
  - für jeden Knoten  $v \neq v_1, v_2, v_n$  werden  $d_g^{\leftarrow}(v) - 1 + d_g^{\rightarrow}(v) - 1 = 2d_G(v) - 2$  Knicke erzeugt
  - für  $v_n$  werden 4 Knicke benötigt, falls  $d_G(v_n) = 4$ , sonst nur  $d_G(v_n) - 1$
  - $\Rightarrow$  maximal:  $\sum_{v \in V} d_G(v) - 2 + 4 = 2m - 2n + 4$
  - erste und letzte Kante  $\{v_1, v_2\}, \{v_{n-1}, v_n\}$  haben maximal 2 Knicke (Konstruktion)
  - jede andere Kante hat höchstens einen Knick auf der Gitterhorizontalen durch beide Endknoten
- ist  $G$  planar und liegen  $s, t$  auf der äußeren Facette
  - $\Rightarrow$  für jede  $st$ -Ordnung  $v_1, \dots, v_n$  liegt  $v_i$  auf der äußeren Facette des von  $v_1, \dots, v_{i-1}$  induzierten Teilgraphen, sowie die Vorgänger von  $v_i$  bilden auf der äußeren Facette ein Intervall
- durch Einfügen von neuen Spalten direkt neben dem gerade zu platzierenden Knoten, berechnet der Algorithmus kreuzungsfreie, orthogonale Gitterlayouts
- gibt einen Graph, der 3 Knicke an einer Kante braucht!

## 1.4 Kombination der Komponentenlayouts (4. Schritt)

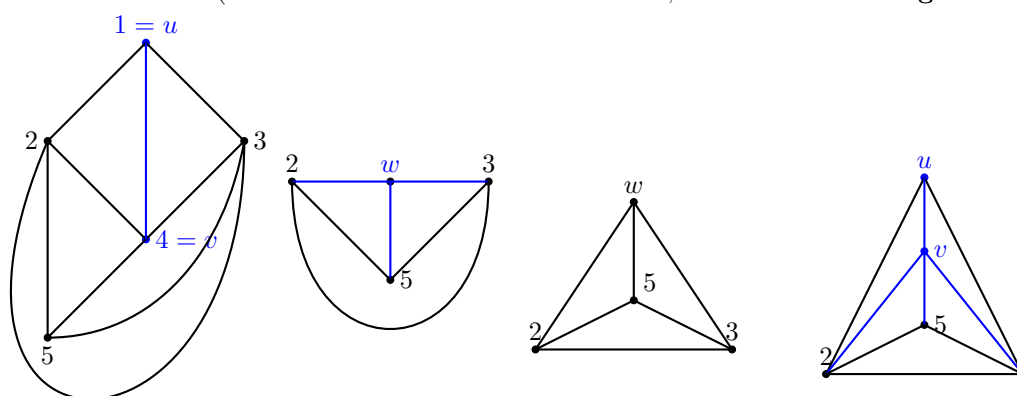


## 2 Kreuzungsfreie geradlinige Gitterlayouts

Jeder planare Graph hat eine kreuzungsfreie, geradlinige Standardrepräsentation. (Nachteil der Konstruktion: hohe Auflösung nötig, weil die Kantenlängen stark variieren können)

**Beweisidee:**

Durch Konstruktion (Funktioniert nicht mit allen Nachbarn, nur bei Knoten mit **genau** zwei Nachbarn!):



## 2.1 Shift-Methode (Gittergröße $\mathcal{O}(n) \times \mathcal{O}(n)$ )

### 2.1.1 Definitionen:

**connectivity:**  $\kappa(G)$  ist die kleinste Anzahl von Knoten, die entfernt werden müssen, damit  $G$  nicht mehr zusammenhängend ist

**maximal planar/trianguliert:** durch Hinzufügen einer Kante, wäre der Graph nicht mehr planar

**kombinatorische Einbettung:** z.B. die kreisförmige Anordnung der Kanten um jeden Knoten des Graphen

**chord:** eine Kante zwischen zwei Knoten  $v, w$  auf einem Kreis  $C$ , wobei  $v$  und  $w$  auf  $C$  nicht nebeneinander liegen

**kanonische Ordnung:** eine Ordnung  $\pi = (v_1, \dots, v_n)$ , für die für jedes  $3 \leq k \leq n$  gilt:

- co1** Knoten  $\{v_1, \dots, v_k\}$  induzieren einen zweifach verbundenen und innen triangulierten Graphen
- co2**  $(v_1, v_2)$  ist eine Außenkante von  $G_k$
- co3**  $k < n \Rightarrow v_{k+1}$  liegt auf der Außenfläche von  $G_k$  und alle Nachbarn von  $v_{k+1}$  in  $G_k$  erscheinen nacheinander auf  $C_0(G_k)$

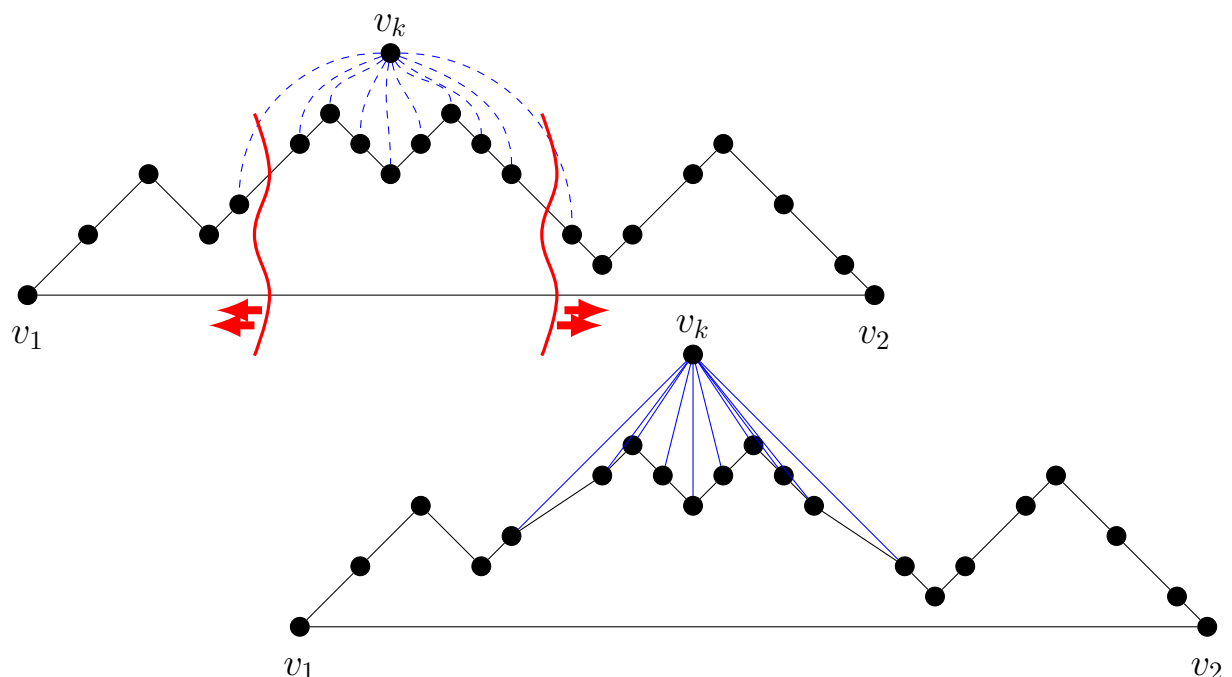
Jeder triangulierte, planare Graph besitzt eine kanonische Ordnung (durch Konstruktion mit **co1-3** klar)

### 2.1.2 Ablauf

1. Hinzufügen von Kanten bis der Graph zweifach zusammenhängend ist
2. Triangulation durch Hinzufügen weiterer Kanten
3. Bestimmung einer kanonischen Ordnung der Knoten (Vergleich Algorithmus 4., in Linearzeit)
4. inkrementelle Bestimmung relativer Koordinaten der Knoten
5. Bestimmung absoluter Koordinaten
6. Herausnehmen aller Kanten, die nicht zum Ausgangsgraphen gehören

### 2.1.3 Algorithmus

- Einfügereihenfolge der Knoten entspricht der kanonischen Ordnung
- Anfangsknoten  $v_1, v_2, v_3$
- Einfügen von  $v_k$  in das Layout von  $G_{k-1}$
- zum Sicherstellen der Planarität werden die Knoten zum Teil horizontal verschoben
- jeder Knoten hat eine Liste  $L(v)$  mit Knoten, die zusammen mit  $v$  verschoben werden müssen



#### 2.1.4 Anmerkungen

- für zwei Gitterpunkte mit *Manhattan distance* ( $L_1(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2|$ ) wird der Gitterpunkt für  $v_k$  durch den Schnittpunkt der Geraden mit Steigung  $-1$  durch  $P_2$  und Steigung  $1$  durch  $P_1$  berechnet (Vergleich Algorithmus 5. )
- gestartet wird mit den Punkten  $P(v_1) = (0, 0), P(v_2) = (2, 0), P(v_3) = (1, 1)$  (entspricht  $G_3$ )
- Anfangslisten  $L(v_i) = \{v_i\}, i = 1, 2, 3$
- zum Zeichnen von  $G_k$  müssen folgende 3 Invarianten gelten:
  1.  $P(v_1) = (0, 0)$  und  $P(v_2) = (2k - 4, 0)$
  2.  $x(w_1) < x(w_2) < \dots < x(w_t)$  für  $C_0(G_k) = (v_1 = w_1, \dots, w_t = v_2)$
  3. alle Kanten  $(w_i, w_{i+1})$  auf  $C_0(G_k)$  sind geradlinig mit Steigung  $1$ , bzw.  $-1$
- Nachteil: sehr kleine Winkel möglich  $\Rightarrow$  unleserliche Graphen können entstehen

# KOMBINATORISCHE OPTIMIERUNG (FLUSSMETHODEN)

## Layoutprobleme für planare Graphen

- geradlinige Layouts mit „guter“ Winkelauflösung
  - orthogonale Layouts mit einer minimalen Anzahl an Knicken
  - aufwärtsgerichtete Layouts gerichteter, azyklischer Graphen
- ⇒ Lösung mit Flussmethoden

## 1 Grundlagen

### 1.1 planare Einbettung

- alle Kanten (Jordan-Kurven) schneiden sich nur in ihren Endpunkten
- feste Lage (Knoten haben feste Positionen mit kombinatorischer Einbettung)
- zerlegt die Ebene in Flächen (Gebiete/Facetten)

Für zusammenhängende, planare Graphen gilt  $n - m + f = 2$ .

### 1.2 klassisches $st$ -Flussmodell

- Netzwerk  $(D = (V, A), s, t, c)$  mit

$D = (V, A)$  gerichteter Graph

$s$  Quelle

$t$  Senke

$c$  Kapazitäten  $c : A \rightarrow \mathbb{R}_0^+$

- $x : A \rightarrow \mathbb{R}_0^+$  heißt Fluss, wenn

**1. Kapazitätsbedingung:**  $\forall (i, j) \in A : 0 \leq x(i, j) \leq c(i, j)$

**2. Flusserhaltungsbedingung:**  $\forall i \in V \setminus \{s, t\} : \sum_{j : (i, j) \in A} x(i, j) - \sum_{j : (j, i) \in A} x(j, i) = 0$

- Wert eines Flusses  $x$ :

$$w(x) = \sum_{j : (s, j) \in A} x(s, j) = \sum_{j : (j, t) \in A} x(j, t)$$

- klassisches Dualitätsresultat:  $w(x)$  entspricht der Kapazität eines  $s$ - $t$ -Schnittes mit

–  $S \subset V$

–  $s \in S, t \in V \setminus S$

–  $C(S, V \setminus S) = \sum_{\substack{(i, j) \in A \\ i \in S, j \in V \setminus S}} c(i, j)$

### 1.3 Allgemeines Flussmodell

- wie klassisches Flussmodell mit zusätzlich

– untere und obere Kapazitäten statt einfachen Kapazitäten ( $l : A \rightarrow \mathbb{R}_0^+, u : A \rightarrow \mathbb{R}_0^+$ )

– Knotenbewertung  $b : v \rightarrow \mathbb{R}$  mit  $\sum_{i \in V} b(i) = 0$  (Knoten mit  $b(i) > 0$  sind Quellen, mit  $b(i) < 0$  sind Senken)

- $x : A \rightarrow \mathbb{R}_0^+$  heißt Fluss, wenn

**1. Kapazitätsbedingung:**  $\forall (i, j) \in A : l(i, j) \leq x(i, j) \leq u(i, j)$

**2. Flusserhaltungsbedingung:**  $\forall i \in V : \sum_{j : (i, j) \in A} x(i, j) - \sum_{j : (j, i) \in A} x(j, i) = b(i)$

## 1.4 Fluss mit minimalen Kosten

Zusätzlich zum Flussmodell ist noch die Funktion  $cost : A \rightarrow \mathbb{R}_0^+$  gegeben.  
Gesucht ist somit die Minimierung von

$$cost(x) = \sum_{(i,j) \in A} cost(i,j) \cdot x(i,j)$$

## 2 Schranken für die Winkelauflösung in geradlinigen Layouts

- gesucht ist eine geradlinige Einbettung von einem planaren Graphen  $G$  mit maximalem

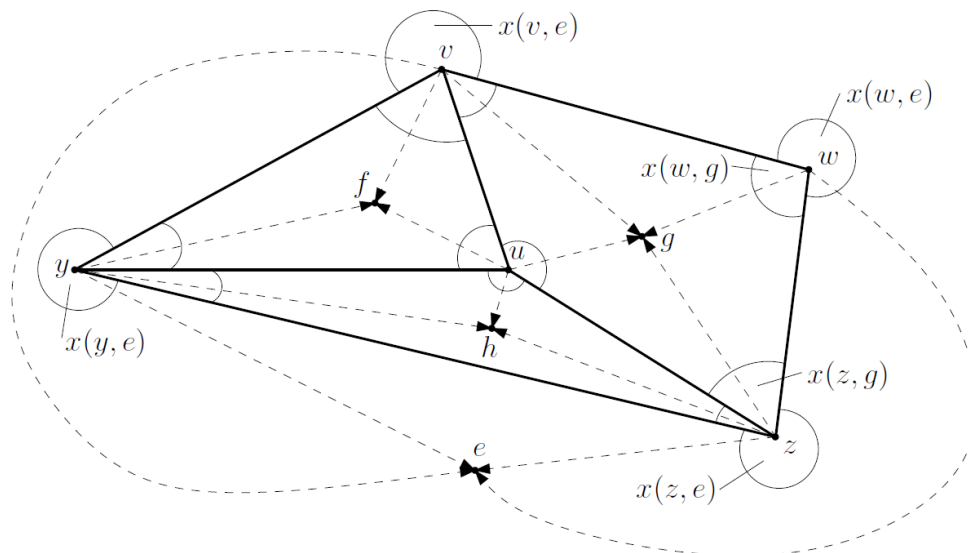
$$x_{min} = \min_{\substack{v \in V, f \in \mathcal{F} \\ v \text{ inzident zu } f}} \{x(v, f)\}$$

- $\mathcal{NP}$ -schwer

**Einschränkung:** planare kombinatorische Einbettung ist gegeben

**Modell:** Formulierung des Problems als Flussmodell (liefert untere Schranke für  $x_{min}$ )

### 2.1 Grundidee der Konstruktion



Es gilt:

1. **Knotenbedingung:**  $\forall v \in V : \sum_{f \in \mathcal{F} \text{ inzident zu } v} x(v, f) = 2 \cdot \pi$

2. **Facettenbedingung:**

$$\begin{aligned} \forall f \in \mathcal{F} \setminus \{f_0\} : \sum_{v \in \text{inzident zu } f} x(v, f) &= (d_G(f) - 2) \cdot \pi \\ f_0 : \sum_{v \in \text{inzident zu } f_0} x(v, f_0) &= d_G(f_0) \cdot \pi - (d_G(f_0) - 2) \cdot \pi \\ &= (d_G(f_0) + 2) \cdot \pi \end{aligned}$$

## 2.2 Definition des Flussnetzwerks $N(G) = (D = (W, A), b, l, u)$

$$\begin{aligned} W &= V \cup \mathcal{F} \\ A &= \{(v, f) \in V \times \mathcal{F}, v \text{ inzident zu } f\} \\ b(v) &= 2\pi, \forall v \in V \\ b(f) &= -(d_G(f) - 2)\pi, \forall f \in \mathcal{F} \setminus \{f_0\} \\ b(f_0) &= -(d_G(f) + 2)\pi \\ l(a) &= 0, \forall a \in A \\ u(a) &= 2\pi, \forall a \in A \end{aligned}$$

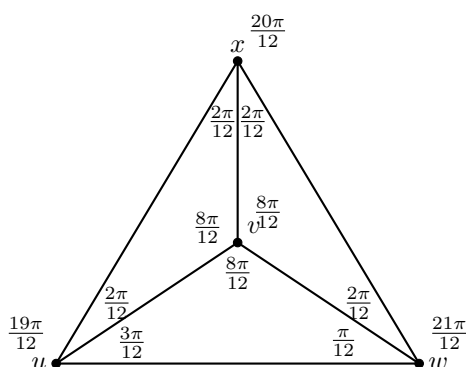
## 2.3 Definition des Flussnetzwerks $N_{s,t}(G) = (D = (W_{s,t}, A_{s,t}), l, u)$

$$\begin{aligned} W_{s,t} &= W \cup \{s, t\} \\ A_{s,t} &= A \cup \{(s, v) | v \in V\} \cup \{(f, t) | f \in \mathcal{F}\} \\ l(s, v) &= l(f, t) = 0, \forall v \in V, f \in \mathcal{F} \\ u(s, v) &= 2\pi, \forall v \in V \\ u(f, t) &= -(d_G(f) - 2)\pi, \forall f \in \mathcal{F} \setminus \{f_0\} \\ u(f_0, t) &= -(d_G(f) + 2)\pi \end{aligned}$$

## 2.4 untere Schranke von $x_{min}$

$\alpha$  ist genau dann eine untere Schranke von  $x_{min}$  für einen maximalen Fluss  $x$ , falls  $\max w(x_{s,t})$  von  $N_{s,t}(G)$  gleich  $\max w(x'_{s,t})$  von  $N'_{s,t}(G) = ((W_{s,t}, A_{s,t}), l', u)$  mit  $l'(a) = \alpha, \forall a \in A$   
 $\Rightarrow$  Beweis mit Ford und Fulkerson (Kapazität jedes  $s$ - $t$ -Schnittes in  $N'_{s,t}(G)$  ist nicht kleiner als die Kapazität eines minimalen  $s$ - $t$ -Schnittes in  $N_{s,t}(G)$ ).

## 2.5 Konstruktion eines Flusses

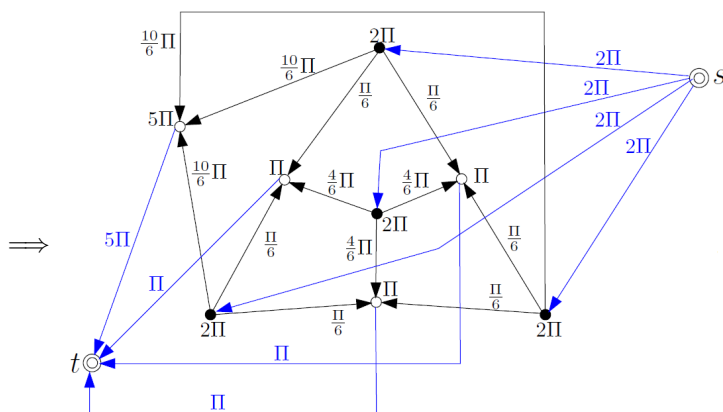
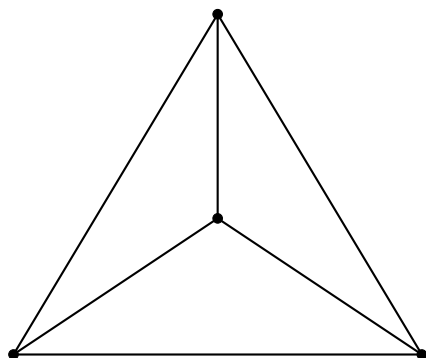


### 2.5.1 Definition

**lokal konsistent:** Zuweisung von Winkelwerten, die Knoten- und Facettenbedingungen erfüllen

! nicht zu jeder lokal konsistenten Zuweisung gibt es auch eine Einbettung, die diese realisiert

Beispiel (Konstruktion):



Fluss  $x$  in  $N'_{s,t}(G)$  mit  $w(x) = 8\pi$  (maximal)  
 $x_{min} = \frac{\pi}{6}$



- es gibt für jeden triangulierten, planar eingebetteten Graphen eine lokal konsistente Winkelzuweisung mit  $x_{min} \in \Omega(\frac{1}{\Delta_G})$
- oberes liefert nur eine obere Schranke für die untere Schranke, da nicht jede lokal konsistente Zuweisung realisierbar ist
- ein **Dreiecksgraph** ist bis auf die äußere Facette trianguliert (z.B. der Wheel-Graph ( $W_d$ ))
- für jeden planaren Dreiecksgraphen mit kombinatorischer Einbettung, beschrieben durch  $\mathcal{F}$ , und einer vorgegebenen Winkelzuweisung  $(\alpha_i, \beta_i, \gamma_i, \forall 1 \leq i \leq d)$  gibt es eine geradlinige Realisierung der Einbettung mit dieser Winkelzuweisung, gdw.

1.  $\sum_{i=0}^d \gamma_i = 2\pi, \forall 1 \leq i \leq d$
2.  $\alpha_i + \beta_i + \gamma_i = \pi, \forall 1 \leq i \leq d$
3.  $\prod_{i=0}^d \frac{\sin \alpha_i}{\sin \beta_i} = 1$

( $\Rightarrow$ ) **Beweis:**

1./2. gilt durch Definition

3.  $L_i$  ist die Länge bei Kante  $e_i$  zwischen den Winkeln  $\beta_i$  und  $\alpha_{(i \bmod d)+1}$

$$\begin{aligned} \Rightarrow \prod_{i=0}^d \frac{L_{(i \bmod d)+1}}{L_i} &= \frac{L_2}{L_1} \cdot \frac{L_3}{L_2} \cdot \dots \cdot \frac{L_1}{L_d} = 1 \\ \Rightarrow \frac{L_{(i \bmod d)+1}}{L_i} &= \frac{\sin \alpha_{(i \bmod d)+1}}{\sin \beta_{(i \bmod d)+1}} \\ \Rightarrow \text{Bedingung 2 ist wahr} \end{aligned}$$

( $\Leftarrow$  (**Konstruktion der Zeichnung mit den Winkeln**)) **Beweis:**

- beliebiges  $L_1$  wählen, Zeichnen der Kante  $\{v, v_2\}$
- Berechnen von  $L_{i+1}$  mit  $L_i$  ( $i = 1, \dots, d-1$ ) durch  $L_{i+1} = L_i \cdot \frac{\sin \alpha_{i+1}}{\sin \beta_{i+1}}$ ,  
Zeichnen der Kante  $\{v, v_{(i-2 \bmod d)+1}\}$  mit  $\gamma_{i+1}$  am Knoten  $v$

$\Rightarrow$  (Bedingung 1/2)

- Konstruktion eines neuen Dreiecks mit vorgeschriebenen Winkeln, die  $\{v, v_{i+1}\}$  gemeinsam haben mit dem vorherigen Dreieck, ist gültig
- Dreiecke überlappen sich nicht

$$\Rightarrow \text{durch Konstruktion erhalten wir } L_d = L_1 \cdot \prod_{i=2}^d \frac{\sin \alpha_i}{\sin \beta_i}$$

$\Rightarrow$  (Bedingung 3)

- $\frac{L_1}{L_d} = \frac{\sin \alpha_1}{\sin \beta_1}$
- somit erfüllen  $L_1$  und  $L_d$  das Dreieck

$\Rightarrow$  dadurch, dass alle Winkel positiv sind, ist die Zeichnung planar

### 3 Knickminimierung in orthogonalen Layouts

Das Knickminimierungsproblem (allgemein) ist  $\mathcal{NP}$ -schwer.

**orthogonale Beschreibung** ( $H$ ) keine Kantenlängen, keine Positionen für Knoten

eine Folge von Facettenbeschreibungen  $H(f)$ ,  $f \in \mathcal{F}$  mit Elementen  $(e, \delta, x)$  definiert durch

- $e \in E$
- $\delta$  eine Folge aus  $\{0, 1\}$ , 0 kodiert einen  $\frac{\pi}{2}$  Knick, 1 einen  $\frac{3\pi}{2}$
- $x$  ein Winkel aus  $\{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\}$

$H$  ist korrekt, falls

**O1** Es gibt eine planare Einbettung, die  $H$  entspricht

**O2**  $(e, \delta_1, x_1), (e, \delta_2, x_2)$

$\Rightarrow \delta_2$  entsteht aus  $\delta_1$  durch kippen jedes einzelnen Bits von  $\delta_1$  und umkehren der Folge  $\delta_1$

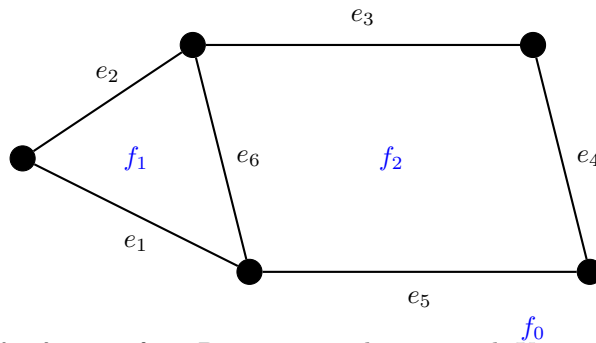
**O3**  $|\delta|_0, |\delta|_1$  sind die Anzahl der 0/1 in  $\delta$

für  $r = (e, \delta, x)$  gilt  $\mathcal{C}(r) = |\delta|_0 - |\delta|_1 + (2 - \frac{2x}{\pi})$

$$\Rightarrow \sum_{r \in H(f)} \mathcal{C}(r) = \begin{cases} 4 & f \in \mathcal{F} \setminus \{f_0\} \\ -4 & f = f_0 \end{cases}$$

**O4**  $\forall v \in V$  ist die Summe der Winkel bei  $v$  gleich  $2\pi$

**Beispiel (orthogonale Beschreibung):**



$$f_0 : (e_1, 11, \frac{\pi}{2}), (e_5, 111, \frac{3\pi}{2}), (e_4, \emptyset, \pi), (e_3, \emptyset, \pi), (e_2, \emptyset, \frac{\pi}{2})$$

$$f_1 : (e_1, 00, \frac{3\pi}{2}), (e_2, \emptyset, \frac{\pi}{2}), (e_6, 00, \pi)$$

$$f_2 : (e_5, 000, \frac{\pi}{2}), (e_6, 11, \frac{\pi}{2}), (e_3, \emptyset, \pi), (e_4, \emptyset, \frac{\pi}{2})$$

**orthogonales Layout** feste Positionen und somit auch Kantenlängen für alle Teile des Graphen

#### 3.1 Knickminimierung mit vorgegebener Einbettung

$$G = (V, E), \mathcal{F}, f_0$$

↓ Minimierung der Knickzahl (1.Schritt)

orthogonale Beschreibung

↓ Kompaktierung (2.Schritt)

orthogonales Layout

##### 3.1.1 Schritt 1: orthogonale Beschreibung

**Definition des Flussnetzwerks**  $N(G) = ((W, A), l, u, b, cost)$

$$W = V \cup \mathcal{F}$$

$$A = \{(v, f) \in V \times \mathcal{F}, v \text{ inzident zu } f\} \cup \{(f, g) \in \mathcal{F}, f \text{ und } g \text{ haben gemeinsame Kante}\}$$

$$b(v) = 4 \frac{\pi}{2} \Rightarrow 4, \forall v \in V$$

$$b(f) = -2(d_G(f) - 2) \frac{\pi}{2} \Rightarrow -2(d_G(f) - 2), \forall f \in \mathcal{F} \setminus \{f_0\}$$

$$b(f_0) = -2(d_G(f) + 2) \frac{\pi}{2} \Rightarrow -2(d_G(f) + 2)$$

$$\begin{aligned}
l(v, f) &= 1, l(f, g) = 0 \\
u(v, f) &= 4, u(f, g) = \infty \\
cost(v, f) &= 0, cost(f, g) = 1
\end{aligned}$$

Eine Flusseinheit entspricht einem  $\frac{\pi}{2}$ -Winkel/-Knick

- $N(G) = ((W, A), l, u, b, cost)$  ist ein Flussnetzwerk (Beweis durch Satz von Euler)
- zu jedem planaren Graph mit  $\Delta(G) \leq 4$  und kombinatorischer Einbettung existiert genau eine orthogonale Beschreibung mit  $k$  Knicken, wenn es einen Fluss  $x$  in  $N(G)$  mit  $k$  Kosten gibt

**Beweis:**

**TODO**

### 3.1.2 Schritt 2: Kompaktierung

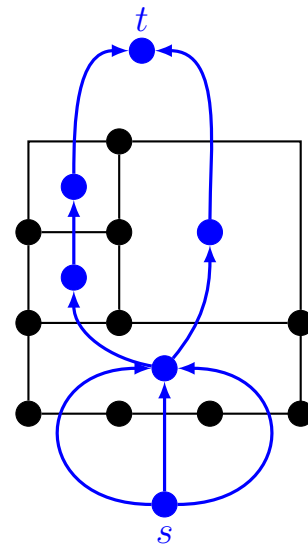
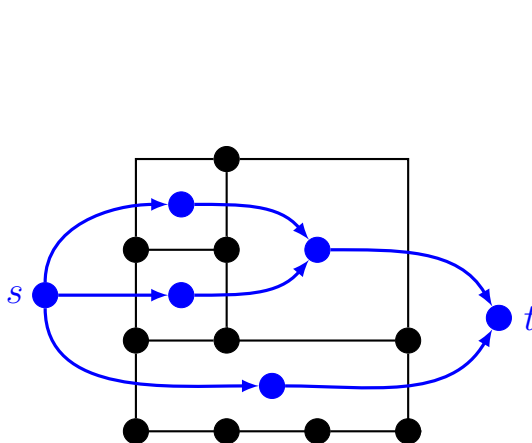
- betrachten den Spezialfall mit der Eigenschaft, dass alle Facetten in  $H(G)$  Rechtecke sind
- zur Konstruktion wird ein Flussnetzwerk verwendet
- für den Spezialfall kann garantiert werden: das konstruierte Layout hat:

1. minimale Gesamtkantenlänge
2. minimale Fläche

- Konstruktion von zwei Flussnetzwerken  $(N_{ver}, N_{hor})$  mit  $cost(a) = 1, l(a) = 1, u(a) = \infty$

$$N_{ver} = ((W_{ver}, A_{ver}), s, t \in W_{ver}, l, u, cost)$$

$$N_{hor} = ((W_{hor}, A_{hor}), s, t \in W_{hor}, l, u, cost)$$



#### • Beobachtungen:

- alle Knicke liegen auf  $f_0$
- wenn gegenüberliegende Seiten die Gleiche Länge zugewiesen bekommen, kann ein korrektes Layout konstruiert werden

- für ganzzahlige Kantenbewertung  $x_{ver}, x_{hor}$  mit minimalen Kosten im entsprechenden Flussnetzwerk und orthogonaler Beschreibung, die nur aus Rechtecken besteht, gilt:

1.  $x_{ver}, x_{hor}$  ist ein Fluss gdw. die Kantenlängen ein korrektes Layout induzieren

**Begründung:** Äquivalenz der Flusserhaltungsbedingung und Layouteigenschaft (gegenüberliegende Seiten haben gleiche Länge)

2. der Wert von  $x_{ver}$  entspricht der Höhe des Layouts,  $x_{hor}$  entspricht der Breite des Layouts

**Begründung:** durch Konstruktion

3.  $x_{hor} + x_{ver}$  entspricht der Gesamtkantenlänge des Layouts

**Begründung:** durch Konstruktion

$\Rightarrow$  Flüsse mit minimalen ganzzahligen Kosten induzieren ein planares, orthogonales Gitterlayout mit minimaler Fläche und Gesamtkantenlänge

### 3.1.3 Erweiterung auf den allgemeinen Fall

**rectangular refinement** von  $H(G)$  ist eine orthogonale Beschreibung  $H'(G')$  von  $G$  mit

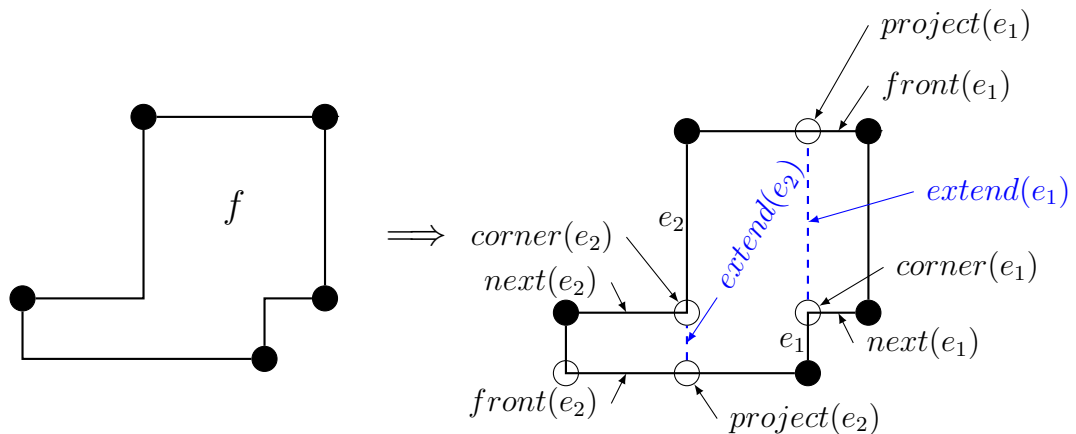
- $G'$  ist entstanden aus einer Sequenz der folgenden Operationen:
    - Hinzufügen eines isolierten Knotens
    - Hinzufügen von Knoten auf Kanten
    - Hinzufügen von Kanten
  - die „Teilbeschreibung“ durch  $H'$  von  $G$  ist die gleiche wie  $H(G)$
  - die Facetten von  $H'(G')$  sind Rechtecke
- ⇒ man erhält eine Zeichnung von  $G$  mithilfe einer Zeichnung von  $G'$  (ohne hinzugefügte Elemente)

#### 1. Realisierung: $G' = G, H'(G') = H(G)$

- Einfügen eines Knotens auf jedem Knick von  $H(G)$
- Aktualisieren von  $H'(G')$

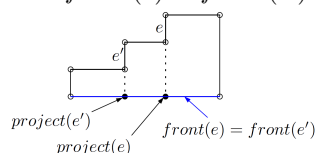
#### 2. Facetten haben beliebige orthogonale Form:

##### 1. innere Facetten:



Ablauf:

- Realisierung für jede Facette  $f \in \mathcal{F}$
- für jede Kante  $e$  in  $H'(f)$  wird folgendes definiert:
  - $next(e)$  : nächste Kante in  $H'(f)$  (counterclockwise)
  - $corner(e)$  : gemeinsamer Knoten von  $e$  und  $next(e)$
  - $turn(e)$  :  $\begin{cases} 1 & : next(e) \text{ knickt nach links ab} \\ 0 & : next(e) \text{ knickt nicht ab} \\ -1 & : next(e) \text{ knickt nach rechts ab} \end{cases}$
  - $front(e)$  : erste Kante  $e'$  in  $H'(f)$  nach  $e$ , für die gilt:  
Summe der  $turn$ -Werte aller Kanten  
von **inklusive**  $e$  bis **exklusiv**  $e'$  gleich 1 ist
- für  $e$  mit  $turn(e) = -1$  wird ein neuer Knoten  $project(e)$  auf  $front(e)$  sowie eine neue geradlinige Kante  $extend(e) = (corner(e), project(e))$  eingefügt, Erweiterung von  $H'(G')$  entsprechend
- falls  $front(e) = front(e') \Rightarrow project(e)$  nach  $project(e')$  eingefügt



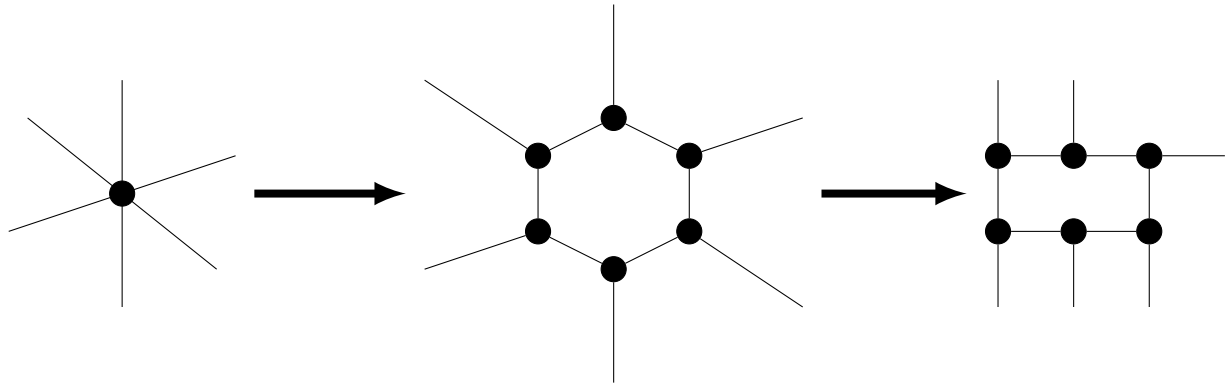
2. **äußere Facette:** um  $G$  wird ein minimales Rechteck gelegt, auf das die Knicke der äußeren Facette projiziert werden

**Bemerkungen:**

- $k$  ist die Anzahl der Knicke in  $H(G)$   
 $\Rightarrow H'(G')$  hat  $\mathcal{O}(n + k)$  Knoten  
 $\Rightarrow H'(G')$  kann in  $\mathcal{O}(n + k)$  konstruiert werden
- die Flussnetzwerke zu  $H'(G')$  garantieren **nicht** mehr minimale Gesamtkantenlänge und minimale Fläche
- mit einem geeigneten Algorithmus für die Flussberechnung (minimale Kosten) kann zu planaren Graphen mit kombinatorischer Einbettung ein orthogonales Layout mit minimaler Knickzahl in  $\mathcal{O}(n^{\frac{7}{4}} \cdot \log n)$  berechnet werden

**Erweiterung auf allgemeine Graphen:**

ohne Gradbeschränkung (als Beispiel):



# LAGEN-LAYOUTS (LAYERED LAYOUT)

Es sollen gerichtete hierarchische Graphen so dargestellt werden, dass die Knoten auf unterschiedlichen Lagen liegen. Es sollen im End-Layout folgende Bedingungen gelten:

1. möglichst alle Kanten sind aufwärts gerichtet
2. möglichst wenig Kanten erzeugen Kreuzungen
3. alle Kanten sind möglichst vertikal und geradlinig dargestellt
4. alle Knoten sind gleichmäßig verteilt und lange Kanten werden vermieden

## Algorithmus zur Konstruktion eines Lagen-Layouts:

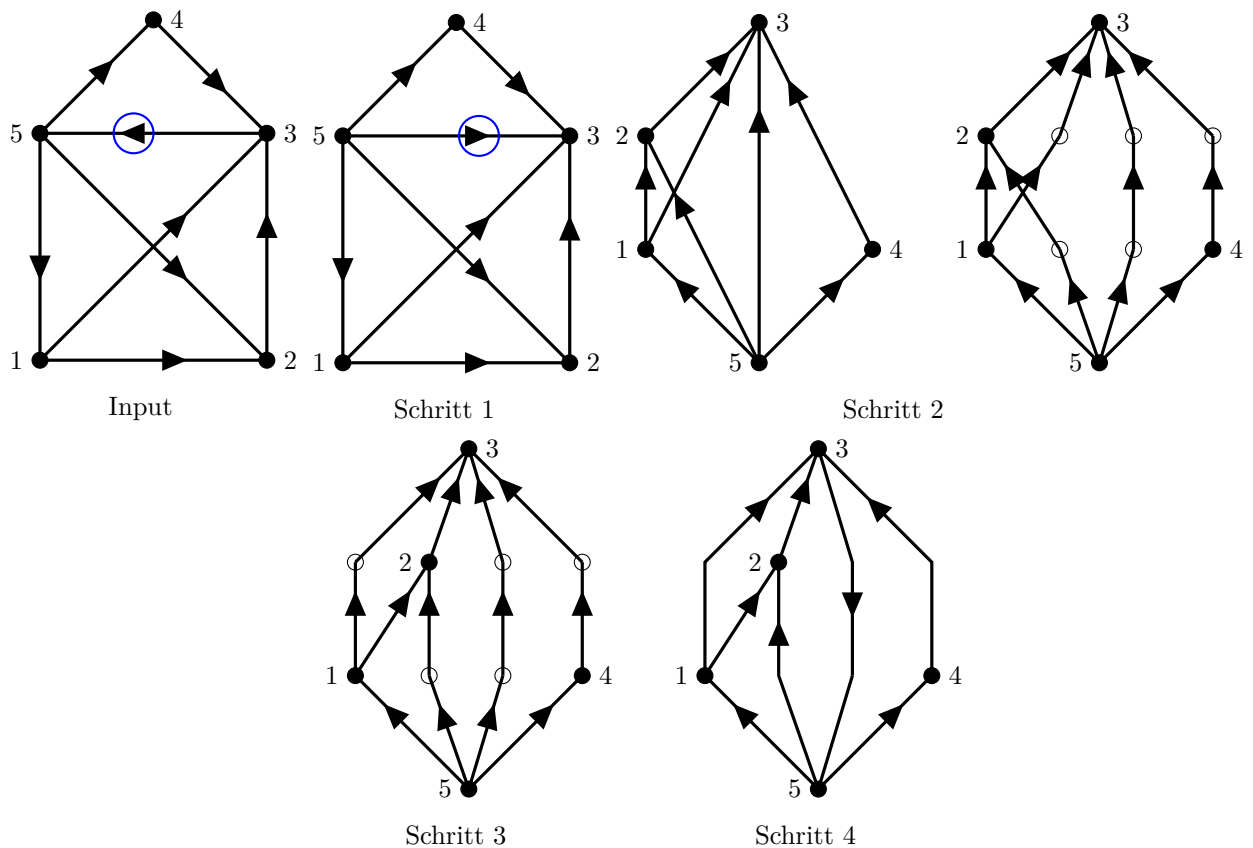
**Schritt 1: Kreise entfernen** Finden einer minimalen Anzahl an Kanten, durch deren Entfernung der Graph azyklisch ist, und drehe ihre Richtung um

**Schritt 2: Lagenzuordnung** Berechnung einer guten Zuordnung der Knoten auf Lagen ( $y$ -Koordinaten), sodass alle Kanten aufwärts gerichtet sind. Alle Kanten, die mindestens eine Lage überqueren werden ersetzt durch einen Pfad mit Dummy-Knoten auf jeder kreuzenden Lage.

**Schritt 3: Kreuzungsreduktion** Berechnung einer Anordnung für jede Lage, für welche die Anzahl der entstehenden Kreuzungen minimal ist

**Schritt 4: Knoten-/Kantenpositionierung (horizontale Koordinatenzuweisung)** Berechnung der  $x$ -Koordinaten der Knoten (& Dummy-Knoten), so dass keine Überlappungen entstehen. Einfügen der Kanten geradlinig und Entfernen der Dummy-Knoten. Wiederherstellen der ursprünglichen Kantenrichtungen

**Beispiel (Sugiyama Framework):**



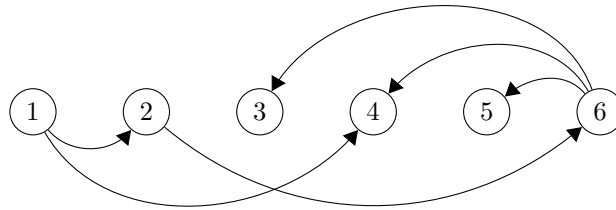
# 1 Entfernen von Kreisen

## maximal azyklischer Teilgraph

- ist  $\mathcal{NP}$ -schwer
- äquivalente Formulierungen:

**Minimum Feedback Arc Set** gesucht ist eine Teilmenge  $A_f \subseteq A$  mit  $|A_f|$  ist minimal, sodass  $D_f = (V, A \setminus A_f)$  azyklisch ist

**Lineare Anordnung** gesucht ist eine Anordnung der Knoten, sodass für  $\sigma : V \rightarrow \{1, \dots, |V|\}$  für jede Kante  $(u, v) \in A$   $\sigma(u) > \sigma(v)$  minimiert wird  
die oberen Kanten sollen minimiert werden



- Bezeichnungen:

$$\begin{aligned}\delta^+(v) &= \{(v, u) : (v, u) \in A\} \\ \delta^-(v) &= \{(u, v) : (u, v) \in A\} \\ \delta(v) &= \delta^-(v) \cup \delta^+(v) && \text{(Nachbarschaft von } v) \\ |\delta^+(v)| &= \deg^+(v) && \text{(Ausgangsgrad von } v) \\ |\delta^-(v)| &= \deg^-(v) && \text{(Eingangsgrad von } v)\end{aligned}$$

- ein naiver Ansatz zur Konstruktion eines kreisfreien Graphen ist der Greedy-Algorithmus (Vergleich Algorithmus 6. mit Laufzeit  $\mathcal{O}(n + m)$  mit  $|A'| \geq \frac{1}{2}|A|$ )
- bessere Lösung ist der verbesserter Greedy-Algorithmus (Vergleich Algorithmus 7. mit Laufzeit  $\mathcal{O}(n + m)$  (mit  $2n - 3$  Buckets für  $\deg^+ - \deg^-$  und der maximalen Anzahl an behaltenen Kanten ( $|A'| \geq \frac{1}{2}|A| + \frac{1}{6}|V|$ )))

## 2 Lagenzuordnung

Für einen *DAG* (gerichteter azyklischer Graph) soll eine zulässige Partition der Knotenmenge auf Lagen  $L_y$  gefunden werden (alle positiven ganzzahligen  $y$ -Koordinaten für alle  $v \in V$ ) mit  $y(u) < y(v), \forall (u, v) \in A$ .

### Bemerkung (Optimierungskriterium):

- kompaktes layering
  - Höhe = # Lagen  $h$
  - Breite =  $\max |L_i|, 0 \leq i \leq h$
- „richtig“: Kanten gehen nur über eine Lage, falls nicht werden Dummy-Knoten eingefügt
- Minimierung der Anzahl an Dummy-Knoten

**Minimierung der Höhe:** hierfür wird ein *longest-path-layering* verwendet:

weise  $v$  der Ebene  $L_p$  zu, falls  $p$  die Länge des längsten Pfades von einer Quelle ist (in Linearzeit für DAGs, Vergleich Algorithmus 8. )

**Minimierung der Höhe bei vorgegebener Breite:** ist  $\mathcal{NP}$ -schwer (auch mit Einheitsbearbeitungsdauern), äquivalent zu  $PRE - SCHED_B\{<\}$

## 2.1 Scheduling mit Vorgängerbedingung

Aus  $n$  Jobs  $J_1, \dots, J_n$  mit Bearbeitungsdauern  $p_1, \dots, p_n$  und der Bedingung  $J_i < J_k$  ( $J_i$  muss vor  $J_k$  abgeschlossen werden) mit  $B$  gleichen Maschinen, soll ein Schedule der Jobs auf den Maschinen gefunden werden, der die Vorgängerbedingungen erfüllt und minimale Bearbeitungsdauer hat.

Es gilt  $PRE - SCHED_2\{<\} \in \mathcal{P}$

Zu entscheiden, ob es zu  $n$  Jobs und  $T \in \mathbb{N}$  ein Scheduling auf  $B$  Maschinen gibt mit  $T_i \leq T, \forall 1 \leq i \leq n$  ist  $\mathcal{NP}$ -vollständig.

**Beweis:**

Es wird gezeigt, dass  $CLIQUE \propto PRE - SCHED_B\{< 3\}$

*CLIQUE* Gibt es für einen Graphen  $G = (V, E)$  und  $K \in \mathbb{N}$ ,  $K \leq |V|$  einen vollständigen Teilgraphen mit mindestens  $K$  Knoten?

Definitionen für das Bilden einer Instanz von *CLIQUE* für  $PRE - SCHED_B\{< T\}$  mit gegebenen  $G, K$ :

- $K' = |V| - K$
- $L = K \frac{K-1}{2}$
- $L' = |E| - L$
- $B = \max\{K, L + K', L'\} + 1$
- Dummy-Jobs mit  $X_j < Y_r < Z_s$ :
  - $X_j, j = 1, \dots, B - K$
  - $Y_r, r = 1, \dots, B - L - K'$
  - $Z_s, s = 1, \dots, B - L'$

Für jeden Knoten  $v$  wird ein Job  $J_v$  eingeführt, sowie für jede Kante  $e$  ein Job  $J_e$ .

$\Rightarrow$  Gesamtzahl der Jobs  $n = 3B$ .

$\Rightarrow PRE - SCHED_B\{< 3\}$  muss alle Slots füllen, bei 3 Maschinen gilt (falls es ein Clique der Größe  $K$  gibt):

- im ersten Zeitslot (von oben nach unten) stehen alle *CLIQUE*-Knoten, aufgefüllt mit Dummy-Jobs ( $K + (B - K) = B$ )
- im zweiten Zeitslot stehen (von oben nach unten) alle *CLIQUE*-Kanten, dann alle übrigen Knoten, aufgefüllt mit Dummy-Jobs ( $L + (B - L - K') + K' = B$ )
- im dritten Zeitslot stehen (von oben nach unten) alle übrigen Kanten, aufgefüllt mit Dummy-Jobs ( $L' + B - L' = B$ )

Wenn keine Clique der Größe  $K$  existiert, gilt:

- in jedem Schedule müssen  $B$  Jobs in der ersten Zeiteinheit ausgeführt werden
- $K$  davon müssen Knoten-Jobs sein

$\Rightarrow$  die existieren aber nicht

**Bemerkung:** mit der minimalen Anzahl der Dummy-Knoten kann es effizient durchgeführt werden, aber die Kombination von minimaler Höhe und minimaler Anzahl an Dummy Knoten ist auch in  $\mathcal{NP}$

Ist  $h_{OPT}$  minimale Höhe  $h$  eines Layerings mit Breite  $B$ , dann erfüllt die resultierende Höhe aus *LIST - SCHEDULING* (Jobs stehen in einer Liste, wenn eine Maschine frei ist, wird der erste Job aus der Liste abgearbeitet)

$$h \leq (2 - \frac{1}{B})h_{OPT}$$

$\Rightarrow$  „ $(2 - \frac{1}{B})$ “-Approximation



# Globale und lokale Optimierung

Zur Optimierung gibt es den Ansatz des „layer-by-layer sweep“ (Vergleich Algorithmus 9. )

**Beobachtung:** Die Anzahl an Kreuzungen ist nicht abhängig von den genauen  $x$ -Koordinaten, sondern von der Permutation (Ordnung) der Knoten. Es ist einfach Ordnungen zu spezifizieren für  $L_1, L_2$  durch Koordinaten-Vektoren  $X_1$  und  $X_2$ .

## 1 Kreuzungsminimierung

- betrachtet wird ein eingeschränktes Kreuzungsminimierungsproblem
- ist schon  $\mathcal{NP}$ -schwer

### 1.1 Zwei-Lagen-Kreuzungsminimierung

- Graph ist gegeben durch eine Partition von  $V$  auf zwei Lagen  $L_1, L_2$
- gesucht sind Anordnungen  $X_1, X_2$  von  $L_1, L_2$  sodass die Anzahl der Kantenpaare, die sich kreuzen möglichst gering ist
- $X_1$  sei fest, gesucht ist also nur noch die Anordnung  $X_2$  von  $L_1$ , dass die Anzahl der Kreuzungen minimal ist
- für  $X_1, X_2$  definieren wir
  - $cross(G, X_1, X_2) = \#((u, w), (v, z))$  mit  $X_1(u) < X_1(v) \wedge X_2(w) > X_2(z)$  oder umgekehrt (Anzahl der Kreuzungen für die Ordnungen  $X_1, X_2$ )
  - $opt(G, X_1) = \min_{X_2} \{cross(G, X_1, X_2)\}$
  - für  $u, v \in L_1$ ,  $X_1(u) < X_1(v)$  definieren wir die **Kreuzungszahl**  $c_{uv}$ :

$$c_{uv} = |\{(u, w), (v, z) \in E : X_2(z) < X_2(w)\}|$$

für  $u = v$  gilt  $c_{uv} = 0$ ,  $\forall u \in L_2$

- für einen 2-Layer-Graph gilt:
  1.  $cross(G, X_1, X_2) = \sum_{\substack{u, v \in L_2 \\ X_2(u) < X_2(v)}} c_{uv}$
  2.  $opt(G, X_1) \geq \sum_{\substack{u, v \in L_2 \\ u \neq v}} \min\{c_{uv}, c_{v, u}\}$
- 2. gilt, da für jede Ordnung (auch die optimale) entweder  $X_2(u) < X_2(v)$  oder andersherum gilt
- es gibt verschiedene **Heuristiken** zur Reduktion von Kreuzungen, zum Beispiel die *Greedy-switch-Heuristik* (Vergleich Algorithmus 10. )

## 2 Schwerpunktlayout

## 3 Spektrallayout

## 4 Spring Embedder

## 5 Multidimensional Scaling (MDS)

# ALGORITHMEN

---

## Algorithmus 1: Radiales Baumlayout

---

**Input:** Binärbaum  $T = (V, E)$  mit Wurzel  $R \in V$

**Data:** Anzahl  $n_v$  der Knoten in Teilbaum  $T(v), v \in V$

**Output:** Polarkoordinaten  $p_v = (d_v, \alpha_v), v \in V$

**Function** postorder(*vertex*  $v$ )

```
     $n_v \leftarrow 1$ ;  
    foreach Nachfolger  $w$  von  $v$  do  
        POSTORDER( $w$ );  
     $n_v \leftarrow n_v + n_w$ ;
```

**Function** preorder(*vertex*  $v$ , *double*  $t, \alpha_{min}, \alpha_{max}$ )

```
     $d_v \leftarrow t$ ;  
     $\alpha_v \leftarrow \frac{\alpha_{min} + \alpha_{max}}{2}$ ;  
    if  $t > 0$  then  
         $\alpha_{min} \leftarrow \max\{\alpha_{min}, \alpha_v - \arccos \frac{t}{t+1}\}$ ;  
         $\alpha_{max} \leftarrow \min\{\alpha_{max}, \alpha_v + \arccos \frac{t}{t+1}\}$ ;  
     $left \leftarrow \alpha_{min}$ ;  
    foreach Nachfolger  $w$  von  $v$  do  
         $right \leftarrow left + \frac{n_w}{n_v - 1} \cdot (\alpha_{max} - \alpha_{min})$ ;  
        PREORDER( $w, t + 1, left, right$ );  
     $left \leftarrow right$ ;
```

**begin**

```
    POSTORDER( $r$ );  
    PREORDER( $r, 0, 0, 2\pi$ );
```

---

---

**Algorithmus 2:** BICOMP

---

**Input:** ungerichteter Graph  $G = (V, E)$

**Data:** Zähler  $i$  für DFS-Nummerierung der Knoten

Stack  $S$  für nicht klassifizierte Kanten

Stack  $C$  der Repräsentanten auf aktuellem DFS-Weg

**Output:** DFS-Nummern (Knoten, Kanten), Blockrepräsentant *BICOMP* zu jeder Kante

**Function** *dfs*(*vertex*  $v$ )

```
     $i \leftarrow i + 1$ 
     $DFS[v] \leftarrow i$ 
    while  $\exists$  unnumerierte Kante  $e = \{v, w\}$  do
         $DFS[e] \leftarrow DFS[v]$ ;
        PUSH( $e, S$ );
        if  $w$  unnumeriert then
            PUSH( $e, C$ );
            DFS( $w$ );
            [Backtracking]
            if  $e = top(C)$  then
                repeat
                     $e' = e$ ;
                until  $e' \leftarrow POP(S)$ ;
                POP( $C$ );
            else
                while  $DFS[top(C)] > DFS[w]$  do
                    POP( $C$ );
    begin
         $i \leftarrow 0$ ;
        foreach  $s \in V$  do
            if  $s$  unnumeriert then
                DFS( $s$ );
```

---

---

**Algorithmus 3:** *st*-Ordnung

---

**Input:** ungerichteter Graph  $G = (V, E)$ , Kante  $\{s, t\} \in E$

**Data:** ausgehende Baumkanten *CHILDEDGE* für Knoten

Vorgängerknoten *PARENT* für Knoten

Pfad  $P$  (aktuelles Ohr)

abhängige Nichtbaumkanten  $D$  von Baumkanten

**Output:** Liste  $L$  der Knoten in Bicomp. von  $\{s, t\}$  (in der *st*-Ordnung)

**Function** *process\_ears*(Baumkante  $w \rightarrow x$ )

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

$u \leftarrow v$ ; **while**  $u \notin L$  **do**

$u \leftarrow \text{PARENT}[u]$  ;

      // Pfad zurückgehen, bis zu altem Ohr,

      // das schon in  $L(\text{orientiert})$  ist

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

**if**  $w \rightarrow x$  von  $w$  nach  $x$  (oder  $x$  nach  $w$ ) orientiert ist **then**

      orientiere  $P$  von  $w$  nach  $x$  (oder  $x$  nach  $w$ );

      füge inneren Knoten von  $P$  unmittelbar vor (oder hinter)  $u$  in  $L$  ein;

**foreach** Baumkante  $w' \rightarrow x'$  von  $P$  **do**

      PROCESS\_EARS( $w' \rightarrow x'$ )

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

**Function** *dfs*(vertex  $v$ )

$i \leftarrow i + 1$

$\text{DFS}[v] \leftarrow i$

**while**  $\exists$  unnummerierte Kante  $e = \{v, w\}$  **do**

$\text{DFS}[e] \leftarrow \text{DFS}[v]$ ;

**if**  $w$  unnummeriert **then**

$\text{CHILDEDGE}[v] \leftarrow e$ ;

$\text{PARENT}[w] \leftarrow v$ ;

      DFS( $w$ );

**else**

$\{w, x\} \leftarrow \text{CHILDEDGE}[w]$ ;

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

**if**  $x \in L$  **then**

        PROCESS\_EARS( $w \rightarrow x$ );

**begin**

  Initialisieren von  $L$  mit  $s \rightarrow t$ ;

$\text{DFS}[s] \leftarrow 1$ ;

$i \leftarrow 1$ ;

$\text{DFS}[\{s, t\}] \leftarrow 1$ ;

$\text{CHILDEDGE}[s] \leftarrow \{s, t\}$ ;

  DFS( $t$ );

---

---

**Algorithmus 4:** kanonische Ordnung

---

**Input:** triangulierter planarer Graph  $G = (V, E)$  mit  $C_0(G) = \{v_1, v_2, v_3\}$

**Data:**  $\text{mark}(v) = \text{true}$ , falls  $v$  zu  $\pi$  hinzugefügt wurde

$\text{out}(v) = \text{true}$ , falls  $v \in C_0(G_k)$

$\text{chords}(v) = \#$  der *chords* von  $C_0(G_k)$ , dessen Endknoten  $v$  ist

**Output:** kanonische Ordnung  $\pi = (v_1, v_2, \dots, v_n)$

**begin**

**forall the**  $v \in V$  **do**

$\text{chords}(v) \leftarrow 0$ ;

$\text{out}(v) \leftarrow \text{false}$ ;

$\text{mark}(v) \leftarrow \text{false}$ ;

$\text{out}(v_1), \text{out}(v_2), \text{out}(v_3) \leftarrow \text{true}$ ; **for**  $k = 3, \dots, 3$  **do**

    wähle  $v \neq v_1, v_2$  mit  $\text{mark}(v) = \text{false}$ ,  $\text{out}(v) = \text{false}$ ,  $\text{chords}(v) = 0$ ;

$v_k \leftarrow v$ ;

$\text{mark}(v) \leftarrow \text{true}$ ;

$(w_1 = v_1, w_2, \dots, w_{t-1}, w_t = v_2) \leftarrow C_0(G_{k-1})$ ;

$(w_p, \dots, w_q) \leftarrow$  unmarkierte Nachbarn von  $v_k$ ;

**for**  $i = p, \dots, q$  **do**

$\text{out}(w_i) \leftarrow \text{true}$ ;

      Update der chord-Zähler für  $w_i$  und alle seine Nachbarn;

---

**Algorithmus 5:** shift-Methode

---

**Input:** triangulierter planarer Graph  $G = (V, E)$ , kanonische Ordnung  $\pi = (v_1, \dots, v_n)$

**Data:** Liste  $L(v)$  mit allen Knoten, die mit dem Knoten  $v$  verschoben werden müssen

**Output:** planare, geradlinige Gitterzeichnung induziert durch die Koordinaten  $P(v) = (x(v), y(v))$  für alle  $v \in V$

**begin**

$P(v_1) = (0, 0)$ ;

$P(v_2) = (2, 0)$ ;

$P(v_3) = (1, 1)$ ;

$L(v_1) = \{v_i\}$  für  $i = 1, 2, 3$ ;

**for**  $k = 4, \dots, n$  **do**

$(w_1 = v_1, w_2, \dots, w_{t-1}, w_t = v_2) \leftarrow C_0(G_{k-1})$ ;

$(w_p, \dots, w_q) \leftarrow$  Nachbarn von  $v_k$  auf  $C_0(G_{k-1})$ ;

**forall the**  $v \in \bigcup_{i=p+1}^{q-1} L(w_i)$  **do**

$x(v) \leftarrow x(v) + 1$ ;

      // Verschiebung um 1 nach rechts

**forall the**  $v \in \bigcup_{i=q}^t L(w_i)$  **do**

$x(v) \leftarrow x(v) + 2$ ;

      // Verschiebung um 2 nach rechts

$P(v_k) \leftarrow \mu(P(w_p), P(w_q))$ ;      //  $\mu(P_1, P_2) = (\frac{1}{2}(x_1 - y_1 + x_2 + y_2), \frac{1}{2}(-x_1 + y_1 + x_2 + y_2))$

$L(v_k) \leftarrow \{v_k\} \cup \bigcup_{i=p+1}^{q-1} L(w_i)$ ;

---

**Algorithmus 6:** Greedy-Algorithmus

---

**Input:** gerichteter Graph  $D = (V, A)$

**Data:**  $A'$  neue Kantenmenge

**Output:** kreisfreier gerichteter Graph

**begin**

$A' \leftarrow \emptyset$ ;

**foreach**  $v \in V$  **do**

**if**  $\deg^+(v) \geq \deg^-(v)$  **then**

$A' \leftarrow A' \cup \delta^+(v)$ ;

**else**

$A' \leftarrow A' \cup \delta^-(v)$ ;

        löschen von  $v$  und  $\delta(v)$  aus  $D$ ;

---

**Algorithmus 7:** verbesserter Greedy-Algorithmus

---

**Input:** gerichteter Graph  $D = (V, A)$

**Data:**  $A'$  neue Kantenmenge

**Output:** kreisfreier gerichteter Graph

**begin**

**while**  $V \neq \emptyset$  **do**

**while**  $\exists \text{ source } v \in V$  **do**

$A' \leftarrow A' \cup \delta^+(v)$ ;

            löschen von  $v$  und  $\delta(v)$  aus  $D$ ;

**while**  $\exists \text{ target } v \in V$  **do**

$A' \leftarrow A' \cup \delta^-(v)$ ;

            löschen von  $v$  und  $\delta(v)$  aus  $D$ ;

        Löschen aller isolierten Knoten aus  $V$ ;

**if**  $V \neq \emptyset$  **then**

            Let  $v \in V$  mit  $\deg^+(v) - \deg^-(v)$  maximal;

$A' \leftarrow A' \cup \delta^+(v)$ ;

            löschen von  $v$  und  $\delta(v)$  aus  $D$ ;

---

**Algorithmus 8:** Longest Path Layering

---

**Input:** kreisfreier gerichteter Graph  $D = (V, A)$

**Output:** Longest path layering von  $D$

**Data:**  $V_{\text{source}}$ : die Menge aller aktuellen *source*-Knoten ( $d(v) = 0$ )

$d(v) = \deg^-(v)$  für alle  $v \in V$

$L_i$ : Liste aller Knoten auf dem Layer  $i$

**begin**

$i \leftarrow 0$ ;

**while**  $V \neq \emptyset$  **do**

**foreach**  $v \in V_{\text{source}}$  **do**

            Zuweisung von  $v$  zu  $L_i$ ;

**foreach**  $(v, w) \in A$  **do**

$d(w) \leftarrow d(w) - 1$ ;

            löschen von  $v$ ;

$i \leftarrow i + 1$ ;

---

**Algorithmus 9:** layer-by-layer sweep

---

**Input:** Lagen-Graph  $G = (L_0 \cup \dots \cup L_h, E)$

1. wähle eine zufällige Permutation für  $L_0$
  2. wiederhole für die angrenzenden Layer  $L_i, L_{i+1}$ :  
minimiere die Anzahl an Kreuzungen durch umsordieren von  $L_{i+1}$  ( $L_i$  bleibt fixiert)
  3. wiederhole Schritt 2 in umgekehrter Reihenfolge (von  $L_h$  nach  $L_0$ )
  4. wiederhole Schritt 2 und 3 bis die Anzahl an Kreuzungen sich nicht mehr verringert
  5. wiederhole Schritt 1-4 mit anderer Start-Permutation
- 

---

**Algorithmus 10:** Greedy switch heuristic

---

**Input:** zweischichtiger Graph (two Layer)  $G = (L_1 \cup L_2, E)$  mit Kreuzungszahlen  $c_{uv}$

**Output:**  $G$  mit weniger Kreuzungen, mit 0 Kreuzungen, falls  $G$  kreuzungsfrei

**begin**

**repeat**

**for**  $v = 1, \dots, |L_2| - 1$  **do**

**if**  $c_{v(v+1)} > c_{(v+1)v}$  **then**

                └ tausche die Positionen von  $v$  und  $v + 1$

**until**  $cross(G, X_1, X_2)$  sich nicht mehr verändert;

---