Kap. 3: Hierarchische Zeichenverfahren
3.5 Koordinatenzuweisung

Prof. Dr. Petra Mutzel

Lehrstuhl für
Algorithm Engineering LS11
Universität Dortmund

12.-14. VO WS07/08 20./26./27. November 2007

Überblick zu Kapitel 3

3.1 Einführung und Überblick ✓
3.2 Schichtzuweisung ✓
3.3 Zählen von Kreuzungen ✓
3.4 Kreuzungsminimierung ✓
3.5 Horizontale Koordinatenzuweisung

Literatur für diese VO

- E.R. Gansner, E. Koutsofios, S.C. North, K.-P. Vo: A technique for drawing directed graphs, IEEE Transactions on Software Engineering 19 (3), 1993, 214-230.

- Originalpaper für 3.5.4 (**bitte lesen!**): U. Brandes, B. Köpf: Fast and simple horizontal coordinate asssignment, GD 2001, LNCS 2265, 2002, 31-44.

Überblick Koordinatenzuweisung

3.5.1 Einführung

3.5.2 Knotenpositionierung mittels Prioritätswerten

3.5.3 Knotenpositionierung mittels LP

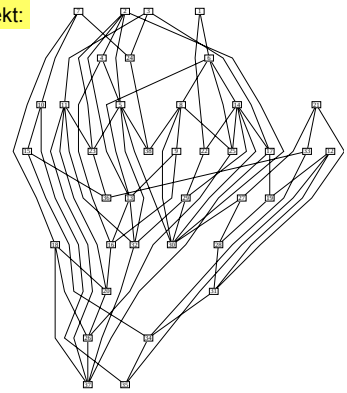
3.5.4 Knotenpositionierung von Brandes und Köpf

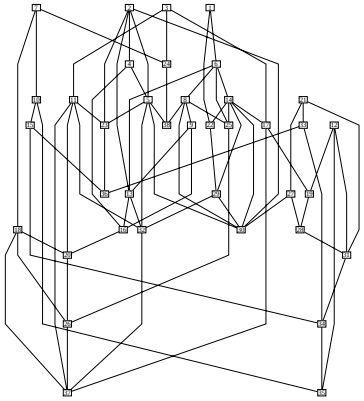
3.5 Horizontale Koordinatenzuweisung
3.5.1 Einführung

- **Situation:** Nach Layering (Phase 1) und Kreuzungsminimierung (Phase 2) steht die Topologie der Zeichnung fest.

- **Ziel:** Festlegung der x-Koordinaten, so dass der „Spaghetti-Effekt“ (lange verwinkelte Kanten) vermieden wird.

Spaghetti-Effekt:





3.5.2 Knotenpositionierung mittels Prioritätswerten

Idee: schichtenweise, d.h. die Positionen der Schicht i ist bereits fixiert, wir wollen die x-Koordinaten der Knoten der Schicht $i+1$ berechnen

1. Berechne für jeden Knoten einen Prioritätswert prio(w)
2. Platziere die Knoten in Prioritätsreihenfolge so nahe wie möglich an der „Wunschposition".

Knotenpositionierung mit Prioritätswerten

Dabei ist zu beachten:

1. Platzierungen werden nicht rückgängig gemacht
2. bereits platzierte Knoten dürfen nicht verschoben werden
3. nur höchstens ein Knoten per Position
4. Positionen sind ganzzahlig

Achtung: Für dazwischenliegende Knoten muss genug Platz bleiben, denn die Sortierung der Knoten auf der Schicht muss erhalten bleiben.

Beispiele für Prioritäten und Wunschpositionen

Median-Platzierung:

Sei $w(e) \in \mathbb{N}$: Wichtigkeit der Kante $e \in E$ (vorgegeben)

Knoten v besitze k Nachbarn $v_1,\ldots,v_k$ mit Koordinaten $x(v_i)$

1. Berechne prio(v) = $\sum_{i=1}^{k} w(v,v_i)$
2. Wunschposition(v) = Median($x(v_1),x(v_2),\ldots,x(v_k)$)

Vermeidung des Spaghetti-Effekts durch Ersetzung von $w(e)$ durch $\Omega(e)w(e)$ mit $\Omega(e)$ interner Parameter, der für lange Kanten sehr hoch ist.

Die Median-Platzierung minimiert den Wert $\sum_{i=1}^{k} |x(v)-x(v_i)|$

Achtung: theoretisch, denn: beachte

Knotenpositionierung mit Prioritätswerten

- Es gibt zahlreiche andere Heuristiken für Prioritäten und Wunschpositionen.
- Vor dem Festlegen der Wunschpositionen muss berücksichtigt werden, dass links und rechts von v genügend Platz für dazwischenliegende Knoten bleibt.

3.5.3 Knotenpositionierung mittels LP

$$min \sum_{e=(v,w)} c(e)|x(w) - x(v)|$$

$$x(b) - x(a) \geq 1 \quad \text{für alle } a,b \text{ mit } a \text{ ist level-order Vorgänger von } b$$

$$x(v) \geq 0 \quad \forall v \in V$$

$$x(v) \text{ ganzzahlig} \quad \forall v \in V$$

Definition der Gewichte $c(e)$

$$c(e) = \begin{cases} 1, & \text{falls } v,w \text{ „echte Knoten"} \\ 2, & \text{falls } v \text{ „echter" und } w \text{ „dummy Knoten"} \\ 8, & \text{falls } v,w \text{ „Dummyknoten"} \end{cases}$$

Knotenpositionierung mittels LP

- Das (LP) liefert immer ganzzahlige optimale Lösungen (weil die Matrix total unimodular ist)
- Das (LP) kann in ein (LP) ohne Absolutbeträge transferiert werden, das immer noch total unimodular ist.
- Das Simplexverfahren liefert also ganzzahlige Optimallösungen.

- Das (LP) tendiert zur Vermeidung des Spaghetti Effekts, kann es aber nicht garantieren.

Knotenpositionierung mittels LP

$$\begin{aligned} min \quad & \sum_{e=(v,w)} c(e)y(v,w) & \\ & y(v,w) \geq x(w) - x(v) & \forall (v,w) \in E \\ & y(v,w) \geq x(v) - x(w) & \forall (v,w) \in E \\ & x(b) - x(a) \geq 1 & \text{für alle a,b mit a ist level-order Vorgänger von b} \\ & x(v) \geq 0 & \end{aligned}$$

Dieses (LP) hat keine Beträge mehr, dafür $|E|$ zusätzliche Variablen und $2|E|$ zusätzliche Restriktionen

3.5.4 Knotenpositionierung mittels Alignierung

Verfahren von Brandes und Köpf
basierend auf Buchheim, Leipert und Jünger

Ziel: Berechne reelle Werte $x(v)$ für alle Knoten (Original und Dummyknoten), so dass die Mindestabstände δ eingehalten werden, d.h. $x(u)+\delta \leq x(v)$ für alle Paare u vor v mit u ist level-order Vorgänger von v.

Eigenschaften:

(1) Knotenpermutationen der Schichten werden respektiert.

(2) Kantensegmente sind geradlinig gezeichnet.

(3) Innere Kantensegmente werden senkrecht gezeichnet ⇒ jede Kante hat höchstens zwei Knicke

(4) Kantenlängen $|x(v)-x(v_i)|$ sollen klein sein

Knotenpositionierung mittels Alignierung

Problem: Falls sich zwei lange Kanten in den inneren Segmenten schneiden, dann sind (1) und (3) nicht gleichzeitig einzuhalten.

Lösungsmöglichkeiten:

- Ignoriere (3): ein inneres Kantensegment wird nicht senkrecht gezeichnet
- Ignoriere (1): vertausche jeweils die beiden Knoten um die Kreuzung nach auf ein äußeres Segment zu legen
- Vermeide dieses Situation bereits in der Kreuzungsreduktionsphase (z.B. Barycenter, 2-opt)

Knotenpositionierung mittels Alignierung

Idee:

- Vertikale Alignierung (**unterhalb** / **links**): Versuche die Knoten jeweils **unterhalb** des Medians zu platzieren, löse Konflikte mittels „**links** geht vor“
- Horizontale Kompaktierung: Berechne die Koordinaten, so dass die alignierten Knoten die gleichen Koordinaten erhalten. Dabei werden Knoten jeweils so nah wie möglich am Vorgängerknoten (also **links**) platziert.
- Wiederhole 1. und 2. für jede der Kombinationen unterhalb / oberhalb sowie links / rechts.
- Kombination der vier Alignierungen zu einer Alignierung.

Vertikale Alignierung

- Vertikale Alignierung (**unterhalb** / **links**): Versuche die Knoten jeweils **unterhalb** des Medians zu platzieren, löse Konflikte mittels „**links** geht vor“.

Algorithmus

- Wandere schichtenweise von oben nach unten $i=1,\ldots,L$
- Traversiere alle Knoten v auf L_i von links nach rechts
- Falls möglich (also kein Konflikt), dann aligniere v mit seinem Median auf L_{i-1} (falls $N(v)$ gerade, dann teste zuerst **linken**, danach den rechten Median)

Konfliktbehandlung: Markiere in Preprocessing Kanten, die Konflikte generieren

Konfliktbehandlung

- **Typ-2 Konflikt:** zwei sich kreuzende innere Segmente
- **Problem:** nicht beide innere Segemente können dann senkrecht platziert werden.
- **Lösung:** Markiere eines von beiden als „nicht vertikal“ und ignoriere dieses beim alignieren
- Alternativ: s. o.

- **Typ-1 Konflikt:** Kreuzung zwischen äußerem und innerem Segment
- **Lösung:** Markiere das äußere Segment als „wird nicht aligniert“ (innere Segmente haben Vorrang).
- Dazu: s. Preprocessing Algorithmus

Konfliktbehandlung

- **Typ-0 Konflikt:** zwei sich kreuzende oder berührende äußere Segmente
- **Problem:** können nicht beide mit Median platziert werden
- **Lösung:** Greedy mittels „links-geht-vor“, dabei ist (u,v) links von (u',v'), wenn v links von v' oder $v=v'$ und u links von u' ist.

Preprocessing: Typ-1 Konflikterkennung

Idee: Markiere die Verlierer-Kanten bei Typ-1 Konflikten

Prämisse: es existieren keine Typ-2 Konflikte

- Durchlaufe alle Schichten von oben nach unten
- Traversiere Schicht $i+1$ von links nach rechts
- Merke dabei jeweils die zwei zuletzt gesehenen inneren Kantensegmente: diese starten bei $l0$ und $l1$ und haben Nachbarn $k0$ und $k1$ in Schicht i
- Nach der Entdeckung eines neuen inneren Segments: Teste alle Segmente, die links von $l1$ und rechts von $l0$ beginnen auf Typ-1 Konflikt: das ist genau dann der Fall, wenn der Nachbar links von $k0$ oder rechts von $k1$ liegt.
- Sonderfall: die Segmente, die rechts von der letzten inneren Kante liegen müssen auch getestet werden

Preprocessing: Typ-1 Konflikterkennung

1. **For** $i=2,\ldots,L-2$ **{**
2. $k_0=0$; $l=1$
3. **For** $l1=1,\ldots,|L_{i+1}|$ {
4. **If** $(l1=|L_{i+1}|)$ oder $v_{l1}^{(i+1)}$ inzident zu innerem Segment {
5. $k1 = |L_i|$
6. **If** $v_{l1}^{(i+1)}$ inzident zu innerem Segment {
7. $k1$ = pos[oberer Nachbar von $v_{l1}^{(i+1)}$]
8. **While** $l \le l1$ { // l ist rechts von vorherigem inneren Seg.
9. **For** jeden oberen Nachbarn $v_k^{(i)}$ von $v_l^{(i+1)}$ {
10. **If** $k<k0$ oder $k>k1$ { Markiere Segment $(v_k^{(i)}, v_l^{(i+1)})$ }
11. $l = l+1$
12. }
13. $k0 = k1$ } **} }**

Preprocessing: Typ-1 Konflikterkennung

- Der Preprocessing Algorithmus kann leicht modifiziert werden, so dass er auch Typ-2 Konflikte erkennt und
- entweder Permutationen vertauscht
- oder ein involviertes inneres Segment auch markiert.
- Achtung: $k0$ liegt in diesem Fall nicht mehr automatisch links von $k1$

Algorithmus Vertikale Alignierung

Definitionen:

- Eine maximale Menge vertikal alignierter Knoten heißt Block.
- Die Wurzel eines Blockes ist der oberste Knoten.

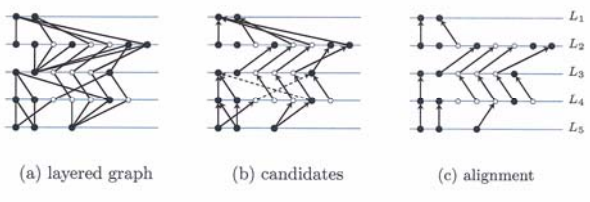
Datenstruktur:

- Ein Block wird durch eine zyklische Liste repräsentiert.
- Jeder Knoten zeigt auf seinen alignierten Nachbarn in der Schicht unterhalb: Knotenarray: align[v]
- Der unterste Knoten zeigt auf den obersten Knoten.
- Jeder Knoten zeigt auf die Wurzel des Blockes: root[v]

Algorithmus Vertikale Alignierung

1. Init: root[v]=v und align[v]=v für alle v
2. **For** jede Schicht i=1,…,L {
3. r=0 // pos des letzten alignierten auf oberer Schicht
4. **For** k=1,…,$|L_i|$ {
5. **If** $v_k^{(i)}$ hat obere Nachbarn {
6. Berechne m=Median (oder beide Mediane)
7. **For** m (oder linker und rechter Median m) {
8. **If** align[$v_k^{(i)}$]= $v_k^{(i)}$ {
9. **If** (u_m, $v_k^{(i)}$) nicht markiert und r<pos[u_m] {
10. align[u_m] = $v_k^{(i)}$
11. root[$v_k^{(i)}$] = root[u_m]
12. align[$v_k^{(i)}$] =root[$v_k^{(i)}$]
13. r = pos[u_m]
14. }}}}}}

Vertikale Alignierung



äußere Segmente involviert in Typ-1 Konflikte sind gestrichelt gezeichnet

Bilder aus: Brandes und Köpf (s. Literatur)

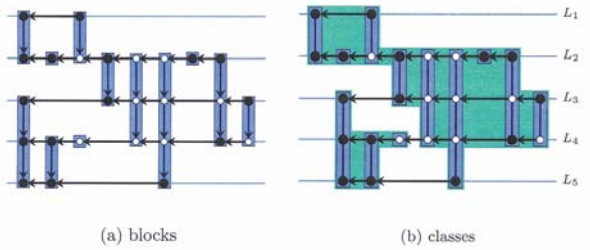
Horizontale Kompaktierung

Gesucht ist eine möglichst kompakte Repräsentation, so dass alle Knoten innerhalb eines Blockes die gleiche Koordinate erhalten (die der Wurzel).

Definition:

- Wir erhalten den Block-Graphen indem wir horizontale gerichtete Kanten von jedem Knoten zu seinem Vorgänger (auf der gleichen Schicht) einführen und die Blöcke zu einem Knoten kontrahieren.

Block-Graph und Klasseneinteilung



Bilder aus: Brandes und Köpf (s. Literatur)

Horizontale Kompaktierung

Eigenschaften des Block-Graphen G_B

- Der Block-Graph ist ein DAG.
- Betrachte die Senken s dieses Block-Graphen:
  - Die dazugehörige Wurzel ist jeweils der erste Knoten auf ihrer Schicht (also ganz links)
- Es gibt höchstens eine Senke pro Schicht.

Definitionen

- Der Block-Graph wird in Klassen eingeteilt:
- Ein Block gehört zu derjenigen Klasse, die die höchste Senke (höchste Schicht) enthält, die von ihm aus im Blockgraphen erreichbar ist.

Horizontale Kompaktierung

Idee des Algorithmus: Kombination von

- Kompaktierung der Knoten mittels „Longest Path Layering“ innerhalb jeder Klasse und
- Packen der verschiedenen Klassen möglichst nahe aneinander

Vorgehen

- Berechne zuerst die relativen Koordinaten innerhalb der Klassen relativ zur Senke dieser Klasse
- Dabei wird die minimale Separierung zwischen den Wurzeln der verschiedenen Klassen gleich mitberechnet (shift)
- Berechnung der absoluten Koordinaten

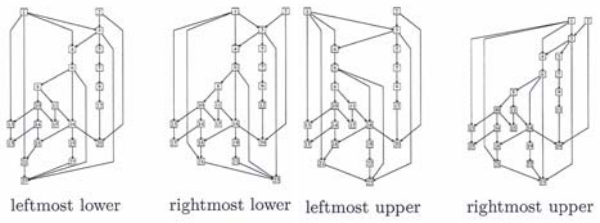
Algorithmus Horizontale Kompaktierung

1. Init: sink[v]=v und x[v]=shift[v]=„big M“ für alle v
2. // Berechne Wurzel Koordinaten relativ zur Senke
3. **For** alle Knoten v {
4. **If** root[v]=v und x[v]=„big M“ { place_block(v) }
5. }
6. // Berechne absolute Koordinaten
7. **For** alle v {
8. x[v] = x[root[v]]
9. **If** shift[sink[root[v]]] < „big M“ {
10. x[v] = x[v] + shift[sink[root[v]]]
11. }
12. }
13. Achtung: nicht explizite Berechnung des Block-Graphen

Funktion place-block(v)

1. Setze x[v]=0 und w=v
2. **Wiederhole**
3. **If** pos[w]>1 {
4. u=root[pred[w]]
5. if x[u]=„big M“ { place_block(u) }
6. **If** sink[v] = v { sink[v]=sink[u] }
7. **If** sink[v] ≠ sink[u] //Mind.abstand zwischen Blöcken
8. shift[sink[u]] = min{shift[sink[u]],x[v]-x[u]-δ}
9. **Else** x[v] = max{x[v],x[u]+δ} //Longest Path innerhalb
1. }
10. w = align[w]
11. **solange** bis w=v

Beispiel: vier Alignierungen



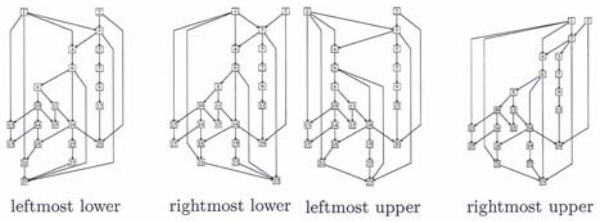
Kombination der Alignierungen

Bilder aus: Brandes und Köpf (s. Literatur)

Kombination der vier Alignierungen

- Für jeden Knoten gibt es nun vier verschiedene x-Koordinaten
- In der Balancierungs-Phase wird die Koordinate für jeden Knoten als der Durchnitts-Median aus diesen vier Werten festgesetzt, d.h. Durchschnitt aus dem zweit- und dritt-kleinstem Wert.
- Man kann zeigen: dies erhält die Permutationen und die Separationsbedingungen.

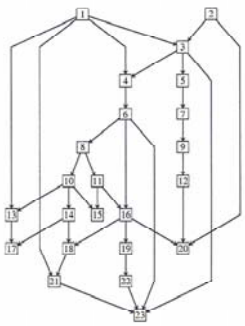
Beispiel: vier Alignierungen



Kombination der Alignierungen

Bilder aus: Brandes und Köpf (s. Literatur)

Ergebnis der Balancierung



Bilder aus: Brandes und Köpf (s. Literatur)

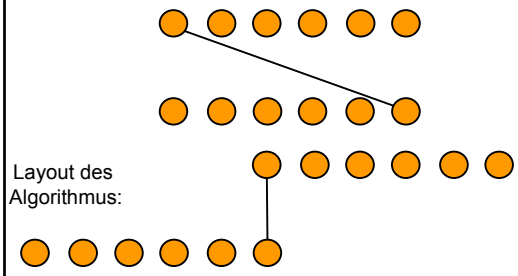
Knotenpositionierung mittels Alignierung

- Sei N die Summe der Anzahl an Originalknoten, Dummyknoten und Kantensegmenten.
- Die Algorithmen Vertikale Alignierung mit Preprocessing, Horizontale Kompaktierung und Balancierung berechnen eine gültige Koordinatenbelegung in linearer Zeit $O(N)$.

- Achtung: Die Anzahl der Dummyknoten und somit der Kantensegmente kann $\Omega(|V||E|)$ betragen. N kann also quadratisch wachsen.

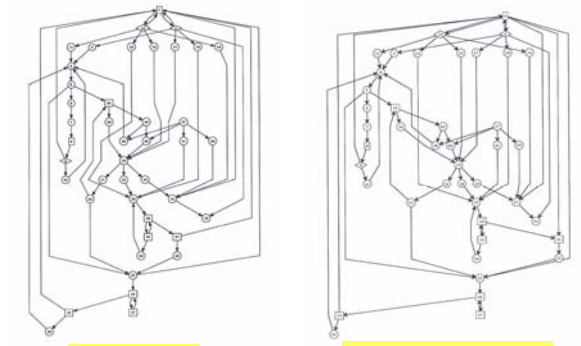
- Falls die minimale Separationsdistanz δ gerade ist, dann sind die berechneten Koordinaten ganzzahlig.

Pathologisches Beispiel



Das Beispiel zeigt, dass die Alignierungs-Bedingung großen negativen Einfluss auf die Weite der Zeichnung besitzen kann.

Realistisches Benchmark Beispiel



da Vinci-Tool

Alignierungs-Methode

Bilder aus: Brandes und Köpf (s. Literatur)

Knotenpositionierung mittels Alignierung

- Das Verfahren enthält viele generische Elemente, bei denen viele Variationen denkbar sind. Beispiele:
  - Alternative Alignierungsregeln (z.B. zum Vermeiden pathologischer Beispiele)
  - Alternative Konfliktverarbeitung
  - Minimum Cost Flow Kompaktierung (s. später bei planaren Verfahren) statt Longest Path
  - Alternative Kombination

ENDE Knotenpositionierung

Alternative Hierarchische Verfahren

- Alternativ zu dem Verfahren von Sugiyama et al. gibt es auch Verfahren, die Planarisierungsverfahren für ungerichtete Graphen auf DAGs übertragen:
  - Verfahren mittels „Aufwärtsplanarisierung“: Problem: Aufwärtsplanaritätstest ist NP-schwer, aber bei einer Quelle (oder Senke) in polynomieller Zeit möglich
  - Verfahren mittels „Schichtenplanarisierung“: hier sind die Schichten bereits festgelegt. Test ist in polynomieller Zeit möglich. Problem: Schichteneinteilung kann schlecht sein.

aktuelle Forschung an LS11

Mögliche Diplomarbeitsthemen zu Hierarchischen Verfahren

- Alternatives Verfahren für Layering mit Weitenbeschränkung (Entwurf, Analyse, Implementierung und Evaluierung)
- Ganz andere Kreuzungsminimierung (evtl. Knoten-layering verändern, noch besser: Suche nach Patterns! und Layering, die diese vermeiden)
- Alternative Knotenpositionierungsverfahren
- „schneller Sugiyama für große Graphen“
- Viele offene Fragen zur Aufwärtsplanarisierung (s. später in VO)

ENDE Hierarchische Verfahren