Automatisches Zeichnen von Graphen

Wintersemester 2000/2001
Dozent:
Prof. Dr. M. Jünger
mit
Dipl.-Math. C. Buchheim
Dipl.-Inform. C. Gutwenger
Dr. S. Leipert

Constantin Hellweg hellweg@informatik.uni-koeln.de
Ramin Sahamie sahamie@informatik.uni-koeln.de

Inhaltsverzeichnis

1 Einführung **7**

1.1 Einführung 7

1.1.1 Einige Grundbegriffe 7

1.1.2 Datenstrukturen für Graphen und Digraphen 9

2 Zeichnen von Bäumen **11**

2.1 Festlegungen 11

2.1.1 Baum 11

2.1.2 Wurzelbaum 11

2.1.3 Binärer Wurzelbaum 11

2.1.4 Generalkonvention 12

2.1.5 Traversieren von Binärbäumen 12

2.2 Zeichnen von Binärbäumen 13

2.2.1 1. Versuch 13

2.2.2 2. Versuch (Wetherell & Shannon) (1979) [WS79] 13

2.2.3 Elegante Lösung: Reingold & Tilford , (1981) [RT81] 17

2.2.4 Komplexität der Breitenminimierung 20

2.3 Zeichnen von allgemeinen Bäumen 27

2.3.1 Idee (nach Walker (1990) [Wal90]) 27

2.3.2 Skizze des Algorithmus 27

2.3.3 Konzepte für einen Linearzeit Algorithmus 28

3 Kräftebasierte Verfahren **33**

3.1 Modell 33

3.1.1 Grundidee 33

3.1.2 Einige Beispielzeichnungen 34

3.1.3 Einfacher Algorithmus 37

3.2 Älteste Variante (Tutte (1960,1963) [Tut60] [Tut63]) 37

3.2.1 Beispiel 38

3.3 Einige Definitionen 41

3.4 Simulation graphentheoretischer Distanzen durch Kräfte 42

3.4.1 Lösungsmethode von Kamada & Kawai [KK89] 43

3.4.2 Magnetische Felder 43

3.5 Allgemeine Energie-Funktionen 44

3.5.1 Nebenbedingungen 45

4 Hierarchische Verfahren **47**

4.1 Grundidee 47
4.2 Phase 1: Schichtzuweisung 49
4.2.1 Längster-Pfad-Schichtung 49
4.2.2 Algorithmus TOPSORT 49
4.2.3 Schichtung mit Breitenminimierung 50
4.2.4 Beweisskizze 50
4.2.5 Coffman-Graham-Schichtung 51
4.2.6 Minimierung der Anzahl künstlicher Knoten 52
4.3 Phase 2: Kreuzungsminimierung 53
4.3.1 Komplexität 53
4.3.2 BIPARTITE MULTIGRAPH KREUZUNGSZAHL (BMKZ) 53
4.3.3 2-Schichten-Kreuzungsminimierung mit einer fixierten Schicht (2SKM1F) 56
4.3.4 Zählen der Kreuzungen in einer gegebenen 2-Schichten-Zeichnung 59
4.3.5 Algorithmus von W. Barth (2000) [Bar00] 60
4.3.6 Verschnellerung durch bessere Datenstruktur 63
4.3.7 Heuristiken, die die Matrix C benutzen 65
4.3.8 Effiziente Heuristiken ohne Berechnung von C 67
4.3.9 Exakte Kreuzungsminimierung 71
4.3.10 Experimentelle Studien aus Jünger & Mutzel (1997) [JM97] . . . 73
4.4 Phase 3: Horizontale Koordinatenzuweisung 76

5 AGD **89**

5.1 Die AGD-Bibliothek zum Zeichnen von Graphen 89

6 Planare Graphen **99**

6.1 $K_{3,3}$ und K_5 99
6.1.1 Rätsel 1 (Dudeney (1917) [Dud17]) 99
6.1.2 Rätsel 2 (Möbius (1840, siehe [BLW77]) 99
6.1.3 Übersetzungen 99
6.2 Satz von Kuratowski (1930) [Kur30] 102
6.3 Einbettungsbegriffe (für planare Graphen auf der Ebene) 102
6.4 Duale Graphen 105
6.5 Euler-Formel 106
6.5.1 Anwendung der Eulerformel 108
6.5.2 Anwendung: Reguläre Polyeder 109

7 Planaritätstest **111**
7.1 Einführung . . . 111
7.1.1 Das Divide und Conquer Konzept . . . 111
7.1.2 Das inkrementelle Konzept . . . 112
7.2 Der Planaritätstest auf der Grundlage der PQ-Bäume . . . 113
7.3 PQ-Bäume . . . 120
7.3.1 PQ-Baum Schablonen . . . 127
7.4 Effiziente Implementierung der PQ-Bäume . . . 137
7.5 Planaritätstest mit PQ-Bäumen . . . 151
7.6 Einbettungsalgorithmus . . . 154

8 Zeichenverfahren für planare Graphen **159**
8.1 Geradliniges Zeichnen . . . 159
8.2 Ein Algorithmus zum geradlinigem Zeichnen . . . 164
8.2.1 Herstellung einer Triangulierung . . . 164
8.2.2 Bestimmung einer kanonischen Numerierung . . . 166
8.2.3 Koordinatenbestimmung . . . 169
8.3 Verbesserungen und weitere Entwicklungen . . . 172
8.4 Orthogonales Zeichnen: Knickminimierung für 4-planare Graphen . . . 175
8.5 Erweiterungen für beliebige Knotengrade . . . 185
8.6 Planarisierungsmethode . . . 186

Literaturverzeichnis **191**

Kapitel 1

Einführung

1.1 Einführung

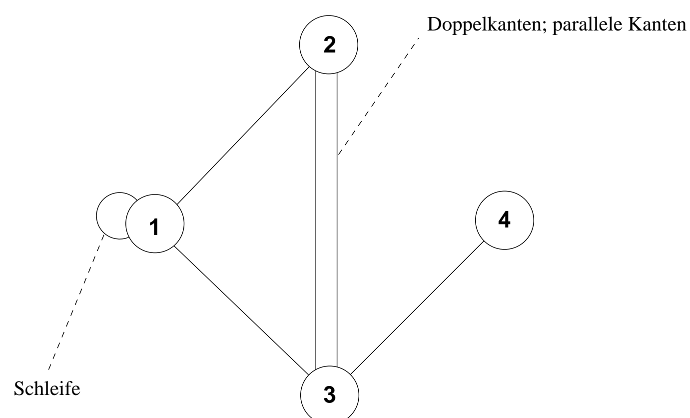
1.1.1 Einige Grundbegriffe

(Ungerichteter) Graph $G = (V, E)$

- hat endliche Menge V von Knoten (nodes, vertices)
- endliche Multimenge E von Kanten (edges), wobei die $e \in E$ ungerichtete Paare von Knoten sind.

Z.B.:

$$\begin{aligned} V &= \{1, ..., 4\} \\ E &= \{(1,1), (1,2), (1,3), (2,3), (2,3), (3,4)\} \end{aligned}$$



Einfacher (simpler) Graph

- keine Schleifen und parallelen Kanten:

$$E \subseteq \{\{v, w\} \mid v, w \in V,\ v \neq w\}$$

- ab jetzt: immer einfach

Für $e = (v, w) \in E$ sind v und w die **Endknoten** von e. Die Kante e ist **inzident** zu v und w, v und w sind **adjazent**.

Für $v \in V$ ist $\Delta(v) = |\{(v, w) \mid (v, w) \in E\}|$ der **Grad** von v. Die zu v adjazenten Knoten sind **Nachbarn** von v.

Gerichteter Graph (Digraph) $G = (V, E)$

Wie ein ungerichteter Graph, aber die Kanten sind geordnete Paare von Knoten der Form $e = (v, w)$. e ist ausgehende Kante von v bzw. eingehende Kante von w.

- Rein-Grad $\Delta^+(v) = |\{(u, v) \mid (u, v) \in E\}|$
- Raus-Grad $\Delta^-(v) = |\{(v, u) \mid (v, u) \in E\}|$
- v ist **Quelle** $\Leftrightarrow \Delta^+(v) = 0$
- v ist **Senke** $\Leftrightarrow \Delta^-(v) = 0$

Zugrunde liegender ungerichteter Graph

Vergessen der Richtung und danach evtl. Simplifizierung (d.h. Entfernung von Doppelkanten).

(Gerichteter) Pfad

Folge von distinkten Knoten $(v_1, \ldots, v_k)$ mit folgender Eigenschaft:

$$(v_i, v_{i+1}) \in E \text{ für } 1 \leq i < k$$

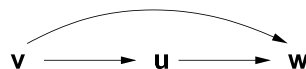
Genauer: (v_1, v_k)-Pfad

Ein (gerichteter) Pfad ist ein **(gerichteter) Kreis**, falls auch $(v_k, v_1) \in E$. Ein Digraph ist **azyklisch**, wenn er keinen gerichteten Kreis hat. Ein Graph $G = (V, E)$ ist **zusammenhängend**, wenn $\forall\, v, w \in V$ ein (v, w)-Pfad existiert.

Ein Digraph ist:

- zusammenhängend, wenn der zugrundeliegende ungerichtete Graph zusammenhängend ist.
- stark zusammenhängend, wenn $\forall\, v, w \in V$ ein gerichteter (v, w)-Pfad existiert.

Eine Kante $(v, w) \in E$ in einem Digraphen $G = (V, E)$ ist **transitiv**, falls in G ein gerichteter (v, w)-Pfad existiert, der (v, w) nicht benutzt.



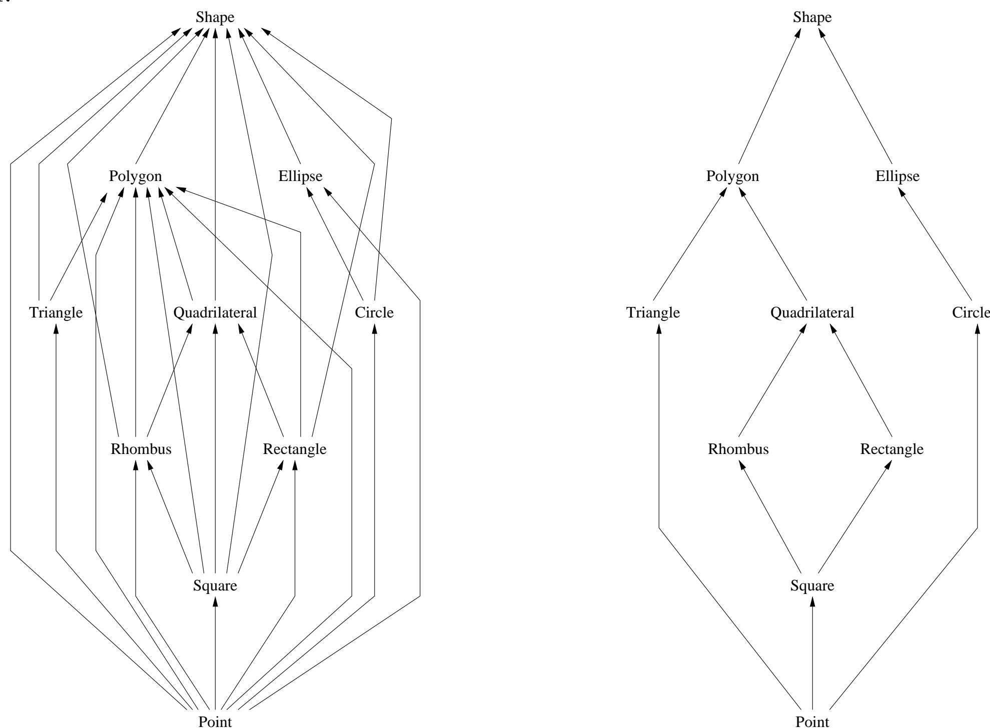
Transitive Hülle

Die transitive Hülle eines Digraphen entsteht durch Hinzunahme aller transitiven Kanten.

Transitive Reduktion

Die transitive Reduktion eines Digraphen entsteht durch Entfernen aller transitiven Kanten.

Beispiel:



Ein Graph $G' = (V', E')$ ist **Subgraph** eines Graphen $G = (V, E)$, falls $V' \subseteq V$ und $E' \subseteq E$.

Knoteninduzierter Subgraph

$V' \subseteq V$ und $E' = \{(v, w) \in E \mid v, w \in V'\}$

1.1.2 Datenstrukturen für Graphen und Digraphen

Adjazenzmatrix

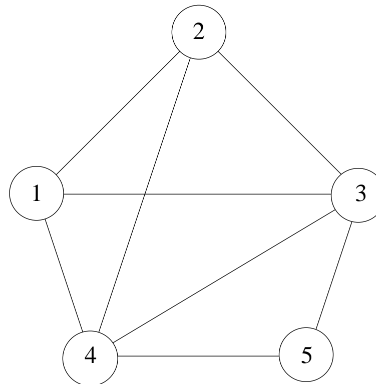
$A = (a_{ij}) \in \{0, 1\}^{n \times n}$ mit $a_{ij} = 1 \Leftrightarrow (i, j) \in E$

Adjazenzlisten

Liste L_v mit $v \in V$ von (ausgehenden) inzidenten Kanten.

Beipiel 1.1.1

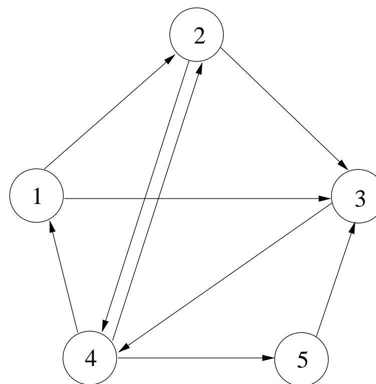
1. ungerichtet



	1	2	3	4	5
1		1	1	1	
2	1		1	1	
3	1	1		1	1
4	1	1	1		1
5			1	1	

$L_1:$ $(1,2),(1,3),(1,4)$
$L_2:$ $(2,1),(2,3),(2,4)$
$L_3:$ $(3,1),(3,2),(3,4),(3,5)$
$L_4:$ $(4,1),(4,2),(4,3),(4,5)$
$L_5:$ $(5,3),(5,4)$

2. gerichtet



	1	2	3	4	5
1		1	1		
2			1	1	
3				1	
4	1	1			1
5			1		

$L_1:$ $(1,2),(1,3)$
$L_2:$ $(2,3),(2,4)$
$L_3:$ $(3,4)$
$L_4:$ $(4,1),(4,2),(4,5)$
$L_5:$ $(5,3)$

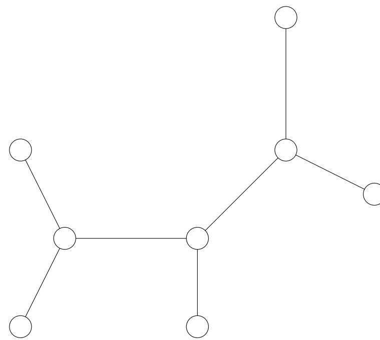
Kapitel 2

Zeichnen von Bäumen

2.1 Festlegungen

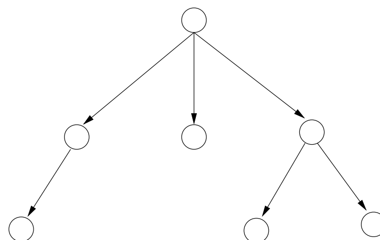
2.1.1 Baum

Zusammenhängender, kreisfreier Graph



2.1.2 Wurzelbaum

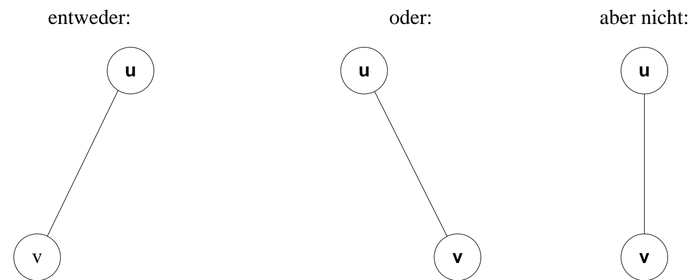
Gerichteter Baum mit einer Wurzel $v \in V$ und für alle $w \in V$, $w \neq v$ existiert ein gerichteter (v, w)-Pfad.



Wir nehmen zusätzlich an: die Reihenfolge der ausgehenden Kanten jedes Knotens ist Teil der Eingabe (fixiert).

2.1.3 Binärer Wurzelbaum

Spezielle Anforderung hier: ein Kind ist linkes oder rechtes Kind



Tiefe von $v \in V$ (Wurzel r):

$$\text{Tiefe}(v) := \#\text{Kanten auf (eindeutigem) } (r, v)\text{-Pfad}$$

Höhe des Wurzelbaums $T = (V, E)$

$$\text{Höhe}(T) := max_{v \in V} \ (\text{Tiefe}(v))$$

2.1.4 Generalkonvention

Ein Knoten v erhält die y-Koordinate $-\text{Tiefe}(v)$

Problem: Bestimmung der x-Koordinate, so daß die Zeichnung „schön“ wird.

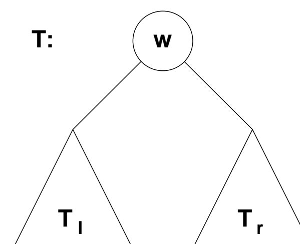
Zunächst Beschränkung auf binäre Wurzelbäume:

2.1.5 Traversieren von Binärbäumen

PREORDER(T)

1. bearbeite Wurzel w
2. IF $T_l \neq \varnothing$ PREORDER(T_l)
3. IF $T_r \neq \varnothing$ PREORDER(T_r)

INORDER(T)



1. IF $T_l \neq \varnothing$ INORDER(T_l)
2. bearbeite Wurzel w
3. IF $T_r \neq \varnothing$ INORDER(T_r)

POSTORDER(T)

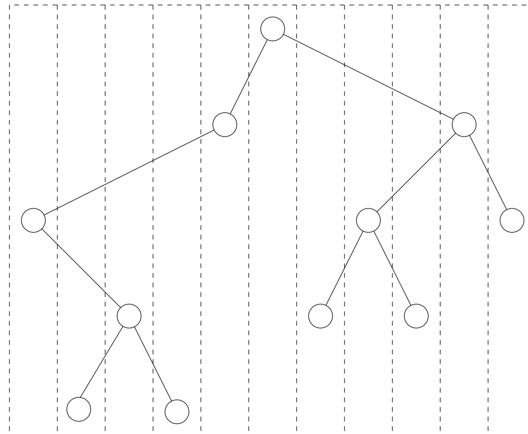
1. IF $T_l \neq \varnothing$ POSTORDER(T_l)
2. IF $T_r \neq \varnothing$ POSTORDER(T_r)
3. bearbeite Wurzel w

2.2 Zeichnen von Binärbäumen

2.2.1 1. Versuch

Vergabe von x-Koordinate in INORDER-Reihenfolge

INORDER-Reihenfolge	v_1	v_2	...	v_n
Koordinate	1	2	...	n



Systematischer: (Ästhetikkriterien)

(A1) Alle Knoten derselben Tiefe liegen auf einer Geraden. Die Geraden werden parallel gezeichnet.

(A2) Ein linkes Kind wird links, ein rechtes Kind wird rechts vom Vater plaziert.

(A3) Hat ein Vater 2 Kinder, so wird er über diesen zentriert.

Optimierungsziel: Minimierung der Breite.

2.2.2 2. Versuch (Wetherell & Shannon) (1979) [WS79]

Idee: 2 Phasen

1. Traversiere T in POSTORDER. Bearbeite v:

   (a) Ist v ein Blatt:

   Plaziere v auf dem nächsten freien Platz auf seiner Tiefengeraden.

   (b) Hat v nur ein linkes Kind:

   Plaziere v eine Position weiter rechts als das Kind.

   (c) Hat v nur ein rechtes Kind:

   Plaziere v eine Position weiter links als das Kind.

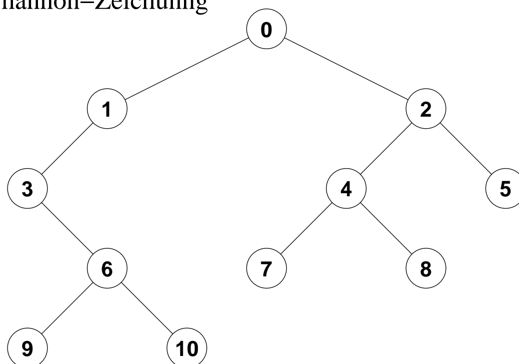
(d) Hat v zwei Kinder:

Zentriere v über den Kindern.

Wird bei (b) - (d) eine Position errechnet, die links von der nächsten freien liegt, so verschiebe hinreichend und merke die Verschiebung des Teilbaums darunter.

2. Traversiere T in PREORDER. Führe alle noch notwendigen Verschiebungen durch.

Für unser Beispiel liefert das die optimale Zeichnung:



Details:

```
#include <stdio.h>

#define TRUE 1
#define FALSE 0

int n, height;
int *xcoord, *ycoord, *shift, *leftchild;
int *rightchild, *nextpos, *modifier;

void assycoor (int node, int level)
{
  if (node > -1)
    {
      ycoord[node] = level;
      if (level > height)
height = level;
      assycoor (leftchild[node], level + 1);
      assycoor (rightchild[node], level + 1);
    }
}

void asspxcoor (int node)
{
  int leftchildexists, rightchildexists, hasonlyleftchild,
      hasonlyrightchild, isleaf, diff, place, h;
  leftchildexists = FALSE;
  if (leftchild[node] > -1)
    {
      leftchildexists = TRUE;
      asspxcoor (leftchild[node]);
    }
  rightchildexists = FALSE;
  if (rightchild[node] > -1)
    {
      rightchildexists = TRUE;
      asspxcoor (rightchild[node]);
    }
  hasonlyleftchild = leftchildexists && (!rightchildexists);
  hasonlyrightchild = rightchildexists && (!leftchildexists);
  isleaf = (!leftchildexists) && (!rightchildexists);
  h = ycoord[node];
  if (isleaf)
    {
      place = nextpos[h];
    }
  else if (hasonlyleftchild)
    {
      place = xcoord[leftchild[node]] + 1;
    }
  else if (hasonlyrightchild)
    {
      place = xcoord[rightchild[node]] - 1;
    }
  else
    { /* has two children */
      place = (xcoord[leftchild[node]]
              + xcoord[rightchild[node]]) / 2;
    }
  diff = nextpos[h] - place;
  if (diff > modifier[h])
    modifier[h] = diff;
  if (isleaf)
    xcoord[node] = place;
  else
    xcoord[node] = place + modifier[h];
  nextpos[h] = xcoord[node] + 2;
  shift[node] = modifier[h];
}

void makeshift (int node, int amount)
{
  xcoord[node] += amount;
  if (leftchild[node] > -1)
    {
      makeshift (leftchild[node], amount + shift[node]);
    }
  if (rightchild[node] > -1)
    {
      makeshift (rightchild[node], amount + shift[node]);
    }
}

void printthetree ()
{
  int i;
  for (i = 0; i < n; i++)
    {
      printf ("Node %2d (%2d,%2d): x = %2d, y = %2d, shift = %2d\n",
    i, leftchild[i], rightchild[i], xcoord[i], ycoord[i], shift[i]);
    }
```

```
  printf ("Height: %d\n", height);
}

void printthemodifiers ()
{
  int i;
  for (i = 0; i < height; i++)
    printf ("Level %2d: %2d\n", i, modifier[i]);
}

main ()
{
  int i, lc, rc;
  scanf ("%d", &n);

  xcoord = (int *) malloc (n * sizeof (int));
  ycoord = (int *) malloc (n * sizeof (int));
  shift = (int *) malloc (n * sizeof (int));
  leftchild = (int *) malloc (n * sizeof (int));
  rightchild = (int *) malloc (n * sizeof (int));

  for (i = 0; i < n; i++)
    {
      scanf ("%d %d", &lc, &rc);
      leftchild[i] = lc;
      rightchild[i] = rc;
      xcoord[i] = 0;
      ycoord[i] = 0;
      shift[i] = 0;
    }

  /* assign y-coordinates */

  height = 0;
  assycoor (0, 0);
  height++;

  nextpos = (int *) malloc (height * sizeof (int));
  modifier = (int *) malloc (height * sizeof (int));

  for (i = 0; i < height; i++)
    {
      nextpos[i] = 0;
      modifier[i] = 0;
    }

  /* assign provisional x-coordinates and shifts */

  asspxcoor (0);
  printf ("\nTree after phase 1:\n\n");
  printthetree ();
  printf ("\nModifiers after phase 1:\n\n");
  printthemodifiers ();

  /* make all shifts */

  makeshift (0, 0);
  printf ("\nTree after phase 2:\n\n");
  printthetree ();
}
```

Output:

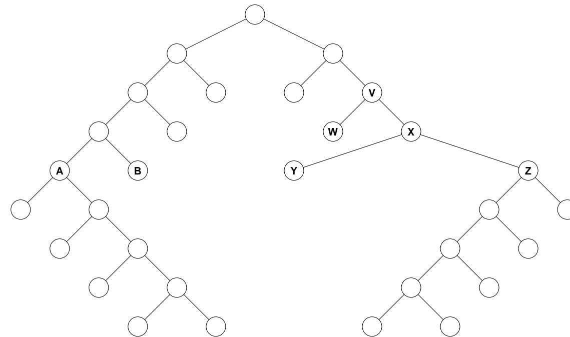
```
Tree after phase 1:

Node  0 ( 1, 2): x =  6, y =  0, shift =  0
Node  1 ( 3, 4): x =  4, y =  1, shift =  0
Node  2 ( 5, 6): x =  8, y =  1, shift =  0
Node  3 ( 7, 8): x =  3, y =  2, shift =  0
Node  4 (-1,-1): x =  5, y =  2, shift =  0
Node  5 (-1,-1): x =  7, y =  2, shift =  0
Node  6 ( 9,10): x =  9, y =  2, shift =  2
Node  7 (11,12): x =  2, y =  3, shift =  0
Node  8 (-1,-1): x =  4, y =  3, shift =  0
Node  9 (-1,-1): x =  6, y =  3, shift =  0
Node 10 (13,14): x =  8, y =  3, shift =  0
Node 11 (15,16): x =  1, y =  4, shift =  0
Node 12 (-1,-1): x =  3, y =  4, shift =  0
Node 13 (-1,-1): x =  5, y =  4, shift =  0
Node 14 (17,18): x = 11, y =  4, shift =  0
Node 15 (-1,-1): x =  0, y =  5, shift =  0
Node 16 (19,20): x =  2, y =  5, shift =  1
Node 17 (21,22): x = 10, y =  5, shift =  1
Node 18 (-1,-1): x = 12, y =  5, shift =  1
Node 19 (-1,-1): x =  0, y =  6, shift =  0
Node 20 (23,24): x =  2, y =  6, shift =  1
Node 21 (25,26): x =  8, y =  6, shift =  1
Node 22 (-1,-1): x = 10, y =  6, shift =  1
Node 23 (-1,-1): x =  0, y =  7, shift =  0
Node 24 (27,28): x =  2, y =  7, shift =  1
Node 25 (29,30): x =  6, y =  7, shift =  1
Node 26 (-1,-1): x =  8, y =  7, shift =  1
Node 27 (-1,-1): x =  0, y =  8, shift =  0
Node 28 (-1,-1): x =  2, y =  8, shift =  0
Node 29 (-1,-1): x =  4, y =  8, shift =  0
Node 30 (-1,-1): x =  6, y =  8, shift =  0
Height: 9

Modifiers after phase 1:

Level  0:  0
Level  1:  0
Level  2:  2
Level  3:  0
Level  4:  0
Level  5:  1
Level  6:  1
Level  7:  1
Level  8:  0

Tree after phase 2:

Node  0 ( 1, 2): x =  6, y =  0, shift =  0
Node  1 ( 3, 4): x =  4, y =  1, shift =  0
Node  2 ( 5, 6): x =  8, y =  1, shift =  0
Node  3 ( 7, 8): x =  3, y =  2, shift =  0
Node  4 (-1,-1): x =  5, y =  2, shift =  0
Node  5 (-1,-1): x =  7, y =  2, shift =  0
Node  6 ( 9,10): x =  9, y =  2, shift =  2
Node  7 (11,12): x =  2, y =  3, shift =  0
Node  8 (-1,-1): x =  4, y =  3, shift =  0
Node  9 (-1,-1): x =  8, y =  3, shift =  0
Node 10 (13,14): x = 10, y =  3, shift =  0
Node 11 (15,16): x =  1, y =  4, shift =  0
Node 12 (-1,-1): x =  3, y =  4, shift =  0
Node 13 (-1,-1): x =  7, y =  4, shift =  0
Node 14 (17,18): x = 13, y =  4, shift =  0
Node 15 (-1,-1): x =  0, y =  5, shift =  0
Node 16 (19,20): x =  2, y =  5, shift =  1
Node 17 (21,22): x = 12, y =  5, shift =  1
Node 18 (-1,-1): x = 14, y =  5, shift =  1
Node 19 (-1,-1): x =  1, y =  6, shift =  0
Node 20 (23,24): x =  3, y =  6, shift =  1
Node 21 (25,26): x = 11, y =  6, shift =  1
Node 22 (-1,-1): x = 13, y =  6, shift =  1
Node 23 (-1,-1): x =  2, y =  7, shift =  0
Node 24 (27,28): x =  4, y =  7, shift =  1
Node 25 (29,30): x = 10, y =  7, shift =  1
Node 26 (-1,-1): x = 12, y =  7, shift =  1
Node 27 (-1,-1): x =  3, y =  8, shift =  0
Node 28 (-1,-1): x =  5, y =  8, shift =  0
Node 29 (-1,-1): x =  9, y =  8, shift =  0
Node 30 (-1,-1): x = 11, y =  8, shift =  0
Height: 9
```

Beobachtung 2.2.1

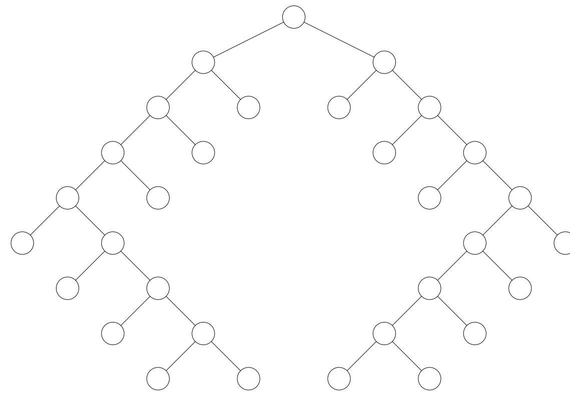
- minimale Breite wird nicht erreicht (hier 2 zuviel)
- auch Zentrieren geht nicht immer gut (wenn x-Differenz zwischen Kindern ungerade)

  REPARIERBAR !

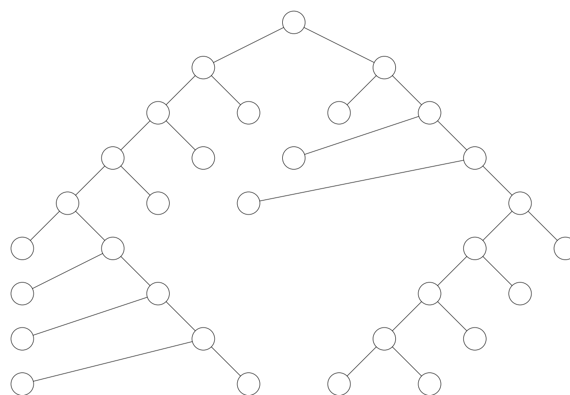
Bemerkung 2.2.2
Unter Aufgabe von (A3) kann WS so modifiziert werden, daß minimale Breite garantiert werden kann. Unser Beispiel:



Endpositionierung des Beispielbaums vom Algorithmus WS.



Beispielbaum gezeichnet vom Algorithmus RT.



Beispielbaum gezeichnet von einem modifizierten WS Algorithmus.

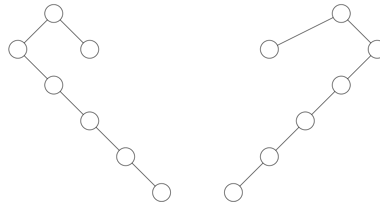
- symmetrische Bäume werden nicht notwendigerweise symmetrisch gezeichnet.

Neues Ästhetikkriterium:

(A4) Ein Baum und sein Spiegelbild sollen bis auf Spiegelung identisch gezeichnet werden, isomorphe Teilbäume sollen unabhängig von ihrer Position gleich gezeichnet werden.

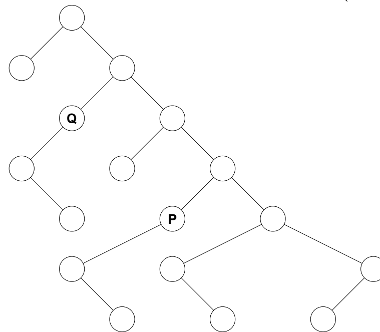
Beobachtung 2.2.3

- WS erfüllt das nicht



Ein Baum und sein Spiegelbild positioniert durch den Algorithmus WS.

- die schmalste Zeichnung kann die Erfüllung von (A4) unmöglich machen

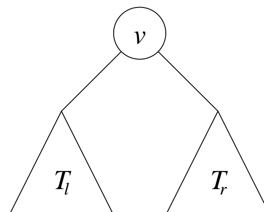


Schmalste Zeichnung eines Baumes die (A1)-(A3) erfüllt und (A4) verletzt. Die Unterbäume die an P und Q wurzeln sind isomorph, müssen aber nicht-isomorph gezeichnet werden, um eine schmalste Zeichung zu erhalten.

Plan: Wir versuchen (A1)-(A4) mit nicht notwendigerweise minimaler Breite.

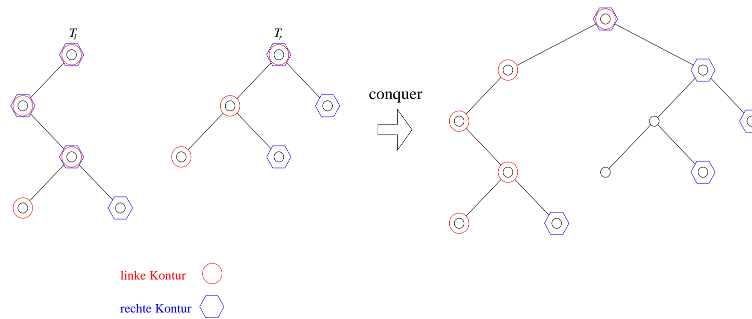
2.2.3 Elegante Lösung: Reingold & Tilford , (1981) [RT81]

Idee: Divide and Conquer implementiert durch POSTORDER Traversierung.



Vollständige Layouts von T_l und T_r bis auf Translation. Dann bis auf Minimalabstand zusammenschieben, v zentriert über Teilbäume plazieren.
$T_l = \varnothing$ oder $T_r = \varnothing$: Eine Position weiter links bzw. rechts von Unterwurzel.

Beipiel 2.2.4



Beobachtung 2.2.5
Eine Unterbaumzeichnung verändert sich nach der Fertigstellung nicht mehr ⇒ Erfüllung von (A4).

Schließlich: Alle Verschiebungen mittels PREORDER-Traversierung.

Gitterzeichnung: Wähle Minimalabstand 2 oder 3 um gerade Differenzen zwischen den Unterwurzeln zu erreichen.

Problem: Wir müssen die Konturen kennen.

Zusammenschieben auf Minimaldistanz

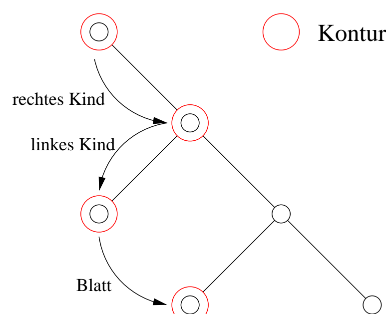
Folge (parallel)

- rechter Kontur der linken Teilbaums T_l
- linker Kontur des rechten Teilbaums T_r

Beobachtung 2.2.6
Zum Finden der linken Kontur können wir die Kinderzeiger verwenden

- entweder tatsächlich ein Kind
- oder Blatt.

Verwende Extrabit um diese Fälle zu unterscheiden.



Bestimmung der Konturen von $T(v)$ nach Zusammenfügung: Nur für den Teilbaum geringer Höhe ist etwas zu tun, im Beispiel rechte Kontur von $T(v)$.
Der letzte in alter rechter Kontur von T_r zeigt (mit extra Bit) auf rechtesten Knoten von T_l eine Ebene tiefer, von dort aus weiter in rechter Kontur von T_l.

Analyse:

- (A1)-(A4) sind erfüllt
- Laufzeit:

$$\begin{aligned} h(v) &:= \text{Höhe}(T(v)) + 1 \\ h_l(v) &:= \text{Höhe}(T_l) + 1 \\ h_r(v) &:= \text{Höhe}(T_r) + 1 \end{aligned}$$

$F(T(v))$: Zeit für Baum $T(v)$.

Es gilt $F(T(v)) = F(T_l) + F(T_r) + \min\{h_l(v), h_r(v)\}$

Behauptung 2.2.7
$F(T(v)) = n - h(v)$.

Beweis: Induktion

$n = 0$: $F(T(v)) = 0 - 0 = 0$

$n = 1$: $F(T(v)) = 1 - 1 = 0$

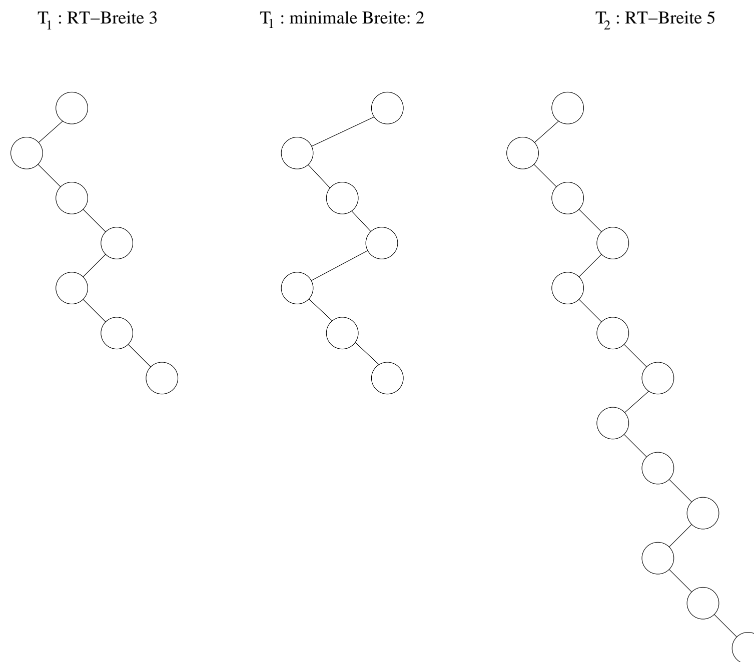
Sei die Behauptung korrekt für Bäume mit $k < n$ Knoten. Für einen Baum $T(v)$ mit n Knoten hat T_l $k < n$ Knoten.

$$\begin{aligned} F(T(v)) &= F(T_l) + F(T_r) + \min\{h_l(v), h_r(v)\} \\ &= k - h_l(v) + (n - k - 1) - h_r(v) + \min\{h_l(v), h_r(v)\} \\ &= n - 1 - h_l(v) - h_r(v) + \min\{h_l(v), h_r(v)\} \\ &= n - (\max\{h_l(v), h_r(v)\} + 1) \\ &= n - h(v) \end{aligned}$$

□

Zeit $O(n)$!!

Wir wissen schon: RT erzielt nicht immer die minimale Breite.
Schlimmer: Es gibt Binärbäume T, für die RT Zeichnungen mit Breite $\frac{n+2}{3}$ produziert, während die optimale Breite 2 ist.



Allgemein: T_k hat $n = 6k + 1$ Knoten $\Rightarrow k = \frac{n-1}{6}$
T_k hat RT-Breite $2k + 1 = \frac{n-1}{3} + 1 = \frac{n+2}{3}$

□

2.2.4 Komplexität der Breitenminimierung

→ unter Einhaltung aller Ästhetikkriterien bis auf Spiegelsymmetrie

Kontinuierliche x-Koordinaten

$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$. Es gilt: Lineare Optimierungsprobleme der Form

$$\begin{array}{rlcll} (LP) \text{ minimiere} & \sum_{j=1}^n c_j x_j & & & \\ \text{so daß} & \sum_{j=1}^n a_{ij} x_j & = & b_i & \forall\, i \in \{1, \ldots, k\} \\ & \sum_{j=1}^n b_{ij} x_j & \geq & d_i & \forall\, i \in \{1, \ldots, l\} \end{array}$$

oder kurz:

$$\begin{array}{rl} (LP) \text{ min} & \{c^T x \mid Ax = b,\ Bx \geq d\} \\ \text{für} & c \in \mathbb{R}^n,\ A \in \mathbb{R}^{k \times n},\ B \in \mathbb{R}^{l \times n},\ b \in \mathbb{R}^k,\ d \in \mathbb{R}^l \end{array}$$

können in polynomieller Zeit gelöst werden.

Plan: wir formulieren das Breitenminimierungsproblem als LP (in polynomieller Zeit).

Variablen: x_j $(j \in \{1, \ldots, n\})$; $L, R \in \mathbb{R}$ mit $L :=$ linkeste x-Koordinate und $R :=$ rechteste x-Koordinate.

Restriktionen:

$$\begin{aligned} x_j &\geq L \quad \forall\, j \in \{1,\ldots,n\} \\ x_j &\leq R \quad \forall\, j \in \{1,\ldots,n\} \end{aligned}$$

Sei $l(j) :=$ linkes Kind von j und $r(j) :=$ rechtes Kind von j; außerdem sei rechtes Kind rechts und linkes Kind links vom Vater:

$$\begin{aligned} x_{r(j)} - x_j &\geq 1 \quad \forall\, j \in \{1,\ldots,n \mid r(j) \text{ existiert}\} \\ x_j - x_{l(j)} &\geq 1 \quad \forall\, j \in \{1,\ldots,n \mid l(j) \text{ existiert}\} \end{aligned}$$

→ Berechnung durch PREORDER-Traversierung

Mindestabstand 2 zwischen je zwei Knoten auf gleicher Tiefe:

$x_k - x_l \geq 2$ $\forall$ Paare k, l von Knoten mit Tiefe(k)=Tiefe(l) und k, l sind auf ihrer Ebene benachbart

→ Berechnung durch Breitensuche

Väter weden über ihren Kindern zentriert:

$$x_{r(j)} - x_j = x_j - x_{l(j)} \quad \forall\, j \in \{1,\ldots,n \mid l(j) \text{ und } r(j) \text{ existieren}\}$$

Isomorphe Teilbäume werden bis auf Translation gleich gezeichnet:

$x_{r(j_i)} - x_{j_i} = x_{r(j_{i+1})} - x_{j_{i+1}}$ $\forall$ Mengen $\{j_1,\ldots,j_k\}$ von Wurzeln isomorpher Teilbäume mit mindestens zwei Knoten und nichtleerem rechten Kind, wobei $i \in \{1,\ldots,k-1\}$

$x_{j_i} - x_{l(j_i)} = x_{j_{i+1}} - x_{l(j_{i+1})}$ ... mit leerem rechten Kind ...

(Korrespondierende Teilbäume isomorpher Teilbäume sind isomorph; also sind so alle isomorphen Teilbäume vollständig erfaßt).

Zielfunktion: minimiere $R - L$

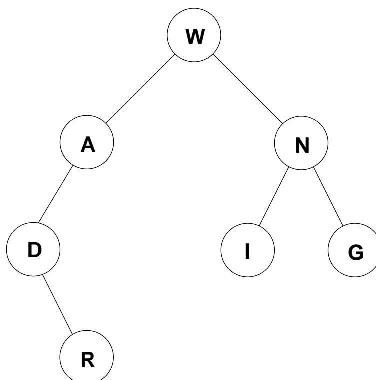
Bestimmung der Mengen isomorpher Teilbäume

Grundidee: Die INORDER- und die PREORDER-Bearbeitungsreihenfolge charakterisieren einen Binärbaum vollständig.

Beipiel 2.2.8

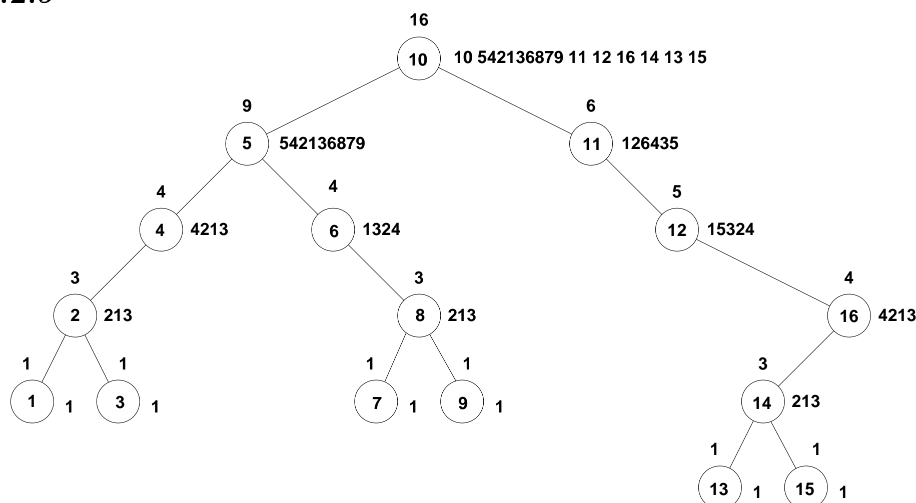
INORDER: D R A W I N G
PREORDER: W A D R N I G

Konstruktion:



Wir numerieren grundsätzlich in INORDER-Reihenfolge von 1 bis $n = |V|$. Dann charakterisiert die PREORDER-Reihenfolge den Baum eindeutig. Im Rahmen einer POSTORDER-Traversierung ordnen wir jedem Teilbaum die Anzahl seiner Knoten sowie die Kodierung des Teilbaums unter ihm zu.

Beipiel 2.2.9



Durch lexikographische Sortierung der Kodierung werden alle isomorphen Teilbäume adjazent in der sortierten Folge (ohne Blätter):

(2)	(8)	(14)	(6)	(4)	(16)	(12)	(11)
2 1 3	2 1 3	2 1 3	1 3 2 4	4 2 1 3	4 2 1 3	1 5 3 2 4	1 2 6 4 3 5

(5)	(10)
5 4 2 1 3 6 8 7 9	10 5 4 2 1 3 6 8 7 9 11 12 16 14 13 15

1. Konstruktion des LPs in polynomieller Zeit
2. Lösen des LPs in polynomieller Zeit

$\Rightarrow$ Das kontinuierliche Breitenminimierungsproblem ist polynomiell lösbar.

Ganzzahlige x-Koordinaten

Das Breitenminimierungsproblem (BMP) kann wie vorhin mit der zusätzlichen Restriktion x_j ganzzahlig $\forall\, j \in \{1, \ldots, n\}$ formuliert werden.

Für ganzzahlige lineare Optimierungsprobleme (IP) sind keine polynomiellen Algorithmen bekannt. Die Entscheidungsvariante BMPEV des Problems lautet:

„Existiert eine Zeichung mit Breite $\leq W$?“

Ist BMP in polynomieller Zeit lösbar, so auch BMPEV.

3SAT

Instanz: Konjunktion $E = F_1 \wedge F_2 \wedge \ldots \wedge F_r$ mit $F_i = (y_{i1} \vee y_{i2} \vee y_{i3})$, $i \in \{1, \ldots, r\}$ und y_{ij} Literale der Form x_l oder $\overline{x_l}$ für eine Boolesche Variable $x_l \in \{x_1, \ldots, x_n\}$.

Frage: Ist E erfüllbar, d.h. gibt es eine Wahrheitsbelegung für $x_1, \ldots, x_n$, so daß E wahr ist ?

Wir skizzieren eine polynomielle Reduktion des 3SAT-Problems auf BMPEV.

$\Rightarrow$ Ist BMPEV polynomiell lösbar, so auch 3SAT. Letzteres ist aber nicht möglich, es sei denn $P = NP$.

Es gilt: BMPEV $\in NP$ (da $IP \in NP$), d.h. für gegebene x_j kann in polynomieller Zeit verifiziert werden, ob die Ästhetikkriterien erfüllt sind und die Breite $\leq W$ ist.

Reduktion:

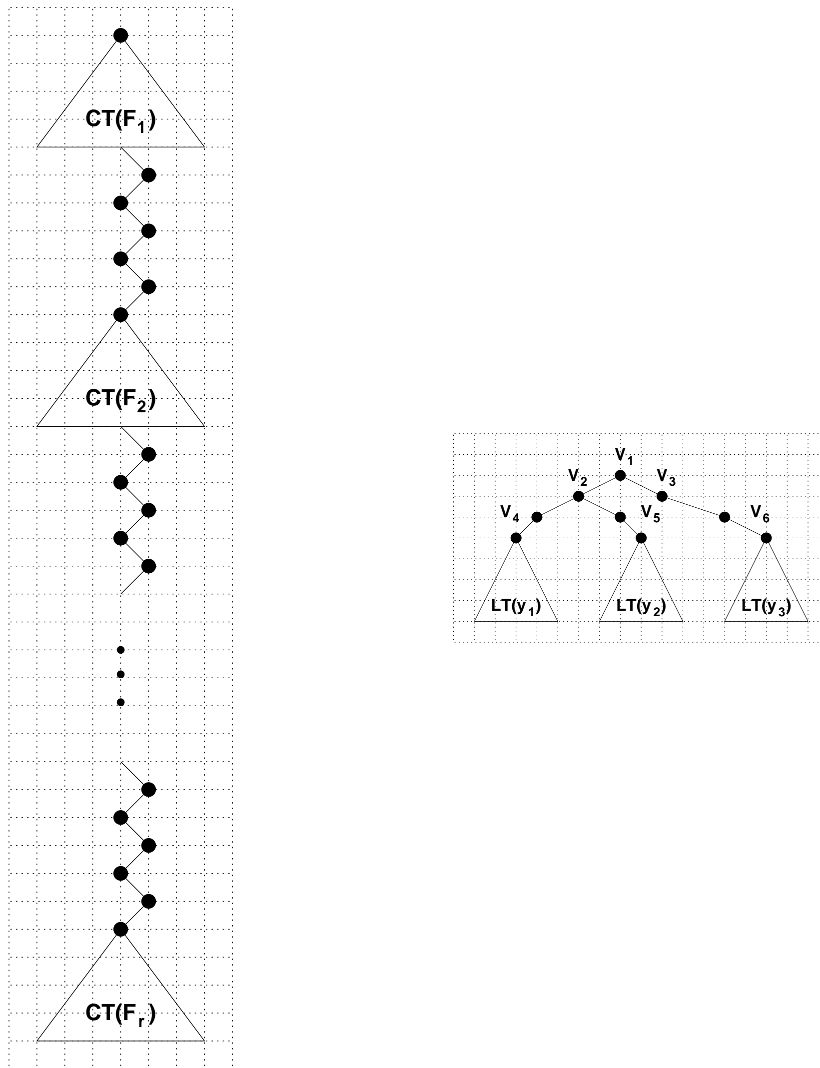
$$\text{Instanz von 3SAT } E$$
$$\Big\downarrow \text{polynomielle Transformation}$$
$$\text{Instanz von BMPEV } (T, W)$$

mit: Baum T ist mit Breite $\leq W$ zeichenbar $\Leftrightarrow$ E ist erfüllbar.

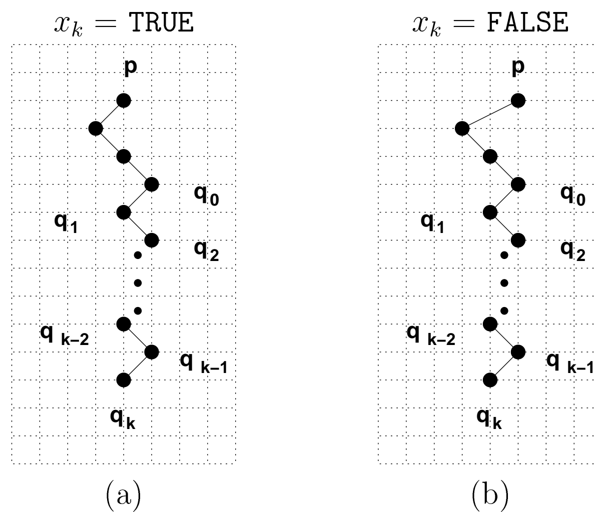
Wenn das gelingt, ist gezeigt: BMPEV ist NP-vollständig, BMP ist NP-schwierig.

Konstruktion

$$E \rightarrow \text{Baum } T(E)$$
$$\text{mit: } E \text{ erfüllbar} \Leftrightarrow T(E) \text{ ist mit Breite 24 zeichenbar}$$

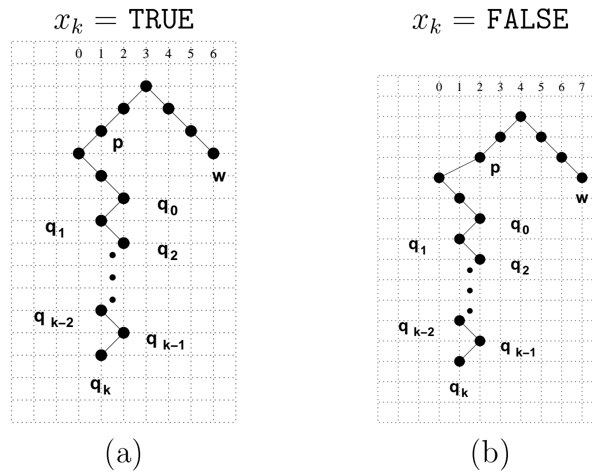


$T(E)$ mit $E = (F_1 \wedge F_2 \wedge \ldots \wedge F_r)$ Klausel-Baum $CT(E)$ mit $F = (y_1 + y_2 + y_3)$

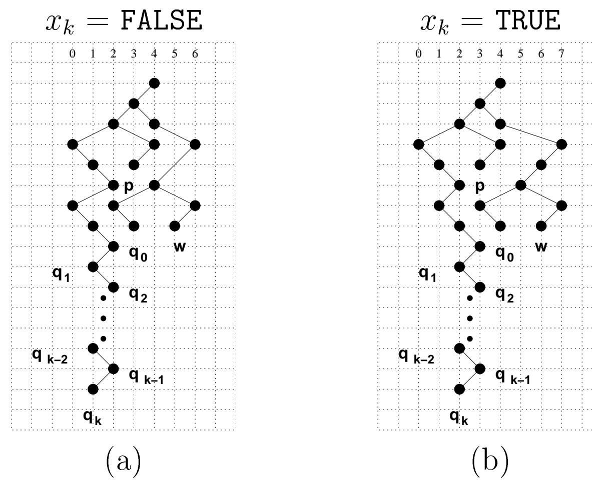


(a) (b)

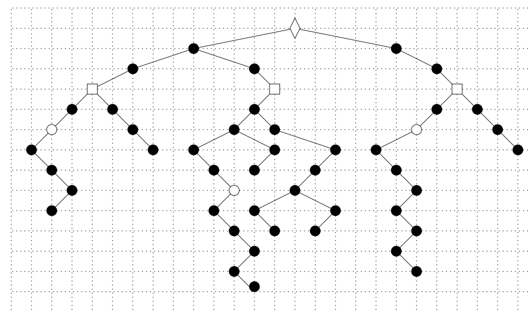
Variablen-Baum $VT(x_k)$



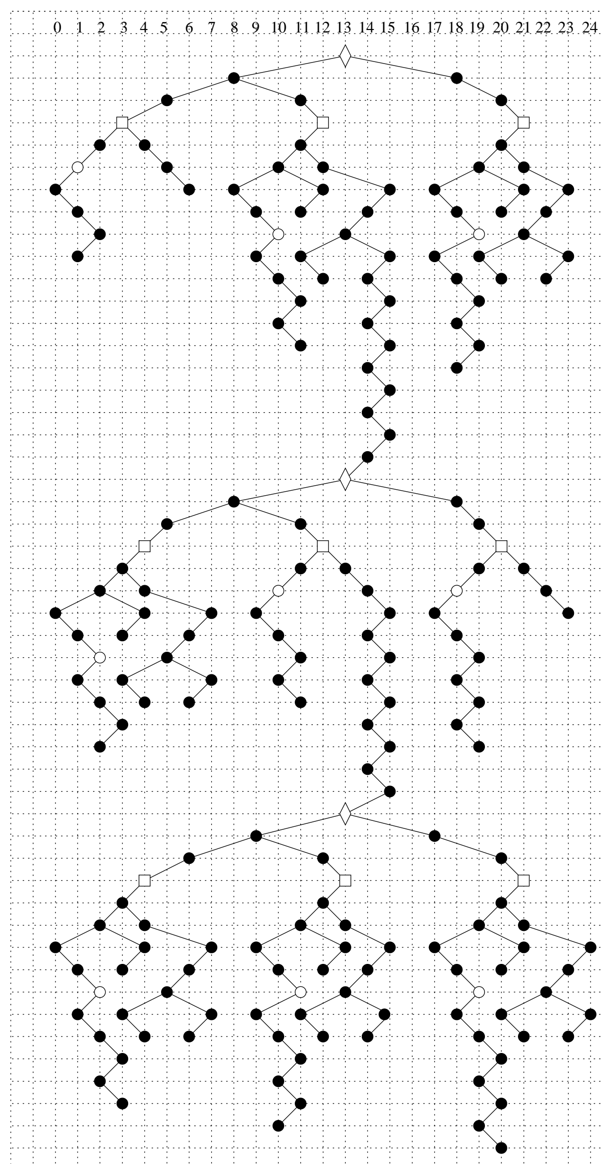
Literal-Baum $LT(y)$ mit $y = x_k$



Literal-Baum $LT(y)$ mit $y = \overline{x}_k$



Der Klausel-Baum $CT(F)$ mit $F = (x_1 + \overline{x}_2 + x_4)$. Die Raute ist Wurzel von $CT(F)$; die Quadrate sind Wurzeln der drei Literal-Bäume; die (leeren) Kreise sind die Wurzeln der drei Variablen-Bäume.



Plazierung π von $T(E)$ mit

$$E = (x_1 + \overline{x}_2 + \overline{x}_3) \wedge (\overline{x}_1 + x_2 + x_4) \wedge (\overline{x}_2 + \overline{x}_3 + \overline{x}_4)$$

Die Rauten sind Wurzeln der Klausel-Bäume; die Quadrate sind Wurzeln der drei Literal-Bäume; die (leeren) Kreise sind die Wurzeln der drei Variablen-Bäume. π entspricht der Wahrheitsbelegung, die x_1, x_2, x_4 wahr und x_3 falsch macht.

Satz 2.2.10
BMPEV ist NP-vollständig, insbesondere auch, wenn $W = 24$.

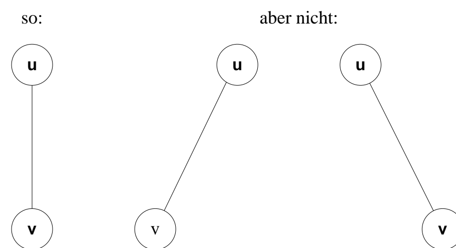
Korollar 2.2.11
Falls $P \neq NP$, so existiert kein polynomieller Algorithmus $\mathcal{A}$, der eine die Ästhetikkriterien erfüllende Zeichnung mit ganzzahligen Koordinaten liefert, so daß die Breite kleiner als $\frac{25}{24}$ der kleinstmöglichen Breite ist.

informal: Die Approximation der minimalen Breite bis auf ca. 4% ist NP-schwierig.

2.3 Zeichnen von allgemeinen Bäumen

Beobachtung 2.3.1
Im allgemeinen Wurzelbaum gibt es kein linkes und kein rechtes Kind, sondern nur ein linkes äußeres und ein rechtes äußeres Kind.

Konvention



⇒ (A2) fällt weg. Algorithmus von RT kann verallgemeinert werden auf allgemeine Bäume. Problem:

1. (A2) wird weiterhin beachtet
2. kleine Unterbäume zwischen großen Unterbäumen werden zum linken Unterbaum hin gezeichnet.
3. Aus 2. folgt, daß (A4) nicht mehr erfüllt wird

2.3.1 Idee (nach Walker (1990) [Wal90])

Traversiere Unterbäume eines Knotens von links nach rechts. Wird ein Unterbaum x nach rechts um einen Wert d verschoben damit es zu keiner Überschneidung mit einem Unterbaum y links von x kommt, dann verschiebe alle zwischen x und y liegenden Unterbäume $z_1, \ldots, z_k$ wie folgt:

z_i wird verschoben um: $i \cdot \frac{d}{k+1} \; \forall \, i = 1, \ldots, k$

2.3.2 Skizze des Algorithmus

(i) Durchlaufe Baum in POSTORDER.

∀ Knoten: bestimme `prelim`, die vorläufige Koordinate, sowie die Modifikation

(ii) PREORDER-Traversierung: verschiebe Unterbaum $T(v)$ um den in v angegebenen Modifier.

Zu (i): Sei u ein Knoten aus Kontur eines Unterbaumes $T(v)$ mit Wurzel r. Dann ist $\sum(u)$ die Summe der Modifikationen aller Knoten auf dem eindeutigen Pfad u nach v (ohne u!).

Die Summe $\sum(u)$ wird benötigt, um zu bestimmen, wieviel ein Knoten relativ zur gerade betrachteten Unterbaumwurzel nach rechts verschoben wird.

$\Rightarrow$ nur so können wir Kollisionen mit benachbarten Bäumen bestimmen.

Konkret: Sei $v \in T(v)$; $v_1, \ldots, v_k$ Kinder von v.

(i) $\forall i = 1, \ldots, k$ bestimme rekursiv Positionen und Modifikationen der Knoten in $T(v_i)$.

(ii) Traversiere $v_1, \ldots, v_k$ von links nach rechts; $\forall\, i = 2, \ldots, k$ verschiebe $T(v_i)$ solange bis $T(v_i)$ mit keinem Baum $T(v_1), \ldots, T(v_{i-1})$ kollidiert. Verteile dabei erzeugte Abstände gleichmäßig auf kleinere dazwischenliegende Bäume.

Genauer: Durchlaufe Knoten w aus linker Kontur von $T(v_i)$ bis entweder

- w ist letzter Knoten aus linker Kontur oder
- w hat keinen linken Nachbarn in $T(v)$

Sei u linker Nachbar aus $T(v_j)$ mit $j \in \{1, \ldots, i-1\}$. Bestimme $\sum(u)$ und $\sum(w)$.
Ist $\texttt{prelim}\,(u) + \sum(u) + c \geq \texttt{prelim}\,(w) + \sum(w)$, so verschiebe $T(v_i)$ um

$$\texttt{prelim}(u) + \sum(u) + c - \ \texttt{prelim}(w) - \sum(w)$$

wobei $c \approx$ minimaler Knotenabstand und evtl. anderes.

Ist u auf unterster Schicht von $T(v_j)$, so sei d_{ij} die Summe der Verschiebungen um Kollisionen zu vermeiden. Für alle dazwischenliegenden Bäume $T(v_l)$ mit $l = j+1, \ldots, i-1$ verschiebe $T(v_l)$ um

$$(l-j) \cdot \frac{d_{ij}}{i-j}$$

Schwächen der Publikation von Walker [Wal90]:

1) $O(n^2)$ zur Bestimmung der Konturen (`getleftmost`)

2) $O(n^2)$ zur Bestimmung von Σ für jeden Knoten aus der Kontur

3) $O(n^2)$ für Verschieben in jeder Schicht

4) $O(n^2)$ für Abbruchkriterium

5) $O(n^2)$ für die Verschiebung der kleineren Bäume

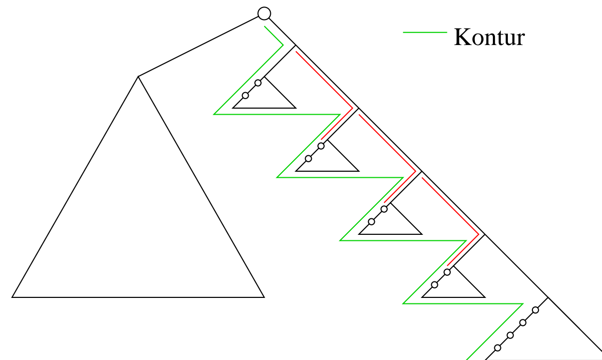
2.3.3 Konzepte für einen Linearzeit Algorithmus

zu 1) Bestimmung der Kontur der Unterbäume wie bei Reingold & Tilford [RT81]. Benötigte Zeit $O(n)$.

zu 2) Bestimmung der summierten Modifikatoren entlang der Kontur.

Problem: Wir können nicht einfach die Kontur entlanglaufen, um für jeden Knoten w sein $\Sigma(w)$ zu bestimmen.

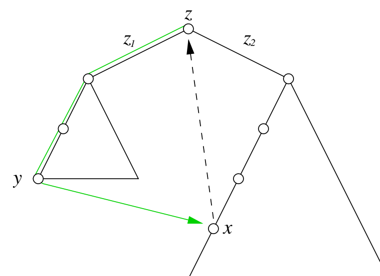
Beipiel 2.3.2



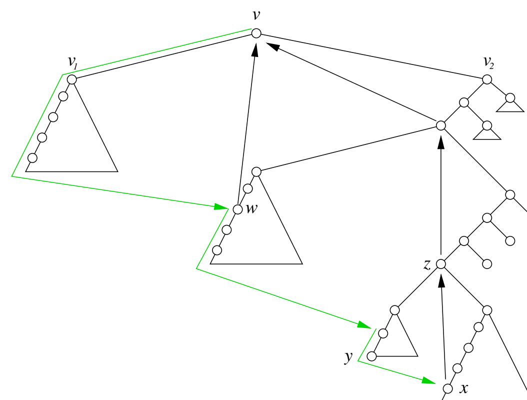
Beobachtung 2.3.3
Solange immer über linkestes (rechtestes) Kind der nächste Knoten auf der linken (rechten) Kontur erreicht wird, kann Σ für alle Knoten in $O(1)$ Zeit pro Knoten bestimmt werden.

Entwicklung eines Lösungsansatzes: Threads entstehen durch Kombination der Konturen zweier benachbarter Unterbäume. Dabei wird die Kontur des grösseren Baumes bis zum Thread gescannt. Wir nutzen dies aus.

Beipiel 2.3.4



Erweitertes Beispiel 2.3.5



Lösung: Sei (y, x) ein Thread mit $lev(y) = lev(x) - 1$ $[lev(v) = Tiefe(v)]$, dann speichere in x den Wert $\Sigma(x) - \Sigma(y)$. Dieser Wert beschreibt den Abstand von y und x in dem neuen Unterbaum.

zu 3) Verschiebung der Unterbäume: Nicht in jeder Schicht, sondern einmal.

zu 4) Abbruchkriterium klar wegen Kontur.

zu 5) Verschiebung der kleineren Unterbäume:

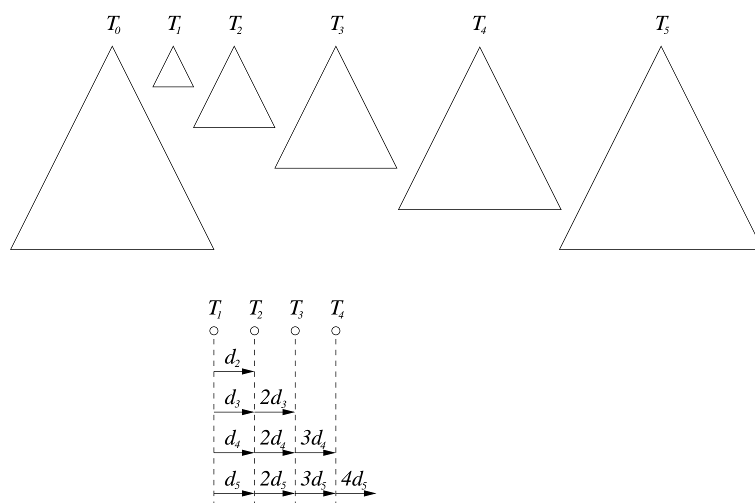
Merke für neu eingefügt Threads zu welchem Unterbaum dieser Thread wechselt.

Wir dürfen nicht nachdem $T(v_i)$ relativ zu $T(v_j)$ $j < i$ verschoben wurde

$$T(v_{j+1}), \ldots, T(v_{i-1})$$

verschieben, sonst $O(n^2)$.

Beipiel 2.3.6



Beobachtung 2.3.7
Muß ein Baum $T(v_l)$ zwischen zwei Bäumen $T(v_j)$ und $T(v_i)$ um den Wert $(l-j)\frac{d_{ij}}{i-j}$ verschoben werden, so muß $T(v_{l-1})$ um den Werte $(l-j)\frac{d_{ij}}{i-j} - \frac{d_{ij}}{i-j}$ verschoben werden.

Definition 2.3.8
$(i-1-j)\frac{d_{ij}}{i-j}$ *sei die* ***maximale Änderung*** , $\frac{d_{ij}}{i-j}$ *sei die Änderung* .

Idee: Durchlaufe die Unterbäume eines Knotens von rechts nach links. Verschiebe einen Unterbaum $T(v_l)$ um den Betrag um den $T(v_{l+1})$ verschoben wurde vermindert um die Summe der Änderungen erhöht um die Summe der maximalen Änderung die in $T(v_l)$ beginnt.

Erzeuge 5 Arrays der Größe $k-1$ in V (K = Anzahl der Kinder von v)

Maximale Änderung	MÄ
Änderung	Ä
Ende der Anderung	EÄ

Fülle diese während der Verschiebung von Unterbäumen von links nach rechts aus,

Summe der Verschiebungen	ΣV
Summe der Änderungen	ΣÄ

Berechnung im 2. Durchlauf von rechts nach links.

MÄ[i]	speichert die maximale Änderung die in Baum $T(v_i)$ beginnt. (Sollten mehrere maximale Änerungen in $T(v_i)$ beginnen, so speichert MÄ[i] die Summer dieser maximalen Änderungen).
Ä[i]	speichert die (Summe der) Änderungen bezogen auf MÄ[i+1].
EÄ[i]	speichert die Summe der Enden der Änderungen.
ΣV[i]	speichert die Summe der Verschiebungen des Baumes $T(v_i)$.
ΣÄ[i]	speichert die kummulierte Änderung.

Rechenregel: (Reihenfolge wichtig)

1. ΣÄ[i]= ΣÄ[i+1]+Ä[i]−EÄ[i]
2. ΣV[i]= ΣV[i+1]+MÄ[i]−ΣÄ[i]

$O(n)$!

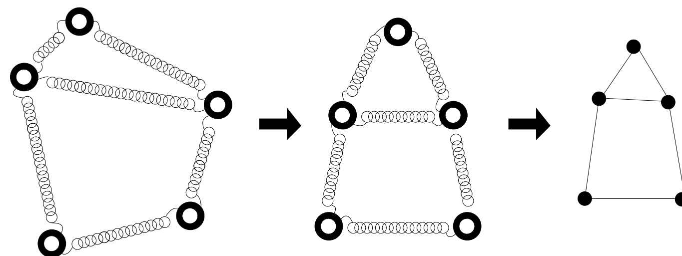
Kapitel 3

Kräftebasierte Verfahren

3.1 Modell

3.1.1 Grundidee

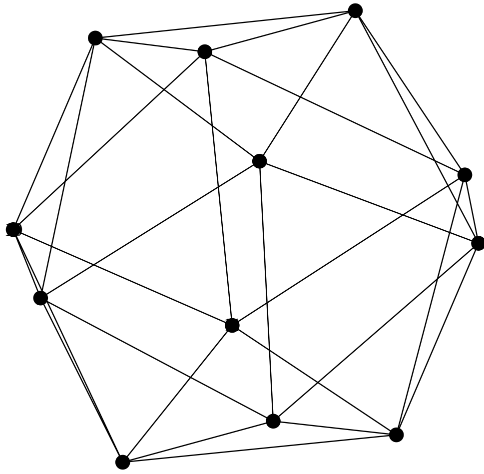
- Knoten: elektrisch gelandene Partikel, die sich gegenseitig abstoßen
- Kanten: Federn
- gesucht: Zustand minimaler Energie



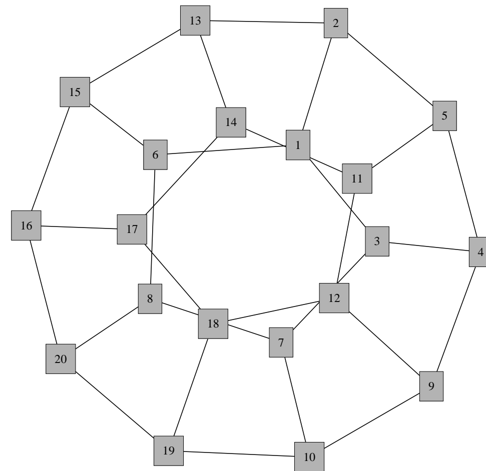
Kräftebasierte Verfahren sind charakterisiert durch:

- das Modell: ein durch die Knoten und Kanten definiertes Kräftesystem, d.h. ein physikalisches Modell → Definition der Ästhetikkriterien
- der Algorithmus: ein Rechenverfahren, welches eine Approximation eines Gleichgewichtszustandes des Kräftesystems liefert: eine Position für jeden Knoten, so daß an jedem Knoten die Kraft 0 wirkt.
- intuitiv leicht verständlich
- leicht zu programmieren
- gute Resultate für dünne Graphen
- existieren schon lange, z.B. Tutte („How to draw a graph“ (1960,1963) [Tut60] [Tut63])
- viele Varianten publiziert

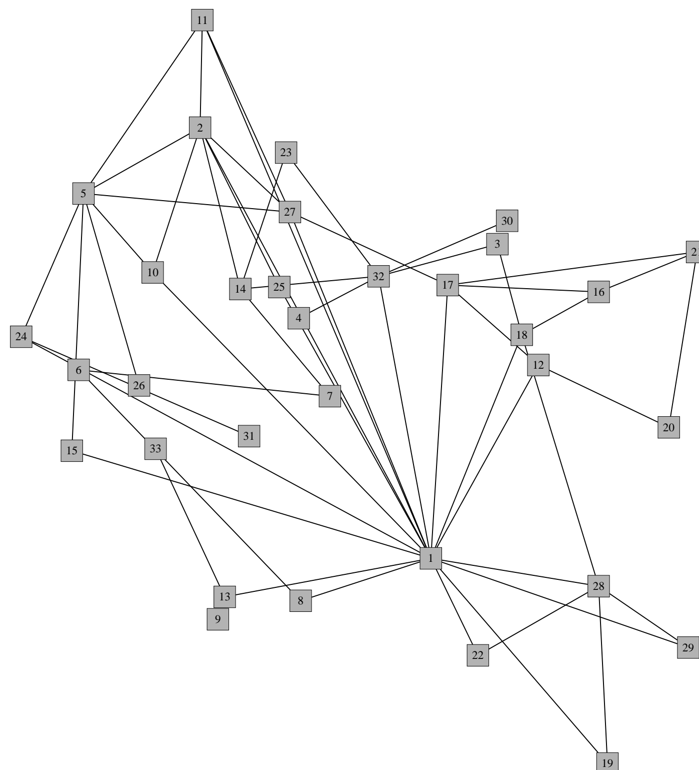
3.1.2 Einige Beispielzeichnungen



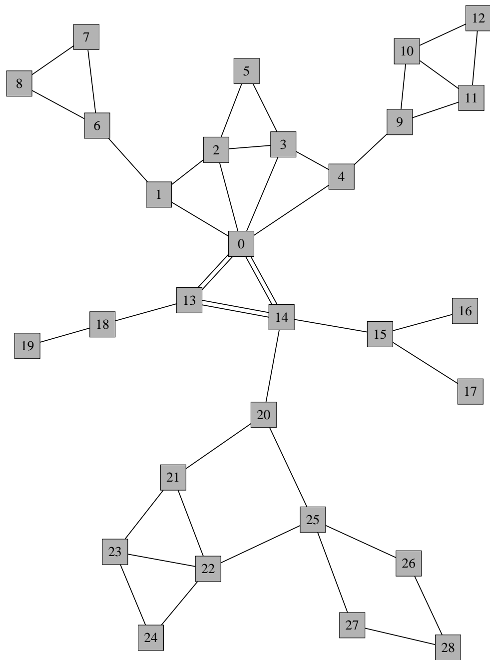
Mit dem GEM-Algorithmus gezeichnetes Ikosaeder (A. Frickund A. Ludwig)



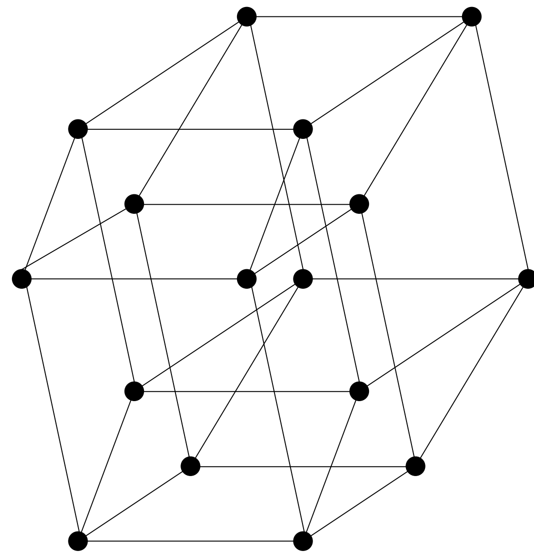
Mit einem Kräfteverfahren gezeichneter Dodekaeder (U. Erlingsson und M. Krishnamoorthy)



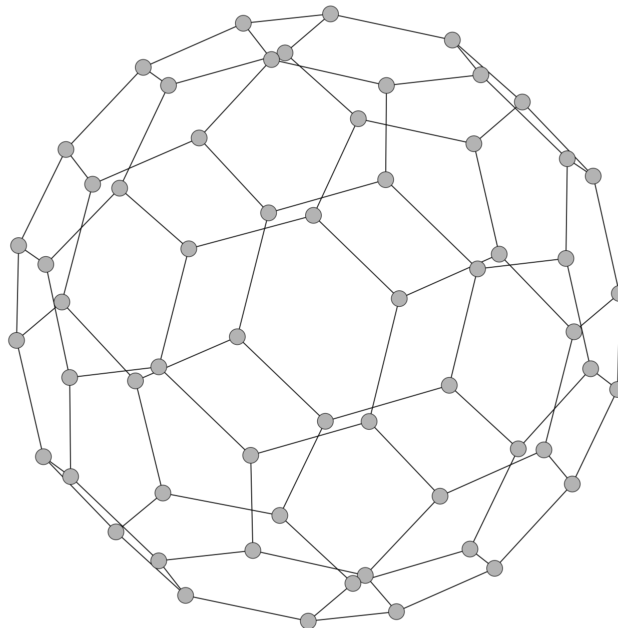
Graph, der das Verfolgen von Links im Internet wiederspiegelt und mit einem einfachen Kreäfteverfahren gezeichnet ist (J. Fenwick, D. Thompson und R. Stacey)



Mit einem einfachen Kräfteverfahren gezeichneter Graph (Tom Sawyer Software)



Dieser Hyperkubus wurde mit einem Verfahren von Tunkelang gezeichnet.



Mit einem experimentellen einfachen Kräfteverfahren von Tom Sawyer Software gezeichneter Fußball (A. Frick)

Kraft an einem Knoten $v \in V$

$$F(v) = \sum_{(u,v)\in E} f_{uv} + \sum_{(u,v)\in V\times V} g_{uv}$$

wobei

$$\begin{aligned} f_{uv} &= \text{Kraft durch Feder zwischen } u \text{ und } v, \\ g_{uv} &= \text{elektrische Abstoßung zwischen } u \text{ und } v \end{aligned}$$

f_{uv} ist proportional zur Differenz zwischen der Distanz von u und v und der 0-Energie-Länge der Feder (Hooke'sches Gesetz) , und g_{uv} umgekehrt proportional zum Quadrat der Distanz von u und v
Für $p, q \in \mathbb{R}^n$ sei

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

der Euklidische Abstand von p und q.

Wir beschränken uns auf den zweidimensionalen Fall ($n = 2$). (Analoge Konstruktionen sind auch im dreidimensionalen Fall möglich.)

Position von Knoten $v \in V :\ p_v \begin{pmatrix} x_0 \\ y_v \end{pmatrix}$
D.h. x-Komponente $F(v)$:

$$\sum_{(u,v)\in E} |k_{uv}^{(1)} \cdot (d(p_u, p_v) - l_{uv})| \cdot \frac{x_v - x_u}{||d(p_v, p_u)||} + \sum_{(u,v)\in V\times V} \frac{k_{uv}^{(2)}}{|(d(p_u, p_v))^2|} \cdot \frac{x_v - x_u}{||d(p_v, p_u)||}$$

(y-Komponente analog).
mit:

$$\begin{aligned} l_{uv} &= \text{natürliche (0-Energie)-Länge der Feder zwischen } u \text{ und } v \\ k_{uv}^{(1)} &= \text{Steife der Feder} \\ k_{uv}^{(2)} &= \text{Stärke der elektrischen Abstoßung zwischen } u \text{ und } v \end{aligned}$$

Modellierte Ästhetikkriterien:

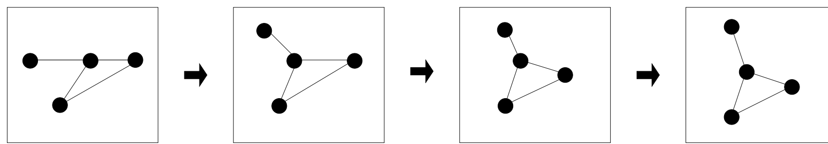
- Tendenz zur Ideallänge von Kanten
- Knoten nicht zu nahe zusammen
- (indirekt:) Tendenz zur Symmetrie
- Beeinflußbarkeit durch Wahl der Parameter $l_{uv}, k_{uv}^{(1)}, k_{uv}^{(2)}$

3.1.3 Einfacher Algorithmus

1. Starte mit zufälligen Positionen $p_v \; \forall \, v \in V$
2. Iteriere:
   - Berechne Kräfte $F(v) \; \forall \, v \in V$
   - Modifiziere p_v leicht in Richtung $F(v)$

$$p_v \longleftarrow p_v + \varepsilon \cdot F(v) \; \forall \, v \in V$$

   bis Modifikation klein genug ist.



3.2 Älteste Variante (Tutte (1960,1963) [Tut60] [Tut63])

- $l_{uv} = 0 \; \forall \, (u,v) \in E$
- $k_{uv}^{(1)} = 1 \; \forall \, (u,v) \in E$
- keine elektrischen Kräfte

D.h.:

$$F(v) = \sum_{(u,v) \in E} (p_u - p_v)$$

Problem: $p_v = 0 \; \forall \, v \in V$ ist optimal: keine gute Zeichnung

Ausweg: Fixiere Position von wenigstens drei Knoten, etwa als Ecken eines konvexen Polygons („Festnageln“) und bestimme Positionen p_v der anderen (freien) Knoten $v \in V$, so daß $F(v) = 0$.

D.h. löse das Gleichungssystem

$$\begin{aligned} \sum_{(u,v) \in E} (x_u - x_v) &= 0 \\ \sum_{(u,v) \in E} (y_u - y_v) &= 0 \end{aligned}$$

Notation: $V = V_0 \cup V_1$ $(V_0 \cap V_1 = \varnothing)$ mit

$$\begin{aligned} V_0(v) &= \text{ Menge der fixierten Knoten} \\ V_1(v) &= \text{ Menge der freien Knoten} \end{aligned}$$

Für $v \in V$ seien:

$$\begin{aligned} N_0(v) &= \{u \in V_0 \mid (u,v) \in E\} \text{ Menge der fixierten Nachbarn von } v \\ N_1(v) &= \{u \in V_1 \mid (u,v) \in E\} \text{ Menge der freien Nachbarn von } v \\ \Delta(v) &= |\{(u,v) \mid (u,v) \in E\}| \text{ Grad von } v \end{aligned}$$

Für $v \in V_0$ sei $\begin{pmatrix} x_v^* \\ y_v^* \end{pmatrix}$ die fixierte Position von v.

Wir erhalten das Gleichungssystem

$$\begin{aligned} \Delta(v)x_v - \sum_{u \in N_1(v)} x_u &= \sum_{w \in N_0(v)} x_w^* \quad \forall v \in V_1 \\ \Delta(v)y_v - \sum_{u \in N_1(v)} y_u &= \sum_{w \in N_0(v)} y_w^* \quad \forall v \in V_1 \end{aligned}$$

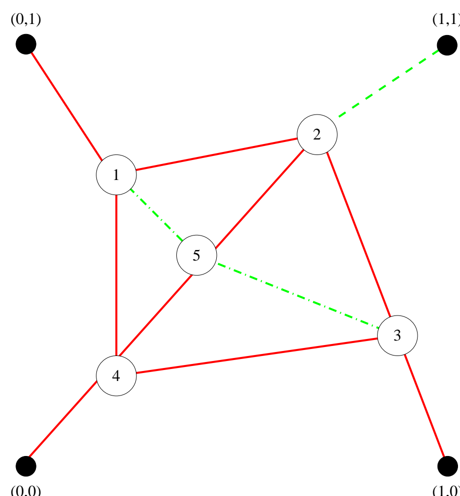
mit je $|V_1|$ Unbekannten und Gleichungen

$\longrightarrow$ Lineare Algebra, Numerik; schnelle Methoden für dünn besetzte Matrizen bekannt.

Geometrische Interpretation der Lösung: Jeder freie Knoten liegt im Schwerpunkt seiner Nachbarn.

$\longrightarrow$ Schwerpunkt-Method (Barycenter Methode).

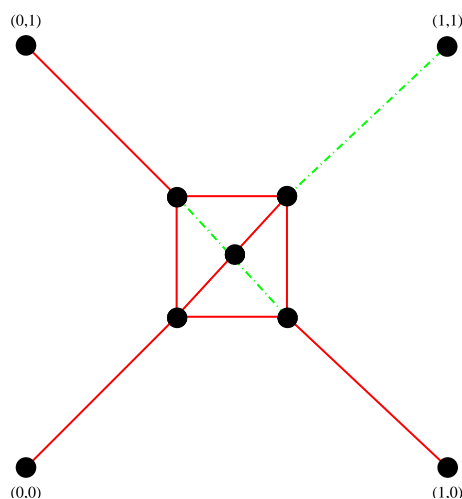
3.2.1 Beispiel



1. Experiment: rote und grüne Kanten

Matrix A: Gleichungssysteme: Lösungen:

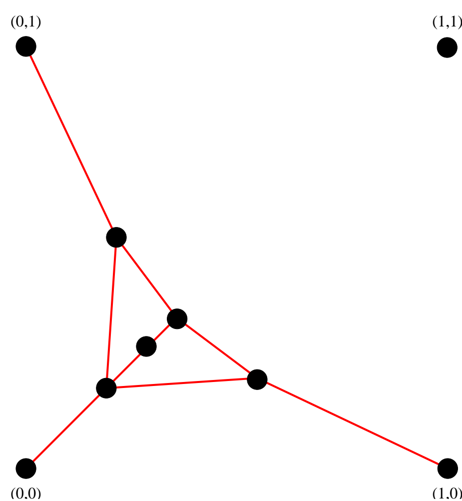
$$\begin{pmatrix} 4 & -1 & 0 & -1 & -1 \\ -1 & 4 & -1 & 0 & -1 \\ 0 & -1 & 4 & -1 & -1 \\ -1 & 0 & -1 & 4 & -1 \\ -1 & -1 & -1 & -1 & 4 \end{pmatrix} \quad Ax = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}, Ay = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad x = \begin{pmatrix} 0.375 \\ 0.625 \\ 0.625 \\ 0.375 \\ 0.5 \end{pmatrix}, y = \begin{pmatrix} 0.675 \\ 0.675 \\ 0.375 \\ 0.375 \\ 0.5 \end{pmatrix}$$



2. Experiment: nur rote Kanten

Matrix A: Lösungen:

$$\begin{pmatrix} 3 & -1 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 \\ 0 & -1 & 3 & -1 & 0 \\ -1 & 0 & -1 & 4 & -1 \\ 0 & -1 & 0 & -1 & 2 \end{pmatrix} \quad x = \begin{pmatrix} 0.203 \\ 0.348 \\ 0.536 \\ 0.261 \\ 0.304 \end{pmatrix}, \ y = \begin{pmatrix} 0.536 \\ 0.348 \\ 0.203 \\ 0.261 \\ 0.304 \end{pmatrix}$$



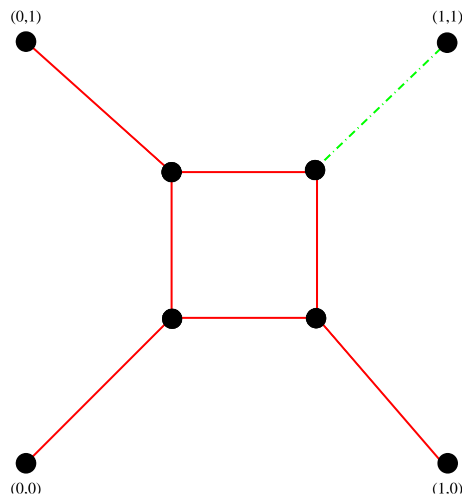
3. Experiment: wie (1) nur ohne Knoten 5

Matrix A:

$$\begin{pmatrix} 3 & -1 & 0 & -1 \\ -1 & 3 & -1 & 0 \\ 0 & -1 & 3 & -1 \\ -1 & 0 & -1 & 3 \end{pmatrix}$$

Lösungen:

$$x = \begin{pmatrix} 1/3 \\ 2/3 \\ 2/3 \\ 1/3 \end{pmatrix}, \; y = \begin{pmatrix} 2/3 \\ 2/3 \\ 1/3 \\ 1/3 \end{pmatrix}$$



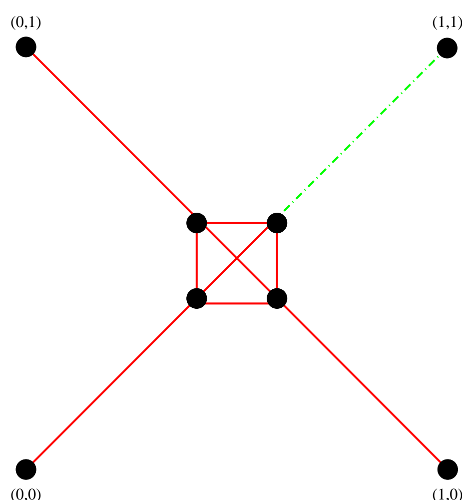
4. Experiment: wie (3), aber mit Diagonalen (1,3), (2,4)

Matrix A:

$$\begin{pmatrix} 4 & -1 & -1 & -1 \\ -1 & 4 & -1 & -1 \\ -1 & -1 & 4 & -1 \\ -1 & -1 & -1 & 4 \end{pmatrix}$$

Lösungen:

$$x = \begin{pmatrix} 0.4 \\ 0.6 \\ 0.6 \\ 0.4 \end{pmatrix}, \; y = \begin{pmatrix} 0.6 \\ 0.6 \\ 0.4 \\ 0.4 \end{pmatrix}$$

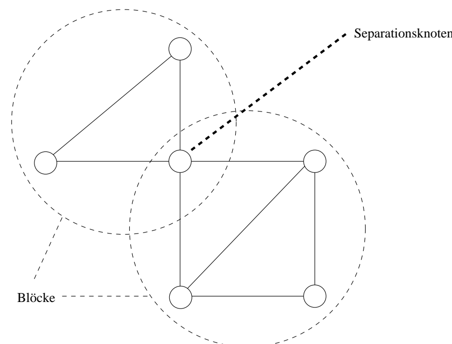


3.3 Einige Definitionen

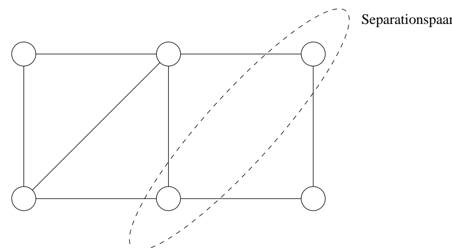
Sei $G = (V, E)$ ein Graph und $G \backslash v := (V', E')$ mit $V' = V \backslash \{v\},\ E' = \{(u, w) \in E \mid u \neq v \neq w\}$.

v ist ein **Separationsknoten**, wenn $G \backslash v$ nicht zusammenhängend ist.

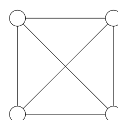
Ein Graph $G = (V, E)$ ohne Separationsknoten ist **2-zusammenhängend**. Die inklusionsminimalen 2-zusammenhängenden Subgraphen heißen **Blöcke** oder **2-Zusammenhangskomponenten** von G.



Zwei Knoten u, v eines 2-zusammenhängenden Graphen $G = (V, E)$ heißen **Separationspaar**, falls $(G \backslash u) \backslash v$ nicht zusammenhängend ist, z.B.

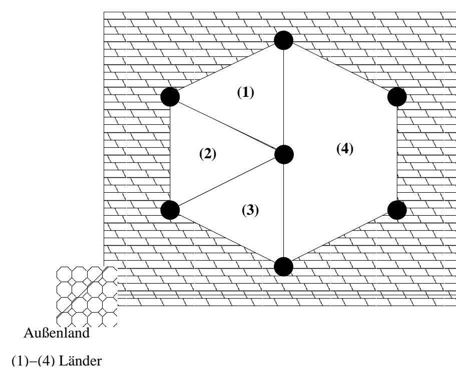


Ein Graph $G = (V, E)$ ohne Separationspaar ist **3-zusammenhängend**, z.B. der K_4:



Eine Zeichnung eines Graphen heißt **planar**, falls sich keine Kanten schneiden (außer an den Endknoten).

Ein Graph heißt **planar**, wenn er eine planare Zeichnung hat, z.B.:



Satz 3.3.1
Sei G ein 3-zusammenhängender planarer Graph, f ein Land in einer planaren Einbettung von G und P eine streng konvexe planare Zeichnung von f. Werden die Knoten von f gemäß P fixiert, so liefert die Schwerpunktmethode eine konvexe planare Zeichnung von G, d.h. jedes Land ist ein konvexes Polygon.

Beipiel 3.3.2
Barycenter-Methode-Abbildung

Beobachtung 3.3.3
Schlechte Auflösung.

Übung: Es wird exponentieller Platz benötigt.

Später: Planare Zeichnungen mit geraden Kanten und nur $O(|V|^2)$ Platz.
(Deshalb verzichten wir hier auf einen Beweis des Satzes.)

3.4 Simulation graphentheoretischer Distanzen durch Kräfte

J. B. Kruskal & J. B. Seary (1980) [KS80]

T. Kamada & S. Kawai (1989) [KK89]

Sei $G = (V, E)$ zusammenhängend und $u, v \in V$

$\delta(u, v)$ = Anzahl der Kanten auf einem kürzesten (u, v)-Weg. (Berechnung: INFORMATIK I: Dijkstra's Algorithmus)

Kraft zwischen u und v:

$$k_{uv}(d(p_u, p_v) - \delta(u, v)).$$

Die potentielle Energie der (u, v)-Feder wird beschrieben durch die Stammfunktion

$$\frac{1}{2}k_{uv}(d(p_u, p_v) - \delta(u, v))^2.$$

Wahl des Steife-Paramters k_{uv}:

$$k_{uv} = \frac{k}{\delta(u, v)^2}$$

mit einer Konstanten k (Stärkere Feder zwischen graphentheoretisch nahen Knoten).
⇒ potentielle Energie in (u, v)-Feder:

$$\frac{k}{2}\left(\frac{d(p_u, p_v)}{\delta(u, v)} - 1\right)^2$$

$\Rightarrow$ potentielle Energie der Zeichnung:

$$\eta = \frac{k}{2} \cdot \sum_{\substack{u,v \in V \\ u \neq v}} \left(\frac{d(p_u, p_v)}{\delta(u,v)} - 1 \right)^2$$

Notwendige Bedingungen für minimales η:

$$\frac{\partial \eta}{\partial x_v} = 0 \qquad \frac{\partial \eta}{\partial y_v} = 0 \qquad \forall v \in V$$

Nichtlineare Gleichungen.

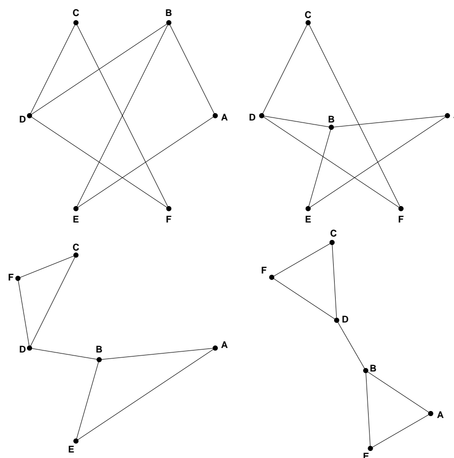
3.4.1 Lösungsmethode von Kamada & Kawai [KK89]

Iteriere
v=Knoten mit größter Kraft

$$\sqrt{\left(\frac{\partial \eta}{\partial x_v}\right)^2 + \left(\frac{\partial \eta}{\partial y_v}\right)^2}$$

Verschiebe v in eine energieminimale Position (bei festen Positionen der anderen Knoten)
bis Verschiebung einen Schwellwert unterschreitet.

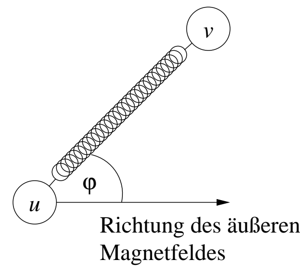
Visualisierung des Prozesses:



3.4.2 Magnetische Felder

K. Sugiyama & K. Misue (1995) [SM95]

Die Felder sind magnetisch und es gibt ein äußeres magnetisches Feld.

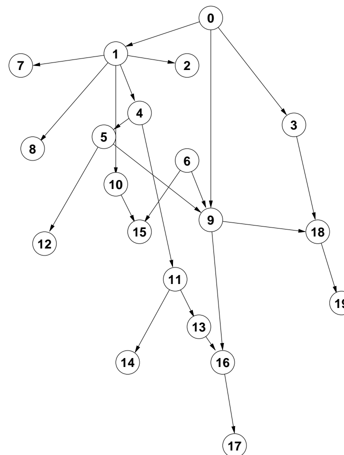


Rotationskraft (Torsion) proportional zu

$$d(p_u, p_v)^\alpha \varphi^\beta \quad (\alpha, \beta \text{ konstant})$$

So kann man z.B. gerichtete Graphen zeichnen, so daß eine Verzugsrichtung eingehalten wird.

Beipiel 3.4.1



3.5 Allgemeine Energie-Funktionen

R. Davidson & D. Harel (1996) [DH96]

minimiere $\eta = \lambda_1\eta_1 + \lambda_2\eta_2 + \cdots + \lambda_k\eta_k$

λ_i : Parameter

η_i : Energiefunktionen als Modelle für Ästhetikkriterien

z.B. DH[1996]:

$$\eta_1 = \sum_{u,v\in V} \frac{1}{(d(p_u, p_v))^2} \quad \text{(Abstoßung von Knoten)}$$

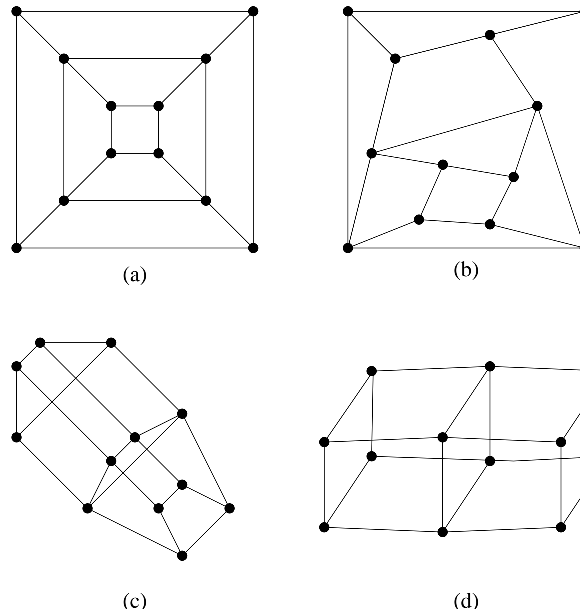
$$\eta_2 = \sum_{u\in V} \left(\frac{1}{r_u^2} + \frac{1}{l_u^2} + \frac{1}{t_u^2} + \frac{1}{b_u^2}\right)$$

$$\eta_3 = \sum_{(u,v)\in E} (d(p_u, p_v))^2 \quad \text{(kurze Kanten)}$$

$\eta_4 =$ Zahl der Kreuzungen

(r_u, l_u, t_u, b_u Abstände zu dem rechten, linken, oberen und unteren Rand der Zeichenfläche.) Approximation eines Energieminimums durch "Simulated Annealing".

Beipiel 3.5.1



Zeichnung eines 12 Knoten Grpahens: (a) ist eine sehr gute Zeichnung, die nicht mit Simulated Annealing erzeugt wurde; (b), (c) und (d) wurden durch den Gebrauch einer Allgemeinen Ernergiefunktion mit unterschiedlichen Koeffzienten erstellt. Der minimale Energiezustand wurde durch Simulated Annealing gefunden (Davidson, Harel [DH96]).

3.5.1 Nebenbedingungen

Verschiedene Varianten implementieren z.B.

- Positionsbedingungen:

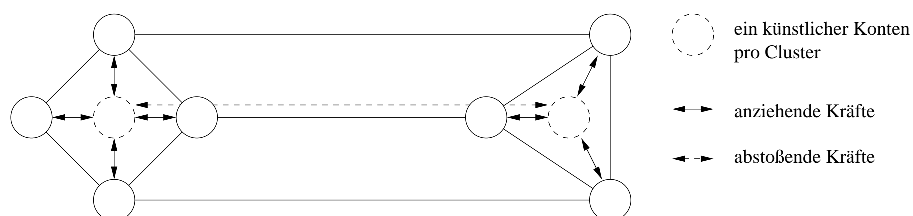
  Einschränkungen des Bereichs, in dem sich Knoten bewegen dürfen.

- Feste Subgraphen:

  Vorgegebene Subgraphen werden bis auf Translation und Rotation immer gleichgezeichnet.

- Geometrisches Clustering:

  Vorgegebene Knotenmengen (Cluster) sollen nahe beieinander gezeichnet werden. Implementierung etwa wie folgt



Kapitel 4

Hierarchische Verfahren

4.1 Grundidee

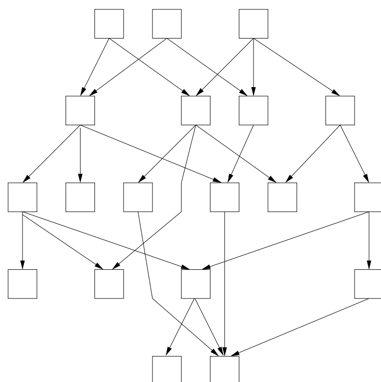
Ursprüngliche Idee: K. Sugiyama, S. Tagawa & M. Toda (1981) [STT81]
und seitdem: Zahlreiche Variationen.

Hier: allgmeine gerichtete Graphen $G = (V, E)$.

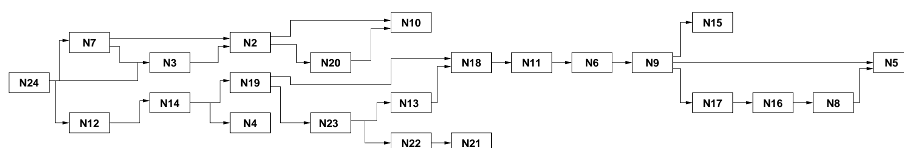
Sehr populär, in fast jeder Graph-Drawing Software implementiert.
Falls G gerichtete Kreise enthält:
Ändere Richtung von möglichst wenigen Kanten, so daß G azyklisch wird.
[Ein NP-schwieriges Problem, Bemerkungen dazu später]
Falls G ungerichtet ist:
Richte die Kanten so, daß keine gerichteten Kreise entstehen.

Ab jetzt: $G = (V, E)$ ist ein *DAG* (= *D*irected *Acyclic Graph*).

Einige Beispielzeichnungen:



Schichtzeichnung konstruiert durch
D-ABDUCTOR (K. Misue)



Schichtzeichnung konstruiert durch Tom Sawyer Toolkit (Tom Sawyer Software)

3 Phasen:

1. Schichtzuweisung

   Zuweisung der Knoten zu Schichten → Zuordnung der y-Koordinaten

2. Kreuzungsminimierung

   Bestimmung von Knotenpermutationen innerhalb der Schichten, so daß wenige Kreuzungen entstehen.

3. Horizontale Koordinatenzuweisung

   → Zuweisung der x-Koordinaten

Schichtzuweisung: Partition $V = V_1 \dot\cup V_2 \dot\cup \ldots \dot\cup V_h$, so daß für alle $(u, v) \in E$ gilt:

$$u \in V_i,\ v \in V_j \Rightarrow i > j$$

h : **Höhe** des geschichteten Digraphen.

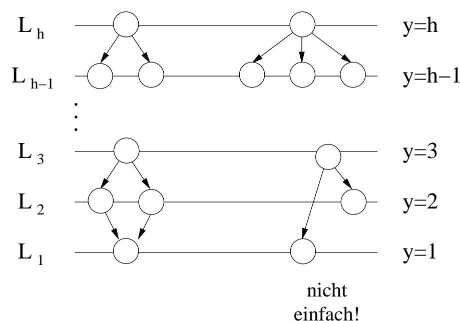
Breite des geschichteten Digraphen:

$$\max_{1 \le i \le h} |V_i|$$

Einfache Schichtung:

$$\forall\ (u, v) \in E :\ u \in V_i,\ v \in V_j \Rightarrow i = j + 1$$

Zeichenkonvention:



Manchmal natürliche Schichtzuweisung, z.B. zeitliche Abläufe.

Sonst: Berechnung

Anforderungen:

1. Kompaktheit: kleine Höhe und Breite
2. einfache Schichtung (wegen Kreuzungsminimierung)
   → erreichbar durch Einführung von künstlichen Knoten
3. wenige künstliche Knoten (bis zu $\approx n^2$ möglich)
   Positiver Einfluß auf spätere Phasen (Laufzeit abhängig von der Knotenanzahl (echt und künstlich))
   - → Knicke nur an künstlichen Knoten
   - → kurze Kanten i.a. besser

4.2 Phase 1: Schichtzuweisung

4.2.1 Längster-Pfad-Schichtung

Zunächst Standardalgorithmus: Topologische Sortierung .

Abbildung: $tsnum: V \rightarrow \{1, \ldots, n\}$ mit:

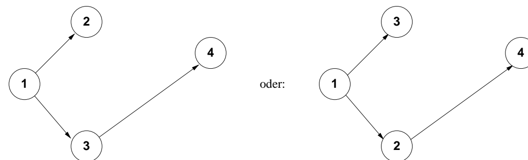
$$(u, v) \in E \Rightarrow tsnum(u) < tsnum(w)$$

4.2.2 Algorithmus Topsort

```
Q = ∅; numinQ=0; newnuminQ=0;
 forall (v ∈ V) indegree[v]=0;
 forall ((u,v) ∈ E) indegree[v]++;
 forall v ∈ V if (indegree[v]==0) {
    Q ⇐ v;
    numinQ++;
}
count=1;
acount=1;   (*) Modifikation
 while (numinQ) {
     for (i=0, i<numinQ, i++) {
         v ⇐ Q;
         tsnum[v] = count++;
         level[v] = acount; // (*) Modifikation
          forall ((v,w) ∈ E) {
             indegree[w]--;
             if (indegree[w]==0) {
             Q ⇐ w; newnuminQ++;
             }
         }
```

```
    }
    numinQ = newnuminQ;
    newnuminQ = 0;
    acount++; // (*) Modifikation
}
// (*) Modifikation
 forall (v ∈ V) level[v] = acount - level[v];
```

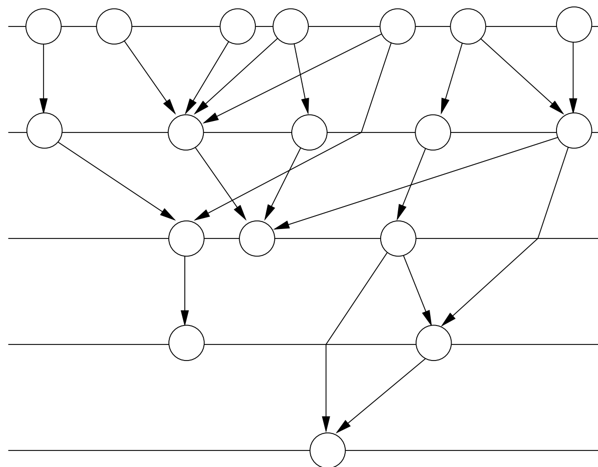
Wir benutzen eine Schlange (queue) Q.
Die Laufzeit ist offensichtlich $O(|V| + |E|)$, aber die topologische Sortierung ist nicht eindeutig, z.B.



Leichte Modifikation (siehe (*) in Algorithmus) ergibt Schichtenzuordnung in gleicher Zeit.

Eigenschaften

- sehr effizient
- Höhe ist Minimum (= Länge eines längsten gerichteten Pfades in G)
- keine Kontrolle über die Breite



4.2.3 Schichtung mit Breitenminimierung

Das Problem bei, minmaler Höhe H die Breite W zu minimieren, ist NP-schwierig.

4.2.4 Beweisskizze

Folgendes Problem ist NP-schwierig:

MULTIPROCESSOR SCHEDULING PROBLEM (MSP)

Gegeben: W Prozessoren, Zeit H,
n Aufgaben mit Laufzeit 1,
DAG $G = (V, E)$ mit $|V| = n$
$(u, v) \in E \Leftrightarrow u$ muß vor v erledigt werden

Frage: Kann alles in Zeit H erledigt werden?
Daraus folgt direkt die NP-Schwierigkeit der Breitenminimierung.

Positiver Aspekt: Heuristiken für MSP sind direkt übertragbar.

4.2.5 Coffman-Graham-Schichtung

Lexikographische Ordnung auf endlichen Teilmengen $\mathbb{N}$. Für $S, T \in \mathbb{N}$, $|S|, |T| < \infty$ gilt:

$S < T$ falls
$S = \varnothing,\ T \neq \varnothing$
oder $S \neq \varnothing,\ T \neq \varnothing,\ max(S) < max(T)$
oder $S \neq \varnothing,\ T \neq \varnothing,\ max(S) = max(T),\ S\backslash\{max(S)\} < T\backslash\{max(T)\}$

Beipiel 4.2.1

$$\begin{aligned} \varnothing &< \{1\} \\ \{1,2,3\} &< \{1,4\} \\ \{2,4,7\} &< \{3,4,7\} \end{aligned}$$

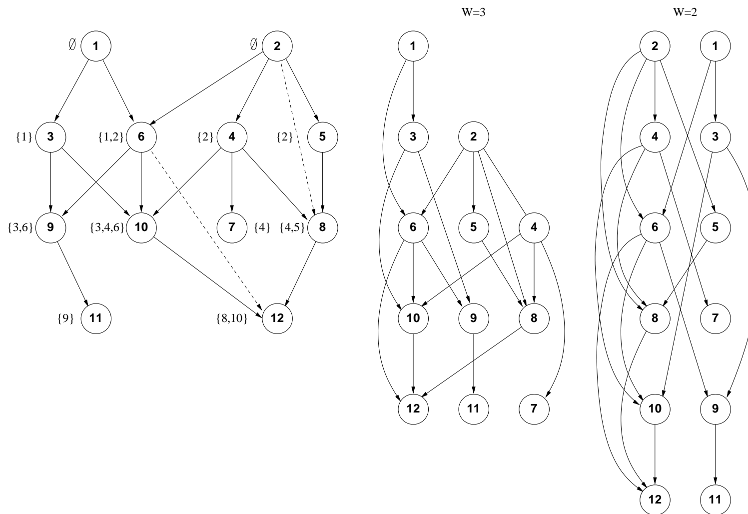
Algorithmus

Input: DAG $G = (V, E)$, $W \in \mathbb{N}$
Output: Schichtung mit Breite $\leq W$

```
G = transitive Reduktion von G;
 for (i=1; i ≤ |V|; i++) π[i] = ∞; // 1. Initialisierung
 for (i=1; i ≤ |V|; i++) {
    wähle v ∈ V mit π[v] = ∞ und
      {π (u) | (u,v) ∈ E } Minimum bzgl. "<"
    π[v] = i;
}
k = 1; V_1 = ∅; U = ∅;
 while (U ≠ V) {
     if |V_k| < W und ein u ∈ V \ U existiert mit
       ∀ (u,w) ∈ E : w ∈ ⋃_{i=1}^{k-1} V_i {
          wähle ein solches u mit π(u) Maximum;
          V_k = V_k ∪ {u};
          U = U ∪ {u};
```

    }
    else {
        $k++;\ \ V_k = \varnothing;$
    }
}

Beipiel 4.2.2



Beide Lösungen sind optimal, aber i.a. gilt „nur“:

Satz 4.2.3
Sei h_{min} die minimale Höhe einer Schichtung mit Breite W. Dann gilt für die Höhe h der Coffmann-Graham-Schichtung:

$$h \leq \left(2 - \frac{2}{W}\right) \cdot h_{min}$$

□

Achtung: Das macht nur Sinn, wenn die Größe künstlicher Knoten sehr viel kleiner als die echter Knoten ist (das ist z.B. der Fall, wenn die Knoten Texte enthalten).

4.2.6 Minimierung der Anzahl künstlicher Knoten

$y_v \in \mathbb{N}$ vertikale Koordinate des Knoten v.

Problem:

$$\begin{array}{ll} \text{minimiere} & \sum_{(u,v)\in E} y_u - y_v \\ \text{so daß} & y_u - y_v \geq 1 \quad (\forall (u,v) \in E) \\ & y_v \in \mathbb{N} \qquad (\forall v \in V) \end{array}$$

Das LP ohne die Ganzzahligkeitsbedingung hat eine ganzzahlige Lösung, der Simplex-Algorithmus findet eine ganzzahlige Lösung, denn die Matrix der Nebenbedingungen ist total unimodular, d.h. die Determinanten aller quadratischen Submatrizen sind 0, 1 oder −1.
Es gibt schnelle spezialisierte Techniken zur Lösung dieses Problems → *Min-Cost-Flow-Algorithmen.*

4.3 Phase 2: Kreuzungsminimierung

Bestimmung von Knotenpermutationen innerhalb der Schichten, so daß so wenige Kreuzungen wie möglich enstehen.

Definition 4.3.1
Ein Graph $G = (V, E)$ *heißt* **bipartit**, *falls es eine Partition* $V = V_1 \dot{\cup} V_2,\ V_1 \cap V_2 = \varnothing$ *gibt, so daß* $\forall\ (u, v) \in E$ *gilt:*

$$u \in V_i,\ v \in V_j \Rightarrow i \neq j$$

4.3.1 Komplexität

Vorbemerkungen

M.R. Garey und D.S.Johnson (1983) [GJ83] haben bewiesen, daß das allgemeine Kreuzungsproblem für $G = (V, E)$ und $K \in \mathbb{N}$

„Gibt es eine Zeichnung von G auf der Ebene mit höchstens K Knotenüberkreuzungen"

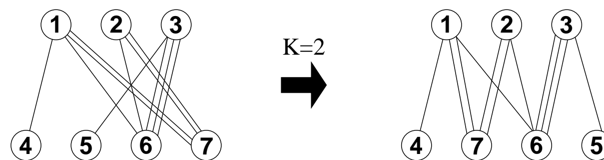
NP-vollständig ist.

Ein „Abfallprodukt" dieser Arbeit ist ein Beweis, daß folgendes Problem NP-vollständig ist:

4.3.2 Bipartite Multigraph Kreuzungszahl (BMKZ)

Eingabe: zusammenhängeder bipartiter Multigraph (parallele Kanten zugelassen) $G = (V_1, V_2, E)$, Zahl $K \in \mathbb{N}$.

Frage: Existiert eine Zeichnung von G im Einheitsquadrat, in der alle $v \in V_1$ auf der Nordgrenze und alle $v \in V_2$ auf der Südgrenze plaziert werden und in der höchstens K Kantenkreuzungen vorkommen?



Satz 4.3.2 (Garey & Johnson) *[GJ83]*
BMKZ ist NP-vollständig.

Beweis: $BMKZ \in NP$: Kreuzungen zählen.

Transformation vom NP-vollständigen Problem:

Optimal Linear Arrangement **(OLA)**

Eingabe: Graph $G = (V, E),\ K \in \mathbb{N}$

Frage: Existiert eine Bijektion

$$f: V \to \{1, \ldots, |V|\} \text{ mit } \sum_{(u,v)\in E} |f(u) - f(v)| \leq K?$$

Sei $(G = (V, E), K)$ eine Instanz von OLA und $V = \{v_1, \ldots, v_n\}$. OBdA sei G zusammenhängend. Konstruktion einer Instanz $(G' = (V_1, V_2, E_1 \cap E_2), K')$ von BMKZ:

$$\begin{aligned} V_1 &= \{u_1, \ldots, u_n\} \\ V_2 &= \{w_1, \ldots, w_n\} \\ E_1 &= \{|E|^2 \text{ Kopien von } (u_i, w_i) \mid 1 \leq i \leq n\} \\ E_2 &= \{(u_i, w_j) \mid i < j \wedge (v_i, v_j) \in E\} \\ K' &= |E|^2 \cdot (|K| - |E|) + (|E|^2 - 1) \end{aligned}$$

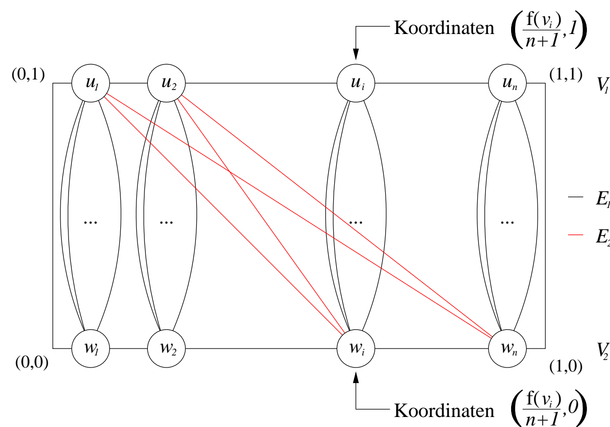
Es gilt:

- Die Konstruktion ist in Polynomialzeit möglich.
- G' ist auch zusammenhängend.

Wir zeigen:

(G, K) ist eine JA-Instanz für OLA
gdw.
(G', K') ist eine JA-Instanz für BMKZ

"⇒" Gegeben f mit $\sum_{(u,v)\in E} |f(u) - f(v)| \leq K$. Konstruktion eines Layouts von G':



Jede rote Kante $(u_i, w_j) \in E_2$ kreuzt $(|f(v_j) - f(v_i)| - 1) \cdot |E|^2$ schwarze Kanten aus E_1.

⇒ Gesamtzahl der rot-schwarz-Kreuzungen

$$\sum_{(u,v)\in E} (|f(u) - f(v)| - 1) \cdot |E|^2 \leq (K - |E|) \cdot |E|^2$$

Gesamtzahl der Kreuzungen zwischen roten Kanten aus E_2 ist höchstens

$$|E|^2 - 1.$$

$\Rightarrow$ Gesamtzahl aller Kreuzungen ist höchstens

$$(K - |E|) \cdot |E|^2 + |E|^2 - 1 = K'.$$

"$\Leftarrow$" Gegeben Zeichnung mit höchstens K' Kreuzungen. Konstruktion von Bijektionen

$$f_1 : V_1 \rightarrow \{1, 2, \dots, n\}$$
$$f_2 : V_2 \rightarrow \{1, 2, \dots, n\}$$

durch Numerierung von Westen nach Osten. Es muß gelten $f_1(u_i) = f_2(w_i) \quad \forall 1 \leq i \leq n$, sonst würde eine Kreuzung von zwei Parallelbündeln bereits $|E|^4$ Kreuzungen schwarz/schwarz erzeugen, Widerspruch zu #Kreuzungen$\leq K'$.

$\Rightarrow$ Die Zeichnung ist im wesentlichen die vorhin gezeigte.

$\Rightarrow$ Jede rote Kante $(u_i, w_j) \in E_2$ wird wenigstens

$$(|f_1(v_i) - f_1(v_j)| - 1) \cdot |E|^2$$

mal gekreuzt. (So oft mit schwarzen, hinzu kommen andere rote Kanten.)

$\Rightarrow \sum_{(u,v)\in E} (|f_1(u) - f_1(v)| - 1) \cdot |E|^2 \leq K' = (K - |E|) \cdot |E|^2 + (|E|^2 - 1)$

$\Rightarrow \sum_{(u,v)\in E} (|f_1(u) - f_1(v)| - 1) \leq K - |E|$

$\Rightarrow \sum_{(u,v)\in E} |f_1(u) - f_1(v)| \leq K$

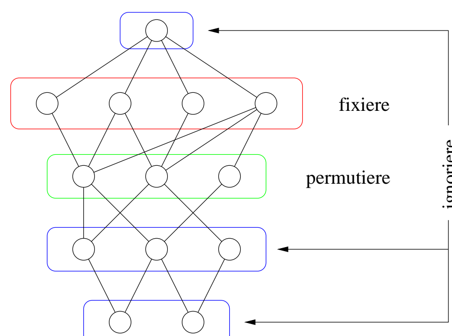
□

Aus diesem Satz folgern unser Lehrbuch (und viele Aufsätze), daß Kreuzungsminimierung in geschichteten Graphen NP-schwierig ist, insbesondere schon für zwei Schichten. Das ist nicht bewiesen, aber vermutlich richtig. (Forschungs-?) Aufgabe: Beweis!
Wir gehen als Arbeitshypothese davon aus.

Grundprinzip der gängigen Heuristiken

"Layer-by-Layer-Sweep": Traversierung der Schichten runter und rauf :

Betrachte nur zwei benachbarte Schichten, fixiere die Permutation der zuvor besuchten Schicht und permutiere auf aktueller Schicht mit dem Ziel weniger Kreuzungen:



neues Problem:

4.3.3 2-Schichten-Kreuzungsminimierung mit einer fixierten Schicht (2SKM1F)

Zugehöriges Entscheidungsproblem (E2SKM1F) wie BKMZ, aber

- Positionen auf der Nordgrenze fixiert (Teil der Eingabe)
- einfacher Graph

Schlechte Nachricht:

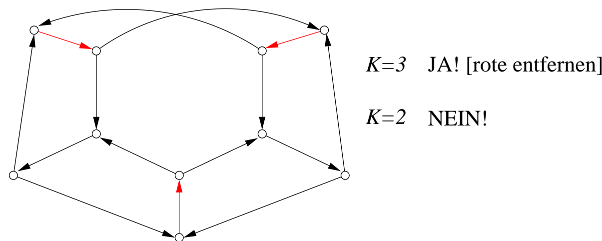
Satz 4.3.3 ***(P. Eades & M. Wormald (1994) [EW94])***
E2SKM1F ist NP-vollständig.

Beweis: E2SKM1F $\in$ NP: klar (zählen).
Transformation vom NP-vollständigen Problem

FEEDBACK ARC SET **(FAS)**

Eingabe: Digraph $D = (U, B)$, $K \in \mathbb{N}$

Frage: Existiert $A \subseteq B$ mit $|A| \leq K$, so daß $D' = (U, B \backslash A)$ azyklisch ist? (A heißt "feedback arc set")



Gegeben Instanz (D, K) von FAS, Konstruktion einer Instanz $(G = (V_1, V_2, E), M)$ von E2SKM1F:

$$V_1 = \bigcup_{b \in B} C(b) \text{ mit } C(b) = \underbrace{\{c_1(b), c_2(b), \ldots, c_6(b)\}}_{\text{"Klumpen" von 6 Knoten}}$$

$$V_2 = U$$

E : Für jedes Paar $u \in U$, $b \in B$ zwei Kanten in E nämlich

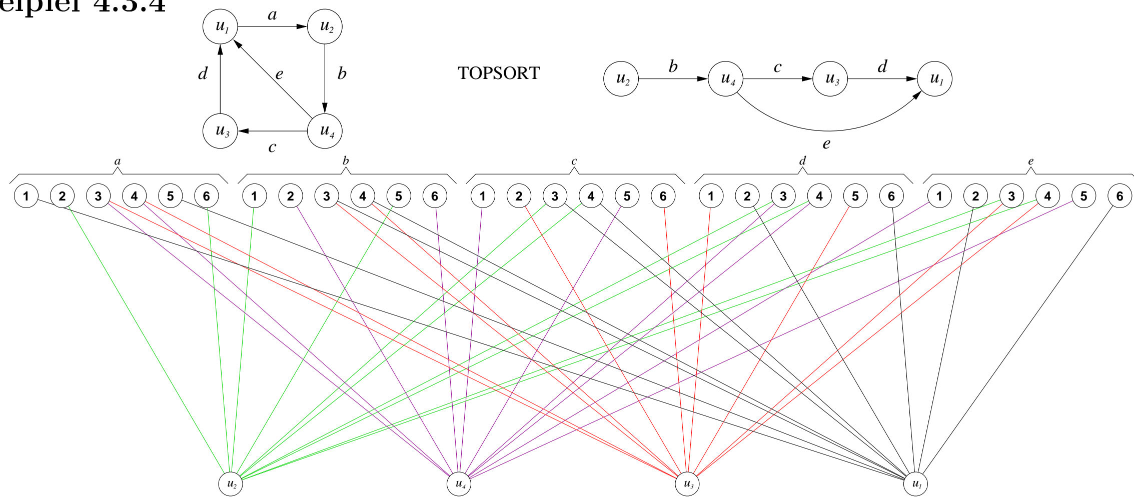
$(u, c_1(b)), (u, c_5(b))$, falls $b = uv$

$(u, c_2(b)), (u, c_6(b))$, falls $b = vu$

$(u, c_3(b)), (u, c_4(b))$, falls u nicht inzident mit b

$$M = 4\binom{|B|}{2}\binom{|U|}{2} + |B|\binom{|U|-2}{2} + 4|B|(|U|-2) + |B| + 2K$$

Beipiel 4.3.4



$$|U| = 4 \quad |B| = 5 \quad K = 1$$

$$\begin{aligned} M &= 4\binom{5}{2}\binom{4}{2} + 5\binom{2}{2} + 4 \cdot 5 \cdot 2 + 5 + 2 \\ &= 4 \cdot 10 \cdot 6 + 5 + 4 \cdot 5 \cdot 2 + 5 + 2 \\ &= 240 + 5 + 40 + 5 + 2 \\ &= 292 \end{aligned}$$

Sei π_1 eine beliebige Permutation von V_1, in der jeder Klumpen zusammen und in natürlicher Ordnung ist, d.h. $\pi_1(c_i(b)) < \pi_1(c_j(b))$ für $1 \leq i < j \leq 6$ und $b \in B$.

Wir zeigen:

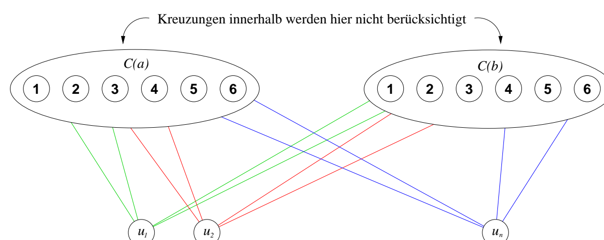
D hat FAS der Größe höchstens K
gdw.
$\exists$ Permutation π_2 von V_2, die höchstens M Kreuzungen erzeugt.

Lemma 4.3.5
Ist π_2 eine Permutation von V_2 und $B' = \{(u, v) \in B \mid \pi_2(u) > \pi_2(v)\}$, so gibt es

$$4\binom{|B|}{2}\binom{|U|}{2} + |B|\binom{|U|-2}{2} + 4|B|(|U|-2) + |B| + 2|B'|$$

Kreuzungen.

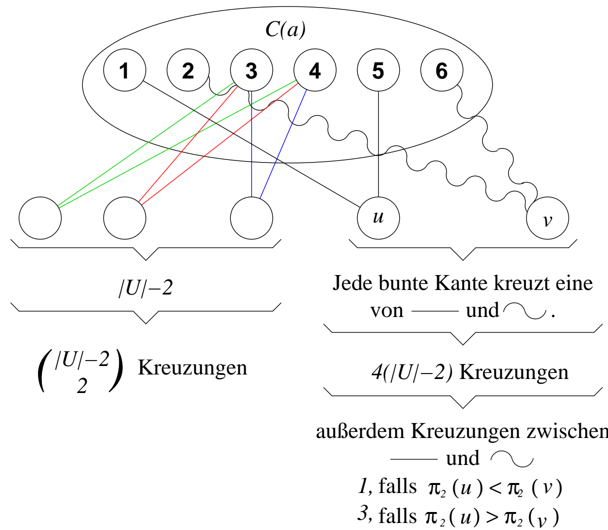
Beweis: Kreuzungen zwischen verschiedenen Klumpen
$a, b \in B$, $a \neq b$



$4\binom{|U|}{2}$ Kreuzungen zwischen Kanten, die inzident mit $C(a)$ und $C(b)$ sind

$\Rightarrow 4\binom{|B|}{2}\binom{|U|}{2}$ Kreuzungen zwischen Kanten verschiedener Klumpen. $(*)$

Kreuzungen innerhalb eines Klumpens
$a = (u, v) \in B$



$\Rightarrow$ Gesamtzahl der Kreuzungen zwischen Kanten von $C(a)$:

$$\binom{|U|-2}{2} + 4(|U|-2) + 1, \text{ falls } \pi_2(u) < \pi_2(v)$$
$$\binom{|U|-2}{2} + 4(|U|-2) + 3, \text{ falls } \underbrace{\pi_2(u) > \pi_2(v)}_{\text{Definition von} B'}$$

$\Rightarrow$ Summe über alle Klumpen

$$|B|\binom{|U|-2}{2} + 4|B|(|U|-2) + |B| + 2|B'| \qquad (**)$$

$\Rightarrow$ insgesamt $(*) + (**)$ wie behauptet.

□

weiter im Beweis des Satzes:

"$\Rightarrow$" D hat FAS B' der Größe höchstens K ($|B'| = K$)
$\Rightarrow D' = (U, B \backslash B')$ ist azyklisch.
Bestimme π_2 mittels TOPSORT.
$\stackrel{\text{Lemma}}{\Rightarrow}$ höchstens M Kreuzungen.

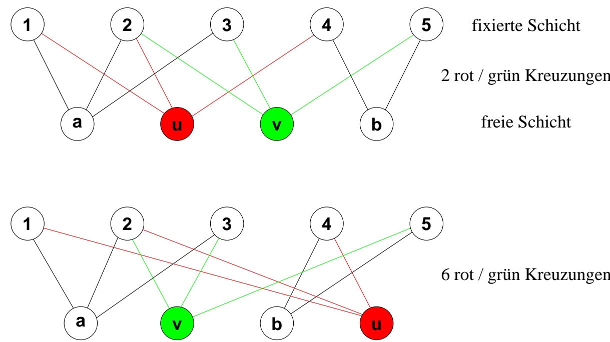
"$\Leftarrow$" $\exists$ Permutation π_2 von V_2, die höchstens M Kreuzungen erzeugt.
Sei $B' = \{(u, v) \in B \mid \pi_2(u) > \pi_2(v)\}$.
$\stackrel{\text{Lemma}}{\Rightarrow} |B'| \leq K$ und $D' = (U, B \backslash B')$ ist azyklisch, d.h. B' ist ein FAS.

□

Plan: Vorstellung und (experimentelle) Analyse verschiedener Heuristiken und (trotzdem) exakter Algorithmus für 2SKM1F.

4.3.4 Zählen der Kreuzungen in einer gegebenen 2-Schichten-Zeichnung

Beobachtung 4.3.6



Die Kreuzungszahl zwischen Kanten inzident mit $u, v \in V_2$ ist nur von deren relativer Position abhängig.

- $\forall\, u, v \in V_2,\ u \neq v$:

$$c_{uv} := \text{Anzahl der Kreuzungen zwischen inzidenten Kanten, falls } u \text{ links von } v \ (\pi_2(u) < \pi_2(v))$$
$$c_{vu} := \text{Anzahl der Kreuzungen zwischen inzidenten Kanten, falls } u \text{ rechts von } v \ (\pi_2(u) > \pi_2(v))$$

- $u = v$: $c_{vv} := 0$

Vollständige Matrix im Beispiel:

$C =$

	a	b	u	v
a	0	0	3	1
b	6	0	5	4
u	4	0	0	2
v	6	1	6	0

Bezeichnung

$$\text{Kreuz}\,(G, \pi_1, \pi_2) = \text{Anzahl der Kreuzungen in einer Zeichnung von } G = (V_1, V_2, E), \text{ in der } V_1 \text{ gemäß } \pi_1 \text{ und } V_2 \text{ gemäß } \pi_2 \text{ permutiert ist.}$$

$$\text{MinKreuz}\,(G, \pi_1) = \min_{\substack{\pi_2 \text{ Permu-}\\ \text{tation von } V_2}} \text{Kreuz}(G, \pi_1, \pi_2)$$

Lemma 4.3.7
(a) $\mathit{Kreuz}(G, \pi_1, \pi_2) = \sum_{\substack{u,v \in V_2\\ \pi_2(u) < \pi_2(v)}} c_{uv}$

(b) $\mathit{MinKreuz}(G, \pi_1) \geq \sum_{u,v \in V_2} \min(c_{uv}, c_{vu})$

Beweis: trivial

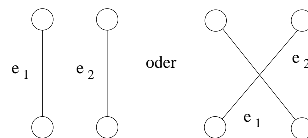
□

Zeit:

- Aufbau der Matrix: $O(|V_2|^2|E|)$
- Addieren der Kreuzungen: $O(|V_2|)$

Wenn wir nur die Zahl der Kreuzungen (und nicht die Matrix) wissen wollen:

naives Zählen: Zeit $O(|E|^2)$



besser:

- J. L. Bentley & T. Ottmann (1979) [BO79]

$$O(|E| \cdot \log |E| + \underbrace{\text{Kreuz}(G, \pi_1, \pi_2)}_{=O(|E|^2)})$$

  mit expliziter Ausgabe einer Liste aller Kreuzungen → kaum interessanter als naives Zählen

- B. Chazelle (1986) [Cha86]

$$O(|E|^{1.695} \cdot \log |E|)$$

- (nur zählen) V. Waddle & A. Malhotra (1999) [WM99] (kompliziert!)

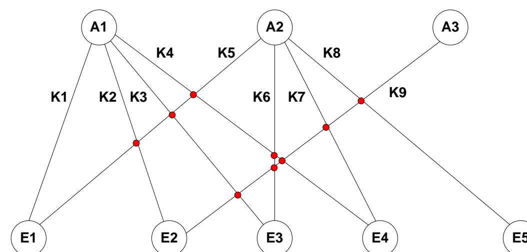
$$O(|E| \cdot \log |E|)$$

bisher am besten:

4.3.5 Algorithmus von W. Barth (2000) [Bar00]

Idee: Sortieren durch Einfügen

Beipiel 4.3.8

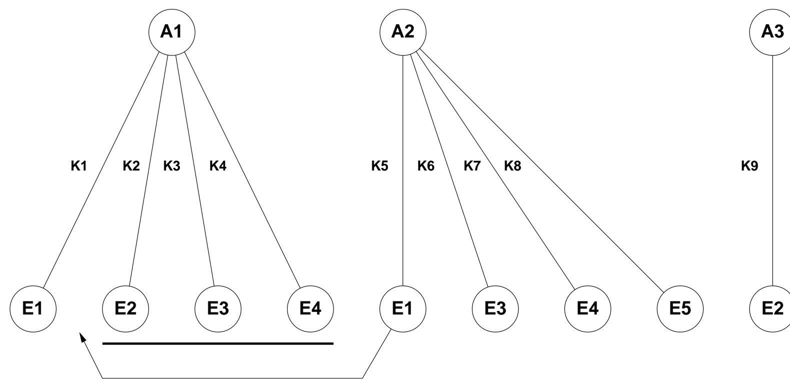


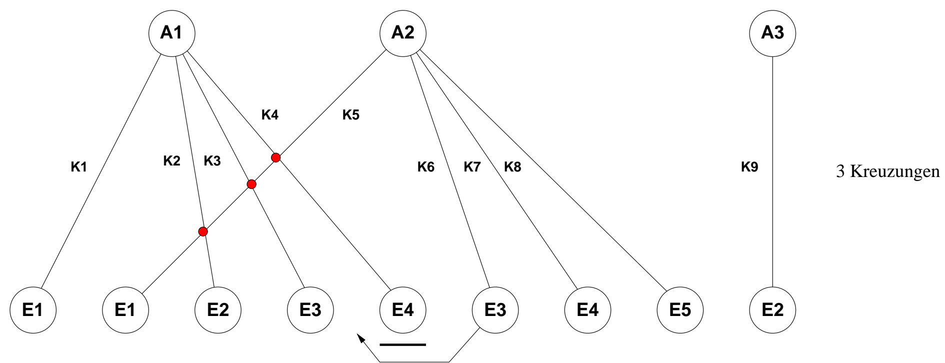
9 Kreuzungen

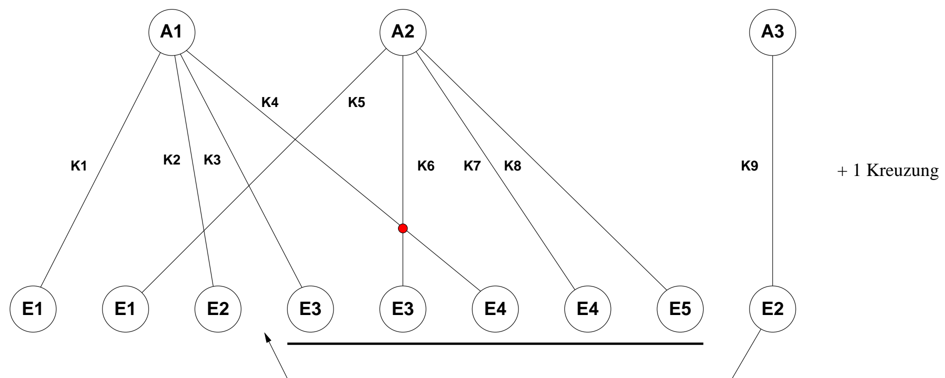
1. lexikographische Sortierung der Kanten
2. Eintragen der Endknoten (E)-Folge in ein Feld:

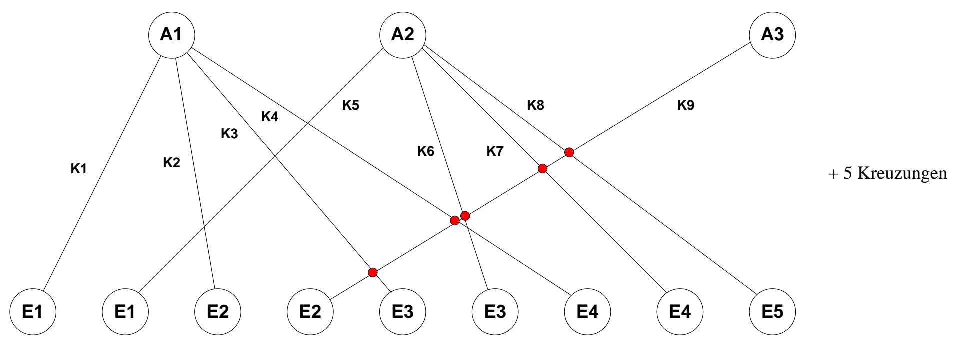
Kante	K1	K2	K3	K4	K5	K6	K7	K8	K9
Anfangsknoten	A1	A1	A1	A1	A2	A2	A2	A2	A3
Endknoten	E1	E2	E3	E4	E1	E3	E4	E5	E2

3. Sortieren durch Einfügen:









= 9 Kreuzungen

Kreuzungszahl:

$$\begin{aligned} \text{Kreuz}(G, \pi_1, \pi_2) &= \text{Anzahl der Vorrückpositionen} \\ &= \text{Anzahl der Inversionen der } E\text{-Folge} \end{aligned}$$

Erklärung: Wenn ein E-Knoten eingefügt wird, stehen die höheren unmittelbar vor ihm; das sind diejenigen, deren Kanten gekreuzt werden.

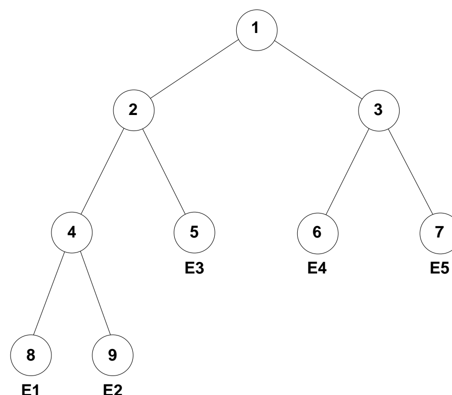
Laufzeit: $O(|E|^2)$ ist keine Verbesserung, aber

4.3.6 Verschnellerung durch bessere Datenstruktur

Akkumulationsbaum

(Modifikation einer Idee von Waddle & Malhotra (1999) [WM99] in ihrem Kontext)

1. Aufbau des Baums



gespeichert als Heap, d.h. in einem Feld

interne Knoten				$E3$	$E4$	$E5$	$E1$	$E2$
1	2	3	4	5	6	7	8	9

Für Knoten i:

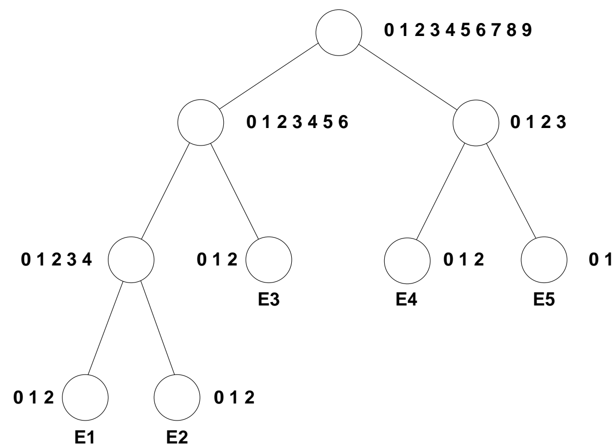
$$\text{Adresse des Vaters} = \begin{cases} \frac{i}{2} & \text{falls } i \text{ gerade} \\ \frac{i-1}{2} & \text{falls } i \text{ ungerade} \end{cases}$$

Einträge in Knoten:

- Blätter (E-Knoten): Anzahl der inzidenten Kanten
- interne Knoten: Summe der beiden Kinder

2. Aufbau der Einträge durch „Einfügen“ der Kanten in lexikographischer Reihenfolge, dabei Zählen der „überholten“, d.h. „rechts liegengelassenen“ Kanten:

$K1$	$K2$	$K3$	$K4$	$K5$	$K6$	$K7$	$K8$	$K9$
0	0	0	0	3	1	0	0	5



Laufzeit: $O(|E| \cdot \log |E|)$

C-Programm:

```
#include <stdio.h>

/* Zaehlen der Kreuzungen in einem 2-Schicht Graphen   */
/* =================================================== */
/* Die Kanten sind bereits in der Eingabe so sortiert, */
/* wie in der Vorlesung angegeben. Wir waehlen der     */
/* Einfachheit halber den kleinsten vollstaendigen     */
/* Binaerbaum mit hinreichend vielen Blaettern.        */

main() {
  int n,m,i,nc,size,index,fa;
  int *e,*nin;

  scanf("%d",&n); /* Zahl der Knoten in V_2 */
  scanf("%d",&m); /* Zahl der Kanten */

  e = (int *) malloc(m*sizeof(int)); /* V_2 Endknoten */

  for (i=0; i<m; i++) scanf("%d",e+i);

  nc = 0; /* Zahl der Kreuzungen */
  fa = 1;
  while (fa<n) fa *= 2;
  size = 2*fa - 1; /* Anzahl der Baumknoten */
  fa -= 1; /* "first adress:" Indexinkrement im Baum */

  nin = (int *) malloc(size*sizeof(int)); /* Grad/Gradsummen */

  for (i=0; i<size; i++) nin[i] = 0;
```

```
  for (i=0; i<m; i++) { /* Einfuegen der Kanten */
    index = e[i] + fa;
    nin[index]++;
    while (index>0) {
      if (index%2) nc += nin[index+1]; /* neue Kreuzungen */
      index = (index - 1)/2;
      nin[index]++;
    }
  }

  printf("Number of crossings: %d\n",nc);
}

Beispiel aus der Vorlesung
==========================
5
9
0
1
2
3
0
2
3
4
1

Aufruf
======
crosscount < exampl
Number of crossings: 9
```

4.3.7 Heuristiken, die die Matrix C benutzen

Greedy-Insert **(Eades & Kelly (1986) [EK86])**

- Vergebe Positionen der Reihe nach von links nach rechts.
- Wähle als jeweils nächsten Knoten denjenigen, der die wenigsten Kreuzungen mit seinen Vorgängern erzeugt.

Laufzeit: $O(|V_2|^2)$

Greedy-Switch **(Eades & Kelly (1986) [EK86])**

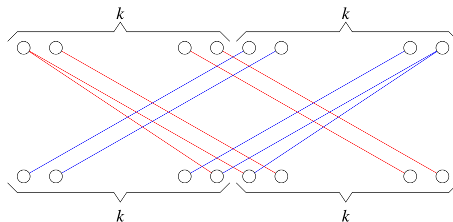
- Starte mit beliebiger Permutation π_2.

- Wiederhole:
  - Traversiere benachbarte Knotenpaare $u, v \in V_2$ von links nach rechts; vertausche Positionen von u und v, falls $c_{uv} > c_{vu}$

  bis keine Kreuzungszahlreduktion erfolgt.

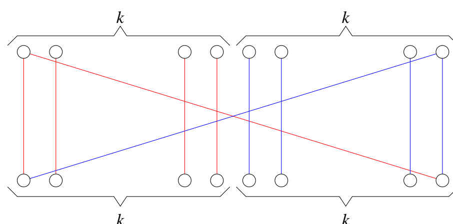
Laufzeit: $O(|V_2|^2)$ (höchstens $|V_2|$ Traversierungen mit jeweils $O(|V_2|)$ Aufwand).

Pathologisches Beispiel



$k^2 + 2(k-1) = O(k^2) = O(n^2)$ Kreuzungen

Optimal:



$$
\begin{aligned}
2(2k-2)+1 &= 4k-3 = O(k) \\
&= O(n) \text{ Kreuzungen}
\end{aligned}
$$

Split (Eades & Kelly (1986) [EK86])

Analog zu Quicksort.

- wähle Pivotelement $p \in V_2$
- Plaziere alle anderen Knoten $u \in V_2$
  - links von p, falls $c_{up} < c_{pu}$
  - rechts von p, falls $c_{up} \geq c_{pu}$
- rekursive Anwendung auf Menge der linksplazierten und Menge der rechtsplazierten Knoten.

Laufzeit: $O(|V_2|^2)$ im worst case, $O(|V_2| \log |V_2|)$ im average case.

Bei allen obigen kommt die Berechnung der Matrix C hinzu.

4.3.8 Effiziente Heuristiken ohne Berechnung von C

Barycenter **(Sugiyama, Tagawa, Toda (1981) [STT81])**

- Berechne für alle $u \in V_2$:

$$avg(u) = \frac{1}{\Delta(u)} \sum_{v \in N(u)} \pi_1(v)$$

  mit $N(u) = \{v \in V_1 \mid (v, u) \in E\}$ Menge der Nachbarn von u.

- Sortiere die $u \in V_2$ mit Sortierschlüssel $avg(u)$, bei Gleichheit beliebige Reihenfolge.

$avgKreuz(G, \pi_1) =$ Zahl der Kreuzungen mit Barycentermethode.

Laufzeit: $O(|E| + |V_2| \cdot \log |V_2|)$.

Median **(Eades & Wormald (1994)[EW94])**

Wie Barycenter, nur Median statt Durchschnitt.

$$\begin{aligned}
&N(u) = \{v_1, v_2, \ldots, v_j\} \text{ und } \pi_1(v_1) < \pi_1(v_2) < \ldots < \pi_1(v_j) \\
&N(u) = \emptyset : med(u) = 0 \\
&N(u) \neq \emptyset : med(u) = \pi_1(v_{\lceil \frac{j}{2} \rceil})
\end{aligned}$$

Sortieren - bei Gleichheit ungerader Grad links von geradem Grad, sonst beliebig. $\rightarrow$ wichtig für die Analyse.

$medKreuz(G, \pi_1) =$ Zahl der Kreuzungen mit Median-Methode.

Laufzeit: $O(|E|)$. (Medianbestimmung: $O(|V_2|) \rightarrow$ Informatik I)

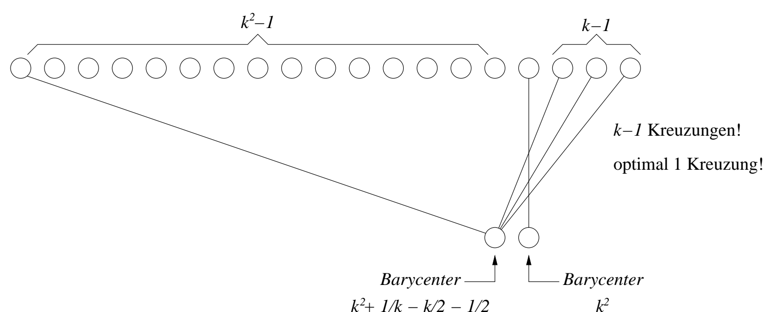
Satz 4.3.9
Falls $minKreuz(G, \pi_1) = 0$, so gilt

$$avgKreuz(G, \pi_1) = medKreuz(G, \pi_1) = 0.$$

Beweis: Übung.

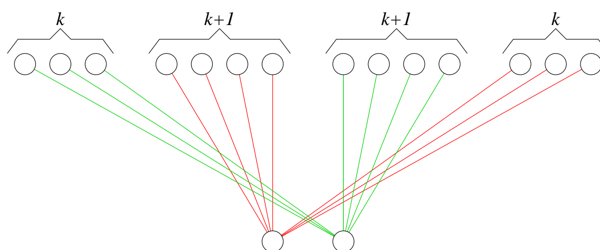
Pathologische Beispiele

(1) Barycenter $k \geq 2$ (hier $k = 4$)



$$\begin{aligned}
&\frac{1}{k}\left(1+(k^2+1)+(k^2+2)+\ldots+(k^2+k-1)\right)\\
=&\frac{1}{k}\left(1+(k-1)k^2+\sum_{i=1}^{k-1} i\right)\\
=&\frac{1}{k}\left(1+k^3-k^2+\frac{k(k-1)}{2}\right)\\
=&\frac{1}{k}+k^2-k+\frac{k-1}{2}\\
=&\frac{1}{k}+k^2-\frac{k}{2}-\frac{1}{2}<k^2 \quad \text{für } k\geq 2
\end{aligned}$$

(2) MEDIAN (hier $k = 3$)



$2k(k+1)+k^2$ Kreuzungen, optimal $(k+1)^2$ Kreuzungen.

Lemma 4.3.10

(1) Zu jedem $k \geq 2$ gibt es einen Digraphen $G = (V_1, V_2, E)$ mit $|V_1| = k^2 + k$ und $|V_2| = 2$ und eine Permutation π_1 von V_1, so daß

$$\frac{avgKreuz(G, \pi_1)}{minKreuz(G, \pi_1)} = \Omega(\sqrt{|V_1|}).$$

(2) Zu jedem k gibt es einen Digraphen $G = (V_1, V_2, E)$ mit $|V_1| = 4k + 2$ und $|V_2| = 2$ eine Permutation π_1 von V_1, so daß

$$\frac{medKreuz(G, \pi_1)}{minKreuz(G, \pi_1)} \geq 3 - O(\tfrac{1}{|V_1|}).$$

Beweis: Wegen pathologischem Beispiel

$$(1) \qquad k = \Theta(\sqrt{|V_1|})$$

$$(2) \qquad \frac{medKreuz(G,\pi_1)}{minKreuz(G,\pi_1)} = \frac{2k\cdot(k+1)+k^2}{(k+1)^2} = \frac{2k^2+2k+k^2}{k^2+2k+1} = \frac{3k^2+2k}{k^2+2k+1} = 3 - O(\tfrac{1}{k})$$

□

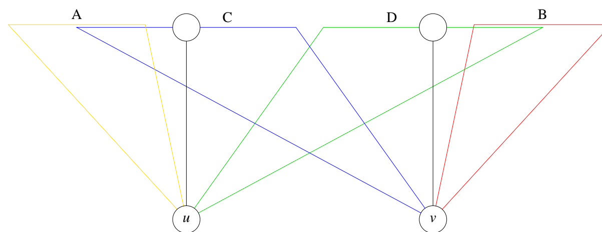
Für Median ist die Schranke scharf!

Satz 4.3.11
Für alle $G = (V_1, V_2, E)$ und alle Permutationen π_1 von V_1 gilt

$$medKreuz(G,\pi_1) \leq 3 \cdot minKreuz(G,\pi_1).$$

Beweis: Seien $u, v \in V_2$ und MEDIAN plaziere u links von v. Sei

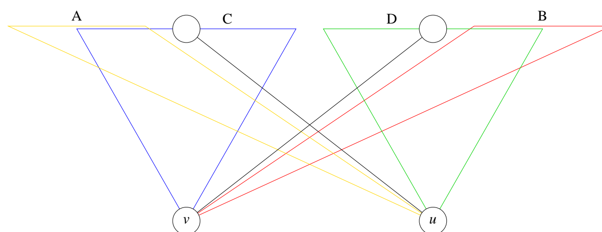
$$A = \{(w,u) \in E \mid \pi_1(w) < med(u)\},$$
$$B = \{(w,v) \in E \mid \pi_1(w) > med(v)\},$$
$$C = \{(w,v) \in E \mid \pi_1(w) < med(v)\},$$
$$D = \{(w,u) \in E \mid \pi_1(w) > med(u)\}.$$



Sei $a = |A|$, $b = |B|$, $c = |C|$, $d = |D|$. Dann gilt

$$c_{uv} \leq ac + cd + bd + c + d \qquad (1)$$

Sei $\varepsilon = \begin{cases} 0, & \text{falls } med(u) = med(v) \\ 1, & \text{sonst} \end{cases}$



Dann gilt: $c_{vu} \geq ab + a + b + \varepsilon \qquad (2)$

Weiterhin gilt:

$$\begin{aligned} a &= d \text{ falls } \Delta(u) \text{ ungerade ist} \\ a+1 &= d \text{ falls } \Delta(u) \text{ gerade ist} \\ c &= b \text{ falls } \Delta(v) \text{ ungerade ist} \\ c+1 &= b \text{ falls } \Delta(v) \text{ gerade ist} \end{aligned}$$

$\Rightarrow d \leq a + 1$ und $c \leq b$

$$\begin{aligned} \overset{(1)}{\Rightarrow} c_{uv} &\leq ab + b(a+1) + b(a+1) + b + a + 1 \\ &= ab + ab + b + ab + b + b + a + 1 \\ &= 3ab + a + 3b + 1 \qquad (3) \end{aligned}$$

Wir zeigen: $c_{uv} \leq 3c_{vu}$.

Annahme: $c_{uv} > 3c_{vu}$. Wir haben

$$\begin{aligned} 3ab + a + 3b + 1 &\overset{(3)}{\geq} c_{uv} \\ &\overset{(\text{Ann.})}{>} 3c_{vu} \\ &\overset{(2)}{\geq} 3ab + 3a + 3b + 3\varepsilon \end{aligned}$$

$$\begin{aligned} \Rightarrow 0 &> 3a - a + 3\varepsilon - 1 \\ &= \underbrace{2a}_{\in \mathbb{N}} + \underbrace{3\varepsilon}_{\in \mathbb{N}} - 1 \end{aligned}$$

$\Rightarrow a = \varepsilon = 0 \Rightarrow d \leq 1 \Rightarrow A = \emptyset,\ |D| \leq 1 \Rightarrow \Delta(u) \leq 2.$

Fall 1: $\Delta(u) = 1 \Rightarrow d = 0$
Einsetzen in (1)

$$c_{uv} \leq ac + cd + bd + c + d = c$$

und in (2)

$$c_{vu} \geq ab + a + b + \varepsilon = b$$

$\Rightarrow b \geq c \overset{(1)}{\geq} c_{uv} \overset{(\text{Ann.})}{>} 3c_{vu} \geq 3b$ Widerspruch.

Fall 2: $\Delta(u) = 2 \Rightarrow d = 1$
Wegen $\varepsilon = 0$ gilt $med(u) = med(v)$.

Regel: Falls $\Delta(u)$ gerade und $\Delta(v)$ ungerade, so gilt $\pi_2(v) < \pi_2(u)$. Also ist $\Delta(v)$ gerade und deshalb

$$c = b - 1.$$

Einsetzen in (1)

$$\begin{aligned} c_{uv} &\leq ac + cd + bd + c + d \\ &= b - 1 + b + b - 1 + 1 \\ &= 3b - 1 \end{aligned}$$

und in (2)

$$\begin{aligned} c_{vu} &\leq ab + a + b + \varepsilon \\ &= b \end{aligned}$$

Also,

$$3b - 1 \overset{(1)}{\geq} c_{uv} \overset{(\text{Ann.})}{>} 3c_{vu} \overset{(2)}{\geq} 3b \text{ Widerspruch!}$$

Also haben wir $c_{uv} \leq 3c_{vu}$ für alle Paare $u, v \in V_2$

$$\Rightarrow c_{uv} \leq 3 \cdot \min(c_{uv}, c_{vu})$$

$$\begin{aligned} \Rightarrow medKreuz(G, \pi_1) &= \sum_{\pi_2(u) < \pi_2(v)} c_{uv} \\ &\leq 3 \cdot \sum_{\pi_2(u) < \pi_2(v)} \min(c_{uv}, c_{vu}) \\ &\leq 3 \cdot minKreuz(G, \pi_1) \end{aligned}$$

□

4.3.9 Exakte Kreuzungsminimierung

Sei $n = |V_2|$. π_2 wird kodiert durch $\chi^{\pi_2} \in \{0,1\}^{\binom{n}{2}}$ mit

$$\chi_{ij}^{\pi_2} = \begin{cases} 0, & \text{falls } \pi_2(j) > \pi_2(i) \\ 1, & \text{falls } \pi_2(i) > \pi_2(j) \end{cases} \quad (1 \leq i < j \leq n).$$

Gegeben χ^{π_2}, so ist die Zahl der Kreuzungen

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (c_{ij}\chi_{ij}^{\pi_2} + c_{ji}(1 - \chi_{ij}^{\pi_2})) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (c_{ij} - c_{ji})\chi_{ij}^{\pi_2} + \underbrace{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ji}}_{=C}$$

Wir wollen diejenigen $x \in \{0,1\}^{\binom{n}{2}}$ charaktersieren, die Permutationen von $\{1, 2, \ldots, n\}$ beschreiben:

$$\begin{array}{l} 0 \leq x_{ij} \leq 1 \quad (1 \leq i < j \leq n) \\ x_{ij} \text{ ganzzahlig} \quad (1 \leq i < j \leq n) \end{array}$$

Ausschließen von „i vor j vor k vor i“

$$x_{ij} + x_{jk} + (1 - x_{ik}) \leq 2 \Leftrightarrow x_{ij} + x_{jk} - x_{ik} \leq 1 \quad (1 \leq i < j < k \leq n)$$

umgekehrt

$$(1 - x_{ij}) + (1 - x_{jk}) + x_{ik} \leq 2 \Leftrightarrow -x_{ij} + -x_{jk} + x_{ik} \leq 0 \quad (1 \leq i < j < k \leq n)$$

Übung: Man zeige, daß es nicht nötig ist, längere Kreise auszuschließen.

Für $a_{ij} = c_{ij} - c_{ji}$ erhalten wir

$$\begin{array}{rl} \text{minimiere} & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} a_{ij} x_{ij} \\ \text{so daß} & 0 \leq x_{ij} + x_{jk} - x_{ik} \leq 1 \quad (1 \leq i < j < k \leq n) \\ & 0 \leq x_{ij} \leq 1 \quad (1 \leq i < j \leq n) \\ & x_{ij} \text{ ganzzahlig } (1 \leq i < j \leq n) \end{array}$$

Ist z der Wert der Optimallösung, so gilt

$$minKreuz(G, \pi_1) = z + C.$$

Größe des IP:

$$\begin{array}{rl} \binom{n}{2} & \text{Variablen} \\ 2\binom{n}{2} & \text{triviale Ungleichungen} \\ 3\binom{n}{3} & \text{“3-Kreis”-Ungleichungen} \end{array}$$

→ LP-Relaxierung ist polynomiell groß, aber zu groß für eine effiziente Lösung.

Lösungsverfahren “Branch and Cut”

(1) Starte mit trivialer Relaxierung

$$0 \leq x_{ij} \leq 1.$$

(2) Wiederhole

Löse LP.

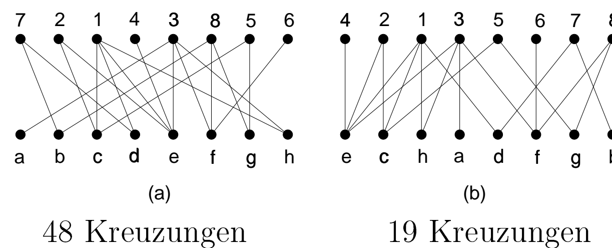
Füge von der Optimallösung verletzte 3-Kreis-Ungleichungen („Schnittebenen“) hinzu, falls nicht vorhanden, gehe zu (3).

(3) Gilt $x_{ij} \in \{0, 1\}$, so ist die Optimallösung gefunden, sonst wähle ein x_{ij} mit $0 < x_{ij} < 1$. Wende gleiches Verfahren rekursiv auf die beiden Probleme in denen x_{ij} auf 0 bzw. 1 fixiert ist.

→ Branch-and-Bound-Verfahren in dem die Schranken (Bounds) durch ein Schnittebenenverfahren berechnet werden.

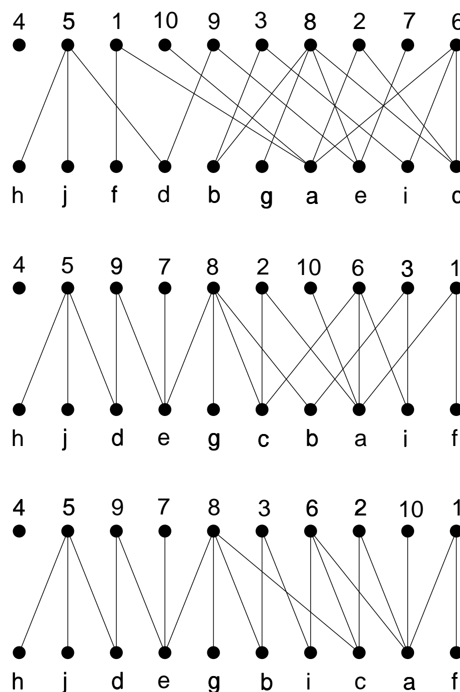
Dieses Verfahren hat exponentielle Laufzeit, funktioniert in der Praxis aber erstaunlich gut. Es erlaubt den experimentellen Vergleich der heuristischen Lösungen mit Optimallösungen.
Aufbauend auf diesem Verfahren kann auch das 2-Schichten-Kreuzungsminimierungsproblem optimal gelöst werden, wenn beide Schichten frei permutiert werden dürfen, allerdings mit sehr hohen Laufzeiten!

4.3.10 Experimentelle Studien aus Jünger & Mutzel (1997) [JM97]

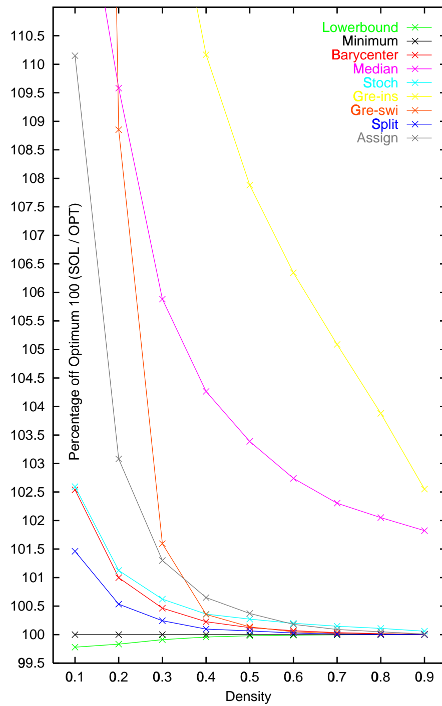


48 Kreuzungen 19 Kreuzungen

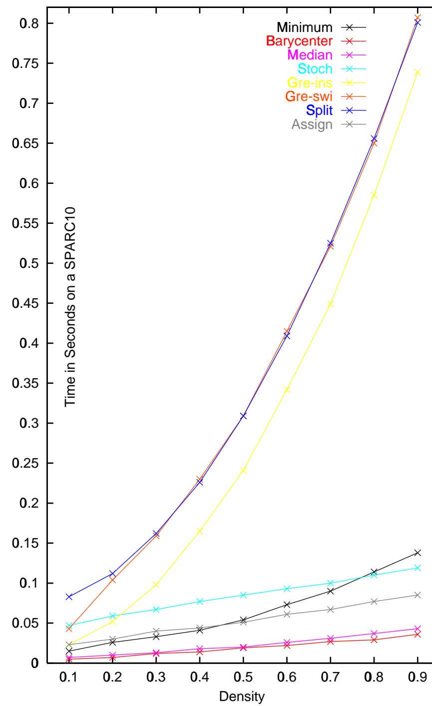
Minimale Kreuzungszahl mit (a) fixierter unterer Schicht und (b) beiden Schichten frei permutierbar.



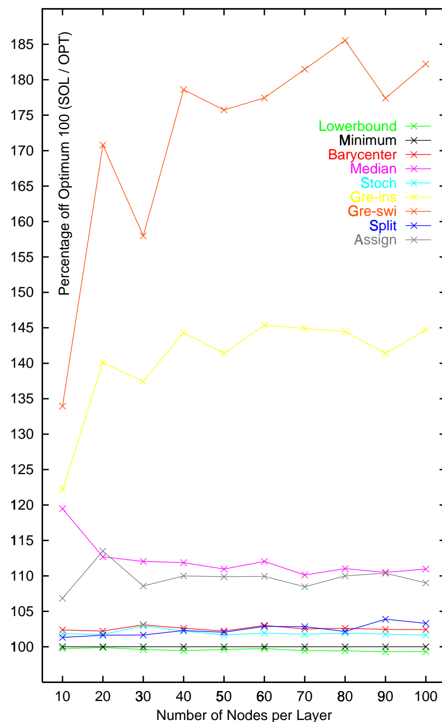
Die erste Zeichnung erhält man durch die LR-opt Heuristik und sie enthält 30 Kreuzungen, die zweite durch die Barycenter Heuristik mit 10 Kreuzungen und die letzte Zeichnung ist die optimale Lösung mit 4 Kreuzungen.



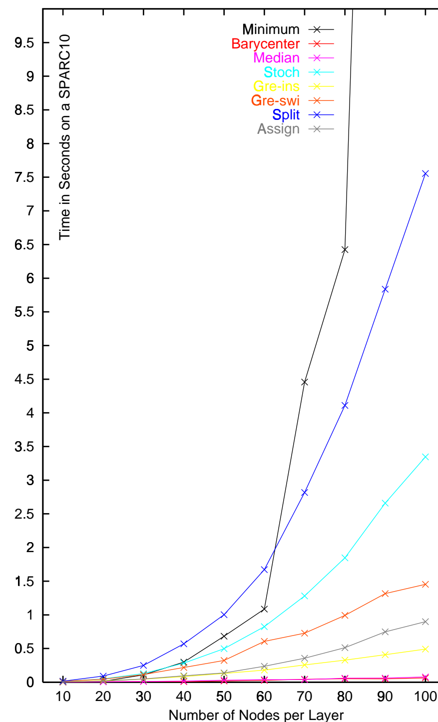
Ergebnisse für 100 Instanzen von 20+20 Knoten mit steigender Dichte



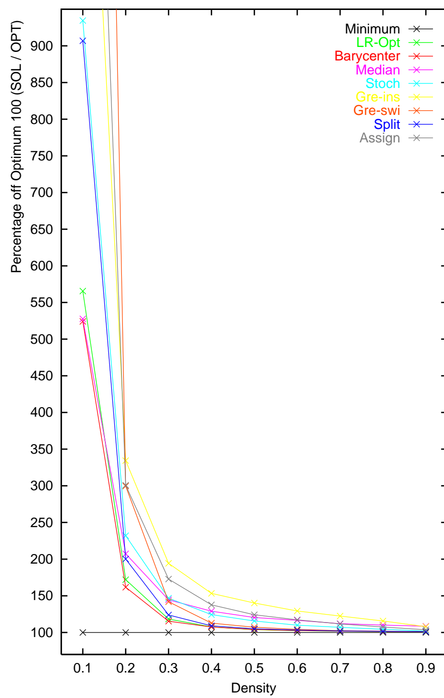
Zeit für 100 Instanzen von 20+20 Knoten mit steigender Dichte



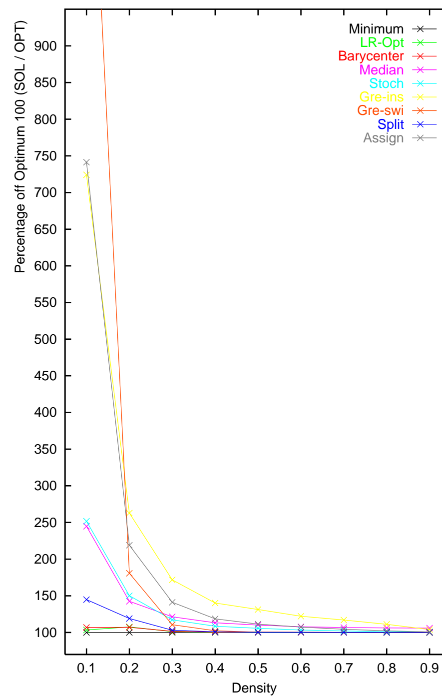
Ergebnisse für 10 Instanzen auf dünnen Graphen mit ansteigender Größe



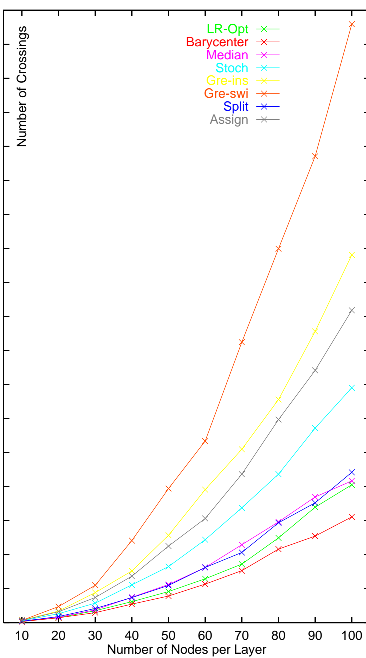
Zeit für 10 Instanzen auf dünnen Graphen mit ansteigender Größe



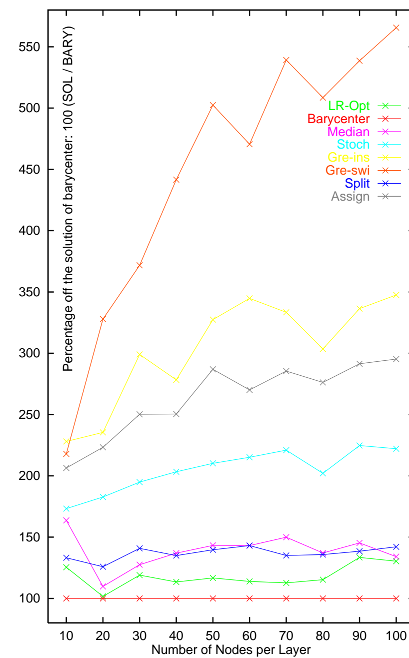
Ergebnisse für 100 Instanzen von 10+10 Knoten mit einem Versuch



Ergebnisse für 100 Instanzen von 10+10 Knoten mit 10 Versuchen



Ergebnisse für 10 Instanzen auf dünnen Graphen (absolut)



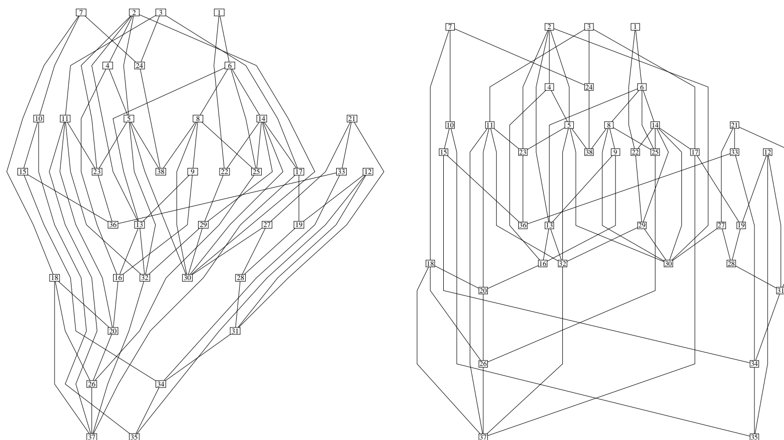
Ergebnisse für 10 Instanzen auf dünnen Graphen (bezogen auf Barycenter)

Schließlich: Falls der Graph kein DAG ist und die Hierarchie nicht aus der Anwendung vorgegeben ist, so gilt es, das Feedback Arc Set Problem zu lösen. Dieses kann optimal (auch in der Praxis) mit einem ähnlichen Branch-and-Cut Verfahren wie vorhin beschrieben gelöst werden. Hinzu kommen zahlreiche Heuristiken, auf die wir hier nicht eingehen.

4.4 Phase 3: Horizontale Koordinatenzuweisung

Situation: Nach Schichtung (Phase I) und Kreuzungsminimierung (Phase II) steht die Topologie der Zeichnung fest.

Ziel: Vermeidung des „Spaghetti-Effekts“.



Grundidee einfacher Heuristiken

E. R. Ganser, E. Koutsofios, S. C. North, K. P. Vo (1993) [GKNV93]

(1) Finde eine Anfangsordnung.

(2) Verbessere das Layout durch wiederholtes Traversieren der Schichten von unten nach oben und oben nach unten bis ein Abbruchkriterium erfüllt ist.

In GKNV[1993]:

(1) Plaziere auf jeder Schicht alle Knoten linksbündig.

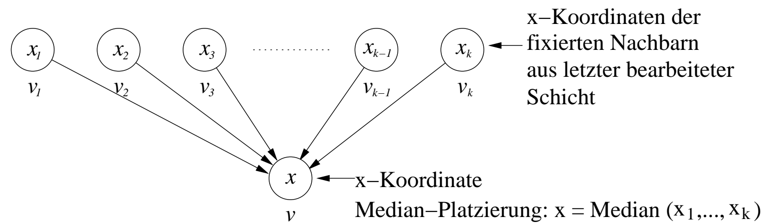
(2)
- Betrachte während einer Traversierung die letzte besuchte Schicht als fixiert und ordne den Knoten auf der nächsten (benachbarten) Schicht Prioritäten zu.
- Plaziere nun die Knoten in der Prioritätsreihenfolge so nahe wie möglich an ihrer „Wunschposition“. (Plazierungen werden nicht rückgängig gemacht.)

Beispiele für Prioritäten und Wunschpositionen

$\omega(e) \in \mathbb{N}$: Wichtigkeit der Kante $e \in E$ (vorgegebene Parameter).

Median-Heuristik

Idee:

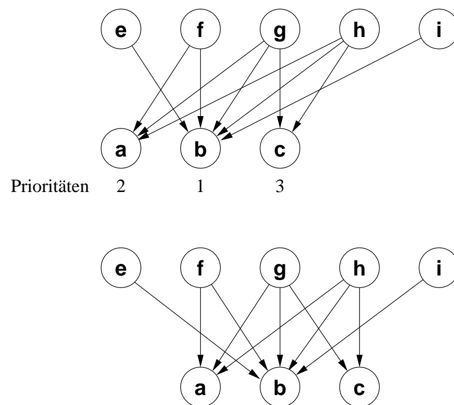


Die Median-Plazierung minimiert

$$\sum_{i=1}^{k} |x - x_i|$$

Priorität$(v) = \sum_{i=1}^{k} \omega(v, v_i)$
Wunschposition(v) = Median$(x_1, x_2, \ldots, x_k)$ (k gerade: linker rechter durchschnittlicher Median).

Beipiel 4.4.1
$(\omega(e) = 1 \forall e \in E)$



Vermeidung des Spaghetti-Effekts durch Ersetzung von $\omega(e)$ durch

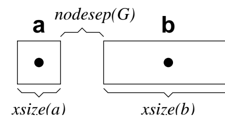
$$\Omega(e) \cdot \omega(e).$$

$\Omega(e)$ interner Parameter, hoch für lange Kanten.
Zahlreiche andere Heuristiken für Prioritäten und Wunschpositionen.

Lösung mittels LP-Techniken

(auch GKNV[1993])
Sei $\rho(a, b)$ der minimale Abstand zwischen den Zentrums-x-Koordinaten der in einer Schicht benachbarten Knoten a und b, z.B.

$\rho(a,b) = \frac{xsize(a)+xsize(b)}{2} + nodesep(G)$



Ganzzahliges Optimierungsproblem

$$
\begin{array}{lrl}
\text{minimiere} & \sum_{(v,w)\in E} \Omega(v,w)\cdot\omega(v,w)\cdot|x_w - x_v| & \\
\text{so daß} & x_b - x_a \geq \rho(a,b) & \forall \text{ Paare } a,b \text{ benachbarter Knoten} \\
& & \text{auf einer Schicht} \\
& x_v \in \mathbb{N} & \forall v \in V
\end{array}
$$

Umformulierung in ein LP

$$
\begin{array}{lrl}
\text{minimiere} & \sum_{(v,w)\in E} \Omega(v,w)\cdot\omega(v,w)\cdot y_{vw} & \\
\text{so daß} & y_{vw} \geq x_w - x_v & \forall (v,w) \in E \\
& y_{vw} \geq x_v - x_w & \forall (v,w) \in E \\
& x_b - x_a \geq \rho(a,b) & \forall \text{ Paare } a,b \text{ benachbarter Knoten} \\
& & \text{auf einer Schicht} \\
& x_v \geq 0 & \forall v \in V
\end{array}
$$

Ganzzahligkeitsbedingungen sind auch hier überflüssig, weil die Restriktionsmatrix total unimodular ist. → effizient lösbar.

Allerdings: $|E|$ zusätzliche Variablen, $2|E|$ zusätzliche Restriktionen → problematisch für große Graphen.

Wahl von $\Omega(e)$ nach GKNV[1993]

$$
\Omega(e) = \begin{cases} 1, & \text{falls beide Endknoten echt} \\ 2, & \text{falls einer echt, einer künstlich} \\ 8, & \text{falls beide künstlich} \end{cases}
$$

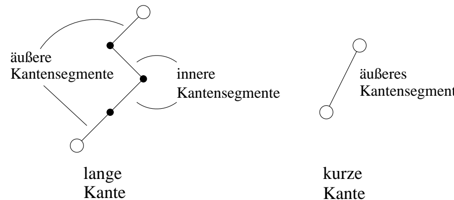
tendiert zur Vermeidung des Spaghetti-Effekts, kann es aber nicht garantieren.

Ein neuer Algorithmus für Phase 3

Bezeichnungen:

- Lange Kanten = Kanten des Originalgraphen, die zwei nicht benachbarte Schichten verbinden.
- Kantensegment = Kante des Graphen nach Einfügen der künstlichen Knoten.
- Inneres Kantensegment = Kantensegment, das zwei künstliche Knoten verbindet.
- • = künstlicher Knoten
- ○ = Originalknoten

- Linker/rechter Bruder = nächster Knoten auf derselben Schicht nach links/rechts.
- Plazierung = Festlegung der x-Koordinate.



Eigenschaften der erzeugten Zeichnungen:

(A) Knoten derselben Schicht erhalten dieselbe y-Koordinate. Mindestabstände zwischen den Schichten werden eingehalten.

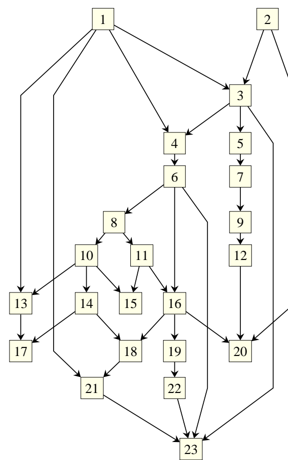
(B) Die Reihenfolge der Knoten auf den Schichten wird respektiert, ebenso die Mindestabstände $\rho(a, b)$.

(C) Die Kantensegmente sollen geradlinig gezeichnet werden.

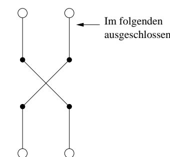
(D) Innere Kantensegmente sollen senkrecht gezeichnet werden.

$\Rightarrow$ Jede Kante hat höchstens zwei Knicke.

$\Rightarrow$ kein Spaghetti-Effekt.



Kleines Problem: Kanten schneiden sich in inneren Segmenten $\Rightarrow$ (B) widerspricht (D)! Diese Situation muß vorher beseitigt werden.



Laufzeit: $O(m'(\log m')^2)$ falls $m' \geq n'$. Dabei ist m' = Anzahl der Kantensegmente und n' = Anzahl **aller** Knoten (Originalknoten + künstliche Knoten).

Grobe Skizze des Algorithmus:

(1) Festlegen der x-Koordinaten der künstlichen Knoten.

(2) Festlegen der x-Koordinaten der Originalknoten.

(3) Festlegen der y-Koordinaten.

zu (1): Hier muß Bedingung (D) erreicht werden.

(D) $\Leftrightarrow$ alle künstlichen Knoten derselben langen Kante erhalten dieselbe x-Koordinate.

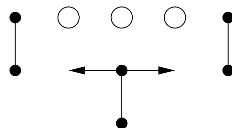
Also hängt (D) nur von Schritt (1) ab.

zu (2): Hier wird schichtweise die Gesamtlänge der äußeren Segmente minimiert. Später genauer.

Definition 4.4.2
Die Länge einer Kante (u, v) *ist im Folgenden definiert als* $|x_v - x_u|$. *(y-Koordinaten gibt es erst nach Schritt (3)).*

(1) Plazierung der künstlichen Knoten

Idee: Schiebe alle Knoten unter Berücksichtigung von (D) horizontal zusammen. Dadurch sind i.A. nicht alle x-Koordinaten festgelegt:



Grobes Vorgehen:

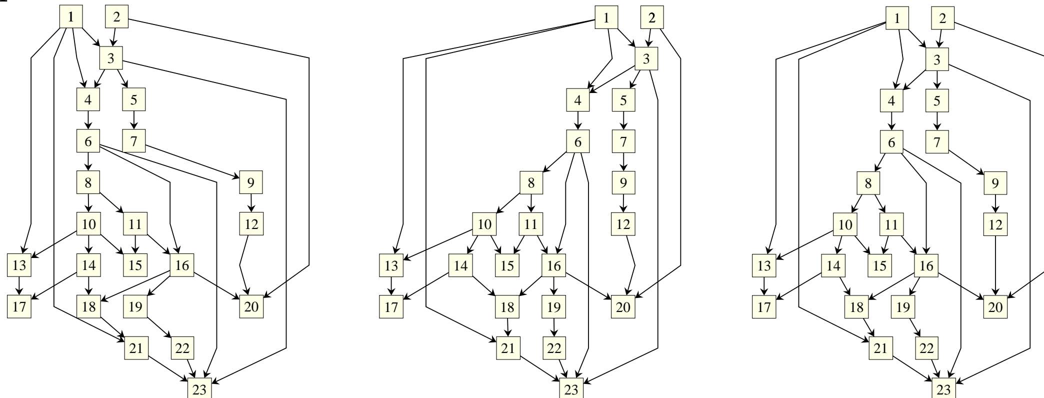
- Plaziere alle Knoten so weit wie möglich nach links.
- Plaziere alle Knoten so weit wie möglich nach rechts.
- Wähle die Durchschnitts-x-Koordinaten beider Plazierungen als endgültige x-Koordinaten für die künstlichen Knoten.

$\Rightarrow$ Vorgehen ist symmetrisch.

Beipiel 4.4.3

$\frac{1}{2}\left(\quad + \quad \right) =$

Beipiel 4.4.4



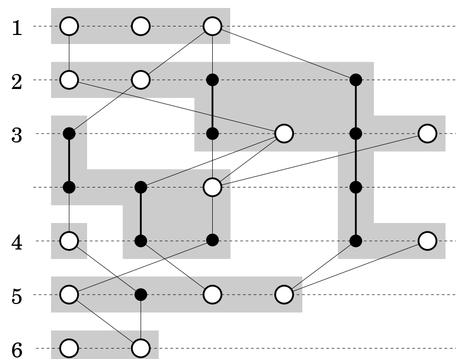
Verschiebung nach links (rechts analog):

1. Teile die Knoten in Klassen auf.
2. Plaziere jeder dieser Klassen separat.
3. Kombiniere die Plazierungen der Klassen.

1. Aufteilung in Klassen

- Durchlaufe alle linkesten Knoten v von oben nach unten.
- Wenn v noch zu keiner Klasse gehört
  - Richte neue Klasse C ein, die v enthält.
  - Füge rekursiv in C die folgenden Knoten ein, falls sie noch zu keiner Klasse gehören:
    * Für jedes $w \in C$ seinen rechten Bruder (falls existent).
    * Wenn $w \in C$ künstlich ist, alle anderen künstlichen Knoten auf derselben langen Kante.

Beipiel 4.4.5



Laufzeit: $O(n')$ (beachte: Anzahl der inneren Segmente $\leq n'$)

2. Plazierung einer Klasse C

- Plaziere den linkesten Knoten in C auf Position 0.
- Plaziere die übrigen Knoten in C so weit wie möglich nach links. (Sinnvolle Formulierung wegen der Definition der Klassen.)

Laufzeit: $O(n')$

3. Kombination verschiedener Klassen

- Plaziere alle Klassen C von oben nach unten (siehe 2.).
- Falls C rechte Brüder aus einer anderen Klasse hat (schon plaziert!)
  - Schiebe die gesamte Klasse C so nah wie möglich rechts an.
- Sonst
  - Minimiere die Gesamtlänge aller Kantensegmente zwischen C und den vohergehenden Klassen durch Verschieben der gesamten Klasse C.

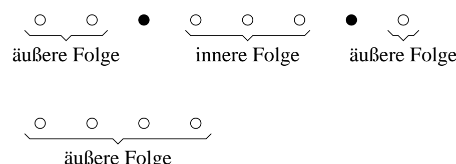
Beipiel 4.4.6
Siehe letztes Beispiel.

Laufzeit: $O(m' \log m')$ (wegen Fall 2)

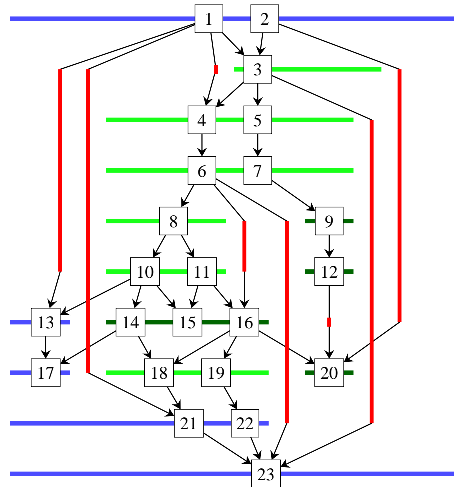
(2) Plazierung der Originalknoten

Beobachtung: Da die künstlichen Knoten festliegen, können Originalknoten unabhängig plaziert werden, wenn sie auf der selben Schicht liegen und ein künstlicher Knoten dazwischen liegt.

Bezeichnungen: Eine **Originalfolge** ist eine maximale Folge von nebeneinanderliegenden Originalknoten. **Innere Folgen** sind von zwei künstlichen Knoten begrenzt, **äußere** von höchstens einem:



Beipiel 4.4.7



Idee: Plaziere die Originalfolgen der Reihe nach. Minimiere dabei jedesmal die Gesamtlänge aller Kanten, die die aktuelle Folge mit allen schon vorher plazierten Knoten verbindet.

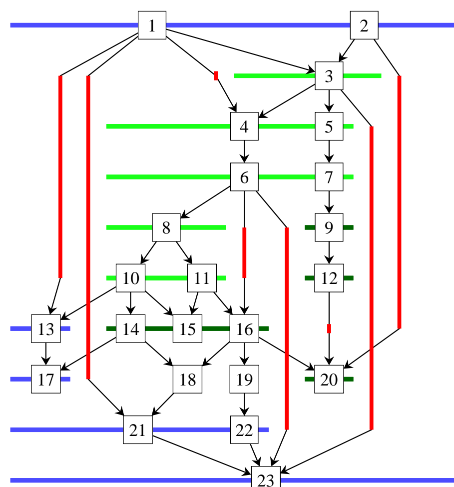
Fragen:

1. In welcher Reihenfolge werden die Folgen plaziert? Wichtig!
2. Wie funktioniert die Minimierung?

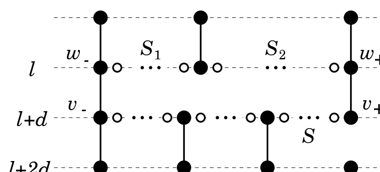
1. Reihenfolge der Folgen

Beobachtung: Nach Schritt (1) existieren viele **fixierte** Folgen, d.h. innere Folgen, die durch ihre künstlichen Brüder festgelegt sind, da diese den minimalen Abstand haben.

Grobe Idee: Bei diesen Folgen anfangen und dann nach oben und unten weitergehen.



Definition 4.4.8
Sei $v_1, \dots, v_n$ eine Originalfolge auf Schicht $l + d$ und $d \in \{-1, 1\}$. Sei v_- der nächste Bruder von v_1 nach links, der künstlich ist und einen künstlichen Nachbarn w_- auf Schicht l hat, analog v_+ und w_+.
*Dann heißen die Originalfolgen zwischen w_- und w_+ die **Nachbarfolgen** von $v_1, \dots, v_n$ auf der Schicht l. Falls v_- oder v_+ nicht existieren, heißt $v_1, \dots, v_n$ eine **äußere Folge** bzgl. l.*

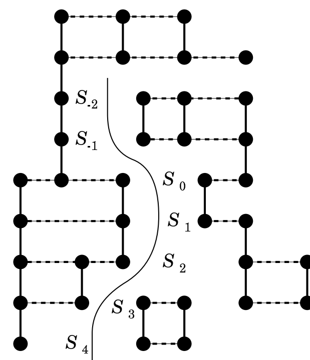


Algorithmus: Benutze zwei Arrays pro innerer Originalfolge S:

$P(S)$:Die Folge S ist schon plaziert/fixiert
$D(S)$:Richtung $\in \{-1, 1, 0\}$, in die S plaziert wird

- Für jede innere Folge S
  - Setze $P(S) = TRUE$, falls S fixiert ist, sonst $FALSE$
  - Setze $D(S) = 0$
- Für $d = 1, -1$
  - Für alle Schichten l von oben nach unten $(d = 1)$/von unten nach oben $(d = -1)$.
    - Plaziere die äußeren Folgen in l, ohne die aktuellen Abstände zwischen Knoten zu verkleinern.
    - Für alle inneren Folgen S auf l
      - Wenn $D(S) = d$
        Plaziere S, setze $P(S) = TRUE$
    - Für alle inneren Folgen S auf $l + d$
      - Wenn für alle Nachbarfolgen S' von S auf Schicht l gilt $P(S') = TRUE$
        Setze $D(S) = d$

Behauptung 4.4.9
Durch diesen Algorithmus werden alle Folgen plaziert, es gibt also „genug“ fixierte Folgen.



Minimierung der Kantenlänge

Ziel: Plaziere $v_1, \ldots, v_r$, so daß

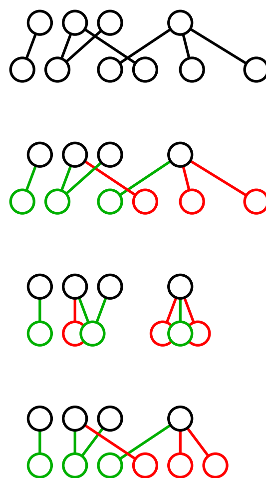
$$\sum_{i=1}^{r} \sum_{v \in \delta(v_i,d)} |x_v - x_{v_i}| \quad (*)$$

minimiert wird unter Berücksichtigung der Mindestabstände zwischen $v_1, \ldots, v_r$ und deren künstlichen Nachbarn.

Dabei ist $v_1, \ldots, v_r$ eine (nicht notwendig maximale) Originalfolge auf der Schicht l, $\delta(v_i, d) =$ Nachbarn von v_i auf Schicht $l + d$.

Idee: Divide & Conquer:

- Teile die Folge in der Mitte auf: $t = \lfloor \frac{r}{2} \rfloor$
- Plaziere beide Teilfolgen $v_1, \ldots, v_t$ und $v_{t+1}, \ldots, v_r$ rekursiv
- Kombiniere die Teilplazierungen.



Die Plazierung eines einzelnen Knotens ist einfach (wähle Median). Es bleibt noch zu klären, wie man Teilfolgen kombiniert.

Idee: In der Mitte auseinanderschieben bis $x_{v_{t+1}} - x_v \geq \varrho(v_t, v_{t+1}) =: \varrho$, d.h. entweder v_t nach links oder v_{t+1} nach rechts. Dabei falls nötig linke Brüder von v_t oder rechte Brüder von v_{t+1} mitverschieben.

Definition 4.4.10
(optimale Plazierungen $x_{v_1}, \ldots, x_{v_t}$ und $x_{v_{t+1}}, \ldots, x_{v_v}$ vorgegeben)
Für $i \in \{1, \ldots, t\}$ sei $x^-_{v_i}(p) = \min\{x_{v_i}, p - \underbrace{\varrho(v_j, v_{j+1})}_{\sum_{j=1}^{t-1} \varrho(v_j, v_t)}\}$.
Sei $j(p) \in \{1, \ldots, t\}$ der minimale Index i mit $x^-_{v_i}(p) < x_{v_i}$

$$
\begin{aligned}
r^-(p) &:= \sum_{i=j(p)}^{t} \left(\#\{v \in \delta(v_i, d) \mid x_v \geq x^-_{v_i}(p)\} - \#\{v \in \delta(v_i, d) \mid x_v < x^-_{v_i}(p)\}\right) \\
f^-(p) &:= \sum_{i=1}^{t} \sum_{v \in \delta(v_i, d)} |x_v - x^-_{v_i}(p)|
\end{aligned}
$$

Beobachtung 4.4.11

1. $r^- : (-\infty, x_{v_t}] \to \mathbb{Z}$ ist eine monoton fallende stückweise konstante Funktion mit $O(m')$ Sprungstellen.
2. $f^- : (-\infty, x_{v_t}] \to \mathbb{R}^+$ ist konvex, stückweise linear und hat $O(m')$ Knicke.
3. $-r^-$ ist die Rechtsableitung von f^-.

Satz 4.4.12 *(ohne Beweis)*
Die Teilfolgen $v_1, \ldots, v_t$ und $v_{t+1}, \ldots, v_r$ seien jeweils optimal plaziert im Sinne von $(*)$ und $x_{v_{t+1}} - x_{v_t} < \varrho$. Falls $p \in \mathbb{R}^+$ die Funktion $f^-(p) + f^+(p + \varrho)$ minimiert, dann ist $x^-_{v_1}(p), \ldots, x^-_{v_t}(p), x^+_{v_{t+1}}(p + \varrho), \ldots, x^+_{v_r}(p + \varrho)$ optimal im Sinne von $(*)$.

Bedeutung: Man muß nur noch über **einen** Parameter p optimieren. **Äquivalent:** Man muß $p^-, p^+ \in \mathbb{R}$ so bestimmen, daß $p^+ - p^- = \varrho$ und $f^-(p^-) + f^+(p^+)$ minimal ist. Man kann dazu die Ableitungen r^- und r^+ verwenden.

Algorithmus

- Setze $p^- = x_{v_t}$ und $p^+ = x_{v_{t+1}}$
- Solange $p^+ - p^- < \varrho$
  - wenn $r^-(p^-) < r^+(p^+)$
    - verringere p^- bis p^- eine Sprungstelle von r^- ist oder $p^+ - p^- = \varrho$
  - sonst, erhöhe p^+ bis p^+ eine Sprungstelle von r^+ ist oder $p^+ - p^- = \varrho$
- Für $i = 1, \ldots, t$ setze $x_{v_i} = x^-_{v_i}(p^-)$
- Für $i = t + 1, \ldots, r$ setze $x_{v_i} = x^+_{v_i}(p^+)$

Aus Laufzeitgründen müssen alle Sprungstellen von r^- und r^+ in linearer Zeit gesammelt werden.

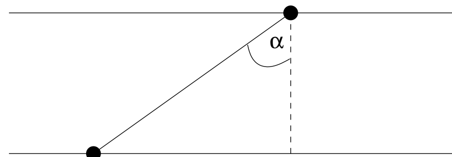
- Sortieren: $O(m' \cdot \log m')$
- Divide and Conquer: $O(m'(\log m')^2)$

⇒ Gesamtlaufzeit: $O(m'(\log m')^2)$

(3) Berechnung der y-Koordinaten

Einfache Methode: $y_v :=$ Schicht von v

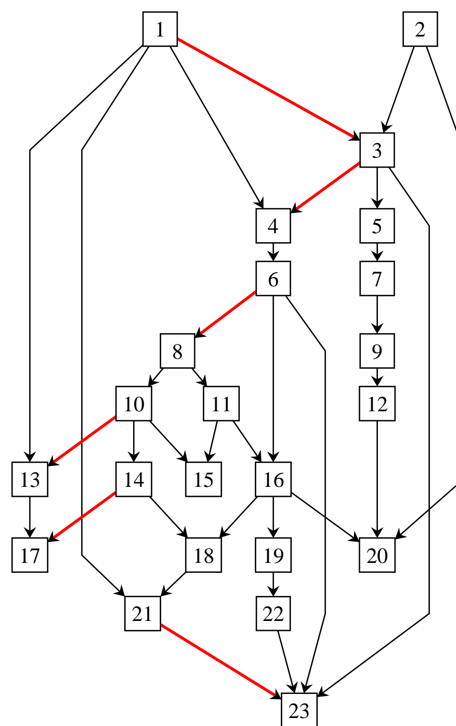
Besser: wähle den Abstand zwischen zwei Schichten, so daß die längste Kante dazwischen einen vorgegebenen Winkel bekommt. (Mindestabstand beachten)



Algorithmus: wähle α

- $c = 0$
- Setze $y_v = 0 \;\forall\; v$ auf Schicht 1
- Für alle Schichten $l = 2, \ldots, k$
  - Setze $d =$ Mindestabstand der Schichten
  - Für alle Knoten v auf Schicht l
    - Für alle Nachbarn w von v auf Schicht $l-1$
      - Setze $d = \max\{d, |x_v - x_w| / \tan\ \alpha\}$
  - $c = c + d$
  - Setze für alle Knoten v auf Schicht l: $y_v = c$

Laufzeit: $O(m')$


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

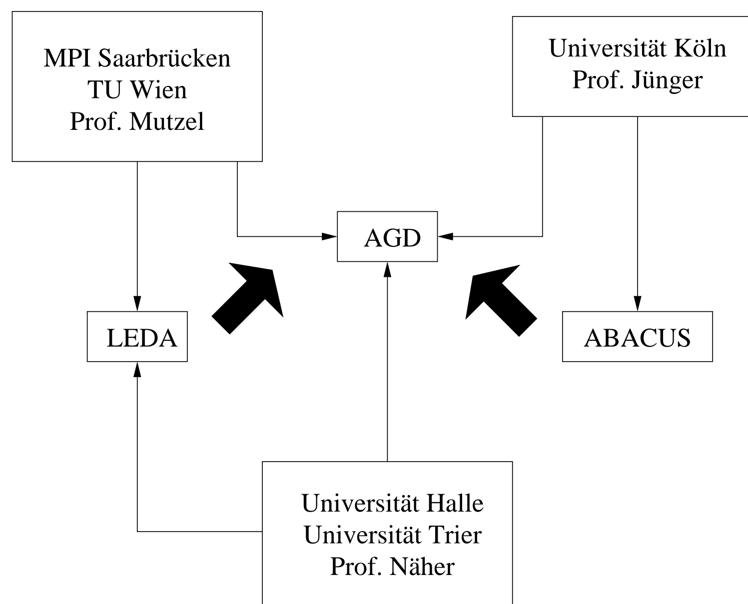
Kapitel 5

AGD

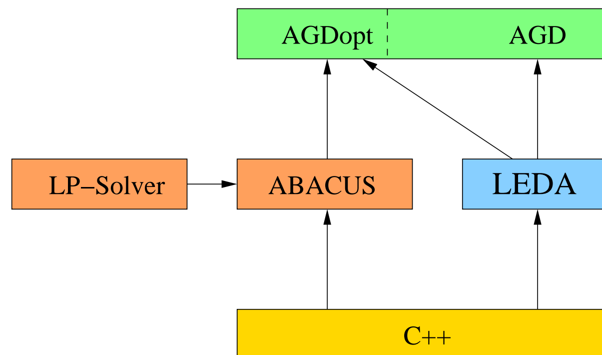
5.1 Die AGD-Bibliothek zum Zeichnen von Graphen

- C++ Klassenbibliothek von Graphzeichenalgorithmen
- basiert auf LEDA und ABACUS
- Zeichenverfahren für verschiedene Graphklassen
- zahlreiche Algorithmen für Teilprobleme erweiterbar
- Forschungsprojekt, von der DFG gefördert

Beteiligte Gruppen



Grundsätzlicher Aufbau



Zeichenalgorithmen

- Sugiyama
- Planarisierungsmethode
- planare Verfahren
  - orthogonale
  - straight-line
  - Sichtbarkeitsdarstellungen
- Kräfteverfahren
  - Spring-Embedder
  - Tutte
- spezialisierte Verfahren
  - Bäume
  - st-planare Graphen

Unabhängigkeit von einer Visualisierungskomponente
Visualisierungskomponente: Graph-Editor, Viewer, Applikation, ...

Ziel:

- einfache Integration der Algorithmen in bestehende Visualisierungskomponenten Algorithmen in „reinem“ C++ portabel

Lösung:

- Basisklasse, die Schnittstelle zum Zugriff auf graphische Attribute festlegt
- graphische Attribute: Koordinaten der Knoten, Knickpunkte der Kanten
- Algorithmen greifen nur über diese Basisklasse auf graphische Attribute zu

- Aufruf des Algorithmus übergibt Instanz einer Implementierung

```
class LayoutInterface {
public:
        LayoutInterface(const graph &G);
        virtual ~LayoutInterface() { }

        virtual double get_x (node v) const = 0;
        virtual void   set_x (node v, double new_x) = 0;

        virtual DPolyline get_bend_points (edge e)
                const = 0;
        virtual void set_bend_points (edge e, const
                DPolyline& l) = 0;

        virtual double get_width  (node v) const = 0;
        virtual void set_width  (node v, double
                new_w) = 0;

        ...
};

class GraphWinInterface : public LayoutInterface {
public:
        GraphWinInterface (GraphWin& GW);

        double get_x (node v) const {
                return gw_p->get_position(v).xcoord();
        }
        void set_x (node v, double new_x) {
                gw_p->set_position(v,point(new_x,get_y(v)));
        }

        double get_width (node v) const {
                return gw_p->get_radius(v)*2;
        }

        void set_width (node v, double new_w) {
                gw_p->set_radius (v, new_w/2);
        }

private:
        GraphWin *gw_p;
};

GraphWin gw;     // erzeuge GraphWin
```

```
gw.open();       // und lasse Benutzer Graph eingeben

// erzeuge Interface für gw
GraphWinInterface gwi(gw);

// g ist darzustellender Graph
const graph &g = gwi.get_graph();

// erzeuge einen Zeichenalgorithmus
SugiyamaLayout algo;

if (algo.check(g))   // teste, ob Vorbed. erfüllt
  algo.call(g,gwi);  // rufe Zeichenalgorithmus auf

// Benutzer kann Ergebnis weiterbearbeiten
gw.edit();
```

Modularisierung

- Viele Zeichenalgorithmen bestehen aus einzelnen Phasen (Teilschritten)
- Beispiele: Sugiyama, Planarisierungsmethode
- für einzelne Phasen sind unterschiedliche Implementierungen möglich
- Qualität der Lösung $\Leftrightarrow$ Laufzeit

Beispiel Sugiyama:

- Schichteinteilung
  - längste Wege
  - minimiere Gesamtkantenlänge
  - vorgegebene Maximalbreite einer Schicht
- Kreuzungsminimierung
  - Barycenter, Median, Split, Sifting, ...
- Zuweisung der Koordinaten
  - Buchheim, Jünger, Leipert
  - LP-basiert

Ziel in AGD:

- Zeichenalgorithmus als Framework implementieren
- einzelne Phasen sind durch (benutzerdefinierte) Implementierungen dynamisch austauschbar

$$\Rightarrow \text{ Module, Modul-Optionen}$$

Module können selbst wieder Modul-Optionen definieren.

`LongestPathRanking`: Bestimmung eines maximal azyklischen Untergraphen

Module

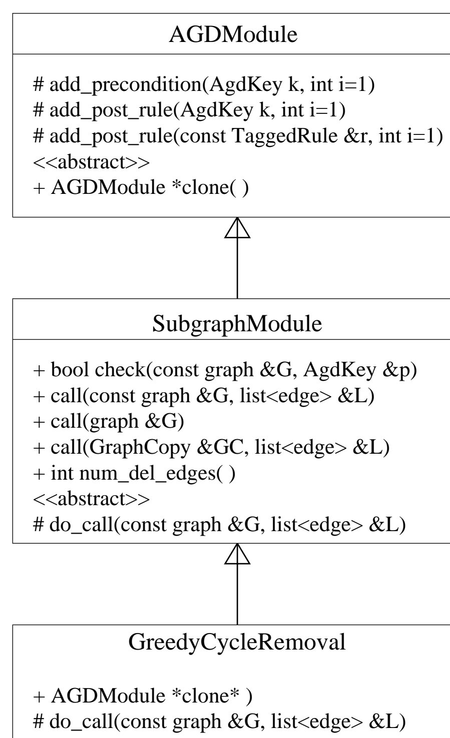
- Basisklasse definiert Typ

  $\Rightarrow$ Schnittstelle (Parameter für Aufruf)

- Vorbedingung (Eigenschaft, die Eingabe erfüllen muß)
- Nachbedingung (Eigenschaft, die Ausgabe erfüllt)

Beispiele für Eigenschaften:

- Graph: zusammenhängend, planar, azyklisch
- Untergraph: maximal azyklisch
- Layout: orthogonal, keine Kreuzungen



```
class SubgraphModule : public AGDModule {
public:
        static const int in_graph;
        static const int out_subgraph;

        SubgraphModule () : AGDModule(1,1) { }
        virtual ~SubgraphModule () { }

        bool check (const graph& G, AgdKey &p) const;

        void call (const graph& G,
                list<edge>& L);
        void call (graph& G);
        void call (GraphCopy& GC, <edge>& L);

        int num_del_edges() const;

protected:
        virtual void do_call (const graph& G,
                list<edge>& L) = 0;
};

class GreedyCycleRemoval : public SubgraphModule {
public:
        GreedyCycleRemoval();
        ~GreedyCycleRemoval() { }

        string name () const {
                return "Greedy Cycle Removal";
        }

        string impl_author () const {
                return "Carsten Gutwenger";
        }

        AGDModule *clone() const;

protected:
        void do_call (const graph& G,
                list<edge>& feedback);
};
```

Setzen von Vor-/Nachbedingung:

```
GreedyCycleRemoval::GreedyCycleRemoval()
{
```

```
        add_precondition (key::directed);
        add_post_rule (key::directed);
        add_post_rule (key::maximal_acyclic);
}
```

Kopieren von Modulen:

```
AGDModule *GreedyCycleRemoval::clone() const
{
        return new GreedyCycleRemoval;
}
```

Modul-Optionen

- Parametrisierte Basisklasse `AGDModuleOption<T>`
- `T` ist Modul-Typ (z.B. SubgraphModule)

Instanz enthält

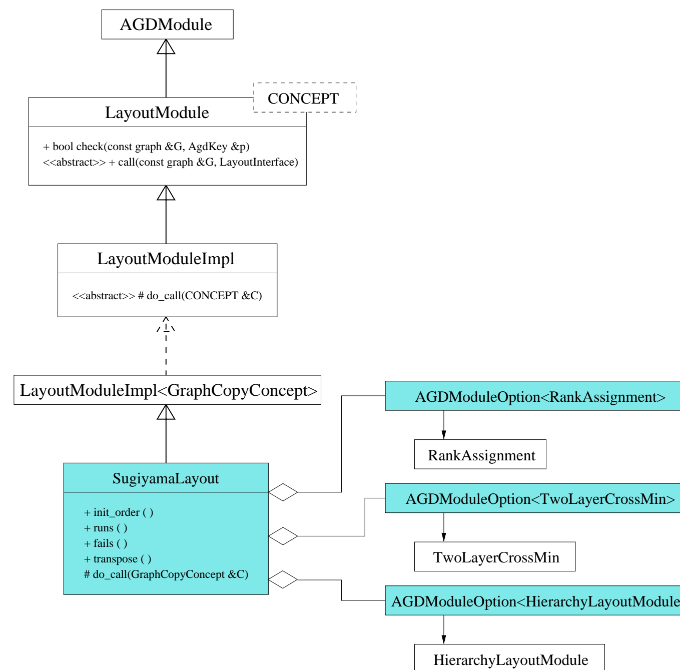
- Zeiger auf Algorithmus
- garantierte Vorbedingung
- erforderliche Nachbedingung

Achtung:

- beim „Setzen“ der Option wird Kopie mittels `clone()` angelegt
- Grund: keine Probleme mit Lebenszeit

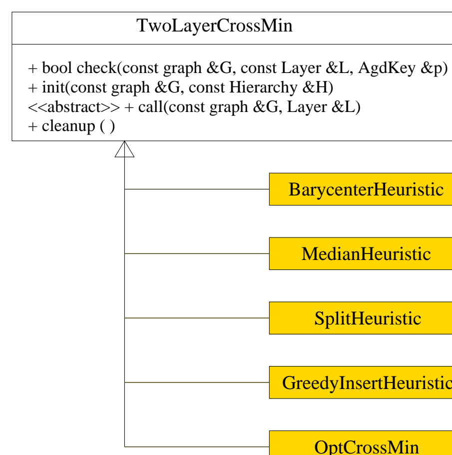
SugiyamaLayout

- Framework für Sugiyama-Algorithmus
- Modul-Optionen für die drei Phasen
- Hierarchy: repräsentierte saubere Hierarchie
- Layer: repräsentiert eine Schicht
- Optionen:
  - `init_order`: initiale Ordnung auf Schichten (DFS)
  - `fails`: Anzahl Iterationen ohne Verbesserung
  - `runs`: Anzahl randomisierte Durchläufe
  - `transpose`: Postprocessing




```
class SugiyamaLayout :
public LayoutModuleImpl<GraphCopyConcept>
{
  AGD_DECLARE_MODULE(RankAssignment,ranking)
  AGD_DECLARE_MODULE(TwoLayerCrossMin,cross_min)
  AGD_DECLARE_MODULE(HierarchyLayoutModule,comp_coord)

public:
  void ext_call (const graph& G, node_array<int>& rank,
    LayoutInterface& A);
  void reduce_crossings (GraphCopy& GC, Hierarchy& H);

  int number_of_crossings() const ;

  AGDModule *clone() const;

  int runs() const ;
  void runs(int n) ;

protected:
  void do_call (GraphCopyConcept &C);
};

SugiyamaLayout::SugiyamaLayout() : _runs(3), _fails(4),
        _transpose(true), _init_order(true)
{
        // precondition
```

```
        add_precondition (key::no_self_loops);
        add_precondition (key::directed);

        // initialize used modules
        PreCond  PRE_1, PRE_2(2);
        PostCond POST_1;

        PRE_1 [1] << key::no_self_loops << key::directed;
        ranking.init (PRE_1,POST_1,LongestPathRanking());

        PRE_2 [1] = PRE_1 [1];
        cross_min.init (PRE_2,POST_1,
                BarycenterHeuristic());

        comp_coord.init (PreCond(),POST_1,
                FastHierarchyLayout());
}

AGDModule *SugiyamaLayout::clone() const
{
        SugiyamaLayout *M = new SugiyamaLayout;

        M->runs(runs());
        M->fails(fails());
        M->transpose(transpose());
        M->init_order(init_order());

        M->set_ranking(ranking.get());
        M->set_cross_min(cross_min.get());
        M->set_comp_coord(comp_coord.get());

        return M;
}
```

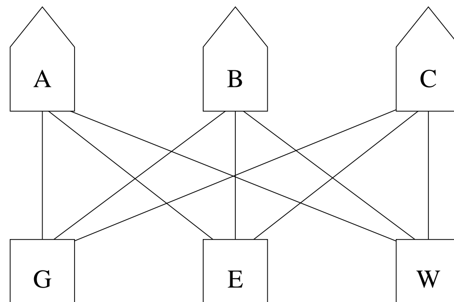


Kapitel 6

Planare Graphen

6.1 $K_{3,3}$ und K_5

6.1.1 Rätsel 1 (Dudeney (1917) [Dud17])



Kann man alle Leitungen kreuzungsfrei legen ?

Antwort: Nein !

6.1.2 Rätsel 2 (Möbius (1840, siehe [BLW77])

Es war einmal ein König mit fünf Söhnen ...

Testament: Die Söhne sollen das Königreich so aufteilen, daß jeder über eine Region regiert, die eine gemeinsame Grenze (nicht nur ein Punkt) mit jeder der anderen vier Regionen hat.

Kann das Testament erfüllt werden ?

Antwort: Nein!

6.1.3 Übersetzungen

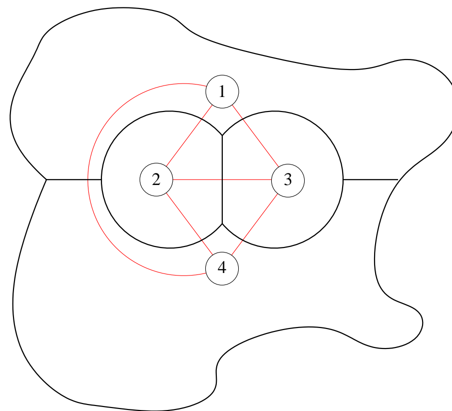
Rätsel 1

Kann der $K_{3,3}$ in der Ebene überkreuzungsfrei gezeichnet werden ?

Rätsel 2

Königreiche sind zusammenhängende Flächen in der Ebene.

Lösung für vier Söhne:



Eine Lösung für n Söhne entspricht einer überkreuzungsfreien Zeichnung des K_n.

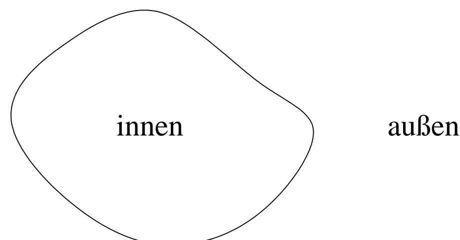
Also: Kann der K_5 in der Ebene überkreuzungsfrei gezeichnet werden?

Definition 6.1.1
Ein Graph ist planar genau dann, wenn er überkreuzungsfrei in der Ebene gezeichnet werden kann.

Satz 6.1.2
$K_{3,3}$ *und* K_5 *sind nicht planar.*

Beweis: Topologisches Grundwerkzeug:

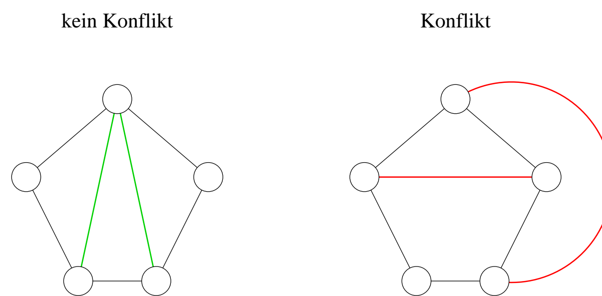
Eine geschlossene Kurve auf der Ebene teilt diese in zwei Regionen (innen und außen):



Sei C ein aufspannender Kreis (Kreis durch alle Knoten).

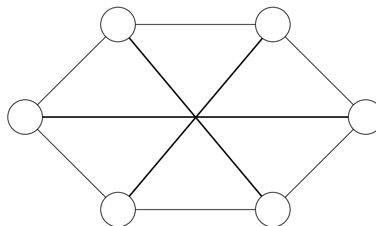
Zeichnung überkreuzungsfrei

- $\Rightarrow$ C ist eine geschlossene Kurve
- $\Rightarrow$ alle Kanten $e \in E \backslash C$ (außerhalb von C („Sehnen“)) werden entweder innen oder außen gezeichnet.



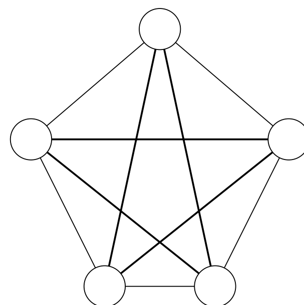
Zwei Sehnen stehen im Konflikt, wenn ihre Endknoten auf C in alternierender Reihenfolge auftreten. Stehen zwei Sehnen im Konflikt, so muß eine innen und die andere außen gezeichnet werden.

$K_{3,3}$**:** Ein 6-Kreis in $K_{3,3}$ hat drei paarweise im Konflikt stehende Sehnen:

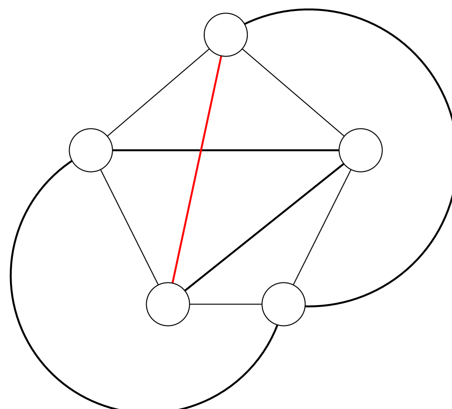


$\Rightarrow$ eine außen, eine innen, dritte kann nicht gezeichnet werden.

K_5**:** analog



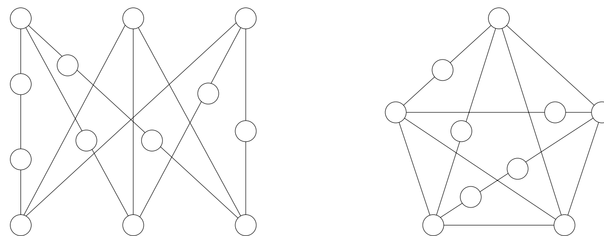
Von den fünf Sehnen können höchstens zwei nach außen, zwei nach innen, aber die fünfte Sehne kann nicht gezeichnet werden.



□

Eine **Unterteilung** (subdivision) eines Graphen entsteht durch Ersetzung von Kanten durch (intern paarweise disjunkte) Pfade.

z.B. Unterteilungen des $K_{3,3}$ und des K_5:



Klar: die Unterteilungen des $K_{3,3}$ und des K_5 sind nicht planar.

6.2 Satz von Kuratowski (1930) [Kur30]

Ein Graph ist planar genau dann, wenn er keine Unterteilungen des $K_{3,3}$ und des K_5 als Subgraph besitzt.

Beweis

Aufwendig ...

Anekdote: Kasimir Kuratowski fragte Frank Harary, woher die Bezeichnungen $K_{3,3}$ und K_5 kämen.

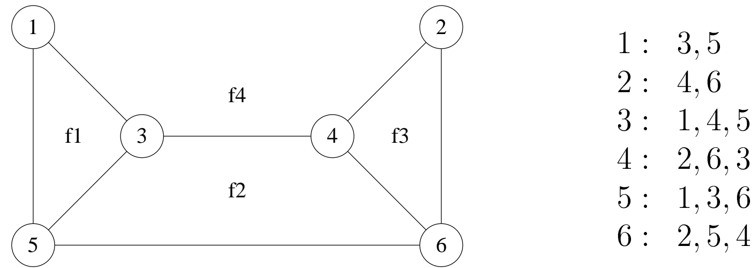
Antwort: Das K in K_5 steht für Kasimir und das K in $K_{3,3}$ steht für Kuratowski!

6.3 Einbettungsbegriffe (für planare Graphen auf der Ebene)

- Jede überkreuzungsfreie Zeichnung ist eine planare Einbettung.
- Für jede planare Einbettung sind die **Knoten-Uhrzeigerlisten** wie folgt definiert:

  Für jeden Knoten werden alle Nachbarn im Uhrzeigersinn aufgelistet. Normierung: beginnend mit kleinstem Knotenindex.

Beipiel 6.3.1



$$\begin{array}{ll} 1: & 3,5 \\ 2: & 4,6 \\ 3: & 1,4,5 \\ 4: & 2,6,3 \\ 5: & 1,3,6 \\ 6: & 2,5,4 \end{array}$$

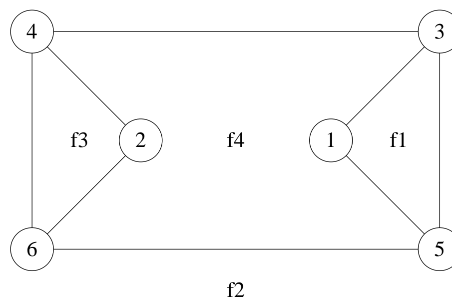
Flächenuhrzeigerlisten:

- im Uhrzeigersinn für Innenflächen
- im Gegenuhrzeigersinn für die Außenfläche

im Beispiel:

$$\begin{array}{ll} f_1: & 1,3,5 \\ f_2: & 3,4,6,5 \\ f_3: & 2,6,4 \\ f_4: & 1,5,6,2,4,3 \end{array}$$

Zeichnung 2 des Beispielgraphen:

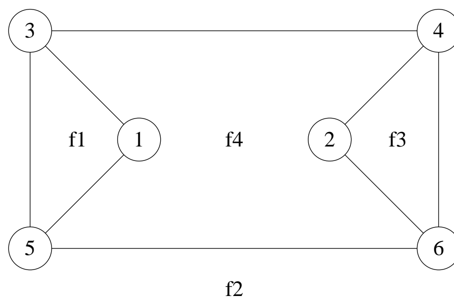


$$\begin{array}{ll} 1: & 3,5 \\ 2: & 4,6 \\ 3: & 1,4,5 \\ 4: & 2,6,3 \\ 5: & 1,3,6 \\ 6: & 2,5,4 \end{array} \qquad \begin{array}{ll} f_1: & 1,3,5 \\ f_2: & 3,4,6,5 \\ f_3: & 2,6,4 \\ f_4: & 1,5,6,2,4,3 \end{array}$$

$\Rightarrow$ gleiche Knoten- / Flächenuhrzeigerlisten

- Die Menge aller planaren Einbettungen mit gleichen Knoten- / Flächenuhrzeigerlisten ist eine **kombinatorische Einbettung** . Zwei planare Einbettungen mit gleichen Knoten- / Flächenuhrzeigerlisten heißen **kombinatorisch äquivalent** .

Zeichnung 3 des Beispielgraphen:

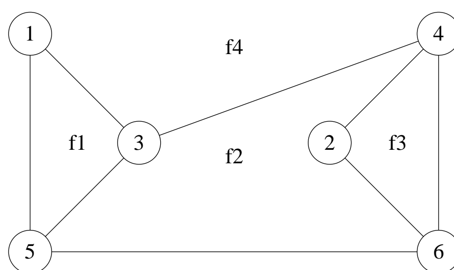


$$\begin{array}{rl} 1: & 3,5 \\ 2: & 4,6 \\ 3: & 1,5,4 \\ 4: & 2,3,6 \\ 5: & 1,6,3 \\ 6: & 2,4,5 \end{array} \xrightarrow[\text{Permutation}]{\text{zyklische}} \begin{array}{rl} 1: & 5,3 \\ 2: & 6,4 \\ 3: & 5,4,1 \\ 4: & 3,6,2 \\ 5: & 6,3,1 \\ 6: & 4,5,2 \end{array}$$

$$\begin{array}{rl} f_1: & 1,5,3 \\ f_2: & 3,5,6,4 \\ f_3: & 2,4,6 \\ f_4: & 1,3,4,2,6,5 \end{array} \xrightarrow[\text{Permutation}]{\text{zyklische}} \begin{array}{rl} f_1: & 5,3,1 \\ f_2: & 5,6,4,3 \\ f_3: & 4,6,2 \\ f_4: & 3,4,2,6,5,1 \end{array}$$

Bis auf Spiegelung kombinatorisch äquivalent.

Zeichnung 4 des Beispielgraphen:

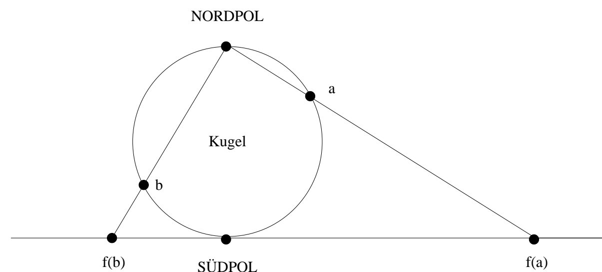


$$\begin{array}{rl} 1: & 3,5 \\ 2: & 4,6 \\ 3: & 1,4,5 \\ 4: & 2,3,6 \\ 5: & 1,3,6 \\ 6: & 2,4,5 \end{array} \qquad \begin{array}{rl} f_1: & 1,3,5 \\ f_2: & 2,6,5,3,4 \\ f_3: & 2,4,6 \\ f_4: & 1,5,6,4,3 \end{array}$$

Zu **keiner** bisherigen kombinatorisch äquivalent.

- Rieman'sche stereographische Projektion :

$$a, b \in \mathbb{R}^3, \; f(a), f(b) \in \mathbb{R}^2$$



  → Einbettungen auf der Kugeloberfläche (ohne Nordpol) $\overset{1-1}{\leftrightarrow}$ Einbettungen auf der Ebene

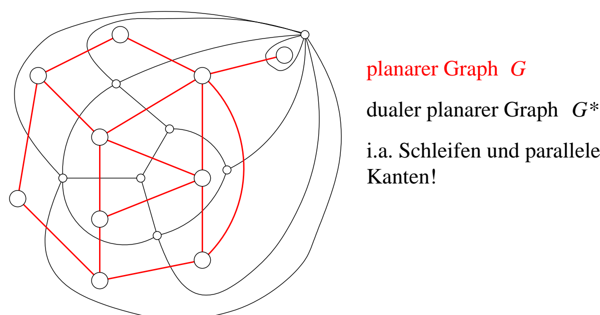
  → Die Fläche (das Land), die (das) den Nordpol enthält, entspricht der Außenfläche auf der Ebene.

  → Alle Projektionen der Kugeleinbettung auf die Ebene sind kombinatorisch äquivalent.

- Für 3-zusammenhängende planare Graphen existieren genau zwei verschiedene kombinatorische Einbettungen, die aber bis auf Spiegelung kombinatorisch äquivalent sind.

- Bei schwächerem Zusammenhang erhält man neue kombinatorische Einbettungen durch „Umklappen" von Flächen (siehe letztes Beispiel).

6.4 Duale Graphen

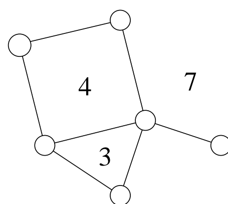


$$G_1, G_2 \text{ kombinatorisch äquivalent} \Leftrightarrow G_1^*, G_2^* \text{ kombinatorisch äquivalent}$$

Länge einer Fläche in planarer Einbettung

$$\begin{aligned} l(f) &= \text{Länge des geschlossenen Pfades entlang der Grenze} \\ &= \text{Grad des Flächenknotens im dualen Graphen} \end{aligned}$$

z.B.



Lemma 6.4.1
Für alle planaren Einbettungen von $G = (V, E)$ mit Flächenmenge F gilt

$$2|E| = \sum_{f \in F} l(f).$$

Beweis: Sei $G^* = (V^*, E^*)$ der duale Graph zu G. Es gilt

$$|E| = |E^*|$$

und deshalb

$$2|E| = 2|E^*| = \sum_{v \in V^*} \Delta(v) = \sum_{f \in F} l(f).$$

□

(Jede Kante wird in beiden Fällen genau zweimal gezählt.)

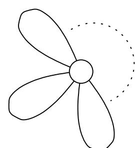
6.5 Euler-Formel

Satz 6.5.1 (Euler (1758))
Für einen zusammenhängenden planar eingebetten Graphen $G = (V, E)$ mit Flächenmenge F gilt

$$|V| - |E| + |F| = 2.$$

Beweis: Induktion

$|V| = 1$:

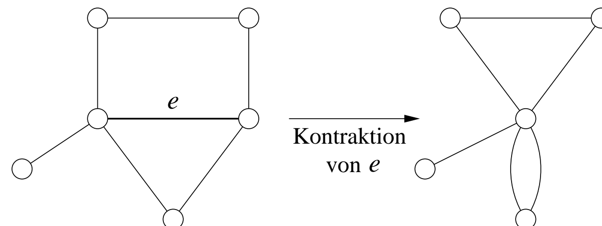


$|E| = 0 \Rightarrow |F| = 1$

Also $|V| - |E| + |F| = 1 - 0 + 1 = 2$. Für jede hinzugefügte Schleife wachsen $|E|$ und $|F|$ um je 1.

$|V| = n > 1$: G zusammenhängend $\Rightarrow \exists$ wenigstens eine Kante, die keine Schleife ist.

Wir konstruieren eine solche Kante $e \in E$, d.h. wir entfernen sie und identifizieren ihre Endknoten:



Ergebnis: $G = (V', E')$ mit Flächenmenge F'. Es gilt $|F'| = |F|$. (Die Flächen bleiben erhalten nur die Flächenlängen werden kürzer.) Außerdem

$$
\begin{aligned}
|V'| &= |V| - 1, |E'| = |E| - 1 \\
\Rightarrow |V| - |E| + |F| &= |V'| + 1 - (|E'| + 1) - |F'| \\
&= |V'| - |E'| + |F'| \\
\text{(Induktionsannahme)} &= 2
\end{aligned}
$$

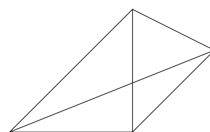
□

Euler entwickelte seine Formel ursprünglich für konvexe 3-dimensionale Polyeder.

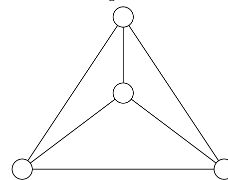
V: Menge der 0-dimensionalen Seitenflächen (Ecken)
E: Menge der 1-dimensionalen Seitenflächen (Kanten)
F: Menge der 2-dimensionalen Seitenflächen (Facetten)

z.B.

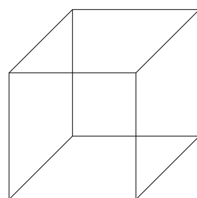
Tetraeder Projetion



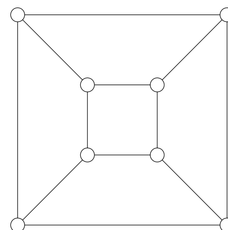
|V| = 4
|E| = 6
|F| = 4



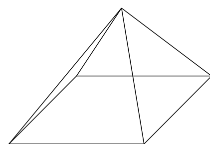
Würfel



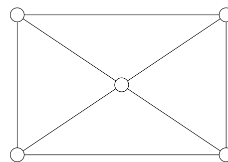
|V| = 8
|E| = 12
|F| = 6



Pyramide



|V| = 5
|E| = 8
|F| = 5



Bemerkung 6.5.2

1) Alle planaren Einbettungen eines zusammenhängenden planaren Graphen haben die gleiche Anzahl von Flächen.

2) Die Eulerformel gilt nicht für unzusammenhängende Graphen. Bei k Zusammenhangskomponenten gilt

$$|V| - |E| + |F| = k + 1.$$

6.5.1 Anwendung der Eulerformel

Satz 6.5.3

Ist $G = (V, E)$ ein einfacher planarer Graph und $|V| \geq 3$, so gilt

$$|E| \leq 3|V| - 6.$$

Ist G zusätzlich dreiecksfrei (kein Dreieck als Subgraph), so gilt

$$|E| \leq 2|V| - 4.$$

Beweis: $|V| \geq 3$, G einfach

$$\Rightarrow \text{Jede Flächengrenze hat wenigstens 3 Kanten}$$

$$\Rightarrow 2|E| \overset{\text{Lemma}}{=} \sum_{f \in F} l(f) \geq 3|F| \Rightarrow |F| \geq \tfrac{2}{3}|E|$$

Einsetzen in Euler-Formel: $|E| = |V| + |F| - 2 \leq |V| + \frac{2}{3}|E| - 2$

$$\Rightarrow \tfrac{1}{3}|E| \leq |V| - 2$$

$$\Rightarrow |E| \leq 3|V| - 6$$

G dreiecksfrei $\Rightarrow (\forall f \in F)\ l(f) \geq 4$

$$\Rightarrow 2|E| = \sum_{f \in F} l(f) \geq 4|F| \Rightarrow |F| \leq \frac{1}{2}|E|$$

$$\Rightarrow |E| = |V| + |F| - 2$$

$$\leq |V| + \frac{1}{2}|E| - 2$$

$$\Rightarrow \frac{1}{2} \leq |V| - 2$$

$$\Rightarrow |E| \leq 2|V| - 4$$

$\square$

Korollar 6.5.4

Die Kuratowski-Graphen K_5 und $K_{3,3}$ sind nicht planar.

Beweis:

K_5: $10 = |E| > 3|V| - 6 = 3 \cdot 5 - 6 = 9$

$K_{3,3}$: $9 = |E| > 2|V| - 4 = 2 \cdot 6 - 4 = 8$ (dreiecksfrei)

□

Definition 6.5.5
*Ein **maximal planarer Graph** ist ein einfacher planarer Subgraph, der nicht aufspannender Subgraph eines anderen planaren Graphen ist. (Hinzufügen einer neuen Kante zerstört Planarität.) Eine **Triangulation** ist ein einfacher planar eingebetteter Graph in dem jede Fläche durch einen 3-Kreis begrenzt wird.*

Satz 6.5.6
Für einen einfachen planar eingebetteten Graphen $G = (V, E)$ sind folgende Aussagen äquivalent:

A) $|E| = 3|V| - 6$

B) G ist eine Triangulation.

C) G ist ein maximal planarer Graph.

Beweis: $A \Leftrightarrow B$: Beweis vorhin $|E| = 3|V| - 6 \Leftrightarrow 2|E| = 3|F| \Leftrightarrow$ Jede Fläche ist durch 3-Kreis begrenzt.
$B \Leftrightarrow C$: $\exists$ Fläche $f \in F$ mit $l(f) > 3 \Leftrightarrow \exists$ Kante $e \neq E$ deren Hinzufügen Planarität erhält.

6.5.2 Anwendung: Reguläre Polyeder

- Flächen: reguläre Polygone gleicher Länge.
- An jeder Ecke gleiche Anzahl von Flächen, „platonische Körper“.

Das Expandieren auf Kugeloberfläche („aufblasen“) gefolgt von Riemann-Projektion mit Nordpol im Inneren einer Seitenfläche ergibt regulären (gleiche Knotengrade) planar eingebetteten Graphen G mit gleichen $l(f)$ $(\forall f \in F) \Rightarrow$ dualer Graph G^* ist auch regulär.

Sei also $G = (V, E)$

- K-regulär (d.h. $\Delta(v) = K \ \forall v \in V$)
- planar eingebettet, so daß
$$l(f) = L \ \forall f \in F.$$

$$\overset{\text{Lemma}}{\Rightarrow} K \cdot |V| = 2|E| = L \cdot |F|$$
$$\overset{\text{Eulerformel}}{\Rightarrow} 2 = |V| - |E| + |F|$$
$$= \frac{2}{k}|E| - |E| + \frac{2}{L}|E|$$
$$= |E| \left(\frac{2}{k} - 1 + \frac{2}{L} \right)$$

Wegen $|E| > 0$ gilt

$$\frac{2}{K} - 1 + \frac{2}{L} > 0 \Rightarrow \frac{2}{K} + \frac{2}{L} > 1$$
$$\Rightarrow 2L + 2K > KL$$
$$\Rightarrow KL - 2L - 2K < 0$$
$$\Rightarrow \boxed{(K-2)(L-2) < 4}$$

$K < 3$: keine 3-D Polyeder
$L < 3$: keine 2-D Seitenflächen
Also: $\boxed{K \geq 3, L \geq 3}$
Wegen $(K-2)(L-2) < 4$ gilt $\boxed{K \leq 5, L \leq 5}$.

Überprüfung der Paare K, L:

K	L	$(K-2)(L-2)$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert F\rvert$	Name
3	3	$1 \cdot 1 = 1 < 4$	4	6	4	Tetraeder
3	4	$1 \cdot 2 = 2 < 4$	8	12	6	Hexaeder (Würfel)
3	5	$1 \cdot 3 = 3 < 4$	20	30	12	Dedekaeder
4	3	$2 \cdot 1 = 2 < 4$	6	12	8	Oktaeder
4	4	$2 \cdot 2 = 4 \geq 4$				
4	5	$2 \cdot 3 = 6 \geq 4$				
5	3	$3 \cdot 1 = 3 < 4$	12	30	20	Ikosaeder
5	4	$3 \cdot 2 = 6 \geq 4$				
5	5	$3 \cdot 3 = 9 \geq 4$				

Kapitel 7

Planaritätstest

7.1 Einführung

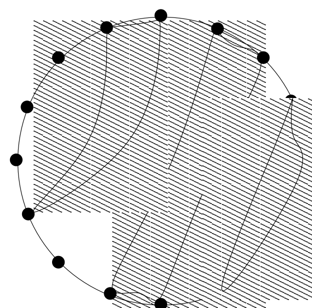
Lemma 7.1.1
Ein Graph G ist planar genau dann, wenn jede Zusammenhangskomponente planar ist.

Lemma 7.1.2
Ein zusammenhängender Graph G ist planar genau dann, wenn jede 2-Zusammenhangskomponente planar ist.

Wegen Lemma 7.1.1 und 7.1.2 sei von nun an ohne Einschränkung G 2-zusammenhängend.

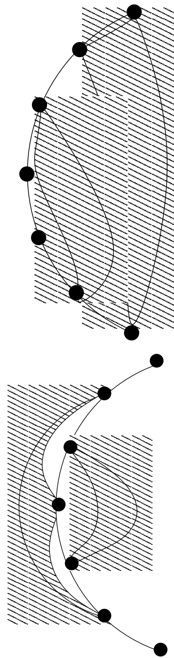
7.1.1 Das Divide und Conquer Konzept

Sei $C = (V_c, E_c)$ ein Kreis in G. Dann partitioniere die Kanten $E \backslash E_c$ wie folgt: $e_1, e_2 \in E \backslash E_c$ sind in derselben Klasse $\Leftrightarrow$ es existiert ein Pfad zwischen e_1 und e_2 und der Pfad enthält keinen Knoten aus dem Kreis V_c.



Vorgehen:

1. Wähle einen (geeigneten) Kreis
2. Teste die so entstehenden Partitionen rekursiv auf Planarität
3. Falls alle Partitionen planar sind, teste ob es eine Anordnung der Partitionen um den Kreis herum gibt, die planar ist.



Idee A: Falls Partitionen sich (paarweise) nicht kreuzen, so können diese auf derselben Seite des Kreises plaziert werden.

Idee B: Falls Partitionen sich kreuzen, müssen sie auf verschiedenen Seiten des Kreise eingebettet werden.

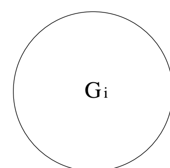
1974 haben Hopcroft, Tarjan eine Algorithmus angegeben, der auf diesem Verfahren beruht. Erst 1996 haben Mehlhorn und Mutzel diesen Algorithmus zu einem Einbettungsverfahren erweitert (nicht trivial!). Das Verfahren ist nämlich extrem schwierig und technisch sehr aufwendig.

7.1.2 Das inkrementelle Konzept

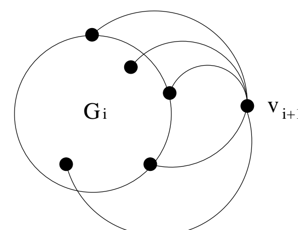
Sei $G = (V, E)$ ein Graph mit $n = |V|$ Knoten. Starte mit einem Knoten v_1 und betrachte eine Folge von Knoten $v_1, \ldots, v_n$ und eine dadurch induzierte Folge von induzierten Untergraphen

$$\begin{aligned} G_1 &= (V_1 = \{v_1\}, E_1) \\ G_2 &= (V_2 = \{v_1, v_2\}, E_2) \\ &\vdots \\ G_n &= (V_n = \{v_1, \ldots, v_n\}, E_n) = G \end{aligned}$$

Teste in jedem Schritt $i = 1, \ldots, n-1$, ob das Hinzufügen des Knotens v_{i+1} zu G_i mit seinen zu Knoten aus V_i inzidenten Kanten die Planarität verletzt.



planar



planar ?

$O(n^2)$	Planaritätstest von Lempel, Even und Cederbaum 1967
$O(n)$	Planaritätstest von Booth und Lücker 1976 mit Hilfe einer Datenstruktur „PQ-Baum“
$O(n)$	Einbettungsalgorithmus von Chiba, Nishizeki, Abe, Ozava 1985

Verfahren ist „nur“ sehr schwierig, dafür technisch extrem aufwendig. Weltweit existieren nur wenige korrekte Implementierungen.

7.2 Der Planaritätstest auf der Grundlage der PQ-Bäume

In diesem Abschnitt werden wir nun, nach der gründlichen Vorstellung der Datenstruktur der PQ-Bäume, den Eckenadditionsalgorithmus von Lempel, Even und Cederbaum [LEC67] vorstellen und dies mit der Anwendung der PQ-Bäume in diesem Algorithmus verbinden. Die PQ-Bäume sind zwar nicht notwendig für die erfolgreiche Anwendung eines Eckenadditionsalgorithmus, durch sie kommt aber erst das gute Laufzeitverhalten zustande.
Ein Planaritätstest entscheidet, ob ein gegebener Graph $G = (V, E)$ planar ist. Nach Harary [Har72] ist ein Graph planar genau dann, wenn seine 2-fach zusammenhängenden Komponenten planar sind. Daher beschränken wir unsere Untersuchungen von nun an auf einfache, 2-fach zusammenhängende Graphen $G = (V, E)$. Wir können ferner annehmen, daß die Anzahl der Kanten m durch $3n-6$ beschränkt ist, da andernfalls der Graph nach dem Korollar 6.5 zu Eulers Formel nicht planar ist.
Die Idee des Planaritätstests von Lempel, Even und Cederbaum [LEC67] ist, mit einer Ecke v_1 des zu testenden Graphen als induzierende Eckenmenge zu starten und sukzessive Ecken v_i zu dieser Eckenmenge zu addieren und gleichzeitig den dadurch induzierten Untergraphen auf Planarität zu testen. Beim Test des durch die Eckenmenge $\{v_1, v_2, \ldots, v_{k-1}, v_k\}$ induzierten Untergraphen macht man sich in gewisser Weise zunutze, daß der durch die Eckenmenge $\{v_1, v_2, \ldots, v_{k-1}\}$ induzierte Untergraph planar ist. Es braucht also bei der Addition der Ecke v_k zur Menge $\{v_1, v_2, \ldots, v_{k-1}\}$ nur getestet werden, ob die dadurch zusätzlich eingefügten Kanten die Planarität des induzierten Untergraphen verletzen. Ist dies der Fall, so ist G nicht planar und der Planaritätstest kann abgebrochen werden. Andernfalls wird eine weitere Ecke zur induzierenden Eckenmenge addiert und der Test wiederholt. Dies wird solange durchgeführt, bis der durch die Eckenmenge induzierte Untergraph gleich dem zu testenden Graphen ist.
Die folgende Definition beschreibt eine Numerierung der Knoten von G, die wesentlich für den Planaritätstest ist. Sie bestimmt, in welcher Reihenfolge die Knoten von G zur induzierenden Eckenmenge addiert werden.

Definition 7.2.1
Sei $G = (V, E)$ ein 2-fach zusammenhängender Graph mit. Eine Numerierung $\omega : V \to \{1, \ldots, n\}$ heißt **st-Numerierung** *genau dann, wenn*

1. *die beiden Knoten $v_1, v_2 \in V$ mit $\omega(v_1) = 1$ und $\omega(v_2) = n$ zueinander adjazent sind.*

2. *für alle Knoten $v_i \in V$ mit $\omega(v_i) = i$, $1 < i < n$ existieren zwei Nachbarn $v_j, v_k \in V$ mit $\omega(v_j) = j$, $\omega(v_k) = k$, so daß $j < i < k$.*

v_1 heißt **Quelle** *und wird mit* **s** *bezeichnet, während v_t* **Senke** *heißt und mit* **t** *bezeichnet wird.*

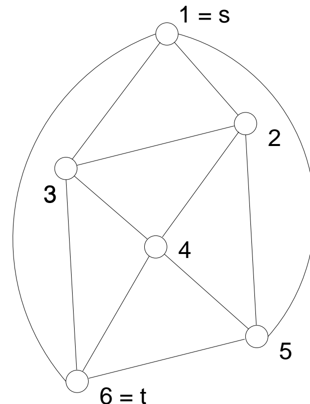


Abbildung 7.1: Beispiel eines Graphen mit einer st-Numerierung.

Jeder 2-fach zusammenhängende Graph besitzt eine solche Numerierung und nach Even und Tarjan [ET76] kann eine solche st-Numerierung eines beliebigen 2-fach zusammenhängenden Graphen in linearer Zeit berechnet werden. Der von ihnen präsentierte Algorithmus basiert auf einem Tiefensuchverfahren (Depth-First-Search) und geht von einem beliebigen Paar benachbarter Ecken aus, die als Quelle und Senke markiert werden.

Wir nehmen an, daß der Graph G eine st-Numerierung besitzt und verwenden von nun an die st-Nummer einer Ecke, um uns auf diese Ecke zu beziehen.

Bemerkung 7.2.2
Die st-Numerierung der Ecken eines Graphen G induziert einen gerichteten Graphen, indem alle Kanten (j, k), $j < k$, von der kleineren Ecke j zur größeren Ecke k hin orientiert werden. Häufig sprechen wir daher auch von den abgehenden Kanten beziehungsweise von den ankommenden Kanten einer Ecke k und meinen damit alle Kanten (k, l) mit $k < l$ beziehungsweise (j, k) mit $j < k$.

Es sei $\mathbf{G_k} = (\mathbf{V_k}, \mathbf{E_k})$ der durch die Ecken $\{1, 2, \dots, k\}$ induzierte Untergraph von G. Ist $k < n$, so existiert nach der Definition der st-Numerierung eine Ecke l, mit $l > k$, die zur Ecke k adjazent ist. Also existiert mindestens eine Kante zwischen den Ecken aus V_k und den Ecken aus $V \setminus V_k$. Sei $\mathbf{G'_k}$ der Graph, der entsteht, indem alle Kanten zwischen Ecken aus V_k und Ecken aus $V \setminus V_k$ zu G_k hinzuaddiert werden, wobei die Enden aus $V \setminus V_k$ getrennt bleiben. Existieren also in V_k zwei verschiedene Ecken i und j, die beide denselben Nachbar $l \in V \setminus V_k$ haben, so werden zu G_k die Kanten (i, l) und (j, l) addiert. Gleichzeitig werden aber die beiden Enden, die zur Ecke l korrespondieren, nicht miteinander verbunden. Diese Kanten heißen **virtuelle Kanten** und ihre losen Enden heißen **virtuelle Ecken**. Die virtuellen Ecken erhalten dieselbe Nummer wie die

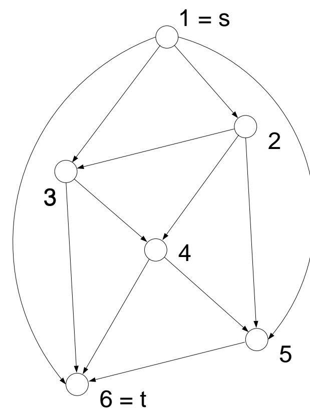


Abbildung 7.2: Der durch die st-Numerierung induzierte, gerichtete Graph.

zu ihnen korrespondierende Ecke aus $V \setminus V_k$. Also erhält sowohl die zur virtuellen Kante (i, l) inzidente virtuelle Ecke, als auch die zu der Kante (j, l) inzidente virtuelle Ecke die Nummer l. In G'_k kann es daher mehrere virtuelle Ecken mit derselben Nummer geben.

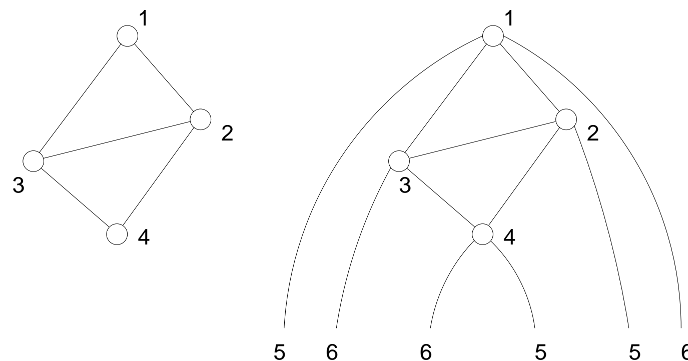


Abbildung 7.3: Links ist der Untergraph G_4 unseres Beispiels dargestellt. Die Buschform B_4 rechts ist die Einbettung des Graphen G'_4, bei der die virtuellen Ecken von G'_4 im äußeren Land plaziert werden.

Existiert eine Einbettung von G'_k, bei der alle virtuellen Ecken im äußeren Land plaziert werden, so nennen wir diese Einbettung eine **Buschform** des Graphen G'_k und bezeichnen sie mit $\mathbf{B_k}$. In einer graphischen Darstellung einer solchen Buschform werden die virtuellen Ecken üblicherweise auf einer horizontalen Linie unterhalb der Buschform plaziert. G_k bezeichnen wir auch als **Rumpf** der Buschform B_k.
Es sei $\tilde{G}$ ein durch die Eckenmenge $\{j, j+1, \ldots, k\}$, $1 \leq j \leq k$, induzierter Untergraph von G_k. Es sei ferner $\tilde{G}'$ der Graph, der entsteht, indem alle Kanten zwischen Ecken aus $\{j, j+1, \ldots, k\}$ und Ecken aus $V \setminus V_k$ zu $\tilde{G}$ hinzuaddiert werden, wobei die Enden in $V \setminus V_k$ getrennt gelassen werden. Eine **Unterbuschform** $\tilde{B}$ einer Buschform B, ist die Einbettung eines solchen Untergraphen $\tilde{G}'$, wobei $\tilde{B}$ mit der Einbettung von $\tilde{G}'$ in B übereinstimmt.

Zu bemerken ist, daß die Plazierung der virtuellen Ecken im äußeren Land für die Buschform wesentlich ist. Da es sich bei der Buschform um eine Einbettung handelt, darf es auch bei der Plazierung der virtuellen Ecken ins äußere Land nicht zu Kreuzungen von virtuellen Kanten und normalen Kanten kommen. Offensichtlich besitzt daher nicht jeder Graph G'_k eine Buschform B_k.
Das nun folgende Lemma nach Even [Eve79] impliziert, daß jeder planare Graph Buschformen B_k, $1 \leq k \leq n$, besitzt:

Lemma 7.2.3
Wird die Kante (s, t) auf der Grenze zum äußeren Land einer Einbettung des planaren Graphen G gezeichnet, so werden in dieser Einbettung alle Ecken und Kanten aus $G - G_k$, $1 \leq k \leq n$, in das äußere Land des ebenen Untergraphen G_k von G gezeichnet.

Beweis:
Wir nehmen an, daß es Ecken $\{v_1, v_2, \ldots, v_l\}$ aus $V - V_k$ gibt, die in einem Land F von G_k liegen. Alle diese Ecken haben aufgrund der st-Numerierung eine höhere Nummer, als die Ecken auf der Grenze von F. Sei $v_i \in \{v_1, v_2, \ldots, v_l\}$ die Ecke mit der größten st-Nummer.
Wir zeigen, daß diese Ecke v_i identisch mit der Senke t von G ist. Hat der ebene Untergraph G_k nur ein Land, so ist dies trivial. Hat G_k mindestens zwei Länder, so nehmen wir an, daß die Senke sich in einem Land $F' \neq F$ befindet. Die st-Numerierung impliziert, daß ein Pfad $(y_1, y_2, \ldots, y_\nu)$ mit $y_1 = v_i$ und $y_\nu = t$ existiert, wobei $v_i < y_2 < y_3 < \ldots < y_{\nu-1} < t$. Da v_i und t beide in verschiedenen Ländern von G_k liegen, schneidet dieser Pfad zumindest die Grenze des Landes F. Die Ecken auf der Grenze von F sind aber sämtlich kleiner als v_i. Also kreuzt mindestens eine Kante des Pfades eine Kante auf der Grenze von F. Dies ist ein Widerspruch zur Voraussetzung, daß der Graph G eingebettet wurde. Also liegen v_i und t im selben Land und da v_i maximal gewählt wurde, ist $v_i = t$. Nach Voraussetzung befindet sich die Kante (s, t) auf der Grenze des äußeren Landes der Einbettung von G, folglich befindet sich auch t auf der Grenze des äußeren Landes von G, also wird t in das äußere Land des ebenen Untergraphen von G_k gezeichnet. F ist somit identisch mit dem äußeren Land von G_k. $\square$
Das Lemma 7.2.3 hat weitreichende Konsequenzen. Ist ein Graph planar, so existiert nach dem Lemma bei einer *beliebigen* st-Numerierung eine Einbettung von G, so daß für alle Untergraphen G_k, $1 \leq k \leq n$, in dieser Einbettung alle Ecken und Kanten aus $G - G_k$ in dem äußeren Land von G_k gezeichnet werden. Daraus folgt, daß für alle G_k, $1 \leq k \leq n$, eine Einbettung von G'_k existiert, so daß alle virtuellen Ecken im äußeren Land von G_k plaziert werden können. Also existiert zu jedem G_k, $1 \leq k \leq n$, eine Buschform B_k. Wir können somit folgendes Korollar formulieren:

Korollar 7.2.4
Ist G ein planarer, st-numerierter Graph, dann existiert die Folge der Buschformen $B_1, B_2, \ldots, B_n$.

Bemerkung 7.2.5
Ist G nicht planar, so gibt es ein k, $1 \leq k \leq n$, so daß die Buschform B_k nicht existiert.

Beweis:
Da $G_n = G$ ist, existiert ein k, $1 \leq k \leq n$, so daß G_k nicht planar ist. Daraus folgt, daß G_k nicht eingebettet werden kann . Somit kann auch G'_k nicht eingebettet werden. Also existiert die Buschform B_k nicht. □

Die ursprüngliche Idee des Eckenadditionsalgorithmus war, festzustellen ob die einzelnen, durch eine Eckenmenge induzierten Untergraphen G_k, $1 \leq k \leq n$, planar sind. Diese Frage läßt sich nach dem vorangegangenen Korollar und dem Lemma 7.2.3 auf die Existenz der Buschformen $B_1, B_2, \ldots, B_n$ reduzieren. Es genügt also, um einen Graphen auf Planarität zu testen, festzustellen, ob alle Buschformen existieren. Die eigentliche Idee des Eckenadditionsalgorithmus von Lempel, Even und Cederbaum [LEC67] ist nun, die Frage der Existenz einer Buschform B_{k+1}, $1 \leq k < n$, auf ein Problem zu transformieren, in dem nach Permutationen und Reversionen innerhalb der Buschform B_k gesucht wird, so daß nach Anwendung dieser Permutationen und Reversionen alle virtuellen Ecken der Nummer $k + 1$ eine zusammenhängende Folge innerhalb der horizontalen Plazierung aller virtuellen Ecken im äußeren Land von G_k bilden.

Die Permutationen und Reversionen orientieren sich an den Schnittecken und Blöcken einer Buschform B_k, das heißt sie werden in der Einbettung des G'_k angewendet. Der Graph G'_k besitzt keine eindeutig bestimmte Buschform B_k, da die Schnittecken die zu ihnen adjazenten Blöcke beliebig um sich herum gruppieren können, während die Blöcke von G'_k umgekehrt werden können. Diese beliebigen Anordnungen bei Schnittecken beziehungsweise die Umkehrungen bei Blöcken sind gerade die Permutationen beziehungsweise Reversionen.

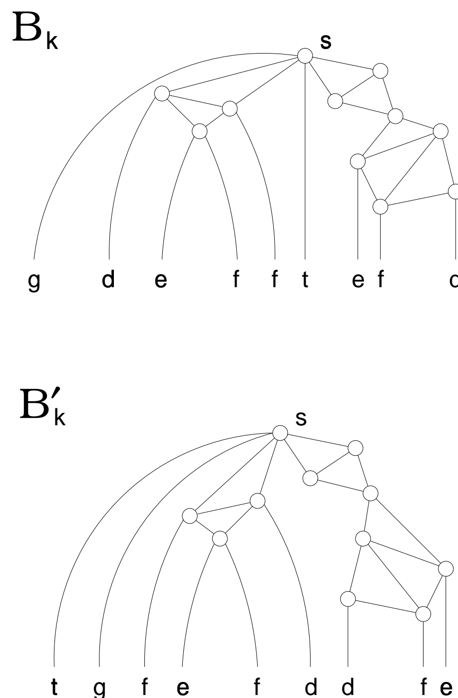


Abbildung 7.4: Zwei verschiedene Buschformen B_k und B'_k eines Untergraphen G_k mit virtuellen Ecken d, e, f, g, und t.

Es ist leicht, aus G'_k den Graphen G'_{k+1} zu konstruieren. Um den Rumpf G_{k+1} zu erzeugen,

brauchen dazu lediglich die losen Enden aller virtuellen Kanten, die zu einer virtuellen Ecke $k+1$ inzident sind, miteinander verbunden zu werden. Anschließend werden die virtuellen Kanten $(k+1, w) \in E$, $w \in V \setminus V_{k+1}$ eingefügt und wir erhalten den Graphen G'_{k+1}.
Leider gibt die beschriebene Konstruktion keine Auskunft darüber, ob eine Buschform B_{k+1} existiert. Wir werden daher eine Buschform B_{k+1}, falls sie existiert, direkt aus der Buschform B_k konstruieren. Eine solche Konstruktion gelingt uns dann, wenn eine Buschform von G'_k existiert, in der die virtuellen Ecken der Nummer $k+1$ in der horizontalen Front aller virtuellen Ecken von G'_k eine zusammenhängende Folge bilden. In diesem Fall gilt nämlich das folgende Lemma:

Lemma 7.2.6
Eine Buschform B_{k+1}, $1 \leq k < n$, existiert genau dann, wenn eine Buschform B_k existiert, in der die virtuellen Ecken der Nummer $k+1$ eine zusammenhängende Folge in der Folge aller virtuellen Ecken bilden.

Beweis:
Es sei also eine Buschform B_k gegeben, in der die virtuellen Ecken der Nummer $k+1$ eine zusammenhängende Folge innerhalb der Folge aller virtuellen Ecken einnehmen. Das bedeutet, in der horizontalen Plazierung der virtuellen Ecken im äußeren Land der Einbettung von G'_k werden die virtuellen Ecken der Nummer $k+1$ durch keine andere virtuelle Ecke voneinander getrennt. Nun sind die virtuellen Kanten $(v, k+1) \in E$, $v \in V_k$, gerade die Kanten, die neu hinzugefügt werden, wenn zu G'_k die Ecke $k+1$ addiert wird, um G'_{k+1} zu erzeugen. Da in der Einbettung B_k die virtuellen Ecken der Nummer $k+1$ eine zusammenhängende Folge bilden, können die losen Enden der virtuellen Kanten $(v, k+1) \in E$, $v \in V_k$, in der Einbettung von B_k miteinander verbunden werden, ohne daß sich Kanten schneiden. Damit ist es uns gelungen, die Ecke $k+1$ mit ihren ankommenden Kanten innerhalb der Einbettung von B_k einzubetten. Offensichtlich befindet sich die so eingebettete Ecke auf der Grenze des äußeren Landes. Daher können nun die virtuellen Kanten $(k+1, w) \in E$, $w \in V \setminus V_{k+1}$, ebenfalls in das äußere Land eingebettet werden. Die so konstruierte Einbettung ist eine Buschform B_{k+1}.
Ist umgekehrt eine Buschform B_{k+1} gegeben, so kann daraus eine Buschform B_k konstruiert werden, in der die virtuellen Kanten $(v, k+1) \in E$, $v \in V_k$, eine zusammenhängende Folge bilden. Dazu entfernen wir lediglich alle virtuellen Kanten $(k+1, w) \in E$, $w \in V \setminus V_k$, aus der Einbettung. Da nach der st-Numerierung mindestens eine solche virtuelle Kante existiert und diese nach Voraussetzung im äußersten Land von B_{k+1} eingebettet ist, befindet sich die Ecke $k+1$ auf der Grenze des äußeren Landes. Durch Entfernen der Ecke und Einführen virtueller Kanten anstelle der ankommenden Kanten der Ecke $k+1$ erhält man das Gewünschte. □
Das Problem der Überprüfung der Existenz der Buschform B_{k+1} reduziert sich dadurch auf die Frage: Gibt es eine Buschform B_k, in der die virtuellen Ecken der Nummer $k+1$ eine zusammenhängende Folge bilden? Nun erfüllt die vorliegende Buschform B_k diese Anforderung im allgemeinen nicht. Daher muß diese Frage anders gestellt werden: Kann die vorliegende Buschform so verändert werden, daß sie anschließend immer noch eine Buschform ist und die virtuellen Ecken der Nummer $k+1$ eine zusammenhängende Folge bilden? Die möglichen Veränderungen, die vorgenommen werden dürfen, sind die bereits

angesprochenen Permutationen von Blöcken bei Schnittecken und die Umkehrung der eingebetteten Blöcke. Das Korollar des folgenden Lemmas nach Even [Eve79] wird diese Vermutungen bestätigen:

Lemma 7.2.7
Es sei G_k ein Untergraph eines planaren Graphen G. Es seien B_{k_α} und B_{k_β} zwei verschiedene Buschformen von G_k. Ferner sei B_α eine Unterbuschform von B_{k_α} und B_β ein Unterbuschform von B_{k_β} derart, daß B_α und B_β dieselben Ecken gemeinsam haben. Dann existiert eine Folge von Permutationen und Reversionen, mit denen die Unterbuschform B_β in eine Unterbuschform B_γ transformiert werden kann, so daß in B_γ die virtuellen Ecken auf der Horizontalen in derselben Reihenfolge erscheinen, wie in B_α.

Beweis:
Wir beweisen das Lemma mittels Induktion über die Anzahl der Ecken in der Buschform. Bestehen die beiden Unterbuschformen nur aus einer Ecke zusammen mit den adjazenten, virtuellen Ecken, so ist die Behauptung trivialerweise erfüllt.
Es sei daher die Behauptung für alle Unterbuschformen B_α und B_β mit $l-1$, $l < k$, Ecken erfüllt, und es seien B_α und B_β Unterbuschformen mit l Ecken. Es sei v die kleinste Ecke gemäß der st-Numerierung in B_α und B_β.

1.Fall: v ist eine Schnittecke in den beiden Unterbuschformen. Dann zerfallen die beiden Unterbuschformen durch Entfernen der Ecke v in Zusammenhangskomponenten $B_{\alpha_1}, B_{\alpha_2}, \ldots, B_{\alpha_i}$ beziehungsweise $B_{\beta_1}, B_{\beta_2}, \ldots, B_{\beta_i}$. Diese Zusammenhangskomponenten sind wegen der Minimalität von v Unterbuschformen von B_α beziehungsweise B_β. Sollten sie in B_β nicht in derselben Reihenfolge erscheinen wie in B_α, so kann, da v eine Schnittecke ist, durch geeignete Permutation der $B_{\beta_1}, B_{\beta_2}, \ldots, B_{\beta_i}$ eine Unterbuschform B'_γ erzeugt werden, in der die $B_{\beta_1}, B_{\beta_2}, \ldots, B_{\beta_i}$ dieselbe Reihenfolge einnehmen wie die $B_{\alpha_1}, B_{\alpha_2}, \ldots, B_{\alpha_i}$ in B_α. Da die $B_{\beta_1}, B_{\beta_2}, \ldots, B_{\beta_i}$ Unterbuschformen mit maximal $l-1$ Ecken sind, existieren nach Induktionsvoraussetzung Permutationen und Reversionen, mit denen diese Unterbuschformen in Unterbuschformen $B_{\gamma_1}, B_{\gamma_2}, \ldots, B_{\gamma_i}$ transformiert werden können, in denen die virtuellen Ecken in derselben Reihenfolge auf der Horizontalen erscheinen, wie in den Unterbuschformen $B_{\alpha_1}, B_{\alpha_2}, \ldots, B_{\alpha_i}$. Das Resultat ist die gesuchte Unterbuschform B_γ.

2.Fall: v ist keine Schnittecke in den beiden Unterbuschformen. Dann existiert eine 2-fach zusammenhängende Komponente, in der die Ecke v enthalten ist. Sei H die größte von all diesen 2-fach zusammenhängenden Komponenten. Seien ferner $u_1, u_2, \ldots, u_j$ die Schnittecken in B_α auf der Grenze des äußeren Landes von H. Diese Ecken erscheinen auf der Grenze des äußeren Landes von H in B_β in derselben Reihenfolge wie in B_α oder in umgekehrter Reihenfolge. Sollte die Reihenfolge umgekehrt sein, so wird eine Reversion von H vorgenommen: H wird um v herum gedreht. Zu jeder dieser Schnittecken $u_1, u_2, \ldots, u_j$ ist mindestens eine weitere Komponente adjazent. Jede dieser Komponenten ist aufgrund der Minimalität von v eine Unterbuschform mit maximal $l-1$ Ecken. Also erhält man wie im 1.Fall nach Induktionsvoraussetzung die Behauptung.

□

Korollar 7.2.8
Sei B_k eine beliebige Buschform eines Untergraphen G_k eines planaren Graphen G. Dann existiert eine Folge von Permutationen und Reversionen, so daß die virtuellen Ecken der Nummer $k+1$ eine zusammenhängende Folge einnehmen.

Beweis:
Nach Korollar 7.2.4 existiert die Buschform B_{k+1}. Nach Lemma 7.2.6 existiert dann eine Buschform $\tilde{B_k}$, in der die virtuellen Knoten der Nummer $k+1$ eine zusammenhängende Folge bilden. Nach Lemma 7.2.7 existiert dann für eine beliebige Buschform B_k eine Folge von Permutationen und Reversionen, so daß B_k in eine Unterbuschform B'_k transformiert werden kann, in der die virtuellen Ecken auf der Horizontalen in derselben Reihenfolge wie in $\tilde{B_k}$ erscheinen. □

Der Weg für einen brauchbaren Ansatz eines Eckenadditionsalgorithmus ist nun geebnet. Nach Korollar 7.2.4 wissen wir, daß jeder 2-fach zusammenhängende, planare Graph G Buschformen B_k, $1 \leq k \leq n$, besitzt. Nach Korollar 7.2.8 kann jede der Buschformen B_k, $1 \leq k < n$, in eine Buschform B'_k transformiert werden, so daß die virtuellen Ecken $k+1$ eine zusammenhängende Folge einnehmen. Damit ergibt sich folgender Planaritätstest für einen 2-fach zusammenhängenden Graphen:

```
Boolean procedure PLANAR(G);
begin
  Berechne eine st-Numerierung der Ecken von G;
  Konstruiere die Buschform B_1;
  for k := 1 to n − 1
    begin
      Versuche aus B_k eine Buschform B'_k zu konstruieren, in der alle
        Ecken der Nummer k + 1 eine zusammenhängende Folge bilden;
      if Konstruktion von B'_k schlägt fehl then
        begin
          Breche Algorithmus ab;
          return FALSE;
        end; {if}
      else
        Konstruiere die Buschform B_{k+1} aus B'_k;
    end; {for}
  return TRUE;
end {begin}
```

Bemerkung: LEC konnten mit diesem Ansatz ein Verfahren mit $O(n^2)$ Laufzeit entwickeln.

7.3 PQ-Bäume

Die PQ-Bäume wurden von Booth und Lueker [BL76] eingeführt, um Probleme zu lösen, bei denen auf einer Menge U von Objekten bestimmte Permutationen gesucht werden,

die gewisse Restriktionen beachten. Permutationen, die solche Restriktionen beachten, werden als **zulässige** Permutationen bezeichnet. Welche Permutationen zulässig sind, ist zwar weitestgehend durch die problemabhängigen Restriktionen gegeben, sie basieren aber bei den von Booth und Lueker betrachteten Problemen auf dem folgenden Grundsatz:

> Von den bereits bekannten zulässigen Permutationen einer Menge U lassen wir nur diejenigen Permutationen zu, bei denen die Elemente einer Teilmenge $S \subset U$ eine zusammenhängende Folge einnehmen.

Eine **Restriktion** entspricht demnach einer Teilmenge $S \subset U$, in der je zwei Elemente aus S nicht durch ein Element aus $U \setminus S$ getrennt werden dürfen. Alle Permutationen, bei denen eine Trennung von zwei Elementen aus S durch ein fremdes Element vorliegt, müssen verboten werden, während die zulässigen Permutationen genau diejenigen sind, bei denen alle Elemente aus S eine **zusammenhängende Folge** bilden.
Der PQ-Baum ist eine Datenstruktur und wird verwendet, um die Klasse aller zulässigen Permutationen der Menge U zu repräsentieren. Ferner ermöglicht uns der PQ-Baum die Klasse der zulässigen Permutationen weiter zu beschränken, falls eine zusätzliche Restriktion in Form einer weiteren Teilmenge S' eingeführt werden soll. Ist eine Menge U mit zulässigen Permutationen bezogen auf Teilmengen $S \subset U$ gegeben, so muß die Klasse der zulässigen Permutationen bei Einführung einer weiteren Menge S' reduziert werden. Das heißt, gesucht wird eine Teilmenge der bis dato zulässigen Permutationen, bei denen die Elemente von S' eine zusammenhängende Folge einnehmen. Gefunden wird diese Teilmenge, indem durch noch näher zu definierende Operationen auf dem entsprechenden PQ-Baum die Elemente von S' in allen Permutationen zusammengehalten werden. Man spricht bei den entsprechenden Operationen auf dem PQ-Baum von einer **Reduktion** in Bezug auf die Menge S, oder einfacher von der Reduktion der Menge S.
Ziel ist es, bei einem Problem mit gegebenen Restriktionen festzustellen, ob überhaupt zulässige Permutationen auf der dem Problem zugrundeliegenden Menge U von Objekten existieren, bei denen alle diese Restriktionen erfüllt sind. Zusätzlich sollen diese zulässigen Permutationen, falls sie existieren, angegeben werden. Die Restriktionen lassen sich durch Teilmengen $S_1, S_2, \ldots, S_k \subset U$ darstellen, deren Elemente jeweils eine zusammenhängende Folge in U einnehmen sollen. Um nun festzustellen, ob zulässige Permutationen existieren, in denen für jede Teilmenge S_i, $i \in \{1, 2, \ldots, k\}$, alle Elemente eine zusammenhängende Folge bilden, geht man sukzessive vor. Dazu startet man mit der Menge aller Permutationen über der Menge U als die zulässigen Permutationen und überprüft zunächst für S_1, ob zulässige Permutationen in U existieren, bei denen alle Elemente von S_1 eine zusammenhängende Folge bilden. Sollte dies der Fall sein, und dies ist für S_1 trivialerweise immer der Fall, so erlauben wir von nun an nur noch die Permutationen, bei denen die Elemente von S_1 eine zusammenhängende Folge bilden. Wir haben also die Menge S_1 reduziert und die Menge der ursprünglich zulässigen Permutationen ist kleiner geworden. Mit den Mengen S_2 bis S_k wird ebenso verfahren, wobei für jede Menge S_i, $i \in \{2, 3, \ldots, k\}$, die Grundlage zur Bestimmung der zulässigen Permutationen gerade die Permutationen sind, in denen die Elemente der Mengen S_j, $1 \leq j < i$, eine zusammenhängende Folge bilden.
Wir wollen nun die verwendete PQ-Baum Datenstruktur definieren. Um dies in vernünftiger Weise durchführen zu können, benötigen wir zunächst einige allgemein gehaltene

Informationen über Bäume als Datenstruktur, wie sie unter anderem bei Cormen, Leiserson und Rivest zu finden sind [CLR89]. Grundlage dieser Datenstruktur bildet der graphentheoretische Begriff des Baumes.

Definition 7.3.1
Ein **verwurzelter Baum** *ist ein Baum mit einer ausgezeichneten Ecke. Diese ausgezeichnete Ecke wird als* **Wurzel** *bezeichnet. Die Ecken eines verwurzelten Baumes nennen wir* **Knoten**.

Bemerkung 7.3.2
Offensichtlich existiert für jede Ecke eines Baumes genau ein Pfad zu jeder anderen Ecke.

.
Sei X ein Knoten in einem verwurzelten Baum T. Sei R die Wurzel von T. Dann gibt es nach Bemerkung 7.3.2 genau einen Pfad von X zur Wurzel R. Alle Knoten auf diesem Pfad sind **Vorfahren** von X. Ist Y ein Knoten auf diesem Pfad, so heißt Y **Vorfahre** von X und X ist **Nachfahre** von Y. Gilt zusätzlich $X \neq Y$, so ist Y **echter Vorfahre** von X und X ist **echter Nachfahre** von Y. Ist Y ein echter Vorfahre von X und (Y, X) eine Kante im Baum, so wird Y üblicherweise als **Vater** von X und X als das **Kind** von Y bezeichnet.

Definition 7.3.3
Sei V' die Menge aller Nachfahren eines Knotens X des verwurzelten Baumes T. Sei T' der durch V' induzierte Unterbaum von T. T' heißt **verwurzelter Unterbaum** *von T genau dann, wenn X Wurzel von T' ist.*

Bis auf die Wurzel besitzt jeder Knoten X genau einen Vater im Baum. Haben zwei Knoten denselben Vater, so bezeichnen wir diese Knoten als **Geschwister**. Knoten, die keine Kinder besitzen, werden als **externe** Knoten oder **Blätter** bezeichnet, alle anderen Knoten heißen **interne** Knoten.
Wir vereinbaren, in Zukunft den Zusatz *echt* bei echten Vorfahren und echten Nachfahren zu unterdrücken.
Die Definition eines verwurzelten Baumes legt nicht fest, in welcher Reihenfolge die Kinder eines Knotens zu erscheinen haben. Zwei Bäume sind daher gleich, wenn sie dieselben Knoten und Kanten besitzen und derselbe Knoten ausgezeichnet ist. Die folgenden Bäume legen gleichzeitig eine Reihenfolge ihrer Kinder fest:

Definition 7.3.4
Ein **geordneter Baum** *ist ein verwurzelter Baum, bei dem die Kinder jedes Knotens geordnet sind.*

Zwei geordnete Bäume sind genau dann gleich, wenn die zugrundeliegenden verwurzelten Bäume gleich sind und die Reihenfolge, in der die Kinder jedes internen Knotens des einen Baumes erscheinen, mit der Reihenfolge der Kinder des gleichen Knotens des anderen Baumes übereinstimmt. In geordneten Bäumen heißen zwei Knoten zueinander **adjazent**, wenn sie Geschwister sind und in der Ordnung der Kinder ihres Vaters nebeneinander erscheinen.

Vielfach werden die geordneten Bäume als Datenstruktur verwendet, in denen die Knoten ganze Datensätze speichern. Die in diesem Abschnitt vorgestellten PQ-Bäume unterscheiden sich in dieser Hinsicht, da das Mitführen von Datensätzen den Blättern eines Baumes vorbehalten ist. Die internen Knoten haben eine völlig andere Funktion, die, wie wir im folgenden feststellen werden, geeignet ist, das eingangs vorgestellte Problem zu lösen.

Definition 7.3.5
Sei eine universelle Menge $U = \{a_1, a_2, \ldots a_m\}$ gegeben. Die Klasse aller **PQ-Bäume** *über diese Menge ist definiert als die Menge aller geordneten Bäume, deren Blätter Elemente aus U sind, und deren interne Knoten entweder* **P-Knoten** *oder* **Q-Knoten** *sind.*

Als Beispiel wird jede der folgenden drei Operationen einen legalen PQ-Baum konstruieren:

1. Jedes Element $a_i \in U$ ist ein PQ-Baum, dessen Wurzel das Element selber ist. Der Baum besteht aus einem einzigen Blatt und wir zeichnen ihn, indem wir lediglich das Element selbst darstellen.

2. Sind $T_1, T_2, \ldots, T_k$ PQ-Bäume, so ergibt der in Abbildung 7.5 dargestellte Baum einen PQ-Baum mit einem P-Knoten als Wurzel. Wir zeichnen den P-Knoten als Kreis und plazieren seine Kinder unterhalb des Kreises.

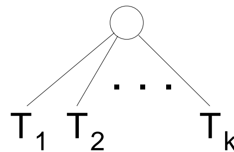


Abbildung 7.5: Ein P-Knoten mit den Kindern $T_1, T_2, \ldots, T_k$.

3. Sind $T_1, T_2, \ldots, T_k$ PQ-Bäume, so ergibt der in Abbildung 7.6 dargestellte Baum einen PQ-Baum, dessen Wurzel ein Q-Knoten ist. Wir zeichnen den Q-Knoten als Rechteck und plazieren seine Kinder unterhalb des Rechteckes.

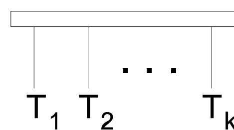


Abbildung 7.6: Ein Q-Knoten mit den Kindern $T_1, T_2, \ldots, T_k$.

Der Unterschied zwischen einem P-Knoten und einem Q-Knoten beruht darauf, wie die Kinder des jeweiligen Knotens behandelt werden. Wir werden dies in Kürze erläutern, doch zunächst sollen noch einige einschränkende Bedingungen an die PQ-Bäume gestellt werden.

Definition 7.3.6
Ein PQ-Baum über eine Menge U heißt **zulässig** *genau dann, wenn die folgenden drei Bedingungen gelten:*

1. *Jedes Element $a_i \in U$ erscheint genau einmal als Blatt in dem PQ-Baum.*
2. *Jeder P-Knoten hat mindestens zwei Kinder.*
3. *Jeder Q-Knoten hat mindestens drei Kinder.*

Die erste Bedingung ist deshalb so wichtig, da mögliche Permutationen auf der Menge U untersucht werden sollen. Daher würde es keinen Sinn machen, wenn Elemente aus U mehr als einmal als Blatt erscheinen und andere dafür gar nicht. Außerdem machen P-Knoten mit nur einem Kind wenig Sinn. Sie ergeben lediglich lange Ketten innerhalb eines Baumes. Dies sollte vermieden werden, da die Untersuchung von Ketten sehr kostspielig ist. Daher darf ein zulässiger PQ-Baum nur P-Knoten mit mindestens zwei Kindern haben. Die dritte Einschränkung basiert, wie wir noch sehen werden, auf den Eigenschaften, die wir den Knoten noch zuschreiben werden. Diese Eigenschaften bedingen, daß sich ein Q-Knoten mit nur zwei Kindern nicht von einem P-Knoten mit zwei Kindern unterscheidet.
Obwohl wir uns auf die Betrachtung zulässiger PQ-Bäume beschränken, konstruieren die in dieser Arbeit vorgestellten Algorithmen zur Reduktion von Teilmengen nicht immer einen zulässigen PQ-Baum. Dies soll uns aber nicht weiter stören, da diese Unzulässigkeiten mit dem Abschluß einer Reduktion immer behoben werden und somit nach ihrer Anwendung wieder ein zulässiger PQ-Baum existiert.
Liest man die Blätter eines PQ-Baumes von links nach rechts, so ergibt sich die **Front** des PQ-Baumes. Diese Front stellt offensichtlich eine Permutation der Elemente der Menge U dar. Beispielsweise besitzt der Baum für die Menge $U = \{A, B, C, D, E, F, G, H, I, J, K\}$ in Abbildung 7.7 als Front gerade die Permutation ABCDEFGHIJK. Analog ergibt sich die Front eines Knotens des Baumes, indem die Blätter unter seinen Nachfahren von links nach rechts gelesen werden. Zwei Blätter in der Front eines PQ-Baumes heißen **adjazent**, wenn sie eine zusammenhängende Folge bilden.

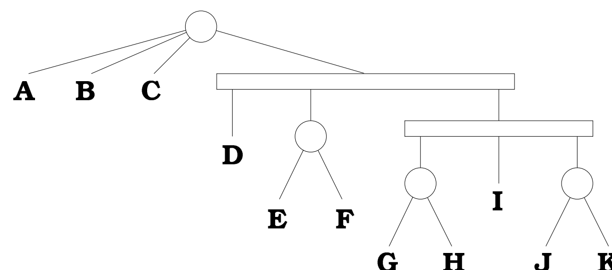


Abbildung 7.7: Ein PQ-Baum über die Menge $U = \{A, B, C, D, E, F, G, H, I, J, K\}$.

Offensichtlich gibt es PQ-Bäume über eine Menge U, deren zugrundeliegenden, verwurzelten Bäume zwar gleich sind, die selber aber nicht zueinander identisch sind. Kennzeichnend ist, daß jeder dieser PQ-Bäume eine andere Front besitzt, jeder Baum also eine

andere Permutation der Elemente von U darstellt. Gelingt es nun, einen Bezug zwischen PQ-Bäumen, deren zugrundeliegenden, verwurzelten Bäume gleich sind, und der Menge der zulässigen Permutationen einer Menge U mit gegebenen Restriktionen herzustellen, so können PQ-Bäume als Repräsentanten von zulässigen Permutationen der Menge U verwendet werden.
Hierbei ist es von Vorteil, daß wir bei der Definition der PQ-Bäume darauf verzichtet haben, die Reihenfolge der Kinder der internen Knoten vorzuschreiben. Stattdessen lassen wir bestimmte Neuordnungen unter den Kindern der internen Knoten zu. Das gibt uns die Möglichkeit, in einem Baum die Ordnung der Kinder der internen Knoten zu verändern, um dadurch eine veränderte Front zu erhalten, die eine weitere Permutation der Elemente der Menge U darstellt. Dies fassen wir formal wie folgt auf:

Definition 7.3.7
Zwei zulässige PQ-Bäume sind **äquivalent** *genau dann, wenn sich einer der beiden Bäume durch* **Äquivalenztransformationen** *in den anderen Baum transformieren läßt. Bei den Äquivalenztransformationen handelt es sich um die folgenden möglichen Neuordnungen der Kinder von internen Knoten:*

1. *Die beliebige* **Permutation** *der Kinder eines P-Knotens.*
2. *Die Umkehrung der Reihenfolge der Kinder eines Q-Knotens, eine sogenannte* **Reversion** *der Kinder.*

Damit haben wir nachträglich den Unterschied zwischen einem P- und einem Q-Knoten geklärt. Üblicherweise schreiben wir verkürzend anstatt über Permutation der Kinder von P-Knoten und Umkehrung der Kinder von Q-Knoten einfach von Permutationen und Reversionen im PQ-Baum. Meist schreiben wir allerdings noch einfacher nur von Permutationen im PQ-Baum und meinen damit eine Folge von Äquivalenztransformationen.
Eine mögliche Permutation der Menge $U = \{A, B, C, D, E, F, G, H, I, J, K\}$, die ohne weiteres mit Hilfe der Äquivalenztransformationen, angewandt auf den in Abbildung 7.7 dargestellten Baum, erreicht werden kann, ist in der Abbildung 7.8 dargestellt.
Die zu einem zulässigen PQ-Baum T äquivalenten PQ-Bäume formen zusammen mit T eine **Äquivalenzklasse**. Jeder Baum aus einer solchen Äquivalenzklasse hat eine andere Front. Da die Blätter in der Front des jeweiligen PQ-Baumes eine Permutation der Menge U darstellen, ergibt die Äquivalenzklasse eines PQ-Baumes eine Menge von Permutationen der Elemente einer Menge U. Wir bezeichnen diese Menge der Permutationen mit $\mathbf{cons(T)}$, wobei T ein beliebiger Repräsentant aus der Äquivalenzklasse ist. Offensichtlich gilt also:

$$cons(T) = \{\text{Front}(T') | T' \equiv T\}$$

Der Baum in der Abbildung 7.8, dessen Front gerade die Permutation BGHIKJFEDCA ergibt, ist demnach ein Element aus der Äquivalenzklasse des Baumes aus der Abbildung 7.7. Diese Äquivalenzklasse enthält nach Booth und Lueker [BL76] insgesamt 768 verschiedene Bäume und repräsentiert somit auch 768 verschiedene Permutationen der Menge $\{A, B, C, D, E, F, G, H, I, J, K\}$.

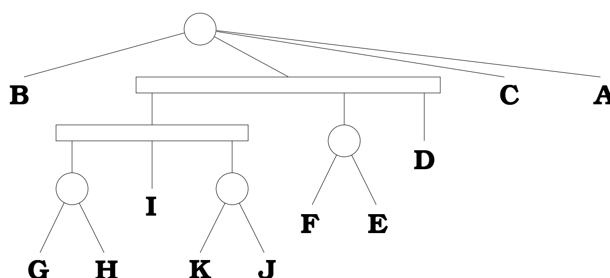


Abbildung 7.8: Ein zum PQ-Baum aus Abbildung 3.3 äquivalenter Baum.

Betrachtet man das Beispiel genauer, so fällt auf, daß Teilmengen $S \in U$ existieren, deren Elemente in allen äquivalenten PQ-Bäumen eine zusammenhängende Folge bilden. Das gilt beispielsweise für die Mengen {E,F} und {H,G,I,J,K}. Auf der anderen Seite gibt es Teilmengen, wie die Menge {C,D,I}, die nie eine zusammenhängende Folge bilden können. Bei der Menge {C,D,I} können zwar C und D eine zusammenhängende Folge bilden, aber die Blätter des Baumes können offensichtlich nie so permutiert werden, daß I zu C oder D benachbart ist.
Den bisherigen Betrachtungen nach zu urteilen, scheinen sich die PQ-Bäume zur Darstellung einer Klasse von Permutationen zu eignen. Damit erhalten wir einen Ansatz zur Lösung unseres ursprünglichen Problems, nämlich bei einer gegebenen Menge von Objekten U sowie k Restriktionen, dargestellt durch Teilmengen $S_i \subset U$, $1 \leq i \leq k$, zulässige Permutationen zu finden beziehungsweise festzustellen, daß keine zulässigen Permutationen existieren. Daß wir dabei tatsächlich sowohl P- als auch Q-Knoten benötigen, und wir nicht nur mit P-Knoten auskommen, kann man sich leicht an einem Beispiel klar machen. Wir müssen in der Lage sein, mit Hilfe der Äquivalenzklasse eines Baumes, alle möglichen Mengen von zulässigen Permutationen einer Menge U darzustellen. Angenommen, eine Menge $U :=$ {A,B,C} besitzt die zulässigen Permutationen {ABC,CBA}, so sind wir nicht in der Lage, diese Menge mit Hilfe eines Baumes darzustellen, der nur aus P-Knoten und Blättern besteht. Somit können wir nicht auf Q-Knoten verzichten. Daß

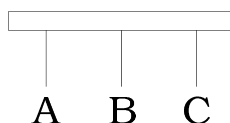


Abbildung 7.9: Der PQ-Baum über der Menge $U = \{A, B, C\}$ mit den zulässigen Permutationen {ABC,CBA}.

wir aber tatsächlich mit den beiden Knotentypen auskommen und damit alle Mengen von möglichen Permutationen auf einer Menge U darstellen können, ist nicht ganz so leicht zu zeigen, wurde aber von Booth und Lueker [BL76] bewiesen.
Um nun bei einer gegebenen Menge U mit den Restriktionen $\{S_1, S_2, \ldots, S_k\}$ festzustellen, ob zulässige Permutationen existieren, starten wir mit einem PQ-Baum, den wir den **universellen Baum** $\mathbf{T(U, U)}$ nennen wollen. Dieser Baum besteht nur aus einem

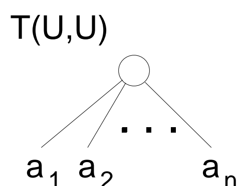


Abbildung 7.10: Der universelle Baum $T(U,U)$ über der Menge $U = \{a_1, a_2, \dots, a_n\}$.

P-Knoten als Wurzel und allen Elemente der Menge U als Kinder dieses Knotens. Der universelle Baum repräsentiert offensichtlich alle Permutationen der Menge U.
Es wird nun so vorgegangen, daß die Mengen S_1 bis S_k der Reihe nach untersucht werden. Wir starten mit der Menge S_1 und untersuchen den PQ-Baum $T_0 := T(U,U)$ daraufhin, ob eine Permutation der Blätter des Baumes existiert, so daß die Elemente aus S_1 eine zusammenhängende Folge innerhalb der Front bilden. Sollte dies der Fall sein, und dies ist offensichtlich bei dem Baum T_0 erfüllt, so wird durch entsprechendes Einfügen von weiteren P- und Q-Knoten in den PQ-Baum die Front des Baumes so gebunden, daß in allen zu dem Baum äquivalenten Bäumen die Elemente aus S_1 immer eine zusammenhängende Folge bilden. Die Menge der Permutationen, die der PQ-Baum darstellt, ist verringert worden. Dadurch verringert sich die Menge der zulässigen Permutationen der Menge U, die Menge S_1 ist reduziert worden.
Für jede weitere Menge S_i, $i \in \{2, \dots, k\}$, wird der PQ-Baum T_{i-1} untersucht und gegebenenfalls reduziert. Dieser Baum T_{i-1} repräsentiert alle zulässigen Permutationen, in denen die Mengen $S_1, S_2, \dots, S_{i-1}$ eine zusammenhängende Folge bilden, und es wird nun versucht, diese zulässigen Permutationen in Bezug auf die Menge S_i weiter zu reduzieren. Sollte die Menge U tatsächlich zulässige Permutationen besitzen, so daß alle Restriktionen erfüllt sind, so finden wir auf diese Art und Weise einen PQ-Baum T_k, der alle zulässigen Permutationen repräsentiert. Sollten allerdings keine zulässigen Permutationen existieren, so werden wir bei dem Versuch scheitern, zulässige Permutationen für eine Menge S_i in einem Baum T_{i-1} festzustellen. Die Menge S_i kann nicht reduziert werden. Um dies in irgendeiner Form kenntlich zu machen, werden wir dann den gesamten Baum T_{i-1} durch einen Nullbaum ersetzen. Ein **Nullbaum** ist ein Baum, der keinerlei Knoten besitzt, nach unserer Definition also gar kein PQ-Baum ist. Wir verwenden ihn trotzdem, da er keine Front besitzt und die Menge der zulässigen Permutationen trivialerweise leer ist. Damit übernimmt er die wichtige Aufgabe, Probleme zu repräsentieren, für die keine zulässigen Permutationen existieren, also nicht lösbar sind.

7.3.1 PQ-Baum Schablonen

Die Beschreibung des vorangegangenen Abschnitts war noch recht informal in Bezug auf die sukzessive Reduktion von Teilmengen $S_1, S_2, \dots, S_k$ einer Menge U in einem PQ-Baum zur Bestimmung der zulässigen Permutationen. Wir wollen uns daher in diesem Abschnitt intensiv mit der Reduktion einer Teilmenge $S \subset U$ auseinandersetzen.
Hat man eine Teilmenge $S \subset U$ und einen zulässigen PQ-Baum T gegeben, so wird ein neuer PQ-Baum T' benötigt, der nur solche Permutationen repräsentiert, in denen die

Blätter, welche die Elemente von S darstellen, eine zusammenhängende Folge einnehmen. Wir nennen T' die Reduktion von T bezüglich S, sprechen aber meist von dem reduzierten Baum T und meinen damit T'.
Eine recht einleuchtende Methode, um den Baum T' zu finden, ist den Baum zu durchmustern. Wir untersuchen jeden einzelnen Knoten des Baumes, wobei darauf geachtet wird, daß ein Knoten erst dann bearbeitet wird, wenn alle seine Kinder bearbeitet worden sind. Auf diese Weise erhält man für jeden Knoten X Informationen darüber, welches seiner Kinder gerade Elemente der Menge S in seiner Front besitzt. Auf der Grundlage dieser Informationen können die Kinder des Knotens X anschließend so gruppiert werden, daß die Kinder mit Elementen aus S in ihrer Front eine zusammenhängende Folge einnehmen. Natürlich kann es passieren, daß eine solche Anordnung der Kinder nicht gelingt und somit eine Reduktion des Baumes bezüglich der Menge S scheitert.
Die Fälle, in denen eine Anordnung der Kinder eines Knotens X existiert, so daß die Elemente aus S, die sich in der Front von X befinden, eine zusammenhängende Folge einnehmen, lassen sich katalogisieren. Es gibt insgesamt dreizehn solcher Fälle. Jedem Fall kann ein zugrundeliegendes **Muster** zugeordnet werden. Dieses Muster entspricht den zulässigen Permutationen der Blätter in der Front des betrachteten Knotens X, bei der die Elemente aus S eine zusammenhängende Folge bilden. Dieses Muster wird im Baum substituiert. Eine solche **Substitution** behält die zulässigen Permutationen des Musters bei. Ferner führt sie zusätzliche Restriktionen in Form von P- und Q-Knoten ein, die verhindern, daß andere Permutationen als die des Musters, zulässig sind. Nach der Substitution können die Elemente aus S in der Front von X nicht mehr eine unzusammenhängende Folge einnehmen.
Muster und Substitution ergeben zusammen eine **Schablone**. Bei der Durchmusterung des Baumes wird nun für jeden Knoten X überprüft, ob eine der Schablonen auf X anwendbar ist. Existiert keine solche Schablone, so ist der Baum nicht reduzierbar. Existiert eine solche Schablone, so wenden wir sie an und substituieren das Muster. Wir sprechen in diesem Zusammenhang auch von einer Reduktion des Knotens X.
Ein Algorithmus zur Reduktion einer Menge $S \subset U$ läßt sich nun grob darstellen. $QUEUE$ sei dabei ein First-in First-out Stapel mit den beiden Operationen *dequeue*, die das erste Element von $QUEUE$ ausgibt, und *enqueue*, welche ein neues Element in $QUEUE$ ablegt. Dieser Stapel garantiert, daß alle Kinder eines Knotens X untersucht werden, bevor X selber untersucht wird:

PQ-Baum **procedure** REDUCE(T,S);
begin
  initialisiere $QUEUE = \emptyset$;
  Lege jedes Blatt $X \in U$ in $QUEUE$ ab;
  while $|QUEUE| > 0$ **do**
    begin
      $X \leftarrow dequeue(QUEUE)$;
      if eine Schablone kann auf X angewendet werden **then**
        substituiere das Muster;
      else
        begin

```
            T := Nullbaum;
            exit from do;
        end; {else}
    if S ⊂ {Y|X ist Vorfahre von Y} then
        exit from do;
    if alle Geschwister von X wurden bereits bearbeitet then
        enqueue(QUEUE) ← (Vater von X);
  end; {while}
 return T;
end {begin}
```

Im Zuge der Anwendung der Schablonen wird gleichzeitig eine Kennung bei jedem bearbeiteten Knoten hinterlassen. Diese Kennung gibt Auskunft über das Verhältnis der Elemente aus S zu den Elementen aus $U \setminus S$ in der Front des Knotens und erleichtert die Anwendung der Schablonen. Daher führen wir zunächst noch folgende Bezeichnungen ein:

Definition 7.3.8
Es sei T ein zulässiger PQ-Baum über der Menge U und S eine Teilmenge von U. Ein Knoten heißt **voll** *genau dann, wenn er nur Elemente aus der Menge S in seiner Front besitzt und* **leer** *genau dann, wenn er nur Elemente aus $U \setminus S$ in seiner Front besitzt. Andernfalls heißt der Knoten* **partiell**. *Die* **Kennung** *eines Knotens ist die Bezeichnung mit voll, leer oder partiell.*
Ein Knoten heißt **relevant** *genau dann, wenn er entweder voll oder partiell ist. Als* **relevanter Unterbaum** *von T wird der kleinste, verwurzelte Unterbaum von T bezeichnet, der alle Elemente von S enthält.*

Offensichtlich ist der relevante Unterbaum aufgrund seiner Minimalitätsbedingung eindeutig. Damit ist auch die Wurzel des relevanten Unterbaumes eindeutig.
Zu Beginn der Prozedur REDUCE ist die Kennung aller internen Knoten unbekannt. Sie erhalten ihre Kennung erst im Zuge der Anwendung einer Schablone. Lediglich für die Blätter ist die Kennung bekannt. Sie sind entweder voll oder leer, je nachdem ob sie ein Element der Menge S sind oder nicht. Entsprechend gibt es für Blätter auch nur zwei triviale Schablonen, in denen die Blätter voll oder leer markiert werden.

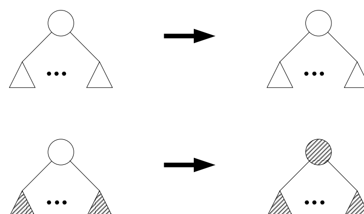


Abbildung 7.11: Die Schablonen $P0$ (oben) für einen P-Knoten mit leeren Kindern und $P1$ (unten) für einen P-Knoten mit vollen Kindern.

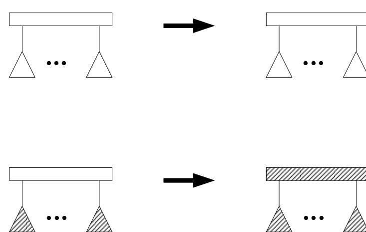


Abbildung 7.12: Die Schablonen $Q0$ (oben) für einen Q-Knoten mit leeren Kindern und $Q1$ (unten) für einen Q-Knoten mit vollen Kindern.

Für P-Knoten und Q-Knoten mit nur vollen Kindern oder nur leeren Kindern sind die Schablonen ebenfalls einfach. Hat ein Knoten nur leere Kinder, wie in den Mustern der Schablonen $P0$ für P-Knoten und $Q0$ für Q-Knoten, so muß der Knoten selbst auch leer sein. Entsprechend braucht in der Substitution des Musters dieser Knoten nur als leer markiert werden. Analog muß ein Knoten mit nur vollen Kindern, wie in dem Muster der Schablonen $P1$ für P-Knoten und $Q1$ für Q-Knoten, lediglich als voll markiert werden. Die Schablonen $P0$ und $P1$ für P-Knoten beziehungsweise die Schablonen $Q0$ und $Q1$ für Q-Knoten sind in der Abbildung 7.11 beziehungsweise 7.12 dargestellt. Die leeren Knoten sind weiß gezeichnet, während die vollen Knoten schraffiert sind. Für die zugrundeliegenden Muster ist es nur entscheidend, ob alle Kinder leer oder voll sind. Daher ist es unerheblich, ob die Kinder P-Knoten, Q-Knoten oder Blätter sind. Entsprechend sind die Kinder als Dreiecke gezeichnet, um ihre Unabhängigkeit von der Wahl ihres Typs deutlich werden zu lassen.

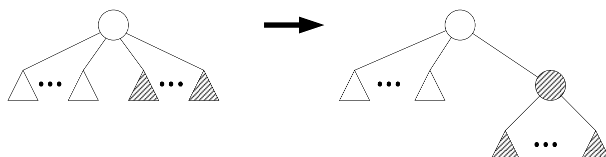


Abbildung 7.13: Die Schablone $P2$ für einen P-Knoten als Wurzel des relevanten Unterbaumes mit vollen und leeren Kindern.

Schwieriger wird es, wenn die zu bearbeitenden Knoten, Kinder mit unterschiedlichen Kennungen besitzen. Betrachten wir zunächst einen P-Knoten X mit vollen und leeren Kindern. Ist der P-Knoten gleichzeitig die Wurzel des relevanten Unterbaumes, so wenden wir die Schablone $P2$ an, dargestellt in Abbildung 7.13. Diese Schablone kann auf die Wurzel eines relevanten Unterbaumes angewendet werden, wenn Äquivalenztransformationen des PQ-Baumes existieren, so daß die Kinder der Wurzel des relevanten Unterbaumes eine Anordnung einnehmen, die dem Muster der Schablone entspricht.
Die Frage ist, was für eine Substitution angewendet werden soll. Die Kinder des P-Knotens X können frei permutiert werden. Dies muß eingeschränkt werden, da nach der Reduktion alle vollen Blätter in der Front des P-Knotens eine zusammenhängende Folge einnehmen sollen. Andererseits sieht die freie Permutation aller Kinder auch Permutationen von leeren und vollen Kindern vor, in denen die vollen Kinder bereits eine zusammenhängende

Folge bilden. Diese Permutationen dürfen durch eine Reduktion nicht unterbunden werden. Daher wird die Folge der vollen Kinder durch einen vollen P-Knoten ersetzt, der nur die vollen Kinder als direkte Nachfahren besitzt.
Mit der Einführung des neuen vollen P-Knotens ist der betrachtete P-Knoten X nicht länger die Wurzel des relevanten Unterbaumes. Der neu eingeführte Knoten ist an dessen Stelle gerückt. Da es mit der Abarbeitung des relevanten Unterbaumes keinen Sinn hat, die noch nicht gemusterten Knoten des Baumes zu bearbeiten, bricht der Algorithmus REDUCE an dieser Stelle ab. Dadurch bleibt X ohne eine Kennung.
Ist der zu bearbeitende P-Knoten X allerdings nicht die Wurzel des relevanten Unterbaumes, so existiert, neben den vollen Blättern in der Front des Knotens X, noch mindestens ein weiteres relevantes Blatt im Baum. Eine Substitution des Musters muß nun sicher stellen, daß nach ihrer Anwendung auf das Muster alle relevanten Blätter eine zusammenhängende Folge einnehmen können und daß die Permutationen verboten werden, die leere Blätter zwischen den vollen Blättern in der Front von X und den restlichen vollen Blättern im Baum zulassen. Daher können wir die Schablone $P2$, trotz ihrer Gruppierung der vollen Knoten, nicht verwenden. Sie läßt Permutationen zu, bei denen die vollen Knoten von leeren Knoten eingeschlossen werden, und sich somit leere Blätter zwischen der Folge der vollen Blätter in der Front des Knotens X und dem zusätzlichen vollen Blatt befinden. Die Substitution des Musters muß also nicht nur die vollen Knoten gruppieren, sondern gleichzeitig die vollen und die leeren Knoten voneinander trennen.

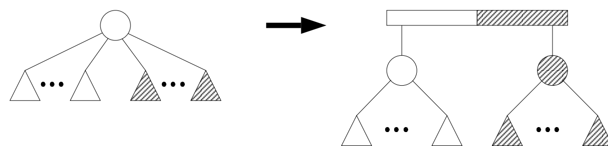


Abbildung 7.14: Die Schablone $P3$ für einen P-Knoten, der nicht die Wurzel des relevanten Unterbaumes ist, mit vollen und leeren Kindern.

Die Schablone $P3$ in der Abbildung 7.14 liefert das gewünschte Resultat. Die leeren und die vollen Knoten werden jeweils unter einem neuen P-Knoten mit der entsprechenden Kennung gesammelt, während der P-Knoten X zu einem Q-Knoten wird. Leere wie volle Knoten können weiterhin frei untereinander permutieren, sind aber fein säuberlich voneinander getrennt.

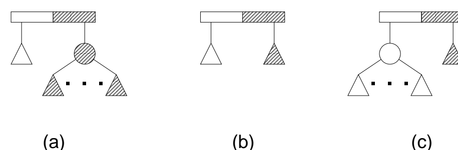


Abbildung 7.15: Alternative Substitutionen zur Schablone $P3$.

Da der relevante Unterbaum noch nicht abgearbeitet ist, muß der Knoten X nach der Substitution des Musters eine Kennung erhalten. Da er sowohl leere als auch volle Blätter in seiner Front besitzt, gilt er nach Definition 7.3.8 als partiell. Wir bezeichnen ihn als

einfach partiell, da die vollen Knoten nur auf einer Seite des Knotens erscheinen und schraffieren in der Darstellung der Schablone eine Seite.
Nach der Reduktion von X besitzt X als Q-Knoten nur noch zwei Kinder. Folglich ist der so entstandene PQ-Baum nach Definition 7.3.6 nicht mehr zulässig. Das soll uns aber nicht weiter stören, da sich noch mindestens ein weiteres volles Blatt im Baum befindet und daher ein Q-Knoten mit mindestens drei Kindern konstruiert wird. Dies gilt natürlich nur, falls die Menge der relevanten Blätter reduzierbar ist. Sollte das nicht der Fall sein, ist dies ebenfalls ohne Bedeutung, da in einem späteren Schritt der Algorithmus REDUCE erkennt, daß der PQ-Baum nicht reduzierbar ist und ihn durch den Nullbaum ersetzt.
Für den Fall, daß der Knoten X nur ein volles Kind oder nur ein leeres Kind besitzt, wird dieses Kind nicht unter einem P-Knoten gesammelt, sondern bleibt direkter Nachfahre des Knotens X. Die entsprechenden Substitutionen sind in der Abbildung 7.15 dargestellt. Damit vermeiden wir P-Knoten mit nur einem Kind, was eine Unzulässigkeit im Sinne der Definition 7.3.6 darstellen würde.

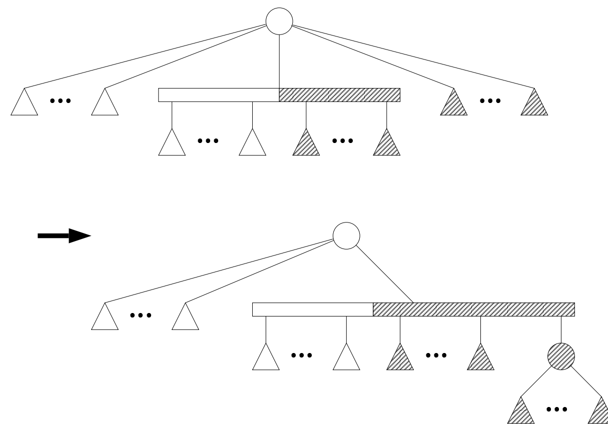


Abbildung 7.16: Die Schablone $P4$ für einen P-Knoten als Wurzel des relevanten Unterbaumes mit einem partiellen Kind.

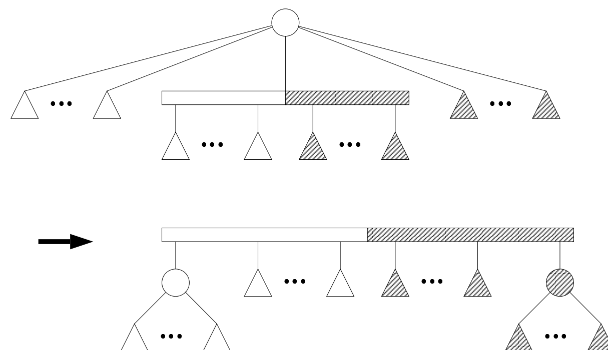


Abbildung 7.17: Die Schablone $P5$ für einen P-Knoten, der nicht Wurzel des relevanten Unterbaumes ist, mit einem partiellen Kind.

Mit der Einführung von partiellen Knoten besteht nun auch die Möglichkeit, daß der zu untersuchende P-Knoten X mehrere partielle Kinder besitzt. Wir betrachten zunächst

einmal den Fall, daß X genau ein partielles Kind besitzt. Hier muß ebenfalls unterschieden werden, ob X die Wurzel des relevanten Unterbaumes ist oder nicht. Die Abbildung 7.16 zeigt die Schablone $P4$ für die Wurzel des relevanten Unterbaumes und Abbildung 7.17 zeigt die Schablone $P5$ für den anderen Fall. Es handelt sich bei den beiden Schablonen $P4$ und $P5$ im wesentlichen um Erweiterungen der Schablonen $P2$ und $P3$, die zusätzlich einen partiellen Knoten in der Front des Knotens X verarbeiten müssen.
Wenn der Knoten X weniger als zwei volle oder leere Kinder hat, so werden diese Kinder nicht erst unter einem P-Knoten gesammelt. Die Substitutionen dieser Kinder können analog zu denen aus Abbildung 7.15 gebildet werden. Es ist sogar möglich, daß X gar keine vollen oder leeren Kinder hat. Die Substitutionen sind dann entsprechend zu modifizieren. Es ist allerdings nicht zulässig, daß X weder volle noch leere Kinder besitzt. Denn in diesem Fall hätte X nur ein Kind, was ein Widerspruch zur Zulässigkeit des PQ-Baumes T wäre.

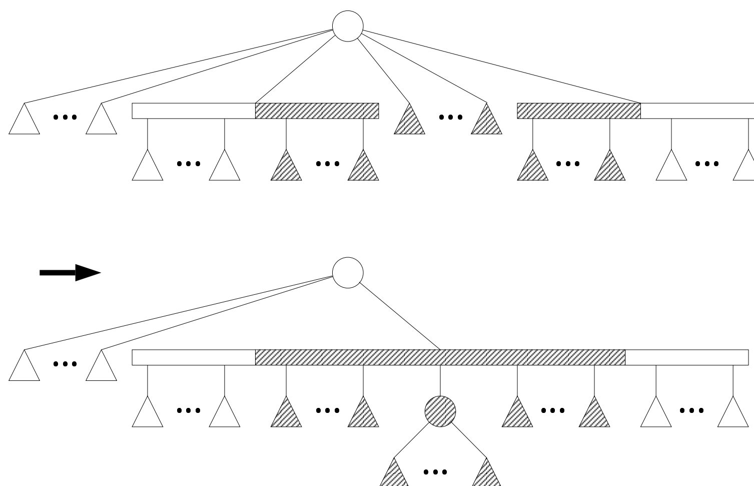


Abbildung 7.18: Die Schablone $P6$ für einen P-Knoten als Wurzel des relevanten Unterbaumes mit zwei partiellen Kindern.

Besitzt der P-Knoten X zwei partielle Kinder, so muß X bereits die Wurzel des relevanten Unterbaumes sein. Jedes der beiden partiellen Kinder besitzt nämlich ein volles und ein leeres Ende. Um nun eine zusammenhängende Folge der vollen Blätter in der Front von X zu erhalten, müssen alle vollen Kinder von X zwischen den beiden partiellen Kindern plaziert werden, während die partiellen Kinder ihre vollen Seiten in Richtung ihrer vollen Geschwister orientieren. Offensichtlich wird dadurch die Folge der vollen Blätter in der Front von X in allen zulässigen Permutationen durch die leeren Blätter in der Front der leeren Seiten der beiden partiellen Knoten eingeschlossen. Befindet sich daher noch ein weiteres relevantes Blatt in dem PQ-Baum, das nicht Nachfahre von X ist, kann es niemals eine benachbarte Position zur relevanten Folge in der Front des Knotens X einnehmen. Der Baum ist somit nicht reduzierbar.
Die entsprechende Schablone $P6$ ist in Abbildung 7.18 dargestellt. Den Q-Knoten in der Substitution, unter dem alle vollen Kinder und die Kinder der beiden partiellen Knoten gesammelt werden, nennt man **doppelt partiell**. Falls es, wie in den vorangegangenen Schablonen, vorkommt, daß X nur ein oder gar kein leeres oder volles Kind besitzt, so

wird analog zu diesen Schablonen verfahren. In der Schablone $P6$ ist es dabei durchaus zulässig, daß X weder leere noch volle Kinder besitzt, da X bereits zwei partielle Kinder hat und somit zulässig im Sinne der Definition 7.3.6 ist. In diesem Fall wird in der Substitution der P-Knoten durch den doppelt partiellen Q-Knoten ersetzt.
Alle anderen P-Knoten, auf die keines der Muster aus den sieben Schablonen für P-Knoten anwendbar ist, bezeichnen wir als **illegal**. Sie sind ein Indikator für die Nicht-reduzierbarkeit eines Baumes. Der Algorithmus REDUCE bricht dementsprechend ab, sobald er einen solchen Knoten entdeckt.
Für Q-Knoten gibt es neben den eingangs bereits erwähnten Schablonen $Q0$ und $Q1$ nur zwei weitere Schablonen. Dies liegt zum einen daran, daß bei den Substitutionen der Muster nicht zwischen der Wurzel des relevanten Unterbaumes und einem beliebigen anderen internen Knoten unterschieden werden muß, zum anderen richten sich die Muster in erster Linie danach, ob die relevanten Kinder am Rand oder in der Mitte des Q-Knotens erscheinen.

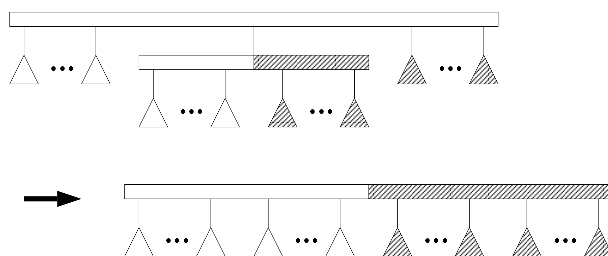


Abbildung 7.19: Die Schablone $Q2$ für einen Q-Knoten mit maximal einem partiellen Kind.

Ein Q-Knoten X ist einfach partiell, wenn er höchstens ein partielles Kind besitzt, das mit den vollen Kindern von X eine zusammenhängende Folge einnimmt, wobei diese Folge sich an einem Ende des Q-Knotens befindet. Die Schablone $Q2$, die in diesem Fall angewendet wird, ist in Abbildung 7.19 dargestellt.
Es ist leicht ersichtlich, daß es keinen Unterschied macht, ob der Q-Knoten X die Wurzel des relevanten Unterbaumes ist oder nicht. Eine Reduktion muß nur sicherstellen, daß der partielle Knoten keine Reversion seiner Kinder vornehmen kann und somit leere Blätter zwischen volle Blätter permutiert. Entsprechend sichert die Substitution des Musters durch die Auflösung des partiellen Q-Knotens die Fixierung der Reihenfolge der Kinder des partiellen Knotens innerhalb der Front des Knotens X.
Das Muster der Schablone $Q2$ läßt zu, daß alle der leeren beziehungsweise vollen Kinder oder das partielle Kind fehlen. Allerdings dürfen nicht alle Kinder die gleiche Kennung besitzen, so daß die Schablonen $Q0$ oder $Q1$ angewendet werden können.
Ein Q-Knoten ist doppelt partiell, wenn er höchstens zwei partielle Kinder besitzt und eine Permutation seiner Nachfahren existiert, so daß die vollen Blätter in der Front der beiden partiellen Kinder mit den Blättern in der Front der vollen Kindern eine zusammenhängende Folge bilden, so daß sich auf beiden Seiten dieser Folge von vollen Blättern nur leere Blätter in der Front von X befinden. Die vollen Blätter werden also durch leere Blätter eingeschlossen. Die entsprechende Schablone ist in der Abbildung 7.20 dargestellt.

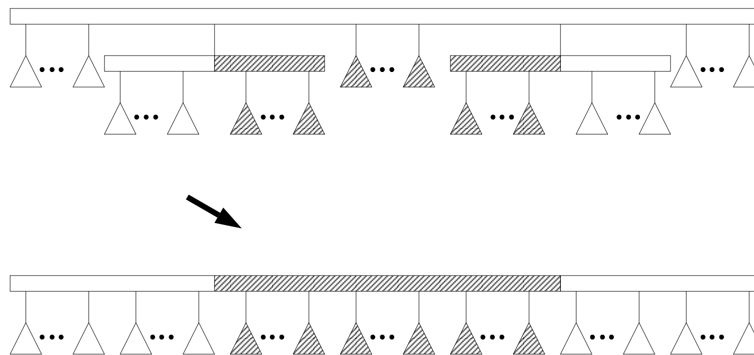


Abbildung 7.20: Die Schablone $Q3$ für einen Q-Knoten mit maximal zwei partiellen Kindern.

Wie im Fall der Schablone $Q2$ können auch hier leere, volle oder partielle Kinder fehlen, solange dies nicht einem der Muster aus den Schablonen $Q0$, $Q1$ oder $Q2$ entspricht.
Der auf diese Weise reduzierte Q-Knoten X muß die Wurzel des relevanten Unterbaumes sein, falls der Baum reduzierbar ist. Ansonsten folgt mit dem gleichen Argument wie zur Schablone $P6$ die Nichtreduzierbarkeit des Baumes. Alle Q-Knoten, die nicht einem Muster der vier Schablonen für Q-Knoten entsprechen, gelten als illegal und verursachen den Abbruch des Algorithmus REDUCE.
Ziel des Algorithmus REDUCE ist es, einen gegebenen PQ-Baum T in bezug auf die Menge S zu reduzieren, das heißt die in dem Baum zulässigen Permutationen müssen soweit eingeschränkt werden, bis nur noch solche Permutationen zulässig sind, bei denen die Elemente der Menge S eine zusammenhängende Folge bilden. Ein Baum, bei dem es offensichtlich ist, daß die Elemente der Menge S bei allen zulässigen Permutationen immer eine zusammenhängende Folge bilden, ist der Baum $\mathbf{T}(\mathbf{U}, \mathbf{S})$, dargestellt in Abbildung 7.21. Er besitzt neben den Elementen aus U als Blätter nur zwei P-Knoten, wobei einer der beiden P-Knoten genau nur die Elemente aus S als Kinder hat. Der andere P-Knoten ist die Wurzel des Baumes und besitzt die Elemente aus $U \setminus S$ und den ersten P-Knoten als Kinder. Die Menge S stellt in diesem Baum die einzige Restriktion dar. Es gibt offensichtlich sonst keine weitere Menge, deren Elemente in allen zulässigen Permutationen eine zusammenhängende Folge bilden können. Dies macht den Baum für uns interessant, da es genau der Baum ist, bei dem alle Permutationen zulässig sind, die ausschließlich die Restriktion der Menge S beachten. Alle Bäume, die dies ebenfalls erfüllen, sind lediglich Elemente aus der Äquivalenzklasse von $T(U, S)$.
Mit Hilfe des Baumes $T(U, S)$ sind wir in der Lage, das folgende Theorem zu formulieren, welches uns bestätigt, daß PQ-Bäume das von uns Gewünschte auch tatsächlich leisten:

Theorem 7.3.9

Es sei T ein beliebiger, zulässiger PQ-Baum über einer Menge U, und es sei S eine Teilmenge von U. Es sei T_S der durch REDUCE(T, S) erzeugte PQ-Baum. Dann gilt:

$$cons(T_S) = cons(T) \cap cons(T(U, S))$$

Beweis:

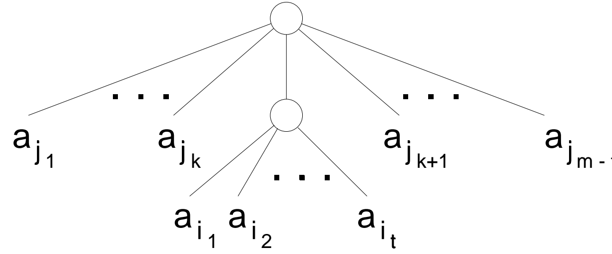


Abbildung 7.21: Der Baum $T(U, S)$ mit $S = \{a_{i_1}, a_{i_2}, \dots, a_{i_t}\}$ als einzige Restriktion über U.

$\subseteq$ Sei $\pi \in cons(T_S) \neq \emptyset$. Dann existiert $\tilde{T_S}$ mit $\tilde{T_S} \equiv T_S$ und $Front(\tilde{T_S}) = \pi$.

Konstruiere aus $\tilde{T_S}$ einen Baum $\tilde{T}$ mit $\tilde{T} \equiv T$ und $Front(\tilde{T}) = \pi$ durch Auflösen der Schablonen in umgekehrter Reihenfolge wie sie durch REDUCE(T, S) angewendet wurden ohne die Äquivalenztransformationen zurückzunehmen $\Rightarrow \pi \in cons(T)$

Ferner: Sei X der kleinste gemeinsame Vorfahre von S in T_S. Dann ist X entweder voll oder ein Q-Knoten, in dem die vollen Kinder eine zusammenhängenden Folge bilden. $\Rightarrow \pi \in cons(T(U, S))$

$\supseteq$ Sei $\pi \in cons(T) \cap cons(T, (U, S)) \Rightarrow \exists\, T'$ mit $T' \equiv T$ und $Front(T') = \pi$ insbesondere sind in $Front(T')$ Elemente aus S aufeinanderfolgend.

$\Rightarrow$ Kein Knoten aus T' mit Ausnahme des kleisten gemeinsamen Vorfahren T von S hat mehr als ein partielles Kind. Der kleinste gemeinsame Vorfahre hat höchstens zwei partielle Kinder und ferner: Volle Kinder bilden bei jedem Knoten eine zusammenhängende Folge.

$\Rightarrow \pi \in cons(REDUCE(T', S))$. Da $T \equiv T'$ und die Schablonen äquivalenzerhalten sind folgt:

$$\pi \in cons(REDUCE(T, S))$$

$\square$

Wir erinnern daran, daß $cons(T)$ die Menge der Permutationen über einer Menge U ist, die durch die Fronten der zu T äquivalenten PQ-Bäume dargestellt wird.

PQ-Bäume können also tatsächlich dazu verwendet werden, alle Permutationen zu repräsentieren, in denen die Elemente einer Teilmenge innerhalb einer Familie von Teilmengen als zusammenhängende Folge erscheinen. Geht man demnach von einem universellen PQ-Baum über einer Menge U aus, so wird durch die Reduktion einer Teilmenge S die Menge der Permutationen soweit eingeschränkt, daß die Elemente von S eine zusammenhängende Folge bilden. Wiederholte Reduktionen, bezogen auf eine Familie von Teilmengen, erzeugen genau die Klasse von Permutationen, in denen die Elemente der einzelnen Teilmengen eine zusammenhängende Folge bilden.

7.4 Effiziente Implementierung der PQ-Bäume

Die Implementierung des Algorithmus hat zwei Probleme zu lösen: Sie muß erkennen, welche Schablonen zu verwenden sind und sie muß diese anschließend anwenden. Dies erfordert eine Bearbeitung aller Kinder eines Knotens X, bevor X selbst bearbeitet wird. Daher sind wir bisher davon ausgegangen, daß der ganze PQ-Baum T während der Reduktion einer Teilmenge S untersucht wird. Dies ist allerdings sehr kostspielig, besonders dann, wenn die Menge S sehr klein ist. Daher beschränkt man sich bei der Reduktion von S auf den relevanten Unterbaum der Menge S. Aber auch der relevante Unterbaum besitzt leere Knoten und es ist in diesem Fall ebenfalls sehr zeitaufwendig und von keinem Nutzen, wenn die leeren Knoten in dem relevanten Unterbaum bearbeitet werden. Wir beschränken uns deshalb bei der Untersuchung auf die relevanten Knoten des relevanten Unterbaumes und bezeichnen den Unterbaum, der genau die relevanten Knoten enthält als den **gekürzt relevanten Unterbaum in bezug auf S**. Dieser Unterbaum ist im allgemeinen kein legaler PQ-Baum.
Um die Schablonen anwenden zu können, dürfen die leeren Kinder eines Knotens nicht einfach außer Betracht gelassen werden. Da sie aber nicht bearbeitet werden, wird eine 2-Phasen Implementierung verwendet, die es ermöglicht, die leeren Knoten durch ihre *Abwesenheit* im gekürzten relevanten Unterbaum zu registrieren. Die erste Phase, die sogenannte Bubble-Up Phase, geht den Baum, ausgehend von den relevanten Blättern, nach oben und markiert alle Knoten, die bearbeitet werden müssen, als relevant. Gleichzeitig wird für jeden relevanten Knoten die Anzahl der zu bearbeitenden Kinder gezählt. Dadurch kann in der zweiten Phase, in der die eigentliche Reduktion stattfindet, festgestellt werden, ob alle relevanten Kinder eines Knotens reduziert worden sind.
Die Bottom-Up Strategie der ersten Phase, die es ermöglicht, auf die Untersuchung des gesamten Baumes T zu verzichten, erfordert für jeden relevanten Knoten einen Zeiger auf seinen Vater. Allerdings kann die Verwaltung all dieser Zeiger ebensoviel Arbeit verursachen, wie die Untersuchung des gesamten Baumes. Besitzt nämlich ein interner Knoten eine große Anzahl von leeren Kindern und wird dieser Knoten bei der Anwendung einer Schablone entfernt, so muß jedes seiner Kinder, also auch die leeren Kinder, einen neuen Zeiger auf seinen neuen Vater erhalten. Auf diese Weise kann es passieren, daß fast jeder Knoten im Baum einen neuen Zeiger erhalten muß, obwohl die Menge S nur eine vergleichsweise geringe Anzahl von Elementen enthält. Wir hätten somit gegenüber der Untersuchung des gesamten Baumes nichts gewonnen, da fast alle Knoten angefaßt werden, um einen neuen Zeiger zu erhalten.
Um dieses Problem in den Griff zu bekommen, wird für bestimmte Knoten darauf verzichtet, daß diese einen Zeiger auf ihren Vater besitzen. Alle Kinder eines P-Knotens sowie die beiden Kinder am Rand eines jeden Q-Knotens erhalten einen Zeiger. Alle anderen Kinder eines Q-Knotens *leihen* sich bei Bedarf den Zeiger auf ihren Vater von einem der beiden Kinder am Rand. Während der Bubble-Up Phase werden nur relevante Knoten untersucht. Daher können relevante Kinder eines Q-Knotens nur dann einen Zeiger auf ihren Vater erhalten, wenn die relevanten Kinder eine zusammenhängende Folge derart bilden, daß eines dieser Kinder gerade eines der beiden am Rand befindlichen Kinder ist. Sieht man einmal von dem Fall ab, daß die Wurzel des relevanten Unterbaumes ein Q-Knoten ist, der an beiden Enden leere Kinder besitzt, verlagert sich dadurch die Frage

nach der Reduzierbarkeit einer Menge S teilweise auf die Frage, ob für alle Knoten ein Zeiger auf den Vater gefunden wird. Existieren dann relevante Knoten, die nach Abschluß der Untersuchung des gekürzt relevanten Unterbaumes keinen Zeiger auf ihren Vater besitzen, so befinden sich an beiden Enden ihres Vaters leere Kinder und es existiert keine Permutation, in der die relevanten Blätter in der Front dieser relevanten Knoten eine zusammenhängende Folge mit den anderen relevanten Blättern einnehmen können. Der Baum T ist somit nicht reduzierbar bezüglich der Menge S und der Algorithmus wird abgebrochen. Es reicht allerdings im allgemeinen nicht aus, die Vaterzeiger der relevanten Knoten zu kennen, um die Reduzierbarkeit der Menge S zu garantieren. Dies wird letztlich erst bei der Anwendung der Schablonen festgestellt.
Booth und Lueker gelang nun mit Hilfe dieser grundlegenden Ideen eine Implementierung, die in $O(m + n + \sum_{i=1}^{n} |S_i|)$ Zeit eine Klasse von zulässigen Permutationen berechnet. Bemerkenswert an dieser Laufzeit ist vor allem ihre Linearität, bezogen auf die Größe des Inputs. Dies macht den Algorithmus zu einem effizienten Werkzeug für die Lösung von Problemen, in denen zulässige Permutationen gefunden werden müssen, die gewisse Restriktionen beachten.
Eine Implementierung muß zwei Aktionen durchführen können:

1. Entscheiden, welche Schablone angewendet werden muß
2. Durchführung der Schablone

Problem: Die naive Implementierung benötigt quadratische Laufzeit.

Lösung:

1. Durchlaufe nicht den ganzen Baum, sondern nur den **relevaten Unterbaum**, d.h. nur die vollen und partiellen Knoten.
2. Führe Updates der Vaterzeiger nur dann durch, wenn diese benötigt werden.

Grundsätzlich gilt:

1. Kinder von P-Knoten haben immer einen zulässigen Vaterzeiger.
2. Kinder von Q-Knoten haben i.a. keinen zulässigen Vaterzeiger mit Ausnahme der beiden Kinder an den äußeren Enden.

Zur Durchführung der Reduktion wird für jeden Knoten des relevanten Unterbaumes ein Vaterzeiger benötigt.

1. Phase: BUBBLE

stellt sicher, daß alle relevanten Knoten einen zulässigen Vaterzeiger besitzen bevor REDUCE aufgerufen wird:

a. Zählt Sequenzen von **blockierten** Knoten (Knoten ohne zulässigen Vaterzeiger)

b. Falls eine solche Sequenz inzident zu einem relevanten Knoten mit zulässigem Vaterzeiger ist, setze die Vaterzeiger der Sequenz.

c. Falls es mehr als eine blockierte Sequenz gibt, so ist T nicht reduzierbar bzgl. S (bei einer kann eventuell immer noch $Q3$ angewendet werden).

Reduction

The function `Reduction` tests whether permissible permutations of the elements of U exist such that the elements of a subset $S \subset U$, stored in `ArrayOfElements`, form a consecutive sequence. If there exists such a permutation, the PQ-tree is reduced and `Reduction` returns `TRUE`.
The function `Reduction` expects the number of elements `numberOfElements` in the set S and an array `ArrayOfElements` of pointers to elements of type `leafKey`, representing all elements of S.
`Reduction` calls the procedure `Bubble` and if `Bubble` was successful, `Reduction` calls the function `Reduce`.

⟨*Reduction*⟩≡

```
/**********************************************************************
                         Reduction
**********************************************************************/

template<class T,class X,class Y>
bool PQTree<T,X,Y>::Reduction(int numberOfConsecutiveElements,
                              leafKey<T,X,Y>** ArrayOfElements)
{

   if (Bubble(numberOfConsecutiveElements,ArrayOfElements))
      return Reduce(numberOfConsecutiveElements,ArrayOfElements);
   else
      return false;
}
```

Bubble

The function `Bubble` realizes a function described in Booth and Lueker 1976. It *bubbles* up from the pertinent leaves to the pertinent root in order to make sure that every pertinent node in the pertinent subtree has a valid pointer to its parent. If `Bubble` does not succed in doing so, then the set of elements, stored in the `ArrayOfElements` cannot form a consecutive sequence. The function `Bubble` uses a wide variaty of variables, explained in detail below.

`_blockcount` is the number of blocks of blocked nodes during the bubbling up phase.

`_numBlocked` is the number of blocked nodes during the bubbling up phase.

`_blockedSiblings` counts the number of blocked siblings that are adjacent to `_checkNode`. A node has 0, 1 or 2 blocked siblings. A child of a P-node has no blocked siblings. Endmost children of Q-nodes have at most 1 blocked sibling. The interior children of a Q-Node have at most 2 blocked siblings.

`_checkLeaf` is a pointer used for finding the pertinent leaves.

`_checkNode` is a pointer to the actual node.

`_checkSib` is a pointer used to examin the siblings of `_checkNode`.

`_offTheTop` is a variable which is either 0 (that is its initial value) or 1 in case that the root of the tree has been process during the first phase.

`_parent` is a pointer to the parent of `_checkNode`, if `_checkNode` has a valid parent pointer.

`_processNodes` is a first-in first-out list that is used for sequencing the order in which the nodes are processed.

`_blockedNodes` is a stack storing all nodes that have been once blocked. In case that the `_pseudoRoot` has to be introduced, the stack contains the blocked nodes.

⟨*Bubble*⟩≡

```
/***********************************************************************
                           Bubble
***********************************************************************/

template<class T,class X,class Y>
bool PQTree<T,X,Y>::Bubble(int number,
                          leafKey<T,X,Y>** ArrayOfElements)
{
   pqQueue<pqNode<T,X,Y>*> *_processNodes = new pqQueue<pqNode<T,X,Y>*>;
   pqStack<pqNode<T,X,Y>*> *_blockedNodes = new pqStack<pqNode<T,X,Y>*>;


   int         _blockCount       = 0;
   int         _numBlocked       = 0;
   int         _offTheTop        = 0;

   int         _blockedSiblings  = 0;
   int         i                 = 0;
   pqNode<T,X,Y>*   _checkLeaf   = NULL;
   pqNode<T,X,Y>*   _checkNode   = NULL;
   pqNode<T,X,Y>*   _checkSib    = NULL;
   pqNode<T,X,Y>*   _holdSib     = NULL;
   pqNode<T,X,Y>*   _oldSib      = NULL;
   pqNode<T,X,Y>*   _parent      = NULL;
   pqNode<T,X,Y>*   _lastBlocked = NULL;

   ⟨Bubble: Enter the Full leaves into the queue _processNodes⟩
```

```
    while ((_processNodes->queueSize() + _blockCount + _offTheTop) > 1)
    {
       if (_processNodes->queueSize() == 0)
          ⟨Bubble: no consecutive sequence possible⟩

       ⟨Bubble: get next _checkNode from queue⟩
       ⟨Bubble: check if node is adjacent to an unblocked node⟩

       if (_checkNode->mark() == UNBLOCKED)
       {
          _parent = _checkNode->_parent;
          ⟨Bubble: Get maximal consecutive set of blocked siblings⟩
          ⟨Bubble: Process parent of _checkNode⟩
       }
       else
       {
          ⟨Bubble: Process blocked _checkNode⟩
       }
    }

    ⟨Bubble: If _blockCount = 1 enter _pseudoRoot to the tree⟩


    delete _processNodes;
    delete _blockedNodes;

    return true;
 }
```

This chunk belongs to the function `Bubble`. In a first step the pertinent leaves have to be identified in the tree and entered on to the queue `_processNodes`. The identification of the leaves can be done with the help of a pointer stored in every `leafKey` in constant time for every element.

⟨*Bubble: Enter the* `Full` *leaves into the queue* `_processNodes`⟩≡

```
 for (i = 0; i <= (number-1); i++)
 {
    _checkLeaf = ArrayOfElements[i]->nodePointer();
    _checkLeaf->mark(QUEUED);
    _processNodes->enQueue(_checkLeaf);
    _pertinentNodes->push(_checkLeaf);
 }
```

This chunk belongs to the function `Bubble` and handles the case that the queue `_processNodes` does not contain any nodes for processing and the sum of `_blockCount` and `_offTheTop` is

greater than 1. If the queue is empty, the root of the pertinent subtree was already processed. Nevertheless, there are blocked nodes since `_offTheTop` is either be `0` or `1`, hence `_blockCount` must be at least `1`. Such blocked nodes cannot form a consecutive sequence with all nodes in the set `ArrayOfElements`.
Observe that this chunk finishes the function `Bubble`. Hence every memory allocated by the function `Bubble` has to be deleted here as well.

⟨*Bubble: no consecutive sequence possible*⟩≡

```
{
   delete _processNodes;
   delete _blockedNodes;
   return false;
}
```

This chunk belongs to the function `Bubble`. If there are still nodes to be processed in which case the queue `_processNodes` is not empty, we get the next node from the queue. By default this node has to be marked as blocked.

⟨*Bubble: get next* `_checkNode` *from queue*⟩≡

```
_checkNode = _processNodes->deQueue();
_blockedNodes->push(_checkNode);
_checkNode->mark(BLOCKED);
_blockedSiblings = 0;
```

This chunk belongs to the function `Bubble`. After getting the node `_checkNode` from the queue, its siblings are checked, whether they are unblocked. If they are, then they have a valid pointer to their parent and the parent pointer of `_checkNode` is updated.

⟨*Bubble: check if node is adjacent to an unblocked node*⟩≡

```
if ((_checkNode->_parentType != P_NODE) && (_checkNode != _root))
                                          // _checkNode is son of a Q_NODE.
                                          // Check if it is blocked.
{
   if (_checkNode->getSib(LEFT) == NULL)
                                          // _checkNode is endmost child of
                                          // a Q_NODE. It has a valid pointer
                                          // to its parent.
   {
      _checkNode->mark(UNBLOCKED);
      if (_checkNode->getSib(RIGHT) && _checkNode->getSib(RIGHT)->mark() == BLOCKED)
         _blockedSiblings++;
   }

   else if (_checkNode->getSib(RIGHT) == NULL)
                                          // _checkNode is endmost child of
                                          // a Q_NODE. It has a valid pointer
                                          // to its parent.
   {
```

```
            _checkNode->mark(UNBLOCKED);
            if (_checkNode->getSib(LEFT) && _checkNode->getSib(LEFT)->mark() == BLOCKED)
               _blockedSiblings++;
         }


         else
                                                 // _checkNode is not endmost child of
                                                 // a Q_NODE. It has not a valid pointer
                                                 // to its parent.
         {
            if (_checkNode->getSib(LEFT)->mark() == UNBLOCKED)
                                                 // _checkNode is adjacent to an
                                                 // unblocked node. Take its parent.
            {
               _checkNode->mark(UNBLOCKED);
               _checkNode->_parent = _checkNode->getSib(LEFT)->_parent;
            }
            else if (_checkNode->getSib(LEFT)->mark() == BLOCKED)
               _blockedSiblings++;

            if (_checkNode->getSib(RIGHT)->mark() == UNBLOCKED)
                                                 // _checkNode is adjacent to an
                                                 // unblocked node. Take its parent.
            {
               _checkNode->mark(UNBLOCKED);
               _checkNode->_parent = _checkNode->getSib(RIGHT)->_parent;
            }
            else if (_checkNode->getSib(RIGHT)->mark() == BLOCKED)
                _blockedSiblings++;
         }
      }

      else                                       // _checkNode is son of a P_NODE
                                                 // and children of P_NODEs
                                                 // cannot be blocked.
            _checkNode->mark(UNBLOCKED);
```

This chunk belongs to the procedure `bubble`. The node `_checkNode` is `UNBLOCKED`. If the parent of `_checkNode` is a Q-Node, then we check the siblings `_checkSib` of `_checkNode` whether they are `BLOCKED`. If they are blocked, they have to be marked `UNBLOCKED` since they are adjacent to the `UNBLOCKED` node `_checkNode`. We then have to proceed with the siblings of `_checkSib` in order to find `BLOCKED` nodes adjacent to `_checkSib`. This is repeated until no `BLOCKED` nodes are found any more.
Observe that while running through the children of the Q-Node which is referred by the pointer `_parent`, their parent pointers, as well as the `_pertChildCount` of `_parent` are updated. Furthermore we reduce simultaneously the count `_numBlocked`.

⟨*Bubble: Get maximal consecutive set of blocked siblings*⟩≡

```
if (_blockedSiblings > 0)
{
   if (_checkNode->getSib(LEFT) != NULL)
   {
      _checkSib = _checkNode->getSib(LEFT);
      _oldSib = _checkNode;
      _holdSib = NULL;
      while (_checkSib->mark() == BLOCKED)
      {
         _checkSib->mark(UNBLOCKED);
         _checkSib->_parent = _parent;
         _numBlocked--;
         _parent->_pertChildCount++;
         _holdSib = clientNextSib(_checkSib,_oldSib);
         _oldSib = _checkSib;
         _checkSib = _holdSib;
         if (_checkSib == NULL)
         {
            cout << ``ERROR: PQTree->Bubble: Blocked node as ''
                 << `` endmost child of a Q_NODE.'' << endl;
            _checkSib = _oldSib;
         }
      }
   }



   if (_checkNode->getSib(RIGHT) != NULL)
   {
      _checkSib = _checkNode->getSib(RIGHT);
      _oldSib = _checkNode;
      _holdSib = NULL;
      while (_checkSib->mark() == BLOCKED)
      {
         _checkSib->mark(UNBLOCKED);
         _checkSib->_parent = _parent;
         _numBlocked--;
         _parent->_pertChildCount++;
         _holdSib = clientNextSib(_checkSib,_oldSib);
         _oldSib = _checkSib;
         _checkSib = _holdSib;
         if (_checkSib == NULL)
         {
            cout << ``ERROR: PQTree->Bubble: Blocked node as ''
                 << `` endmost child of a Q_NODE.'' << endl;
            _checkSib = _oldSib;
         }
```

```
        }
    }

}
```

This chunk belongs to the procedure `bubble`. After processing the siblings of the `UNBLOCKED` `_checkNode` the parent has to be processed. If `_checkNode` is the root of the tree we do nothing except setting the flag `_offTheTop`. If it is not the root and `_parent` has not been placed onto the queue `_processNodes`, the `_parent` is placed on to `_processNodes`.
Observe that the number `_blockCount` is updated. Since `_checkNode` was `UNBLOCKED` all pertinent nodes adjacent to that node became `UNBLOCKED` as well. Therefore the number of blocks is reduced by the number of `BLOCKED` siblings of `_checkNode`.

⟨*Bubble: Process parent of* `_checkNode`⟩≡

```
if (_parent == NULL)
                                        // _checkNode is root of the tree.
    _offTheTop = 1;

else
                                        // _checkNode is not the root.
{
    _parent->_pertChildCount++;
    if (_parent->mark() == UNMARKED)
    {
        _processNodes->enQueue(_parent);
        _pertinentNodes->push(_parent);
        _parent->mark(QUEUED);
    }

}

_blockCount = _blockCount - _blockedSiblings;
_blockedSiblings = 0;
```

This chunk belongs to the procedure `bubble`. Since `_checkNode` is `BLOCKED`, we cannot continue processing at this point in the Tree. We have to wait until this node becomes unblocked. So only the variables `_blockCount` and `_numBlocked` are updated.

⟨*Bubble: Process blocked* `_checkNode`⟩≡

```
_blockCount = _blockCount + 1 - _blockedSiblings;
_numBlocked++;
_lastBlocked = _checkNode;
```

This chunk belongs to the procedure `bubble`. In case that the root of the pertinent subtree is a Q-node with empty children on both sides and the pertinent children in the middle, it is possible that the PQ-tree is reducible. But since the sequence of pertinent children of the

Q-node is blocked, the procedure is not able to find the parent of its pertinent children. This is due to the fact that the interior children of a Q-node do not have a valid parent pointer.
So the root of the pertinent subtree is not known, hence cannot be entered into the processing queue used in the function call `Reduce`. To solve this problem a special node only designed for this cases is used: `_pseudoRoot`. It simulates the root of the pertinent subtree. This works out well, since for this node the only possible template maching is `template_Q3`, where no pointers to the endmost children of a Q-node are used.

⟨*Bubble: If* `_blockCount` $= 1$ *enter* `_pseudoRoot` *to the tree*⟩≡

```
if (_blockCount == 1)
{
   while (!_blockedNodes->stackEmpty())
   {
      _checkNode = _blockedNodes->pop();
      if (_checkNode->mark() == BLOCKED)
      {
         _checkNode->mark(UNBLOCKED);
         _checkNode->_parent = _pseudoRoot;
         _pseudoRoot->_pertChildCount++;
         if (_checkNode->endmostChild())
         {
            cout << ``ERROR: PQTree->Bubble: Blocked node as ''
                 << `` endmost child of a Q_NODE.'' << endl;
            _checkSib = _oldSib;
         }
      }
   }
}
```

2. Phase: Reduktion

`Reduce`

The function `Reduce` does the reduction of the pertinent leaves with the help of the template matchings, designed by Booth and Lueker. The reader should observe that this function can only be called after every pertinent node in the pertinent subtree has gotten a valid parent pointer. If this is not the case, the programm will be interrupted by run-time errors such as seqmentation faults. The pertinent nodes can get valid parent pointers by using the function `Bubble` (7.4). If the function `Bubble` returns `true`, then it was succesful in giving each pertinent node in the pertinent subtree a valid parent pointer. If the function returns `false`, then some nodes do not have a valid parent pointer and the pertinent leaves are not reducable.
The function `Reduce` starts with the pertinent leaves and stores them in a queue `_processNodes`. Every time a node is processed, its parent is checked whether all its pertinent children are already processed. If this is the case, the parent is allowed to be processed as well and stored in the queue.
Processing a node means that the function `Reduce` tries to apply one of the template matchings. In case that one template matching was successful, the node was reduced and `Reduce` tries to

reduce the next node. In case that no template matching was successfully applied, the tree is is irreducible. This causes the reduction process to be halted returning `false`.
The folllowing variables are used by the function `Reduce`.

`_checkLeaf` is a pointer to a various leaf of the set of elements that has to be reduced.

`_checkNode` is a pointer to a various node of the pertinent subtree.

`_pertLeafCount` counts the number of pertinent leaves in the *PQ*-tree. Since `Reduce` takes care that every node knows the number of pertinent leaves in its frontier, the root of the pertinent subtree can be identified with the help of `_pertLeafCount`.

`_processNodes` is a queue storing nodes of the pertinent subtree that are considered to be reduced next. A node may be reduced (and therefore is pushed on to `_processNodes`) as soon as all its pertinent children have been reduced.

⟨*Reduce*⟩≡

```
/*********************************************************************
                            Reduce
*********************************************************************/

template<class T,class X,class Y>
bool PQTree<T,X,Y>::Reduce(int number,
                          leafKey<T,X,Y> **ArrayOfElements)
{

   int                     i             = 0;
   pqNode<T,X,Y>*          _checkLeaf    = NULL;
   pqNode<T,X,Y>*          _checkNode    = NULL;
   int                     _pertLeafCount = 0;
   pqQueue<pqNode<T,X,Y>*>*  _processNodes  = new pqQueue<pqNode<T,X,Y>*>;



   ⟨Reduce: Enter the Full leaves into the queue _processNodes⟩


   _checkNode = _processNodes->front();
   while ((_checkNode != NULL) && (_processNodes->queueSize() > 0))
   {
      _checkNode = _processNodes->deQueue();


      if (_checkNode->_pertLeafCount < _pertLeafCount)
      {
         ⟨Reduce: Apply template to _checkNode ≠ root of pertinent subtree⟩
      }
      else
      {
```

```
        ⟨Reduce: Apply template to _checkNode = root of pertinent subtree⟩
      }
    }

    _pertinentRoot = _checkNode;

    delete _processNodes;

    if (_pertinentRoot == NULL)
      return false;
    else
      return true;
}
```

This chunk belongs to the function `Reduce`. In a first step the pertinent leaves have to be identified in the tree and are entered on to the queue `_processNodes`. The identification of the leaves can be done with the help of a pointer stored in every `leafKey` in constant time for every element.

⟨*Reduce: Enter the* `Full` *leaves into the queue* `_processNodes`⟩≡

```
for (i = 0; i <= (number-1); i++)
{
    _checkLeaf = ArrayOfElements[i]->nodePointer();
    _checkLeaf->status(FULL);
    _checkLeaf->_pertLeafCount = 1;
    _processNodes->enQueue(_checkLeaf);
    _pertLeafCount++;
}
```

This chunk belongs to the function `Reduce` and describes the application of the template matchings to a pointer `_checkNode` storing the adress of a node that is **not the root** of the pertinent subtree.
Before applying the template matchings to `_checkNode`, some values of the parent of `_checkNode` are updated. The number of the parents pertinent children stored in `_pertChildCount` is count down by one. In case that `_checkNode->_parent->_pertChildCount == 0`, we know that all pertinent children of the parent have been processed. Since the parent then is allowed to be processed as well, `_checkNode->_parent` is stored in the queue `_processNodes`.

⟨*Reduce: Apply template to* `_checkNode` $\neq$ *root of pertinent subtree*⟩≡

```
_checkNode->_parent->_pertLeafCount = _checkNode->_parent->_pertLeafCount
                                    + _checkNode->_pertLeafCount;
_checkNode->_parent->_pertChildCount--;
if (!_checkNode->_parent->_pertChildCount)
    _processNodes->enQueue(_checkNode->_parent);
if (!template_L1(_checkNode,false))
if (!template_P1(_checkNode,false))
```

```
if (!template_P3(_checkNode))
if (!template_P5(_checkNode))
if (!template_Q1(_checkNode,false))
if (!template_Q2(_checkNode,false))
   _checkNode= NULL;
```

This chunk belongs to the function `Reduce` and describes the application of the template matchings to a pointer `_checkNode` that stores the adress of a node that **is the root** of the pertinent subtree. In a case that a template matching was successfully applied, the pointer `_checkNode` stores after the application the adress of the root of pertinent subtree. This includes nodes that have been newly introduced as root of the pertinent subtree during the application. If no template matching was successfully applied `_checkNode` is a `NULL` pointer.

⟨*Reduce: Apply template to* `_checkNode` = *root of pertinent subtree*⟩≡

```
if (!template_L1(_checkNode,true))
if (!template_P1(_checkNode,true))
if (!template_P2(&_checkNode))
if (!template_P4(&_checkNode))
if (!template_P6(&_checkNode))
if (!template_Q1(_checkNode,true))
if (!template_Q2(_checkNode,true))
if (!template_Q3(_checkNode))
_checkNode = NULL;
```

Lemma 7.4.1
Sei k die Anzahl der relevanten Knoten des relevanten Unterbaumes in T bzgl. S. Dann benötigt BUBBLE $O(k)$ Operationen.

Lemma 7.4.2
Sei k die Anzahl der relevanten Knoten des relevanten Unterbaumes in T bzgl. S. Dann benötigt REDUCE $O(k)$ Operationen.

Ein Knoten im relevanten Unterbaum heißt **unär**, falls er nur ein relevantes Kind hat. Sei $\Omega(T,S)$ die Menge aller unären Knoten in T bzgl. S.

Lemma 7.4.3
Eine Reduktion von T bzgl. S benötigt höchstens $O(|S| + |\Omega(T,S)|)$ Zeit.

Beweis: $|S|$ Blätter. Binäres Branching impliziert höchstens $O(|S|)$ nichtunäre Knoten. Nach Lemma 7.4.1 und 7.4.2 folgt die Behauptung.

Theorem 7.4.4
Die Datenstruktur PQ-Bäume und der Template Matching Algorithmus können so implementiert werden, daß die Klasse der Permutationen, in der die Elemente jeder Menge S_i einer Familie $S = \{s_1, s_2, \ldots, s_n\}$ von Teilmengen von U eine zusammenhängende Folge bilden, in $O(|U| + n + \sum_{i=1}^{n} |S_i|)$ bestimmt werden können.

Beweis: Außerhalb von BUBBLE und REDUCE $O(|U|+n)$ Zeit. Nach Lemma 7.4.3: Zeit benötigt über alle BUBBLE und REDUCE Aufrufe:

$$O(\sum_{i=1}^{n}|S_i| + \sum_{i=1}^{n}|\Omega(T_i, S_i)|)$$

wobei T_i = Reduktion von T_{i-1} bzgl. S_i mit $i = 1, \ldots, n-1$ und $T_0 = T$.
Offensichtlich:

(i) unärer Knoten kann nicht Wurzel des relevanten Unterbaumes sein.

(ii) unärer Knoten ist partiell

$\Rightarrow$ nur P_3, P_5 und Q_2 anwendbar.

Zu P_3: Falls P_3 mehr als zwei mal angewendet wird bzgl. S_i, $i \in \{1, \ldots, n\}$

$\Rightarrow$ mindestens drei partielle Knoten paarweise keine Vorfahren voneinander.

$\Rightarrow$ Reduktion schlägt fehl

$\Rightarrow$ P_3 wird in $O(|U|+n)$ angewendet

Zu Q_2 und P_5:

1. Fall: kein partielles Kind: analog zu P_3

2. Fall: ein partielles Kind

Sei $Q(T) = |\{Q\text{-Knoten in } T\}| + |\{\text{Knoten in } T \text{ mit } P\text{-Knoten als Vater}\}|$.

Offensichtlich

(i) Zu Beginn: $Q(T) \leq |U|$

(ii) Alle Templates erhöhen $Q(T)$ um höchstens 1

(iii) P_5 und Q_2 (2.Fall) vermindern $Q(T)$ um mindestens 1.

Alle Templates ohne P_5 und Q_2 (2. Fall) werden in $O(|U| + n + \sum_{i=1}^{n}|S_i|)$ angewendet.
$\overset{(ii)}{\Rightarrow}$ $Q(T)$ wird um höchstens $O(|U| + n + \sum_{i=1}^{n}|S_i|)$ erhöht
$\overset{(i)-(iii)}{\Rightarrow}$ P_5 und Q_2 (2. Fall) werden in $O(|U| + n + \sum_{i=1}^{n}|S_i|)$ angewendet

□

7.5 Planaritätstest mit PQ-Bäumen

Idee: Ersetze die Buschform B_i durch PQ-Baum T_i wie folgt durch

(i) ein Blatt für jede virtuelle Kante in B_i

(ii) ein P-Knoten für jeden Schnittknoten in B_i

(iii) ein Q-Knoten für jede 2-Zusammenhangskomponente in B_i

Außerdem:

(iv) Wurzel in T_i sei Knoten der S enthält.

```
bool Planaritätstest (G = (V, E)
{
   if (|E| > 3|V| − 6) return 0;
   U := {e = (s, v) ∈ E}
   T := universeller Baum über U
   for (i = 2; i ≤ n − 1; i + +)
   {
      S_i := {e = (v_i, w) ∈ E; ω(v_i) > ω(w)}
      T := Bubble (T, S_i);
      T := Reduce (T, S_i);
      if T = ∅ return 0;
      S' := {e ∈ (v_i, w) ∈ E | ω(v_i) < ω(w)}
      T_S' := universeller Baum über S';
      if (Wurzel relativer Unterbaum in T ist Q-Knoten)
         ersetze volle Kinder und Nachfahren durch T_S'
      else
         ersetze Wurzel durch T_S'
      U := U − S_i ∪ S'
   }
   return 1;
}
```

Theorem 7.5.1 *Sei $G = (V, E)$ ein 2-zusammenhängender Graph. Dann kann G in $O(|V|)$ Zeit auf Planarität getestet werden.*

Beweis: Korrektheit klar.

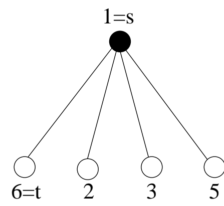
Außerhalb von REDUCE und BUBBLE: $O(|V| + |E|)$ Zeit. Ersetzen der vollen Knoten durch $T_{S'}$ erhöht $O(T)$ um höchstens $|S'|$. Summe über alle $|S'| \leq |E|$

$\Rightarrow$ REDUCE und BUBBLE benötigen über alle Iterationen $O(|V| + |E|)$ Zeit.

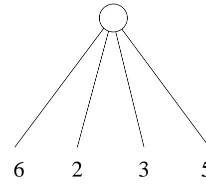
$\Rightarrow$ Eulersche Formel ergibt $O(|V|)$

Korollar 7.5.2 *Ein beliebiger Graph $G = (V, E)$ ohne Multikanten kann in $O(|V|)$ Zeit auf Planarität getestet werden.*

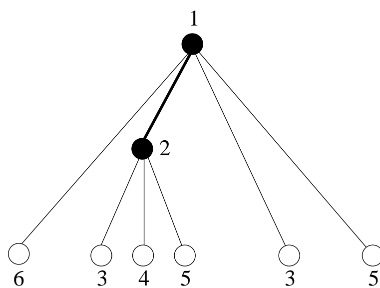
In den folgenden Abbildungen sind $(a) - (i)$ in Buschform und $(a') - (i')$ sind PQ-Bäume.



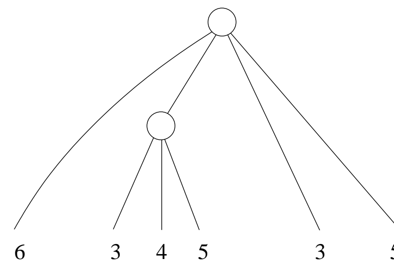
(a) B_1



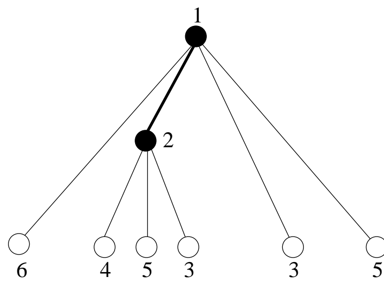
(a')



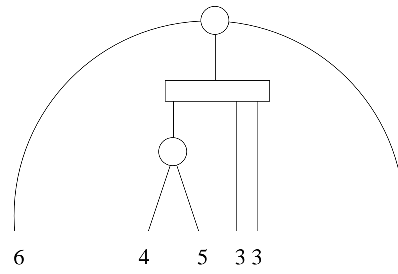
(b) B_2



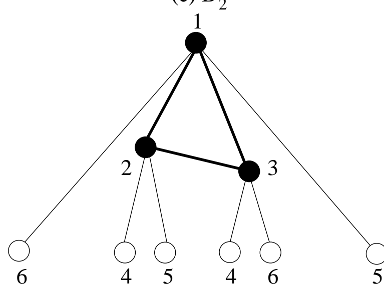
(b')



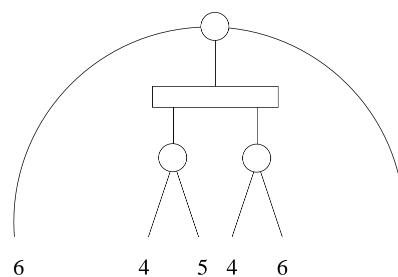
(c) B'_2



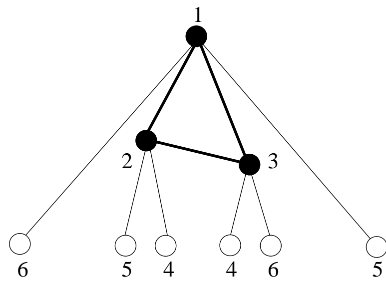
(c')



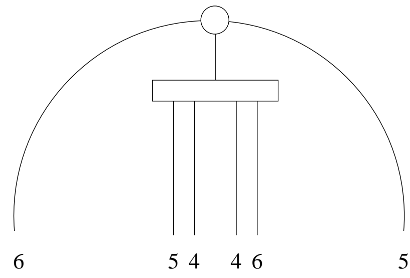
(d) B_3



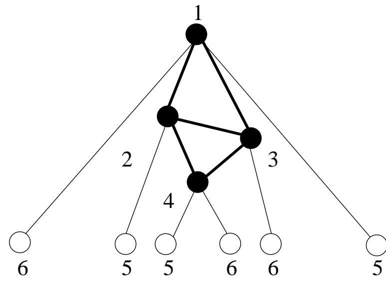
(d')



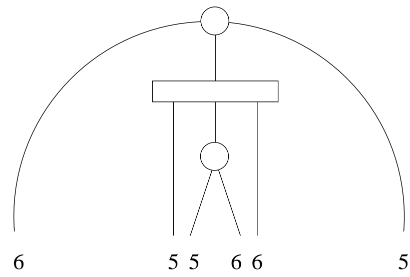
(e) B'_3



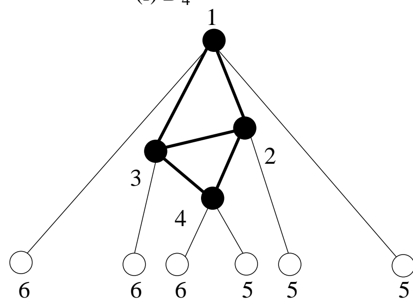
(e')



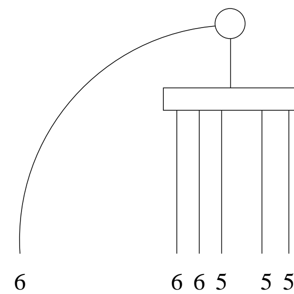
(f) B_4



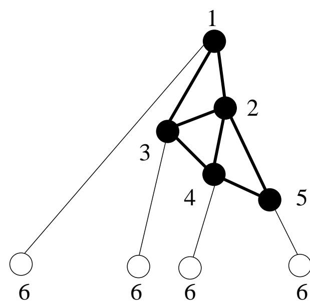
(f')



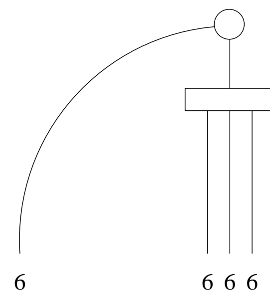
(g) B_4



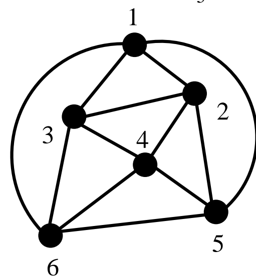
(g')



(h) B_5



(h')



(i) $G = G_6 = B_6$

7.6 Einbettungsalgorithmus

Ansatz: Modifiziere Planaritätstest: Mit jeder Reduktion der PQ-Bäume aktualisiere die Adjazenzlisten der Buschform. $\Rightarrow O(n^2)$

Idee: Speichere nur die Adjazenzliste der reduzierten virtuellen Kanten für jeden Knoten, sowie die impliziten Änderungen in nachfolgenden Reduktionen.

st-Numerierung impliziert eine Richtung im Graphen durch Richten der Kanten von kleinerer zu größerer Knotennummer $\Rightarrow$ azyklischer Digraph. Interpretiere st-numerierten $G = (V, E)$ nun immer als azyklischen Digraphen $D = (V, A)$.

Definition 7.6.1
Eine **aufwärts (gerichtete) planare Zeichnung** *eines azyklischen Digraphen* $D = (V, A)$ *ist eine planare Zeichnung bei der alle Kanten monoton aufsteigend gezeichnet werden.*

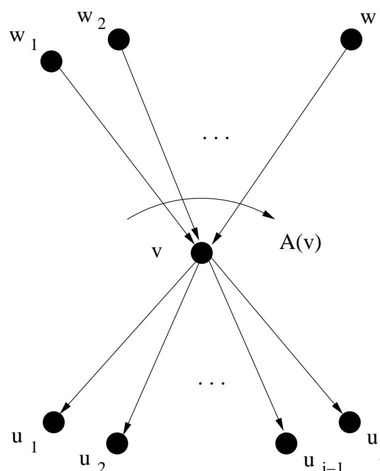
Erzeuge zunächst eine aufwärtsplanare Einbettung von G und konstruiere eine planare Einbettung.

```
void Embed(G)
{
   UpwardEmbed(G);
   EntireEmbed(G);
}
```

Lemma 7.6.2
In einer planaren Einbettung Adj eines Graphen G, die mit dem naiven Einbettungsalgorithmus erzeugt wurde, erscheinen für einen beliebigen Knoten v die Nachbarn mit kleinerer Nummer zusammenhängend um v (und in Adj).

Beweis: klar.

Aufwärtsplanare Einbettung enthält nur Einträge $A(v) = w_1, w_2, \ldots, w_i$

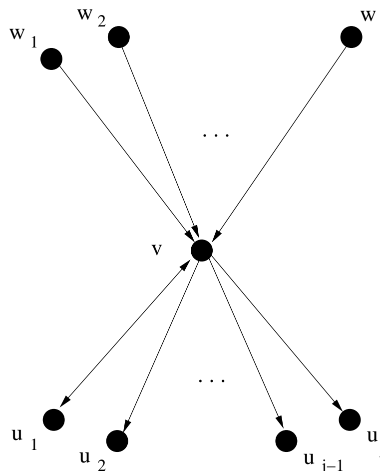


```
void EntireEmbed(G)
{
   kopiere Listen A nach Adj;
   markiere alle Knoten als "neu"
   DFS(t);
}
void DFS(v)
{
   markiere v "alt";
   forall w ∈ A(v)
   {
      kopiere v an den Anfang von Adj(w);
      if (w ist "neu")
         DFS(w)
   }
}
```

Lemma 7.6.3
Sei D der aufwärtsplanare Digraph bzgl. G. Enthalte A die aufwärtsplanare Einbettung von D. EntireEmbed erweitert A in eine planare Einbettung Adj in $O(n)$ Zeit.

Beweis: Laufzeit klar wegen der Verwendung von Depth First Search.

Korrektheit: Wegen st-Numerierung existiert ein gerichteter Pfad von $v \in V \backslash \{t\}$ nach $t \Rightarrow$ jeder Knoten und jede Kante werden abgearbeitet. $\Rightarrow$ $Adj(v)$ enthält Nachbarn in richtiger Reihenfolge. Betrachte die Nachbarschaft von v.



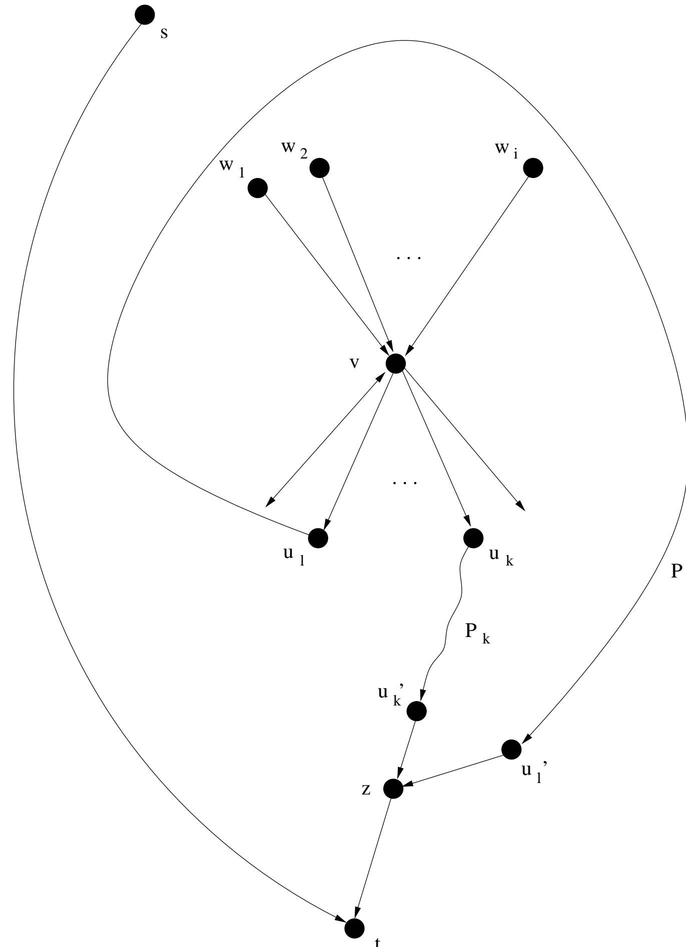
$w_1, w_2, \ldots, w_i$ sind bereits in richtiger Reihenfolge. Zeige, daß die Kanten (v, u_1), (v, u_2), $\ldots$, (v, u_j) in dieser Reihenfolge besucht werden und

$$Adj(v) = u_j, u_{j-1}, \ldots, u_2, u_1, w_1, w_2, \ldots, w_i$$

ist.

Annahme: $k, l \in \{1, \ldots, j\}$ mit $k > l$ aber (v, u_k) wird vor (v, u_l) besucht. Sei P_k Pfad von u_k zu t in DFS-Baum, sei P_l Pfad von u_l zu t in DFS-Baum, sei z Knoten an dem P_k und P_l sich treffen und $(u'_k, z) \in P_k$ und $(u'_l, z) \in P_l$.

Wegen Annahme: erscheint (u'_k, z) **vor** (u'_l, z) in $A(z)$.



Die Subpfade $P'_k = z, u'_k, \ldots, u_k$ und $P'_l = z, u'_l, \ldots, u_l$ haben keinen gemeinsamen Knoten außer z.

$\Rightarrow P'_l, P'_k, (v, u_l), (v, u_k)$ formen einen Kreis C. Wegen Lemma 7.6.2 liegen alle Nachbarn von v aus $A(v)$ innerhalb von C. s ist außerhalb von C.

$$\forall\, x \in C,\ \omega(x) \geq \omega(v)$$
$$\forall\, y \in A(v),\ \omega(y) < \omega(v)$$

Nach Definition st-Numerierung existiert $\forall\, y \in A(v)$ ein gerichteter Pfad P von s nach y mit $\omega(\tilde{y}) < \omega(v)\ \forall\, \tilde{y} \in P$. Dies ist ein Widerspruch zur Voraussetzung.

□

Um in UpwardEmbed $\forall\, v \in V\ A(v)$ zu bestimmen:

1. Nach Reduktion der virtuellen Kanten (= eingehende Kanten) von v, durchlaufe die relevanten Blätter des PQ-Baumes. Dies ergibt (vorläufige) $A(v) = w_1, \ldots, w_i$

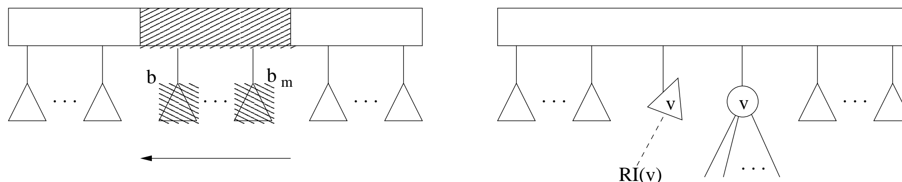
2. (a) Falls Wurzel des relevanten Unterbaumes voll ist
   - reversible Komponenten
   - wir können annehmen, daß $A(v)$ bereits in Ordnung ist

   (b) Falls Wurzel partiell

   $A(v)$ nur vorläufig. Wir müssen zählen wie oft $A(v)$ (implizit) durch spätere PQ-Baumoperationen (= Reversionen von Q-Knoten) umgedreht wird.
   - Falls gerade Zahl: $A(v)$ in Ordnung
   - Falls ungerade Zahl: drehe $A(v)$ um

Einfaches Zählen der Reversionen: $O(n^2)$ Zeit.

Lösung: Füge **Direction Indikator** ein.

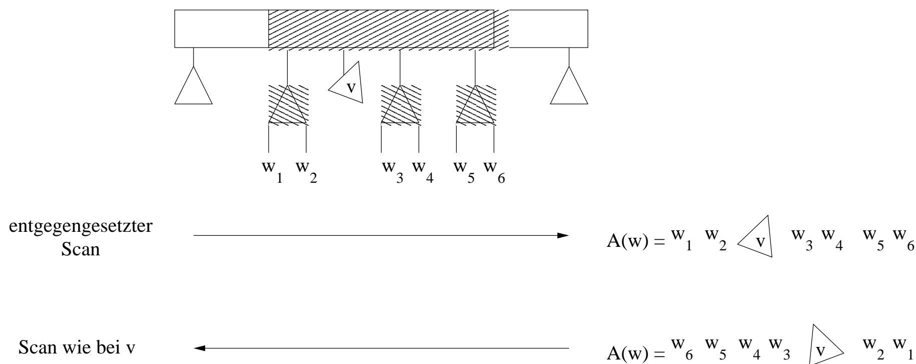


Der Richtungsindikator $RI(v)$ ist ein Blatt, welches in allen Template Operationen ignoriert wird. Dieser hat zwei Aufgaben:

1. Registriert Reversionen.
2. Gibt Durchlaufungsrichtung von v relativ zu den Geschwistern in T an.

Wenn $RI(v)$ innerhalb einer Folge von vollen Blättern des Knotens w ist, entferne $RI(v)$ ebenso wie relevanten Unterbaum und speichere $RI(v)$ mit $A(w)$ ab.

Beispiel:



Wir erhalten so für jeden Knoten $v \in V$ eine (vorläufige) $A(v)$, die $RI(v)$ enthält.

Korrekturschritt:

for $(i = n;\ i \geq 1;\ i--)$
{
  for jedes $x \in A(v)$
    if (x ist RI)
    {
      entferne x aus $A(v)$;
      sei w der korrespondierende Knoten zu x
      if x entgegengesetzt zu $A(v)$
        drehe $A(v)$ um;
    {
{

Lemma 7.6.4
UpwardEmbed bestimmt eine aufwärtsplanare Einbettung A eines planaren Graphen G in $O(n)$ Zeit.

Beweis: Laufzeit klar.

Korrektheit: Zeige $\forall\ v \in V$, daß $A(v)$ ist ohne Einschränkung im Uhrzeigersinn. Sei $A'(v)$ die Liste der eingehenden Kanten, die beim Knotenadditionsschritt bestimmt wird.
1. Fall: $RI(v)$ wird für v nicht hinzugefügt. Setze $A(v) := A'(v)$.

2. Fall: $RI(v)$ wird für v hinzugefügt. $RI(v)$ wird im Knotenadditionsschritt von w, $\omega(w) > \omega(v)$, entfernt.

(i) Falls $RI(v)$ entgegengesetzt zu $A'(w)$: Zahl der Reversionen des Q-Knotens bis zur Reduktion von w ist ungerade

(ii) andernfalls: gerade

Wird für w kein $RI(v)$ eingefügt, setze $A(w) := A'(w)$. Falls (i), setze $A(v) :=$ Reversion $(A'(v))$. Wird für w ein $RI(v)$ eingefügt, so zählt dieser die Zahl der Reversionen des Q-Knotens bzgl. w und damit implizit bzgl. v. Iterierte Anwendung ergibt das Gewünschte.

□

Lemma 7.6.5 *Der Algorithmus Embed bestimmt eine planare Einbettung eines planaren Graphen G in $O(n)$ Zeit.*

Beweis: Folgt aus Lemmata 7.6.3 und 7.6.4.

Kapitel 8

Zeichenverfahren für planare Graphen

8.1 Geradliniges Zeichnen

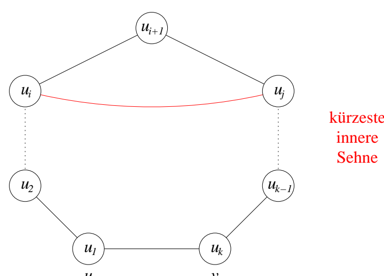
Wir wissen aus Kapitel 3, daß wir planare Graphen immer geradlinig zeichnen können (Algorithmus von Tutte), allerdings mit schlechter Auflösung.

Ein Beweis dafür, daß jeder planare Graph geraden Linien für die Kanten gezeichnet werden kann, stammt von Fary (1948) [Far48]. Wir nennen solche Zeichnungen deshalb **Fary-Einbettungen**. Wir entwickeln einen effizienten Algorithmus für Fary-Einbettungen auf einem $O(n) \times O(n)$-Gitter.

Quellen: H. de Fraysseix, J. Pach, R. Pollak: *How to draw a planar graph on an grid.* Combinatorica 10 (1990), 41-51. [dFPP90]
H. Chrobak, T. Payne: *A linear time algorithm for drawing planar graphs.* Information Processing Letters 54 (1995), 241-246. [CP95]

Lemma 8.1.1
Sei G ein einfacher planar eigebetteter Graph und $u = u_1, u_2, \ldots, u_k = v$ ein Kreis in G, dann existiert ein Knoten $w' \notin \{u, v\}$ auf dem Kreis, der nicht adjazent zu einer inneren Sehne ist. (Ebenso existiert ein Knoten $w'' \notin \{u, v\}$, der nicht adjazent zu einer äußeren Sehne ist.)

Beweis: Keine Sehne: trivial.
Sei (u_i, u_j) $j > i + 1$ eine innere Sehne mit $j - i$ minimum.



- Wegen Minimalität gilt:

  Keine innere Sehne adjazent zu u_{i+1} endet im Bereich $u_i, u_{i+1}, \dots, u_j$.

- Wegen Planarität gilt:

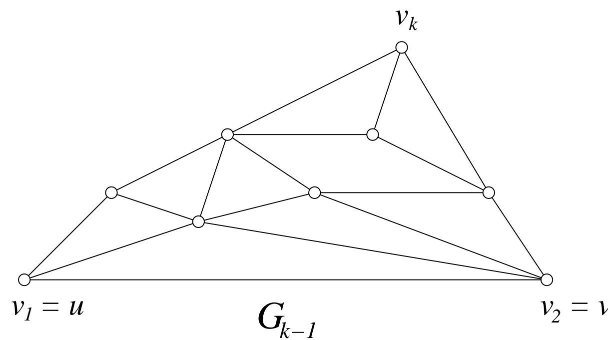
  Keine innere Sehne adjazent zu u_{i+1} endet im Bereich $\{u_1, \dots, u_k\} \backslash \{u_i, \dots, u_j\}$.

$\square$

Lemma 8.1.2 (Kanonische Numerierung)
Sei $G = (V, E)$, $|V| = n$ ein maximal planar eingebetteter Graph mit Außenfläche $u, v, w \in V$. Dann existiert eine Knotennumerierung $v_1 = u, v_2 = v, v_3, v_4, \dots, v_{n-1}, v_n = w$ mit den folgenden Eigenschaften für $4 \leq k \leq n$.

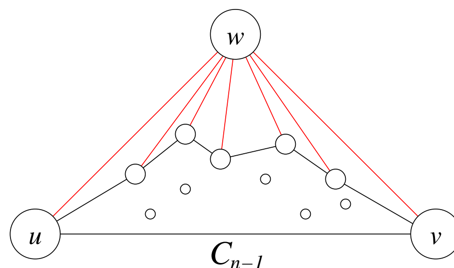
(i) Der von $v_1, v_2, \dots, v_{k-1}$ induzierte Subgraph G_{k-1} ist 2-zusammenhängend und die Außenfläche wird durch einen Kreis C_{k-1} begrenzt, der die Kante (u, v) enthält.

(ii) v_k liegt in der Außenfläche von G_{k-1} und seine Nachbarn in G_{k-1} bilden wenigstens ein 2-elementiges Intervall auf dem Weg $C_{k-1} \backslash \{u, v\}$.



Beweis: Wir definieren $v_n, v_{n-1}, \dots, v_3$ durch Rückwärtsinduktion. $v_n = w$, G_{n-1} Subgraph von G nach Entfernung von w.

Maximalität $\Rightarrow$ Nachbarn von w formen Kreis C_{n-1} der (u, v) enthält und Außenfläche von C_{n-1} definiert.



Sei v_k definiert für $k > i$, so daß G_{k-1} die Bedingungen (i) und (ii) erfüllt. Sei C_{k-1} der die Außenfläche begrenzende Kreis. Wir wenden Lemma 8.1.1 auf C_i in G_i an:

$\Rightarrow$ $(\exists w' \in C_i \backslash \{u, v\})$ w' ist nicht adjazent zu einer Innensehne von C_i (es gibt keine Außensehnen).

$\Rightarrow$ Für $v_i := w'$ erfüllt G_{i-1} die Bedingungen (i) und (ii).

□

Wir nennen die Numerierung aus Lemma 8.1.2 **kanonische Numerierung**.

Satz 8.1.3 (de Fraysseix, Pach, Pollack [dFPP90])
Jeder planar eingebettete Graph $G = (V, E)$ $|V| = n$ hat eine Fary-Einbettung auf einem $(2n-4) \times (n-2)$ Gitter.

Beweis: Gitterposition von $v \in V$:

$$P(v) = (x(v), y(v)) \in \mathbb{R}^2.$$

Manhattan Distanz von $A = (x_1, y_1)$ und $B = (x_2, y_2)$:

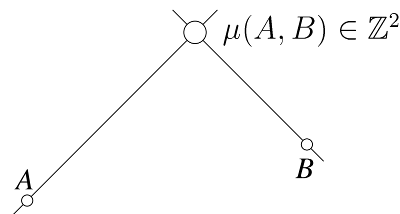
$$d_1(A, B) = |x_1 - x_2| + |y_1 - y_2|.$$

Haben A und B eine gerade Manhattan Distanz, so hat der Schnitt der Geraden

durch A mit Steigung 1
durch B mit Steigung -1

ganzzahlige Koordinaten

$$\mu(A, B) = \frac{1}{2}(x_1 - y_1 + x_2 + y_2, -x_1 + y_1 + x_2 + y_2).$$



Wir beschreiben einen Algorithmus: $v_1, v_2, \ldots, v_n$ sei kanonische Numerierung. Knotenlisten $L(w)$ für alle $w \in V$.

Initialisierung:

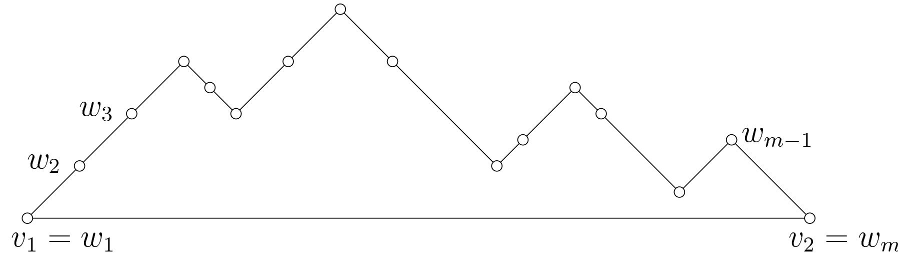
$P(v_1) := (0, 0)$	$L(v_1) := \{v_1\}$
$P(v_1) := (0, 2)$	$L(v_1) := \{v_2\}$
$P(v_1) := (1, 1)$	$L(v_1) := \{v_3\}$

In Schritt k sei G_{k-1} bereits eingebettet (d.h. Initialisierung entspricht Schritt 3), so daß

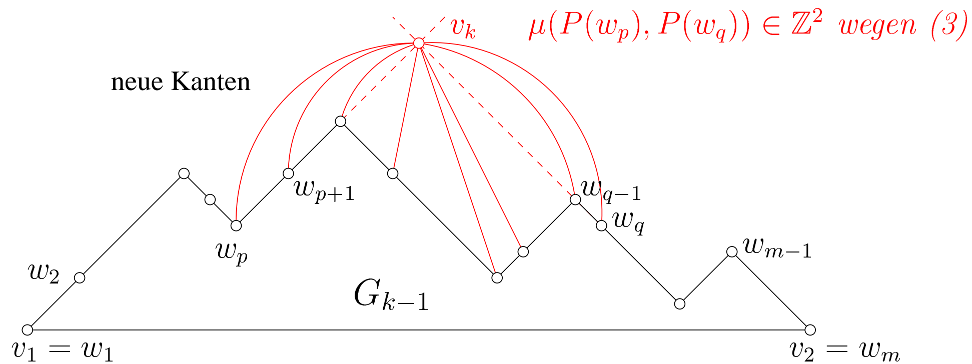
(1) $P(v_1) = (0, 0)$
$P(v_2) = (2(k-1) - 4, 0)$

(2) $x(w_1) < x_(w_2) < \ldots < x(w_m)$, wobei $C_{k-1} = (v_1 = w_1, w_2, \ldots, w_{m-1}, w_m = v_2)$.

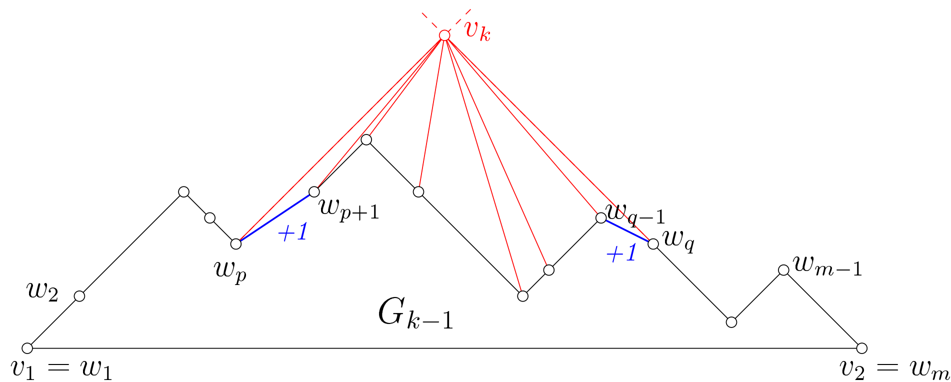
(3) Alle Kanten(-Zeichnungen) $(P(w_1, P(w_{i+1}))$ $(i = 1, 2, \ldots, m-1)$ haben Steigung $+1$ oder -1.



Seien $w_p, w_{p+1}, \ldots, w_{q-1}, w_q$ die Nachbarn von v_k auf C_{k-1}. Dann **überdeckt** v_k die Knoten $w_{p+1}, \ldots, w_{q-1}$.



Problem: v_k muß alle Knoten $w_p, \ldots, w_q$ sehen. Sicherstellung durch geeignete Verschiebungen:



“Innenleben muß geeignet verschoben werden, so daß Fary Eigenschaft auch innen erhalten bleibt”

1. for each $v \in \bigcup_{i=q}^{m} L(w_i)$ do $x(v) := x(v) + 2$

2. for each $v \in \bigcup_{i=p+1}^{q-1} L(w_i)$ do $x(v) := x(v) + 1$

3. $P(v_k) := \mu(P(w_p), P(w_q))$

4. $L(v_k) := \{v_k\} \cup \bigcup_{i=p+1}^{q-1} L(w_i)$

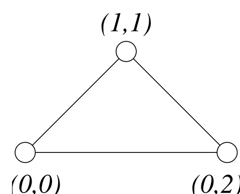
(Im Fall $p + 1 = q$ ist $\bigcup_{i=p+1}^{q-1} = \emptyset$.)

Invariante: Ist G_{k-1} Fary eingebettet mit (1), (2), (3) und man berechnet die Einbettung von G_k wie oben, so ist G_k Fary eingebettet mit (1), (2), (3). Erhaltung von (1), (2), (3) ist trivial.

Fary: $\forall 3 \leq k \leq n$ gilt: Sind $0 \leq \alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_m$ nichtnegative ganze Zahlen und werden für $i = 1, 2, \ldots, m$ alle Knoten in L_i um α_i nach rechts verschoben, so bleibt die Fary-Eigenschaft erhalten.

Induktion über k:

$k = 3$: offensichtlich



Die Behauptung gelte für G_{k-1} mit Kontur $w_1, w_2, \ldots, w_m$. Wir fügen v_k hinzu und erhalten die neue Kontur $w_1, w_2, \ldots, w_p, v_k, w_q, \ldots, w_m$. Seien $0 \leq \alpha_1 \leq \ldots \leq \alpha_p \leq \alpha \leq \alpha_q \leq \ldots \leq \alpha_m$ ganze Zahlen und betrachte die entsprechenden Verschiebungen der Listen:

$$
\begin{aligned}
&L(w_i) \; (i = 1, 2, \ldots, p, q, \ldots, m) \text{ um } \alpha_i \\
&L(v_k) \;\; \text{um } \alpha
\end{aligned}
$$

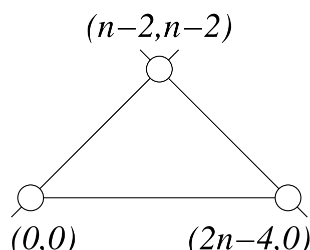
Wir zeigen: G_k ist danach Fary-eingebettet.
Seien $0 \leq \alpha'_1 \leq \alpha'_2 \leq \ldots \leq \alpha'_m$ wie folgt

$$
a'_i = \begin{cases} \alpha_i, & \text{für } i = 1, 2, \ldots, p \\ \alpha + 1, & \text{für } i = p+1, p+2, \ldots, q-1 \\ \alpha + 2, & \text{für } i = q, q+1, \ldots, m \end{cases}
$$

Induktionsannahme
$\Rightarrow$ Alles unterhalb der Kontur (G_{k-1}) bleibt Fary eingebettet.
Außerdem bewegt sich die Kontur zwischen w_{p+1} und w_{q-1} fest um 1 nach rechts.
$\Rightarrow$ Alles oberhalb der Kontur (v_k und inzidente Kanten) ist Fary eingebettet.

Schließlich gilt in $G = G_n$:

$$
\begin{array}{r} P(v_1) = (0,0) \\ P(v_2) = (2n-4, 0) \end{array} \Rightarrow \mu(P(v_1), P(v_2)) = (n-2, n-2)
$$

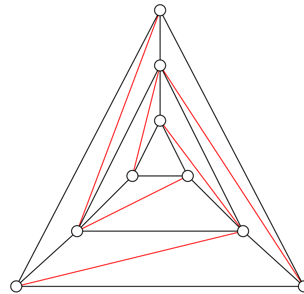


$\Rightarrow G$ ist im $(2n-4) \times (n-2)$ Gitter Fary-eingebettet.

□

Qualität des Platzbedarfs

Kleiner als $\left(\frac{2}{3}n-1\right) \times \left(\frac{2}{3}n-1\right)$ geht es nicht. Betrachte



$\frac{n}{3}$ verschachtelte Dreiecke plus rote und grüne Kanten.

inneres Dreieck: Breite/Höhe ≥ 1
für jedes weitere Dreieck: +2 Breite, +2 Höhe

$$\begin{aligned}\Rightarrow B/H &\geq 1 + 2(\#\Delta - 1) \\ &= 1 + \left(\frac{n}{3} - 1\right) \\ &= \frac{2}{3}n - 1\end{aligned}$$

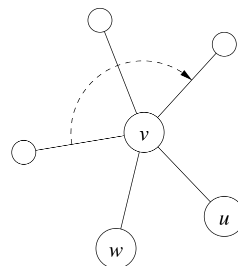
8.2 Ein Algorithmus zum geradlinigem Zeichnen

Eingabe: maximal planarer Graph $G = (V, E)$ mit kombinatorischer Einbettung mittels Uhrzeigerlisten.

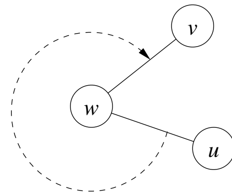
8.2.1 Herstellung einer Triangulierung

Bearbeite alle $v \in V$ wie folgt:

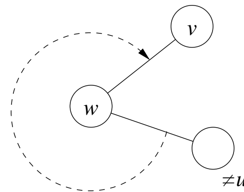
- Lese geordnete Uhrzeigerliste von v



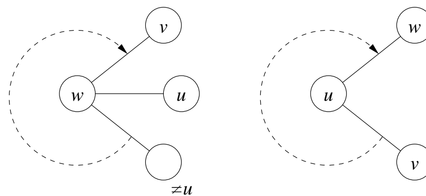
Sei w direkter Nachfolger von u, von w aus gesehen: Entweder



dann tue nichts oder

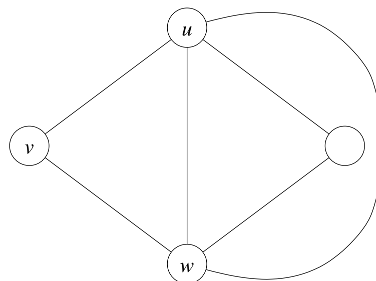


füge rote Kante (u, w) an richtiger Stelle ein:



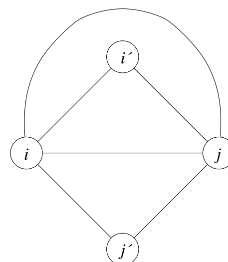
Konstante Zeit

Wir erhalten einen Multigraphen, denn (u, w) mag bereits in einer anderen Position existieren, z.B.



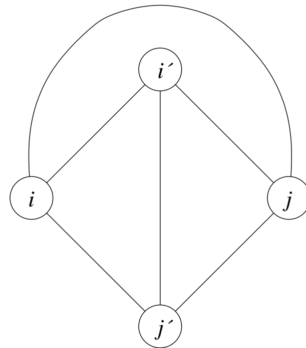
Ergebnis: Triangulierter planarer Multigraph. G' der G als Subgraph enthält.

Für jede rote Kante (i, j), die eine Parallelkante hat, existieren i', j' und ein eindeutiges Viereck



in dem keine Kante (i', j') möglich ist (wegen Planarität).

Aktion: Ersetze (i, j) durch $(i', j') \Rightarrow$ keine parallele Kante zu (i', j').

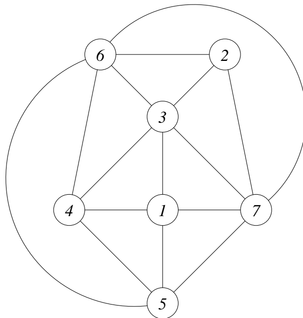


Effiziente Durchführung durch lexikographische Sortierung aller Kanten in G' zum Auffinden von Duplikaten, für diese Ersetzung in konstanter Zeit.

Sortieren in $O(n)$ Zeit mit Radix-Sortierung.

Insgesamt $O(n)$ Zeit (und Platz).

Beipiel 8.2.1
$G = (V, E)$ $V = \{1, 2, 3, 4, 5, 6, 7\}$

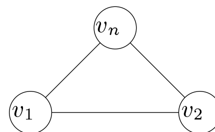


Uhrzeigerlisten		nach Triangulierung	
1:	$3, 7, 5, 4$	1:	$3, 7, 5, 4$
2:	$3, 6, 7$	2:	$3, 6, 7$
3:	$1, 4, 6, 2$	3:	$1, 4, 6, 2, 7$
4:	$1, 5, 3$	4:	$1, 5, 6, 3$
5:	$1, 6, 4$	5:	$1, 7, 6, 4$
6:	$2, 3, 5, 7$	6:	$2, 3, 4, 5, 7$
7:	$1, 2, 6$	7:	$1, 3, 2, 6, 5$

8.2.2 Bestimmung einer kanonischen Numerierung

Bestimmung von v_1, v_2, v_n

- Wähle v_1 beliebig.
- Wähle v_2 als Nachbarn von v_1 beliebig.



- Wähle v_n als v_1-Uhrzeiger-Nachfolger von v_2.

Bestimmung von $v_{n-1}, v_{n-2}, \ldots, v_3$

Datenstrukturen

Knotenfelder

- Zahlinnensehnen
- Istaußen
- Kanonische_Nummer
- w_p
- w_q

- Liste
  - Kandidatenliste

Initialisierung

for $k = 1, 2, \ldots, n$
  Zahlinnensehnen[k] = 0;
  Istaußen[k] = 0;
Istaußen[v_1] = Istaußen[v_2] = Istaußen[v_3] = wahr;
Kanonische_Nummer[v_1] = 1;
Kanonische_Nummer[v_2] = 2;
Kandidatenliste: v_n;

Für $k = n, n-1, \ldots, 3$ {
  Entferne v_k aus der Kandidatenliste;
  Kanonische_Nummer[v_k] = k;
  Bestimme $w_p[v_k]$ und $w_q[v_k]$ in C_k;
  Istaußen[v_k] = falsch;
  Falls $w_p[v_k]$ und $w_q[v_k]$ einzige Nachbarn von v_k in G_k sind {
    Zahlinnensehnen[$w_p[v_k]$]−−;
    Zahlinnensehnen[$w_q[v_k]$]−−;
    Falls (Zahlinnensehnen[$w_p[v_k]$] == 0)
      Füge $w_p[v_k]$ in Kandidatenliste ein;
    Falls (Zahlinnensehnen[$w_q[v_k]$] == 0)
      Füge $w_q[v_k]$ in Kandidatenliste ein;
  }
  Für alle Nachbarn u von v_k echt zwischen $w_p[v_k]$ und $w_q[v_k]$ {
    Istaußen[u] = wahr;
    Bestimme $w_p[u]$ und $w_q[u]$;
    Für alle Nachbarn w von u echt zwischen $w_p[u]$ und $w_q[u]$
      Falls (Istaußen[w]) {
        Zahlinnensehnen[u]++;

```
            Zahlinnensehnen[w]++;
            Entferne ggf. w aus der Kandidatenliste;
         }
      Falls (Zahlinnensehnen[u] == 0)
         Füge u in Kandidatenliste ein;
   }
}
```

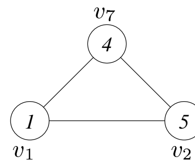
Laufzeit: $O(n)$ Jede Kante wird höchstens zweimal betrachtet.

Im Beispiel: Initialisierung

Wähle $v_1 = 1$

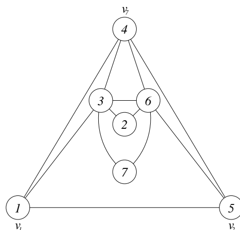
Wähle $v_2 = 5$

$\Rightarrow v_7 = 4$



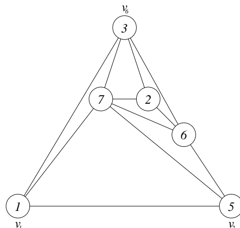
	1	2	3	4	5	6	7
Zahlinnensehnen	0	0	0	0	0	0	0
Istaußen	W	F	F	W	W	F	F
Kanonische_Nummer	1				2		
Kandidatenliste:	4						

$k = 7 : v_7 = 4 \quad w_p = 1 \quad w_q = 5$



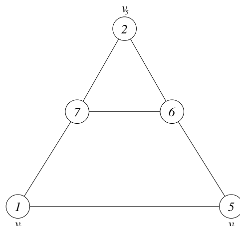
	1	2	3	4	5	6	7
Zahlinnensehnen	0	0	0	0	0	0	0
Istaußen	W	F	W	F	W	W	F
Kanonische_Nummer	1			7	2		
Kandidatenliste:	3	6					

$k = 6 : v_6 = 3 \quad w_p = 1 \quad w_q = 6$



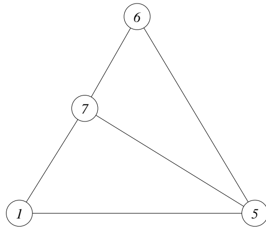
	1	2	3	4	5	6	7
Zahlinnensehnen	0	0	0	0	1	1	2
Istaußen	W	W	F	F	W	W	W
Kanonische_Nummer	1		6	7	2		
Kandidatenliste:	2						

$k = 5 : v_5 = 2 \quad \underbrace{w_p = 7 \quad w_q = 6}_{\text{einzige Nachbarn}}$



	1	2	3	4	5	6	7
Zahlinnensehnen	0	0	0	0	1	0	1
Istaußen	W	F	F	F	W	W	W
Kanonische_Nummer	1	5	6	7	2		
Kandidatenliste:	6						

$k = 4 : v_4 = 6 \quad \underbrace{w_p = 7 \quad w_q = 5}_{\text{einzige Nachbarn}}$



	1	2	3	4	5	6	7
Zahlinnensehnen	0	0	0	0	0	0	0
Istaußen	W	F	F	F	W	F	W
Kanonische_Nummer	1	5	6	7	2	4	

$k = 3 : v_3 = 7$

8.2.3 Koordinatenbestimmung

Darstellung von G_k als Wald bestehend aus Bäumen für $L(w_1), L(w_2), \ldots, L(w_m)$ mit Wurzeln $w_1, w_2, \ldots, w_m$.

Beobachtung 8.2.2
Wenn v_k eingebettet wird, brauchen wir die exakten Positionen von w_p und w_q nicht zu kennen.

Aus

- $y(w_p), y(w_q)$
- $x(w_q) - x(w_p)$ [relative x-Koordinaten]

können wir berechnen

- $y(v_k)$
- $x(v_k) - x(w_p)$ [x-Verschiebung von v_k relativ zu w_p]

Phase 1

- Hinzufügen der Knoten.
- Berechnung von x-Verschiebungen und y-Koordinaten.

Phase 2

- Berechnung der endgültigen x-Koordinaten durch Akkumulation von x-Verschiebungen

Berechnung der Koordinaten

Der folgende Algorithmus berechnet die Koordinaten gemäß dem Algorithmus von de Fraysseix, Pach, Pollack [dFPP90] bei gegebener kanonischer Ordnung der Knoten v_1, …,v_n. Während des Algorithmus enthält $x(w_i)$ für einen Knoten w_i der Außenregion zunächst den Abstand in x-Richtung zum Vorgänger w_{i+1} auf der Außenregion.

Beginne mit v_1 auf Punkt $(0, 0)$.
$x(v_1) = x(v_2) = y(v_1) = y(v_2) = 0$

for $k := 3$ **to** n **do**
Die Außenfläche vor Einfügen von v_k sei $w_1, \ldots, w_r$.
Bestimme w_p und w_q.
Berechne Abstand $x_q := x_{abs}(w_q) - x_{abs}(w_p)$ nach dem Hinzufügen von v_k,
d.h. nach Verschieben von w_q um 2.
$x_q := 2 + \sum_{i=p+1}^{q} x(w_i)$
Berechne Schnittpunkt der Geraden durch w_p und w_q mit Steigung +1 bzw. −1.
$x(v_k) := (x_q + y(w_q) - y(w_p))/2$
$y(v_k) := (x_q + y(w_q) - y(w_p))/2$
$x(w_q) := x_q - x(v_k)$
Die Knoten $w_{p+1}, \ldots, w_{q-1}$ werden nach Hinzufügen von v_k zu inneren Knoten.
Wir merken uns für diese Knoten die x-Position relativ zu v_k nach Hinzufügen von v_k, da dieser relative Abstand im weiteren Verlauf des Algorithmus nicht mehr verändert wird. Außerdem merken wir uns in $upper(w_i)$ den Bezugsknoten v_k.
for $i := p + 1$ **to** $q - 1$ **do**

$$x(w_i) := \underbrace{1 + \sum_{j=p+1}^{i} x(w_j)}_{\text{Abstand von } w_i \text{ zu } w_p \text{ nach Hinzufügen von } v_k} - x(v_k)$$

$upper(w_i) := v_k$
od
od

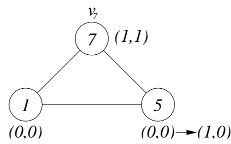
Bestimme absolute x-Koordinaten auf der Außenregion $(= v_1, v_2, v_n)$.
$x(v_2) := x(v_n) + x(v_2)$

Bestimme absolute x-Koordinaten für alle inneren Knoten.
for $k := n - 1$ **downto** 3 **do**
$x(v_k) := x(v_k) + x(upper(v_k))$
od

Im Beispiel: kanonische Numerierung:

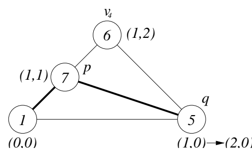
$$v_1 = 1,\ v_2 = 5,\ v_3 = 7,\ v_4 = 6,\ v_5 = 2,\ v_6 = 3,\ v_7 = 4$$

$k = 3$:



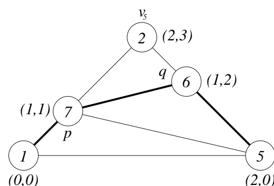
$$\begin{aligned} x_q &= 2 \\ x(v_3) &= \frac{(2+0-0)}{2} = 1 \\ y(v_3) &= \frac{(2+0+0)}{2} = 1 \\ x(w_q) &= 2 - 1 = 1 \end{aligned}$$

$k = 4$:



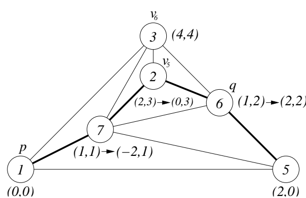
$$\begin{aligned} x_q &= 2 + 1 = 3 \\ x(v_4) &= (3 + 0 - 1)/2 = 1 \\ y(v_4) &= (3 + 0 + 1)/2 = 2 \\ x(w_q) &= 3 - 1 = 1 \end{aligned}$$

$k = 5$:

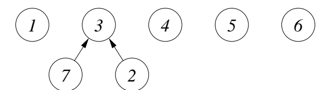


$$\begin{aligned} x_q &= 2 + 1 = 3 \\ x(v_5) &= (3 + 2 - 1)/2 = 2 \\ y(v_5) &= (3 + 2 + 1)/2 = 3 \\ x(w_q) &= 3 - 2 = 1 \end{aligned}$$

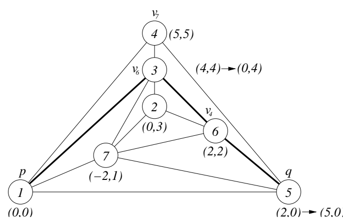
$k = 6$:



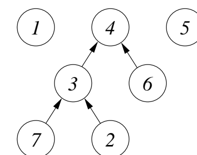
$$\begin{aligned} x_q &= 2 + 1 + 2 + 1 = 6 \\ x(v_6) &= (6 + 2 - 0)/2 = 4 \\ y(v_6) &= (6 + 2 + 0)/2 = 4 \\ x(w_q) &= 6 - 4 = 2 \\ x(v_3) &= 1 + x(v_3) + x(v_5) - x(v_6) \\ &= 1 + 1 - 4 = -2 \\ x(v_5) &= 1 + x(v_3) + x(v_5) - x(v_6) \\ &= 1 + 1 + 2 - 4 = 0 \end{aligned}$$



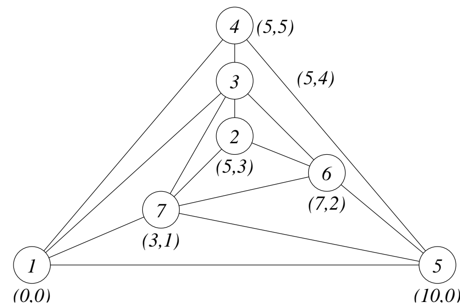
$k = 76$:



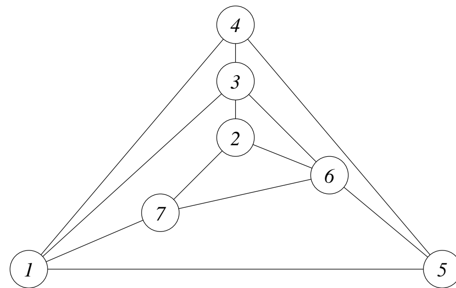
$$\begin{aligned} x_q &= 2 + 4 + 2 + 2 = 10 \\ x(v_7) &= (10 + 0 - 0)/2 = 5 \\ y(v_7) &= (10 + 0 + 0)/2 = 5 \\ x(w_q) &= 10 - 5 = 5 \\ x(v_6) &= 1 + 4 - 5 = 0 \\ x(v_4) &= 1 + 4 + 2 - 5 = 2 \end{aligned}$$



Berechnung der absoluten x-Koordinaten:



Fertige Zeichnung nach Entfernung der künstlichen Kanten:

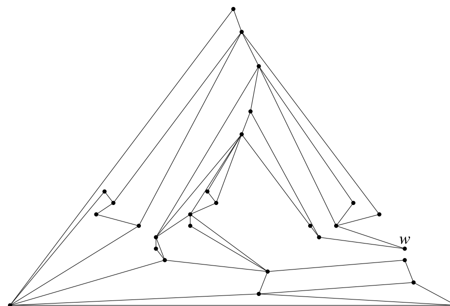


8.3 Verbesserungen und weitere Entwicklungen

- **Nachteil: Forderung der maximalen Planarität**

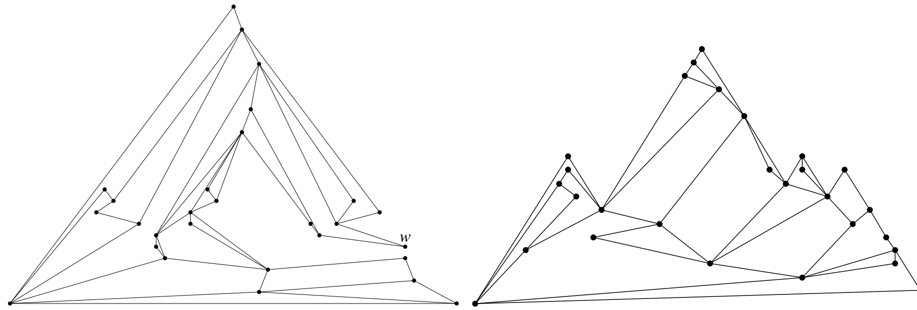
  Nach Entfernung der künstlichen Kanten sieht die Zeichnung eventuell merkwürdig aus.

  Beipiel 8.3.1 (Harel & Sardas (1998) [HS98])

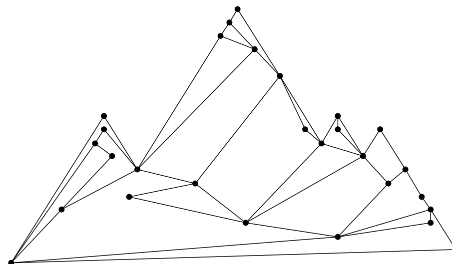
  

  → Kanonische Numerierung für 3-zusammenhängende (nicht notwendigerweise triangulierte) Graphen: Kant (1996) [Kan96].

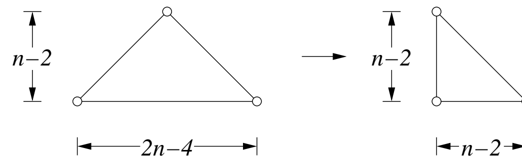
  → Kanonische Numerierung für 2-zusammenhängende Graphen: Gutwenger & Mutzel (1998) [GM98], Harel & Sardas (1998) [HS98]. (auch $O(n)$ Zeit)



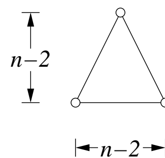
Bessere Zeichnung des Beispielgraphen (HS 2000):



- **Kleineres Gitter** (Chobak & Kant (1997) [CK97])



ganz anderes Verfahren, nur für maximal planare Graphen: Schnyder (1990) [Sch90]



- **Nachteil: Kleine Winkel**

Ausweg: Quasiorthogonales Zeichnen, d.h. orthogonal bis auf rechteckige Umgebungen nur die Knoten. Kant (1996) [Kan96], Gutwenger & Mutzel (1998) [GM98].

K1996:

3-zusammenhängende Graphen
$(2n-5) \times (3n-6)$-Gitter
$5n-15$ Knicke (max 3 pro Kante)
Winkel $\frac{2}{D}$ (D = maxmialer Knotengrad)
D.h. für 1-zusammenhängende Graphen Winkel $\frac{4}{3D+7}$.

GM1998:

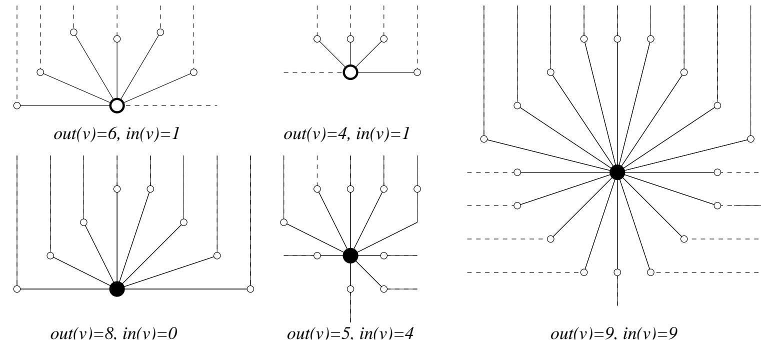
1-zusammenhängende Graphen

$(2n-5) \times \left(\frac{3}{2}n - \frac{7}{2}\right)$-Gitter

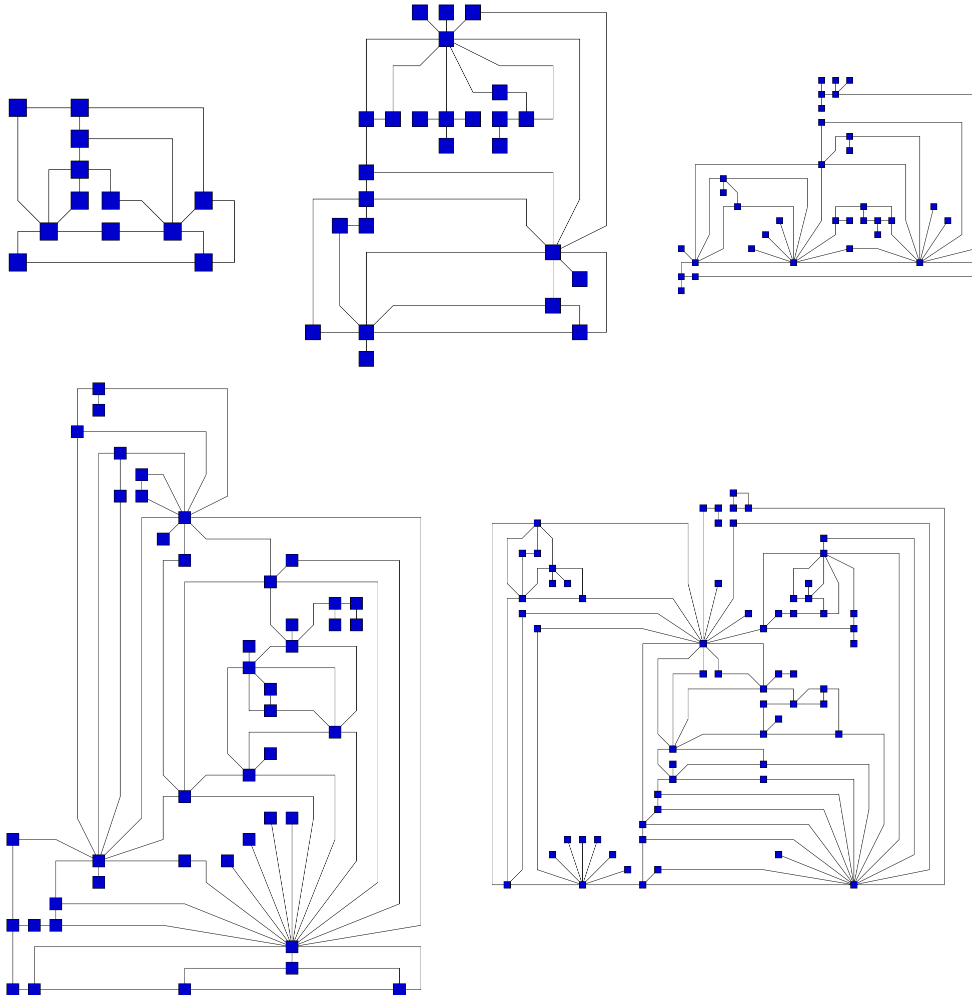
$5n - 15$ Knicke

Winkel $\frac{2}{D}$

Idee: Knotenboxen



$\text{in}(v_k) = \#\text{Nachbarn in } G_{k-1}$, $\text{out}(v_k) = \#\text{restliche Nachbarn}$



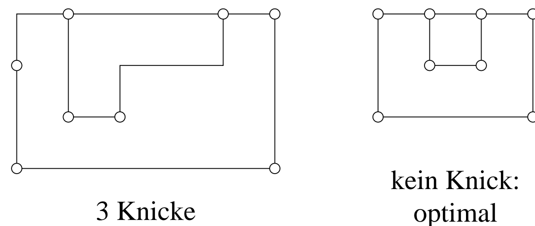
8.4 Orthogonales Zeichnen: Knickminimierung für 4-planare Graphen

4-planar:: planar und $\delta(v) \leq 4 \; \forall v \in V$.

Gegeben: 4-planarer Graph $G = (V, E)$.

Gesucht: planare orthogonale Gitterzeichnung mit minimaler Anzahl von Knicken.

Beipiel 8.4.1



Komplexität

Es ist $\mathcal{NP}$-vollständig, zu entscheiden ob G eine orthogonale Zeichnung ohne Knicke hat (Formann et al (1990) [FHH+93]). Aber:

gegeben: 4-planarer Graph mit kombinatorischer planarer Einbettung.

gesucht: Planare orthogonale Gitterzeichnung, die die planare Einbettung beibehält, mit minimaler Anzahl von Knicken.

$\rightarrow$ R. Tamassia (1987) [Tam87]: polynomieller Algorithmus.

Eine **orthogonale Repräsentation** H beschreibt zusätzlich zur Topologie die **Form** einer planaren orthogonalen Zeichnung, d.h. H ist eine Erweiterung einer kombinatorischen Einbettung P. Die Beschreibung ist dimensionslos, d.h. es werden keine Angaben über die Längen der Kantensegmente gemacht.
H besteht aus erweiterten Flächenuhrzeigerlisten $H(f)$ für alle Flächen $f \in F$ von G. Ein Element $r \in H(f)$ hat die Form (e_r, s_r, a_r) wobei

(i) e_r eine f begrenzende Kante,

(ii) s_r ein Binärstring und

(iii) $a_r \in \{90, 180, 270, 360\}$ ist.

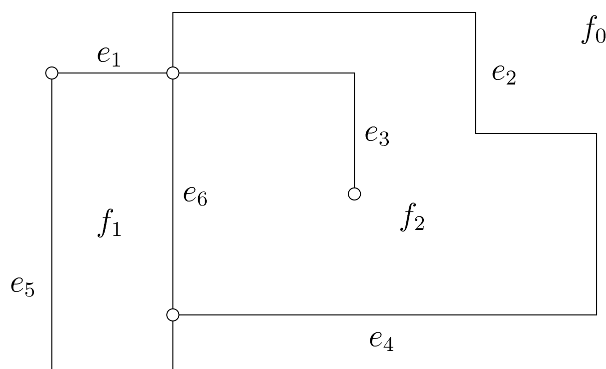
zu (ii): s_r beschreibt die Form der Kante e_r: Das k-te Bit in s_r beschreibt den k-ten Knick auf e_r:

$$\begin{array}{l} 0 \text{ für } 90 - Knick \\ 1 \text{ für } 270 - Knick \end{array}$$

relativ zu Durchlaufrichtung. (Erinnerung: im Uhrzeigersinn für Innenflächen, gegen den Uhrzeigersinn für die Außenfläche.)

zu (iii): Ist q das auf r folgende Element in $H(f)$, so beschreibt a_r den Winkel zwischen e_r und e_q in f.

Beipiel 8.4.2



$$\begin{aligned} H(f_1) &= ((e_1, \varepsilon, 90), (e_6, \varepsilon, 180), (e_5, 00, 90)) \\ H(f_2) &= ((e_6, \varepsilon, 90), (e_3, 0, 360), (e_3, 1, 90), (e_2, 0010, 90), (e_4, \varepsilon, 90)) \\ H(f_3) &= ((e_4, \varepsilon, 270), (e_2, 1011, 90), (e_1, \varepsilon, 270), (e_5, 11, 90)) \end{aligned}$$

(Jede Kante kommt genau zweimal vor.)
Die Anzahl der Knicke einer orthogonalen Repräsentation H ist

$$b(H) + \frac{1}{2} \sum_{f \in F} \sum_{r \in H(f)} |s_r|$$

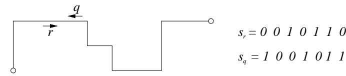
wobei $|s_r|$ die Länge von s_r bezeichnet.

Satz 8.4.3
Eine Menge von geordneten Listen H ist genau dann die orthogonale Repräsentation einer planaren orthogonalen Zeichnung, wenn folgende Eigenschaften erfüllt sind:

(E1) H ohne die s- und e-Felder ist eine planare Einbettung.

(E2) Seien r und q, $r \neq q$, zwei Elemente in H mit $e_r = e_q$. Dann erhält man s_q aus s_r durch Umdrehen der Reihenfolge und Flippen der Bits.

Beipiel 8.4.4



(E3) *Für jedes r sei $\varrho(r) = |s_r|_0 - |s_r|_1 + \left(2 - \frac{a_r}{90}\right)$, wobei $|s_r|_0$ ($|s_r|_2$) die Anzahlen der Nullen (Einsen) in s_r bezeichnet. Dann gilt*

$$(\forall f \in F) : \sum_{r \in H(f)} \varrho(r) = \begin{cases} -4, & \textit{falls } f \textit{ Außenfläche} \\ +4, & \textit{falls } f \textit{ Innenfläche} \end{cases}$$

D.h. jede Fläche ist ein rektilineares Polygon.

(E4) *F jeden Knoten $v \in V$ gilt:*

$$\sum_{e_r \textit{ inzident mit } v} a_r = 360.$$

Beweis: elementare geometrische Überlegungen.

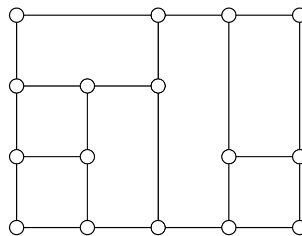
□.

Aus einer orthogonalen Repräsentation H kann man leicht eine entsprechende orthogonale Zeichnung erzeugen:

Skizze eines Algorithmus zur Längenbestimmung

(1) Konstruiere normalisierte orthogonale Repräsentation H', in der jede Fläche Rechteckform hat, durch Hinzufügen künstlicher Kanten und Knoten.

Beipiel 8.4.5



(2) Bestimme Längen der Kanten in H' → orthogonale Zeichnung für H'.

(3) Entferne die künstlichen Kanten und Knoten → orthogonale Zeichnung für H.

Netzwerkflüsse mit minimalen Kosten

Ein **Netzwerk** N ist ein 6-Tupel

$$N = (U, A, b, l, n, c)$$

mit:

- (U, A) ist ein gerichteter Graph
- $b \in \mathbb{Z}^U$ mit $\sum_{i \in U} b_i = 0$,
  $i \in U$ ist Angebotsknoten $\Leftrightarrow b_i > 0$, Nachfrageknoten $\Leftrightarrow b_i < 0$.
- $l, n \in \mathbb{Z}^A$,
  $a \in A$: l_a untere, u_a obere Schranke für Fluß auf a.
- $c \in \mathbb{Z}^A$,
  $a \in A$: c_a Kosten für eine Einheit Fluß auf a.

Minimum Kosten Netzwerk-Flußproblem

$$\begin{aligned}(MKNFP) \quad \text{minimiere } c^T x &= \sum_{a \in A} c_a x_a \\ \text{so daß } \sum_{j|(i,j) \in A} x_{(i,j)} - \sum_{j|(j,i) \in A} x_{(j,i)} &= b_i \ \forall i \\ l_a \leq x_a \leq u_a \ \forall a \in A& \\ x_a \in \mathbb{Z} \ \forall a \in A&\end{aligned}$$

$MKNFP$

- hat zahlreiche Anwendungen im Operations Research (insbesondere Warentransporte)
- ist in polynomieller Zeit lösbar

Bestimmung einer knickoptimalen orthogonalen Repräsentation

gegeben: Planarer Graph $G = (V, E)$ mit kombinatorischer Einbettung $\{P(f) \mid f \in F\}$, Außenfläche $f_o \in F$.

gesucht: orthogonale Repräsentation $\{H(f) \mid f \in F\}$ mit minimaler Anzahl von Knicken.

Plan: Transformation auf ein $MKNFP$.

Aufbau des Netzwerkes $N_P = (U, A, b, l, u, c)$

$U = U_V \dot\cup U_F$ mit

$$
\begin{aligned}
U_V &:= \{i_v \mid v \in V\} \\
U_F &:= \{i_f \mid f \in F\} \\
b_{i_v} &= 4 \quad \forall i_v \in U_V \\
b_{i_f} &= \begin{cases} -2|P(f)| + 4 & f \neq f_0 \\ -2|P(f)| - 4 & f = f_0 \end{cases}
\end{aligned}
$$

$\Rightarrow$ Gesamtes Potential =

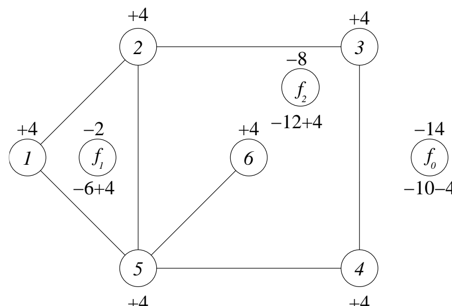
$$
\begin{aligned}
\sum_{v \in V} b_{i_v} + \sum_{f \in F} b_{i_f} &= 4|V| + \sum_{f \in F,\, f \neq f_0} (-2|P(f)| + 4) - 2|P(f_0)| - 4 \\
&= 4|V| - 2\sum_{f \in F} |P(f)| + 4(|F| - 1) - 4 \\
&= 4|V| - 4|E| + 4|F| - 8 \\
&= 4(\underbrace{|V| - |E| + |F| - 2}_{=0 \text{ Euler}}) \\
&= 4 \cdot 0 \\
&= 0
\end{aligned}
$$

Interpretation

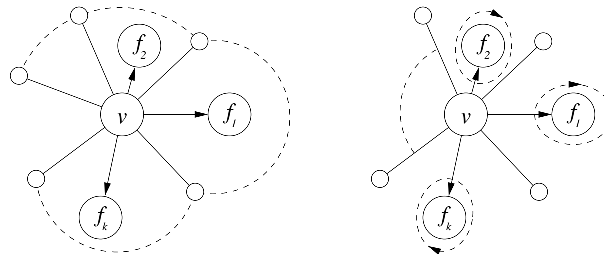
Fluß $\hat{=}$ Winkeln. Winkel enstspricht an den Knoten produzierte Ware pro Flußeinheit 90°. Flußerhaltung:

> $i_v \in U_V \hat{=}$ der Winkelsumme 360° an jedem Knoten.
> $i_f \in U_F \hat{=} f$ ist rektilineares Polygon mit korrekter Summe von Innen- und Außenwinkel

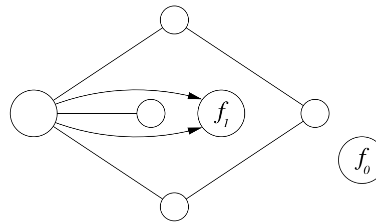
Beipiel 8.4.6



Kantenmenge $A = A_V \dot\cup A_F$.



A_V enthält eine Kante (i_v, i_f) für jedes Paar adjazenter Kanten in der Uhrzeigerliste von v wobei f die von dem Paar berandete Fläche ist. Mehrfachkanten sind möglich, z.B.



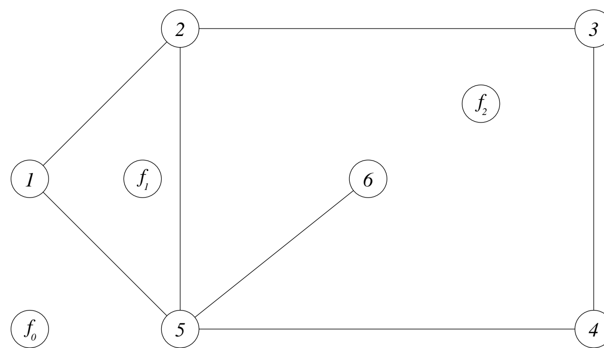
D.h. es entsteht ein Multidigraph $\forall (i_v, i_f) \in A_V$:

$$
\begin{aligned}
l_{(i_v, i_f)} &= 1 \\
u_{(i_v, i_f)} &= 4 \\
c_{(i_v, i_f)} &= 0
\end{aligned}
$$

Interpretation

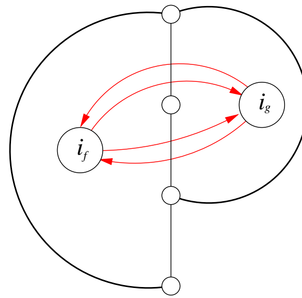
Die Knoten verteilen ihre Winkel an die adjazenten Flächen. Winkel sind wenigstens 90°, höchstens 360°. Es entstehen keine Knicke: Kosten = 0.

Beipiel 8.4.7

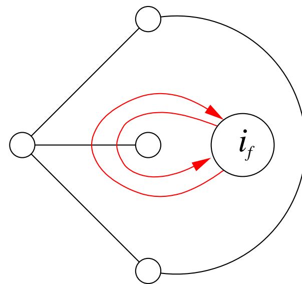


$$A_F := \{(i_f, i_g), (i_g, i_f) \mid e \in F \text{ trennt } f \text{ und } g\}.$$

Mehrfachkanten möglich, z.B.



Schleifen möglich, z.B.



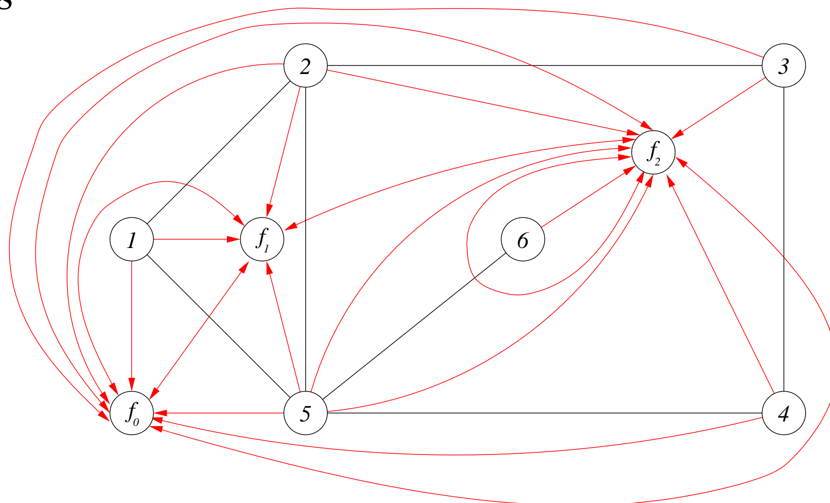
$\forall (i_f, i_g) \in A_F$:

$$l_{(i_f,i_g)} = 0$$
$$u_{(i_f,i_g)} = \infty$$
$$c_{(i_f,i_g)} = 1$$

Interpretation

Jede Flußeinheit auf einer A_F-Kante entspricht einem Knick, deshalb Kosten 1.

Beipiel 8.4.8



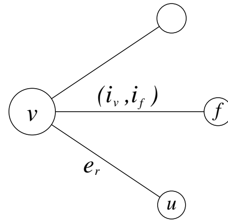
Satz 8.4.9

Zu jeder orthogonalen Repräsentation mit zugrunde liegender planarer Einbettung P existiert ein ganzzahliger Fluß x in N_P dessen Kosten $c^T x$ gleich der Anzahl der Knicke $b(H)$ ist.

Beweis: Konstruktion von Fluß x:

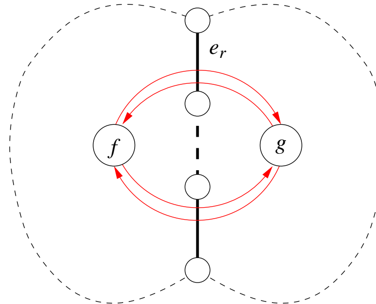
$$R(v,f) := \{r \in H(f) \mid e_r \text{ und ihre Nachfolgerin in } H(f) \text{ sind mit } v \text{ inzident}\}$$

Mit jeder grünen Kante (i_v, i_f) assoziieren wir das „Vorgängerelement“ $r \in R(v,f)$:



Wir setzen $x_{(i_v,i_f)} = \frac{a_r}{90}$

$$R(f,g) := \{r \in H(f) \mid e_r \in P(g)\}$$



Mit jeder roten Kante (i_f, i_g) assoziieren wur die „duale“ schwarze Kante. Wir setzen $x_{(i_f,i_g)} = |s_r|_0$

Zu zeigen:

1. x ist ein Fluß, d.h.
   (a) Kapazitäten werden eingehalten
   (b) Flußerhaltung ist erfüllt
2. Flußknoten = Anzahl der Knicke

zu 1a:

$$a \in A_V \;\; \alpha \in \{90, 180, 270, 360\} \Rightarrow x_a \in \{1,2,3,4\}$$
$$a \in A_F \text{ Länge eines Bitstrings ist } \geq 0$$

zu 1b: Bilanz an $i_v \in U_V$:

$$\begin{aligned}\sum_{f \in F} x_{(i_v,i_f)} - 0 &= \sum_{f \in F} \sum_{r \in R(v,f)} \frac{a_r}{90} \\ &= \sum_{e_r=(u,v)} \frac{a_r}{90} \\ &\overset{(E4)}{=} \frac{360}{90} \\ &= 4 = b_{i_v}\end{aligned}$$

Bilanz an $i_f \in U_F$:

$$\begin{aligned}
\sum_{g\in F} x_{(i_f,i_g)} - \sum_{v\in V_f} x_{(i_v,i_f)} - \sum_{h\in F} x_{(i_h,i_f)} &= \sum_{r\in H(f)} |s_r|_0 - \sum_{r\in H(f)} \frac{a_r}{90} - \sum r \in H(f)|s_r|_1 \\
&= \sum_{r\in H(f)} \left(2 - \frac{a_r}{90} - |s_r|_1 + |s_r|_0\right) - 2|P(f)| \\
&= -2|P(f)| + \sum_{r\in H(f)} \varrho(r) \\
&\overset{(E3)}{=} -2|P(f)| \begin{cases} +4 & \text{falls } f \text{ Innenfläche} \\ -4 & \text{falls } f = f_0 \end{cases} \\
&= b_{i_f}
\end{aligned}$$

zu 2:

$$\begin{aligned}
b(H) &\overset{\text{Def.}}{=} \frac{1}{2} \sum_{f\in F} \sum_{r\in H(f)} |s_r| \\
&\overset{(E2)}{=} \sum_{f\in F} \sum_{g\in F} x_{(i_f,i_g)} \\
&= \sum_{a\in A_F} x_a \\
&= \sum_{a\in A} c_a x_a
\end{aligned}$$

□

Satz 8.4.10 *Zu jedem ganzzahligen Fluß x in N_P existiert eine orthogonale Repräsentation mit zugrunde liegender planarer Einbettung P, so daß $c^T x = b(H)$*

Beweis: Konstruktion von H:

$$a_r = x_{(i_v,i_f)} \cdot 90 \;\forall\, r \in R(v,f)$$

Zu jedem $r \in R(f,g)$ sei $q \in H(g)$ mit $e_r = e_q$. Wir setzen

$$\begin{aligned}
s_r &= 0^{x_{(i_f,i_g)}} 1^{x_{(i_g,i_f)}} \\
s_q &= 0^{x_{(i_g,i_f)}} 1^{x_{(i_f,i_g)}}
\end{aligned}$$

Zu zeigen: $(E1) - (E4)$ sind erfüllt.

$(E1)$: klar (P bleibt erhalten)

$(E2)$: nach Konstruktion

$(E3)$:

$$
\begin{aligned}
\sum_{r\in H(f)} \varrho(r) &= \sum_{r\in H(f)} \left(\left(2-\frac{a_r}{90}\right)+|s_r|_0-|s_r|_1\right)\\
&= \sum_{r\in H(f)} \left(2-\frac{x_{(i_v,i_f)}\cdot 90}{90}+\left|0^{x_{(i_f,i_g)}}\right|-\left|1^{x_{(i_g,i_f)}}\right|\right)\\
&= 2|P(f)| \underbrace{-\sum_{v\in V_F} x_{(i_v,i_f)}+\sum_{g\in F} x_{(i_f,i_g)}-\sum_{g\in F} x_{(i_g,i_f)}}_{b_{i_f}}\\
&= 2|P(f)|+(-2|P(f)|\pm 4)\\
&= \pm 4
\end{aligned}
$$

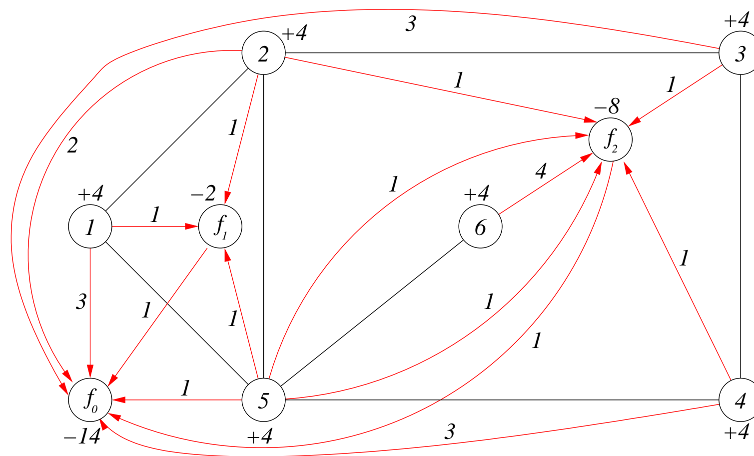
$(E4)$:

$$
\begin{aligned}
\sum_{e_r \text{inz. mit } v} a_r &= \sum_{f|v\in V_f} x_{(i_v,i_f)}\cdot 90\\
&= b_{i_v}\cdot 90\\
&= 4\cdot 90\\
&= 360
\end{aligned}
$$

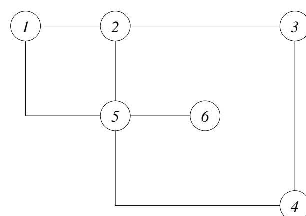
Knickanzahl: wie in Beispiel 8.4.2.

□

Beipiel 8.4.11



$c^T x = 2 = b(H)$:



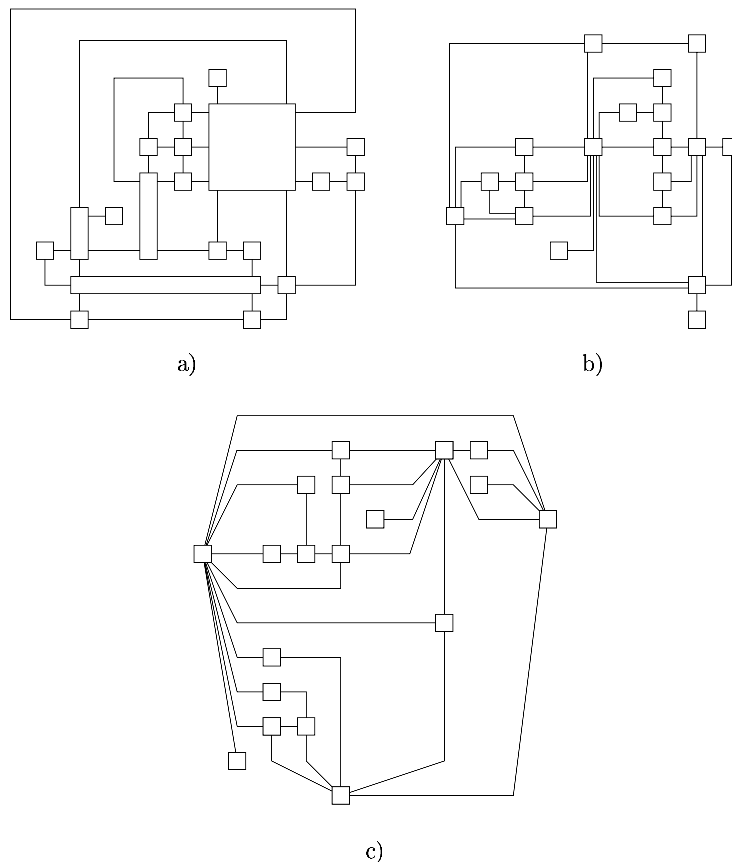
Gesamtes Verfahren

1. Konstruiere Netzwerk N_P.
2. Berechne kostenminimalen Fluß.
3. Konstruiere orthogonale Repräsentation.
4. Bestimme Kantenlängen.

$\rightarrow$ kann mit Laufzeit $O(|V|^2 \cdot \log |V|)$ implementiert werden.

8.5 Erweiterungen für beliebige Knotengrade

- Giotto (R. Tamassia, G. Di Battista, C. Batini (1988) [TBB88])
- Kandinski (U. Fößmeier, Kaufmann (1996) [FK96])
- AGD (G. Klau, P. Mutzel (1998) [KM98])



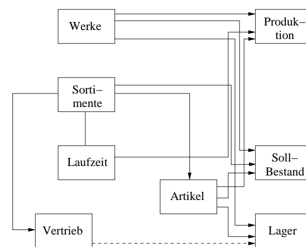
8.6 Planarisierungsmethode

Nutzung planarer Zeichenmethoden für nichtplanare Graphen.

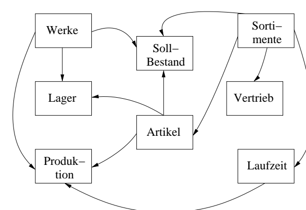
1. Bestimme einen planaren Subgraphen G_P von G durch Entfernung möglichst weniger Kanten.
2. Bette G_P planar ein.
3. Füge entfernte Kanten unter Erzeugung möglichst weniger Kreuzungen wieder ein.
4. Ersetze alle Kreuzungen durch künstliche Knoten → planarer Graph G'
5. Zeichne G' mit planarer Zeichenmethode
6. Entferne die künstlichen Knoten.

Planarisierungsmethode: Beispiel

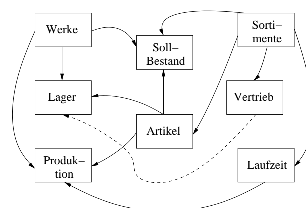
(1) Bestimmung eines maximum planaren Subgraphen → Entfernung der gestrichelten Kante



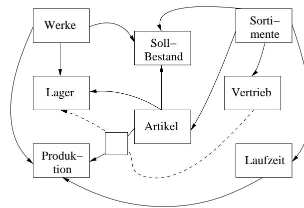
(2) Bestimmung einer planare Einbettung



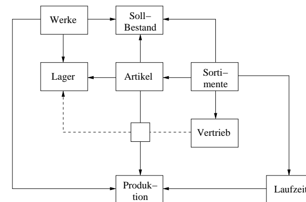
(3) Kreuzungsminimale Einfügung der entfernten Kante



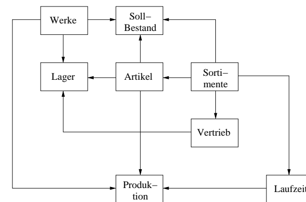
(4) Ersetzung der Kreuzung durch künstlichen Knoten



(5) Knickminimale Zeichnung mit Tamassias Verfahren



(6) Entfernung des künstlichen Knotens → fertige Zeichnung



zu (1):

- Greedy-Heuristik für $G = (V, E)$:

  $F = \emptyset$;
  foreach $e \in E$:
    if $G = (V, F \cup \{e\})$ planar
      $F := F \cup \{e\}$
  Output (V, E) maximal planar

  (jede neue Kante zerstört Planarität)

- Modifikation des Planaritätstest: Entferne „störende“ Kanten während des Tests (Jünger, Leipert, Mutzel (1997) [JLM97])

- Optimallösung

  minimiere $\sum_{e \in E} x_e$
  so daß $\sum_{e \in K} x_e \leq |K| - 1 \; \forall$ Kuratowski-Subgraphen$(V(K), K)$ von G
  $x_e \in \{0, 1\} \; \forall \, e \in E$

  via Branch and Cut (Jünger & Mutzel (1996) [JM96])

zu (3):

- Heuristik: kürzeste Wege in einem gewissen erweiterten dualen Graphen.

- Optimum: Mutzel & Ziegler (1999) [MZ99].

Index

Änderung, 28
  maximale, 28
Ästhetikkriterien, 11
Ästhetikkriterium, 15
überdecken, 160
2-Schichten-kreuzungsminimierung mit einer fixierten Schicht, 54
2-Zusammenhangskomponenten, 39
2-zusammenhängend, 39
2SKM1F, 54
3-zusammenhängend, 39
3SAT, 21
4-planar, 173

Adjazenzliste, 7
Adjazenzmatrix, 7
Akkumulationsbaum, 61

Barycenter, 65
Barycenter Method, 36
Baum, 9
Bipartite Multigraph Kreuzungszahl, 51
Blöcke, 39
Branch and Cut, 70

Coffman-Graham-Schichtung, 49

E2SKM1F, 54

Fary-Einbettung, 157
FAS, 54
Feedback Arc Set, 54
Flächenuhrzeigerlisten, 101
Folge
  äußere, 82
Folgen
  äußere, 80
  fixiert, 81
  innere, 80
Form, 173

Graph, 5
  bipartit, 51
  gerichtet, 6
  maximal planar, 107
  simpler, 5
  stark zusammenhängend, 6
  Subgraph, 7
  ungerichtet, 5
  zusammenhängend, 6
Greedy-Insert, 63
Greedy-Switch, 63

Hooke'sches Gesetz, 34

Klumpen, 55
Knoten-Uhrzeigerlisten, 100
kombinatorisch äquivalent, 101
kombinatorische Einbettung, 101
Konturen, 16
Kreis, 6

LP, 18

Median, 65
Multiprocessor Scheduling Problem, 49

Nachbarfolge, 82
Netzwerk, 176
Numerierung
  kanonisch, 158, 159

Optimal LinearArrangement, 51
Originalfolge, 80

Pfad, 6
planar, 39

Quelle, 6

Reingold & Tilford, 15
Repräsentation

orthogonal, 173
Rieman'sche stereogr. Projektion, 103

Satz von Kuratowski, 100
Schichtzuweisung, 46
Sehnen, 98
Senke, 6
Separationsknoten, 39
Separationspaar, 39
Split, 64

Topologische Sortierung, 47
Topsort, 47
transitive Hülle, 7
transitive Reduktion, 7
Triangulation, 107

unär, 147
Unterteilung, 100

Walker, 25
Wetherell & Shannon, 11
Wurzelbaum, 9
binärer, 9

Literaturverzeichnis

[ACNO84] S. Abe, N. Chiba, T. Nishizeki and T. Ozawa, *A Linear Algorithm for Embedding Planar Graphs Using PQ-Trees*, J. of Computer and System Science, 30 (1985) 54–76.

[AHU74] A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley (1974).

[AP61] L. Auslander and S. V. Parter, *On Embedding Graphs in the Plane*, J. Math. Mech. 11 (1961), 517–523.

[Aig84] M. Aigner, *Graphentheorie, eine Entwicklung aus dem 4-Farben Problem*, Teubner Studienbücher: Mathematik (1984).

[Bar00] W. Barth, unveröffentlicht (2000)

[BGHS92] H. K. B. Beck, H.-P. Galil, R. Henkel and E. Sedlmayr, *Chemistry in circumstellar shells, I. Chromospheric radiation fields and dust formation in optically thin shells of M-giants*, Astron. Astrophys. 265 (1992) 626–642.

[BBLM91] P. Bertolazzi, G. Di Battista, G. Liotta, C. Mannino, *Upward Drawing of Triconnected Digraphs*, Istituto Di Analisi Dei Sistemi Ed Informatica, Roma, R. 328 Dicembre 1991.

[BBMT93] P. Bertolazzi, G. Di Battista, C. Mannino, R. Tamassia, *Optimal Upward Planarity Testing of Single-Source Digraphs* (Extended Abstract), ESA 1993.

[BL76] K. S. Booth and G. S. Lueker, *Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity using PQ-Tree Algorithms*, Journal of Computer and System Sciences 13 (1976), 335–379.

[BLW77] N. L. Biggs, E. K. Lloyd, R. L. Wilson *Graph Theory 1736-1936*, Clarendon Press, Oxford (1977)

[BN87] G. Di Battista and E. Nardelli, *An Algorithm for Testing Planarity of Hierachical Graphs*, in G. Tinhofer and G. Schmidt, eds., Graph-Theoretic Concepts in Computer Science 246, Springer, Berlin (1987), 277–289.

[BO79] J. L. Bentley, T. Ottmann *Algorithms for reporting and counting geometric intersections* IEEE Trans. Comput. C-28 (1979) 643–647

[BT88] G. Di Battista and R. Tamassia, *Algorithms for Plane Representations of Acyclic Digraphs*, Theoretical Computer Science 61 (1988), 175–198.

[BT89] G. Di Battista and R. Tamassia, *Incremental Planarity testing*. Proc. 30th Annual IEEE-Symp. on Found. on Comp. Science, North Carolina (1989) 436–441.

[Cha86] B. Chazelle *Reporting and counting segment intersections*, Journal of Computer and System Sciences, 32 (1986) 156–182

[Cim92a] R. J. Cimikowski, *Graph Planarization and Skewness*, Congr. Numer. 88 (1992), 14–29.

[Cim92b] R. J. Cimikowski, *An Empirical Analysis of Graph Planarization Heuristics*, unpublished manuscript, Computer Science Deptartment, Montana State Univ. (1992).

[CHT89] J. Cai, X. Han and R. E. Tarjan, *New Solutions to four Planar Graph Problems*, Technical Report, Dept. of Computer Science, New York University, Courant Institute (1989).

[CHT93] J. Cai, X. Han and R. E. Tarjan, *An $O(m \log n)$-Time Algorithm for the Maximal Planar Subgraph Problem*, SIAM J. of Comput. 22 (1993) 1142–1162.

[CK97] M. Chobak, G. Kant *Convex grid drawing three-connected planar graphs* International Journal of Computational Geometry & Applications 7 (1997) 211–224

[CLR89] T. H. Cormen, C. E. Leiserson and L. Rivest, *Introduction to Algorithms*, The MIT Press, Massachusetts Institute of Thechnology (1989).

[CNS79] T. Chiba, I. Nishioka and I. Shirakawa, *An Algorithm for Maximal Planarization of Graphs*, Proc. 1979 IEEE Intern. Symp. on Circuits and Systems (1979) 336–441.

[CN88] N. Chiba and T. Nishizeki, *Planar Graphs: Theory and Algorithms*, Anals of Discrete Mathematics, North Holland (1988).

[CP95] M. Chrobak, T. Payne: *A linear time algorithm for drawing planar graphs.*, Inform. Process. Lett., 54 (1995), 241-246.

[dFPP90] H. de Fraysseix, J. Pach, R. Pollack *How to Draw a Planar Graph on a Grid*, Combinatorica, 10 (1990) 41–51

[DFF85] M. E. Dyer, L. R. Foulds and A. M. Frize, *Analysis of Heuristics for Finding a Maximum Weight Planar Subgraph*, Eurpopean J. Oper. Res. 20, (1985), 102–114.

[Dud17] H. E. Dudeney *Amusements in mathematics* Melson, London 1917

[DH96] R. Davidson, D. Harel *Drawing Graphics Nicely Using Simulated Annealing*, ACM Trans. Graph., 15, no. 4 (1996) 301–331

[Ead93] P. Eades, *personal communication* (1993).

[Eve79] S. Even, *Graph Algorithms*, Computer Science Press, Potomac, Maryland (1979).

[ET76] S. Even und R. E. Tarjan, *Computing an st-numbering*, Theor. Comput. Sci. 2 (1976), 339–334.

[EFG82] P. Eades, L. R. Foulds and J. W. Giffin, *An Efficient Heuristic for Identifying a Maximum Weight Planar Subgraph*, Combinatorial Mathematics IX, Lecture Notes in Mathematics 952, Springer, Berlin (1982).

[EK86] P. Eades, D. Kelly *Heuristics for Reducing Crossings in 2-Layered Networks*, Ars Combin., 21.A (1986) 89–98

[EKW86] P. Eades, B. McKay and N. Wormald, *On an Edge Crossing Problem*, Proc 9th Australian Computer Science Conf. (1986) 327–334.

[EM94] P. Eades and J. Marks, *personal communication* (1994).

[EW94] P. Eades, N. Wormald *Edge Crossings in Drawings of Bipartite Graphs*, Algorithmica 11 (1994) 379–403

[Far48] I. Fary *On Straight Lines Representation of Planar Graphs*, Acta Sci. Math. Szeged, 11 (1948) 229–233

[FGG85] L. R. Foulds, P. B. Gibbons and J. W. Giffin, *The Facilities Layout Problem: An Experimental Comparison of Graph Theoretic Heuristics*, (1985)

[FHH+93] M. Formann, T. Hagerup, J. Haralambides, M. Kaufmann, F. T. Leighton, A. Simvonis, E. Welzl, G. Woeginger *Drawing Graphs in the Plane with High Resolution*, SIAM J. Comput., 22 (1993) 1035–1052

[FK96] U. Fößmeier, M. Kaufmann *Drawing High Degree Graphs with Low Bend Numbers*, In F. J. Brandenbrug (Ed.), Graph Drawing 1995, Lecture Notes in Computer Science 1027 (1996) 254–266

[FR78] L. R. Foulds and D. F. Robinson, *Graph Theoretic Heuristics for the Plant Layout Problem*, Int. J. of Produktion Res. 16 (1978), 27–37.

[GKNV93] E. R. Ganser, E. Koutsofios, S. C. Norht and K.-P. Vo, *A Technique for Drawing Directed Graphs*, IEEE Trans. on Software Engineering, Vol. 19, (1993), 214–230.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman & Co., San Francisco (1979).

[GJ83] M. R. Garey, D. S. Johnson *Crossing Number is NP-Complete*, SIAM J. Algebraic Discrete Methods, 4, no. 3 (1983) 312–316

[GM98] C. Gutwenger, P. Mutzel *Planar Polyline Drawings with good angular Resolution* in S. H. Whitesites (Ed.) Graph Drawing 1998, Lecture Notes in Computer Science Vol. 1547 (1998) 167–182

[GT93] A. Garg and R. Tamassia, *On the Computational Complexity of Upward and Rectlinear Planarity Testing*, Technical Report RI 02912-1910, Dept. of Computer Science, Brown University (1993).

[GT92] O. Goldschmidt and A. Takvorian, *An Efficient Graph Planarization Two-Phase Heuristic*, University of Texas, Austin (1992).

[Har72] F. Harary, *Graph Theory*, Addison Wesley, Reading, Mass. (1972).

[HS93] D. Harel and M. Sardas, *Randomized Graph Drawing with Heavy-Duty Preprocessing*, INRIA, Raport de recherche N. 2147, Sophia Antipolis, Cedes 1993.

[Him93a] M. Himsolt, *Konzeption und Implementierung von Grapheneditoren*, Dissertation, Universität Passau (1993).

[Him93b] M. Himsolt, *personal communication* (1993).

[HT74] J. Hopcroft and R. E. Tarjan, *Efficient Planarity Testing*, J. ACM 21 (1974) 549–568.

[HL90] M. D. Hutton and A. Lubiw, *Upward Planar Drawing of Single Source Acyclic Digraphs*, Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms (1990).

[HL93] M. D. Hutton and A. Lubiw, *Upward Planar Drawing of Single Source Acyclic Digraphs*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 9, 1993.

[HS98] D. Harel, D. Sardas *An algorithm for straight-line drawing of planar graphs*, Algorithmica 20 (1998) 119–135

[Jor93] E. Jordan, *personal communication* (1993).

[JLM97] M. Jünger, S. Leipert, P. Mutzel *Pitfalls of Using PQ-Trees in Automatic Graph Drawing* in G. Battista (Ed.) Graph Drawing 1997, Lecture Notes in Computer Science 1353 (1997) 193–204

[JM92] M. Jünger, P. Mutzel, *Solving the Maximum Weight Planar Subgraph Problem by Branch and Cut*, Edit by L. A. Wolsey and G. Rinaldi, Proceedings of the 3rd IPCO Conference, Erice (1993).

[JM96] M. Jünger, P. Mutzel *Maximum Planar Subgraphs and Nice Embeddings: Practical Layout Tools*, Algorithmica, 16 (1996) 33–59

[JM97] M. Jünger, P. Mutzel *2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms*, Journal of Graph Algorithms and Applications, 1, no. 1 (1997) 1–25

[JTS86] R. Jayakumar, K. Thulasiraman and M. N. S. Swamy, *On Maximal Planarization of Non-Planar Graphs*, IEEE Trans. Circuits Syst., vol, CAS-33, no. 8 (1986) 843–844.

[JTS89] R. Jayakumar, K. Thulasiraman and M. N. S. Swamy, *$O(n^2)$ Algorithms for Graph Planarization*, IEEE Transactions on Computer-Aided Design, Vol. 8. No 3. (1989) 257–267.

[Kan92] G. Kant, *An $O(n^2)$ Maximal Planarization Algorithm based on PQ-trees*, Technical Report, RUU-CS-92-03, Dept. of Computer Science, Utrecht University (1992).

[Kan96] G. Kant *Drawing Planar Graphs Using the Canonical Ordering*, Algorithmica, 16 (1996) 4–32

[Kel87] D. Kelly, *Fundamentals of Planar Ordered Sets*, Discrete Mathematics 63 (1987), 197–216.

[Kur30] K. Kuratowski *Sur le problè me des courbes gauches en topologie* Fund. Math. 15 (1930) 271–283

[KK89] T. Kamada, S. Kawai *An Algorithm for Drawing General Undirected Graphs*, Inform. Process. Lett. 31 (1989) 7–15

[KM98] G. W. Klau, P. Mutzel *Quasi-Orthogonal Drawing of Planar Graphs* Technical Report MPI-I-98-1-013, Max-Planck-Institut für Informatik Saarbrücken, 1998

[KS80] J. B. Kruskal and J. B. Seary, *Designing Network Diagrams*, In Proc. First General Conference on Social Graphics (1980) 22–50

[LEC67] A. Lempel, S. Even and I. Cederbaum, *An Algorithm for Planarity Testing of Graphs, Theory of Graphs: International Symposium: Rome, July, 1966*, Gordon and Breach, New York (1967), 215–232.

[LG78] P. Liu and R. Geldmacher, *On the Deletion of Nonplanar Edges of a Graph*, Proc 10. Conf. on Combinatorics, Graph Theory and Computing (1978), 727–738.

[Mar93] A. Martin, *personal communication* (1993).

[Mut94a] P. Mutzel, *personal communication* (1994).

[Mut94b] P. Mutzel, *The Maximum Planar Subgraph Problem*, Dissertation, Universität zu Köln (1994).

[MZ99] P. Mutzel, T. Ziegler *The Constrained Crossing Minimization Problem* J. Kratochvíl (Ed.) Graph Drawing 1999, Lecture Notes in Computer Science, Vol. 1731 (1999) 175–185

[OT81] T. Ozawa, H. Takahashi, *A Graph-planarization Algorithm and its Application to Random Graphs*, Graph Theory and Algorithms, Springer Verlag, Lecture Notes in Computer Science, vol. 108 (1981) 95–107.

[Pla76] C. R. Platt, *Planar Lattices and Planar Graphs*, J. Comb. Theory (B) 21 (1976), 30–39.

[RT81] E. Reingold, J. Tilford *Tidier Drawing of Trees*, IEEE Trans. Softw. Eng., SE-7, no. 2 (1981) 223–228

[Sch94] U. Schnieders, *personal communication* (1994).

[Sch90] W. Schnyder *Embedding Planar Graphs on the Grid*, In Proc. 1st ACM-SIAM Sympos. Discrete Algorithms (1990) 138–148

[SM95] K. Sugiyama, K. Misue *Graph Drawing by Magnetic-Spring Model*, J. Visual lang. Comput., 6, no. 3 (1995)

[STT81] K. Sugiyama, S. Tagawa, M. Toda *Methods for Visual Understanding of Hierachial Systems*, IEEE Trans. Syste. Man Cybern., SMC-11, no.2 (1981) 109–125

[Tam87] R. Tamassia *On Embedding a Graph in the Grid with the Minimum Number of Bends*, SIAM J. Comput., 16, no. 3 (1987) 421–444

[TBB88] R. Tamassia, G. Di Battista and C. Batini, *Automatic Graph Drawing and Readability of Diagrams*, IEEE Transactions on Systems, Man and Cybernetics 18 (1988) 61–79.

[TT86] R. Tamassia and I. G. Tollis, *A Unified Approach to Visibility Representations of Planar Graphs*, Discrete & Compute Geom. 1 (1986), 321–341.

[Tho80] C. Thomassen, *Planarity and Duality of Finite and Infinite Graphs*, J. Combinat. Theory, series B29, (1980), 244–271.

[Tho89] C. Thomassen, *Planar Acyclic Oriented Graphs*, Order 5 (1989), 349–361.

[Tut60] tut60 W. T. Tutte, *Convex Representation of graphs*, Proceedings Mathematical Society, no. 3 (1960) 304–320.

[Tut63] W. T. Tutte, *How to draw a graph*, Proceedings Mathematical Society, no. 3 (1963) 743–768.

[Wal90] J. Q. Walker II *A Node-positioning Algorithm for General Trees*, Softw.-Pract. Exp., 20, no. 7 (1990) 685–705

[War77] J. Warfield, *Crossing Theory and Hierarchy Mapping*, IEEE Trans. Syst. Man Cybern., SMC-7 no. 7 (1977) 505–523.

[Whi33] H. Whitney, 2-*isomorphic graphs*, Amer. J. Math. 55 (1933), 245–254.

[WM99] V. Waddle, A. Malhotra *An E log E line crossing algorithm for levelled graphs* in J. Kratochvíl (Ed.) Graph Drawing 1999, Lecture Notes in Computer Science, Vol. 1731 (1999) 59–70

[WS79] C. Wetherell, A. Shannon *Tidy drawings of trees*, IEEE Trans. Software Engineering 5 (1979) 514–520.