# Konflikte vermeiden. Planarisierungsalgorithmen für Graphen im Praxistest.

Petra Scheffler

FH Stralsund, FB Wirtschaft
Zur Schwedenschanze 15
D-18435 Stralsund

Petra.Scheffler@fh-stralsund.de

**Zusammenfassung:** An der FH Stralsund wird das Software-Framework VinetS* zum interaktiven und automatischen Bearbeiten von Graphen und Netzwerken entwickelt. In diesem Beitrag werden die bei der Implementierung verschiedener Layoutalgorithmen gewonnenen Erfahrungen dargestellt. Es werden insbesondere Algorithmen zum Erzeugen kreuzungsfreier Zeichnungen betrachtet, wobei sowohl auf das Finden kombinatorischer Einbettungen als auch auf die anschließende Phase der geometrischen Einbettung eingegangen wird. Auch softwaretechnische Fragen werden behandelt.

## 1. Einleitung

Mit zunehmendem Computer- und Multimediaeinsatz nimmt in der Wirtschaft und in der gesamten modernen Gesellschaft die Notwendigkeit, Informationen überzeugend zu präsentieren, einen immer größeren Stellenwert ein. Hierbei spielen abstrakte Grafiken eine große Rolle. Der Mensch kann komplexe Beziehungen wesentlich schneller über schematische Darstellungen erfassen, als über verbale Beschreibungen. Bilder ermöglichen ihm einen besseren Überblick über Hierarchien, Systemstrukturen und Zusammenhänge.

In vielen Anwendungsbereichen werden Zusammenhänge (Beziehungen, Relationen) in natürlicher Weise als Graphen modelliert. Mathematisch betrachtet werden reale oder begriffliche Objekte zu Knoten und ihre Verbindungen zu Kanten eines Graphen. Dieses Modell ermöglicht eine theoretisch fundierte Analyse der Eigenschaften einer gegebenen Struktur, liefert aber auch eine gute Grundlage für ihre bildliche Darstellung. Klassische Beispiele für grafische Schemata sind technische Dokumentationen wie elektrische Schaltpläne, Verlege- und Wartungspläne für Rohrleitungssysteme in der Wasserwirtschaft und der Chemieindustrie, Netzpläne für Kabel, Rohre oder Verkehrswege im Facility-Management, aber auch Block-, Funktions- und Flussdiagramme in der Leittechnik und Diagramme für vernetzte Produktionsabläufe und Organigramme in der Wirtschaft. In all diesen Fällen wird eine möglichst übersichtliche Abbildung der Graphen auf einer zweidimensionalen Zeichenfläche gewünscht, die weitgehend automatisch aus domänenspezifischen Daten erzeugt werden soll.

An der FH Stralsund wird ein modulares Softwareframework für solche Aufgaben des Graph Drawing entwickelt (vgl. [19] und zu ähnlichen Projekten auch [13]). Das Tool VinetS ist plattformunabhängig in Java implementiert. Gegenwärtig bietet es Möglichkeiten zum interaktiven Erzeugen von Graphen und zum Speichern und Laden in dem erweiterbaren GraphML-Format (vgl. hierzu [10]). Die Software ist konfigurierbar und über eine einfache allgemeine Schnittstelle um spezielle Algorithmen erweiterbar. Sie wird bereits in der Lehre zum Vermitteln von Kenntnissen über Graphalgorithmen und deren softwaretechnische Umsetzung eingesetzt. Auch einige Algorithmen zur Analyse von Grapheigenschaften und für das automatische Layout sind realisiert.

Es gibt inzwischen viele Publikationen zu Graph Drawing Problemen, in den Sammelbänden [6] und [15] werden beispielhaft verschiedene Layoutstile und -algorithmen vorgestellt. Wir wollen uns hier auf das wichtige Problem der Planarisierung beschränken. Die Übersichtlichkeit einer schematischen Zeichnung hängt – neben anderen Kriterien – sehr stark von der Anzahl der sichtbaren Konflikte ab. Wünschenswert ist eine Zeichnung mit disjunkten Knoten und geradlinigen Verbindungen, die sich nicht kreuzen. Ein Graph, der ohne Überschneidungen zwischen seinen Kanten in der Ebene dargestellt werden kann, wird planar genannt. Bereits 1930 bewies Kuratowski, dass genau die Graphen planar sind, die sich weder auf den vollständigen Graphen $K_5$ noch auf den vollständigen paaren Graphen $K_{3,3}$ zusammenziehen lassen, s. [16]. In Abbildung 1 werden diese beiden minimalen nicht planaren Graphen gezeigt, die man auch Kuratowski-Graphen nennt.

* Das Projekt VinetS – "Software zur Visualisierung vernetzter Strukturen" wurde 2002/2003 durch das Ministerium für Bildung, Wissenschaft und Kultur des Landes Mecklenburg-Vorpommern mit Mitteln aus dem Hochschulwissenschaftsprogramm der Bundesregierung (HWP) unter dem Kennzeichen TEAM-FH 01 021 50 gefördert. 2003/2004 förderte die FH Stralsund unter der Nummer 13041401 ein weiteres Projekt "Java Programm zum Zeichnen planarer Netzwerke".

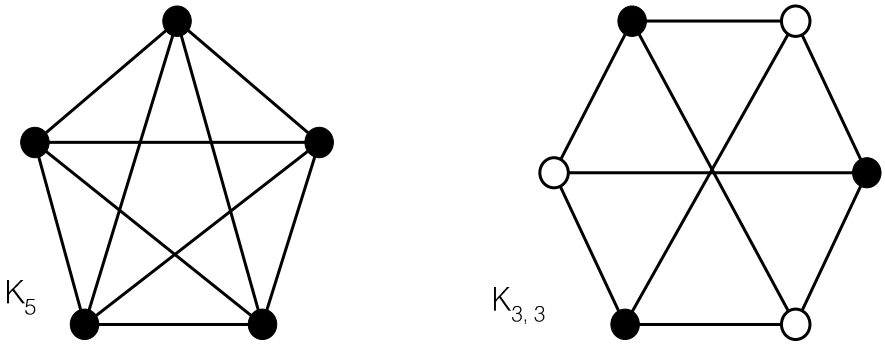



Abbildung 2 Die beiden von Kuratowski beschriebenen nicht planaren Graphen.

Seitdem wurden in der Graphentheorie viele Planaritätskriterien und auch Erkennungsalgorithmen für planare Graphen beschrieben, zur genaueren Orientierung hierüber möchte ich beispielsweise auf die Lehrbücher [9] und [24] sowie auf aktuelle Übersichten wie [17] und [24] verweisen. 1974 publizierten Hopcroft und Tarjan den ersten Linearzeitalgorithmus, der entscheidet, ob ein gegebener Graph planar ist und der ggf. auch eine kreuzungsfreie Darstellung findet. Die praktische Umsetzung dieses theoretisch optimalen Algorithmus stellte jedoch noch lange Zeit eine Herausforderung dar, vgl. [18]. Da das Problem entscheidend für das automatische Zeichnen übersichtlicher Schemata ist, wird bis in die jüngste Zeit nach einfacheren Algorithmen dafür gesucht, s. z.B. [2]. In der Praxis tritt das Problem zudem oft mit Nebenbedingungen auf: Knoten sind vorgegebene Blöcke mit fester Länge und Breite, Kanten müssen zu bestimmten Anschlussstellen führen und die Zeichnung insgesamt muss domänenspezifische Entwurfsregeln berücksichtigen (vgl. [8]).

Im Rahmen des Frameworks VinetS wird daher ein Spezialmodul zum (möglichst) kreuzungsfreien Zeichnen von allgemeinen hierarchischen Netzwerken entwickelt. Als erster Schritt wurde bisher ein solches Layoutmodul für schlichte ungerichtete Graphen implementiert und getestet, wobei aus der Literatur bekannte Algorithmen benutzt, modifiziert und kombiniert wurden. Die dabei gewonnenen Erfahrungen bezüglich der Leistungsfähigkeit und bezüglich der softwaretechnischen Realisierung werden in diesem Artikel dargestellt.

Wie allgemein üblich, gehen auch wir in zwei Phasen vor. Zuerst wird getestet, ob der Eingabegraph planar ist und es wird, wenn das der Fall ist, eine kombinatorische Einbettung bestimmt. Aus dieser wird in der zweiten Phase die genaue Geometrie einer ebenen Zeichnung berechnet. Für beide Phasen sind verschiedene Algorithmen realisiert, die wahlweise ausgeführt werden können. Die Details werden in Kapitel 5 bzw. 6 beschrieben. Unter einer kombinatorischen Einbettung versteht man hierbei die Auflistung der Facetten (der Gebiete der Zeichenfläche mit ihren Rändern aus Knoten und Kanten) bzw. die Festlegung der zyklischen Reihenfolgen aller Kanten um jeden Knoten, die jeweils eine kreuzungsfreie Zeichnung ermöglichen.

Im nächsten Kapitel definieren und erläutern wir zunächst die notwendigen mathematischen Begriffe. Danach gehen wir in Kapitel 3 auf die Datenstrukturen ein, die sich für die Realisierung der Algorithmen gut bewährt haben. Im Kapitel 4 wird die softwaretechnische Realisierung von Graphalgorithmen insbesondere unter dem Blickwinkel der Wiederverwendbarkeit der implementierten Lösungen beleuchtet.

## 2. Planare Graphen – das mathematische Modell

Wir interessieren uns für geradlinige Zeichnungen eines schlichten ungerichteten Graphen $G = (V, E)$ mit der Knotenmenge $V = \{v_1, v_2, \ldots, v_n\}$ und der Kantenmenge $E \subseteq V^2$ in der Ebene. Zur Vereinfachung werden wir nur zweifach zusammenhängende Graphen betrachten, d.h. solche, die keinen Artikulationsknoten haben. Dies stellt keine wesentliche Einschränkung dar, denn das Entfernen eines Artikulationsknotens aus $G$ würde zum Zerfallen des Graphen in mindestens zwei nicht miteinander verbundene Teile führen. In diesem Fall kann man die ebenen Zeichnungen der Teile geeignet zusammenfügen.

Eine geometrische Einbettung eines Graphen $G$ wird durch eine injektive Abbildung $g: V \rightarrow R^2$ bestimmt, die jedem Knoten $v$ des Graphen seine Koordinaten $g(v) = (x_v, y_v)$ in der Ebene zuordnet. Zudem bildet die geometrische Einbettung $g$ jede Kante $e = \{u, v\}$ des Graphen auf die gerade Verbindungsstrecke $[g(u), g(v)]$ zwischen den Bildern ihrer beiden Endpunkte ab. Man spricht von einer ebenen Einbettung, wenn jede Kantendarstellung genau zwei geometrische Knoten enthält (die Abbilder ihrer eigenen Endknoten) und sich je zwei Kantendarstellungen höchstens in dem Bild eines gemeinsamen Endknotens schneiden.

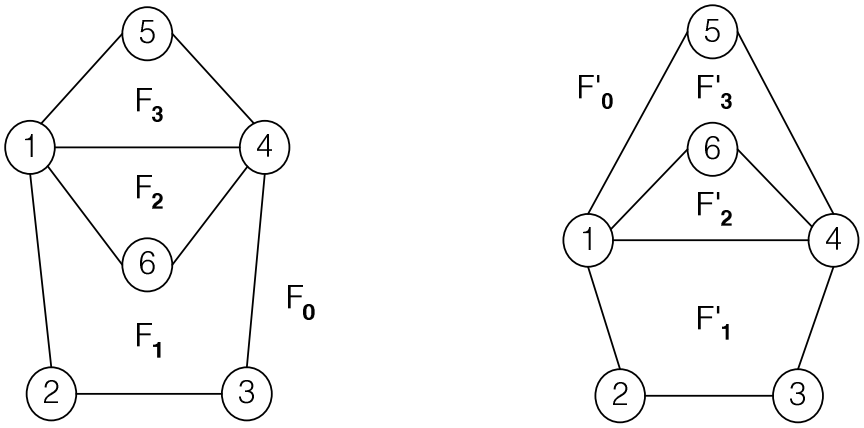



Abbildung 3 Zwei verschiedene ebene Einbettungen für einen Graphen $G$.

Der Graph $G$ hat $n = 6$ Knoten und $m = 8$ Kanten, nach der Eulerschen Polyederformel besitzt daher jede ebene Einbettung $f = m - n + 2 = 4$ Facetten. $G$ ist 2fach, aber nicht 3fach zusammenhängend. Je nach Einbettung treten Kreise unterschiedlicher Länge als Facetten auf. Unterschiede in der Reihenfolge der Nachbarknoten gibt es natürlich nur bei Knoten mit mehr als zwei Nachbarn.

Die Facetten der linken Zeichnung $g$:
$F_0 = (1, 5, 4, 3, 2)$, $F_1 = (1, 2, 3, 4, 6)$,
$F_2 = (1, 6, 4)$ und $F_3 = (1, 4, 5)$,

Die Facetten der rechten Zeichnung $g'$:
$F'_0 = (1, 5, 4, 3, 2)$, $F'_1 = (1, 2, 3, 4)$,
$F'_2 = (1, 4, 6)$ und $F'_3 = (1, 6, 4, 5)$.

Zyklisch geordnete Nachbarschaftslisten in $g$:
$N_1 = (2, 5, 4, 6)$ und $N_4 = (1, 5, 3, 6)$

Zyklisch geordnete Nachbarschaftslisten in $g'$:
$N'_1 = (2, 5, 6, 4)$ und $N'_4 = (1, 6, 5, 3)$

Ein Graph heißt **planar** (auch **plättbar**), wenn er wenigstens eine ebene Einbettung besitzt. In Abbildung 2 werden zwei verschiedene ebene Einbettungen eines planaren Graphen gezeigt. Sie sind auch topologisch nicht äquivalent, d.h. sie können nicht durch stetiges Verschieben der Knotenabbilder ineinander transformiert werden.

Jede ebene Einbettung $g$ eines planaren zweifach zusammenhängenden Graphen $G$ zerschneidet die Ebene in zusammenhängende **Gebiete**, deren Rand jeweils von den Darstellungen der Knoten und Kanten eines **Kreises** von $G$ gebildet wird. Genau eines dieser Gebiete ist unbegrenzt. Man nennt diejenigen Kreise von $G$, die auf den Rand eines leeren Gebietes abgebildet werden, die **Facetten** der gegebenen Einbettung. Dabei bildet die **äußere Facette** $F_0$ den Rand des unbegrenzten Gebietes. Die anderen Facetten treten in der Zeichnung als Rand eines im Inneren leeren Polygons auf, wobei konvexe Polygone besonders anschaulich sind. Die Einbettungen in Abbildung 2 enthalten jeweils ein nicht konvexes inneres Gebiet ($F_1$ bzw. $F'_3$).

Das **Planarisierungsproblem** besteht nun darin, für einen gegebenen Graphen $G$ eine ebene Einbettung zu finden, falls er planar ist. Im ersten Lösungsschritt wird dabei stets entschieden, welche Kreise von $G$ als Facetten infrage kommen. Danach werden diese geeignet in die Ebene abgebildet. Üblicherweise betrachtet man zwei ebene Einbettungen eines Graphen mit derselben Facettenmenge als äquivalent und vernachlässigt dabei auch die Orientierung und die Auswahl der äußeren Facette. Jede so gebildete Äquivalenzklasse ebener Einbettungen bezeichnet man als eine **kombinatorische Einbettung** von $G$.

Jede kombinatorische Einbettung $\tilde{G}$ wird eindeutig durch die Menge ihrer Facetten repräsentiert. Eine Facette $F$ ist dabei ein Kreis des ungerichteten Graphen, also ein zusammenhängender Teilgraph, in dem alle Knoten genau zwei Nachbarn haben. Soll eine kombinatorische Einbettung auch geometrisch realisiert werden, muss zunächst die Orientierung der Ebene gewählt und dann eine der Facetten als äußere ausgezeichnet werden. Jede Facette $F$ wird dabei zu einer Menge von Knoten $w_i$ und Kanten $e_i$ des Graphen mit einer bestimmten zyklischen Reihenfolge $F = (w_0, e_1, w_1, e_2, \ldots, w_{k-1}, e_k)$, wobei der erste Knoten beliebig gewählt werden kann. Es muss $e_i = \{w_{i-1}, w_{i(\mathrm{mod})k}\}$ für alle $i = 1, \ldots, k$ gelten. Wir werden für Facetten auch die Kurzform $F = (w_0, w_1, \ldots, w_{k-1})$ benutzen, da in einem schlichten Graphen alle Kanten durch ihre beiden Endknoten eindeutig bestimmt sind. Normalerweise werden wir den Knoten mit der kleinsten Nummer als ersten aufführen. Als Beispiel für diese Definitionen sind in Abbildung 2 zwei kombinatorische Einbettungen eines Graphen angegeben. Für die Orientierung der Elemente jeder Facette wurde dabei der Gegenuhrzeigersinn in der Zeichnung gewählt: Diese Reihenfolge ergibt sich, wenn man bei einem Spaziergang auf dem Rand eines Gebietes dessen Inneres stets

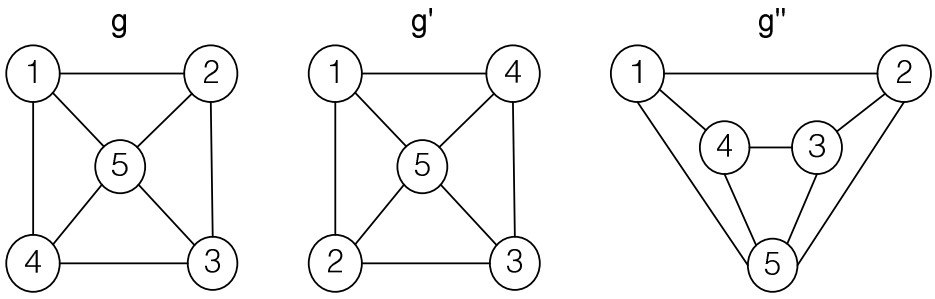



**Abbildung 4** Verschiedene geometrische Realisierungen der einzigen kombinatorischen Einbettung für einen dreifach zusammenhängenden Graphen *G*.

Die kombinatorische Einbettung dieses Graphen ist charakterisiert durch die Facettenmenge *{(1, 2, 3, 4), (1, 4, 5), (1, 5, 2), (2, 5, 3), (3, 5, 4)}* bzw. durch die geordneten Nachbarschaftslisten *(2, 5, 4), (1, 3, 5), (2, 4, 5), (1, 5, 3)* und *(1, 2, 3, 4)*.
Für die Realisierung *g* wurde dabei der erste Kreis als äußere Facette gewählt und die Orientierung im Gegenuhrzeigersinn. Bei der geometrischen Einbettung *g'* wurden alle Facetten umgekehrt orientiert, für *g"* wurde zudem eine andere äußere Facette ausgewählt. Die zyklischen Nachbarschaftslisten haben entsprechend auch eine umgekehrte Orientierung.

linker Hand sieht. Auf diese Weise ist jede Kante des Graphen in zwei benachbarten Facetten enthalten, genau einmal in jeder Durchlaufrichtung.

Kombinatorische Einbettungen können auch auf eine andere Weise eindeutig charakterisiert werden: Zwei ebene Einbettungen eines zweifach zusammenhängenden Graphen sind genau dann topologisch äquivalent, wenn für alle Knoten die zyklische Reihenfolge der inzidenten Kanten gleich ist. In Abbildung 2 sind auch die entsprechenden zyklisch (im Uhrzeigersinn) geordneten Nachbarschaftslisten $N_v = (w_1, w_2, \ldots, w_d)$ für die beiden Einbettungen angegeben. In jedem schlichten Graphen ist diese Schreibweise äquivalent zur zyklisch geordneten Inzidenzliste der Kanten $I_v = (\{v, w_1\}, \{v, w_2\}, \ldots, \{v, w_d\})$.

Jeder dreifach zusammenhängende planare Graph besitzt nur eine kombinatorische Einbettung. Durch Auswählen einer anderen äußeren Facette und durch Spiegeln kann man trotzdem unterschiedliche Zeichnungen erhalten, wie das Beispiel in Abbildung 3 demonstriert. Es ist die Aufgabe der zweiten Phase bei der Planarisierung, eine übersichtliche geometrische Realisierung für eine gegebene kombinatorische Einbettung zu bestimmen. Bevor wir in den Kapiteln 5 und 6 einige Lösungsmöglichkeiten diskutieren, sollen nun diese beiden algorithmischen Probleme, die beim Finden einer kreuzungsfreien Zeichnung eines Graphen normalerweise auftreten, explizit formuliert werden.

**Kombinatorische Einbettung**

Input: Ein schlichter, ungerichteter, zweifach zusammenhängender Graph $G = (V, E)$.

Output: Eine kombinatorische Einbettung $\tilde{G}$ des Graphen dargestellt in Form der Menge ihrer Facetten oder einer Beschreibung der zyklisch geordneten Nachbarschaftslisten für alle Knoten bzw. die (wahre) Aussage, dass *G* nicht planar ist.

**Geometrische Einbettung**

Input: Eine Darstellung einer kombinatorischen Einbettung $\tilde{G}$ eines Graphen *G*.

Output: Eine ebene geometrische Einbettung *g* von *G* in Form der Koordinaten $g(v) = (x_v, y_v)$ für alle Knoten, die die gegebene Einbettung $\tilde{G}$ kreuzungsfrei realisiert.

Publizierte Algorithmen für beide Teilprobleme benutzen zur Beschreibung einer kombinatorischen Einbettung entweder die eine oder die andere oben erwähnte Darstellung. Unsere Datenstrukturen stellen für jeden planaren Graphen beide Beschreibungen seiner kombinatorischen Einbettung zur Verfügung.

## 3. Datenstrukturen für planare Graphen und ihre Einbettungen

Bei allen Planarisierungsalgorithmen im System VinetS wird zunächst geprüft, ob der Graph überhaupt planar ist, und ggf. eine kombinatorische Einbettung konstruiert. Bei Erfolg wird dann in einer zweiten Phase die genaue Geometrie berechnet. Als Eingabe dient dabei ein abstrakter Graph, von dem nur die Struktur bekannt ist. Als Ausgabe stehen die vollständige kombinatorische Beschreibung einer ebenen Einbettung des Graphen und ihrer Geometrie, die im Wesentlichen aus den Koordinaten der Knoten besteht, zur Verfügung. Hier soll skizziert werden, welche Datenstrukturen für die kombinatorische Einbettung planarer Graphen sich dabei im System VinetS bewährt haben.* Insbesondere wird auch gezeigt, wie die beiden Darstellungen der Einbettung als Facettenmenge bzw. als geordnete Inzidenzlisten ineinander überführt werden können.

Das Softwareframework VinetS stellt für abstrakte Graphen und ihre Elemente die Datentypen **`Graph`**, **`Node`** und **`Edge`** zur Verfügung. Jedes `Graph`-Objekt verwaltet die Mengen seiner Knoten und Kanten, jedes `Node`-Objekt enthält eine Liste seiner inzidenten Kanten und jedes `Edge`-Objekt hält Referenzen auf seine beiden Endknoten. Entsprechende Zugriffsmethoden und Iteratoren über die Elemente werden jeweils bereitgestellt. Daneben verwalten alle Graphelemente weitere Attribute, beispielsweise Referenzen auf ihre geometrische Darstellung. Für planare Graphen werden die Graphelemente mit zusätzlichen Informationen dekoriert – das Erzeugen dieser Dekoration ist die Aufgabe der ersten Phase jedes Planarisierungsalgorithmus.

Wenn der Planaritätstest für eine Eingabe eine positive Antwort liefert, dann wird der Eingabegraph in ein **`PlanarGraph`**-Objekt gekapselt. Dieses hält neben Knoten und Kanten auch die Menge der Facetten einer kombinatorischen Einbettung. Für die Verwaltung der Knoten-, Kanten und Facettenmengen eines Graphen wird normalerweise die Klasse `java.util.HashSet` verwendet. Als Datenstruktur für die Facetten wird eine Klasse **`Face`** benutzt, die intern eine zyklische Kantenliste verwaltet.

Wichtig für die Laufzeit der Algorithmen ist, dass auf die Listenelemente direkt in konstanter Zeit zugegriffen werden kann. Java stellt solche Listen nicht standardmäßig bereit, deshalb wurde eine eigene Datenstruktur **`CyclicList`** implementiert. Sie enthält die üblichen Methoden zum Einfügen und Löschen, wobei als Parameter stets auch eine **`Position`** übergeben werden kann. Der dabei verwendete interne Datentyp `Position` fasst ein Graphelement mit Informationen über seine Verkettung in der zyklischen Liste zusammen. Die `CyclicList`-Methode `add(GraphElement)` liefert stets die Einfügeposition zurück, so dass diese auch extern zugänglich wird. Es steht auch eine Methode `getNext(Position)` zur Verfügung, die in konstanter Zeit eine Referenz auf das ausgehend von der übergebenen `Position` nächste Listenelement zurückliefert. Darüber hinaus ist die Liste typsicher. Dieselbe `CyclicList`-Klasse wird für die zyklisch geordneten Inzidenzlisten bei den Knoten eines planar eingebetteten Graphen genutzt.

Bei der Berechnung einer kombinatorischen Einbettung eines planaren Graphen erhält jedes `Node`-Objekt als Dekoration eine zyklisch geordnete Liste seiner inzidenten Kanten. Jedes `Edge`-Objekt kennt dann neben seinen beiden Endknoten `Node1` und `Node2` auch die eigene Position `Pos1` und `Pos2` in den entsprechenden zyklisch geordneten Inzidenzlisten von `Node1` bzw. `Node2` sowie die beiden Facetten, zu denen die Kante gehört. Hierbei wird je eine Referenz auf die `Position` der Kante in dem entsprechenden `Face`-Objekt gehalten, wobei das Attribut `Face1` auf die Facette verweist, die der Durchlaufrichtung (`Node1`, `Node2`) entspricht, während `Face2` zur umgekehrten Durchlaufrichtung (`Node2`, `Node1`) gehört. Für die linke Einbettung $g$ in Abbildung 2 wird beispielsweise die Kante $e = \{1, 4\}$ mit den Attributen `Node1` = *1*, `Node2` = *4*, `Pos1` = `Position`$(e$ in $l_1)$ und `Pos2` = `Position`$(e$ in $l_4)$ sowie `Face1` = `Position`$(e$ in $F_3)$ und `Face2` = `Position`$(e$ in $F_2)$ gespeichert, wobei die rechten Seiten der Gleichungen jeweils Referenzen auf die entsprechenden Objekte symbolisieren sollen.

Wie kann diese Dekoration nun aus einer kombinatorischen Einbettung gewonnen werden? Üblicherweise liefert ein Algorithmus entweder die Facettenmenge oder die zyklisch geordneten Inzidenzlisten der Knoten. Auf der nächsten Seite sind die entsprechenden Programme zum Erzeugen unserer Datenstruktur für beide Fälle angegeben. Sie gehen von einer vorliegenden kombinatorischen Einbettung des Graphen mit einer bestimmten Orientierung aus. Beide Algorithmen arbeiten in Linearzeit, da insgesamt für jede Kante und jeden Knoten des Graphen nur konstant viele Operationen ausgeführt werden müssen.

---

* Tatsächlich werden im Programm teilweise noch spezielle, auf die einzelnen Algorithmen zugeschnittene Datenstrukturen verwendet - hier beschrieben sind die aus den Erfahrungen gewonnenen Schlussfolgerungen darüber, wie allgemeine, effiziente und leistungsfähige Schnittstellen für planare Graphen implementiert werden sollten.

**Algorithmus1: Umwandeln einer Facettenmenge in zyklisch geordnete Inzidenzlisten**

Input: Ein eben eingebetteter Graph $G = (V, E)$ mit seiner Facettenmenge $M = \{F_1, \ldots, F_f\}$.

Output: Eine Dekoration von $G$ mit entsprechend zyklisch geordneten Inzidenzlisten $I_v$ für alle Knoten $v \in V$ und für alle Kanten $e \in E$ mit Referenzen auf ihre Positionen darin sowie in den Facettenlisten.

Ablauf:

```
Für alle Facetten F ∈ M do: {
   Für alle Kanten e ∈ F do: Setze die Referenz e.Face1 bzw. e.Face2
} Für alle Knoten v ∈ V do: {
   Erzeuge eine zyklische Inzidenzliste v.Inz
   Wähle eine inzidente Kante e = {v, w}
   Wiederhole deg(v) mal (für alle zu v inzidenten Kanten e): {
      if (v == e.Node1) {
         e.Pos1 = v.Inz.add(e); e = getNext(e.Face2)
      } else {
         e.Pos2 = v.Inz.add(e); e = getNext(e.Face1)
      }
   }
}
```

**Algorithmus2: Umwandeln der zyklisch geordneten Inzidenzlisten in eine Facettenmenge**

Input: Ein ebener Graph $G = (V, E)$ mit zyklisch geordneten Inzidenzlisten $I_v$ für alle $v \in V$.

Output: Die entsprechende Facettenmenge $M = \{F_1, \ldots, F_f\}$ von $G$ und eine Dekoration für alle Kanten $e \in E$ mit Referenzen auf ihre Position in den Facetten- und Inzidenzlisten .

Ablauf:

```
Für alle Knoten v ∈ V do: {
   Für alle Kanten e ∈ v.Inz do: Setze Referenz e.Pos1 bzw. e.Pos2
} Für alle Kanten e ∈ E do: {
   if (e.Face1 == null) {
      Erzeuge eine neue Facette F, merke e_start = e
      e.Face1 = F.add(e); v = e.Node2; e = getNext(e.Pos2)
      Wiederhole solange (e != e_start) : {
         if (v == e.Node1) {
            e.Face1 = F.add(e); v = e.Node2; e = getNext(e.Pos2)
         } else {
            e.Face2 = F.add(e); v = e.Node1; e = getNext(e.Pos1)
         }
      }
   } if (e.Face2 == null) {
      Erzeuge eine neue Facette F, merke e_start = e
      e.Face2 = F.add(e); v = e.Node1; e = getNext(e.Pos1)
      Wiederhole solange (e != e_start) : {
         if (v == e.Node1) {
            e.Face1 = F.add(e); v = e.Node2; e = getNext(e.Pos2)
         } else {
            e.Face2 = F.add(e); v = e.Node1; e = getNext(e.Pos1)
         }
      }
   }
}
```

## 4. Eine flexibel nutzbare Implementierung der Tiefensuche

Viele Graphalgorithmen beruhen auf Erweiterungen von Standarddurchlauftechniken wie Breitensuche oder Tiefensuche, mit deren Hilfe alle Knoten und Kanten eines Graphen in wohl definierter Reihenfolge bearbeitet werden. Bei der Planarisierung erweist sich insbesondere die Tiefensuche (depth first search = DFS) an verschiedenen Stellen als nützlich. Aus diesem Grund wurde im Framework VinetS ein DFS-Algorithmus implementiert, der das nachträgliche Zufügen von Funktionalität unterstützt und dessen Ablauf auch von Außen in Abhängigkeit von der vorgefundenen Graphstruktur steuerbar ist. Dies wurde mit einem ereignisbasierten Modell erreicht, das wir aus dem Observer-Pattern abgeleitet haben. Für Hintergrundinformationen zu diesem und anderen Entwurfsmustern bei der Softwareentwicklung verweisen wir auf das Standardwerk [7].

Unsere Erfahrungen belegen, dass der gewählte Lösungsansatz sehr gut die Wiederverwendung sorgfältig implementierter Programmmodule in verschiedenen Kontexten ermöglicht und somit zur Einsparung von Entwicklerzeit und zu robuster Software mit hoher Qualität beiträgt. Unser Entwurfsmuster ist auch auf Implementierungen anderer Standardalgorithmen übertragbar und soll deshalb hier näher erläutert werden.

In unserem Modell wird der Basisalgorithmus als Auslöser von Ereignissen angesehen. Jeder logische Schritt im Ablauf des Algorithmus wird als Ereignis interpretiert, über das ein oder auch mehrere Nutzer des Algorithmus informiert werden. Es gibt in dem Modell die folgenden Beteiligten (vgl. auch Abbildung 4):

- einen Ereignisauslöser (den Basisalgorithmus – hier `DFS`)
- eine Menge von Ereignissen (die Schritte des Algorithmus)
- einen oder mehrere Ereignisempfänger (spezielle Anwendungsalgorithmen – hier `PathFinder`, `LargeCycleFinder` und `LowPointCalculator`)
- einen Proxy zur Verwaltung der Kommunikation zwischen dem Algorithmus und seinen Empfängern

Der Basisalgorithmus ist in der `execute`-Methode der `DFS`-Klasse implementiert. Zusätzlich besitzt jedes `DFS`-Objekt zwei Methoden zum An- bzw. Abmelden der Ereignisempfänger. Diese sind auf eine gemeinsame Schnittstelle `DFSListener` typisiert.

Die Kommunikation zwischen dem Basisalgorithmus und seinen Empfängern wird in ein Objekt der Klasse `MulticastProxy` ausgelagert. Diese von der Klasse `java.lang.reflect.Proxy` abgeleitete Klasse ist nur einmal für alle Algorithmen implementiert und kann immer in gleicher Weise benutzt werden. Ein Basisalgorithmus reicht die Aufrufe der Methoden `addListener` bzw. `removeListener` stets an einen `MulticastProxy` weiter, umgekehrt werden die Methodenaufrufe des Algorithmus auf der Ereignisschnittstelle durch den `MulticastProxy` an die registrierten Empfänger verteilt. Dieser Mechanismus ist für die Empfänger nicht sichtbar.

Die Empfänger-Schnittstelle `DFSListener` definiert für jeden potentiell interessanten Schritt des Basisalgorithmus eine CallBack-Methode, deren Signatur auch Parameter enthalten kann, über die der Empfänger dann zusätzliche mit dem Ereignis verbundene Informationen erhält. Für den `DFS`-Algorithmus sind folgende Hauptschritte als Ereignisse relevant:

- Beginn des Algorithmus: `startDFS()`
- Beginn einer neuen Zusammenhangskomponente: `newComponent(Node root)`
- Entdecken eines neuen Knotens: `nodeDiscovered(Node node, Node parent, int step)`
- Bearbeiten einer Kante: `edgeFound(Edge edge, Node from, Object edgetype)`
- Abschluss eines Knotens: `nodeFinished(Node node, int step)`
- Ende des Algorithmus: `endDFS()`

Jede Klasse, die als Empfänger auf den `DFS`-Algorithmus zugreift, muss diese Methoden implementieren. Als Alternative wurde – nach dem Adapter-Entwurfsmuster – eine Klasse mit leeren Implementierungen dieser Methoden bereitgestellt. In einer Empfängerklasse, die von dieser Klasse `DFSAdapter` erbt, müssen nur die Methoden überschrieben werden, die zu bestimmten interessierenden Ereignissen gehören.

Als spezielle Algorithmen wurden im System VinetS beispielsweise die Erweiterungen der Tiefensuche zum Finden der zweifach zusammenhängenden Blöcke eines Graphen in der Klasse `LowPointCalculator` und eine Heuristik zum Finden eines möglichst langen Kreises im Graphen in der Klasse `LargeCycleFinder` als Empfänger realisiert. Diese werden wie auch die Klasse `PathFinder` für den DMP-Algorithmus (vgl. Kapitel 5) benötigt. In der Abbildung 4 ist das entsprechende UML-Schema aller am Entwurfsmuster beteiligten Klassen angegeben.

Das beschriebene Entwurfsmuster Ereignisbasierter Algorithmus ermöglicht es sehr einfach und flexibel, die Funktionalität des einmal implementierten `DFS`-Algorithmus zu erweitern. Die Callback-Methoden enthalten zusätzlichen Code, der dann praktisch zwischen den Schritten des Basisalgorithmus ausgeführt wird. Der `LowPointCalcuator` benutzt intern einen `DFS` und meldet sich bei ihm als Ereignisempfänger an. Er markiert im

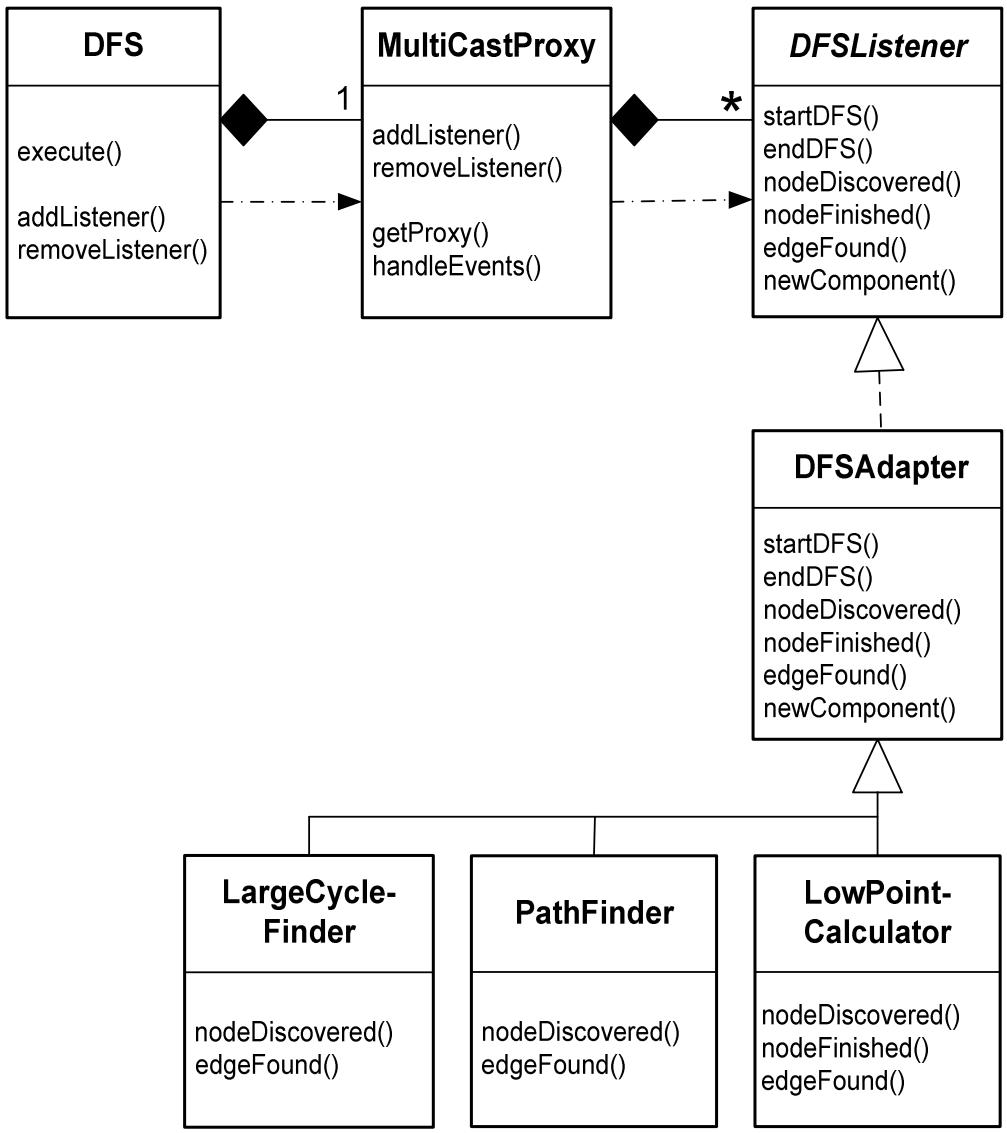



Abbildung 1 Das Entwurfsmuster Ereignisbasierter Algorithmus am Beispiel der Tiefensuche

Verlauf der Tiefensuche bearbeitete Knoten und Kanten mit für ihn wichtigen Informationen. Wird vom `DFS` signalisiert, dass die Bearbeitung eines Knotens abgeschlossen ist, kann der Lowpoint des Knotens von der Callback-Methode `nodeFinished` bestimmt werden. Weitere Empfänger können durchaus gleichzeitig bei dem `DFS`-Objekt angemeldet sein und unabhängig voneinander agieren. Darüber hinaus fungiert die `LowPointCalculator`-Klasse ihrerseits wieder als Basisalgorithmus im Programm, wobei hier der Planaritätstest von Boyer als Empfänger-Erweiterung implementiert ist.

Ein weiterer Vorteil unseres Herangehens ist, dass jeder der beteiligten Algorithmen nur die Informationen verwalten muss, die ihn speziell interessieren – die eigentlichen Datenstrukturen des Graphen müssen weder kopiert noch konvertiert werden. Zusätzliche algorithmusspezifische Daten können dabei auch lokal im Graph als Label bei einem `Node`- oder `Edge`-Objekt gespeichert werden. Auf diese Weise können auch Informationen für nachfolgende Algorithmus-Phasen bereitgestellt werden.

## 5. Planaritätstest und Finden einer kombinatorischen Einbettung

Im System VinetS haben alle Planarisierungsalgorithmen einen ungerichteten zusammenhängenden schlichten Graphen als Eingabe. Für gerichtete Graphen gibt es zusätzlich eine Konvertierungsmethode, die den Eingabegraphen so kapselt, dass jeder Layoutalgorithmus auf dem zugrunde liegenden schlichten ungerichteten Graphen ausgeführt werden kann. Wenn ein Graph nicht zusammenhängend ist, müssen seine Zusammenhangskomponenten einzeln eingebettet werden. Danach ist ein Platzierungsproblem zu lösen, bei dem eine optimale

Anordnung der entstehenden Teilzeichnungen nebeneinander auf der Fläche gesucht wird. Darauf wird hier nicht eingegangen.

Wir gehen im Weiteren also immer von zusammenhängenden Graphen aus. Nach der Eulerschen Polyederformel kann ein planarer Graph mit $n$ Knoten höchstens $3n - 6$ Kanten enthalten. Diese einfache notwendige Planaritätsbedingung wird stets im ersten Schritt getestet. Nur, wenn sie erfüllt ist, werden nacheinander die beiden Phasen der Planarisierung gestartet.

Für das Problem Kombinatorische Einbettung wurden zwei Algorithmen aus der Literatur implementiert. Beide Varianten haben sich in der Praxis bewährt und liefern für Graphen moderater Größe sehr schnell korrekte Resultate. Im Folgenden sollen die Algorithmen, Probleme bei ihrer Realisierung und die Vor- und Nachteile genauer dargestellt werden.

### Der Algorithmus von Demoucron, Malgrange und Pertuiset (DMP-Algorithmus)

Dieser klassische Algorithmus [5] zum Test der Planarität eines zweifach zusammenhängenden Graphen erreicht bei geschickter Implementierung bestenfalls eine quadratische Zeitkomplexität, was sich jedoch bei allen interaktiv erzeugten Graphen als ausreichend schnell erwies. Zudem scheint er besonders gut für eine mögliche Erweiterung auf allgemeine hierarchische Netzwerke geeignet zu sein. Für planare Eingabegraphen liefert der DMP-Algorithmus die Facetten einer kombinatorischen Einbettung zurück. Andernfalls wird eine partielle Einbettung für einen planaren Teilgraphen geliefert. Die Grundidee des Algorithmus ist relativ einfach, er wird auch in vielen Lehrbüchern (z.B. in [9] und [24]) beschrieben. Dort werden jedoch normalerweise keine Aussagen über die Implementierung und die benötigten Datenstrukturen getroffen, deshalb soll hier besonders darauf eingegangen werden, wie die Details im System VinetS gestaltet und modifiziert wurden. Bereits der folgende abstrakte Pseudocode stellt eine Vereinfachung des üblichen Algorithmus dar, bei welchem in jedem Schritt alle Fragmente vollständig berechnet werden.

**DMP–Algorithmus**

Input: Ein zweifach zusammenhängender ungerichteter Graph $G = (V, E)$ mit $m \leq 3n - 6$.

Output: Eine kombinatorische Einbettung $\tilde{G}$ des Graphen dargestellt als Menge $M$ von $f = m - n + 2$ Facetten bzw. die (wahre) Aussage, dass $G$ nicht planar ist.

Ablauf:

```
(1)  Finde einen möglichst langen Kreis C in G und bette ihn ein.
(2)  Finde einen Pfad P in G \ C und bette ihn ein.
(3)  Wiederhole für i = 3, ..., f – 1: {
(4)     Wiederhole bis Erfolg oder G \ Gi vollständig durchsucht {
(5)        Finde ein Fragment B in G \ Gi mit seinen zulässigen Facetten
(6)        if (B ist in keine Facette einbettbar) stop (G ist nicht planar)
(7)        if (B ist nur in eine Facette einbettbar) Erfolg; bette P ein
(8)     } if (kein Erfolg) bette P aus erstem in (4) gefundenen B ein
     }
```

Für das Finden eines langen Kreises im Schritt (1) wird die in Kapitel 4 erwähnte Modifikation der Tiefensuche verwendet. Beim Einbetten des Kreises $C$ werden zwei `Face`-Objekte erzeugt, die jeweils alle Kanten von $C$ aber mit entgegengesetzter Orientierung enthalten. Für jeden Knoten $v$ von $C$ werden die beiden Facetten in die Menge seiner angrenzenden Facetten aufgenommen. Zudem wird bei $v$ je eine Referenz auf die Position der Kante in der Facette gespeichert, die in diesen Knoten $v$ hineinführt.

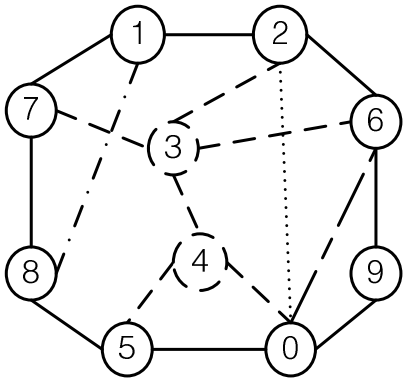
Schritt (2) benutzt die ebenfalls in Kapitel 4 erwähnte `PathFinder`-Klasse. Dabei wird eine Tiefensuche in einem Kontaktknoten $v$ des Kreises $C$ gestartet und solange ausgeführt, bis ein zweiter Kontaktknoten $w$ auf $C$ getroffen wird. Der Pfad $P = w, ..., v$ kann dann im DFS-Baum zurückverfolgt werden. Ein Kontaktknoten ist ein solcher Knoten, der noch nicht eingebettete inzidente Kanten enthält, jedoch selbst schon eingebettet ist. Die Menge der Kontaktknoten wird im System VinetS in einer speziell entwickelten Datenstruktur (genannt `ClusterBucketSet`) gehalten, die sicherstellt, dass ein Iterator die Elemente stets in aufsteigender Reihenfolge bezüglich der Anzahl der angrenzenden Facetten liefert. Dies führt potentiell dazu, dass die Schleife (4) schneller abbricht als bei einer zufälligen Reihenfolge. Der in Schritt (2) gefundene Pfad kann immer in beide Facetten eingebettet werden, es wird beliebig die Facette $F$ gewählt.

Beim Einbetten eines Pfades *P* wird die entsprechende Facette *F* an den Kontaktknoten *v* und *w* aufgeschnitten und zwei neue Facetten $F_1$ und $F_2$ werden aus den Teilen zusammen mit dem Pfad *P* in der jeweils richtigen Orientierung gebildet. Das Aufschneiden einer Facette ist dank der direkten Zugriffsmöglichkeit auf die Position der Knoten leicht möglich. Für jeden Knoten von *P* werden die beiden Facetten $F_1$ und $F_2$ in die Menge seiner angrenzenden Facetten aufgenommen und Referenzen auf seine Position darin gespeichert, bei allen Kontaktknoten aus der alten Facette *F* muss diese aus der angrenzenden Menge entfernt und entsprechend durch die neue Facette $F_1$ oder $F_2$ ersetzt werden.

Nach Schritt (2) wird der bereits eingebettete Teilgraph mit $G_3$ bezeichnet, er hat 3 Facetten. Bei jedem Durchlauf durch die Schleife (3) wird dann durch das Einbetten eines weiteren Pfades die Anzahl *i* der schon eingebetteten Facetten um eins erhöht. Dabei müssen Pfade aus Fragmenten, die nur noch eine zulässige Facette besitzen, bevorzugt werden. Ein Fragment in dem noch nicht eingebetteten Teilgraph wird mit einer Tiefensuche bestimmt, die stets beim Erreichen eines Kontaktknotens abbricht. Zulässig ist eine Facette für ein Fragment genau dann, wenn sie an alle seine Kontaktknoten angrenzt. Der Aufwand für das Bestimmen dieser Schnittmenge wird durch die Reihenfolge der Kontaktknoten im `ClusterBucketSet` ebenfalls minimiert, denn gestartet wird ein Fragment immer von einem Kontaktknoten mit nur zwei angrenzenden Facetten.

Abbildung 5 Fragmente beim DMP-Algorithmus

Vom Algorithmus könnte beispielsweise der Kreis *C = (1, 2, 6, 9, 0, 5, 8, 7)* als erster eingebettet werden. Bis auf Knoten *9* sind alle Knoten von *C* Kontaktknoten. Es gibt vier Fragmente, die im Bild mit unterschiedlichen Linien gekennzeichnet sind, davon bestehen drei nur aus je einer Kante. Im zweiten Schritt könnte eventuell der Pfad *P = (2, 3, 4, 5)* in die innere Facette des Kreises *C* eingebettet werden. Danach gibt es für drei neue Fragmente nur noch je eine zulässige Facette: Für *{3, 7}*, *{3, 6}* und *{4, 0}*. Diese Kanten werden also der Reihe nach eingebettet. Dann werden die Fragmente *{1, 8}* und *{2, 4}* in die nunmehr für sie einzig zulässige äußere Facette von *C* eingebettet. Zuletzt wird die Kante *{6, 0}* in eine ihrer beiden zulässigen Facetten gezeichnet.
Der abgebildete Graph ist also planar.




Es ist nicht nötig, immer wieder alle Fragmente vollständig zu berechnen: die Schleife (4) kann sofort beendet werden, sowie einer der Tests in Zeile (6) oder (7) positiv ausfällt. In Schritt (7) bzw. (8) kann ein beliebiger Pfad *P* aus *B* zum Einbetten gewählt werden, in (8) auch eine beliebige der zulässigen Facetten. Der Algorithmus findet garantiert eine ebene Einbettung, es sei denn der Eingabegraph war nicht planar. In Abbildung 5 wird der Ablauf des DMP-Algorithmus an einem Beispiel demonstriert.

### Der Algorithmus von Boyer

Dieser zweite Algorithmus besitzt lineare Zeitkomplexität, daher ist er besonders bei großen Graphen effizient. Vorteilhaft ist, dass er keine Anforderungen an die Zusammenhangseigenschaften des Eingabegraphen stellt. Man kann also bezüglich seiner Leistungsfähigkeit von einem optimalen Algorithmus sprechen. Dies wird allerdings durch komplizierte Datenstrukturen und einen trickreichen Ablauf erkauft, was hohe Ansprüche an den Programmierer stellt. Zudem wies die publizierte Beschreibung des Algorithmus (s. [1] und [2]) einige Lücken auf, die von A.-M. Törsel während der Arbeit an seiner Diplomarbeit [20] geschlossen wurden. Mit der jetzt vorliegenden detaillierten Dokumentation lässt sich die Implementierung für den Algorithmus von Boyer durchaus auch in einem anderen Kontext nachvollziehen (vgl. auch [21]), auf jeden Fall leichter als etwa für den klassischen Linearzeit-Algorithmus von Tarjan (vgl. [12] und [18]).

An dieser Stelle kann nur die Idee des Algorithmus von Boyer angedeutet werden: Wie Tarjans Methode arbeitet auch dieser Algorithmus auf einem durch Tiefensuche erzeugten aufspannenden Baum des Eingabegraphen. Die kombinatorische Einbettung wird schrittweise durch Zufügen von Kanten zu einem schon eingebetteten Teilgraphen bestimmt. Zuerst werden die Baumkanten der Tiefensuche einzeln als zweifach zusammenhängende Komponenten eingebettet. Alle inneren Knoten eines Tiefensuchbaums sind somit zunächst Artikulationsknoten im eingebetteten Teilgraphen. Dann werden die Rückwärtskanten des Graphen nacheinander entlang der aktuellen äußeren Facette eingebettet. Dabei werden die zweifach zusammenhängenden Komponenten des Teilgraphen zu größeren verschmolzen, da jede Rückwärtskante einen alternativen Weg zwischen den beteiligten Komponenten bildet, der die früheren Artikulationsknoten umgeht. Damit bei der Einbettung keine Kantenkreuzung erzeugt wird, hält der Algorithmus für planare Graphen die Invariante aufrecht, dass zu jedem Zeitpunkt alle noch nicht verarbeiteten Rückwärtskanten in die aktuelle äußere Facette eingebettet werden können. Die Regeln des Algorithmus sichern, dass sich alle Knoten, die noch einzubettende inzidente Rückwärtskanten besitzen, jederzeit auf der äußeren Facette der entstehenden Einbettung befinden.

Die Ausgabe des Boyer-Algorithmus ist die zyklische Reihenfolge der inzidenten Kanten um jeden Knoten für eine kombinatorische Einbettung. Mit dem in Kapitel 3 beschriebenen Algorithmus 2 kann daraus auch die Menge der Facetten für diese Einbettung bestimmt werden. Andererseits lässt sich gerade die Darstellung der kombinatorischen Einbettung mit zyklischen Inzidenzlisten sehr gut als Ausgangspunkt für Heuristiken zur geometrischen Einbettung nutzen. Auch für nicht zweifach zusammenhängende Graphen werden von Boyers Algorithmus geeignete Ausgangsdaten für die zweite Phase der Berechnung einer geometrischen Einbettung geliefert.

## 6. Bestimmen der Geometrie

In diesem Kapitel wird nun das Problem Geometrische Einbettung behandelt. Gegeben ist also ein schlichter, ungerichteter planarer Graph $G = (V, E)$ zusammen mit einer kombinatorischen Einbettung, beispielsweise in Form einer Liste von Facetten $\{F_1, F_2, ..., F_f\}$. Gesucht wird eine ebene geometrische Einbettung $g$, die diese Facetten realisiert. Dabei müssen jedem Knoten $v$ des Graphen eineindeutig seine Koordinaten $g(v) = (x_v , y_v)$ zugeordnet werden. Es ist nicht schwierig, diese Abbildung dann auf die Kanten zu erweitern, wenn alle Verbindungen geradlinig sein sollen. Dazu muss lediglich jeder Kante $e = \{u, v\}$ die Verbindungsstrecke $[g(u), g(v)]$ zugeordnet werden.

Im System VinetS sind für diese Phase der Planarisierung mehrere Algorithmen implementiert. Dies sind mehrere eigene Varianten des auf dem Kräftemodell basierenden Algorithmus von Tutte (vgl. [22] und [23]), der Algorithmus von Kant (s. [14], [3] und [4]) sowie zwei selbst entwickelte Heuristiken, die nicht immer kreuzungsfreie Zeichnungen liefern. Hier können diese Verfahren nur skizziert werden.

#### Der Algorithmus von Tutte

Der Algorithmus von Tutte ist ein klassisches Resultat des Graph Drawing (s. [6]). Das Verfahren liefert für jeden dreifach zusammenhängenden Graphen eine ebene Einbettung mit konvexen Gebieten. Jedoch liegen in der Zeichnung Knoten oft sehr dicht nebeneinander, was nicht besonders übersichtlich ist. Vorteile des Verfahrens sind dagegen, dass es leicht zu implementieren ist und dass es für jeden Graph, der wenigstens einen Kreis enthält, eine Zeichnung liefert.

Die Grundidee des Algorithmus besteht darin, zunächst die Knoten eines Kreises als äußere Facette zu platzieren und dann in deren Innerem alle anderen Knoten in einer "stabilen Position" anzuordnen. Man geht im Modell davon aus, dass sich in $G$ benachbarte Knoten jeweils mit einer zu ihrer Entfernung proportionalen Kraft anziehen. Die Knoten der äußeren Facette haben feste Koordinaten, alle anderen Knoten sind frei beweglich. In einem solchen mechanischen System stellt sich ein Gleichgewicht ein – die entsprechenden Positionen der Knoten ergeben eine ebene Einbettung. Bei der Berechnung nähert man sich in einigen Iterationen sehr schnell dem Endzustand, wenn alle inneren Knoten im Schwerpunkt ihrer Nachbarn platziert werden.

Im System VinetS kann der Nutzer interaktiv den Kreis für die äußere Facette wählen, dabei werden ihm für planare Graphen nur die Facetten einer kombinatorischen Einbettung angeboten. Für alle anderen Graphen kann ein beliebiger Kreis als äußerer Rand benutzt werden, die Zeichnung enthält dann natürlich Kreuzungen. Nur für dreifach zusammenhängende Graphen ist garantiert, dass die Facetten einer gelieferten geometrischen Einbettung tatsächlich der zuvor bekannten kombinatorischen Einbettung entsprechen. Die Knoten der äußeren Facette können nach Wahl auf einem Kreis oder einem Rechteck angeordnet werden. Eine leichte Verbesserung der Zeichnungen kann auch noch dadurch erreicht werden, dass große Facetten durch temporär eingefügte zusätzliche Knoten in ihrer Mitte stärker zusammengezogen werden.

#### Der Algorithmus von Kant

Der Algorithmus von Kant ist nur auf dreifach zusammenhängende Graphen anwendbar. Er garantiert die Anordnung der Knoten auf disjunkten ganzzahligen Gitterpunkten in einer beschränkten Zeichenfläche, jedoch

liefert er oft "schiefe" Bilder. Er erfordert relativ viel Aufwand beim Implementieren, insbesondere wenn lineare Laufzeit sichergestellt werden soll. Seine Idee wird auch in [24] beschrieben.

Im ersten Schritt des Algorithmus werden die Knoten des Graphen in der Reihenfolge eines so genannten "canonical ordering" geordnet. Dabei wird die vorliegende kombinatorische Einbettung des Graphen benutzt. Eine kanonische Ordnung ist eine Zerlegung der Knotenmenge von $G$ mit folgenden Eigenschaften:

- $V = V_1 \cup V_2 \cup ... \cup V_k$ und $V_i \cap V_j = \varnothing$ für $i \neq j$.
- Die Knoten der Menge $V_i$ werden im $i$-ten Schritt eingebettet, jeweils zusammen mit allen inzidenten Kanten, die zu schon vorher eingebetteten Knoten führen. Der nach Schritt $i$ bereits eingebettete Teilgraph wird mit $G_i$, seine äußere Facette mit $C_i$ bezeichnet.
- In der bereits konstruierten Einbettung von $G_i$ liegen alle Knoten, die zu noch nicht eingebetteten Knoten adjazent sind, auf dem äußeren Rand.
- Alle $G_i$ sind 2-fach zusammenhängend und intern 3-fach zusammenhängend.
- Die erste Menge $V_1 = \{v_1, v_2\}$ besteht aus genau zwei adjazenten Knoten. Die letzte Menge $V_k = \{v_n\}$ enthält genau einen Knoten, dieser ist zu $v_1$ benachbart.
- Für alle $i = 2, ..., k-1$ ist entweder $V_i = \{z\}$ , wobei $z$ mindestens einen noch nicht eingebetteten Nachbarn in $G \setminus G_i$ hat oder $V_i = \{z_1, z_2, ... , z_l\}$, wobei $z_1$ und $z_l$ je genau einen Nachbarn in $C_{i-1}$ haben und die anderen $z_j$ haben keinen Nachbarn in $G_{i-1}$ und $z_1, z_2, ..., z_l$ ist ein Pfad, der auf dem äußeren Rand $C_i$ eingebettet werden muss, da alle $z_j$ Nachbarn in $G \setminus G_i$ haben.
- $v_1$ hat die Position $(0, 0)$. Im $i$-ten Schritt hat $v_2$ die Position $(2i, 0)$.
- Die Knoten aus $V_i$ erhalten die $y$-Koordinate $i - 1$.
- Die $x$-Koordinaten der Knoten aus $V_i$ werden entlang des Pfades gleichmäßig zwischen dem linksten und rechtesten Nachbarn in dem schon eingebetteten Teilgraphen verteilt. Dabei müssen möglicherweise Korrekturen in den vorhergehenden Schichten vorgenommen werden.

Die berechnete ebene Einbettung hat garantiert nur konvexe Gebiete und alle Knoten liegen auf einem ganzzahligen Gitter, der Flächenbedarf der Zeichnung ist höchstens quadratisch in der Anzahl der Knoten.

#### Die Level-Heuristik und die Cycle-Heuristik

Ausgehend von den zyklisch geordneten Inzidenzlisten einer kombinatorischen Einbettung kann man versuchen, die Knoten des Graphen in Schichten anzuordnen, so dass die Planarität gewahrt bleibt.

Im System VinetS wurden zwei Varianten davon im Anschluss an den Boyer-Algorithmus implementiert. Bei der Level-Heuristik kann der Nutzer interaktiv einen Knoten für die oberste Schicht wählen. Seine Nachbarn werden in einer Schicht darunter platziert usw. für deren Nachbarn. Bei diesem Herangehen entstehen auch bei planaren Graphen nicht immer kreuzungsfreie Bilder, da die ebene kombinatorische Einbettung auf der Kugel durch den Algorithmus beim Übergang zur Zeichenfläche u. U. an einer ungünstigen Stelle aufgeschnitten wird. Oft lassen sich bessere Resultate mit einem anderen Wurzelknoten erzielen.

Eine ähnliche Methode nutzt die Cycle-Heuristik, dabei werden die Knoten jedoch auf konzentrischen Kreisen angeordnet. Außen wird mit einer Facette begonnen und danach werden Schichten entsprechend der Kantenreihenfolgen aus der ersten Planarisierungsphase gebildet. Auch bei dieser Heuristik spielt die Größe der Facetten eine Rolle für die Qualität des Layouts. Es entstehen unnötige Kreuzungen, wenn im Inneren eines Kreises mit wenigen Knoten einer mit vielen Knoten angeordnet werden muss. Der Nutzer hat wiederum die Möglichkeit, interaktiv in den Algorithmus einzugreifen.

In Abbildung 6 kann man sehr gut erkennen, dass die verschiedenen Algorithmen Bilder mit ganz unterschiedlichem Aussehen liefern. Der Nutzer hat die Möglichkeit, durch das Einstellen verschiedener Parameter für seinen Graphen ansprechende Zeichnungen zu generieren.

Gegenwärtig wird daran gearbeitet, die Algorithmen noch breiter nutzbar zu machen. Die im Augenblick bei einigen Algorithmen noch notwendigen Beschränkungen bezüglich des Zusammenhangs der Graphen sollen entfallen. Darüber hinaus sollen domänenspezifische Nebenbedingungen beim Layout berücksichtigt werden. Es gibt bereits Überlegungen für Verfahren zur Planarisierung von Hypergraphen mit hierarchischen Blöcken.

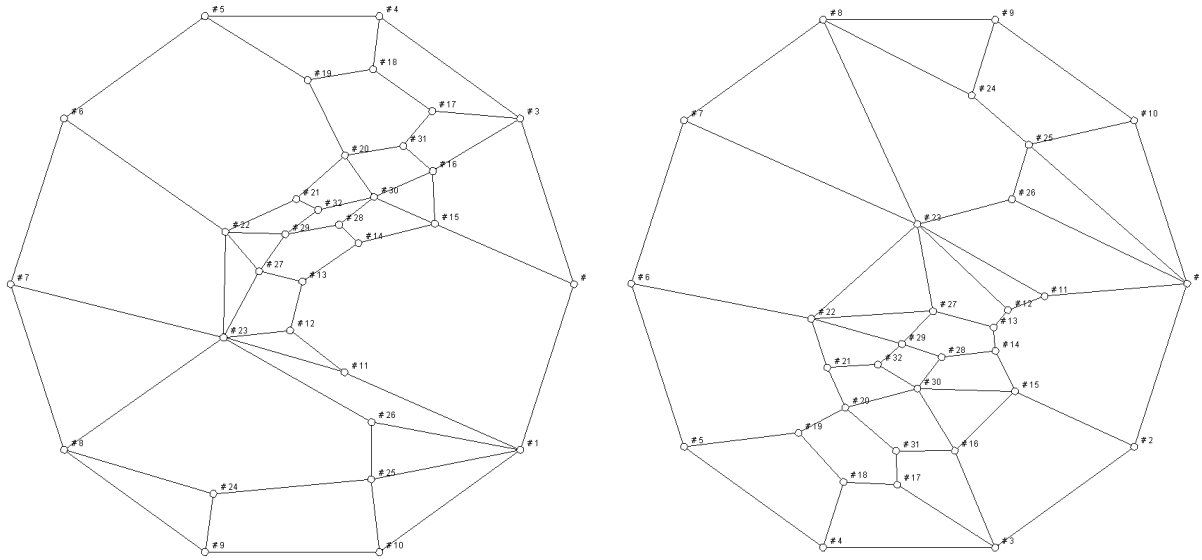
## 7. Danksagung

Bei der Entwicklung des Systems VinetS haben die Absolventen bzw. Studenten der Wirtschaftsinformatik Arne-Michael Törsel, Ronny Haak, Christian Pervölz, Steffen Proksch und Matthias Zillmann engagiert mitgearbeitet. Viele ihrer Ideen sind in die vorliegende Arbeit eingeflossen. Sehr interessante Anregungen verdanke ich auch Dr. Bernhard Goetze von der Gesellschaft zur Förderung angewandter Informatik (GFaI e.V.) Berlin, mit der wir im Rahmen des gesamten Projektes sehr fruchtbar kooperiert haben.

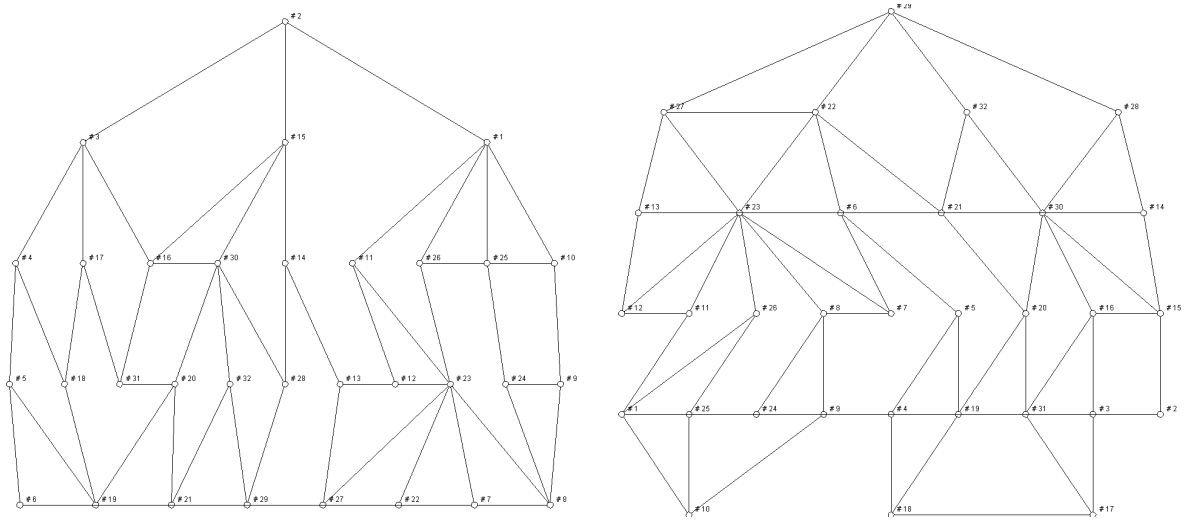
## 8. Literatur


1. J. Boyer, W. Myrvold: Stop minding your P's and Q's: A simplified O(n) planar embedding algorithm. In: Proc. 10th Ann. ACM-SIAM Symp. on Discrete Algorithms SODA 1999, 140–146.
2. J.M. Boyer, P.F. Cortese, M. Partignani, G. Di Batista: Stop minding your P's and Q's: Implementing a Fast and Simple DFS-based Planarity Testing and Embedding Algorithm. In: Graph Drawing (ed. G. Liotta), Proc. 11th Intern. Symp. GD 2003, LNCS 2912, Springer 2004, 25–36.
3. M. Chrobak, G. Kant: Convex grid drawings of 3-connected planar graphs. International Journal of Computational Geometry and Applications 7 (1997) 3, 211–223.
4. M. Chrobak, D. Payne: A linear-time algorithm for drawing planar graphs. Information Processing Letters 54 (1995), 241–246.
5. G. Demoucron, Y. Malgrange, R. Pertuiset: Graphes planaires: Reconnaissance et construction de representation planaires topologiques. Rev. Francaise de Rech. Operationelle 8 (1964), 33–47.
6. G. Di Batista, P. Eades, R. Tamassia, I.G. Tollis: Graph Drawing. Algorithms for the Visualization of Graphs. Prentice Hall 1999.
7. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley 1995.
8. B. Goetze: Planarisierung von Netzwerken. Anforderungsdokument GFaI-VinetS-09-02, GFaI e.V. Berlin 2003.
9. R.J. Gould: Graph Theory. Benjamin Cummings Publishing 1988. (insbesondere Chapter 6: Planarity) s. auch: `http://www.mathcs.emory.edu/~rg/book/chap6.ps`
10. GraphML Team: The GraphML File Format. Homepage: `http://graphml.graphdrawing.org`
11. R. Haak: Eine effiziente Implementation des Algorithmus von Demoucron-Malgrange-Pertuiset zur Einbettung planarer Graphen. Diplomarbeit, FH Stralsund 2002.
12. J.E. Hopcroft, R.E. Tarjan: Efficient planarity testing. J. Assoc. Comput. Mach. 21 (1974) 4, 549–568.
13. M. Jünger, P. Mutzel (eds.): Graph Drawing Software. Springer 2004.
14. G. Kant: Drawing planar graphs using the canonical ordering. Algorithmica 16 (1996), 14–32.
15. M. Kaufmann, D. Wagner (eds.): Drawing Graphs. Methods and Models. LNCS 2025, Springer 2001.
16. C. Kuratowski: Sur le problème des courbes gauches en topologie. Fund. Math. 15 (1930), 271–283.
17. A. Liebers, A.: Planarizing Graphs – A Survey and Annotated Bibliography. J. of Graph Algorithms and Applications 5 (2001) 1, 1–74. s. auch: `http://www.cs.brown.edu./publications/jgaa/volume05.html`
18. K. Mehlhorn, P. Mutzel: On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. Algorithmica 16 (1996) 2, 233–242.
19. P. Scheffler: Projekt VinetS – Software zur Visualisierung vernetzter Strukturen. Homepage: `http://vinets.fh-stralsund.de/index.html`
20. A.-M. Törsel: Implementation und Analyse des Algorithmus von Boyer-Myrvold zur Einbettung planarer Graphen. Diplomarbeit, FH Stralsund 2002.
21. A.-M. Törsel: An implementation of the Boyer-Myrvold algorithm for embedding planar graphs. FH Stralsund, FB Wirtschaft, Diskussionsbeiträge, Heft 17/2003. s. auch: `http://home.arcor.de/arne-michael/docs/BoyerImplementation.pdf`
22. W.T. Tutte: Convex representations of graphs. Proc. London Math. Soc. 10 (1960) 3, 304–320.
23. W.T. Tutte: How to draw a graph. Proc. London Math. Soc. 13 (1963) 3, 743–768.
24. R. Weiskircher: Drawing Planar Graphs. In [15], Kapitel 2, 23–45. s. auch: `http://www.ads.tuwien.ac.at/people/Weiskircher/gigd/gigd.html`
25. D.B. West: Introduction to Graph Theory. 2nd ed., Prentice-Hall 2001.

**Tutte-Heuristik (ohne bzw. mit Triangulierung großer Facetten):**



**Level-Heuristik (mit verschiedenen Startknoten):**



**Cycle-Heuristik:**

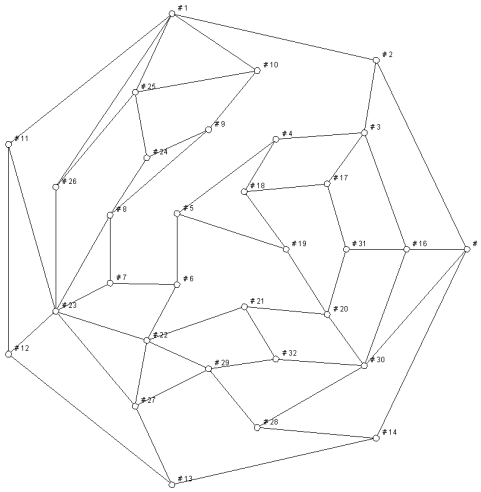


Abbildung 6 Automatisch erzeugtes Layout eines Beispielgraphen im System VinetS