Diplomarbeit

Automatisches Layout von Geschäftsprozessen

vorgelegt von:
Philip Effinger

Mai 2008

Betreuer:
Prof. Dr. Michael Kaufmann
Martin Siebenhaller

Arbeitsbereich Paralleles Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften
Eberhard-Karls-Universität Tübingen

Erklärung:

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe und dabei keine anderen Hilfsmittel und Quellen als die angegebenen verwendet wurden.

Tübingen, den 31.05.2008

Danksagung:

Zuallererst möchte ich denjenigen Menschen danken, die zum Entstehen dieser Arbeit beigetragen haben.
Zu nennen ist hier Professor Kaufmann, der mir ermöglich hat, Teil seiner Arbeitsgruppe zu sein. Am beeindruckendsten fand ich, dass trotz Forschung auf hohem Niveau und dem damit verbundenen Druck die Entspanntheit und Freundlichkeit in der Arbeitsgruppe immer spürbar war.
Zu danken habe ich auch meinem Betreuer Martin Siebenhaller, der mich durch einige lange Stunden Codeentwicklung begleitet hat. Seine Hinweise und Tipps in Sachen yFiles waren immer Gold wert und dank seiner Erfahrung konnte er mir oft für noch kommende Hürden hilfreiche Ratschläge geben. Beim Korrekturlesen hat Martin auch zahlreiche Verbesserungsvorschläge beigetragen, die ich gerne übernommen habe.
Danken möchte ich auch dem gesamten Team aus dem Lehrstuhl, das mich mit Kaffee und lustigen Sprüchen durch die Zeit begleitet hat, und dadurch die Motivation hoch hielt.
Zum letzten sprachlichen Feinschliff haben noch Johannes Spielmann und Kristina Abels beigetragen, als man schon hätte meinen können, ich hätte die deutsche Sprache verlernt.

Inhaltsverzeichnis

1 Einführung

1.1 Einleitung

> „Wir müssen unsere Prozesse permanent optimieren, um wettbewerbsfähig zu bleiben."

Dieses Zitat eines Vorstandsvorsitzenden in einem großen, deutschen Wirtschaftsunternehmen ist kein Sonderfall, sondern in der Welt der Betriebswirtschaft bereits eher ein geflügeltes Wort, wenn es sich um eine angepeilte Steigerung der Produktivität und der Effizienz handelt.
Automatisch kommt die Frage auf, welche Prozesse denn gemeint sind und wie man sie optimiert? Genauer betrachtet sind damit Geschäftsprozesse gemeint, wie sie in Unternehmen tagtäglich durchgeführt werden. Als Beispiele sind hier zu nennen: das Bestellwesen, die Logistik, der Verkauf, die Personalplanung, u.v.m. Jeder Vorgang in einem Teil dieser unternehmerischen Tätigkeiten ist als Prozess festgehalten und geregelt.
Die zweite Teilfrage, wie eine Optimierung vorgenommen werden kann, ist komplexer. Dies ist der Bereich, in dem zahlreiche Beratungshäuser ihr Kerngeschäft besitzen. Für eine Optimierung eines Prozesses muss zunächst einmal der bisherige, wenn überhaupt existente Prozess modelliert oder auf andere Weise festgehalten werden. Im nächsten Schritt folgt eine Prozessanalyse, in welcher Schwachstellen, die hohe Kosten verursachen oder sehr zeitaufwändig sind, aufgedeckt werden sollen. Und in einem letzten Schritt wird der neue, optimierte Prozess wieder in die Prozessumgebung des Unternehmens implementiert.
In dieser Arbeit liegt der Fokus auf dem ersten der Teilschritte, der Prozessmodellierung. Die anderen Schritte sind nicht weniger große Gebiete, in denen unzählige Betriebswirtschaftler ihr Tätigkeitsfeld finden.
Die Prozessmodellierung, oder auch die Prozessvisualisierung, hat zum Ziel, einen Prozess anschaulich und verständlich zu machen für denjenigen, der mit ihm arbeiten oder ihn anderen erläutern muss. Dazu bedient man sich Modellierungswerkzeugen in Form von Software, welche in den Beratungshäusern zum Standardrepertoire gehören. Diese Werkzeuge unterstützen den grafischen Entwurf von Prozessmodellen. Teilweise bieten sie auch eine Simulation des Prozesses an, um eine Auswertung vornehmen zu können.
Der Schwerpunkt dieser Arbeit liegt auf dem Entwurf eines Prozessmodelles. Der Aufwand, um ein Prozessmodell zu entwerfen, ist sehr groß, wenn man keine automatische Unterstützung durch die verwendeten Werkzeuge erfährt. Daher ist das Ziel dieser Arbeit, ein Werkzeug mit einem automatischen Layout-Modul für Geschäftsprozesse zu entwickeln. Das Modul konstruiert ein Layout für einen Entwurf, das orthogonale Kanten besitzt und weiteren ästhetischen Merkmalen, wie

z.B. Flussrichtung des Prozesses, genügt. Die Implementierung erlaubt die Modellierung eines neuen Prozesses, das Konstruieren eines automatischen Layouts und auch das Verändern eines bestehenden Prozesses. Die Konstruktion eines Layouts ist keinesfalls eine statische Aktion, sondern besteht auch aus interaktiven Elementen, wenn mittels skizzenbasiertem Layout in ein bestehendes Layout ein neu eingefügtes Element eingepasst werden soll.
Das Problem des automatischen Layouts ist sehr komplex, daher greife ich in dieser Arbeit auf graphtheoretische Ansätze zurück, da die Forschung in diesem Fachgebiet der Informatik seit Jahren einen rasanten Schub erfährt und für unsere Zwecke schon sehr gute Ergebnisse vorweisen kann.
Für die Modellierung der Prozesse verwenden wir die Notationssprache *BPMN*, die sich aktuell in einem Standardisierungsverfahren für Prozessmodellierung in der *Object Management Group* (OMG) befindet. Für die Berechnung des Layouts kommt das Kandinsky-Modell und der Sugiyama-Algorithmus zum Tragen. Um das zugrunde liegende Optimierungsproblem des orthogonalen Layout zu lösen, verwenden wir ein Netzwerkflussmodell. Für die zusätzliche Vereinfachung von berechneten Layouts wird das Konzept der Konnektoren und der Schnittbildung erläutert, also die Aufteilung des Prozessmodells in kleinere Teile unter Berücksichtigung von Bedingungen, wie z.B. minimaler Schnitt. Das Prinzip der Konnektoren ist, lange und unübersichtliche Kanten überflüssig zu machen und zusätzlich die Übersichtlichkeit zu verbessern. Dabei werden diese Kanten durch zwei Stellvertreter in Form von Knoten ersetzt. Außerdem können sie bei Schnittbildung von Modellen auf Verbindungen verweisen, die außerhalb des Teilmodells liegen.

1.2 Überblick

Im nächsten Kapitel 2 werden die graphtheoretischen Grundlagen vorgestellt, die in dieser Arbeit verwendet werden. Dazu werden die Idee des *min-cost-flow*-Netzwerk vorgestellt und die Repräsentation eines Prozessmodells erläutert. In Kapitel 3 wird die Notationssprache *BPMN* (Business Process Modeling Notation) vorgestellt und ihr Funktionsumfang und die einzelnen Elemente eingeführt. Die kompletten theoretischen Ansätze, die für das Layout verwendet werden, finden sich in Kapitel 4. Das skizzenbasierte Layout, das auch Interaktivität mit dem Benutzer fördert, wird in Kapitel 6 mit vielen Beispielen vorgeführt. Ein theoretischer Ansatz zur Schnittbildung in Graphen, die das automatische Aufteilen eines Prozessmodells in beliebig viele, kleinere Teile ermöglicht, stelle ich in Kapitel 7 vor. Eine kurze Einführung in die Funktionalität und Umfang der Implementierung, die Teil dieser Arbeit war, findet sich in Kapitel 8.

2 Grundlagen

2.1 Graphen und Bäume

Im Folgenden sollen die notwendigen graphtheoretischen Begriffe, die in dieser Arbeit verwendet werden, definiert werden. Eine allgemeinere Zusammenfassung ist z.B. in [CLRS01] zu finden.

Ein Graph G ist ein geordnetes Paar $G = (V, E)$, wobei V die Menge der Knoten und E die Menge der Kanten darstellt. Die Menge E der Kanten kann sowohl gerichtete Kanten E_D als auch ungerichtete Kanten E_U enthalten. Dabei ist eine gerichtete Kante ein geordnetes Paar (u, v) mit $u, v \in V$. Ungerichtete Kanten sind ungeordnete Paare (u, v), d.h. bei ungerichteten Kanten gilt, dass $(u, v) = (v, u)$. Für die Menge aller Kanten gilt: $E = E_U \cup E_D$. Eine Kante $(v, v), v \in V$ nennt man Selbstschleife (self-loop). Bei einer Selbstschleife sind also Anfangs- und Endpunkt identisch.
Die Unterscheidung zwischen Inzidenz und Adjazenz wird hier wie folgt definiert: Eine Kante e ist inzident zu einem Knoten v, wenn v der Endpunkt der Kante ist, entsprechend ist v adjazent zu einem Knoten u, falls $(u, v) \in E$. Benachbart ist ein Knoten v zu einem zweiten Knoten u, wenn entweder $(v, u) \in E$ oder $(u, v) \in E$. Die Beziehung zwischen benachbarten und adjazenten Knoten ist zu differenzieren indem man weiß: Ein zu u adjazenter Knoten v ist auch ein Nachbar von u, umgekehrt gilt die Aussage nur in ungerichteten Graphen.
Ein gerichteter Graph $G = (V, E)$ ist ein Graph mit $E = E_D$, ein ungerichteter entsprechend mit $E = E_U$. Ist $E = E_U \cup E_D,\ E_U, E_D \neq \emptyset$, so nennt man G teilgerichtet (mixed).
Ein Subgraph $G' = (V', E')$ von G ist ein Graph mit $V' \subseteq V$ und $E' \subseteq E$. Gilt Gleichheit für $V' = V$, so ist G' ein Teilgraph von G. Der Subgraph G' mit $V' \subseteq V$ heißt induziert, falls G' alle Kanten $(u, v) \in E' \subseteq E$ mit $u, v \in V'$ enthält.
Eine wichtige Eigenschaft eines Knotens v ist der Grad $\delta(v)$, der die Anzahl der zu v inzidenten Kanten angibt. Ist der Graph gerichtet, so wird der Grad eines Knotens v genauer unterschieden in Ausgangsgrad $\delta_{out}(v)$, der durch $|\{u : (v, u) \in E\}|$ definiert ist, und den Eingangsgrad $\delta_{in}(v) := |\{u : (u, v) \in E\}|$. Die Summe von $\delta_{out}(v)$ und $\delta_{in}(v)$ entspricht $\delta(v)$. Ein k-Graph ist ein Graph, dessen Knoten einen maximalen Grad von k besitzen, also: $max_{v \in V}\delta(v) \leq k$. Selbstschleifen werden bei der Bestimmung des Grades doppelt gezählt.
Die Isomorphie zweier Graphen $G = (V, E)$ und $G' = (V', E')$ ist definiert durch die Existenz einer bijektiven Abbildung $f : V \to V'$, sodass $(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$. Ein Graph $G = (V, E)$ heißt bipartit, falls die Knotenmenge V in zwei disjunkte Mengen V_1, V_2 geteilt werden kann, wobei $V = V_1 \cup V_2, V_1 \cap V_2 = \emptyset$ erfüllt und für alle Kanten $(u, v) \in E$ die logische Bedingung $(u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1)$

wahr sein muss. Jede Kante verbindet also jeweils einen Knoten aus V_1 mit einem Knoten aus V_2.
Ein Weg oder Pfad p in einem Graphen ist eine Knotenfolge $(v_0, \ldots, v_n)$ von Knoten $v_i \in V$ mit Startknoten $u = v_0$ und Endknoten $v = v_n$, wobei $(v_i, v_{i+1}) \in E$, $\forall\, i, 0 \leq i < n$. Die Länge des Weges ist n, also gleich der Anzahl der überlaufenen Kanten. Gilt $v_i \neq v_j, \forall\, 0 \leq i, j \leq n, i \neq j$, d.h. die Knoten in p sind paarweise verschieden, so nennt man p einen einfachen Pfad. Ein Pfad heißt Zyklus, falls $v_0 = v_n$. Ein Graph ist azyklisch, wenn kein Knotenpaar (u, v) existiert, für das ein einfacher, zyklischer Pfad gefunden werden kann. Wenn für jedes Knotenpaar $(u, v) \subseteq (V \times V), v \neq u$ ein Pfad von u nach w in G existiert, so nennt man G einen zusammenhängenden Graphen.
Ein Baum T ist ein zusammenhängender, azyklischer und ungerichteter Graph. Ein Spannbaum (spanning tree) ist ein Baum, der Teilgraph von G ist und genau (|V|-1) Kanten enthält. Ist auf den Kanten E eine Gewichtsfunktion $w : E \to \mathbb{N}$ definiert, so ist der Baum genau dann ein minimaler Spannbaum von G (minimal spanning tree, MST), wenn die Summe der Kantengewichte $\sum_{e \in E} w(e)$ minimal ist. Ein MST eines Graphen heißt auch Minimalgerüst.
Jeder gewurzelte Baum (rooted tree) besitzt genau eine Wurzel (root) r. Die Knoten $u \subseteq V$, die auf dem Pfad von r nach v, für einen Knoten $v \in V$, durchlaufen werden, heißen Vorgängerknoten von v und v ist ein Nachfolger von u. Die Wurzel r eines Baumes hat keinen Vorgängerknoten. Ein Subbaum T' von T ist ein Baum, der durch Wahl eines Knotens v als Wurzel von T' induziert wird. Ein Knoten u ist Vater von v, wenn u ein Vorgänger von v ist und $(u, v) \in E$, v heißt Kind von u. Jeden Knoten, der einen Vater hat aber keine Kinder, nennt man Blatt. Die Wurzel besitzt als einziger Knoten keinen Vater.

2.2 Einbettung

Eine Abbildung M eines Graphen $G = (V, E)$ in die Ebene $\mathbb{R}^2$ heißt „ebene Einbettung“, falls für M gilt:

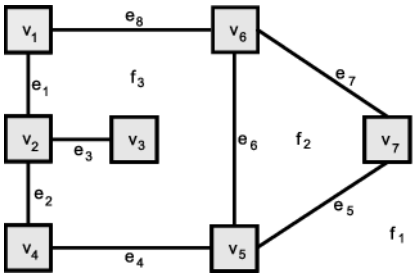
- V wird auf verschiedene Punkte der Ebene abgebildet (injektiv)
- E wird auf offene Jordan-Kurven abgebildet. Eine Jordan-Kurve ist eine planare Kurve, die topologisch äquivalent zum Einheitskreis ist. Die Abbildungen der Endpunkte der entsprechenden Kante geben die Punkte der Fläche vor, die durch die Kurve verbinden werden soll.
- Kein Paar von Kurven überschneidet sich.

Die Planarität, also die Überkreuzungsfreiheit, eines Graphen ist damit definiert durch:

Definition 1 (Planarität):
Ein Graph ist dann planar, wenn für ihn eine ebene Einbettung existiert.

Die Einbettung teilt die Fläche des Graphen in verschiedenen Regionen, welche Faces genannt werden. Das den Graphen umgebende, unendliche Face bildet das äußere

Face. Jede Kante trennt zwei Faces voneinander. Sind diese Faces identisch, so heißt die trennende Kante „Brücke“. Die Schnittmenge zweier Faces ist immer leer, ein Face kann jedoch geometrisch von einem anderen umgeben sein.
Die Einbettung ist unabhängig davon, ob der Graph gerichtet oder ungerichtet ist. Die Unterscheidung zwischen der Einbettung und der geometrischen Zeichnung ist wesentlich: Eine kreuzungsfreie Zeichnung gibt immer eine mögliche planare Einbettung an; ein Graph mit einer kreuzungsbehafteten Zeichnung aber kann dennoch eine planare Einbettung besitzen. Besitzt ein Graph eine planare Einbettung, kann er auch kreuzungsfrei gezeichnet werden.
Die Einbettung wird formal beschrieben durch die planare Repräsentation P. Sie enthält die Topologie eines planaren Graphen $G = (V, E, F)$, wobei F die Menge der Faces darstellt. Für jedes Face $f \in F$ enthält die Repräsentation eine Liste $P(f)$, die die Kanten enthält, die das Face f definieren. Die Reihenfolge der Kanten in den Listen wird für die zwei Typen von Faces unterschieden. Ist f das äußere Face, dann entspricht die Liste der zyklischen Reihenfolge der Kanten, wenn diese entgegen dem Uhrzeigersinn durchlaufen werden. Ist das Face nicht das äußere Face, sondern ein inneres, so gibt das Durchlaufen der zyklischen Reihenfolge der Kanten im Uhrzeigersinn den Inhalt der Liste $P(f)$ wieder. Bei Brücken wird die entsprechende Kante zweimal durchlaufen, wobei die Richtung jeweils umgekehrt ist. Ein Beispiel für eine planare Repräsentation ist in Abb. 2.1 zu sehen.



f_1 = äußeres Face
$P(f_1) = (e_1, e_2, e_4, e_5, e_7, e_8)$
$P(f_2) = (e_7, e_5, e_6)$
$P(f_3) = (e_8, e_6, e_4, e_2, e_3, e_3, e_1)$

Abbildung 2.1: Beispiel für eine planare Repräsentation $P(F)$ und den Face-Listen.

Die planare Repräsentation eines Graphen ist nicht eindeutig, da der Startpunkt des Durchlaufens beim Erstellen der Listen $P(F)$ nicht eindeutig ist. Da jede Kante zweimal in den Listen auftaucht, gilt allgemein für die planare Repräsentation P:

$$\sum_{f \in F} |P(f)| = 2 \cdot |E|$$

Auch Faces haben einen Grad $\delta(f)$, der die Länge der Liste $|P(f)|$ und damit die Anzahl der das Face begrenzenden Kanten angibt.
Der Zusammenhang zwischen Knoten, Kanten und Faces wird bei der Betrachtung des folgenden Satzes ersichtlich.

Satz 1 (Eulersche Formel): *Sei $G = (V, E)$ ein zusammenhängender, planarer Graph und F die Menge der Faces von G, dann gilt*

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

Ein Beweis für Satz 1 findet sich in [EWHK+96] im Abschnitt 6.1.2.

2.3 Orthogonale Gittereinbettung

Im vorigen Teil haben wir eine planare Einbettung für einen Graph G definiert. Nun wird wie in [Sie03] die Abbildung auf ein Gitter und damit auf eine orthogonale Einbettung Q des Graphen erweitert. Zur Repräsentation erläutern wir dann die orthogonale Repräsentation H des Graphen G.

Definition 2 (Orthogonales Gitter):
Ein orthogonales Gitter ist eine Einbettung eines unendlichen 4-Graphen, die nicht nur eben ist, sondern dessen Knoten ganzzahligen Koordinaten auf dem Gitter besitzen und dessen Kanten Einheitslänge besitzen.

In dieses orthogonale Gitter betten wir den Graphen ein.

Definition 3 (Orthogonale Gittereinbettung):
Eine orthogonale Gittereinbettung eines Graphen $G = (V, E)$ ist eine Abbildung Q von G in das orthogonale Gitter mit folgenden Eigenschaften:

- *Alle Kanten $e = (u, v) \in E$ werden auf jeweils einen Pfad $Q(e)$ des Gitters abgebildet, wobei $Q(u)$ und $Q(v)$ die Endpunkte bilden.*
- *Allen Knoten V von G wird auf dem Gitter ein Punkt zugewiesen, wobei Überschneidungen ausgeschlossen werden,*
  d.h. $Q(u) \neq Q(v)\ \forall\ u, v \in V,\ u \neq v$.
- *Zwei Pfade $Q(e_1)$ und $Q(e_2)$ eines Paares $e_1, e_2 \in E$ haben außer den Endpunkten keine gemeinsamen Schnittpunkte.*

Die einzelnen Teile eines Pfades auf dem Gitter nennt man Segmente. Ein Segmentabschluss, der nicht in einem Knoten endet bildet einen Knick, also einen Übergang von einem horizontalem in ein vertikales Segment bzw. umgekehrt. Durch die Einbettung in das orthogonale Gitter werden alle Winkel zu Vielfachen von 90 Grad.

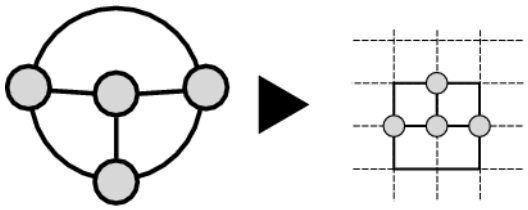


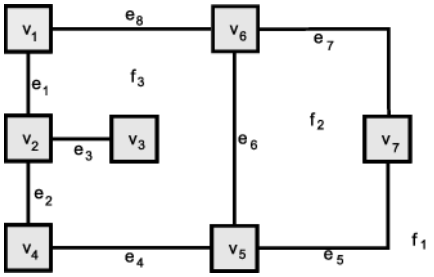
Abbildung 2.2: Beispiel für die orthogonale Gittereinbettung eines einfachen Graphen.

Die orthogonale Gittereinbettung eines Graphen G wird formal in einer orthogonalen Repräsentation H dokumentiert, analog zur planaren Repräsentation P der planaren Einbettung. In der orthogonalen Repräsentation werden zusätzlich im Vergleich zur

planaren Repräsentation P Informationen über den Verlauf der Segmente einer Kante und der Winkel zwischen Kanten eines Faces gespeichert.

Definition 4 (Orthogonale Repräsentation):
Sei $G = (V, E, F)$ ein planarer 4-Graph, d.h. $\delta(v) \leq 4, \forall v \in V$. Eine orthogonale Repräsentation H eines planaren 4-Graphen G und der planaren Repräsentation P enthält für jedes Face $f \in F$ eine Liste $H(f)$, wobei jeder Listeneintrag folgende Information beinhaltet:

- *eine Kante $e \in E$, die f begrenzt*
- *eine binäre Zeichenkette s (Bitvektor). Der Vektor s enthält das Leerelement ε, falls die Kante keine Knicke besitzt. Eine 0 entspricht einem 90-Grad Winkel, eine 1 einem 270-Grad Winkel. Die Reihenfolge der Bits wird durch die Ordnung der Segmente der Kante vorgegeben, wenn man das Face im Uhrzeigersinn durchläuft (bzw. gegen den Uhrzeigersinn beim äußeren Face). Das i-te Bit spiegelt also den i-ten Knick der Kante wieder.*
- *eine Zahl $a \in \{90, 180, 270, 360\}$, die den Winkel zwischen e und der Kante des in $H(f)$ darauf folgenden Listeneintrages angibt.*



f_1 = äußeres Face
$H(f_1) = ((e_1, \varepsilon, 180), (e_2, \varepsilon, 270), (e_4, \varepsilon, 180),$
$(e_5, 1, 180), (e_7, 1, 180), (e_8, \varepsilon, 270))$
$H(f_2) = ((e_5, 0, 90), (e_6, \varepsilon, 90), (e_7, 0, 180))$
$H(f_3) = ((e_4, \varepsilon, 90), (e_2, \varepsilon, 90), (e_3, \varepsilon, 360), (e_3, \varepsilon, 90),$
$(e_1, \varepsilon, 90), (e_8, \varepsilon, 90), (e_6, \varepsilon, 90))$

Abbildung 2.3: Beispiel für eine orthogonale Repräsentation.

Eine orthogonale Repräsentation H eines planaren Graphen G hat folgende Eigenschaften:

1. Seien $L_i.e$ die Kanteneinträge der Liste $L = H(f_i)$ mit $f_i \in F$. Diese bilden eine planare Repräsentation für einen planaren 4-Graphen.
2. Seien $l_1 \in H(f_i)$ und $l_2 \in H(f_j)$ zwei Listeneinträge, d.h. $l_1 \in L_i$ und $l_2 \in L_j$, wobei $i \neq j$ und $l_1.e = l_2.e$, die dieselbe Kante als Feld haben, jedoch zwei unterschiedlichen Faces zugeordnet sind. Die Kante $e = l_1.e = l_2.e$ trennt somit f_i und f_j. Die binäre Zeichenkette $l_1.s$ entspricht dann $l_2.s$, wenn man $l_2.s$ umkehrt und die einzelnen Bits flippt.

3. Sei L_v die Menge der Listeneinträge l_i, deren Kanten $l_i.e$ den Knoten v als Endpunkt haben, wenn $H(f_i)$ in Richtung des Knotens durchlaufen wird. Es gilt dann:
$$\sum_{l_i \in L_v} l_i.a = 360$$
Die Summe der Winkel um einen Knoten ist also immer gleich 360 Grad.

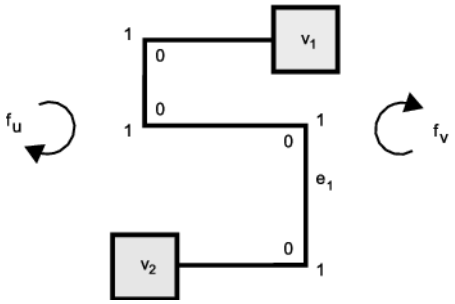


Abbildung 2.4: Beispiel für die Zeichenkette (bit string) einer Kante, die an zwei Faces f_u und f_v grenzt.

2.4 Min-Cost-Flow

Um eine orthogonale Repräsentation eines eingebetteten Graphen $G = (V, E, F)$ zu konstruieren, wird ein Netzwerkflussproblem erstellt, ein sogenanntes `min-cost--flow-problem`. Dieses wird hier allgemein beschrieben. Die nötigen Anpassungen werden später in Kap. 4 erläutert.
Ein Netzwerk $N = (V, E)$ ist ein gerichteter Graph mit zwei ausgezeichneten Knoten, der Quelle s und der Senke t mit $\delta_{in}(s) = 0$ und $\delta_{out}(t) = 0$. Für die Menge der Kanten werden zwei Abbildungen definiert:

- Die Kapazitätsfunktion $cap : E \to \mathbb{R}_0^+$, welche die maximale Flusskapazität einer Kante beschreibt.
- Die Kostenfunktion $cost : E \to \mathbb{R}$, die die für einen Fluss auf Kante $e \in E$ anfallenden Kosten angibt.

Der Fluss f wird nun als eine Abbildung $f : V \times V \to \mathbb{R}$ definiert, die folgende Eigenschaften besitzt:

1. Schiefsymmetrie: $f(v, u) = -f(u, v) \; \forall \; u, v \in V$
2. Kapazitätsbeschränkung: $f(v, u) \leq cap(v, u) \; \forall \; u, v \in V$
3. Flusserhaltungsbedingung: $\sum_{u \in V} f(v, u) = 0 \; \forall \; u, v \in V \setminus \{s, t\}$

Für den Flusswert $|f|$ eines Netzwerkes gilt, dass er genau dem Flusswert des Quellknotens s entspricht:

$$|f| = \sum_{v \in V} f(s, v)$$

Ein maximaler Fluss f ist ein Fluss, dessen Wert $|f|$ größer oder gleich ist als der jedes anderen Flusses. Im `min-cost-flow-problem` soll nun ein maximaler Fluss mit minimalen Kosten $min(cost(f))$ gefunden werden, wobei die Kosten eines Flusses definiert werden durch:

$$cost(f) = \sum_{e \in E} f(e) \cdot cost(e)$$

3 Die Notationssprache BPMN

BPMN steht für ***B**usiness **P**rocess **M**odeling **N**otation*. BPMN ist eine Notationssprache, die von der Object Managemant Group (OMG) entwickelt wurde.

BPMN stellt standardisierte, graphische Elemente zur Verfügung, um den Arbeitsablauf (workflow) in Geschäftsprozessen (business processes, BP) zu modellieren. Im Vordergrund der Modellierung stehen die Sequenzierung der Prozessschritte und die Darstellung der Nachrichten, die während der Ausführung eines Prozesses zwischen den Elementen versandt werden.

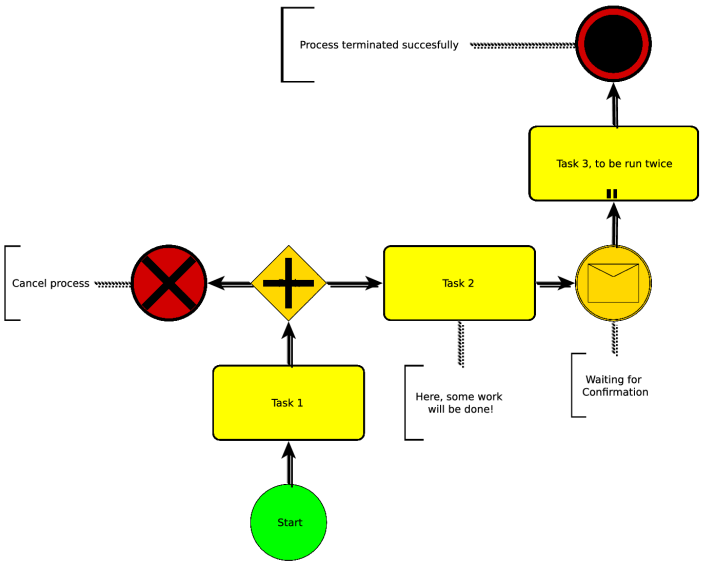


Abbildung 3.1: Beispiel für ein Prozessmodell in BPMN

Zum Zeitpunkt dieser Arbeit ist ein *request for proposal* (RFP) für Version 2.0 [OMG07] der Notationssprache erschienen. Gültig und der aktuelle Standard ist somit noch Version 1.0. Die Unterschiede zwischen den Versionen 1.0 und 2.0 ändern nichts an den Aussagen in dieser Arbeit. Die Änderungen sind meist semantisch und haben wenig Einfluss auf die Syntax der Sprache, die zum Layout herangezogen wird.

3.1 Hintergrund von BPMN

BPMN wurde wesentlich von Stephen A. White (IBM) [Whi04a, Whi04b, Whi05] vorangetrieben. Seine Motivation war vor allem die Bestrebung, eine einheitliche und möglichst vollständige Notation für Geschäftsprozesse zu entwickeln. Ziel ist eine Standardisierung unter dem Dach der OMG, damit die Verbreitung und Akzeptanz gewährleistet ist.

Die *business process execution language* BPEL ist im Vergleich[1] zu BPMN eine implementierungsnahe Sprache zur Beschreibung von Prozessen. Diese Beschreibungen in XML werden nach ihrem Entwurf mittels einer sog. BPEL-Engine in Form eines Web-Services implementiert. Eine Verwandtschaft zur bekannten BPEL besteht insoweit, dass eine Konversion von in BPMN modellierten Prozessen zu implementierfähigem BPEL bereits im Standard von BPMN dargestellt wird. Da BPMN etwas mächtiger ist als BPEL, ist eine Konversion bis auf wenige Ausnahmen, wie z.B. ad-hoc-Prozesse, möglich [Whi05].

Im Wirtschaftsleben ist die Modellierung und Visualisierung von Prozessen ein unentbehrliches Mittel, wenn es darum geht, Prozesse zu verändern [OR03]. Die Veränderung zielt meist auf eine Effizienzsteigerung und Verschlankung des Prozesses ab. Das heißt, die Prozessschritte sollten von geringer Anzahl und kurzer Laufzeit sein. Der Vorgang der Prozessoptimierung kann z.B. Geschäfts-, Produktions- und Entwicklungsprozesse umfassen. Besonders im Beratungsumfeld gehört das Arbeiten mit Prozessoptimierungen zum Alltag. Eine gute Visualisierung kann hier ein entscheidendes Kriterium in der Auftragsaggregation beim Kunden sein. Wenn der Kunde die Möglichkeiten der Prozessoptimierung vom Ist- zum vorteilsreichen Soll-Zustand bildlich dargestellt bekommt, so kann er die Notwendigkeit einer Beratungsleistung zur Prozessumstellung mit eigenen Augen sehen.

BPMN bietet jedoch keine Möglichkeiten der Prozesssimulation. Es können keine Kennzahlen für BPMN-Elemente hinterlegt werden und zeitgesteuerte Modellrechnungen durchgeführt werden. Gerade in der Prozessoptimierung ist jedoch die Simulation des Ist- und des Soll-Zustandes vor einer Umstellung wichtig. Hier kann aber die implementierungsnähere BPEL verwendet werden. Eine Erweiterung von BPMN hin zu einer simulationsfähigen Prozessmodellierung ist über den Umweg einer Konversion zu BPEL möglich.

3.2 Elemente von BPMN

3.2.1 Objekte zur Flusskontrolle im Prozess

Der Fluss in einem BPMN-Graphen wird durch Objekte (flow objects) gesteuert, die den Fluss initiieren, umleiten oder terminieren können.

[1] siehe Anmerkung von Stephen A. White unter:
http://www.bpmn.org/Documents/FAQ.htm#Technical

1. Ereignisse (events):

   Ereignisse steuern den Fluss des Prozesses und gliedern sich in folgende drei Arten, die sich im Modell durch ihr Format unterscheiden:

   a) Startereignisse:
   Startereignisse (start events) initiieren den Prozessfluss. Sie haben eine runde Form und sind grün gefärbt. Im Graphen besitzen sie als Initiatoren nur ausgehende Kanten.

   b) Zwischenereignisse:
   Zwischenereignisse (intermediate events) steuern den Prozessfluss innerhalb eines Prozesses. Sie können zum Beispiel den Fluss verzögern oder auf den Eintritt eines anderes Ereignisses warten, bevor der eigentliche Prozess weiterläuft. Die Form wird durch einen gelb gefüllten Kreis mit dünnem doppelten Rand vorgegeben.

   c) Endereignisse:
   Endereignisse (end events) beenden den Fluss und stellen einen Endpunkt eines Prozesses bzw. eines Prozesszweiges dar. Wenn es nicht das Standard-Endereignis (default end event) ist, dann handelt es sich meist um das Ende eines Prozesszweigs und ist ein Ausnahmeereignis für das Prozessende. Für das reguläre Ende des Prozesses ohne Nebenbedingung wird das Standard-Endereignis gewählt. Im Modell sind die Endereignisse rote Kreise mit dickem schwarzen Rand.

   Die Ereignisvarianten sind z.B. Einflüsse von außen, Datenübertragungen oder interne Vorgänge wie ablaufende Zeitgeber. Die Auflistungen 3.1 und 3.2 beinhalten sämtliche Arten von Ereignissen in der BPMN.

2. Aktivitäten:
   Aktivitäten (activities) sind Vorgänge, die der Prozess erst vollständig beenden muss, bevor der Fluss sich fortsetzt. Je nach Art der Aktivität wird der Vorgang auch mehrmals durchlaufen. Die Form von Aktivitäten im Modell ist ein gelbes Rechteck mit abgerundeten Ecken. In dieser Arbeit werden die in Tabelle 3.3 aufgelisteten Arten von Aktivitäten benutzt.

3. Gateways:
   Gateways behandeln mehrere Prozessflüsse. Sie operieren als logische Gatter und unterstützen sowohl einfache logische Operationen wie *AND*- und *OR*-Verknüpfungen unter eingehenden Flüssen als auch komplexere Verschaltungen von Flüssen, z.B. wie in Multiplexern. Alle BPMN-Gateways sind in Tabelle 3.4 dargestellt.

Grafik	Name	Beschreibung
	default_start	Default Start Event
	default_intermediate	Default Intermediate Event
	default_end	Default End Event
	message_start	Message Start Event
	message_intermediate	Message Intermediate Event
	message_end	Message End Event
	link_start	Link Start Event
	link_intermediate	Link Intermediate Event
	link_end	Link End Event
	rule_start	Rule Start Event
	rule_intermediate	Rule Intermediate Event
	cancel_intermediate	Cancel Intermediate Event
	cancel_end	Cancel End Event
	compensation_intermediate	Compensation Intermediate Event
	compensation_end	Compensation End Event

Tabelle 3.1: Tabelle der BPMN-Ereignisse, Teil 1

Grafik	Name	Beschreibung
	error_intermediate	Error Intermediate Event
	error_end	Error End Event
	timer_intermediate	Timer Intermediate Event
	timer_end	Timer End Event
	terminate_end	Terminate End Event

Tabelle 3.2: Tabelle der BPMN-Ereignisse, Teil 2

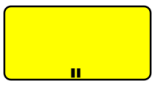
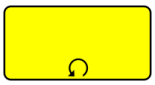
Grafik	Name	Beschreibung
	activity_task	Task
	activity_multiple_instance	Multiple Instance
	activity_loop	Loop

Tabelle 3.3: Tabelle der BPMN-Aktivitäten

3.2.2 Verbindungsobjekte

Die Verbindungsobjekte (Connecting Objects) stellen die Kanten im Graphen dar. Die Modellierung mit BPMN bietet drei Arten von Kanten, die sich durch die von ihnen übertragenen Objekte unterscheiden.

- Prozessabfolge:
  Die Prozessabfolge (Sequence Flow), s. Abb. 3.2, ist der normale Fluss in einem Prozess.

Abbildung 3.2: Sequence Flow

- Nachrichten-/Datenfluss:
  Über den Nachrichten-/Datenfluss (Message Flow), s. Abb. 3.3, werden Nach-

Grafik	Name	Beschreibung
	gateway_fork_join	Fork/Join
	gateway_inclusive	Inclusive Decision/Merge (OR)
	gateway_exclusive_data	Exclusive Decision/Merge (XOR) (data-based)
	gateway_exclusive_event	Exclusive Decision/Merge (XOR) (event-based)
	gateway_complex	Complex Decision/Merge

Tabelle 3.4: Tabelle der BPMN-Gateways

richten zwischen einzelnen Elementen ausgetauscht oder weitergereicht, z.B. Informationen über Prozessstatus, Fehlermeldungen oder externe Daten wie z.B. Bestelllisten.

Abbildung 3.3: Message Flow

- Lose Verbindung:
  Mit einer losen Verbindung (Association), s. Abb. 3.4, können Beziehungen zwischen Elementen verdeutlicht werden oder Annotationen an Elementen angefügt werden.

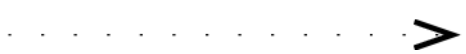


Abbildung 3.4: Association

3.2.3 Bahnen

BPMN-Elemente werden zu Bahnen (Swimlanes) zugeordnet. Bahnen können z.B. verschiedene Abteilungen in einem Unternehmen darstellen. Vorgänge in einem Pro-

zess werden so graphisch einer Abteilung zugewiesen. Durch unterschiedliche Färbung kann die Zuordnung eindeutig gehalten werden.

- Bahn:
  Mehrere Bahnen (Lanes) in einem Pool bieten ein gutes Abbild eines Unternehmens. Die Bahnen beinhalten die einzelnen BPMN-Elemente, wie z.B. Ereignisse, Tasks oder Gateways. Ein Verbund mehrerer Bahnen ist in 3.5 dargestellt.

Abbildung 3.5: Swimlanes

- Pool:
  Ein Pool, s. Abb. 3.6, umfasst mehrere Bahnen und ist eine übergeordnete Gruppierung von Bahnen. Elemente werden nicht dem Pool selbst zugeordnet, sondern den darin enthaltenen Bahnen. Die Verwendung von Pools ist optional, eine Gruppierung der Bahnen für ein gültiges BPMN-Modell ist nicht zwingend.

Abbildung 3.6: Pool

3.2.4 Artefakte

Artefakte (Artifacts) sind Elemente von BPMN, die in keine der obigen Gruppen einzuordnen sind.

- Datenobjekt:
  Ein Datenobjekt (Data Object, s. Abb. 3.7) stellt von außen bereitgestellte oder durch einen Vorgang produzierte Daten dar. Es ist somit einem BPMN-Element anhängig und hat keine eigenständige Rolle. Sie werden durch lose Verbindungen angehängt. Bei einem Einkaufsvorgang ist z.B. die Bestellliste ein Dateneingang, der als Datenobjekt modelliert wird.

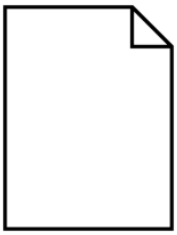


Abbildung 3.7: Datenobjekt

- Gruppe:
  Eine Gruppe (Group), s. Abb. 3.8, ist eine symbolische Zusammenstellung mehrerer BPMN-Elemente. Sie dient der Dokumentation oder Analyse, hat jedoch keinen Einfluss auf die Prozessabfolge.

Abbildung 3.8: Gruppe

- Anmerkungen:
  Anmerkungen (Annotations) sind ein wichtiges Dokumentationshilfsmittel und dienen der Beschreibung und Funktionserläuterung von BPMN-Elementen. Sie werden mit losen Verbindungen angehängt. Ihr Symbol ist in Abb. 3.9 abgebildet.

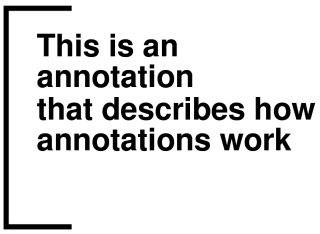


Abbildung 3.9: Anmerkung

Um die Elementtypen mit einem Graph $G = (V, E)$ verknüpfen zu können, definieren wir einen *BPMN-Graphen.*

Definition 5 (BPMN-Graph):
Ein BPMN-Graph $G = (V, E)$ ist ein Graph mit folgenden Zusatzinformationen:

- *Eine Abbildung $note\_type : V \to T$, wobei T die Menge der möglichen BPMN-Elementtypen für Knoten darstellt.*

- *Eine Abbildung edge_type : $E \rightarrow C$, wobei C die Menge der möglichen Verbindungsobjekte ist, s. Kapitel 3.2.2.*

Das BPMN-Konstrukt der Gruppen wird in dieser Arbeit nicht weiter verwendet. Auch Sub-Prozesse werden aufgrund der Komplexität beim Layout, die durch ineinander verschachtelte Prozesse verursacht wird, zum jetzigen Zeitpunkt noch nicht unterstützt. Bahnen, siehe 3.2.3 wiederum werden zur Modellierung herangezogen und vom automatischen Layout voll unterstützt. Dadurch werden keine großen semantischen Beschränkungen eingegangen, indem wir Gruppen für das automatische Layout nicht nutzen. Alle anderen oben genannten BPMN-Elemente können in ihrer vollen Funktionalität benutzt werden.
Semantische Überprüfungen der Elemente werden nicht vorgenommen. Dies ist auch ohne eine Implementierung, die Simulation und ein komplexes Regelwerk umfasst, nicht möglich. So sind dazu die ähnlichen Probleme bei der Simulation, die auch bei UML-Aktivitätsdiagrammen auftreten zu beachten, s. [SF07].

4 Das TSM-Framework und der Sugiyama-Ansatz

4.1 TSM-Framework

TSM steht für ***T**opology, **S**hape, **M**etrics*. Das Framework besteht aus diesen drei Phasen, welche hier genauer beschrieben werden.

Ziel des TSM-Framework ist ein orthogonales Layout des Eingangsgraphen $G = (V, E)$. Sollte der Eingangsgraph nicht zusammenhängend sein, so wird für die Komponenten einzeln ein Layout berechnet und dann zum Gesamtgraph zusammengefügt.

Das hier zugrunde liegende TSM-Framework beruht auf Ergebnissen und Vorarbeiten von [Tam87], [Eig03] und [Sie03].
Außerdem benutzen wir im TSM-Framework das Kandinsky-Modell, das von Fössmeier und Kaufmann in [FK95] vorgestellt wurde.

4.1.1 Topologie

In der ersten Phase des TSM-Framework wird eine Topologie (topology) des Graphen, also die Anordnung der Graphenelemente, berechnet. Die Topologie heißt bei Graphenlayouts auch „Einbettung“, wie in Kap. 2.2 erläutert.

Definition 6 (topologisch isomorph):
Zwei planare Graphen G_1 und G_2 haben die gleiche Topologie, sind also topologisch isomorph, wenn eine Umformung von G_1 in G_2 oder umgekehrt ohne eine Änderung der Faces existiert.

In der planaren Repräsentation P (planar representation) werden die Informationen über die Topologie gespeichert, die aus der Einbettung abgeleitet wird. Dazu gehören Daten von Knoten, Kanten, Kreuzungen und Faces. Kreuzungen werden bei nichtplanaren Graphen durch Dummy-Knoten ersetzt, die nach der Kompaktierungsphase wieder entfernt werden.
Ziel der Topologie-Phase ist die Berechnung einer Einbettung mit möglichst wenigen Kreuzungen (Dummy-Knoten).

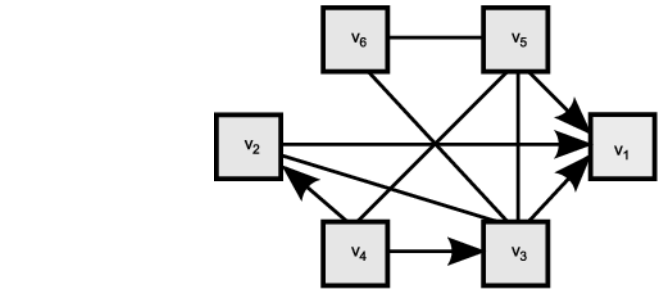
Die Planarisierung wird, wie in [Eig03] vorgestellt, wie folgt berechnet:

1. Wir gehen von einem maximalen planaren Subgraph $G' = (V, F)$ mit $F \subseteq E$ aus. Dazu werden alle Mehrfachkanten und Selbstschleifen entfernt. Die Kantenmenge E wird in gerichtete Kanten E_D und ungerichtete E_U unterteilt. Die Berechnung von G' ist NP-vollständig, es existieren jedoch mehrere Heuristiken. Mit einer in [GT94] vorgeschlagenen Heuristik wird die Berechnung von G' zu einem Finden der größten unabhängigen Kantenmenge (MIS, maximum independent set). Die Heuristik wird nach ihren Entwicklern Goldschmidt und Takvorian auch GT-Heuristik genannt. In der ersten Phase wird eine Knotenfolge berechnet, in die die Kanten dann ober- oder unterhalb einsortiert werden, sodass möglichst keine Überschneidungen entstehen.
   Der Aufbau der Knotenfolge ist nicht zufällig, sondern sollte einen Hamilton-Pfad induzieren. Da die Berechnung des Hamilton-Pfades ebenfalls NP-vollständig ist, benutzt man folgende Variante aus [RR97]: Die Knotenfolge beginnt mit dem Knoten kleinsten Grades. Es folgen jeweils die Nachbarknoten kleinsten Grades. Die Berechnung ist nicht eindeutig, sodass die Randomisierung darin besteht, bei nichteindeutiger Auswahl von Knoten kleinsten Grades einen Kandidaten zufällig zu wählen. Es wird also eine topologische Sortierung berechnet.
   Für die Kanten wird nun eine Gewichtsfunktion $w : E \to \mathbb{N}$ definiert, wobei die gerichteten Kanten ein höheres Gewicht erhalten, da sie im zweiten Schritt ungleich schwerer einzufügen sind als die ungerichteten. Der Subgraph G' enthält so mehr gerichtete Kanten als bei ungewichteter Berechnung. In [AIM91] wird ein Algorithmus für das gewichtete MIS-Problem beschrieben.
   Die Laufzeit für die GT-Heuristik beträgt für ein $k \in \mathbb{N}$, beliebig, aber konstant, auch bei k-maliger Wiederholung $O(|V||E^2|)$. Bei Wiederholungen werden die Gewichtssummen der Kantenmenge F verglichen und das Maximum gewählt.
2. In einem zweiten Schritt werden die Restkanten $R = (E \backslash F)$ in den Subgraphen G' eingefügt. Dabei sollten natürlich möglichst wenige Kreuzungen entstehen. Dieser Einfügevorgang wird *Kantenrouting* genannt und unterscheidet sich für gerichtete und ungerichtete Kanten. Eine ausführliche Beschreibung findet sich in [EKE00].

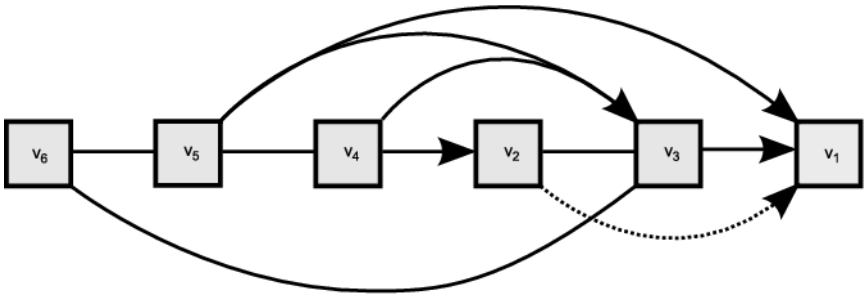
   - Gerichtete Kanten $R_D \subseteq R$:
     Wir konstruieren einen zu G' ähnlichen st-Graphen G_{st} mit $G' \subseteq G_{st}$. Die beiden Graphen unterscheiden bei der Konstruktion nur sog. Augmentierungskanten, die die Existenz von mehreren Quellen und Senken ausschließen.

     Definition 7 (st-Graph):
     Ein st-Graph ist ein gerichteter, azyklischer Graph $G_{st} = (V_{st}, E_{st})$ mit genau einer Quelle s und einer Senke t, wobei $\delta_{in}(s) = 0$ und $\delta_{out}(t) = 0$.

     Mit der Knotenfolge der GT-Heuristik und den Augmentierungskanten wird eine Levelisierung der Knoten von G' konstruiert. Die Augmentierungskanten werden nach dem Kantenrouting der gerichteten Kanten wieder entfernt. Für die Levelisierung wird eine Funktion $level : V \to \mathbb{N}$ definiert. *level* gibt zu jedem Knoten v die Höhe an, also die maximale



(a)



(b)

Abbildung 4.1: Beispiel für die GT-Heuristik. In (a) ist ein Beispielgraph abgebildet, für den mittels der GT-Heurisitk die in (b) dargestellte Knotenfolge und Verteilung der Kanten berechnet wurde.

Pfadlänge von der Quelle s zu v.
Ungerichtete Kanten werden temporär gerichtet. Die Richtung ergibt sich aus der Knotenfolge aus Schritt 1, die eine topologische Sortierung darstellt, damit keine Zykel entstehen. Für die Berechnung der Höhe müssen alle Kanten, auch die, die nicht in G_{st} enthalten sind, berücksichtigt werden.
Nun wird ein Routinggraph $G_{Routing} = (V_{Routing}, E_{Routing})$ und eine Kostenfunktion $cost : E_{Routing} \rightarrow \{0, 1\}$ definiert.

Definition 8 (Routinggraph):
Ein Routinggraph $G_{Routing} = (V_{Routing}, E_{Routing})$ und die Kostenfunktion $cost : E_{Routing} \rightarrow \{0, 1\}$ werden aus einem st-Graph G_{st} erzeugt und der Routinggraph enthält folgende Knoten $V_{Routing}$ und die gerichteten Kanten $E_{Routing}$:

- *Für jeden Level i und jedes innere Face f wird ein Knoten $v_{f,i}$ erzeugt. Die Knoten in f werden durch Kanten $(v_{f,i}, v_{f,i+1})$ mit Kosten 0 verbunden.*
- *Das äußere Face wird in eine rechte und eine linke Hälfte, $f_{out,l}$ und $f_{out,r}$, geteilt und erhält pro Level i jeweils einen Knoten, $v_{f_{out,r},i}$ und $v_{f_{out,l},i}$. Die aufeinanderfolgenden Knoten von $f_{out,l}$ und $f_{out,r}$ werden nun mit Kanten der Kosten 0 verbunden.*
- *Für eine Ebene i und zwei benachbarte Faces f und g werden nun*

die Kanten $(v_{f,i}, v_{g,i})$ *und* $(v_{g,i}, v_{f,i})$ *eingefügt. Falls die abgrenzende Kante eine Originalkante aus* G_{st} *ist, so erhalten die Kanten Kosten* 1*, bei einer Augmentierungskante Kosten* 0*, weil die Augmentierungskante nur eine Hilfskante ist und nach dem gesamten Einfügevorgang wieder entfernt wird.*

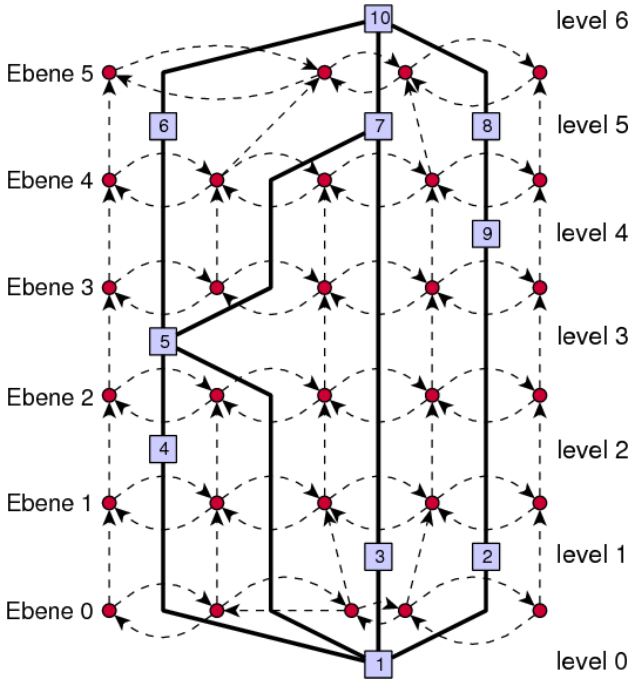


Abbildung 4.2: Ein st-Graph und und sein zugehöriger Routinggraph, aus [Eig03].

Der Verlauf einer einzufügenden Kante $e \in R_D$ über eine Originalkante $e_{st} \in G_{st}$ entspricht einer späteren Kreuzung und wird deshalb mit höheren Kosten bewertet.
Um nun eine Kante $e = (v, w) \in R_D$ einzufügen, müssen noch die Hilfsknoten v' und w' in $G_{Routing}$ konstruiert werden. Knoten v' wird mit allen Knoten aus $G_{Routing}$ verbunden, die auf Level $level(i)$ in einem an v angrenzenden Face liegen. Knoten w' erhält Kanten zu allen entsprechenden Knoten auf Level $level(w) - 1$. Alle für v' und w' einzufügenden Kanten erhalten Kosten 0. Die Kante e wird entlang des billigsten Weges von v' nach w' in G_{st} eingefügt. Nach dem Einfügen der Kante in G_{st} werden v' und w' wieder entfernt und der Routinggraph $G_{Routing}$ muss aktualisiert werden. Das Finden eines billigsten Pfades in einem planaren Graphen ist in $O(|V| + c)$ möglich, wobei c der Gesamtzahl der Kantenkreuzungen entspricht [KRRS94].
Die gesamte Laufzeit für das Einfügen der ungerichteten Kanten beträgt $O(|R_D| \cdot (|V| + c)^2)$.

- Ungerichtete Kanten $R_U \subseteq R$:

Das Einfügen der ungerichteten Kanten ist ungleich einfacher als das der gerichteten Kanten, da keine Richtungen zu beachten sind. Daher wird hierzu das Verfahren des *Dual-Graph-Routing* verwendet.

Definition 9 (Dualer Graph):
Sei $G = (V, E)$ ein planar eingebetteter Graph und F die Menge der Faces von G. Für den dualen Graphen $G_D = (V', E')$ von G gilt:

- *Für jedes Face $f \in F$ existiert ein Knoten in V'.*
- *Für jede Kante $e \in E$ existiert eine ungerichtete Kante in E' zwischen den beiden Knoten aus V', die die durch die Kante e getrennten Faces repräsentieren.*

Der zu G duale Graph G_D ist also planar, zusammenhängend und ungerichtet.

Im dualen Graphen wird nun wieder eine Kostenfunktion definiert, wobei alle Kanten aus E' Kosten 1 zugewiesen bekommen. Die Hilfsknoten v' und w' werden analog wie beim Einfügen der gerichteten Kanten mit den Knoten der umliegenden Faces mit Kosten 0 verbunden. Der billigste Weg von v' nach w' entspricht nun dem Verlauf der einzufügenden, ungerichteten Kante. Die Laufzeit für das Einfügen der Kante R_U beträgt $O(|R_U| \cdot (|V| + c))$.

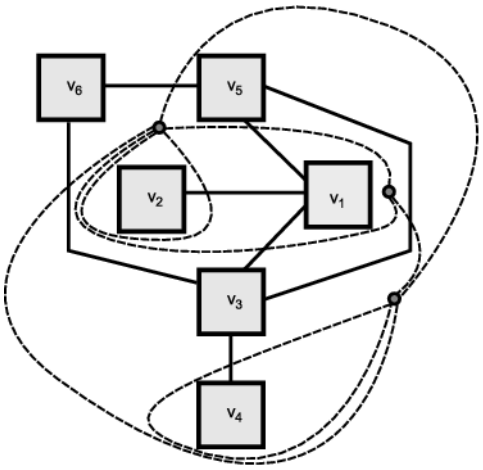


Abbildung 4.3: Ein planarer Graph und sein dualer Graph (gestrichelte Kanten und Faceknoten).

Nach dem Kantenrouting erhalten wir einen Graphen, der eine Erweiterung von G darstellt, da er zusätzlich die entstandenen Kreuzungsknoten enthält. Die Gesamtlaufzeit für das Kantenrouting beläuft sich auf $O(|V||E|^2 + |E| \cdot (|V| + c)^2))$, siehe [Eig03].

4.1.2 Form

In der zweiten Phase des TSM-Framework wird die Form (shape) berechnet. Aus der ersten Phase wurde bereits eine planare Repräsentation P erhalten, die nun zu einer orthogonalen Repräsentation H erweitert wird. Dazu müssen die Knicke (bends) der Kanten und die Winkel zwischen den Kanten an den Knoten bestimmt werden.
Ziel dieser Phase ist es, eine Form des Graphen G und eine orthogonale Repräsentation H mit möglichst wenigen Kantenknicken zu berechnen.

Definition 10 (Formisomorphie):
Zwei Graphen G_1 und G_2 mit zugehöriger orthogonaler Repräsentation H_1 und H_2 haben die gleiche Form, sind also formisomorph, wenn eine Umformung von G_1 in G_2 oder umgekehrt allein durch Veränderung der Kantenlängen und Knotengrößen existiert.

Bei orthogonalen Layouts haben die Knicke der Kanten in der Repräsentation H immer Werte, die Vielfache von 90 Grad sind, also Winkel $|w| = (x + 1) \cdot 90$ mit $x = 0 \ldots 3$.

Zur Bestimmung der orthogonalen Repräsentation H wird eine Erweiterung des in [Tam87] vorgestellten Algorithmus benutzt, der auf dem Lösen eines Min-Cost-Flow-Problems beruht. Der Algorithmus erfüllt folgenden Satz.

Satz 2 (Tamassia-Algorithmus):
Für einen planaren 4-Graphen G mit n Knoten und einer planaren Repräsentation P findet der Tamassia-Algorithmus eine knickminimale orthogonale Repräsentation H in der Zeit $O(n^2 \log n)$.

Der Beweis zu Satz 2 kann in [Tam87] nachvollzogen werden. In [GT96, Nor97] wird gezeigt, dass für den *Tamassia-Algorithmus* auch in $O(n^{7/4} \log n)$ eine Lösung berechnet werden kann.
Im Folgenden werden zuerst die Abbildung auf das Min-Cost-Flow-Problem und dann die für unsere Anwendung notwendige Erweiterung durch das Kandinsky-Modell beschrieben.

Min-Cost-Flow-Network

Die Abbildung von einem planaren 4-Graphen $G = (V, E)$ mit planarer Repräsentation P auf ein Netzwerk $N(P) = (V_N, E_N)$ wird durch die Definition von Kosten und Kapazitäten auf den Kanten des Netzwerks realisiert. In diesem Netzwerk sollen nun minimale Flusskosten (*min-cost-flow-network*) berechnet werden.

Definition 11 (Min-Cost-Flow-Network):
Ein Netzwerk mit minimalen Flusskosten $N(P) = (V_N, E_N)$, im Folgenden auch „Netzwerk“ genannt, besitzt die Knotenmenge V_N mit $V_N = U_V \cup U_F \cup \{s, t\}$ mit folgenden Eigenschaften:

- *Die Knoten s und t sind Quelle bzw. Senke des Netzwerks.*
- *Die Knotenmenge U_V enthält einen Knoten u_v für jeden Knoten $u \in V$.*
- *Die Knotenmenge U_F enthält einen Knoten u_f für jedes Face f von G.*

Die Kantenmenge E_N besteht aus Kanten, die definiert sind durch:

- *Kanten (s, u_v) mit $u_v \in U_V$. Diese Kanten haben Kosten 0 und die Kapazität $4 - \delta(v)$.*
- *Kanten (s, u_f) mit $u_f \in U_F$, wobei f ein inneres Face von G mit $\delta(f) \leq 3$ ist. Diese Kanten haben Kosten 0 und die Kapazität $4 - \delta(f)$.*
- *Kanten (u_f, t) mit $u_f \in U_F$, wobei f entweder das äußere Face von G oder ein inneres Face von G mit $\delta(f) \geq 5$ ist. Diese Kanten haben Kosten 0 und die Kapazität $\delta(f)+4$, falls f das äußere Face ist und sonst $\delta(f) - 4$.*
- *Kanten (u_v, u_f) mit $u_f \in U_F$, $u_v \in U_V$, wobei v ein Endpunkt einer Kante aus $P(f)$ ist. Diese Kanten haben Kosten 0 und die Kapazität ∞.*
- *Kanten (u_f, u_g) mit $u_g, u_f \in U_F$ und die Faces f und g besitzen mindestens eine gemeinsame Kante. Besitzt ein Face eine Brücke so entsteht eine Selbstschleife (u_f, u_f). Diese Kanten haben Kosten 1 und die Kapazität ∞.*

Bei der Interpretation des Flusses im Netzwerk sind nun die Übergänge (u_f, u_g) zwischen zwei Faces f und g und der Fluss (u_v, u_f) an den Knoten v von G zu beachten. Ein Fluss an (u_f, u_g) stellt einen Knick einer Kante dar, die die beiden Faces voneinander abgrenzt. Der Knick ist ein 90-Grad-Winkel im Face f. Wenn mehrere abgrenzende Kanten existieren, so kann der Knick einer beliebigen zugewiesen werden. Ein Fluss von x über (u_v, u_f) stellt einen Winkel von $(x+1)\cdot 90$ Grad zwischen den beiden Kanten von $P(f)$, die zu v inzident sind, dar. Ein Fluss von 0 bedeutet also immer einen rechten Winkel zwischen zwei benachbarten Kanten an einem Knoten. Die Summe der Winkel um einen Knoten muss immer 360 Grad betragen.
Ein gültiger Fluss mit Kosten k führt also zu einer orthogonalen Repräsentation H mit k Knicken. Wenn k minimal wird, so wird auch die Anzahl der Knicke minimal. Die Lösung des Problems bietet der Tamassia-Algorithmus aus Satz 2, siehe [Tam87].

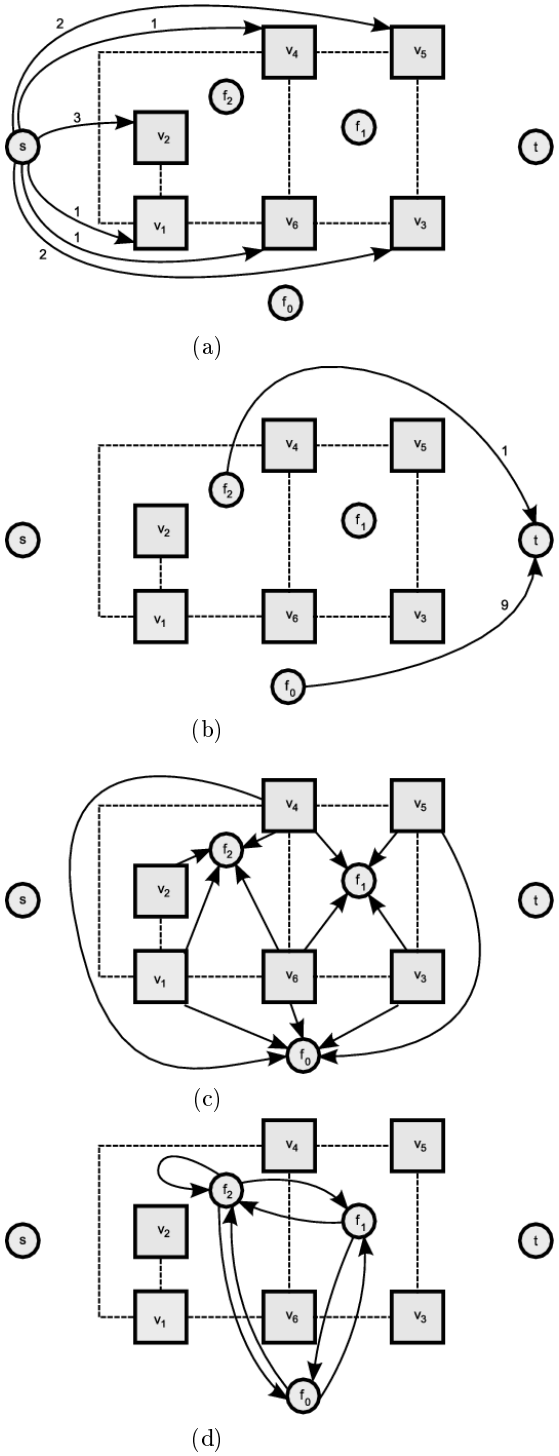


Abbildung 4.4: Beispiel-Konstruktion eines orthogonalen Netzwerkes wie in Abschnitt 4.1.2 beschrieben.

Kandinsky-Modell

Der auf Seite 26 vorgestellte Tamassia-Algorithmus behandelt als Eingabegraphen nur 4-Graphen. Wenn der Grad eines Knotens größer wäre, so würde dies zu Überlappungen führen, da der Knoten nur eine Kante pro Seite haben dürfte und die Anordnung von zwei Kanten auf der gleichen Seite nicht definiert wäre.
Das Kandinsky-Modell, wie in [FK95] konstruiert, bietet uns die Möglichkeit, das Netzwerk so zu erweitern, dass wir auch für allgemeinere Graphen eine Lösung des Min-Cost-Flow-Problems erhalten. Für das Kandinsky-Modell benötigen wir folgende

Definition 12 (Quasi-orthogonal shape):
Sei $G = (V, E, F)$ ein planarer Graph, wobei F die Menge der Faces ist. Eine quasi-orthogonale Form (quasi-orthogonal shape) Q ist eine Abbildung von F auf im Uhrzeigersinn geordnete Listen von Tupeln (e, b, a). Der erste Eintrag des Tupels entspricht einer Kante im Face. Der zweite Eintrag stellt einen Bit-String mit den Knicken der Kante, siehe Definition 4, dar und der dritte Eintrag ist der ganzzahlige Wert des Vielfachen von 90 Grad, dem der Winkel der Kante zu ihrem Vorgänger im Face entspricht, also $0 \leq a \leq 4$.

Mit der quasi-orthogonalen Form Q können wir die beiden Eigenschaften für eine gültige Form im Kandinsky-Modell konstruieren. Die beiden Eigenschaften sind die Knick-oder-Ende-Eigenschaft (bend-or-end property) und die Nichtleeres-Face-Eigenschaft (non-empty-face-property).

Definition 13 (bend-or-end property):
Gegeben sei $G = (V, E, F)$, ein planarer Graph. Eine quasi-orthogonale Form Q von G hat die Knick-oder-Ende-Eigenschaft (bend-or-end property), falls für jede zwei Kanten (u, v) und (v, w), die in einem Face $f \in F$ aufeinander folgen entweder der letzte Knick der Kante (u, v) oder der erste Knick der Kante (v, w) konkav ist.

Definition 14 (non-empty face property):
Gegeben sei $G = (V, E, F)$, ein planarer Graph und Q eine quasi-orthogonale Form von G. Sei f ein Face mit drei Kanten.
Das Face $f = ((u, v), (v, w), (w, u))$ heißt

- *L-triangular, falls*
  $Q(f) = \{((u, v), 1, 0), ((v, w), \varepsilon, 0), (w, u), \varepsilon, 1)\}$.
- *T-triangular, falls*
  $Q(f) = \{((u, v), 1, 0), ((v, w), 1, 0), (w, u), \varepsilon, 0)\}$.

Eine quasi-orthogonale Form Q hat die Nichtleeres – Face – Eigenschaft (non-empty face property), falls kein Face $f \in F$ L- oder T-triangular ist.

Um die Eigenschaften einer quasi-orthogonalen Form auf unser auf S. 26 definiertes Netzwerk zu übertragen, müssen wir es wie folgt erweitern:

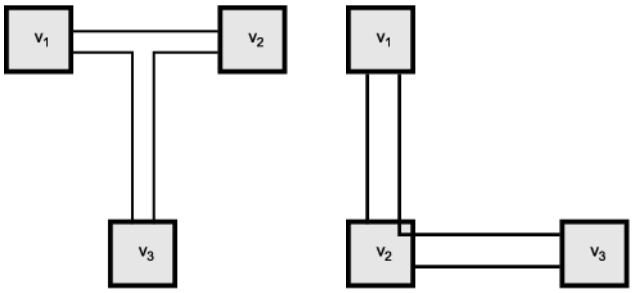


Abbildung 4.5: Beispielfälle für leere Faces, die eine gültige Zeichnung im Kandinsky-Modell nicht besitzen darf.

Für jede Kante e_i eines Knotens $v \in V$ werden zwei Knoten $H_{e_i}^{v,l}$ und $H_{e_i}^{v,r}$ und für jedes Face $f \in F$, das an v anliegt, ein Knoten H_f^v eingefügt. Zusätzlich werden abschließend noch folgende Kanten eingefügt:

- Kanten $(u_f, H_{e_i}^{v,r})$ und $(u_f, H_{e_j}^{v,l})$, wobei f das Face ist, das durch e_i und e_j begrenzt wird und e_j die Kante in f ist, die in Richtung des Knotens verläuft. Diese Kanten haben Kapazität 1 und Kosten $2 \cdot c + 1$.
- Kanten (H_f^v, u_v) mit Kapazität 1 und Kosten 0. Diese Kanten garantieren die Knick-oder-Ende-Eigenschaft.
- Kanten $(H_{e_i}^{v,l}, H_f^v)$ und $(H_{e_j}^{v,r}, H_f^v)$, mit f, e_i und e_j wie oben, mit Kapazität 1 und Kosten 0.
- Kanten $(H_{e_i}^{v,l}, H_{e_i}^{v,r})$ und $(H_{e_i}^{v,r}, H_{e_i}^{v,l})$ mit Kapazität 1 und Kosten $-c$. Diese Kantenpaare bilden Zykel mit den Kosten $-2c$, sodass ein Pfad (u_f, u_v) die Kosten $2c+1-2c = 1$ besitzt. Bei gesättigtem Zykel hat der Pfad Kosten $2c+1$.

Das entstandene Problem, in [Eig03] auch KANDINSKY BEND MINIMIZATION-Problem genannt, ist durch ein ILP (integer linear program) lösbar. Ob es in polynomieller Zeit zu lösen ist, ist nicht klar. Mit einer Heuristik, die die kürzesten Wege im Netzwerk berechnet, kann in $O(n^2 log(n))$ eine Lösung gefunden werden, welche jedoch nicht optimal sein muss. In [Eig03] wird eine 2-Approximation für das KANDINSKY BEND MINIMIZATION-Problem angegeben.

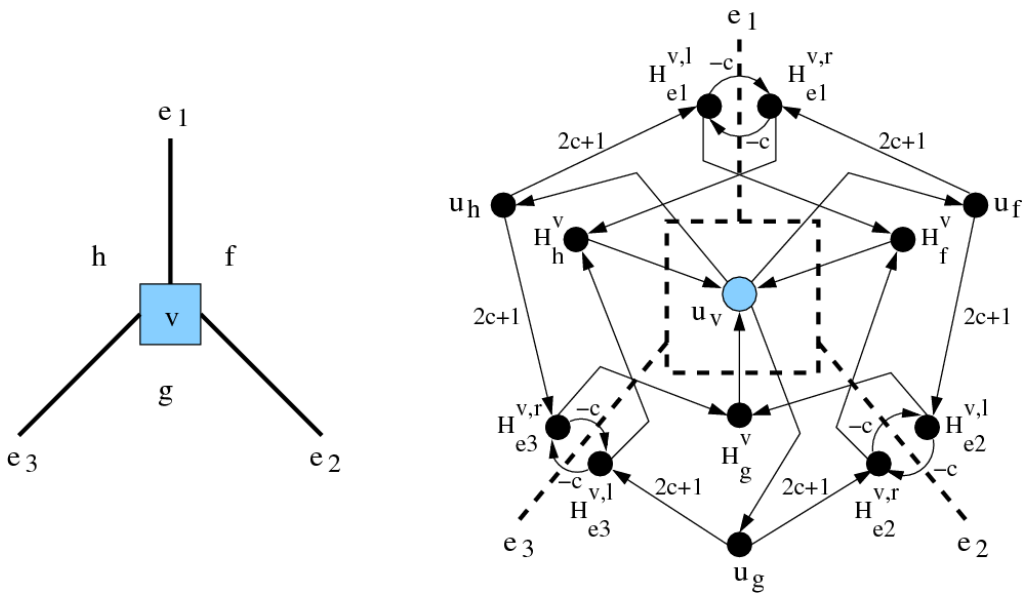


Abbildung 4.6: Das vollständige Knotenmodell im *min-cost-flow-network* eines Knotens v mit den umliegenden Faces f, g und h.

4.1.3 Metrik

In der letzten Phase des TSM-Frameworks wird die Metrik (metrics), also die Längen der Kanten und Größen der Knoten festgelegt.

Definition 15 (Metrikisomorphie):
Zwei Graphen G_1 und G_2 haben die gleiche Metrik, sind also metrikisomorph, wenn sie die gleiche Topologie und Form besitzen und eine Umformung existiert, die nur Translation und Rotation verwendet.

Hier heißt die Metrikphase auch „Kompaktierung“. Das Ziel der Kompaktierung ist die Minimierung der Fläche, die der Graph einnimmt. In der Kompaktierungsphase für 4-Graphen werden die Faces F der orthogonalen Repräsentation H normalisiert, indem die Faces durch zusätzliche Kanten und Knoten in Rechtecke zerlegt werden. Für die Rechtecke werden einzeln ganzzahlige Kantenlängen berechnet. So erhält man eine normalisierte, orthogonale Repräsentation H' des Graphen G. Die Laufzeit beträgt für dieses Verfahren $O(|V| + |B|)$, wobei B die Menge der Knicke ist.

Für das Kandinsky-Modell muss der Algorithmus noch Knoten höheren Grades und Knoten unterschiedlicher Größe behandeln können. Für Knoten unterschiedlicher Größe wird folgendes Verfahren verwendet:

> **Behandlung von Knoten unterschiedlicher Größe:** Ein Originalknoten v wird durch Rechtecke aus mehreren kleinen Knoten ersetzt, wodurch ein planarer 4-Graph entsteht, der mit einer Variante des oben genannten Algorithmus kompaktiert werden kann. Bei genügend feiner

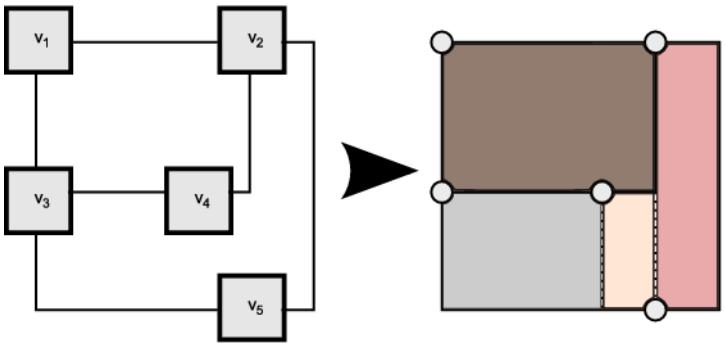


Abbildung 4.7: Beispiel für die Rechteckzerlegung in der Kompaktierungsphase

Zerlegung, also hoher Anzahl kleiner Knoten, können die Kanten, die vorher mittig am Knoten verliefen, auch wieder am mittleren Knoten der Seite angefügt werden, siehe Abbildung 4.8.

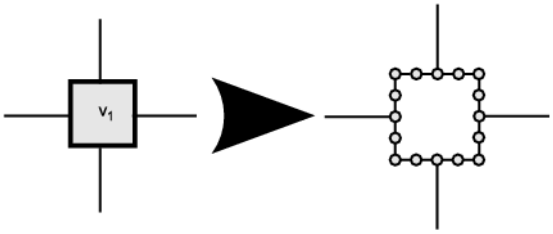


Abbildung 4.8: Darstellung für die Ersetzung eines großen Knotens durch kleinere (node splitting).

In [Eig03] wird ein Kompaktierungsalgorithmus für eine orthogonale Repräsentation Q vorgestellt, die das Kandinsky-Modell erfüllt. Der dort konstruierte Algorithmus findet die Kompaktierung in Linearzeit.

Für das Beschriften der Kanten (labeling) gibt es zwei alternative Wege. Der meistbegangene ist, dass das Beschriften der Kanten in einem Extraschritt nach der Kompaktierung vorgenommen wird. Dies ist begrenzt durch die nach der Kompaktierung noch vorhandene freie Fläche im Graphen. Sollten die Beschriftungen größere Flächen benötigen, kommt es zu unerwünschten Überlappungen. Daher werden in [KM99] und [BDLN01] Verfahren vorgestellt, in denen die Beschriftungen der Kanten bereits im Kompaktierungsschritt berücksichtigt werden.

4.2 Sugiyama-Ansatz

Der Sugiyama-Ansatz ist eine weitverbreitete Methode zur Konstruktion von Zeichnungen, die aus Schichten (layers) bestehen und als Eingabe gerichtete Graphen

haben. Das abstrakte Framework wurde in [STT81] vorgestellt und bringt seit 1999 auch eine Implementierung in Form einer Bibliothek zur Berechnung mit (SugiBIB, [Eic99]).

Das Sugiyama-Framework arbeitet in vier Phasen:

1. **Zykelentfernung (cycle removal):**
   Im ersten Schritt werden die Zykel im Eingabegraph entfernt, siehe Abbildung 4.9b. Einen azyklischen Graph erhält man, wenn man in jedem Zykel eine Kante in ihrer Richtung umkehrt, sodass der Zykel unterbrochen wird. Die Kante darf dadurch jedoch nicht Teil eines anderen Zykels werden, sonst muss eine andere Kante des Zykel umgekehrt werden.
2. **Schichtenzuweisung (layer assignment):**
   In der Phase der Schichtenzuweisung wird jedem Knoten eine horizontale Schicht zugewiesen, siehe Abbildung 4.9c. Die Schichten nennt man auch *Level*. Die Schichtenzuweisung ist gerichtet und erstellt damit eine topologische Ordnung der Knoten des Graphen: Für $L_1, \ldots, L_h$ und jede Kante $e = (v, w) \in E$ mit $v \in L_i$ und $w \in L_j$ gilt, dass $i < j$.
   Nach der Schichtenzuweisung der Knoten wird der entstandene Graph noch normalisiert, indem für lange Kanten (long edges), die über mehr als eine Schicht verlaufen, Ketten von Dummy-Knoten eingefügt werden. Die Dummy-Knoten sind für den nächsten Schritt der Kreuzungsminimierung notwendig, damit lange Kanten erkannt werden. Am Ende der Schichtenzuweisung erhält man den normalisierten Graph $N_G = (V_N, E_N)$.
3. **Kreuzungsreduzierung (crossing reduction):**
   Im normalisierten Graphen N_G wird nun in jeder Schicht die Ordnung der Knoten so verändert, dass die Anzahl der Kantenkreuzungen reduziert wird, siehe Abbildung 4.9d. Mit dem Kompaktierungsgraphen (compaction graph) $C_G = \{V_N, (a, b)\}$, wobei a, b in L_i, $1 \leq i \leq h$, aufeinanderfolgend sind, erhalten wir eine totale Ordnung der Knoten jeder Schicht. Kreuzungsminimierung ist im Allgemeinen NP-hart.
   Die Kreuzungsminimierung wird schichtweise (sweep-line) ausgeführt. Jeder Schritt minimiert also die Kreuzungen der Kanten zweier benachbarter Schichten. Der verwendete Algorithmus der Kreuzungsminimierung ist der sogenannte *one-sided two-layer crossing minimization*-Algorithmus.
4. **Zuweisung der horizontalen Koordinate:**
   Im letzten Schritt wird für jeden Knoten eine x-Koordinate berechnet, indem im Kompaktierungsgraph die Dummy-Knoten der langen Kanten möglichst vertikal ausgerichtet werden, siehe Abbildung 4.9e. Wenn die Dummy-Knoten untereinander liegen, enthält die resultierende lange Kante wenig Knicke. Zuletzt werden umgedrehte Kanten wieder in die ursprüngliche Richtung gekehrt und zusätzlich eingefügte Knoten wie die Dummy-Knoten entfernt, siehe Abbildung 4.9f.

   Auch die Komplexität des *one-sided two-layer crossing minimization* ist NP-hart [EW94], kann jedoch durch Heuristiken, die ein Maß für jeden Knoten ver-

Algorithm:one-sided two-layer crossing minimization

Data: normalized graph N_G
Output: compaction graph C_G

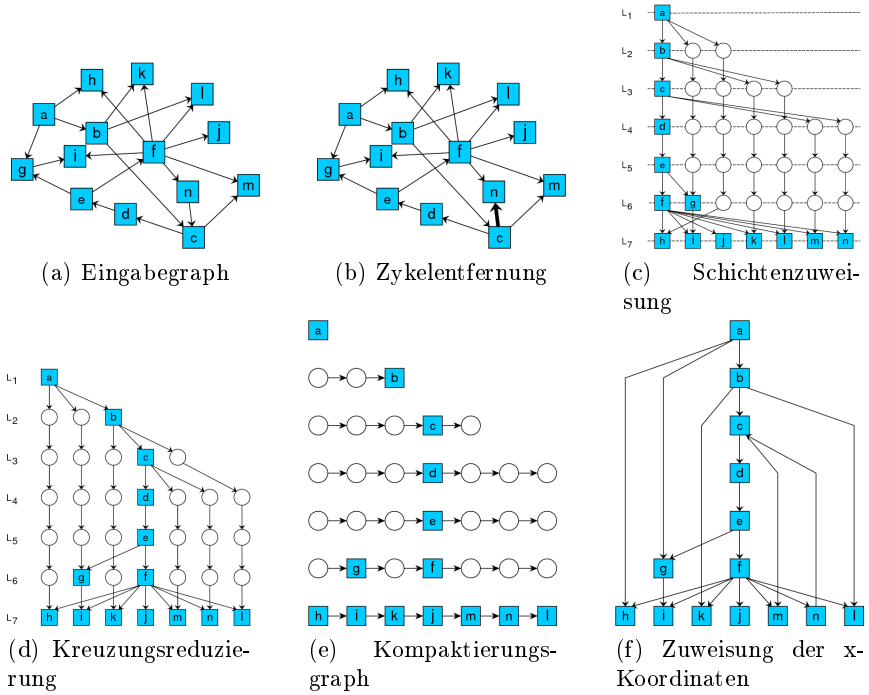
```
1  i := 1;
2  maxLayer := (#layers);
3  directionUp := true;
4  choose a random order for the first layer L_1
5  cross := (#crossings in graph);
6  while (true) do
7      keep order in L_{i-1} fixed
8      for (Nodes n,ne in L_i) do
9          if (swap(n,ne) reduces #crossings) then
10             swap(n, ne);
11     if (i == maxLayers) || (i == 0) then
12         directionUp = !directionUp;
13         if (#crossings == cross) then return ;
14         cross := (#crossings in graph);
15     if (directionUp) then i++ else i--;
```

wenden, verbessert werden. Als Maß für einen Knoten v kann z.B. der Schwerpunkt (Durchschnitt) [STT81] oder Median [EW94] der Nachbarn der oberen Schichten gebildet werden. Die Position erhält man dann, indem nach den Maßwerten sortiert wird. Die Komplexität des Sugiyama-Algorithmus hängt stark davon ab, wie viele Dummy-Knoten einzufügen sind. Nach [Fri97] können diese aber minimiert werden, liegen jedoch immer noch in der Ordnung von $\Theta(|V||E|)$. Die Gesamtkomplexität des heuristischen Ansatzes liegt also bei $O(|V||E|log|E|)$. In Abb.4.9 wird der Ansatz nach Sugiyama beispielhaft vorgeführt.

In dieser Arbeit werden beide obige Ansätze, also sowohl das TSM-Framework als auch der Sugiyama-Ansatz, verwendet und ihre Vorteile ausgenutzt. Das TSM-Framework eignet sich gut für Layouts, die orthogonal sein sollen und bei denen die Knotengrößen unterschiedlich und daher zu berücksichtigen sind, damit keine Überlappungen entstehen. Der Sugiyama-Ansatz spielt durch die schichtenweise Anordnung seine Vorteile in der Planarisierung aus. Der Gesamtablauf für das Layout eines Graphen ist somit folgender:

1. Berechnung einer Planarisierung mit Sugiyama
2. Berechnung einer gültigen orthogonalen Form und Konstruktion der orthogonalen Repräsentation
3. Kompaktierung der Zeichnung

Für BPMN-Layouts verwenden wir daher die Kombination, die auch in [SK06] an-



(a) Eingabegraph (b) Zykelentfernung (c) Schichtenzuweisung

(d) Kreuzungsreduzierung (e) Kompaktierungsgraph (f) Zuweisung der x-Koordinaten

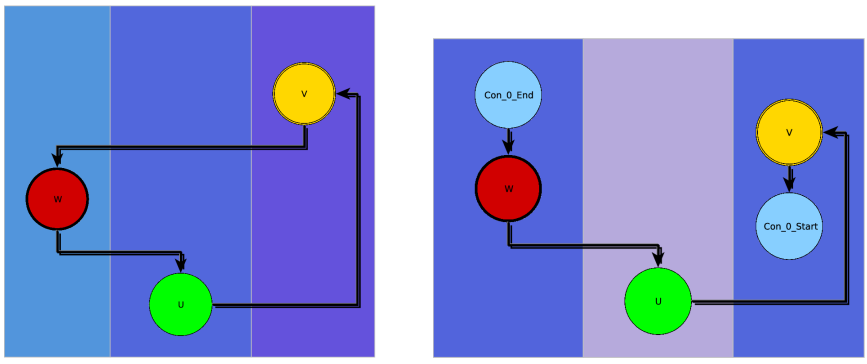
Abbildung 4.9: Beispiel für die Berechnung einer Schichtenzuweisung für einen einfachen Graphen, gefunden in [SK06].

gewandt wird. Zuerst erzeugt der Sugiyama-Ansatz eine planare Einbettung, dann wird mit dem TSM-Ansatz, also der Orthogonalisierung und Kompaktierung, fortgefahren. Zwischengeschaltet ist noch ein Rerouting, wie es oben im TSM-Framework beschrieben wurde für die Kanten, die während des Sugiyama verändert wurden (ungerichtete und in der Zykelentferung umgedrehte Kanten).

5 Konnektoren

In diesem Kapitel stellen wir das Konzept der Konnektoren vor. Dabei sind die hier vorgestellten Konnektoren nicht verwandt mit den BPMN-Verbindungsobjekten, siehe Kapitel 3.2.2, die manchmal ebenfalls als Verbinder (connectors) bezeichnet werden. Richtigerweise ist der Name der BPMN-Elemente jedoch Verbindungsobjekte (connections) und nicht mit den hier vorgestellten Konnektoren zu verwechseln.

Ein Konnektor ist ein Knoten eines durch eine Kante verbundene Knotenpaares, das als Repräsentant für eine gelöschte Originalkante steht. Dazu wird jeweils am Start- und Endpunkt der zu ersetzenden Kante ein Knoten (Konnektor) angebunden. Die neu angebundenen Knoten enthalten gegenseitig Links aufeinander, damit die Informationen der End- und Startpunkte der Originalkante erhalten bleibt.



(a) Original (b) mit Konnektoren

Abbildung 5.1: Beispiel für die Ersetzung der Kante (v, w) durch ein zusammengehöriges Paar von Konnektoren

Die Idee der Konnektoren ist es, den Graphen durch Ausdünnen übersichtlicher zu machen. Um den Graphen geschickt optisch auszudünnen, werden die schlimmsten (nach weiter unten erläuterteten Kriterien) Kanten berechnet und durch Konnektoren ersetzt. Die Ersetzung zielt darauf ab, schlimme Kanten durch zwei kurze Enden mit jeweils einem Konnektor auszutauschen.
Die Kriterien für die Berechnung der schlimmsten Kanten sind hier:

- Anzahl der Knicke der Kante
- Anzahl der Kreuzungen der Kante mit anderen Kanten
- Geometrische Länge der Kante in einer Zeichnung

Die Anzahl der Knicke und Kreuzungen einer Kante sind linear zu der Anzahl der Segmente in der orthogonalen Repräsentation. Eine Reduzierung der Segmente verkleinert somit auch die internen Datenstrukturen, die die orthogonale Repräsentation halten, und spart Speicherplatz.

Die Konnektoren werden erst nach einem Durchlauf durch das TSM-Framework eingefügt, da man für die Berechnung der Kriterien einen bereits kompaktierten Graph benötigt. Nach dem Einfügen der Konnektoren wird der Graph noch einmal kompaktiert, damit die frei gewordenen Flächen der gelöschten Kanten ausgenutzt und die Konnektoren ohne Überlappung platziert werden können.

Der Ablauf des TSM-Frameworks mit Konnektoren wird im Vergleich zu Kap. 3 verändert zu:

1. Berechnung einer Planarisierung mit Sugiyama
2. Berechnung einer gültigen, orthogonalen Form und Konstruktion der orthogonalen Repräsentation
3. Kompaktierung der Zeichnung
4. Berechnung der schlimmsten Kanten und Einfügen der Konnektoren
5. Wiederholte Kompaktierung der modifizierten Zeichnung

5.1 Automatische Bestimmung der Konnektorkandidaten

Die Kanten, die sich für eine Ersetzung durch Konnektoren eignen, lassen sich manuell oder automatisch auswählen. Die automatische Bestimmung geschieht durch Vorgabe einer Gewichtung von ästhetischen Kriterien, sodass möglichst viele oder auch alle schlimmen Kanten auf Knopfdruck durch Konnektoren ausgetauscht werden. Dieser Ansatz wird hier erläutert.

Die Berechnung der Kanten, die Kandidaten für eine Ersetzung durch Konnektoren sind, ergibt sich aus der Bestimmung ihrer gewichteten Bösartigkeit.

Definition 16 (Bösartigkeit einer Kante): *Die Bösartigkeit (badness)* $badness : E \to \mathbb{N}$ *einer Kante* $e \in E$ *in einer orthogonalen Repräsentation* H *ergibt sich aus dem Wert, der mit der Anzahl der Kreuzungen (crossings) von* e *mit anderen Kanten* $e' \neq e$*, der Anzahl der Knicke von* e *(bends) und der geometrischen Länge (Summe der einzelnen Segmentlängen, length) wie folgt berechnet wird:*

$$badness(e) = \alpha \cdot |crossings| + \beta \cdot |bends| + \gamma \cdot length$$

wobei $\alpha, \beta, \gamma \in \mathbb{N}$ *die Gewichte für die einzelnen Faktoren darstellen.*

Durch Verändern der Gewichtung kann die Berechnung der Bösartigkeiten auf die Eigenschaften eines Graphen angepasst werden.

Für die Bestimmung der Konnektorkandidaten in einem Graphen $G = (V, E)$ wird ein minimaler Spannbaum (minimum spanning tree, MST) mit den Bösartigkeiten *badness* als Kantengewichten konstruiert, s. Abb. 5.2. Ein Spannbaum ist notwendig, da für das TSM-Framework der Eingangsgraph immer zusammenhängend sein muss. Ansonsten wird der Graph als zwei unzusammenhängende Teilgraphen betrachtet, die separat berechnet würden. Da der Graph jedoch nach der Konnektorphase noch einmal kompaktiert wird, muss er zusammenhängend bleiben, um u.a. die Einhaltung der Swimlanes zu garantieren. Die Kanten des minimalen Spannbaums $E' \subseteq E$ fallen als Kandidaten für Konnektoren heraus, da sie notwendig sind, um den Graphen zusammenhängend zu erhalten und ihre Bösartigkeit auch kleiner ist als die der übrigen Kandidaten-Kanten $E_{cand} = E \setminus E'$.

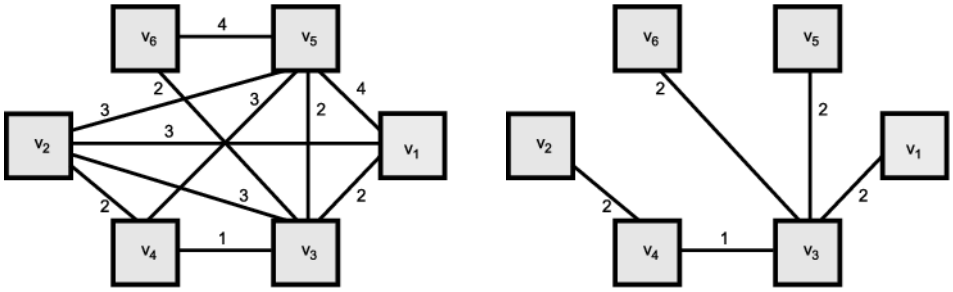


Abbildung 5.2: Beispiel für einen Graphen (a) und einen zugehörigen minimalen Spannbaum (b)

Aus der Menge der Kandidaten-Kanten können nun so lange Kanten durch Konnektoren ersetzt werden, bis eine vorgegebene Schranke s erreicht wird oder bis $E_{cand} = \emptyset$. Der Faktor s ist dabei begrenzt durch die Bedingung, dass G zusammenhängend sein muss; s ist also beschränkt durch $s \leq \frac{|E|-|E'|}{|E|} = \frac{|E_{cand}|}{|E|}$. Durch eine Sortierung nach der Bösartigkeit können bei vorgegebener Schranke die schlimmsten Kanten zuerst entfernt werden.
Nach jedem Ersetzungschritt, siehe Kap. 5.2, müssen die Bösartigkeiten und damit auch der MST neu berechnet werden, da die Entfernung einer Kante Einfluss auf den Wert der Bösartigkeit anderer Kanten hat, z. B. bei Kreuzungen. Dies hat erheblichen Einfluss auf die Laufzeit von $O(|V| \cdot |E| \cdot \log |E|)$, die sich aus der MST-Berechnung für jeden Ersetzungsschritt zusammensetzt, wobei die Ersetzungschritte beschränkt sind durch $s < |E|$. Wenn man mit der Beibehaltung eines statischen MST rechnen würde, so entstünden immer ungenauere Bösartigkeiten, je mehr Kanten entfernt würden, auch wenn man hier eine Laufzeit von $O(|V||E|)$, der Laufzeit der MST-Berechnung, erreichen könnte.

5.2 Löschen und Einfügen der Konnektoren

Beim Einfügen eines Konnektors in einen Graphen G müssen wir auch die quasi-orthogonale Repräsentation H modifizieren, da wir mitten im TSM-Framework auf einer gültigen Repräsentation arbeiten und diese nach Einfügen der Konnektoren wieder als gültige Repräsentation an den finalen Kompaktierungsschritt reichen.
Um ein gültiges Kandinsky-Modell weiterhin zu erhalten, müssen die Kandinsky-Eigenschaften aus Kap. 4.1.2 erfüllt sein. Bevor wir die Originalkante löschen, speichern wir folgende Informationen:

1. Winkel zur Nachfolgekante, jeweils am Anfangs- und Endpunkt
2. Verlauf der ersten und letzten beiden Segmente (jeweils 1 Bit aus Bitvektor)
3. Namen von Start- und Endpunkt

Die erste Information ist notwendig, um die Eigenschaft (3) der quasi-orthogonalen Repräsentation, siehe Def. 4, zu erhalten, die besagt, dass die Summe der Winkel um einen Knoten, die die Kanten zu ihrer Nachfolgekante beschreiben, immer 360 Grad ist. Die zweite Information macht die Einhaltung der Kandinsky-Eigenschaft „Knick-oder-Ende“ möglich, siehe Def. 13, indem der Verlauf der Originalkante bis zum zweiten Knick gespeichert wird. So kann die Konnektorkante mit dem gleichen Verlauf eingefügt können, falls die Originalkante nicht die Mittelkante war. Falls die Originalkante die Mittelkante und damit die einzige Kante des Knotens mit geradem Verlauf an dieser Seite war, so benötigt man nur das erste Bit, um den Verlauf des Konnektors zu bestimmen.

Da ein Konnektor jeweils zwei Knoten, einen am Start- und einen am Endpunkt der Originalkante bedingt, werden die beiden neuen Knoten mit einer Beschriftung oder einem Link auf den Partnerkonnektor versehen, damit kenntlich ist, welche Originalkante die beiden Konnektorknoten ersetzen. Dabei ist ein Partnerkonnektor immer das in einem Konnektorpaar zu einem Konnektorknoten gehörende Pendant. Die Beschriftung beinhaltet hier auch den Namen des Knotens mit dem der Konnektorknoten verbunden ist, also den ursprünglichen Start- bzw. Endknoten der Originalkante.

Beim Löschen der Kanten (bzw. Kantensegmente) wird darauf geachtet, dass die interne Datenstruktur der quasi-orthogonalen Form konsistent gehalten wird, da sonst eine fehlerfreie Weiterverarbeitung nicht möglich ist.

5.3 Auswertung der Konnektoren

Um Aussagen über die Performanz treffen zu können, wurde das existierende Statistiktool um die Werterfassung der Konnektoren erweitert. Es werden Knotenanzahl- und Kantenanzahlreduzierungen erfasst, wie auch Zeitmessungen festgehalten.

Aus Tabelle 5.1 ist ersichtlich, dass Konnektoren die Anzahl der Knoten und Kanten in der quasi-orthogonalen Repräsentation stark reduzieren und damit die zweite Kompaktierungsphase beschleunigen. Viele der Knoten und Kanten, die entfernt wurden, waren in der Zeichnung unsichtbar, da Dummy-Knoten, die für Kreuzungen stehen, nicht gezeichnet werden. Für jeden gelöschten Knoten werden auch die zugehörigen Kanten entfernt, darum ist die Anzahl der gelöschten Kanten um ein Vielfaches höher bei dichteren Graphen im Vergleich zu dünneren. Stark dominierend bei den Laufzeiten ist neben der Orthogonalisierung auch die Berechnungen der bösartigen Kanten, siehe Anmerkungen zur Laufzeit in Kap. 5.1.

				Laufzeiten / ms					Kanten			Knoten		
Knoten	Kanten	Kreuz.	Knicke	gesamt	Orthogon.	1. Kompakt.	Konnekt.	2. Kompakt.	vorher	nachher	Reduz./%	vorher	nachher	Reduz./%
20	60	14	96	1299	549	88	443	45	1696	504	70,28	534	230	56,93
20	100	21	180	4179	1274	94	2203	71	4096	808	80,27	1252	390	68,85
35	90	45	140	2497	571	91	1546	44	3444	780	77,35	1060	346	67,36
35	110	36	199	5451	1628	138	3168	76	5324	1052	80,24	1619	465	71,28
35	230	2	418	109445	41840	626	57919	166	21518	1880	91,26	6336	903	85,75
50	120	42	187	4565	1047	168	2709	49	4882	1260	74,19	1485	515	65,32
50	240	16	438	96512	24317	661	62697	191	22616	2134	90,56	6664	986	85,20
65	131	37	245	6600	2179	180	3569	91	6278	1710	72,76	1949	671	65,57
80	164	58	349	15505	4747	254	9488	137	10052	2520	74,93	3018	957	68,29
80	240	56	477	78475	19204	585	53481	202	21970	2954	86,55	6569	1220	81,43

Tabelle 5.1: Tabelle über Laufzeiten des Layoutalgorithmus auf Zufallsgraphen. Die Reduzierungen (rechts in der Tabelle) zeigen, dass das Einfügen von Konnektoren die Datenstruktur der orthogonalen Repräsentation reduziert.

6 Sketch-Driven Layout

6.1 Idee des Skizzenentwurfs

6.1.1 Motivation

Die Motivation für ein skizzenbasiertes Layout (sketch-driven-layout) lautet wie folgt:

> *The objection of sketch-driven-layout is to keep changes between consecutive frames to a minimum in order not to destroy the viewer's mental map*
> [Das Ziel des skizzenbasierten Layouts ist es, Änderungen von aufeinanderfolgenden Zeichnungen auf ein Minimum zu beschränken, um das geistige Bild des Betrachters nicht zu zerstören.]

Die Vorstellung des Betrachters/Benutzers wie ein Layout aussehen kann, soll beim skizzenbasierten Layout also berücksichtigt werden. Ein weiterer Vorteil dieses Vorgehens ist, dass der Benutzer die Kontrolle über den Layoutentwurf teilweise behält, indem er die Skizze entwirft und damit grobe Maßgaben für das spätere Layout liefert.

Eine Skizze (sketch) ist hier eine Zeichnung, die die Hauptfunktionen erfüllt, die der Benutzer wünscht, jedoch nach ästhetischen Aspekten ungenügend ist.
Es gibt zwei Varianten von Anwendungsfällen, in denen skizzenbasiertes Layout sinnvoll ist. Diese sind:

- Der Benutzer gibt einen groben Entwurf des Layouts vor. Das automatische Layout übernimmt dann die Aufgabe, die ästhetischen Bedingungen zu erfüllen und erstellt ein fertiges Layout.
- Ein Layout ist bereits vorhanden und soll verändert werden. Änderungsaktionen sind z.B. Einfügen, Löschen oder Verschieben von Graphelementen, sowie das Ändern von Zuordnungen (z.B. ein Knoten wird einer anderen Bahn zugeordnet)). Dieser Ansatz ist sehr vielversprechend, da hier die **Interaktivität** durch den Benutzer auf dem Weg zu einem finalen Layout und auch spätere Änderungen ermöglicht werden. Dies ist auch Teil des dynamischen Graphenzeichnens (darüber gibt es in [Bra01] eine Übersicht).

6.1.2 Kandinsky und das Bayesian Framework

Dem skizzenbasierten Entwurf liegen zwei theoretische Ansätze zugrunde. Das Kandinsky-Modell ([FK95]) wurde bereits in Kap.4.1.2 vorgestellt. Um nun über ein Maß für Veränderungen zwischen Graphen zu verfügen, wird das Bayesian Framework [Bra99] verwendet. Es bietet eine Unterschiedsmetrik (difference metric) für

das dynamische Graphenzeichnen, siehe [BT98], und wird nun als Maß (penalty) für eine Zielfunktion (objective function) betrachtet. Mit dieser Zielfunktion, die Veränderungen in Winkeln und Knicken als Unterschiedsmaß berücksichtigt, können im Folgenden das Optimierungsproblem für den skizzenbasierten Entwurf, wie in [BKWE02] formuliert, definiert werden.

Formales Ziel des Sketch-Driven Layout: Bestimmung einer orthogonalen Zeichnung eines Graphen G_Σ mit den Eigenschaften:

- der Ausgangsgraph G_Σ ist topologisch isomorph zum Eingangsgraphen
- die Zeichnung von G_Σ erfüllt die Kandinsky-Eigenschaften (siehe Definitionen 13 und 14)
- die Winkel in der Zeichnung weichen nur wenig von denen der Skizze ab (Stabilität)
- die Zeichnung enthält nur wenig Knicke (Lesbarkeit)

6.2 Algorithmus

Der Ablauf des Algorithmus ist ähnlich zum TSM-Framework aus Kap. 4:

1. Planarisierung unter Berücksichtigung der Skizze
2. Berechnung einer quasi-orthogonalen Form:
   Die Winkel werden zum nächsten Vielfachen von 90 Grad gerundet, die Knicke werden auf 90 bzw. 270 Grad gesetzt. Hier muss ein Tradeoff zwischen Stabilität und Lesbarkeit gemacht werden.
3. Kompaktierung

Die Schwierigkeit liegt in der Bestimmung der Winkel und Knicke unter Berücksichtigung der Vorgaben der Gewichtung von Stabilität/Lesbarkeit. Der Schwerpunkt soll variabel entweder mehr auf die Lesbarkeit oder die Stabilität gelegt werden können. Das Finden einer quasi-orthogonalen Form Q ist also ein zweiseitiges Optimierungsproblem mit der Stabilität (Änderungen der Form) und der Lesbarkeit (Anzahl der Knicke) als Zielfunktion.

Definition 17 (Lesbarkeit): *Die Lesbarkeit einer Form Q ist definiert als die gesamte Anzahl der Knicke in Q, und ist gegeben durch*

$$B(Q) = \frac{1}{2} \sum_{f \in F} \sum_{(e,a,b) \in Q(f)} |b|$$

Die Lesbarkeit ist also die Summe der Länge der Bit-Strings, die die Knicke der Kanten repräsentieren.

Definition 18 (Stabilität): *Die Stabilität einer orthogonalen Form Q ist die Differenz zwischen den Winkeln in Q und den entsprechenden Winkeln in der Form S*

der Skizze S

$$\Delta_A(Q,S) = \sum_{f \in F} \sum_{1 \leq i \leq |f|} |a(S,f,i) - a(Q,f,i)|$$

und der Differenz in den Kantenknicken

$$\Delta_B(Q,S) = \sum_{f \in F} \sum_{1 \leq i \leq |f|} \Delta(b(S,f,i), b(Q,f,i))$$

wobei $\Delta(s_1, s_2)$ die bitweise Editier-Distanz zwischen zwei Bit-Strings ist. Beim Editieren der Bit-Strings sind nur Einfüge- und Lösch-Operationen erlaubt.

Nun können wir die Optimierungsfunktion des Sketch-Driven-Layouts aufstellen:

Definition 19 (Zielfunktion des Sketch-Driven-Layout): *Die Zielfunktion für das Optimierungsproblem im skizzenbasierten Layout (Sketch-Driven-Layout) ist gegeben durch die gewichtete Funktion*

$$D(Q|S) = \underbrace{\alpha \cdot \Delta_A(Q,S) + \beta \cdot \Delta_B(Q,S)}_{Stabilität} + \underbrace{\gamma \cdot (B(Q) - B(S))}_{Lesbarkeit}$$

Somit können wir nun das in [BKWE02] aufgestellte Problem formulieren:

Problem 1 (Sketch-Driven-Problem): *Sei S eine quasi-orthogonale Form eines planaren Graphen G. Das Sketch-Driven-Problem besteht in der Suche einer quasi-orthogonalen Form Q von G, die die Kandinsky-Eigenschaften erfüllt und in der $D(Q|S)$ minimal ist.*

Um das Problem als Netzwerkflussproblem zu modellieren, erweitern wir unser Knotenmodell von Abbildung 4.6 und ändern die Modellierung von Knicken, um das Sketch-Driven-Problem lösen zu können.

- Erweiterung des Knotenmodells:
  Für jede zwei aufeinanderfolgenden Kanten e und e' um einen Knoten v definieren wir einen Winkelknoten (angle-node). Der Winkelknoten $a^v_{e,e'}$ repräsentiert den Winkel um Knoten v zwischen den Kanten e und e'. Kanten, die bisher zum Knoten verliefen, werden nun an die entsprechenden Winkelknoten angehängt. Für jeden Winkelknoten wird jeweils eine Kante zum Knoten und eine in umgekehrter Richtung eingefügt. Diese haben unbegrenzte Kapazität und Kosten α.
  Für jeden Winkelknoten w gibt die Form S der Skizze einen Zielwinkel $a(w)$ vor. Ist $a(w) > 0$, so wird w mit der Netzwerkquelle mit einer neuen Kante mit Kapazität $a(w)$ und Kosten 0 verbunden. Ist $a(w) < 0$, so wird w mit der Netzwerksenke mit einer neuen Kante mit Kapazität $a(w)$ und Kosten 0 verbunden. Der Zufluss $a(w)$ zum Knoten wird entfernt. Das neue Knotenmodell ist in Abbildung 6.1 zu sehen.

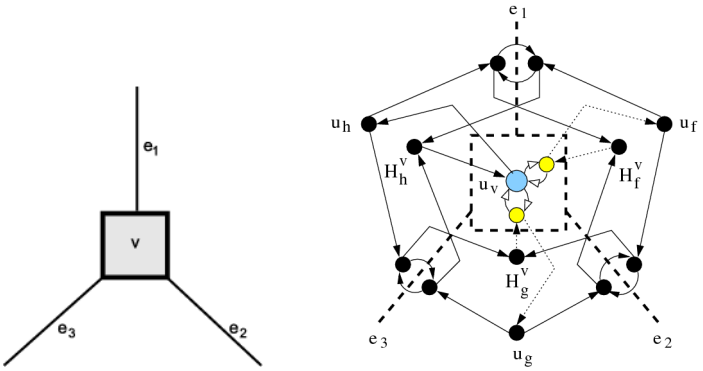


Abbildung 6.1: Ein Knoten u_v mit Grad 3 und sein modifizierter Kandinsky-Knoten mit den neu an u_v angehängten Winkelknoten (gelb).

- Knickmodellierung:
  Knicke werden wie in Abbildung 6.2 im Netzwerk modelliert. Für Knicke, die adjazent zu Knoten liegen, wird zusätzlich eine Kante, wie in Abbildung 6.2b gezeigt, eingefügt.

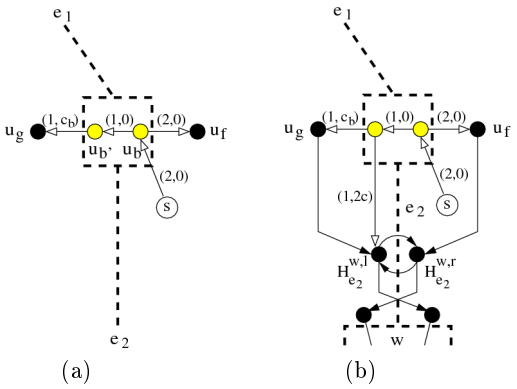


Abbildung 6.2: Erweiterung des Kandinsky-Netzwerkes um vorgegebene Knicke (a) und Knicke, die zu Knoten benachbart sind und Kandinsky-Knicke sein können (b).

In [BKWE02] wird eine Beweisskizze dafür gegeben, dass obige Änderungen im Kandinsky-Modell eine Lösung für Problem 1 ermöglichen.

6.3 Anwendung und Beispiel

In dieser Arbeit wird der Sketch-Driven-Layout-Ansatz in drei Anwendungsfällen verwendet. Diese sind:

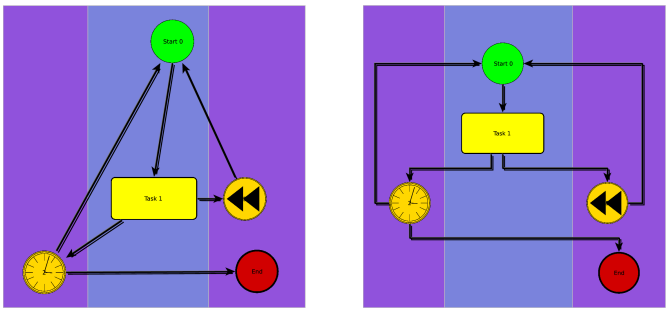
- Layout aus Skizzenentwurf

- Einfügen und Löschen von Graphelementen
- Automatisches Kantenrerouting

Im Folgenden werden die Anwendungen erläutert und mit Beispielen unterlegt.

6.3.1 Layout aus Skizzenentwurf

In diesem Anwendungsfall gibt der Benutzer eine grobe Skizze vor. Zusätzlich kann der Benutzer eine Gewichtung der Knicke und Winkel (α- und β-Gewichte und ein γ-Wert) für das Optimierungsproblem 1 angeben, die ansonsten gleichgewichtet vorbelegt werden. Daraus berechnet das automatische Layout dann einen Entwurf, der dem mentalen Bild des Benutzers entspricht.
In Abbildung 6.3 ist ein Beispiel mit einem kleineren Graphen zu sehen, in den Abbildungen 6.4 (Skizze) und 6.5 (Automatisches Layout) mit einen größeren Entwurf.



(a) (b)

Abbildung 6.3: Ein Entwurf (a) und das resultierende Layout (b) mit Hilfe des Sketch-Driven-Ansatzes.

6.3.2 Einfügen und Löschen von Graphelementen

Um Einfüge- oder Löschoperationen von Graphelementen zu visualisieren, nehmen wir an, dass bereits ein Graph mit einem orthogonalen Layout existiert, in dem die Operationen vorgenommen werden. Die Löschoperation ist nicht so komplex wie die Einfügeoperation, daher wird sie nur in zwei kleinen Beispielen visualisiert:

- Löschen eines Knotens: siehe Abbildungen 6.6a (Skizze) und 6.6b (Sketch-Driven-Layout)
- Löschen von Kanten: siehe Abbildungen 6.7a (Skizze) und 6.7b (Sketch-Driven-Layout)

Die Einfügeoperation wird durch den Benutzer vorgenommen, indem er das Element möglichst dort platziert, wo es nach seinem mentalen Bild und also auch nach dem skizzenbasierten Entwurf stehen soll. In den Abbildungen 6.8a und 6.8b wird

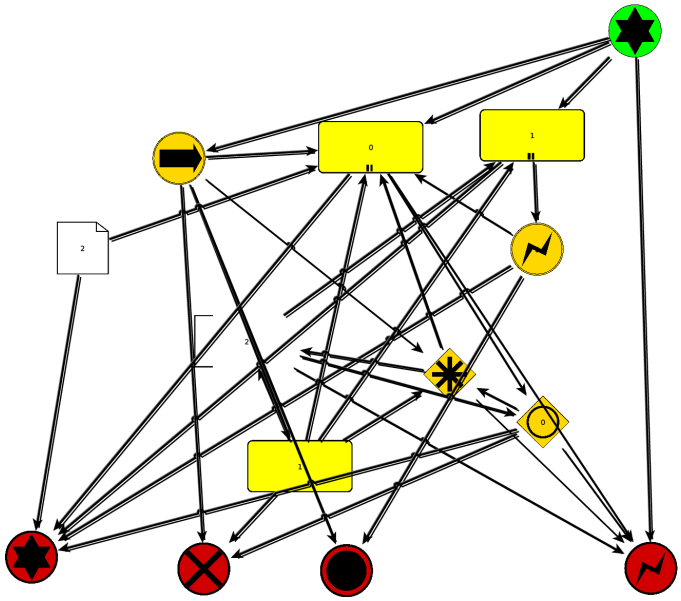


Abbildung 6.4: Ein umfangreicherer Skizzenentwurf mit hoher Dichte, der als Vorlage zum automatischen Layout dient (Resultat des autom. Layout, s. Abb. 6.5).

beispielhaft ein neuer Knoten in ein bereits fertiges Layout eingefügt und mit Sketch-Driven-Layout neu berechnet.

6.3.3 Automatisches Kantenrerouting

Das automatische Kantenrerouting ist ein Anwendungsfall, der auf ein fertiges Layout angewandt wird. In vielen Prozessmodellen ist der Grad der Knoten nicht gleichmäßig, d.h. es gibt einige „zentrale“ Knoten, z.B. Prozessschritt-Gruppen, die mehrmals durchlaufen werden und/oder viele Auswahlmöglichkeiten. Manchmal sollen auch viele mögliche Fälle zwischen zwei Prozesschritten aufgefangen werden. Dies erhöht die Dichte des Graphen und macht ein orthogonales Layout komplexer.
Falls das automatische Layout in manchen Fällen die Kanten eines Knoten „offensichtlich“ (mit dem Auge des Betrachters gesehen) nicht optimal geroutet hat, so können mit dem automatischen Kantenrerouting eine einzelne Kante oder auch alle Kanten eines Knotens neu geroutet werden.

Das Vorgehen in einem Graph $G = (V, E, F)$ für ein Kantenrerouting der Kanten $E_v \in E$ um einen Knoten v geschieht in folgenden Schritten:

1. Lösche alle Kanten E_v aus G und speichere sie in einer separaten Liste L_{E_v}.
2. Füge eine Hilfskante e_{aux} in den Graphen ein, die sicherstellt, dass der Graph zusammenhängend bleibt. Die Kante e_{aux} verbindet v mit dem nächsten Nachbarn in der gleichen Bahn. Selbst wenn ein Knoten einer anderen Bahn näher läge, so würde eine Kante über die Bahngrenze hinweg neue Kreuzungen in der

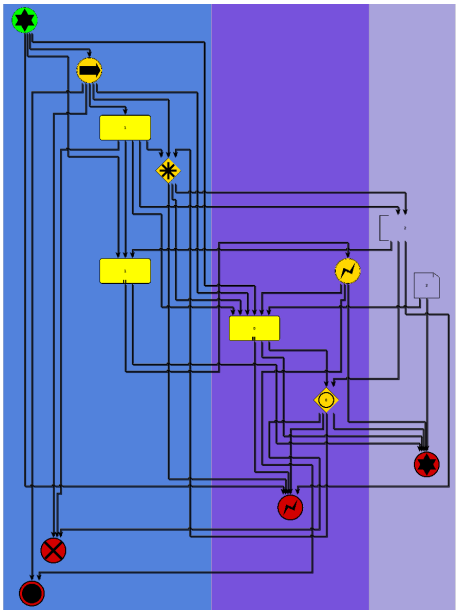
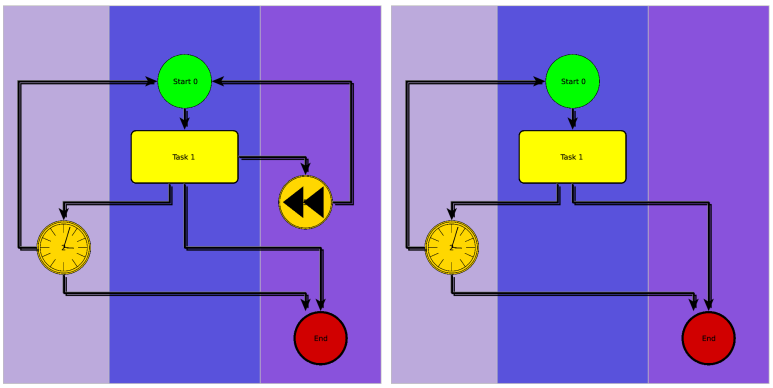


Abbildung 6.5: Das resultierende automatische Layout zum Entwurf aus Abb. 6.4.

orthogonalen Form verursachen. Nach Einfügen der Hilfskante erhält man den Zwischengraphen $G_{aux} = (V, E_{aux}, F)$ mit $E_{aux} = \{E \cup \{e_{aux}\}\} \setminus E_v$.

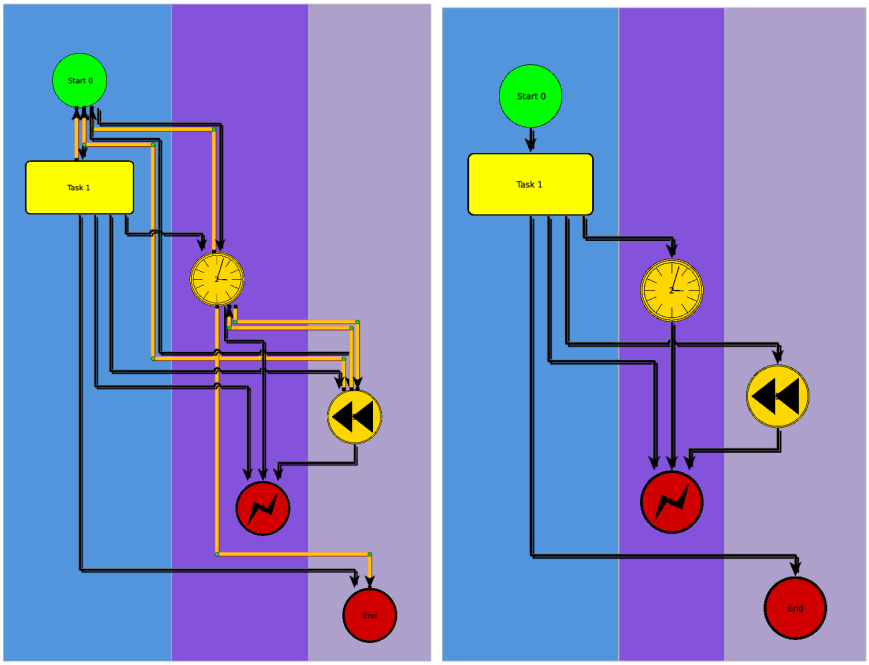
3. Führe den Algorithmus des skizzenbasierten Entwurfs aus, damit mögliche Flächenreduzierungen druchgeführt werden.
4. Lösche e_{aux} aus E_{aux} und füge alle $e \in L_{E_v}$ wieder ein (dabei wird in der Implementierung schon ein orthogonaler Kantenleger, s. [yG07] verwendet, der nur Winkel mit Vielfachen von 90 Grad zulässt)
5. Führe den Algorithmus des skizzenbasierten Entwurfs wiederholt aus, um das fertige Layout zu erhalten, damit mögliche Überlappungen entfernt und die Flächennutzung des Graphen durch die Kompaktierung verbessert werden.

Mit dem automatischen Kantenrerouting von allen Kanten eines Knoten oder dem Rerouting einzelner Kanten können „optische Fehler“, die dem Betrachter ins Auge fallen und vielleicht stören, leicht behoben werden, s. Abbildungen 6.9 (für eine einzelne Kante) und 6.10 (für inzidente Kanten eines Knotens).



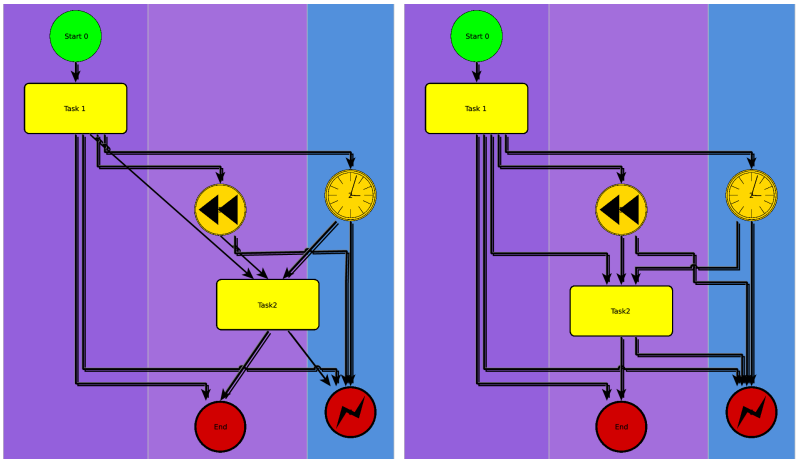
(a) (b)

Abbildung 6.6: Beispiel für die Löschung eines Knotens und das Layout des reduzierten Graphen mit Sketch-Driven-Layout. Es wurde der Knoten „Event Intermediate Forward“ gelöscht. Es ist erkennbar, dass die Skizze des Ausgangsgraphen (a) beibehalten wurde.



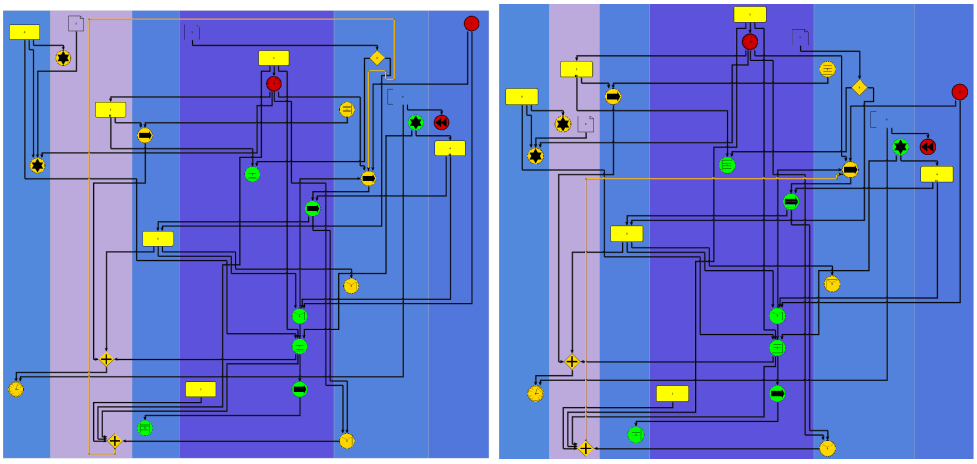
(a) (b)

Abbildung 6.7: Beispiel für die Löschung von Kanten und das erneute Erstellen eines Layout mittels Sketch-Driven-Layout. Die gelb markierten Kanten in Abb. 6.7a wurden gelöscht und dann ein Sketch-Driven-Layout auf dem Ausgangsgraphen ausgeführt. Der Layout-Ansatz verbessert die Kantenverläufe der übrigen Kante und reduziert die benötigte Fläche.



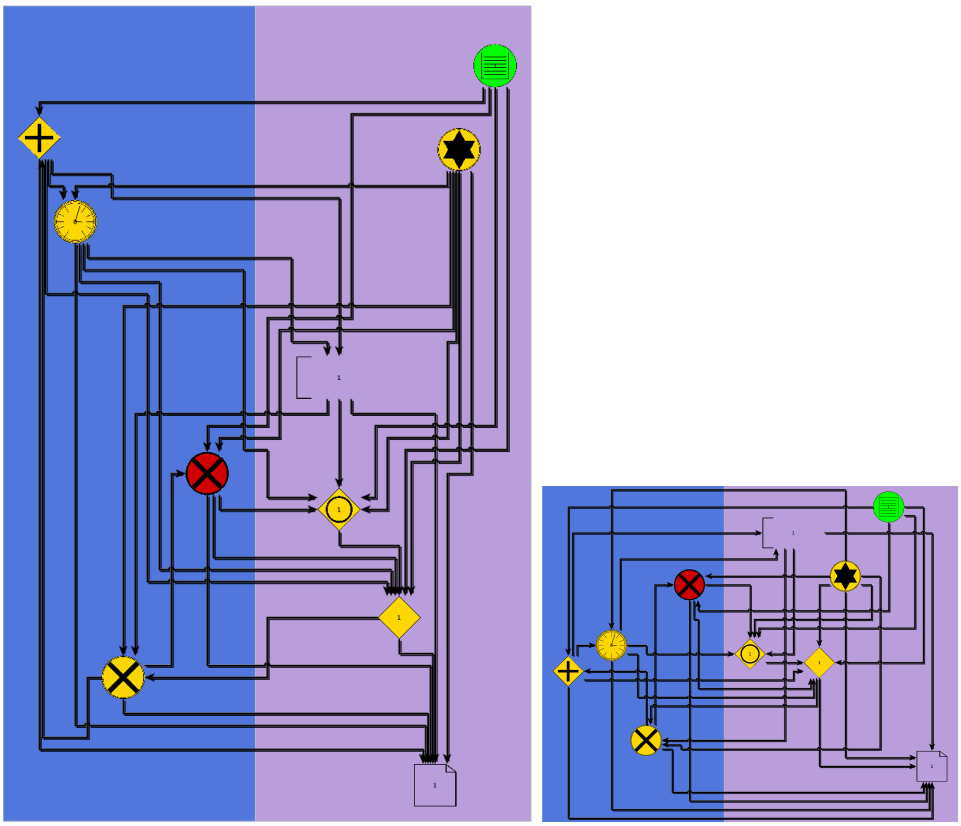
(a) (b)

Abbildung 6.8: Einfügen eines Knotens in eine bestehende Zeichnung. Der Knoten „Task 2„ wurde neu vom Benutzer in Abb. 6.8a skizziert und mit dem Sketch-Driven-Layout wurde dann ein orthogonales Layout berechnet, s. Abb. 6.8b.



(a) (b)

Abbildung 6.9: Rerouting einer einzelnen Kante: Die gelb markierte Kante wird im Graphen (b) neu geroutet. Sie besitzt nun weniger Knicke und wurde in der Länge reduziert.



(a) (b)

Abbildung 6.10: Rerouting von Kanten. Hier wurden an allen Knoten die Kanten neu geroutet. Der neue Graph benötigt deutlich weniger Fläche, beachtet aber die Aufwärtszeichnung der Kanten in diesem Fall nicht.

7 Schnitte von BPMN-Graphen

Die Zeichnungen von BPMN-Graphen können je nach Komplexität des abzubildenden Prozesses auch bei Benutzung einer optimalen Form unübersichtlich wirken, wenn man den kompletten Prozess in einer Zeichnung abbildet. Auch viele Swimlanes können eine Prozesszeichnung sehr großflächig werden lassen. In diesem Kapitel werden Methoden vorgestellt, um die Zeichnungen aufzuschneiden bzw. zu teilen und diese Teile separat zeichnen zu können. Die Schnitte dazu können durch verschiedene Methoden ausgeführt werden, welche unten vorgestellt werden. Dabei ist anzumerken, dass das automatische Finden eines optimalen Subgraphen unter gegebenen Bedingungen ein Optimierungsproblem darstellt.
Daher bieten wir in Kap. 7.1 eine semi-automatisierte Methode zur Schnittbildung an, bei der der Benutzer Teile des Prozesses, die er gezeichnet haben möchte, durch Auswahl vorgibt. Im Kapitel 7.2 stellen wir dann eine Heuristik vor, mit der das Optimierungsproblem des automatischen Schnitts gelöst werden kann.

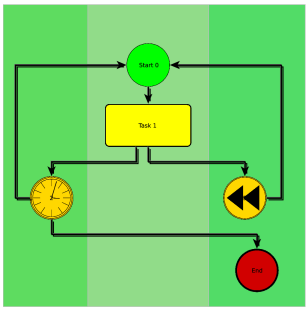
7.1 Schnitt durch Auswahl

Die semi-automatische Schnittbildung durch eine Zeichnung kann auf mehrere Arten erfolgen. Diese sind:

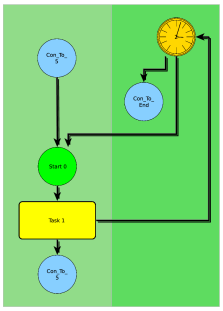
1. Auswahl der zusammengehörigen Teile des Prozesses, die ausgeschnitten werden sollen. Die Auswahl umfasst Knoten, Kanten und Elemente verschiedener Art und unterschiedlich zugeordneter Swimlanes. Der gewählte Graph ist ein Teilgraph des ursprünglichen BPMN-Graphen, siehe Abb. 7.1b.

2. Angabe von Bahnen (swimlanes), die neu zu zeichnen sind. Swimlanes entsprechen im BPMN-Verständnis oft Abteilungen von Unternehmen bzw. verschiedenen Prozessteilnehmern. Eine Teilmenge der Swimlanes kann somit für eine Untermenge der Teilnehmer ihren Part im Prozess veranschaulichen, siehe Abb. 7.1c. Die gewünschten Bahnen werden durch einen regulären Ausdruck ausgewählt, der dem des gewöhnlichen Angebens der Seitenzahlen an den Drucker gleicht, so z.B. „$3 + 4, 5, 7 - 8$“.

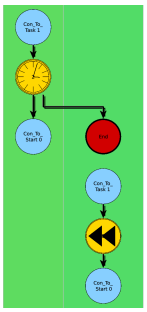
3. Auswahl von Knoten im BPMN-Graphen. Eine Darstellung nur bestimmter Knoten eines BPMN-Graphen ist sinnvoll, wenn z.B. alle Tasks in einem Prozess visualisiert werden sollen. Die Auswahl einzelner Knoten ist dann sinnvoll, wenn spezielle Teile des Prozesses und ihre Verbindungen innerhalb oder nach außen hin untersucht werden sollen. So kann z.B. ein Task mit all seinen



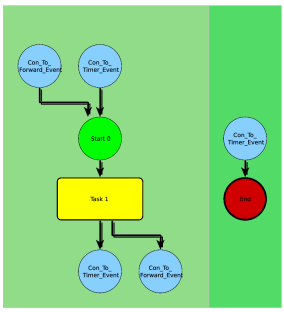
(a) Ausgangsgraph



(b) Teilgraph durch Elementauswahl



(c) Schnitt durch Swimlane-Auswahl



(d) Schnitt durch Knotenauswahl

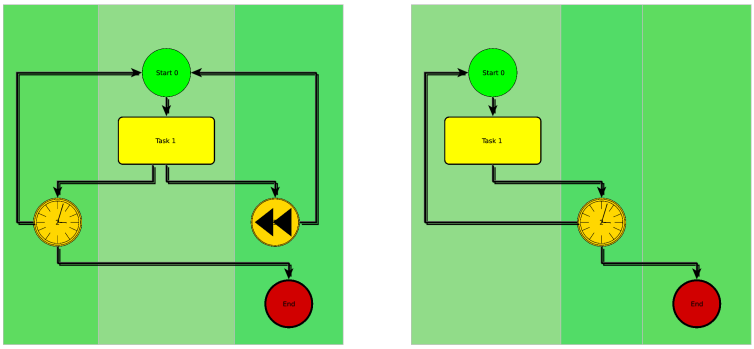
Abbildung 7.1: Beispiel für die verschiedenen semi-automatischen Schnitte.

vorgehenden Aufgaben und sein Grad als Maß für seine Bedeutung und/oder Häufigkeit im Prozess visualisiert werden, siehe Abb. 7.1d.

Bei allen drei Vorgehensweisen werden Teilgraphen mit den manuell ausgewählten Elementen gebildet. Diese Teilgraphen werden dann als eigenständige Graphen behandelt. Im TSM-Framework wird ein automatisches Layout dafür berechnet.
Eine Besonderheit bei den entstehenden Teilgraphen sind die Kanten, die durchschnitten werden, weil der jeweilige Ziel- oder Quellknoten nicht Bestandteil des Graphen wird. Diese Kanten können entfernt werden oder es kann an ihr offenes Ende ein Konnektor angehängt werden, der auf eine Verbindung nach „außen“ hinweist, d.h. zu einem Knoten des ursprünglichen Graphen. Beide Möglichkeiten werden in dieser Arbeit angewandt und hier anhand von Beispielen in den Abbildungen 7.1b – 7.2 dargestellt.

7.2 Automatischer Schnitt

Ein zweites Hilfsmittel für bessere Übersicht in großen Diagrammen ist die Aufteilung in mehrere kleine Diagramme. Die kleineren Diagramme können durch Konnektoren anstelle der geschnittenen Verbindungskanten mit den anderen neu entstandenen Diagrammen verknüpft werden.



(a) Ausgangsgraph (b) Schnitt ohne Konnektoren

Abbildung 7.2: Knotenschnitt des Ausgangsgraphen (s. Abb. 7.1a) ohne das Einfügen von Konnektoren für durchschnittene Kanten.

Die Schnitte (cuts) von Graphen in Subgraphen sind bereits bekannter Bestandteil weiterer Forschung (so z.B. das *Ratio-Cut*-Problem in [Len90], und einer Anwendung in Bilddateien, siehe [WS03]).

Im Fall der Geschäftsprozesse ist ein günstiger Schnitt auch gewünscht, um den Prozess zum Ausdruck auf mehrere Blätter vorzubereiten. Dazu wird der Graph in Subgraphen geschnitten. Die automatische Suche nach geeigneten Subgraphen kann nach mehreren Kriterien erfolgen, so sollte zum Beispiel die Anzahl der Knoten in Subgraphen ausgewogen sein, die notwendigen Kantenschnitte des Schnittes minimiert werden oder die Swimlanes möglichst unangetastet bleiben und nie leer sein, d.h. mindestens ein Knoten pro Swimlane verbleiben.

Für die Berechnung des Schnittes führen wir die Mittelkante (center edge) ein, die eine Schnittkante durch den Graphen darstellt und im Laufe des Algorithmus zu einem Mittelstreifen (center band) verbreitert wird. Sie stellt am Anfang des Algorithmus den initialen Schnittkantenkandidaten dar und bei einer Verbreiterung zu einem Streifen ist dieser Mittelstreifen der Korridor, in dem die ideale Schnittkante gesucht wird.

Der Kernpunkt der Schnittbildung ist das Routing im dualen Graphen. Der duale Graph G_D, siehe Def. 4.3, wird über dem Graphen G gebildet, der durch das Rechteck des Mittelstreifen und den (Teil-) Kanten von G erzeugt wird. Sei G' der durch das Rechteck und Teilkanten induzierte Graph. Gesucht wird ein Pfad durch diesen Graphen G'. Die Kanten werden unterschiedlich gewichtet, damit der Routingalgorithmus wirklich durch den Graphen hindurch routet. Ansonsten erhält man schlimmstenfalls keinen Schnitt, sondern, wenn das Routing im dualen Graphen das Rechteck zu früh verlässt, nur einen Teilschnitt, da der Routingalgorithmus mit einem vermeintlichen minimalen Schnitt abbricht, der Graph aber nicht komplett durchschnitten wurde.

Durch sehr hohe Kantengewichte auf den Segmenten der Rechteckkanten E', die den Mittelstreifen von G trennen, verhindert man ein ungewolltes Routen in $G \backslash G'$ hinein.

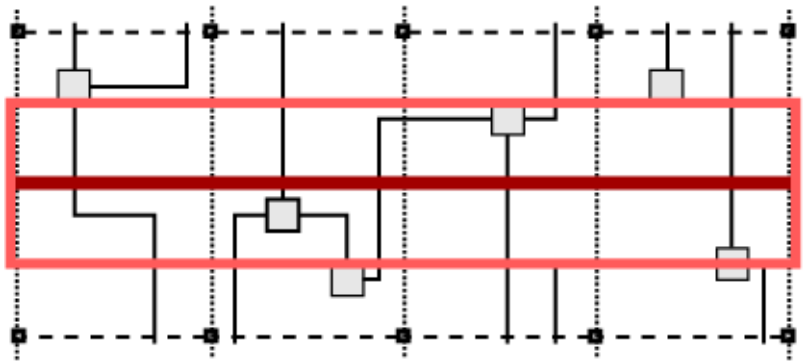


Abbildung 7.3: Ausschnitt aus einem Graphen. Die vertikalen Swimlanes werden durch die gepunkteten Linien gekennzeichnet. Die Mittelkante y_m ist hier rot markiert, der sie umgebende Mittelstreifen ist hellrot umrahmt. Der Mittelstreifen wird geometrisch durch y_o und y_u vorgegeben. Im vom Mittelstreifen vorgegebenen Bereich soll ein Schnitt mittels Dualgraph-Routing gefunden werden.

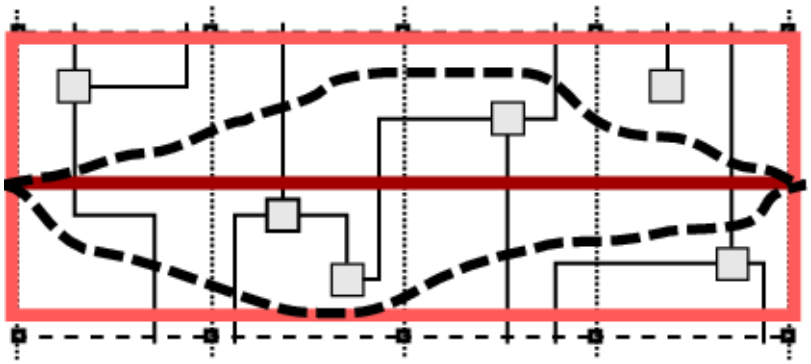


Abbildung 7.4: Zwei Routen (dick gestrichelt) für Schnitte mit jeweils 4 Kanten.

Eine Alternative ist es, die Kanten von G_D, die durch diese Rechteckkanten gehen, zu löschen. Ein ungewolltes Routing aus G' wird dann nicht möglich. Die Kantengewichte der anderen beiden Rechteckkanten sind sehr gering. Die Kantengewichte für die (Teil-) Kanten der Originalgraphen werden auf einen Einheitswert gesetzt.
Der Schnitt ist nicht nur horizontal möglich, wie in Abbildung 7.4. Ein vertikaler Schnitt ist ebenso möglich, indem man den Mittelstreifen vertikal konstruiert. Vertikal verlaufende Swimlanes schneidet der Mittelstreifen teilweise, wobei die Anzahl der geschnittenen Swimlanes geringer ist als im horizontalen Schnitt.
Um den Graph in eine vorgegebene Anzahl von Teilen zu zerschneiden, werden mehrere Mittelkanten in gleichmäßigen geometrischen Abständen eingefügt. Die Mittelstreifen werden dann um jede Mittelkante gebildet und die oben beschriebene Methode wird für jeden Mittelstreifen angewandt, solange sie nicht überlappen. Die Laufzeit wird dominiert durch die Berechnung des Dual-Routing, das auf dem Finden eines Shortest-Path basiert. Wenn man für Shortest-Path den Dijkstra-Algorithmus verwendet, so hat das Dual-Routing eine Laufzeit von $O(|V| \cdot \log |V| + |E|)$.
Wenn man den Graph zum Beispiel auf eine gegebene Anzahl von Blättern zerschnei-

den möchte, wird der Algorithmus rekursiv auf die durch einen Schnitt entstandenen Teilgraphen angewandt, solange sie noch zu groß für ein Blatt sind. Einen Pseudo-Code-Algorithmus der den Schnitt auf Blätter mittels Dual-Routing berechnet, stellen wir in Algorithmus *constrained-bpmn-cut* vor. Für die Gesamtlaufzeit muss nun noch die Rekursion berücksichtigt werden. So kommt man auf eine Laufzeit von $O((|V| \cdot \log(|V|) + |E|) \cdot |V|)$.

Algorithm:constrained-bpmn-cut

Data: Graph G, int sheets, constraints
Output: subgraphs-array $[G_{sub}]$

Step 1:
// check if number of subgraphs match with wanted number of sheets
// to be printed
if *(* $[G_{sub}].length < sheets$*)* **then**
  GOTO Step 2;
  else **GOTO** Step 7;

Step 2:
// initialization of min-cut run on subgraph
initialize center edge by calculating barycentric x- and y-position,
set center band to center edge with thickness of few grid points;

Step 3:
// dual routing is performed on center band
build subgraphs induced by center band and call dual routing in center band;

Step 4:
// check constraints otherwise repeat dual-routing with larger center band
if *(constraints fulfilled)* **then**
  GOTO Step 6;
  else enlarge center band towards subgraph with higher area size;

Step 5:
GOTO Step 2;

Step 6:
// recursive call of algorithm on subgraph to be cut
$[G_{sub}]$.add{$subgraphs$};
GOTO Step 1 with subgraphs and sheets/2 as input;

Step 7: return $[G_{sub}]$;

---

8 Implementierung der BPMN-Layoutanwendung

Im praktischen Teil dieser Arbeit wurde eine Anwendung zur Arbeit mit BPMN-Prozessen entworfen und entwickelt. Die Anwendung ist ein GUI-Programm, das technisch auf Java 1.5, den Java Swing-Bibliotheken, und yFiles (Version 2.5, Dokumentation s. [yG07]) basiert. Sie ist voll funktionsfähig und stellt ein wesentliches Ergebnis dieser Arbeit dar.
Zum Funktionsumfang der Anwendung gehören folgende implementierte Anforderungen:

- Die Erstellung von BPMN-Prozessen mit Hilfe der GUI-Werkzeuge. Es stehen alle BPMN-Elemente aus Kap. 3 zur Verfügung. Dazu kann jedem Element ein Kommentar hinzugefügt werden, der dem Benutzer zusätzliche Informationen, z.B. ausführlichere Beschreibungen, bereitstellt.
- Die Verwaltung bestehender BPMN-Prozesse. Fertig entworfene Prozesse können verändert werden, d.h. neue Elemente eingefügt, existierende gelöscht oder ihre Eigenschaften verändert werden. Auch eine Import-/Export-Funktion von sowohl graphischen Bild-Formaten (*.jpg, *.svg, *.bmp) als auch gängigen Graph-Formaten, die von yFiles unterstützt werden, steht zur Auswahl.
  Fertige Zeichnungen (Layouts) von BPMN-Prozessen können mittels kleinerer Werkzeuge noch individuell angepasst und verbessert werden, siehe Kap. 8.3.
- Das automatische Konstruieren von automatischen Zeichnungen bzw. Layouts von BPMN-Prozessen. Die Hauptaufgabe der Anwendung ist das Berechnen der Layouts. Hier stehen mehrere Layoutalgorithmen mit individuellen Optionen zur Verfügung, die in Kap. 8.2 vorgestellt werden. Die Anwendung bietet zwei Arbeitsmodi: einen zum Entwurf der BPMN-Prozesse und einen zum Berechnen der Layouts, in dem dann der Graph nicht veränderbar ist. Jedoch kann der Benutzer frei zwischen den Modi wechseln.

8.1 Der BPMN-Modellierer und -Layouter

Das Hauptfenster der Anwendung ist in Abb. 8.1 abgebildet. Die große Zeichenfläche in der Mitte ist die Arbeitsfläche für den BPMN-Builder. Der BPMN-Builder dient der Erstellung von Modellen der BPMN-Prozessen. Befindet sich der Benutzer im *Edit-Mode*, so kann er durch einfaches Klicken auf die freie Zeichnungsfläche den Dialog zur Erstellung eines BPMN-Elements aufrufen. Der Dialog ist in Abbildung 8.2 abgebildet. Er kann auch nach der Erstellung des Knotens durch Auswahl in

einem Popup-Menü aufgerufen werden, das sich nach Rechtsklick auf das jeweilige BPMN-Element öffnet. Der Dialog dient auch der späteren Veränderung von Eigenschaften des Elements. Auch Verbindungsobjekte werden durch einen Dialog auf ihre Verwendungsart, siehe Kap. 3.2.2 angepasst. Die Konstruktion erfolgt im *Edit-Mode* einfach durch Drag-and-Drop zweier BPMN-Knoten.
Das Verschieben von Knoten ist nach Markieren frei möglich, wobei sich das Verhalten stark an den Möglichkeiten der yFiles-Bibliothek orientiert. Per Rechtsklick über der Zeichenfläche oder über einem BPMN-Element öffnet sich ein Popup-Menü, über das auch die Ansicht verändert werden kann. Es lässt sich frei zoomen oder auf ein markiertes Element bzw. eine markierte Elementengruppe zentrieren und zoomen. Befinden sich Konnektoren in der Zeichnung, so ist ihr Popup-Menü um spezielle Aktionen erweitert: Die Ansicht lässt sich zwischen den Konnektoren eines Konnektorpaares hin und her wechseln und der Fokus kann auf das Rechteck, das beide Konnektoren enthält, zentriert werden. Darüber hinaus blendet sich bei Mausfokus auf einem Konnektor eine dynamische Kante, siehe Abb. 8.3, die zum jeweiligen Konnektorpaar verläuft und nach Verlust des Fokus auf diesem Knoten wieder ausgeblendet wird. Dies dient der Virtualisierung der Konnektorpaare und dem Verlauf ihrer Originalkanten, die sie ersetzt haben.

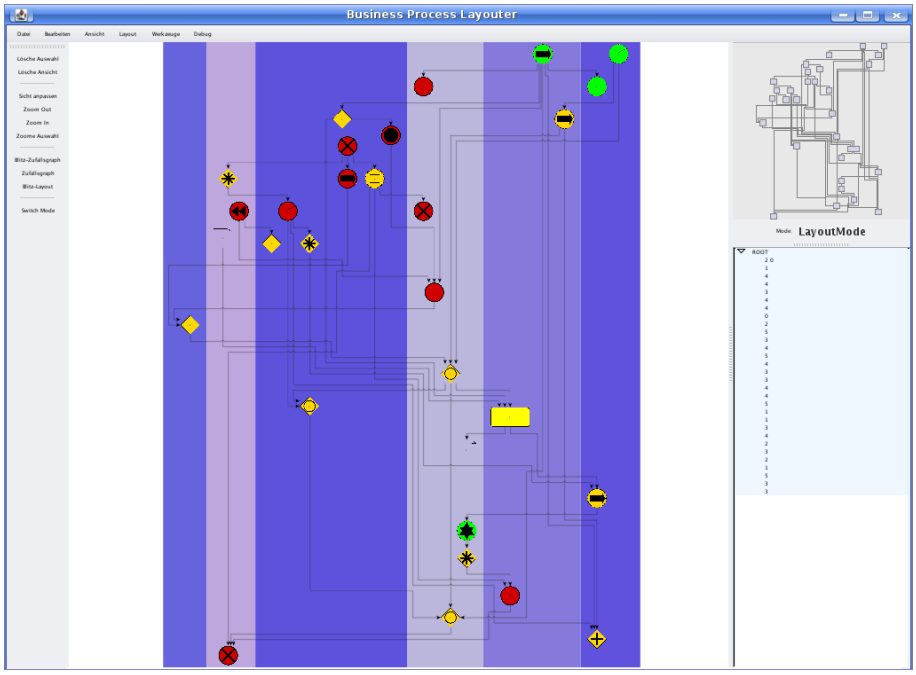


Abbildung 8.1: Hauptfenster der Applikation.

8.2 Layout-Algorithmen

Um für einen Prozess ein Layout zu berechnen, stellt die Anwendung mehrere Layoutalgorithmen zur Verfügung, wobei der *Automatic BPMN-Layout*-Algorithmus der

Abbildung 8.2: Eigenschaften-Dialog für die Knoten.

für die BPMN-Modelle speziellste und vollständigste ist. Als Beispiel für die Abbildungen Prozesse der verschiedenen Algorithmen liegt der Beispielprozess aus Abb. 8.4 zugrunde. Die Algorithmen sind im einzelnen:

- Circular Layout:
  Dieser Algorithmus berechnet eine Zeichnung, wobei eine Hauptachse in Form eines Kreises konstruiert wird und die übrigen Knoten orthogonal zur Kreistangente angeordnet werden. Ein Beispiel ist in Abb. 8.5 dargestellt. Eine Zeichnung im Circular Layout ist für BPMN-Prozesse nur begrenzt sinnvoll, jedoch lassen sich in diesem Layout leicht die zentralen Knoten eines Prozesses, d.h. Knoten mit hohem Grad, erkennen oder entdecken, da sie sich in der Kreisachse des Circular Layout befinden.

- Organic Layout:
  Der Layout-Algorithmus des Organic Layout stammt aus dem physikalischen Umfeld und wurde mit dem Paradigma des kraftgesteuerten Layout (force-directed layout) entwickelt. Ein Organic Layout eignet sich z.B. um Clusterkomponenten in einem Prozess festzustellen. Das sind diejenigen Prozessteile, die enger miteinander durch Verbindungsobjekte verbundenten sind als mit dem restlichen Prozess. In Abb. 8.6 ist das Beispiel in einer Zeichnung mit Organic Layout dargestellt.

- Orthogonal Layout:
  Dieser in den Bibliotheken der yFiles enthaltene Layoutalgorithmus berechnet für einen Graphen eine orthogonale Zeichnung. Dabei beachtet er jedoch noch die Eigenheiten eines BPMN-Modells. So werden z.B. nicht die Zuordnungen der Knoten zu den Bahnen berücksichtigt. Daher ist dieses Layout geeignet, um ein einfaches Layout in orthogonaler Form zu erhalten, ohne jedoch auf die Vollständigkeit eines BPMN-Modells Wert zu legen. Das Beispiel in orthogo-

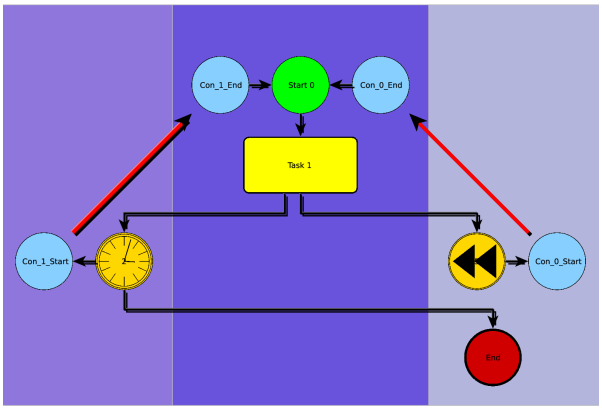


Abbildung 8.3: Visualisierung von zugehörigen Konnektorpaaren. Die roten Visualisierungskanten werden beim mouse-over über einem Konnektor ausgelöst. Sie können auch wie hier im Beispiel alle gleichzeitig aktiviert werden.

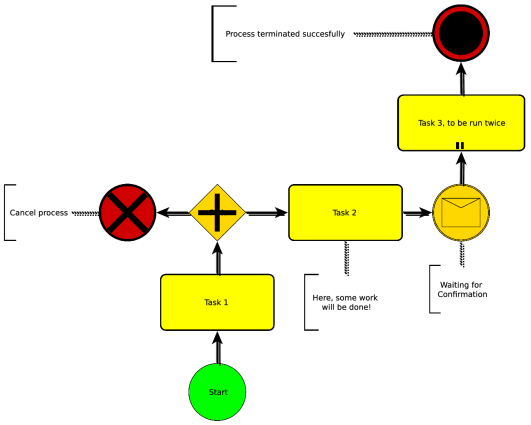


Abbildung 8.4: Beispiel für ein Prozessmodell in BPMN, das für die Darstellung der Layout-Algorithmen benutzt wird.

naler Form ist in Abb. 8.7 dargestellt.

- Sketch-Driven Layout:
  Der Algorithmus des skizzenbasierten Layouts (sketch-driven layouts) wurde ausführlich in Kap. 6 vorgestellt. Er eignet sich, um für eine vorgegebene Skizze ein Layout zu berechnen oder für eine Zeichnung, die verändert wurde, (erneut) eine orthogonale BPMN-Zeichnung zu konstruieren.
- Automatic BPMN-Layout:
  Das Konstruieren einer Zeichnung mittels des Algorithmus des automatischen BPMN-Layouts (Automatic BPMN-Layout) basiert auf den Ansätzen des Kapitel 4 und stellt den wesentlichen Layout-Algorithmus für die BPMN-Prozesse dar. Er berücksichtigt die Zuordnung der BPMN-Elemente zu den unterschiedlichen Bahnen, auf Wunsch des Benutzers auch den Fluss des Prozesses und erstellt eine aufwärtsgerichtete Zeichnung (upward drawing). Des Weiteren kann

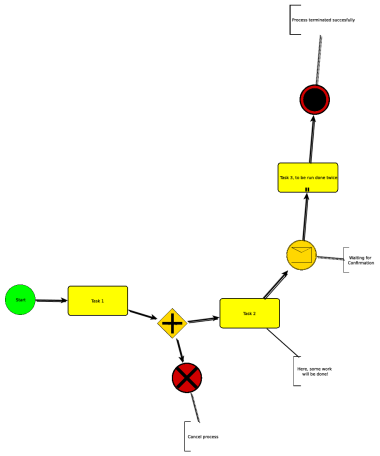


Abbildung 8.5: Circle-Layout des Prozessmodells aus Abb. 8.4.

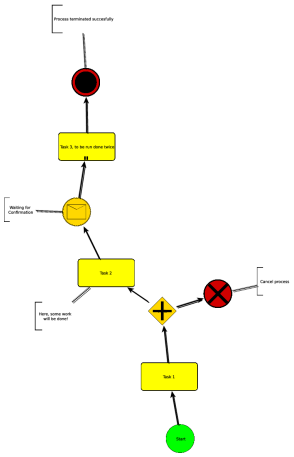


Abbildung 8.6: Organic-Layout des Prozessmodells aus Abb. 8.4.

er Konnektoren einfügen, wie sie in Kap. 5 vorgestellt wurden. Auch die teilautomatische Zeichnung von Schnitten, siehe Kap. 7, ist per Dialog auswählbar. Der Beispielprozess ist in Abb. 8.8 im Automatic BPMN-Layout dargestellt.

8.3 Verfeinerungen

Nach der Erstellung einer Zeichnung mit dem automatischen BPMN-Layout erhält man ein Layout, das den theoretischen Anforderungen genügt. Jedoch kann es in Details dem subjektivem Empfinden des Benutzers widersprechen. Daher gibt es die Möglichkeit, auch nachträglich, also nachdem die Zeichnung bereits berechnet wurde, Kantenverläufe zu verändern – das sog. Rerouting von Kanten. Die zwei verwendeten Varianten sind folgende:

- Rerouting einzelner Kanten (single-edge-rerouting):
  Um eine Kante neu zu routen, wird sie per Rechtsklick ausgewählt und das

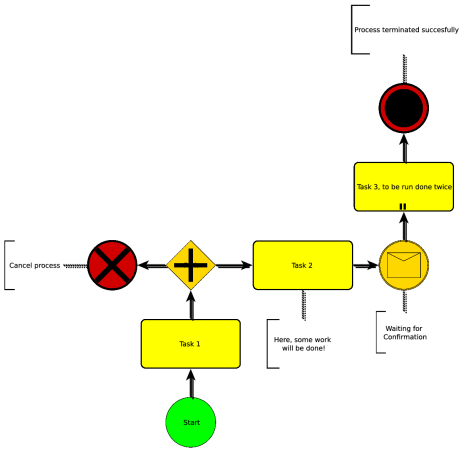


Abbildung 8.7: Orthogonal-Layout des Prozessmodells aus Abb. 8.4.

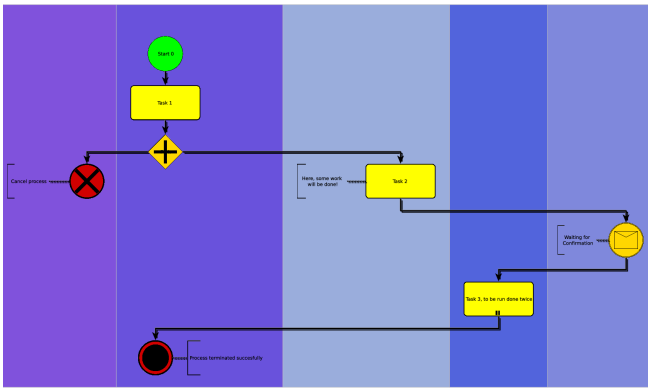


Abbildung 8.8: BPMN-Layout des Prozessmodells aus Abb. 8.4. Um die Semantik eines Prozesses vollständig abzubilden, bietet dieses orthogonale Layout u.A. die Berücksichtigung von Swimlanes und Knotenanordnungen (mit Sketch-Driven-Layout). Die o.g. Layouts haben nicht die Möglichkeit, solcherlei Zusatzinformationen in das Layout einfließen zu lassen.

Rerouting aktiviert. Die Kante wird dann automatisch temporär gelöscht, es wird ein Sketch-Driven Layout auf dem Graphen ausgeführt, um die Fläche, die durch die gelöschte Kante frei wurde, wieder zu nutzen. Die Kante wird dann mit dem orthogonalen Kantenleger aus der yFiles-Bibliothek ([yG07]) neu eingefügt und zuletzt wird noch einmal ein Sketch-Driven Layout vorgenommen, um Überlappungen mit Knoten und anderen Kanten zu vermeiden und die Abstände korrekt festzusetzen.

- Rerouting aller Kanten um einen Knoten (node edges rerouting):
  Die ein- und ausgehenden Kanten eines Knotens können ebenfalls auf Wunsch des Benutzers neu gezeichnet werden. Dazu wird das Popup-Menü des Knotens per Rechtsklick aufgerufen und das Rerouting der Kanten ausgewählt. Das Vorgehen ist nun im Vergleich zum single edge rerouting leicht abgewandelt, da bei einem Knoten, dessen Kanten alle gelöscht werden, der Fall eintreten kann, dass

der Gesamtgraph nicht mehr zusammenhängend ist. Daher wird nun nach dem temporären Löschen der Kanten des Knotens eine Hilfskante eingefügt, die den Knoten mit dem Restgraphen verbindet. Die Hilfskante wird so gewählt, dass sie möglichst kurz ist, wenig Knicke besitzt und keine Kreuzungen verursacht. Ein BPMN-Element aus der gleichen Bahn, das nah am Knoten liegt eignet sich sehr gut als Zielknoten der Hilfskante. Ist die Hilfskante eingefügt, wird ein Sketch-Driven-Layout durchgeführt, um wieder Fläche der gelöschten Kanten freizugeben. Nachdem der Kantenleger die Kanten erneut eingefügt hat, wird abschließend analog zum single-edge-rerouting ein Sketch-Driven Layout durchgeführt, denn auch hier müssen Überlappungen verhindert werden und die Abstände der neu eingefügten Kanten zu den restlichen BPMN-Elementen müssen korrekt gesetzt werden, da der Kantenleger die Abstände nicht einhält.

8.4 Features

Wichtig war für die Anwendung die Interaktivität. Bei der Arbeit mit mehreren Layouts kann durch eine Redo-/Undo-Funktion auch ein unerwünschtes Layout rückgängig gemacht und ein subjektiv wünschenswerteres aufgerufen werden. Zudem kann die Ansicht auf die Bedürfnisse des Benutzers angepasst werden. Die Zoomfunktion erlaubt auch speziell den Umgang mit Konnektoren, so dass z.B. genau auf ein Konnektorpaar im Fokus gezoomt werden kann. Ein Konnektorpaar, das dem Benutzer missfällt, kann mittels dem Popup-Menü wieder zu einer regulären Kante zurückgebildet werden. Diese neue Kante kann dann einfach mit dem Rerouting, siehe 8.3, wieder in den orthogonalen Graphen eingepasst werden.
Auch die Visualisierungskanten, die bei Mausbewegungen auf einem Konnektor temporär eine farbige Kante, die zum Konnektorpartner verläuft, einblenden, erhöhen die Übersichtleichkeit bei der Arbeit mit Konnektoren. Auch lassen sich auf Knopfdruck alle Visualisierungskanten gleichzeitig einblenden, um zu sehen, welche Kanten durch Konnektoren ersetzt wurden.
Die Farbwahl der Swimlanes ist ebenfalls individuell einstellbar. Es kann aber auch eines der sechs zur Verfügung stehenden Farbschemata gewählt werden, die die Grundfarbe der Swimlanes vorgeben, wobei keine zwei Swimlanes dieselbe Farbe zugewiesen bekommen, um Verwechslungen auszuschließen.

Ein bestehendes Layout kann durch Knopfdruck und Wechseln in den „*edit-mode*“ verändert werden und mittels Sketch-Driven aktualisiert werden, ohne dass ein komplett neues Layout berechnet werden muss. Der Benutzer kann zwischen dem Modellier- und dem Layoutwerkzeug einfach durch Knopfdruck wechseln. Dadurch wird die Arbeit mit der Anwendung deutlich beschleunigt und der Benutzer wird nicht nach jeder Änderung mit einem völlig neuen, ungewohnten Layout behelligt. Die Auswahl eines BPMN-Elements, das im „*edit-mode*“ neu hinzugeüfgt werden soll, erfolgt mittels Auswahl aus einer Liste im Eigenschaften-Dialog der Knoten, siehe 8.2. So wird auch die Modelliertätigkeit auf die nötigsten Schritte reduziert.

8.5 Liste offener Features

Bei der Implementierung der BPMN-Anwendung ist eine Liste mit Features entstanden, die im Rahmen dieser Arbeit nicht mehr implementiert werden konnten. Zwei dieser Features sollen hier noch kurz erwähnt werden:

- XML-Import/Export:
  Ein Datenaustausch auf XML-Basis ist gerade in Hinsicht auf die Verwendung verschiedenster Prozesse wünschenswert. Dieses Feature ist rein technischer Natur, aber XML macht es möglich, dass die BPMN-Graphen auch auf andere Anwendungen übertragen werden können, ohne dass dabei Informationen verloren gehen. Für die Realisierung müsste in einem ersten Schritt ein XML-Schema entworfen werden, das alle Informationen eines BPMN-Graphen beinhaltet, damit die XML-Dateien mit den Prozessen mittels des Schemas beim Parsen auf Gültigkeit geprüft werden können. Ein XML-Schema sprengt auch die Grenzen der Formate, die yFiles unterstützt, da für jedes BPMN-Element Zusatzinformationen gespeichert werden können.

- Semantische Checks:
  Beim Entwurf von BPMN-Modellen im BPMN-Builder könnten semantische Checks auf Inkonsistenzen innerhalb des Modells hinweisen. Dies kann spätere Fehlerquellen bei der Implementierung des Prozesses in einem frühen Stadium verhindern. Semantische Checks benötigen Wissen der Anwendung über die Bedeutung der BPMN-Elemente. So ist z.B. notwendig, dass jedes Gateway eine gewisse Anzahl von ein- und ausgehenden Kanten besitzt, jedes Join-Gateway muss mindestens zwei eingehende und genau eine ausgehende Kante haben. Auch Ereignis-Elemente erlauben semantische Checks: Ein Terminierungsknoten hat keine ausgehende Kante, ein Startknoten keine eingehende, ein Zwischenknoten (intermediate) jeweils eine ausgehende und eine eingehende Kante. Letztlich kann dann der Fluss in einem Prozess schon im Voraus, also vor der Implementierung, simuliert werden. Sind diese Simulationen möglich, z.B. mittels einer Flussanalyse, kann man Prozesse simulieren, die bestimmte Eigenschaften der Elemente, so z.B. Kosten und Dauer berücksichtigen. Dies reicht dann schon in die Domäne der Prozesssimulation hinein, die einen großen Bereich im Rahmen des unternehmerischen Prozessmanagement füllt.

- Erhöhung der Interaktivität:
  Weiterhin kann die Benutzerfreundlichkeit (Usability) durch Erhöhen der Interaktivität gesteigert werden. So kann eine Schnellauswahlleiste an der Fensterseite die Auswahl neuer BPMN-Elemente beim Modellieren beschleunigen. Die Implementierung der Schnitte in BPMN-Graphen, siehe 7, kann auch durch eine Vorschau der Teilgraphen bereichert werden. Die Erscheinung der Anwendung insgesamt kann durch Verwendung von leicht verständlichen Symbolen aufgebessert werden. Es bieten sich auch die Einbeziehung von umfangreicheren GUI-Tools als Swing für JAVA an, z.B. JGoodies (s. *http://www.jgoodies.com*).

- Positionen der Konnektoren verbessern:
  Die Implementierung der Konnektoren kann dahingehend verbessert werden,

dass die Lage der Position, an der sie eingefügt werden, exakter ausgerichtet wird. So kann sie zum Beispiel in Richtung des Konnektorpartners ausgerichtet werden. Möglich ist bei Konnektoren, die durch einen Schnitt entstanden, sie z.B. am Rand der Swimlane zu positionieren, damit klar wird, dass sie Konnektoren zu Elementen außerhalb dieses Graphen darstellen.

- Schnittmöglichkeiten ausbauen:
  Im Kapitel 7 wurde ein Algorithmus vorgestellt, der rekursiv einen BPMN-Graphen in Teile für Blätter zerschneidet. Für die Implementierung der Schnittmöglichkeiten können diese nun insoweit ausgebaut werden, dass auch auf Aufteilungen mit anderen Faktoren ermöglicht werden, z.B. einen Schnitt in drei, vier, fünf,... Teile. Dazu können mehrere Mittelkanten in geometrisch gleichen Abständen eingefügt werden und dann die Schnittkanten nacheinander für jede Mittelkante berechnet werden.

9 Zusammenfassung

In dieser Arbeit wurde ein Layoutalgorithmus für Geschäftsprozesse vorgestellt. Der Algorithmus basiert auf den theoretischen Ansätzen des TSM-Framework und des Sugiyama-Ansatzes. Die beiden Methoden wurden kombiniert und auf die Anforderungen von Modellen von Geschäftsprozessen übertragen. Die Anforderungen wurden aus der Syntax der Notationssprache BPMN abgeleitet. Die in Kürze von der Object Management Group (OMG) standardisierte Sprache zur Visualisierung von Geschäftsprozessmodellen erlaubt ein umfangreiches und umfassendes graphisches Festhalten von Prozessen anhand von einfachen Diagrammen. Die Einfachheit bezieht sich dabei auf die Verständlichkeit durch den Benutzer. Die von BPMN vorgegebenen Regeln und deren Funktionsumfang bilden die Basis für die Funktionalität, die der Layoutalgorithmus und die ihn beinhaltende Anwendung bieten muss. Dazu gehören die Unterstützung von Swimlanes, verschiedene Arten von Kanten, mehrere Knoteneigenschaften, Berücksichtigung von Prozessflüssen u.v.m.
Das Ziel des Layoutalgorithmus ist neben der Einhaltung der von BPMN vorgegebenen Regeln vor allem die Anwendung von ästhetischen Kriterien bei der Berechnung der Einbettung in ein orthogonales Gitter. Die ästhetischen Kriterien sind u.a. Anzahl der Kantenknicke und Kreuzungen, Länge von Kanten und benötigte Fläche des gesamten Diagramms. Der Layoutalgorithmus produziert aus einem Eingabegraph, der aus BPMN-Elementen besteht, eine Einbettung in ein orthogonales Gitter, sodass die Swimlanes in Rechtecken dargestellt werden und die Kanten orthogonal verlaufen.
Die Qualität der vom Layoutalgorithmus produzierten Ergebnisse hängt wesentlich von der Dichte des Graphen ab. Da Kanten in einer orthogonalen Gittereinbettung einen Mindestabstand haben müssen, damit sie sich nicht überlagern, führen viele Kanten zu breiten „Kantenbahnen“ in einem Layout, s. Abb. 9.1. Ebenso führt eine höhere Dichte zu einer wesentlich höheren Laufzeit, da die benötigte Laufzeit durch die heuristische Lösung des Netzwerkflussproblems mindestens quadratisch von der Anzahl der Kanten abhängt. Daher ist ein ästhetisch wertvolles Layout nur für dünne Graphen von $20-100$ Knoten zu erreichen. Dies ist für die Praxis jedoch ausreichend, da ein größerer Prozess nur selten auch implementierbar ist und damit auch nicht modelliert werden muss. Die Prozesse aus der Praxis haben zudem die Eigenschaften, dass sie wenige zentrale Knoten mit hohem Grad besitzen, während viele Knoten nur niedrigen Grad besitzen. Somit bieten reale Prozessmodelle gute Eigenschaften für ein automatisches Layout.

Der Layoutalgorithmus wurde in eine graphische Anwendung integriert, die ebenso die Modellierung von Prozessen von Grund auf ermöglicht. Dabei werden alle Elemente von BPMN unterstützt. Auch strukturell werden die von BPMN angebotenen Swimlanes verwendet. Lediglich Pools werden nicht direkt unterstützt, können

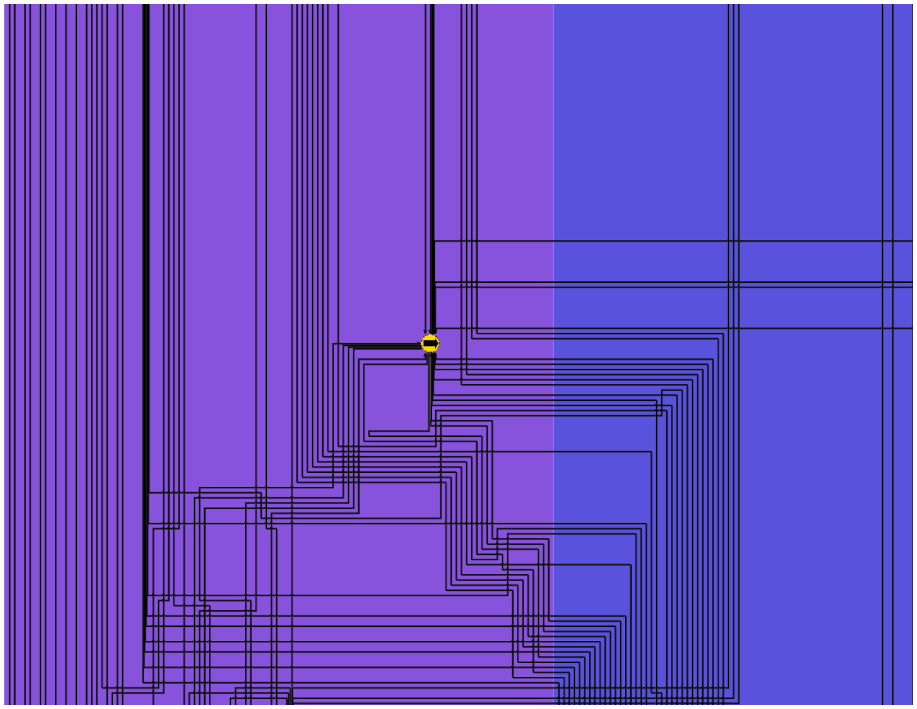


Abbildung 9.1: Ausschnitt aus einem größeren Layout, das auf einem sehr dichten Graphen basiert. Die breiten Kantenbahnen um einen einzelnen Knoten sind klar erkennbar.

aber durch leere Swimlanes simuliert werden. Die Anwendung erlaubt auch die Verwendung von anderen Layoutalgorithmen, wie z.B. einem gewöhnlich orthogonalen, einem organischen oder einem zirkulären Layoutalgorithmus. Zur Benutzerunterstützung kann die Ansicht verändert und angepasst werden und berechnete Layouts oder Prozessmodellskizzen auch exportiert oder importiert werden. Bei der Arbeit mit mehreren Layouts kann durch eine Redo-/Undo-Funktion auch ein unerwünschtes Layout rückgängig gemacht und ein subjektiv wünschenswerteres aufgerufen werden.
Das Sketch-Driven-Layout ist ein wesentliches Werkzeug zur interaktiven Arbeit mit Diagrammen, da bei einer Änderung nicht ein komplett neues Layout berechnet werden muss, sondern das existierende, vom Benutzer so gewollt vorgegebene Layout bzw. die Skizze mit einbezogen wird. Daher ist Sketch-Driven für die hier entwickelte BPMN-Builder-Anwendung unverzichtbar geworden.
Das neue Konzept der Konnektoren erlaubt eine deutliche Reduzierung der Größe der internen, orthogonalen Repräsentation, da durch die Löschung von vielen kleinen Kantensegmenten nicht nur diese selbst, sondern auch viele Kreuzungsknoten aus der Datenstruktur entfernt werden können. Optisch sind die Konnektoren als Anhänger zu den Knoten eingepasst, welche die Kante als Start- bzw. Zielknoten besaß. Dabei wird hier das letzte bzw. erste Kantensegment zum Anschluss an den Knoten verwendet. Dies führt unter ästhetischen Gesichtspunkten nicht immer zu optimalen Ergebnissen. Hier kann eine Verbesserung an einem ansprechenderen Anschluss der Konnektoren, z.B. an einer noch kantenlosen Seite des Knotens, ansetzen. Die Ge-

samtlaufzeit wird durch das Einfügen von Konnektoren nicht wesentlich erhöht. Am aufwändigsten ist die Berechnung der Bösartigkeit einer Kante. Stark abhängig ist die Laufzeit von der Anzahl der Kanten, da $O(|E^2|)$ viele Berechnungen der Bösartigkeiten vorgenommen werden müssen.
Der Ansatz zur Schnittberechnung stellt eine gute Möglichkeit dar, um ein größeres Prozessmodell auf mehrere Blätter zu schneiden. Dabei wird je nach Abbruchkriterium nach einer optimalen Aufteilung des Graphen gesucht, wobei die mögliche Schnittkante in einem immer größer werdenden Mittelstreifen gesucht wird.

Abbildungsverzeichnis

Literaturverzeichnis

[AIM91] Asano, T., H. Imai und A. Mukaiyama: *Finding a maximum weight independent set of a circle graph.* IEICE Transactions, E74(4):681–683, 1991.

[BDLN01] Binucci, C., W. Didimo, G. Liotta und M. Nonato: *Labeling Heuristics for Orthogonal Drawings.* Proceedings of the 9th International Symposium on Graph Drawing (GD'2001), Bd. 2275 d. Reihe LNCS:139–153, 2001.

[BJM97] Brandenburg, J.F., M. Jünger und P. Mutzel: *Algorithmen zum automatischen Zeichnen von Graphen.* Angewandte Mathematik und Informarik, Universität zu Köln, Report 97.264, 1997.

[BKWE02] Brandes, Ulrik, Michael Kaufmann, Dorothea Wagner und Markus Eiglsperger: *Sketch-Driven Orthogonal Graph Drawing.* Lecture Notes In Computer Science, 2528, 2002.

[Bra99] Brandes, Ulrik: *Layout of Graph Visualizations.* Doktorarbeit, University of Konstanz, http://www.ub.uni-konstanz/kops/volltexte/1999/255/, 1999.

[Bra01] Branke, Jürgen: *Dynamic graph drawing.* Drawing graphs: methods and models, Seiten 228–246, 2001.

[BT98] Bridgeman, Stina S. und Roberto Tamassia: *Difference Metrics for Interactive Orthogonal Graph Drawing Algorithms.* In: *GD '98: Proceedings of the 6th International Symposium on Graph Drawing*, Seiten 57–71, London, UK, 1998. Springer-Verlag.

[BW97] Brandes, Ulrik und Dorothea Wagner: *A Bayesian Paradigm for Dynamic Graph Layout.* In: *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, Seiten 236–247, London, UK, 1997. Springer-Verlag.

[CLRS01] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[DDPP99] DiBattista, Guiseppe, Walter Didimo, Maurizio Patrignani und Maurizio Pizzonia: *Orthogonal and Quasi-upward Drawings with Vertices of Prescribed Size.* Lecture Notes In Computer Science, 1731, 1999.

[Eff08] Effinger, Philip: *Automatisches Layout von Geschäftsprozessen.* Graph drawing, BPMN modeling, University of Tuebingen, Wilhelm-Schickard-Insitute, Sand 13, May 2008.

[Eic99] Eichelberger, Holger: *Einwicklung eines Frameworks zum automatischen Zeichnen von Software-Entwurfsdiagrammen.* Diplomarbeit, Universität Würzburg, 1999.

[Eig03] Eiglsperger, Markus: *Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach.* Doktorarbeit, Universität Tübingen, 2003.

[EK02] Eiglsperger, Markus und Michael Kaufmann: *Fast compaction for orthogonal drawings with vertices of prescribed size.* In: *In Proceedings of the 9th International Symposium on Graph Drawing (GD '01)*, Seiten 124–138. Springer, 2002.

[EKE00] Eiglsperger, Markus, Michael Kaufmann und Frank Eppinger: *An Approach for Mixed Upward Planarization.* Lecture Notes in Computer Science, 2125:352–??, 2000.

[ESK05] Eiglsperger, Markus, Martin Siebenhaller und Michael Kaufmann: *An efficient implementation of sugiyama's algorithm for layered graph drawing.* In: *Proceedings of the 12th Symposium on Graph Drawing (GD '04)*, Band 3383, Seiten 155–166. Springer, 2005.

[ESML91] Eades, Peter, Kozo Sugiyama, Kazuo Misue und Wei Lai: *Preserving the mental map.* In: *Proceedings of Compugraphics*, 1991.

[EW94] Eades, P. und N.C. Wormald: *Edge crossings in drawings of bipartite graphs.* Algorithmica, 11:379–403, 1994.

[EWHK+96] Emden-Weinert, Thomas, Stefan Hougardy, Bernd Kreuter, Hans Jürgen Prömel und Angelika Steger: *Einführung in Graphen und Algorithmen.* Online-Skriptum des Lehrstuhl für Algorithmen und Komplexität (HU Berlin), 1996.

[FK95] Foessmeier, Uwe und Michael Kaufmann: *Drawing High Degree Graphs with Low Bend Numbers.* In: *Proc. 4th Symposium on Graph Drawing (GD'95), LNCS 1011, pp.* Springer-Verlag, 1995.

[Fri97] Frick, A.: *Upper bounds on the number of hidden nodes in sugiyama's algorithm.* Proceedings of the 4th International Symposium on Graph Drawing (GD'96), 1190 of LNCS:169–183, 1997.

[GT94] Goldschmidt, O. und A. Takvorian: *An efficient graph planarization two-phase heuristic.* Networks, 24:69–73, 1994.

[GT96] Garg, Ashim und Roberto Tamassia: *A new Minimum Cost Flow Algorithm with Applications to Graph Drawing.* In: *Graph Drawing*, Band 96 der Reihe *Graph Drawing*, Seiten 201–216, 1996.

[JJDG06] Jucknath-John, Susanne, D.Graf und G.Taentzer: *Evolutionary Layout: Preserving the Mental Map during the Development of Class Models.* In: *ACM Symposium on Software Visualization (SoftVis 2006)*, 2006.

[KM99] Klau, G. W. und P. Mutzel: *Combining Graph Labeling and Compaction.* Proceedings of the 7th International Symposium on Graph Drawing (GD 99), Nr. 1731 in LNCS:27–37, 1999.

[Kre05] Krempel, Lothar: *Visualisierung komplexer Strukturen.* Campus Verlag, 2005. Grundlagen der Darstellung mehrdimensionaler Netzwerke.

[KRRS94] Klein, P. N., S. Rao, M. Rauch und S. Subramanian: *Faster shortest-path algorithms for planar graphs.* In: *Proceedings ACM Symp. on Theory of Computing*, 1994.

[KV06] Kalnins, Audris und Valdis Vitolins: *Use of UML and Model Transformations for Workflow Process Definitions.* Communications of the 7th International Baltic Conference on Databases and Information Systems, Seite 12, 2006.

[Len90] Lengauer, Thomas: *VLSI Theory.* In: *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, Seiten 835–868. Elsevier and MIT Press, 1990.

[LLY06] Lee, Yi-Yi, Chun-Cheng Lin und Hsu-Chun Yen: *Mental Map Preserving Graph Drawing Using Simulated Annealing.* In: *Asia Pacific Symposium on Information Visualization 2006 (APVIS 2006), Tokyo, Japan, February 2006. Conferences in Research and Practice in Information Technology, Vol. 60. K. Misue, K. Sugiyama and J. Tanaka, Eds.*, 2006.

[MK98] Mutzel, Petra und Gunnar W. Klau: *Quasi-orthogonal drawing of planar graphs.* Technischer Bericht MPI-I-98-1-013, MPI Saarbrücken, 1998.

[Nor97] North, Stephen C. (Herausgeber): *Graph Drawing, Symposium on Graph Drawing, GD '96,Berkeley, California,USA*, Band 1190 der Reihe *Lecture Notes in Computer Science.* Springer, 1997.

[OMG07] Object-Management-Group: *OMG - BPMN 2.0 Request for Proposal*, 2007. OMG Document: BMI/2007-06-05.

[OR03] Owen, Martin und Jog Raj: *BPMN and Business Process Management.* Popkin Software, 2003.

[Pur97] Purchase, Helen: *Which aesthetic has the greatest effect on human understanding.* In: *Proceedings of the 5th Symposium on Graph Drawing (GD '97)*, Band 1353, Seiten 248–261. Springer, 1997.

[RR97] Resende, Mauricio und Celso C. Ribeiro: *A grasp for graph planarization.* NETWORKS: Networks: An International Journal, 29, 1997.

[SF07] Schattkowsky, Tim und Alexander Förster: *On the Pitfalls of UML 2 Activity Modeling.* International Workshop on Modeling in Software Engineering (MISE'07), 2007.

[Sie03] Siebenhaller, Martin: *Automatisches Layout von UML-Klassendiagrammen.* Diplomarbeit, Universität Tübingen, jun 2003.

[SK05] Siebenhaller, Martin und Michael Kaufmann: *Mixed upward planarization - fast and robust.* In: *Proceedings of the 13th Symposium*

on Graph Drawing (GD '05), Band 3843, Seiten 522–523. Springer, 2005.

[SK06] Siebenhaller, Martin und Michael Kaufmann: *Drawing Activity Diagrams.* WSI-2006-02, 2006.

[Sti06] Stiege, Günther: *Graphen und Graphalgorithmen.* Informatik. Shaker Verlag, Aachen, 2006.

[STT81] Sugiyama, K., S. Tagawa und M Toda: *Methods for visual understanding of hierarchical system structures.* IEEE Transactions on Systems, Man, and Cybernetics, SMC-11,2:109–125, 1981.

[SW97] Stoer, Mechthild und Frank Wagner: *A simple min-cut algorithm.* ACM, 44(4):585–591, 1997. 0004-5411.

[Tam87] Tamassia, Robert: *On Embedding a Graph in the Grid with the Minimum Number of Bends.* SIAM Journal on Computing, 16(3):421–444, 1987.

[TDET99] Tamassia, Robert, Guiseppe DiBattista, Peter Eades und Ioannis Tollis: *Graph Drawing.* Prentice Hall, 1999.

[Whi04a] White, Stephen A.: *Introduction to BPMN.* bpmn.org, Mai 2004.

[Whi04b] White, Stephen A.: *Process Modeling Notations and Workflow Patterns.* bpmn.org, Seite 25, Januar 2004.

[Whi05] White, Stephen A.: *Using BPMN to Model a BPEL Process.* http://www.bpmn.org, Seite 18, Februar 2005.

[WS03] Wang, S. und J. Siskind: *Image segmentation with ratio cut.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 25(6):675–690, 2003.

[yG07] GmbH yWorks: *yFiles Developer's Guide.* yWorks GmbH, 2007.