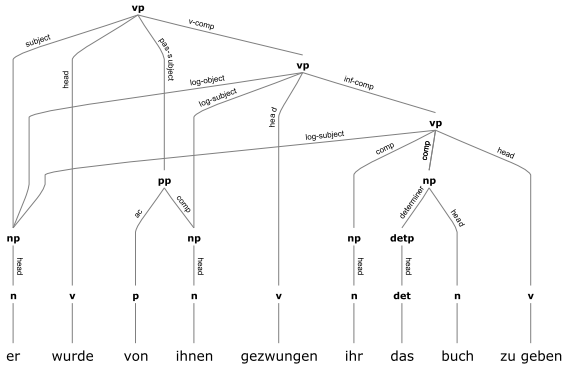
Darstellung von Abhängigkeitsgraphen



Sandro Blum

Semesterarbeit SA-2007-11

Wintersemester 2006/07

Institut für Technische Informatik
und Kommunukationsnetze

Betreuer: Tobias Kaufmann

Verantwortlicher: Prof. Dr. L. Thiele

Contents

Kurzzusammenfassung

Sowohl in der Spracherkennung als auch in der Sprachsynthese werden verschiedene Grammatikformalismen zur Textanalyse eingesetzt. Hierbei entstehen sogenannte Dependenzgraphen (siehe 1.1), welche die Zusammenhänge zwischen den sprachlichen Einheiten des analysierten Texts beschreiben. Ein wiederkehrendes Problem ist nun, die entstandenen *abstrakten* Dependenzgraphen übersichtlich in einer graphischen Form darzustellen.

Meine Semesterarbeit hatte zum Inhalt, ein solches System zur optisch ansprechenden Visualisierung von Dependenzgraphen zu entwerfen und zu implementieren. (siehe 1.2) Hervorzuheben ist ebenfalls die zusätzliche Anforderung an das System, bezüglich Darstellungsarten erweiterbar zu sein. Abhängigkeitsgraphen können sehr unterschiedlich dargestellt werden, was nicht zuletzt davon abhängt, welche Aspekte hervorgehoben werden sollen. Aus diesem Grund ist es äusserst sinnvoll, die Layout Algorithmen von den darstellungsspezifischen Komponenten abzukapseln und eine Schnittstelle zu entwerfen, über welche sich das System mit konkreten Darstellungsarten erweitern lässt.

Nebst der Erweiterbarkeit bestand die Herausforderung in meiner Semesterarbeit hauptsächlich im Entwurf und in der Implementierung von Layout-Algorithmen, welche genug allgemein sein würden, um Dependenzgraphen in all ihrer Vielfältigkeit darstellen zu können.

Im Verlaufe dieses Berichts versuche ich zu schildern, wie ich die eben erwähnten Schwierigkeiten gelöst habe. Um das Resultat dieser Semesterarbeit auch visuell zu betrachten, sind im Anhang dieses Berichts noch einige Beispiele von visualisierten Dependenzgraphen mit unterschiedlichen Darstellungsarten aufgeführt, welche von dem in dieser Semesterarbeit entworfenen und implementierten Programm erzeugt worden sind.

1 Problemstellung

1.1 Dependenzgraphen

Für den weiteren Bericht ist es hilfreich zu verstehen, in welcher Vielfalt Dependenzgraphen auftreten können und was Dependenzgraphen überhaupt sind. Dies soll im folgenden Abschnitt kurz erläutert werden.[1]
Wie in der Einleitung bereits erwähnt, beschreibet ein Dependenzgraph die Beziehungen zwischen sprachlichen Einheiten eines Texts. Diese sprachlichen Einheiten (oder *Konstituenten*) entsprechen Worten oder Phrasen und werden innerhalb des Graphen durch einen Knoten dargestellt. Die Blattknoten entsprechen hierbei im allgemeinen einzelnen Wörtern und auf ihnen ist eine totale Ordnung definiert, welche der Reihenfolge ihres Auftretens im Text entspricht. Die Kanten innerhalb des Graphen stellen die Abhängigkeitsbeziehungen zwischen den Konstituenten dar.

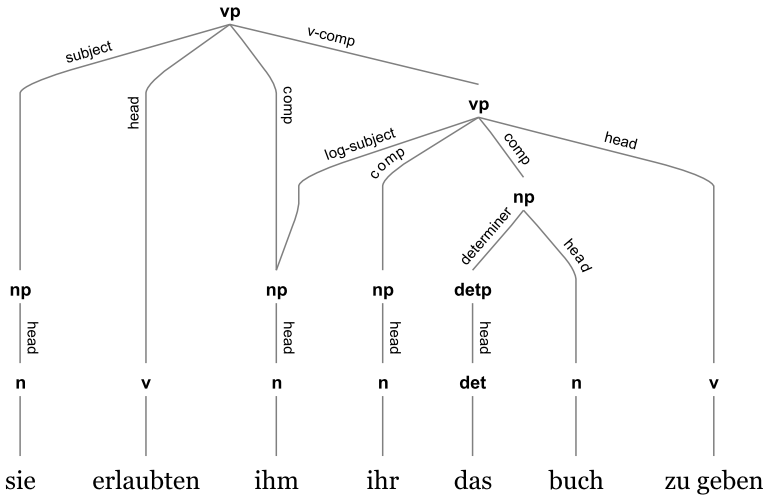


Illustration 1.1 - Beispiel eines simplen visualisierten Dependenzgraphen

[1] Weiterführende Informationen, insbesondere bezüglich dem linguistischen Informationsgehalt eines Dependenzgraphen, sind unter [1] zu finden.

Ein Dependenzgraph hat folgende Eigenschaften:

- er ist ein gerichteter Graph
- er besitzt genau einen Knoten ohne eingehende Kanten (Wurzel)
- auf den Blattknoten ist eine totale Ordnung definiert
- jeder Knoten (ausser die Wurzel) *kann* mehrere Eltern besitzen
- es können Zyklen auftreten[2]

Ein Dependenzgraph kann also in einfacheren Fällen eine Baumstruktur aufweisen, kann im allgemeinen jedoch auch viel komplexer aufgebaut sein.[3]

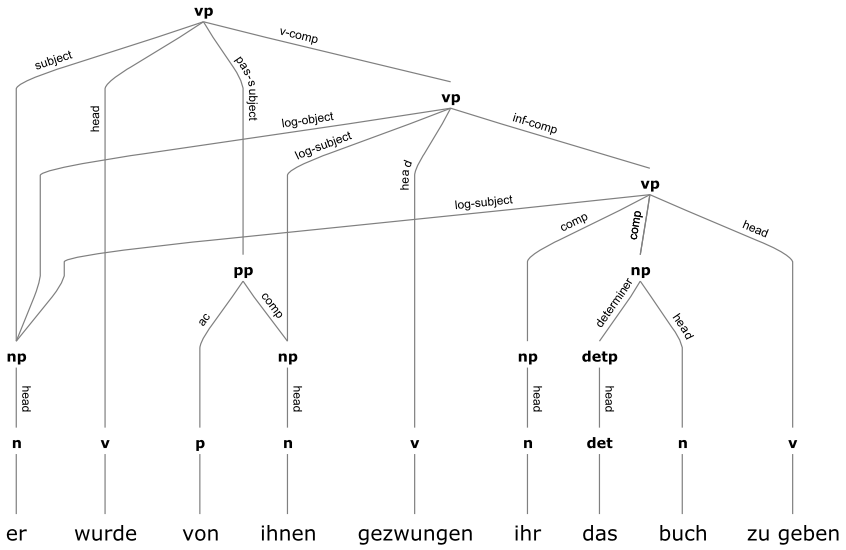


Illustration 1.2 - Beispiel eines visualisierten Dependenzgraphen mit Kantenkreuzungen

[2]Zyklen treten nur in seltenen Fällen auf und werden vom vorgestellten System nicht behandelt. Man bemerke, dass für eine Anpassung an Zyklen keine Änderung in den Layoutalgorithmen notwendig wären. (wenn man zyklische Kanten zu Beginn markiert und in der Richtung dreht, kann mit einem pseudo-nichtzyklischen Graphen weitergearbeitet werden)

[3]Insbesondere kann es sein, dass man einen Dependenzgraphen planar nicht ohne Kantenüberkreuzungen auslegen kann.

1.2 Aufgabenstellung

Wie bereits eingangs erwähnt umfasst diese Semesterarbeit den Entwurf und die Implementierung eines Systems zur Visualisierung von Dependenzgraphen, welches mit (von den Layout Algorithmen unabhängigen) Darstellungsarten erweitert werden kann. Konkret umfasst die Aufgabenstellung folgende Teilaufgaben:

- Spezifikation eines Eingabeformats zur Beschreibung von (abstrakten) Abhängigkeitsgraphen
- Spezifikation eines Ausgabeformats zur Beschreibung der konkreten Visualisierung
- es soll eine Möglichkeit geschaffen werden, die Ausgabedatei anzuzeigen
- Implementierung von geeigneten Layout-Algorithmen
- Entwicklung einer Schnittstelle zwischen Layout-Algorithmen und darstellungsspezifischen Komponenten
- Implementierung von zwei Darstellungsarten (zur Demonstration der Erweiterbarkeit)

Eine Kopie der ausformulierten Aufgabenstellung des Instituts ist im Anhang E zu finden.

2 Lösungsansätze

Das automatische Zeichnen von Graphen ist in der Wissenschaft ein Gebiet, in welchem seit einigen Jahren intensiv geforscht wird und es gibt selbst Symposien, die sich ausschliesslich mit der Visualisierung von Graphen befassen. Anders als man im ersten Moment vielleicht denken könnte, bietet die Graphenvisualisierung hinsichtlich der Komplexität grosse Herausforderungen. Als Beispiel sei hierzu das Problem der Minimierung von Kantenüberkreuzungen in gerichteten Graphen genannt, von welchem gezeigt wurde (siehe [2]), dass es NP-schwierig ist. Da bereits einige Implementationen von komplexen Graphenvisualisierungsalgorithmen existieren, welche die eben angesprochenen Schwierigkeiten mit heuristischen Ansätzen zu lösen versuchen, habe ich mir die Frage gestellt, ob es überhaupt sinnvoll sein würde, ein Visualisierungssystem von Grund auf neu zu programmieren. Um diese Frage beantworten zu können, habe ich mir in Form von *GraphViz* (siehe [4]) eine gängige Graphenvisualisierungs-Software und deren implementierte Algorithmen (siehe [3]) angeschaut. Hierbei bin ich zum Schluss gekommen, dass die Visualisierung von Dependenzgraphen (in der Form wie sie für diese Semesterarbeit gebraucht werden) zu spezifische Anforderungen an die Layout-Algorithmen von *GraphViz* und wohl auch an andere Graphenvisualisierungssoftware stellen würde und man mit einer eigens auf das Problem angepassten Software bessere Resultate erhalten würde.
Hierbei möchte ich noch anmerken, dass ich nichtsdestotrotz zwei der in [3] vorgestellten Ideen zur Lösungen von Teilproblemen in der Layouting-Phase auch für meinen Layout-Algorithmus übernommen habe. [4]

Konkret eine kurze Auflistung der Probleme welche sich bei der Verwendung von *GraphViz* ergeben hätten:

- Es ist schwierig oder teilweise sogar unmöglich, die totale Ordnung auf den Blättern zu gewährleisten, insbesondere, wenn sich diese auf verschiedenen Höhenebenen befinden.
- Konkrete Darstellungsarten von Knoten und Kanten sind auf die Möglichkeiten von *GraphViz* eingegrenzt und nicht wirklich erweiterbar.

[4] Konkret sind dies: Erstens die strikte Trennung von relativem und absoluten Knotenaufbau, zweitens die Idee der Verwendung von Dummy-Knoten. Hierbei habe ich aber lediglich die Idee und nicht etwa eine konkrete Implementierung übernommen.

- Kantenverläufe und Knotenplatzierungen hängen von kompliziertem Gewichtungssystem ab, was teilweise zu unerwünschten Resultaten führen kann. Ein Wrapper hätte schlicht zu wenig Kontrolle über die Resultate.

3 System zur Visualisierung von Dependenzgraphen

Im folgenden stelle ich das in der Semesterarbeit entworfene und implementierte System vor.

Im Abschnitt 3.1 gebe ich als erstes einen kurzen Überblick über das Design des Systems im groben und stellte in einer knappen Form die wichtigsten Komponenten vor. Hierbei gehe ich insbesonders bei den Layout-Algorithmen noch nicht auf Details ein, dies folgt später in einem separaten Abschnitt.

Im Abschnitt 3.2 erläutere ich dann wo und wie sich die Flexibilität und Erweiterbarkeit des Systems zeigt.

Abschnitt 3.3 befasst sich dann kurz mit dem Eingabeformat. Hierbei wird ersichtlich, welche Daten in einer Eingabedatei enthalten sind und teilweise auch bereits, welcher Funktion sie dienen.

Der zweitletzte Abschnitt (3.4) ist dann etwas länger und befasst sich etwas ausführlicher mit den angewendeten Layout-Algorithmen. Hierbei muss ich darauf hinweisen, dass ich nicht alle Algorithmen bis ins kleinste Detail vorstellen, sondern lediglich einen überblick geben kann. Da dies auch so bisweilen noch etwas trocken sein kann, habe ich versucht, dies - wo möglich - mit Bildern zu untermalen.

Im letzten Abschnitt (3.5) befasse ich mich dann noch in kurzer Form mit dem Ausgabeformat des Systems. Hierbei gehe ich mehr auf die Wahl des Formats als auf das Format selber ein, da es dazu bereits vielerlei Quellen im Internet gibt.

3.1 Design

3.1.1 Überblick

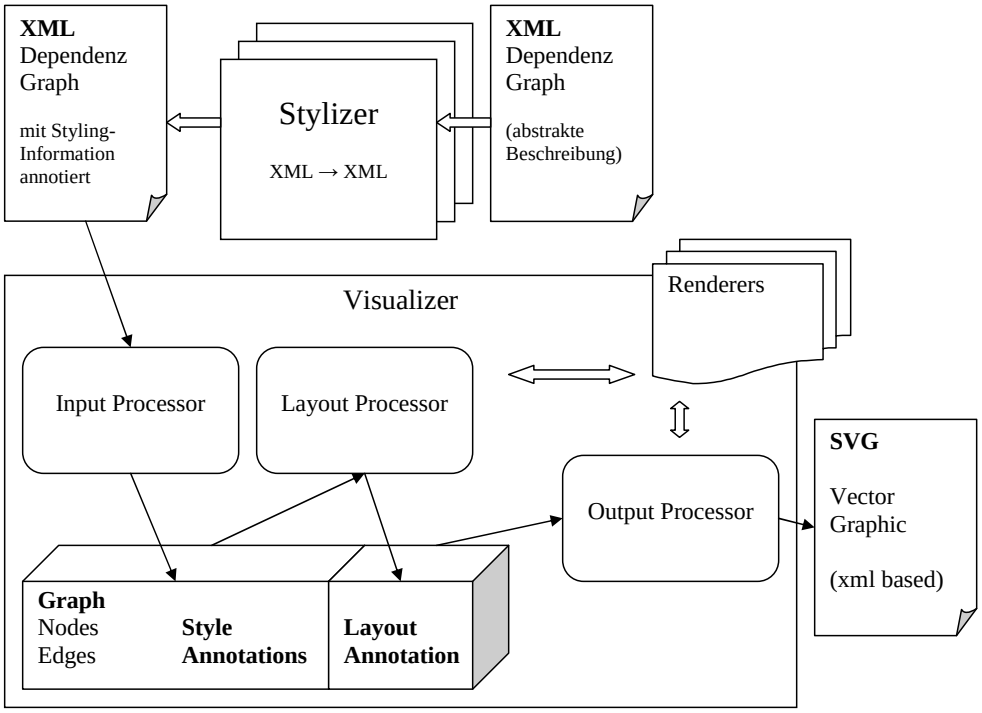


Illustration 3.1 - Übersicht System Design

Grober Ablauf:

1. Der Stylizer annotiert eine XML Datei eines gegebenen abstrakten Graphen mit Styling Information.
2. Diese annotierte XML Datei wird dann vom Input Processor des Visualizers eingelesen und in eine interne Darstellung umgewandelt.
3. Der Layout Processor berechnet unter Einbezug der Renderer die Positionen der Knoten und Kanten.
4. Der Output Processor setzt mit Hilfe der Renderer und den eben berechneten Positionen die Knoten und Kanten zu einer Grafik zusammmen.

3.1.2 Stylizer

Der Stylizer ist ein vom Visualizer unabhängig funktionierendes Programm. Seine Funktion liegt darin, eine XML Datei welche einen abstrakten Dependenzgraphen beschreibt, um Styling Information zu erweitern.[5] Styling Informationen enthalten folgende Informationen:

- Für jeden Knoten und jede Kante wird definiert, welcher Node- bzw. Edge-Renderer verwendet werden soll und mit welchen (renderer-spezifischen) Parametern.
- Von den Knoten und Kanten unabhängige Layout-Parameter, welche die Funktionsweise des Layout Processors steuern.

3.1.3 Input Processor

Der Input Processor liest eine XML-Datei eines Dependenzgraphen ein und erstellt die entsprechenden internen Datenstrukturen, hierbei werden folgende Informationen eingelesen:

- Struktur des Graphen (Knoten und Kanten)
- Styling Informationen (siehe vorheriger Abschnitt)
- den Knoten und Kanten zugeordnete Daten (sogenannte Data Chunks[6])

3.1.4 Layout Processor

Der Layout Processor berechnet über einige Zwischenschritte (siehe 3.4) die folgenden Informationen:

- für jeden Knoten die exakte Position innerhalb des resultierenden Graphikdokuments

[5]Dies ist grundsätzlich auch von Hand möglich, insofern ist der Stylizer nur ein Hilfstool welches die Arbeit erleichtert.

[6]Ein Data Chunk ist ein Typ-Wert Paar, welches Informationen zum entsprechenden Knoten oder der entspechenden Kante enthält. Knoten und Kanten können eine unbegrenzte Anzahl von Data Chunks jeglichen Typs enthalten.

- für jede Kante die Start- und End-Position sowie allfällige Zwischenpositionen

- die Dokumentgrösse

Bei der Berechnung der Knotenpositionen greift der Layout Processor für jeden Knoten auf den entsprechenden in den Styling Informationen definierten Node-Renderer zu, um die für die Berechnungen notwendige Bounding Box[7] des Knotens zu erhalten.

3.1.5 Output Processor

Der Output Processor erstellt ein SVG Dokument (siehe 3.5) und fügt unter Verwendung der Knoten- und Kantenrenderer die entsprechenden Grafik-Elemente in das Dokument ein.

3.1.6 Node & Edge Renderers

Renderer müssen ein definiertes Interface implementieren und haben die Funktion, einen Knoten oder eine Kante - je nach Renderer-Typ - in SVG umzuwandeln. Zusätzlich haben Renderer die folgenden Eigenschaften:

- Jeder Renderer besitzt eine freie Anzahl von Parametern welche sein Rendering steuern. (z.B die Textfarbe, Kantendicke etc.) Die Parameterwerte werden durch Angaben in den Styling Informationen gesetzt, andernfalls besitzt jeder Renderer Standartwerte.

- Renderer haben Zugriff auf Data Chunks jedes Knotens und jeder Kante. Der Renderer kann diese Daten verwenden, muss aber nicht.

- Knotenrenderer müssen ihrem Interface entsprechend die Bounding Box des Knotens berechnen können.

[7] Durch Knotenhöhe und -breite definiertes Rechteck

3.2 Flexibilität

Wenn man von Flexibilität spricht, kann man hier mindestens zwischen drei verschiedenen Arten unterscheiden. Für jede dieser Arten der Flexibilität möchte ich kurz erwähnen, wo diese im Visualisierungssystem auftreten:

1. Erweiterbarkeit

   - Um das Visualisierungssystem um weitere Darstellungsarten zu erweitern, sind keine Änderungen an den Layout Algorithmen notwendig. Das Interface, welches ein Knoten- oder Kanten-Renderer implementieren muss, ist simpel und setzt keine Kenntnisse über die Vorgänge innerhalb des Layout Processors voraus.

2. Parametrisierbarkeit

   - Jeder Renderer ist frei nach gutdünken des Entwicklers parametrisierbar. So besitzt beispielsweise der bereits implementierte Kanten-Renderer Parameter für die Kantenfarbe, Kantendicke, Rundheit der Kante .. usw.
   - Der Layout Processor besitzt eine fixe Anzahl von definierten Parametern, welche das Verhalten der Layout-Algorithmen beeinflussen.[8] Diese Parameter können in der XML Datei festgelegt werden, andernfalls werden Standardwerte übernommen.

3. Manuelle Eingriffsmöglichkeiten

   - Da das auf XML basierende Eingabeformat human-readable ist, können sowohl die abstrakte Struktur des Graphen, als auch die Styling Informationen (beispielsweise einzelne Renderer-Parameter) noch von Hand geändert werden.
   - Das Ausgabeformat SVG basiert ebenfalls auf XML, wodurch auch dieses noch von Hand verändert werden kann, sofern man das möchte.

[8] Dazu gehört beispielsweise, ob Blattknoten nach unten oder nach oben gezogen werden oder der Mindestabstand zwischen Knoten und ähnliche Layout-Optionen

3.3 Eingabeformat

Wie bereits erwähnt ist das Eingabeformat XML-basiert. Eine formale Definition des Eingabeformats in Form einer DTD ist diesem Bericht unter (4.3) angefügt. Damit eine Eingabedatei vom Visualizer verarbeitet werden kann, muss sie folgende Informationen enthalten:

- Die abstrakte Beschreibung des Dependenzgraphen wird in Form von Knoten und Kanten beschrieben, wobei die Knoten in Terminale (Blattknoten) und Nichtterminale (innere Knoten) unterteilt werden.
- Terminalen wie auch Nichtterminalen ist eine eindeutige ID zugeordnet, zudem enthalten sie einen Verweis auf einen Knoten-Renderer.
- Nichtterminal enthalten zudem ein Label welches ihre Funktion umschreibt, während Terminale die Position innerhalb der totalen Ordnung sowie das Wort (oder die Phrase) enthalten wofür sie stehen.
- Kanten enthalten die ID des Zielknotens sowie ein Label, einen Vermerk, ob es sich um eine sogenannte Head-Kante handelt und einen Verweis auf einen Kanden-Renderer.
- Renderer enthalten eine eindeutige ID sowie einen Klassennamen und optional eine beliebige Anzahl von Parametern.
- Terminale, Nichtterminale und Kanten enthalten ausserdem optional eine beliebige Menge von Data Chunks, welche wiederum aus einem Typ und einem Wert bestehen.

Ein gekürztes Beispiel einer Eingabedatei ist im Anhang C zu finden.

3.4 Layout Algorithmus

Die Aufgabe des Layout Processors besteht darin, für jeden Knoten eine Position und für jede Kante einen Verlauf zu bestimmen, so dass das resultierende Grafikdokument den Graphen möglichst schön darstellt.

3.4.1 Ästhetikkriterien

Wenn wir verschiedene Visualisierung desselben Graphen betrachten, dann werden uns oftmals einige der Darstellungen besser gefallen als andere. Wenn man nun einen Layout-Algorithmus entwickeln möchte, welcher Graphen optisch ansprechend und möglichst übersichtlich darstellt, dann muss man sich als erstes überlegen, welche Kriterien die wahrgenommene Übersichtlichkeit oder Ästhetik bestimmen.
Nachfolgend zähle ich die Kriterien auf anhand derer ich den Layout-Algorithmus entwickelt habe und die sich während meiner Arbeit als nützlich erwiesen haben. Selbstverständlich kann der Fall auftreten, dass ein Kriterium nur auf Kosten eines anderen Kriterium optimiert werden kann, deshalb wird nachfolgend zwischen Randbedingen und Optimierungs-Kriterien unterschieden. Hierbei können Optimierungskriterien nur soweit optimiert werden, wie dabei keine der Randbedingungen verletzt wird.

Randbedingungen:

- Knoten die einer (strukturellen) Höhe des Dependenzgraphen angehören, sollen auch so visualisiert werden, dass dies ersichtlich wird. Deshalb werden die Knoten einer Ebene nach oben bündig dargestellt.
- Knoten sollen zentriert über ihren Kindern dargestellt werden.[9]
- Eine Kante darf, auch wenn sie mehrere Ebenen überspannt, keine Knoten überkreuzen.
- Kanten müssen immer nach unten verlaufen und niemals nach oben.

Optimierungs-Kriterien (voneinander unabhängig):

- Vermeidung von nicht notwendigen Freiräumen innerhalb des Graphen.
- Bei Kanten die frei von überkreuzungen mit anderen Kanten gezeichnet werden können soll dies gemacht werden. Falls die nicht möglich ist, soll sie so gelegt werden, dass die Anzahl der überkreuzungen (mindestens lokal) minimiert wird.

[9] Beispielsweise soll ein Knoten mit nur einem Kind direkt über dem entsprechenden Kind dargestellt werden und nicht asymmetrisch links oder rechts

3.4.2 Verarbeitungsschritte

Das Problem für jeden Knoten und jede Kante die exakte Position zu bestimmen mag vielleicht nicht nach einem grossen Problem klingen, tatsächlich scheint es jedoch so, dass sich dieses Problem nur mit einer Vielzahl von Verarbeitungsschritten mit vielen Teilproblemen lösen lässt. In diesem Abschnitt erläutere ich nun die einzelnen Teilschritte wie sie der Layout Processor durchläuft und die Schwierigkeiten, auf welche ich dabei gestossen bin:

1. **Reduktion auf planaren Graphen**
   Im Allgemeinen muss die Planarität bei einem Dependenzgraphen nicht gegeben sein. Dies erschwert den Entwurf von Layout-Algorithmen ungemein. Wenn man nun jedoch Beispiele von praxisüblichen Dependenzgraphen betrachtet, so stellt man fest, dass üblicherweise nur sehr wenige Kanten entfernt werden müssen, um Planarität herzustellen. Dies legt die Idee nahe, temporär einige der Kanten im Graphen zu löschen und diese in einem späteren Teil des Layoutings wieder einzufügen.
   Dies ist denn auch genau was im ersten Schritt des Layout Processors passiert: Der Graph wird von den Blättern her zusammengesetzt wobei für jeden inneren Knoten die ausgehenden Kanten betrachtet werden und für jede Kante rekursiv der Blatt-Bereich berechnet wird[10], welcher von der entsprechenden Kante umschlossen wird. Ergeben sich hieraus mehrere Intervalle muss ein Intervall (welches unter Umständen durch mehrere Kanten gebildet wird) ausgewählt werden, alle den anderen Intervallen zugeordnete Kanten werden nun als temporär gelöscht markiert. (siehe Illustration 3.2)

[10] Zur Bereichsberechnung werden nur jenige Kanten betrachtet, welche nicht bereits als temporär gelöscht markiert sind.

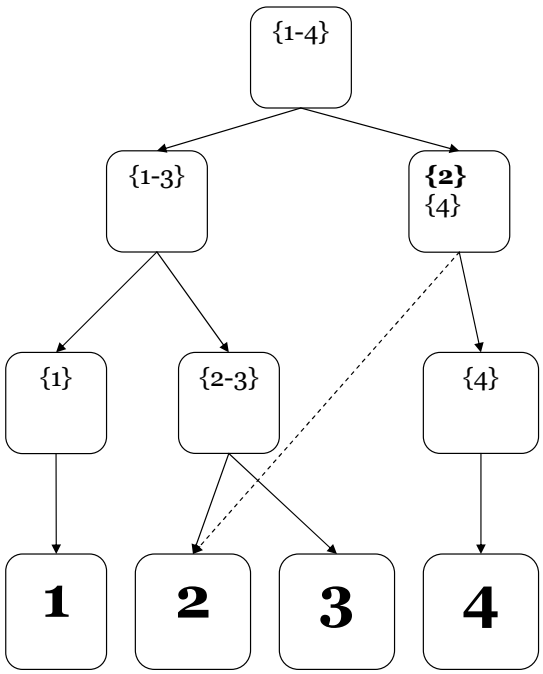


Illustration 3.2 - Erkennung einer zu löschenden Kante basierend auf Blatt-Bereichen.

2. **Relative Positionsberechnung**
   Da der Graph nun eine planare Struktur hat, ist es möglich für jeden Knoten eine relative Position zu bestimmen. Dazu wird der Baum von den Blättern her aufgebaut und für jeden Knoten eine Höhenachse sowie die Position innerhalb dieser Höhe im Vergleich zu den anderen Knoten dieser Höhe bestimmt.

3. **Einfügen von Dummy Knoten**
   Nun ist es so, dass durch Kanten verbundene Knoten teilweise mehr als nur eine Ebenen voneinander entfernt liegen. Dies hat zur Folge, dass man aufpassen muss, dass eine solche Kante nicht die Knoten der dazwischenliegenden Ebenen überkreuzt. Um dies zu gewährleisten und auch für die nachfolgenden Schritte ist es notwendig, entlang von mehrebigen Kanten

'Dummy Knoten' einzufügen. Diese Dummy Knoten sind speziell markiert und können somit von 'normalen' Knoten unterschieden werden.

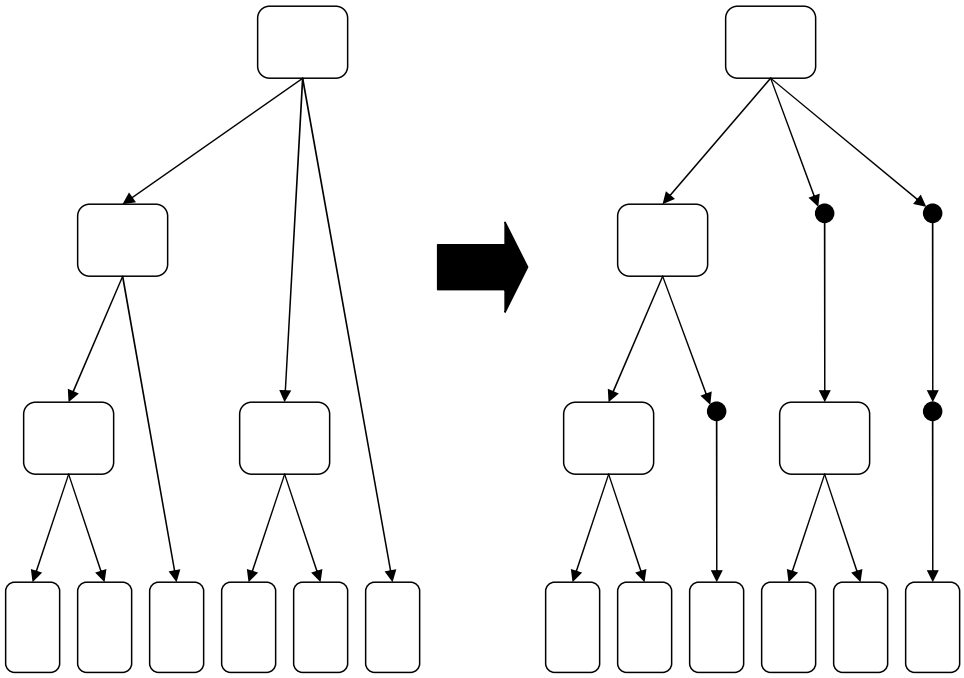


Illustration 3.3 - Einfügen von Dummy-Knoten.

4. **Verschiebung von Knoten entlang Dummy Knoten**
   Da wir den Baum bei der relativen Positionsbestimmung von unten her aufgebaut haben, befinden sich nun alle Blattknoten gemeinsam auf der untersten Ebene. Falls wir möchten, können wir nun alle Knoten entlang von Achsen mit Dummyknoten schrittweise soweit wir wollen nach oben verschieben. Dieses Verhalten ist mittels einem Layout-Parameter steuerbar. Beim nach oben verschieben muss aufgepasst werden, dass man einen Knoten nicht weiter nach oben schiebt als eine Ebene unter seinen nähesten Elternknoten. Ausserdem gibt es einige komplizierte Fallunterscheidungen wenn man bedenkt, dass die Kindknoten des zu verschiebenden Knotens auf den Seiten auch noch andere Eltern haben können und deshalb wieder neue Dummy Knoten eingefügt werden müssen. (siehe beispielsweise Illustration 3.4)

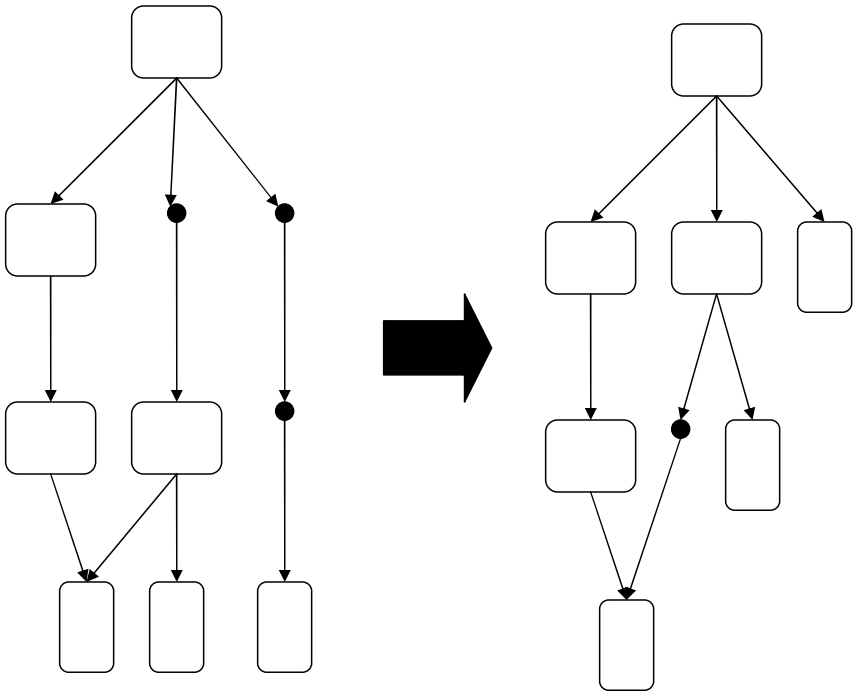


Illustration 3.4 - Knoten werden entlang Dummy-Achsen nach oben geschoben.

5. **Dummy Knoten entlang temporär gelöschten Kanten**
   Nun sind wir soweit, dass wir den Verlauf der bisher als gelöscht betrachteten Kanten berechnen können. Dazu müssen wir ähnlich wie im vorletzten Schritt auf jeder Zwischenebene einen Dummy Knoten einfügen. Nun ist jedoch so, dass diese Kante grundsätzlich durch jeden möglichen Knotenzwischenraum (Dummy Knoten mit eingeschlossen) hindurchlaufen kann. (siehe Illustration 3.5) Deshalb wählen wir aus allen möglichen Lösungen diejenige mit einer minimalen Anzahl von Kantenkreuzungen. Dies lässt sich am effizientesten mittels der dynamischen Programmierung lösen!

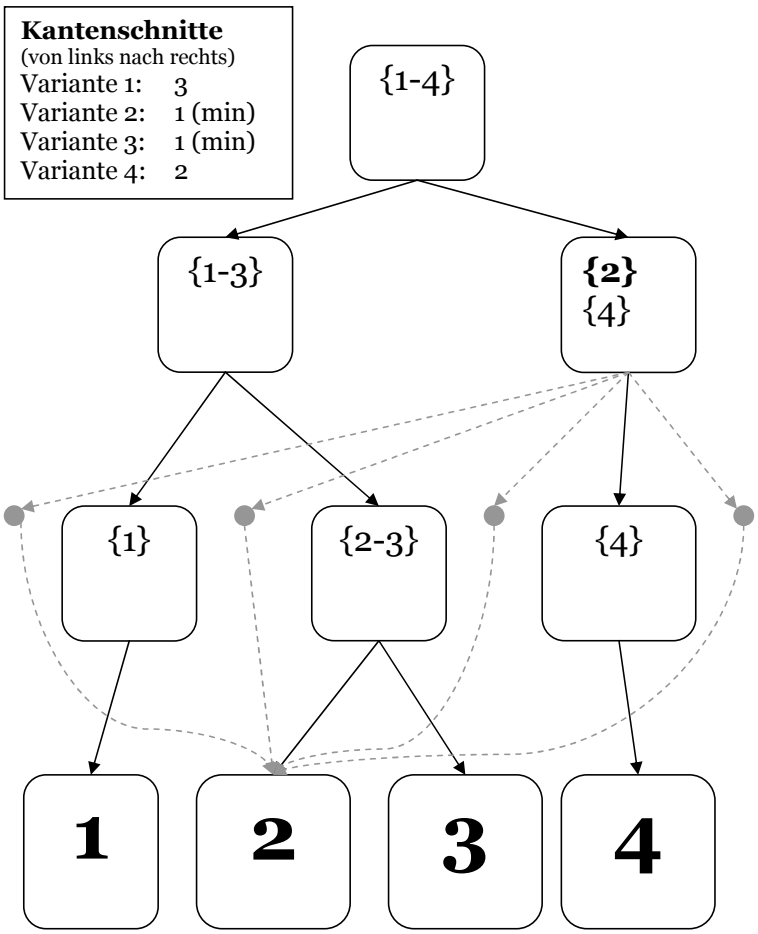


Illustration 3.5 - Beispiel möglicher Kantenverläufe (hier nur 1 Ebene!).

6. **Berechnen von absoluten Knotenpositionen**
   Zu diesem Zeitpunkt sind alle relativen Knotenanordnungen inklusive Dummyknoten aller Kanten gegeben. Der nächste Schritt besteht nun darin, aus dieser relativen Anordnung für jeden Knoten (und somit implizit auch für jede Kante) eine absolute Position zu berechnen. Obwohl die Implementierung in den Details nicht einfach ist, ist die Grundidee eher einfach: Man beginnt den Baum von links und von der untersten Ebene her aufzubauen.

Die Berechnung der y-Koordinaten ist simpel und diese sind jeweils für alle Knoten einer Ebene gleich. Hierbei ist anzumerken, dass für die Berechnung der y-Koordinaten (und x-Koordinaten) die Höhen (und Breiten) der einzelnen visualisierten Knoten bekannt sein müssen. Dies geschieht über eine Anfrage an den Knoten-Renderer.
Die x-Koordinaten werden vorerst als Mittelwert der x-Koordinaten der Kinder gewählt, wobei ursprünglich gelöschte Kanten nicht mitgezählt werden, Dummy-Knoten werden speziell behandelt. Nun kann es sein, dass die berechnete x-Koordinate kleiner ist als die des rechten Rands des vorhergehenden Knotens der gleichen Ebene. In diesem Fall muss die x-Koordinate des aktuell zu berechnenden Knotens angepasst und nach rechts verschoben werden. Damit der Graph nicht unsymmetrisch wird müssen auch seine Kinder und Kindeskinder mitverschoben werden. Der Umstand, dass ein Teil der Kinder noch andere Eltern haben können, führt hierbei zu einigen komplizierten Fallunterscheidungen.

7. **Umwandeln der Dummy Knoten in Kantenverläufe**
   Zum Schluss werden die Dummy Knoten wieder entfernt und in Form von Eckpunkten in den Kanten gespeichert. Weiter muss beachtet werden, dass Knoten von unterschiedlicher Höhe sein können. Nach oben sind alle Knoten einer Ebene bündig, nach unten aufgrund der unterschiedlichen Grössen selbstverständlich nicht. Dies kann dazu führen, dass eine Kante eines kleinen Knotens einen oder mehrere benachbarte grössere Knoten überkreuzt. Dies kann vermieden werden, indem die Kanten beim Knotenausgange zuerst ein kleines Teilstück weiter nach unten gezogen werden.

3.5 Ausgabeformat - SVG

Wie bereits erwähnt entspricht das Ausgabeformat dem SVG Standard. SVG steht für Scalable Vector Graphics und ist ein verbreiteter offener Standard zur Beschreibung zweidimensionaler Vektorgrafiken in der XML-Syntax.

Für die Wahl von SVG als Ausgabeformat sprechen vielfältige Gründe:

- mächtiger Standard
- human-readable

- Der SVG-Standard wird auf einer grossen Zahl von Plattformen und Software unterstützt. So können beispielsweise nahezu alle aktuellen Browser einen Teil des Sprachumfangs selbst ohne Plugin darstellen. Zudem gibt es eine grosse Zahl von Software und Libraries zur Konvertierung des Formats in gebräuchlichen Grafikformate oder gar zur direkten Bearbeitung von SVG.

- Vorteil der unbegrenzten Auflösung von Vektorgrafik gegenüber Rasterformaten. (lässt sich ohne Qualitätsverlust frei skalieren)

Für die Konvertierung von SVG in Rasterformate oder PDF möchte ich noch besonders auf die freie Library *Batik* von *Apache* verweisen. [5] Diese enthält unter anderem einen sehr ausgereiften 'Rasterizer' welcher sich für die eben erwähnte Aufgabe bestens eignet.

4 Schlussfolgerungen

4.1 Aufgabenstellung vs. Resultat

Sämtliche in der Aufgabenstellung enthalten Forderungen können mit dem vorgestellten und implementierten System erfüllt werden. Hierbei besondere Bedeutung erhält die Forderung der Erweiterbarkeit, hierzu wurde bereits im Abschnitt 3.2 aufgeführt, an welchen Stellen und inwiefern sie im System zutrifft. Auch wenn die Bewertung der visuellen Ausgabe schlussendlich subjektiv ist, kann man meiner Meinung nach doch sagen, dass die vom System erzeugten Graphen-Visualisierungen übersichtlich und optisch ansprechend sind. Bezüglich der Layout-Algorithmen lässt sich sagen, dass diese für alle mir gegeben Beispielgraphen einwandfrei funktionieren. Nichtsdestotrotz soll gesagt sein, dass ich das Problem des Layoutings nicht auf einen einfachen formalen Algorithmus herunterbrechen konnte und kann deshalb auch keine Garantie für die Korrektheit der Algorithmen im Allgemeinen abgeben. Für den Fall, dass das System irgendwann noch verändert/angepasst/erweitert werden sollte habe ich versucht, den Code möglichst klar zu dokumentieren.

4.2 Mögliche Folgearbeiten

Auch wenn das System im jetztigen Zustand gebrauchsbereit und soweit abgeschlossen ist, kann es durchaus in Teilbereichen noch verbessert oder erweitert werden:

- Bei der Suche nach optimalen Kantenverläufen mit minimer Anzahl Kantenkreuzungen können unter Umständen mehrere (bezüglich der Anzahl Kantenüberschneidungen) gleich optimale Lösungen gefunden werden. Im allgemeinen ist es sehr kompliziert zu entscheiden, welche dieser Lösungen die optisch ansprechendste ist, allerdings könnte die Auswahl der schlussendlich verwendeten Lösung zumindest parametrisiert werden. (beispielsweise könnten Kantenverläufe die schnell nach rechts/links ziehen bevorzugt werden etc.)

- Integration von kollisionsfreier Positionierung von Labels (beispielsweise entlang Kanten) in den Layout-Processor.[11] Dazu müsste eine Kollisionserkennung implementiert werden.

- Implementierung zusätzlicher Parameter für den Layout-Processor. Beispielsweise gibt es in der vorliegenden Implementierung einen Layout-Parameter, welcher festlegt, ob die Knoten entweder soweit wie möglich nach unten oder soweit wie möglich nach oben gezogen werden. Dieser Parameter lässt sich dahingehend erweitern, dass Knoten beispielsweise auch zentriert entlang von langen Kanten platziert werden oder versucht wird, die Knoten möglichst gleichmässig über die verschiedenen Ebenen zu verteilen. (was zu einer besseren Platzausnutzung führen kann (siehe 4.2), allerdings wäre damit dann auch die erste Randbedingung 3.4.1 nicht mehr gegeben)

[11] In der vorliegenden Implementierung werden Kantenlabel vom Kantenrenderer gesetzt. Allfällige überschneidungen mit anderen Grafikkomponenten müssen daher manuell erkannt werden, können jedoch dann mittels einer parametrisierten Festlegung des Positionierungsorts entlang der Kante mit minimalem Aufwand behoben werden.

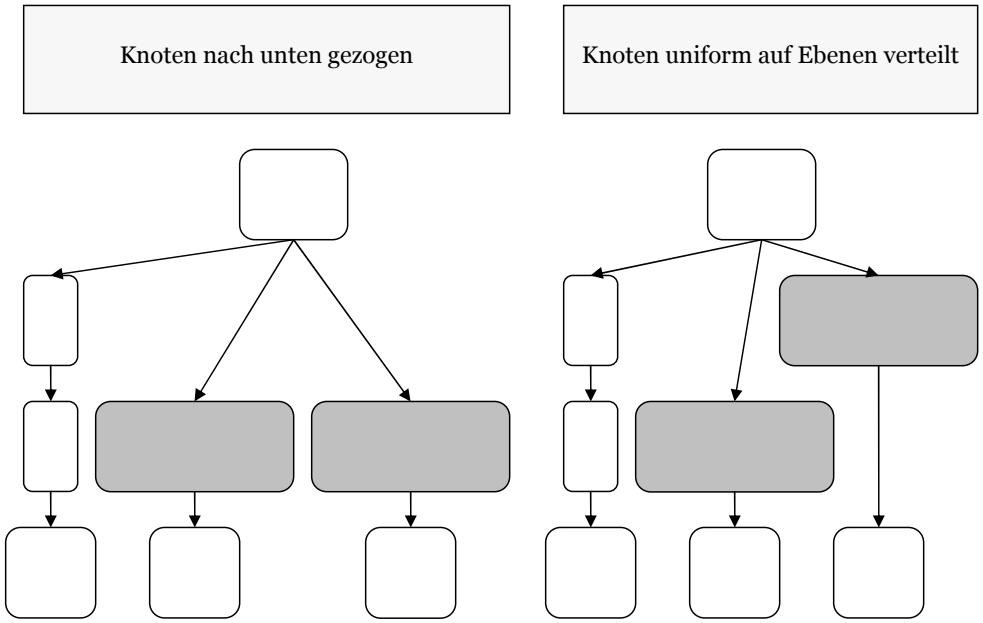


Illustration 4.1 - Knotenverschiebungsvarianten

4.3 Persönliches Fazit

Persönlich ziehe ich aus dieser Semesterarbeit ein sehr positives Fazit. Nicht zuletzt dank der äusserst engagierten Unterstützung durch Tobias Kaufmann habe ich einiges gelernt und sehr wertvolle Praxiserfahrungen sammeln können! Ausserdem habe ich mich im Verlaufe der Arbeit sehr stark mit dem Projekt identifizieren können und es war befriedigend zu sehen, dass viel mehr möglich ist, als ich mir das anfänglich gedacht hatte. Ebenfalls sehr positiv bewerte ich den Umstand, dass diese Semesterarbeit sehr vielschichtige Anforderungen stellte. (objekt-orientierte Designüberlegungen im Grossen einerseits und Detailarbeit mit Graphenalgorithmen andererseits)

Literaturhinweise

[1] Tobias Kaufmann: *Spezifikation der Dependenzgraphen*, Dokumentation, Institut TIK, ETH Zürich (2006)

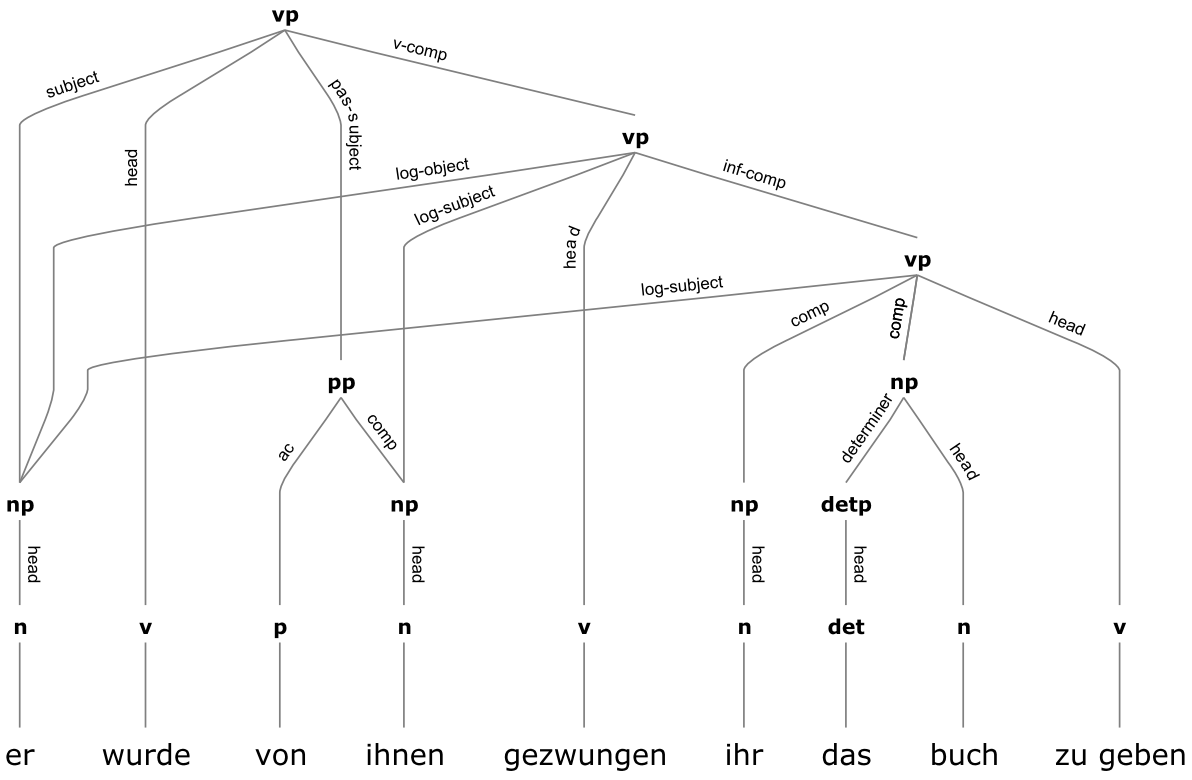
[2] P.Eades: *Edge crossings in drawings of bipartite graphs*, Algorithmica,10:379-403, (1994)

[3] E.R.Gansner, E.Koutsofios, S.C.North, K.Vo: *A Technique for Drawing Directed Graphs*, AT&T Bell Laboratories (1993)

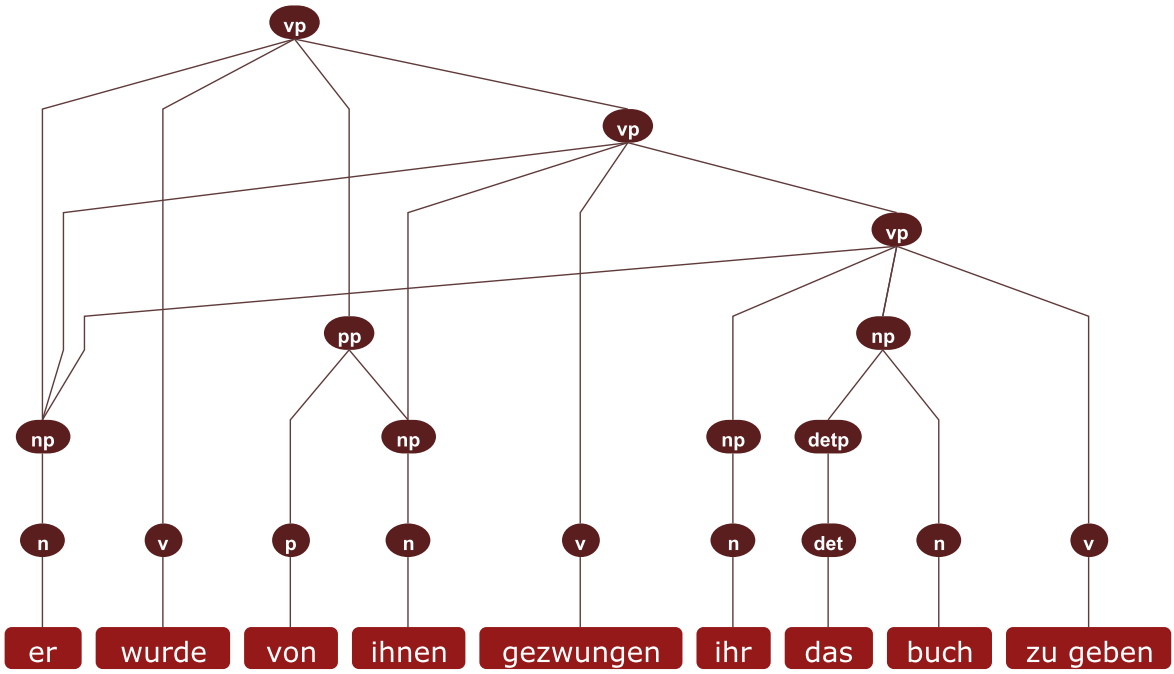
[4] Graphviz - Graph Visualization Software *http://www.graphviz.org/*

[5] Batik SVG-Library *http://xmlgraphics.apache.org/batik*

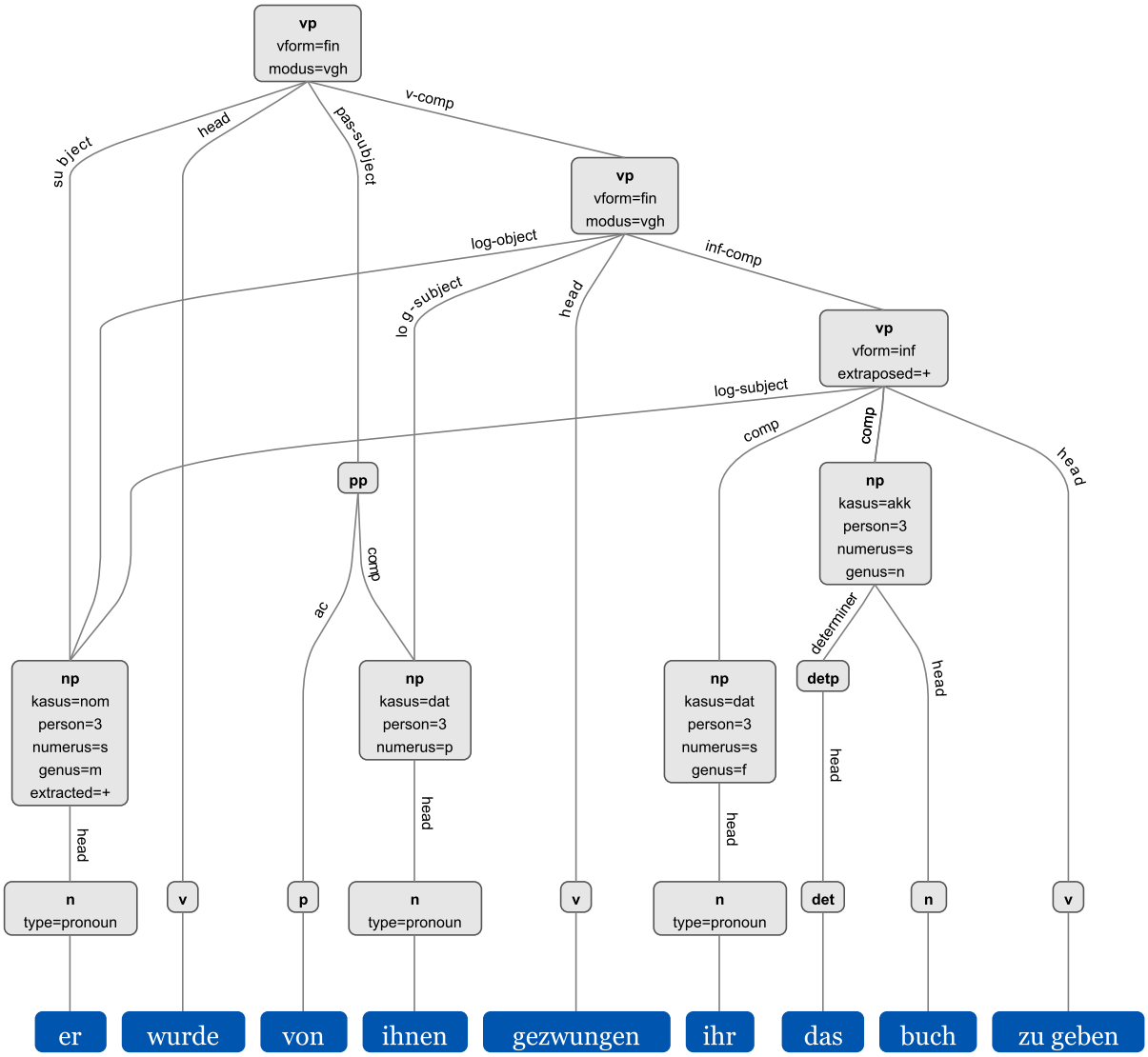
Anhang A - Beispiele visualisierter Graphen



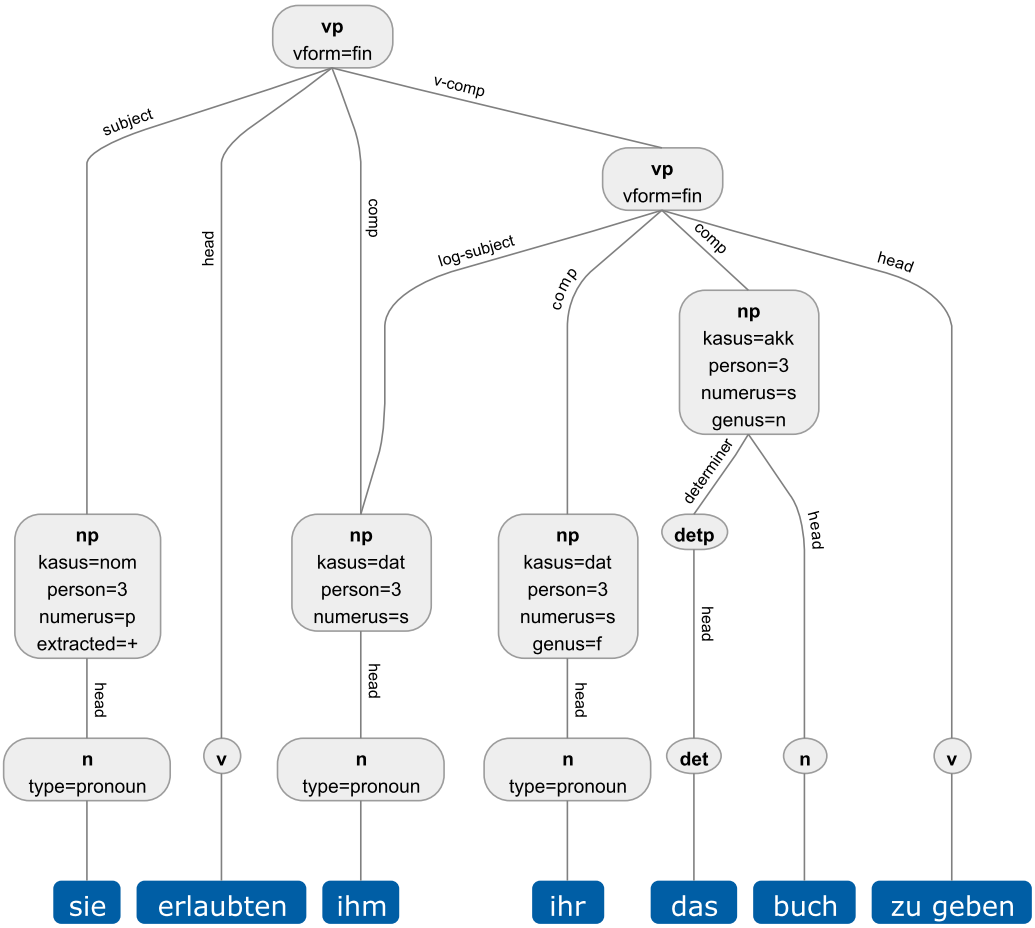
Beispiel 1a



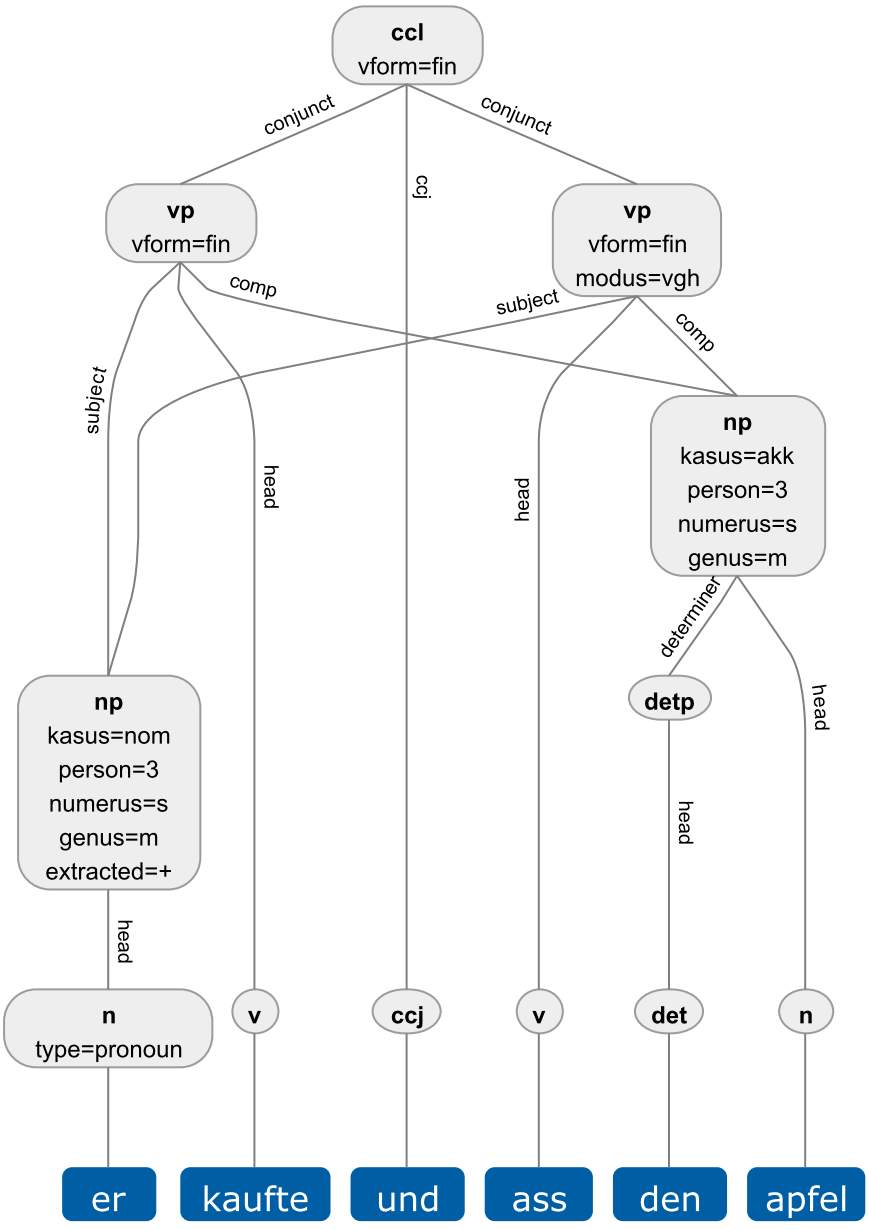
Beispiel 1b - Graph aus 1a mit anderem Knoten-Renderer



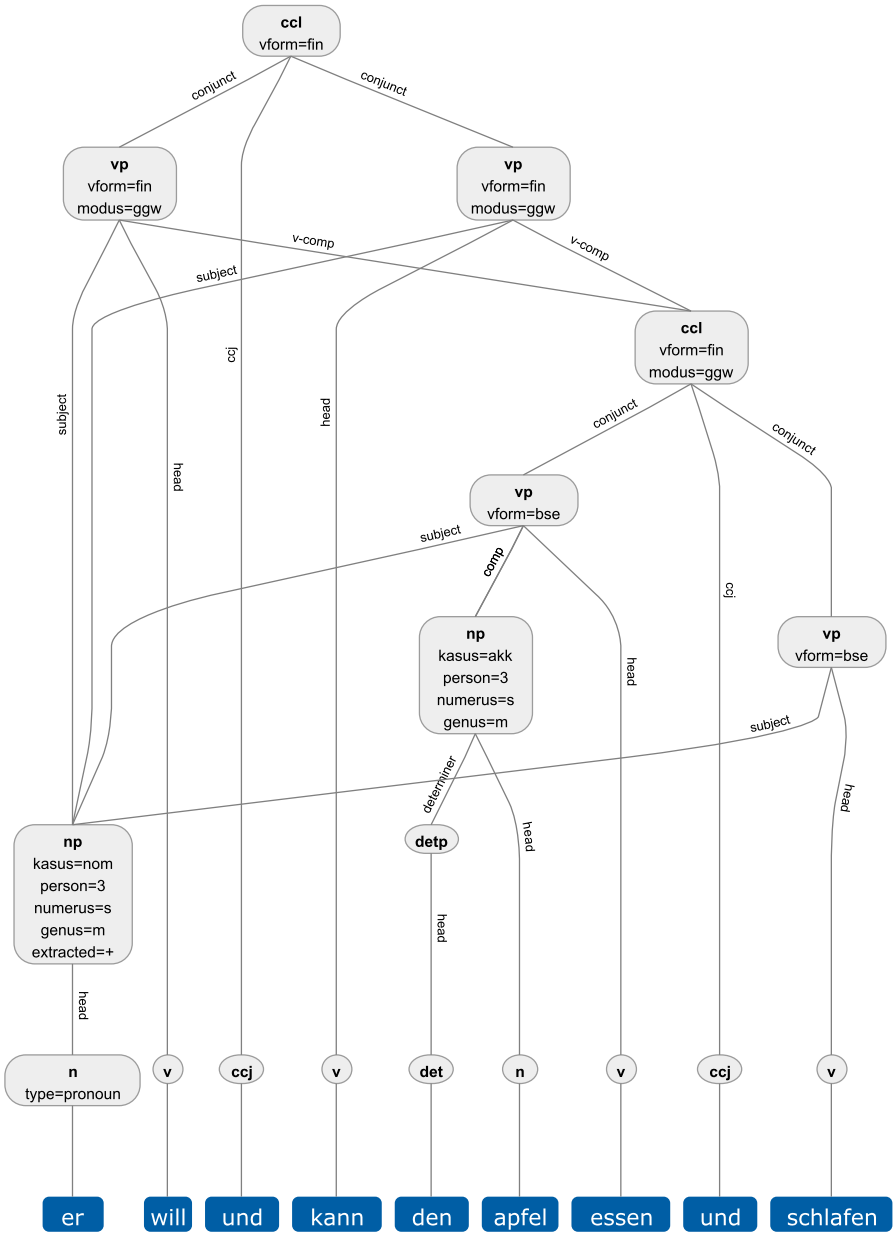
Beispiel 1c - wie 1b, nur mit anderen Render-Parameter


vp
vform=fin
subject
v-comp
head
comp
vp
vform=fin
log-subject
comp
comp
head
np
kasus=akk
person=3
numerus=s
genus=n
determiner
head
np
kasus=nom
person=3
numerus=p
extracted=+
np
kasus=dat
person=3
numerus=s
np
kasus=dat
person=3
numerus=s
genus=f
detp
head
head
head
head
n
type=pronoun
v
n
type=pronoun
n
type=pronoun
det
n
v
sie
erlaubten
ihm
ihr
das
buch
zu geben

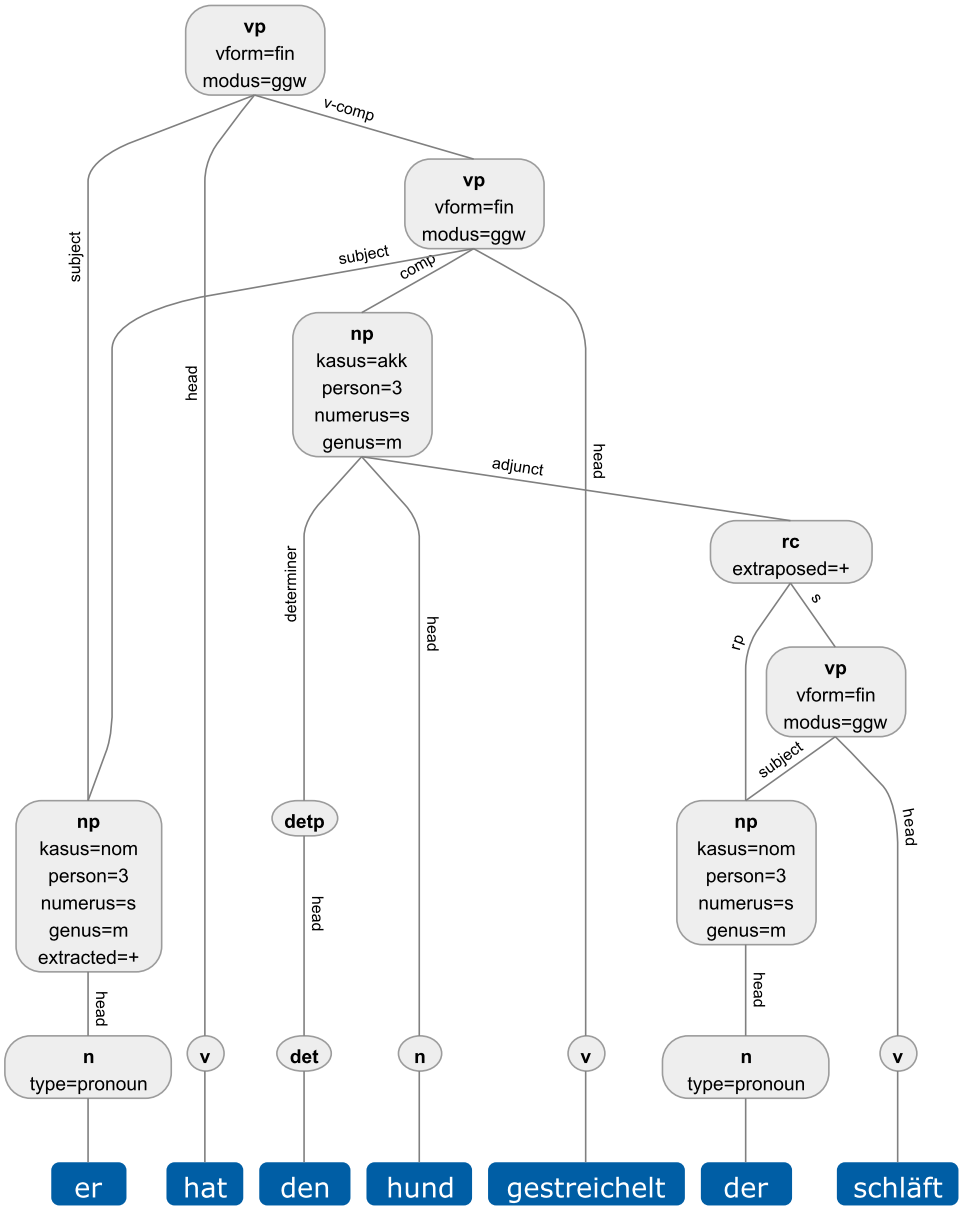
Beispiel 2a


ccl
vform=fin
conjunct
cj
conjunct
vp
vform=fin
vp
vform=fin
modus=vgh
comp
subject
comp
subject
head
head
np
kasus=akk
person=3
numerus=s
genus=m
determiner
np
kasus=nom
person=3
numerus=s
genus=m
extracted=+
detp
head
head
head
n
type=pronoun
v
ccj
v
det
n
er
kaufte
und
ass
den
apfel

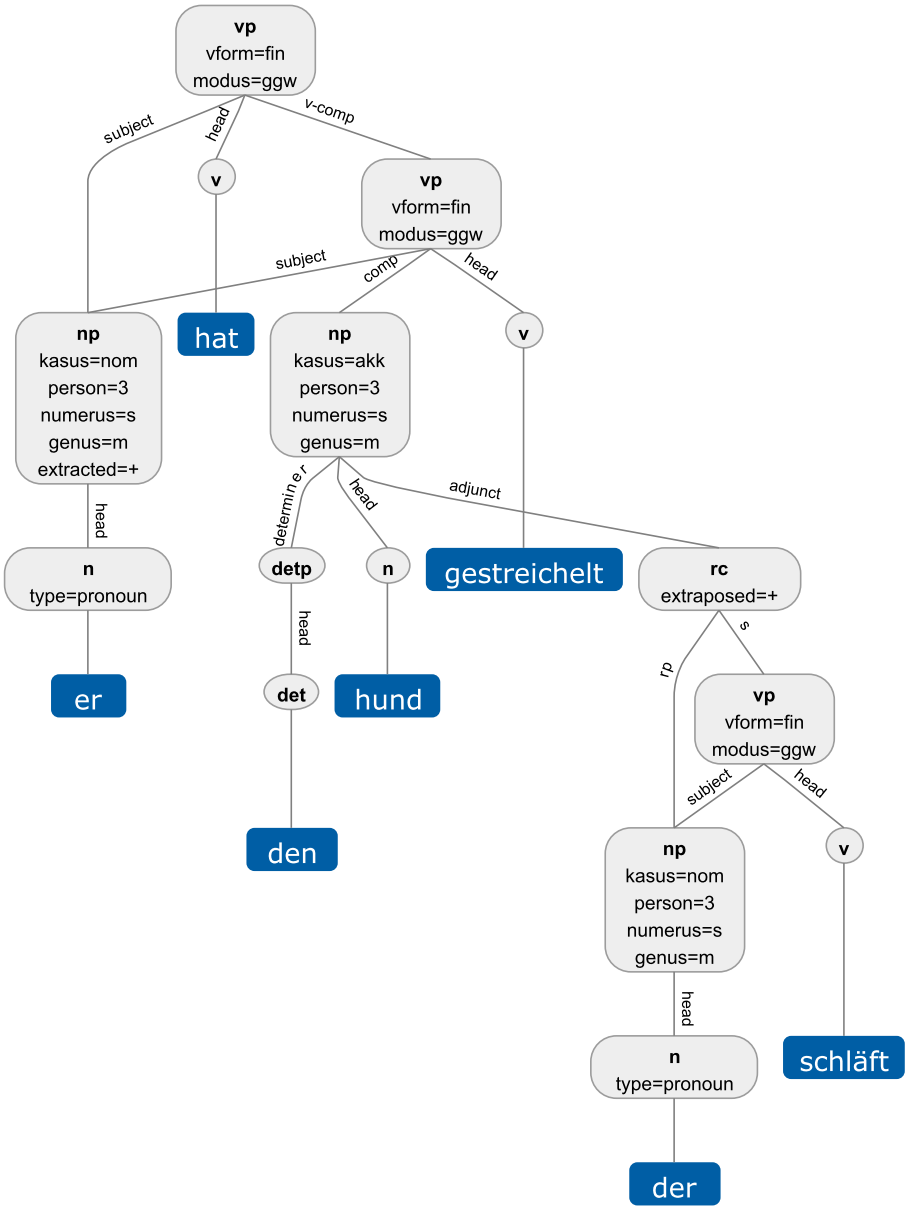
Beispiel 3a


ccl
vform=fin
conjunct
conjunct
vp
vform=fin
modus=ggw
vp
vform=fin
modus=ggw
v-comp
v-comp
subject
ccl
vform=fin
modus=ggw
subject
head
ccj
head
conjunct
conjunct
vp
vform=bse
subject
comp
ccj
np
kasus=akk
person=3
numerus=s
genus=m
vp
vform=bse
head
subject
determiner
head
head
detp
np
kasus=nom
person=3
numerus=s
genus=m
extracted=+
head
head
n
type=pronoun
v
ccj
v
det
n
v
ccj
v
er
will
und
kann
den
apfel
essen
und
schlafen

Beispiel 4a


vp
vform=fin
modus=ggw
v-comp
subject
head
vp
vform=fin
modus=ggw
subject
comp
np
kasus=akk
person=3
numerus=s
genus=m
adjunct
head
determiner
head
rc
extraposed=+
rp
s
vp
vform=fin
modus=ggw
subject
head
np
kasus=nom
person=3
numerus=s
genus=m
extracted=+
detp
head
np
kasus=nom
person=3
numerus=s
genus=m
head
n
type=pronoun
v
det
n
v
n
type=pronoun
v
er
hat
den
hund
gestreichelt
der
schläft

Beispiel 5a



Beispiel 5b - Gleicher Graph/Renderer wie 5a, andere Layout-Parameter

Anhang B - DTD Eingabeformat

```
<?xml version='1.0' encoding='utf-8'?>

<!ELEMENT dependencygraph
((parameter|datachunk|nonterminal|terminal|renderer)*) >

<!ELEMENT nonterminal   ( datachunk*, edge* ) >

<!ATTLIST nonterminal   id CDATA #REQUIRED
                        label CDATA #IMPLIED
                        rendererid CDATA #IMPLIED >

<!ELEMENT terminal      ( datachunk* ) >

<!ATTLIST terminal      id CDATA #REQUIRED
                        text CDATA #REQUIRED
                        position CDATA #REQUIRED
                        rendererid CDATA #IMPLIED >

<!ELEMENT edge          ( datachunk* ) >
<!ATTLIST edge          targetid CDATA #REQUIRED
                        head (true|false) "false"
                        label CDATA #IMPLIED
                        rendererid CDATA #IMPLIED >

<!ELEMENT renderer      ( parameter* ) >
<!ATTLIST renderer      id CDATA #REQUIRED
                        class CDATA #REQUIRED >

<!ELEMENT parameter     EMPTY >
<!ATTLIST parameter     name CDATA #REQUIRED
                        value CDATA #REQUIRED >

<!ELEMENT datachunk     (#PCDATA) >
<!ATTLIST datachunk     type CDATA #REQUIRED >
```

Anhang C - Input-XML Beispiel (gekürzt)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dependencygraph SYSTEM "dependencygraph.dtd">

<dependencygraph>
  <terminal id="id1" text="wurde" position="1" rendererid="terminalRenderer001" />
  <nonterminal id="id2" label="v" rendererid="nonterminalRenderer001">
    <edge targetid="id1" head="true" rendererid="edgeRenderer001" />
  </nonterminal>
  <terminal id="id3" text="gezwungen" position="4" rendererid="terminalRenderer001" />
  <nonterminal id="id4" label="v" rendererid="nonterminalRenderer001">
    <edge targetid="id3" head="true" rendererid="edgeRenderer001" />
  </nonterminal>
  <terminal id="id5" text="zu geben" position="8" rendererid="terminalRenderer001" />
  <nonterminal id="id6" label="v" rendererid="nonterminalRenderer001">
    <edge targetid="id5" head="true" rendererid="edgeRenderer001" />
  </nonterminal>
  <terminal id="id7" text="ihr" position="5" rendererid="terminalRenderer001" />

  .
  .
  .

  <parameter name="multipleParentsShiftMode" value="1" />
  <parameter name="emptySpaceBetweenNodes" value="10" />
  <parameter name="dummyNodeWidth" value="10" />
  <parameter name="nodeArrangeType" value="0" />
  <parameter name="deltaY" value="50" />
  <parameter name="marginTop" value="40" />
  <parameter name="marginLeft" value="20" />
  <parameter name="marginBottom" value="10" />
  <parameter name="marginRight" value="20" />
  <renderer id="terminalRenderer001"
   class="ch.ethz.dgv.processors.output.renderers.FancyNodeRenderer">
    <parameter name="backgroundcolor" value="#0555af" />
    <parameter name="fontfamily" value="georgia" />
    <parameter name="fontsize" value="17" />
    <parameter name="textcolor" value="white" />
    <parameter name="roundcornerradius" value="5" />
    <parameter name="stroke" value="false" />
    <parameter name="printfeatures" value="false" />
    <parameter name="printtext" value="true" />
    <parameter name="printlabel" value="false" />
  </renderer>
  <renderer id="edgeRenderer001"
   class="ch.ethz.dgv.processors.output.renderers.SmoothEdgeRenderer">
    <parameter name="color" value="grey" />
    <parameter name="labelcolor" value="black" />
    <parameter name="strokewidth" value="1" />
    <parameter name="smoothness" value="0.5" />
    <parameter name="labelatpercent" value="30" />
  </renderer>
</dependencygraph>
```

Anhang D - Übersicht Packages

ch.ethz.dgv.api
Enthält Klassen auf die von aussen zugegriffen werden soll. (Visualizer & Stylizer)

ch.ethz.dgv.graph
Enthält Klassen zur Modellierung des abstrakten Dependenzgraphen.

ch.ethz.dgv.processors.input
Enthält InputProcessor Klasse, zum einlesen der Daten von Eingabedatei.

ch.ethz.dgv.processors.input.annotation.data
ch.ethz.dgv.processors.input.annotation.style
Enthalten Klassen zur Annotation des abstrakten Graphen mit Daten (data chunks) beziehungsweise Styling Informationen.

ch.ethz.dgv.processors.layout
Enthält LayoutProcessor Klasse, zur Berechnung des Graphenlayouts.

ch.ethz.dgv.processors.layout.annotation
Enthält Klassen zur Annotation des abstrakten Graphen mit Layout-Informationen (Knoten und Kantenposition, Dokumentgrösse)

ch.ethz.dgv.processors.layout.datastructures
Enthält Klassen zur Modellierung der für die Layout-Algorithmen notwendigen Datenstrukturen.

ch.ethz.dgv.processors.output
Enthält OutputProcessor Klasse, zum erstellen des SVG-Dokuments.

ch.ethz.dgv.processors.output.renderers
Enthält NoderRenderer und EdgeRenderer Interfaces sowie bereits verfügbare Implementationen.

Anhang E - Aufgabenstellung des Instituts

Darstellung von Abhängigkeitsgraphen

SEMESTERARBEIT
für
Herr Sandro Blum

Wintersemester 2006
Betreuer: Tobias Kaufmann ETZ D97.7
Vertreter: René Beutler ETZ D97.7

Starttermin: 23. Oktober 2006
Abgabetermin: 2. Februar 2007

Einleitung

Die Gruppe für Sprachverarbeitung setzt sowohl in der Spracherkennung als auch in der Sprachsynthese Grammatiken ein, wobei unterschiedliche Grammatikformalismen zur Anwendung kommen. Ein wiederkehrendes Problem bei der Arbeit mit Grammatiken ist die Darstellung von Abhängigkeitsgraphen.

Ein Abhängigkeitsgraph ist ein gerichteter Graph, der genau einen Knoten ohne eingehende Kanten (die Wurzel) besitzt. Die Knoten repräsentieren Konstituenten (die sprachlichen Einheiten, aus denen Sätze aufgebaut sind), während die Kanten Abhängigkeitsbeziehungen zwischen Konstituenten darstellen. Auf der Menge der Blätter, die im Allgemeinen einzelnen Wörtern entsprechen, ist eine totale Ordnung definiert.

Je nach Grammatikformalismus unterliegen Abhängigkeitsgraphen weiteren Einschränkungen. Im einfachsten Fall handelt es sich um Baumstrukturen. Es kann jedoch auch vorkommen, dass ein Knoten mehrere eingehenden Kanten besitzt oder dass Zyklen auftreten.

Abhängigkeitsgraphen können sehr unterschiedlich dargestellt werden. Die Art und Weise der Darstellung kann vom jeweiligen Grammatikformalismus abhängen, aber auch davon, welche Aspekte hervorgehoben werden sollen. So kann etwa ein Knoten unterschiedlichste Informationen über die zugehörige Konstituente enthalten.

Trotz der Vielzahl von möglichen Visualisierungen sind die zugrundeliegenden Algorithmen, welche die Knoten und Kanten möglichst ansprechend anordnen, immer diesel-

ben. Es ist daher naheliegend, diese Layout-Algorithmen in einem Modul zu kapseln und mittels geeigneter Schnittstellen die Spezifikation von konkreten Darstellungsarten zu ermöglichen.

Aufgabenstellung

In dieser Semesterarbeit soll ein System entwickelt werden, welches die Darstellung von Abhängigkeitsgraphen ermöglicht. Dies beinhaltet die folgenden Teilaufgaben:

- die Spezifikation eines Eingabeformats zur Beschreibung von Abhängigkeitsgraphen
- die Spezifikation eines Ausgabeformats, das konkrete Visualisierungen beschreibt
- es soll eine Möglichkeit geschaffen werden, eine Ausgabedatei anzuzeigen, entweder durch Konversion in ein gebräuchliches Format oder mit Hilfe eines eigenen Viewer-Programms
- die Implementierung von geeigneten Layout-Algorithmen
- die Entwicklung einer Schnittstelle zwischen dem Layout-Modul und den darstellungsspezifischen Komponenten
- die Implementierung von zwei Darstellungsarten, um die Erweiterbarkeit des Systems zu demonstrieren

Die durchgeführten Arbeiten und die erhaltenen Resultate sind in einem Bericht zu dokumentieren, der in gedruckter Form (gebunden) und als PDF abzugeben ist. Zusätzlich sind im Rahmen eines Kolloquiums zwei Präsentationen vorgesehen: etwa drei Wochen nach Beginn soll der Arbeitsplan und am Ende der Arbeit die Resultate vorgestellt werden. Die Termine werden später bekannt gegeben.