Anhang A

Automatische Erzeugung der Graphen

Die Idee zu den Graphen am Kapitelanfang habe ich aus einer Vorlesung, die Juan Diego Caycedo in Vertretung von Prof. Ziegler gehalten hatte. Dort wollten wir einen schwierigen Satz beweisen und um die vielen Lemmata etwas zu ordnen, zeichnete Juan Diego eine Art Roadmap an die Tafel, an der man sich immer orientieren konnte, wie das aktuelle Lemma ins Gesamtbild paßt. Mir hat das so gut gefallen, daß ich etwas Ähnliches für meine Diplomarbeit haben wollte.

Das Python-Programm, das aus diesem Wunsch entstanden ist, will ich hier nun etwas dokumentieren. Es ist noch weit davon entfernt, als fertiges LaTeX-Paket auf die Welt losgelassen zu werden, aber nach der Diplomarbeit werde ich das Projekt weiter verfolgen, sofern mir die Zeit dazu reicht.

Ursprünglich wollte ich mir einfach einen Überblick über die Zusammenhänge aller Sätze verschaffen, um strukturelle Fehler oder Unschönheiten schneller zu entdecken. Als mir bewußt wurde, daß diese Diagramme nicht nur für mich hilfreich sind, entschloß ich mich, diese fest in die Diplomarbeit zu integrieren.

Komponenten und Programmablauf

Natürlich habe ich das Rad nicht komplett neu erfunden, so verwende ich zur Erzeugung der Grafiken ein Tool namens GraphViz, das aus abstrakten, textuellen Beschreibungen fertige Graphen erstellt. GraphViz versucht, ein möglichst optimales Layout für den Graphen zu erzeugen, was meist recht gut funktioniert. Meine Aufgabe war also, die Graphstruktur aus dem LaTeX-Dokument zu extrahieren und in sogenannte .dot-Dateien zu schreiben, die von GraphViz verarbeitet werden können.

Die Arbeitsweise ist wie folgt:

- Die Zeilen der Latex-Datei einlesen
- Include-Anweisungen auflösen
- Sätze, Lemmata, etc. und eventuelle Zusatzinformationen als Knoten extrahieren
- *ref-Anweisungen als Kanten extrahieren
- Transitiv implizierte Kanten entfernen (der Graph wird so etwas ausgedünnt, ohne logische Fehler zu produzieren)
- .dot-Datei schreiben
- GraphViz ausführen, um die Graphen als PDFs zu rendern
- Graphen ins LaTeX-Dokument einbinden

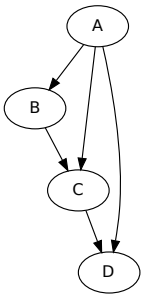
Zum Zweck der Speicherung aller für den Graph relevanten Informationen innerhalb

der .tex-Datei habe ich zwei neue LaTeX-Commands definiert, welche im PDF keine Ausgabe verursachen, sondern nur bei der Graph-Generierung ausgewertet werden. Dies sind:

\graphcaption{*txt*}	Wird in Sätzen/Lemmata verwendet, um die Beschriftung des zugehörigen Graph-Knotens zu definieren.
\graphref{*lbl*}	Definiert eine zusätzliche Kante im Graphen. Zur Verwendung bei Referenzen, die nicht explizit per \ref{} zitiert werden, sondern aus dem Kontext klar werden.

Unvorhergesehene Probleme

Ein großes Problem lag in der Art und Weise, wie GraphViz beim Aufbau eines Graphen vorgeht: die Knoten werden in horizontalen Ebenen (Ranks) organisiert, die Kanten verlaufen ausschließlich *zwischen* den Ebenen nach unten. Gibt es sehr viele Knoten und wenig Kanten, so sieht GraphViz keine Notwendigkeit, neue Ebenen anzulegen und der Graph wächst extrem in die Breite.
Besonders in Kapitel 1 dieser Arbeit hat mir das Probleme gemacht, da die Abhängigkeiten der Sätze dort weniger linear sind als in den anderen Kapiteln. Um dem Problem entgegenzutreten, ist leider etwas menschliche Intervention nötig; die Fähigkeiten von GraphViz reichen nicht aus, die Knoten vernünftig umzuorganisieren. Zu diesem Zweck habe ich ein zusätzliches Kommando \invref{*lbl*} definiert; es fügt eine unsichtbare Kante in den Graphen ein und erzwingt damit die Einführung neuer Ebenen, d.h. der Graph wird höher und weniger breit. Meist reichen bereits zwei geschickt gesetzte unsichtbare Kanten, um den Graphen entsprechend umzuformen.



Ein weiteres Problem, an das ich zuerst überhaupt nicht dachte, war die große Menge der redundanten Kanten, die den Graphen manchmal recht unübersichtlich machten. Die Grafik rechts zeigt dieses Problem sehr deutlich: die Kanten $A \to C$ und $A \to D$ sind für die logischen Abhängigkeiten nicht relevant, da sie sich aus anderen Kanten kombinieren lassen (sie werden transitiv impliziert). Mathematische Dokumente produzieren leider eine hohe Anzahl solcher Kanten, da bestimmte Sätze und Lemmata immer wieder Anwendung finden. Zur Bekämpfung dieses Problems wende ich folgenden Algorithmus an: entferne eine Kante $X \to Y$, falls X einen Kindknoten besitzt, der Vorfahre von Y ist. In der Beispielgrafik trifft dies auf $A \to C$ und $A \to D$ zu, wie man leicht nachprüft. Notwendig zur Anwendung des Algorithmus war natürlich die Berechnung aller Vorfahren eines gegebenen Knotens. Hierfür darf der Graph keine Zirkel enthalten, denn sonst ist der Begriff des Vorfahren nicht definierbar. Dies sollte bei mathematischen Dokumenten aber immer der Fall sein, denn sonst hat der Autor sowieso einen logischen Fehler gemacht.

Details zur Implementation

Konfiguration

Zur Generierung der Graphen müssen einige Daten bekannt sein, welche vorab im Quellcode definiert sein müssen. Dies sind:

Variable	**Verwendung**
numbered	Liste aller numerierten TeX-Umgebungen
graphable	Liste aller Umgebungen, die in den Graphen gezeichnet werden
proofless	Liste aller Umgebungen, die keine Beweise erfordern
refs	Liste aller *ref-Befehle, die als Kanten in den Graph eingehen sollen
translations	Übersetzungen für die Namen der Umgebungen
colors	Farben für die Kapitel

Einlesen der Dateien

Das Python-Skript nimmt als Argument eine TeX-Datei, deren Inhalt in eine große String-Variable eingelesen wird. Als nächstes werden alle \include{}-Befehle durch den Inhalt der entsprechenden Dateien ersetzt. Zur besseren Verarbeitung werden die Daten dann tokenisiert, d.h. nur die relevanten Befehle (Tokens) extrahiert. Nicht relevante Daten wie Freitext, Kommentare, Fußnoten, Indexeinträge, usw. werden entfernt und stören in den folgenden Schritten nicht weiter oder verkomplizieren die Algorithmen unnötig.

Momentan wird nur eine Ebene von \includes aufgelöst. Durch einen rekursiven Algorithmus ließe sich diese Beschränkung aber leicht aufheben.

Extraktion der Knoten und Kanten

Nun wird über die Liste der Token iteriert. Je nach Token wird eine der folgenden Aktionen ausgeführt:

Token	**Aktion**
\chapter	Kapitelzähler um eins erhöhen, Theoremzähler auf 0 setzen.
\begin{*env*}	ist *env* ∈ *numbered*, Theoremzähler um eins erhöhen. ist *env* ∈ *graphable*, neuen Knoten anlegen. Der Knoten wird mit folgenden Daten initialisiert: • einer ID der Form theorem.2.13 • einem Label (zunächst automatisch generiert, wird später evtl. durch ein TeX-Label ersetzt) • Numerierung (aus Kapitel-/Theoremzähler ermittelt) • Farbe (aus *colors* ermittelt) • Beschriftung, zunächst automatisch erzeugt (mit *translations*) • Einem *proven*-Flag, das anzeigt, ob das Theorem bereits beweisen wurde; true, falls *env* ∈ *proofless*, sonst false.
\label{*lbl*}	Setze das Label des zuletzt erzeugten Knoten.
\graphcaption{*txt*}	Setze die Beschriftung des zuletzt erzeugten Knoten.
\end{*env*}	ist *env* = *proof*, setze das *proven*-Flag des des zuletzt erzeugten Knoten auf false.
Kommando aus *refs*	Vermerke die Referenz beim zuletzt erzeugten Knoten, sofern *proven* = false.

Das *proven*-Flag wird benötigt, da LaTeX-Referenzen im Freitext nach Abschluß eines Beweis sonst weitere Kanten verursachen würden. Ein Problem dieses Ansatzes ist jedoch, daß Beweise immer direkt nach dem jeweiligen Theorem stehen müssen. Aufgeschobene Beweise sind momentan nicht möglich.

Aufbereitung der Daten

Da die Rohdaten durchaus noch Fehler enthalten können (falsch getippte Referenzen, etc.) und in dieser Form recht unpraktisch für die weitere Verarbeitung sind, werden sie noch entsprechend aufbereitet. Es werden folgende Schritte ausgeführt:

- Zum effizienteren Zugriff in den folgenden Algorithmen werden die Knoten nach Label und ID indiziert.
- Referenzen auf nicht existente Knoten werden wieder entfernt.
- Aus der von LaTeX generierten .thm-Datei werden die Seitenzahlen aller Theoreme ermittelt. Die ID in der .thm-Datei entspricht dabei der ID des Knotens.
- Zu jedem Knoten werden die Kindknoten berechnet, d.h. die Theoreme, die direkt aus einem gegebenen folgen.
- Zu jedem Knoten wird die Menge der Vorfahren berechnet.
- Transitiv implizierte Kanten werden entfernt (wie zuvor erklärt).

Schreiben der .dot-Dateien und Rendern

Alle Informationen sind gesammelt und der Graph kann nun gezeichnet werden. Hierzu wird für jedes Kapitel eine eigene .dot-Datei geschrieben und anschließend gerendert. Der Aufbau einer solchen Datei ist recht einfach. Anfangs werden einige allgemeine Eigenschaften definiert (Abstände, Formen, etc), dann folgt die Liste der Knoten und schließlich die Liste der Kanten. Die Knoten, die zu einem Kapitel gerendert werden müssen, sind die Knoten des Kapitels selbst und deren direkte Vorgänger und Nachfolger (aus anderen Kapiteln). Somit sind die Verbindungen zwischen den Kapiteln schnell ersichtlich.
Die .dot-Dateien werden nun als PDFs gerendert, GraphViz kann das out-of-the-box.

Einbindung ins .tex-Dokument

Die Einbindung in das .tex-Dokument erfolgt mit einem schlichten \includegraphics. Da die Grafiken im Kapitelanfang möglichst automatisch eingebunden werden sollen, habe ich das \chapter-Kommando modifiziert. Es prüft, ob eine entsprechende Grafik vorhanden ist, bindet sie mit dem textpos-Paket seitenfüllend ein und beginnt das eigentliche Kapitel dann auf der Folgeseite.

Der Quellcode

Die folgenden Seiten zeigen den Python-Quellcode zum Zeitpunkt der Abgabe der Arbeit. Ich bitte um etwas Nachsicht - die Algorithmen sind noch nicht besonders optimiert und das Programm stellt auch (implizit) einige Bedingungen an das TeX-Dokument, was zu unerwarteten Ergebnissen führen kann, wenn diese nicht erfüllt sind (z.B. bei Trennung von Sätzen und Beweisen).