Studienarbeitsthemen 2016/2017

Stephan Schulz

stephan.schulz@dhbw-stuttgart.de
http://www.dhbw-stuttgart.de/~sschulz

Inhaltsverzeichnis

1 Einführung

Deduktionssysteme bilden logische Denkprozesse nach, um so aus bekannten Fakten gesicherte neue Erkenntnisse abzuleiten. Automatische Deduktion ist ein Teilgebiet der künstlichen Intelligenz, aber auch der theoretischen Informatik. Wir entwickeln an der DHBW Stuttgart den Theorembeweiser E [10, 7], einen der derzeit leistungsstärksten Beweiser für die Prädikatenlogik erster Stufe, und den stärksten dieser Beweiser, der unter einer Open-Source/Free-Software-Lizenz steht. E nimmt regelmäßig an der CADE ATP System Competition teil, und ist in seiner Klasse in den letzten Jahren nur vom Beweiser Vampire der Universität Manchester geschlagen worden.

Automatische Theorembeweiser sind Computerprogramme, die nachweisen, dass bestimmte Aussagen zwingend aus einer gegebenen formalen Beschreibung folgen. Ein wichtiges Anwendungsbeispiel ist die Verifikation von Software, z.B. der Nachweis, dass ein Fahrerassistenzsystem einen PKW niemals beschleunigt, wenn der Fahrer die Bremse tritt, oder dass eine Banktransaktion immer entweder vollständig oder gar nicht abgewickelt wird. Ein anderes Anwendungsbeispiel

ist die Beantwortung von komplexen Fragen über großen formalen Wissensbasen, etwas die Frage "Welche europäischen Großstädte sind von Überflutungen bedroht?" Um diese Fragen zu beantworten, muss ein Beweiser im Raum der möglichen logischen Ableitungen nach einem (oder auch mehreren) Beweisen für die Behauptung bzw. Vermutung suchen.

E ist ein *saturierender* Theorembeweiser für die Prädikatenlogik erster Stufe. Ein solcher Beweiser stellt den Suchzustand als Menge von als wahr angenommenen logischen Formeln (*Klauseln*) dar, und kombiniert nach festen Regeln Formeln, die Aussagen über die selben Objekte machen, um so neues Wissen explizit herzuleiten. Dieser Prozess wird ausgeführt, bis der Beweis offensichtlich wird, oder bis der Beweisversuch aufgegeben wird. In der Praxis sucht das System nach einem *Beweis per Widerspruch*, bei dem dann der explizite Widerspruch zwischen den Axiomen und der negierten Vermutung hergeleitet wird.

Fast immer ist die Menge der ableitbaren Formeln unendlich, und die Größe der Wissensbasis wächst sehr schnell. Das erschwert die Beweissuche stark. Um diesen Effekt zu reduzieren haben moderne Kalküle ein Redundanzkonzept, mit der Formeln entweder vereinfacht oder sogar komplett gestrichen werden können. Moderne Beweiser verbringen einen großen Teil der Beweissuche mit solchen *Simplifikationen*.

Um einen Beweiser möglichst effizient zu machen gibt es zwei grundsätzliche Ansätze. Zum einen kann man die Geschwindigkeit steigern, mit der neue Formeln generiert und vereinfacht werden. Dadurch wird ein größerer Teil des Suchraums bearbeitet, was die Chance erhöht, einen Beweis zu finden. Zum anderen kann man versuchen, aus der Menge aller ableitbaren Formeln die für einen Beweisversuch besonders interessanten Formeln erkennen und so die Suche auf viel versprechende Teile des Suchraums konzentrieren. So kann man gezielt tiefer in den Suchraums vordringen und potentiell auch längere Beweise finden.

Neben diesen Kernfragen gibt es auch interessante Aufgabenstellung in der Vorverarbeitung der Beweisprobleme, der Axiomenauswahl, der Bearbeitung von Hintergrundtheorien und der Beweisdarstellung.

1.1 Rahmenbedingungen

Für alle angebotenen Arbeiten gilt:

- Implementierungssprache im Beweiser selbst ist C. Für Programmieraufgaben im Umfeld kommen auch andere Sprachen mit Open-Source-Implementierungen in Frage. Viele Teilprojekte verwenden Python.
- E wird primär für UNIX-artige Betriebssysteme entwickelt, insbesondere Linux und OS-X. Das Versionskontrollsystem ist `git`.
- Wenn Code in den Beweiser integriert wird, so wird er damit implizit wie das System selbst lizenziert (im Moment GPL 2 und LGLP 2). Dass heißt auch, dass nur Bibliotheken zum Einsatz kommen können, die ebenfalls unter geeigneten Open-Source-Lizenzen stehen. Wir bemühen uns im allgemeinen, wenig externe Abhängigkeiten in das Kernsystem einzubringen.

- Es wird angestrebt, Ergebnisse der Arbeiten auf Workshops, Konferenzen, oder in Journal-Artikeln zu veröffentlichen. Der Erfolg dieser Bemühungen hängt von der Qualität der Ergebnisse und von der Hartnäckigkeit der Autoren ab.

2 Themen

2.1 *Step by Step* - Detaillierte Beweise für E

Für viele Anwendungen von Theorembeweisern werden explizite Beweisobjekte gefordert. Es reicht also nicht, wenn der Beweiser intern ein Theorem nachweisen kann, der Nutzer erwartet auch eine nachvollziehbare Begründung, *warum* das Theorem gültig ist. Solche Beweise können mit unabhängigen Werkzeugen oder manuell verifiziert werden, und die Struktur des Beweises und interessante Zwischenergebnisse geben einen Einblick in die Anwendungsdomäne und das Suchverhalten.

E kann interne Beweisobjekte mit minimalem Aufwand erzeugen und diese als Folge von begründeten Schritten ausgeben. Allerdings sind die einzelnen Beweisschritte dieses Beweisobjektes in der Regel weder eindeutig noch atomar. Sie fassen typischerweise eine generierende Inferenz und alle darauf folgenden Vereinfachungen zusammen. Damit sind sie für manche Anwendungen nicht optimal geeignet.

Im Rahmen dieser Studienarbeit soll der vorhandene Mechanismus für die Erzeugung von Beweisobjekten verfeinert werden. Dabei stehen folgende Teilaufgaben an:

1. Genauere Beschreibung der Ableitungen im Beweisobjekt, insbesondere Buchführung darüber, an welchem Positionen in Formeln und Termen Inferenzregeln angewendet werden

2. Erstellung einer Bibliothek und eines Programms, die die Beweise in Einzelschritte zerlegen und alle Zwischenergebnisse explizit macht

3. Update eines Beweisprüfers, der einzelne Beweisschritte mit anderen Theorembeweisern nachvollzieht

Die Arbeit eignet sich für zwei Studenten. Wenn Sie von einem einzelnen Studenten bearbeitet wird, sollten mindesten Punkt 1 und die Bibliothek von Punkt 2 erstellt werden.

Literatur

[10, 7, 6]

Team

1–2 Studenten

2.2 *Can you take a hint?* Verbesserte Integration von Hinweisen und Lemmas für die Beweissuche von E

Der lokale Suchraum, also die Anzahl der ableitbaren Formeln, wächst in der Regel exponentiell mit der Länge der betrachteten Ableitung. Deswegen werden insbesondere lange und komplexe Beweise nur selten direkt gefunden. Ein erfolgreicher Ansatz, trotzdem komplexe mathematische Theoreme zu beweisen, ist es, dem Beweiser bestimmte Zwischenergebnisse, so genannte *hints*, vorzugeben. Dies können entweder von Nutzer als wahr vermutete Lemmata sein, oder auch automatisch aus anderen, einfacheren Beweisen extrahiert werden. Dieser Mechanismus wurde ursprünglich für den Beweiser Otter [5] und seinen Nachfolger Prover9 [4] entwickelt, und wurde z.B. für den berühmten Beweis der Robbins-Vermutung [4] verwendet, kommt aber auch in der aktuellen mathematischen Forschung zum Einsatz [3].

E verwendet eine vergleichsweise einfache Implementierung von Hints. Dabei wird dem Beweiser eine Liste mit Zwischenergebnissen vorgegeben, und jede neu hergeleitete Formel wird mit dieser Liste verglichen. Klauseln, die Hint-Klauseln subsumieren, werden bevorzugt für die weitere Herleitung von neuen Fakten verwendet. Im Rahmen dieser Arbeit soll dieser Mechanismus verbessert werden. Insbesondere fallen dabei folgende Aufgaben an:

1. Passende Hints werden über eine *Index* (im Fall von E über einen *Feature Vector Index)* gefunden. Dieser Index ist für den häufigen Fall der Unit-Klauseln (also Klauseln mit nur einem einzelnen Literal) nicht optimal und sollte um einen *Discrimination Tree Index* für diesen Fall ergänzt werden.

2. Statt auf strikter Subsumption zu bestehen, können auch andere (schwächere oder stärkere) Ähnlichkeitsrelation verwendet werden. Speziell sollte mindestens eine Version implementiert werden, bei der konstante Funktionssymbole nicht unterschieden werden (d.h. eine Klausel $p(a, X)$ könnte einen Hint $p(b, X)$ "weich subsumieren".

3. Hints können einmal oder mehrmals verwendet werden.

Neben der Implementierung sollen die verschiedenen Optionen experimentell erprobt werden.

Literatur

[11, 3, 10, 7]

Team

1 Student

2.3 Folded and Compressed Feature Vector Indexing

Eine der wichtigsten Techniken zur Vermeidung von Redundanz im Theorembeweisen ist die (syntaktische) Subsumption. Dabei kann eine spezielle Klausel (die also schwächere Aussagen über kleinere Mengen von Objekten macht) ignoriert oder gelöscht werden, wenn eine allgemeinere Klausel gefunden wird, die mindestens die selben Aussagen umfasst. So wird z. B. die Klausel $p(f(X), a) \vee q(X, b)$ von der Klausel $p(Y, Z)$ subsumiert.

Schon ein einzelner Subsumptionsversuch ist relativ aufwändig, da viele Kombinationen durchgespielt werden müssen. Während der Beweissuche müssen alle neuen Klauseln gegen die bereits verarbeiteten Klauseln auf Subsumption getestet werden - und zwar in beide Richtungen. Um diesen Prozess zu beschleunigen, verwendet E *Feature Vector Indexing*. Der Index nutzt die Eigenschaft, dass eine allgemeinere Klausel nicht mehr Funktionssymbole enthalten kann, als eine speziellere Klausel. Diese Eigenschaft gilt auch pro Symbol, und kann weiter verschärft werden. So können pro Funktionssymbol mehrere numerische Features berechnet werden, von denen eine potentiell subsumierende Klausel in keinem Wert größer als die subsumierte Klausel sein darf. Durch geschickte Organisation der Vektoren in einem Trie können die meisten erfolglosen Subsumptionsversuche schon frühzeitig und billig vermieden werden.

Allerdings steigen die Kosten für Feature Vector Indexing mit der Länge der Vektoren, also mit der Anzahl der Symbole in der Spezifikation. Bei sehr großen Spezifikationen sind die meisten Feature-Werte 0 (nur wenige Symbole kommen in einer einzelnen Klausel vor), so dass der Informationsgehalt pro Feature gering ist. Dieses Problem soll in dieser Studienarbeit auf zwei verschiedene Arten gelöst werden. Die beiden Ansätze sollen verglichen werden.

1. *Folded Feature Vector Indexing* macht sich zu Nutzen, dass die Summe zweier Features wieder ein gültiges Feature sind. So kann ein langer Feature-Vektor virtuell mehrfach "gefaltet" werden. Dabei zusammenfallende Feature-Werte werden addiert. So wird die Informationsdichte erhöht. Allerdings geht durch die Addition auch Information verloren. Eine einfache Version von Folded Feature Vector Indexing ist bereits implementiert. Im Rahmen dieser Arbeit soll dieses Verfahren verallgemeinert werden und es soll mit verschiedenen Parametrierungen evaluiert werden.

2. *Compressed Feature Vector Indexing* ist ein neuer Ansatz, bei dem nur Features mit Werten größer 0 explizit dargestellt werden. So werden nur aussagekräftige Features verwendet. Allerdings wird die Struktur von Feature-Vektoren und speziell von Feature Vector Tries komplexer.

Literatur

[10, 7, 9]

Team

1 Student

2.4 *Deep Reasoning* - Hardware-beschleunigte Künstliche Intelligenz

In diesem Projekt geht es um das maschinelle Lernen von Wissen zur Steuerung der Beweissuche aus erfolgreichen Beweisen. Im Theorembeweisen stehen auf der einen Seite die syntaktischer Manipulation von symbolischen Formeln nach strikten Regeln, auf der anderen Seite die unscharfe Intuition über die Struktur des Beweises und viel versprechende Zwischenergebnisse. Eine ähnliche Situation findet sich bei vielen Spielen, wie z.B. Schach oder Go, wo die strikten Spielregeln den Raum der möglichen Züge definieren, aber eine heuristisch Bewertungsfunktion die Qualität der verschiedenen Stellungen bewertet. In diesem Bereich hat das Programm AlphaGo der Firma *Google DeepMind* Anfang 2016 einen Durchbruch erzielt, als es den weltbesten Profispieler 4 zu 1 besiegen konnte. Wir hoffen auf einen ähnlichen Erfolg im Theorembeweisen.

Zur Entwicklung dieser Suchheuristiken sollen mit Hilfe von künstlichen neuronalen Netzwerken große Datenmengen analysiert werden. NVIDIA fördert diesen Ansatz, indem sie dem Projekt im Rahmen eines Academic Hardware Grants einen Rechenbeschleuniger des Typs GeForce GTX TITAN X zur Verfügung stellt. Die GeForce GTX TITAN X besitzt 3072 Rechenkerne und 12 Gigabyte RAM mit einer Speicherbandbreite von 336,5 Gigabyte pro Sekunde. Die maximale theoretisch erreichbare Rechenleistung der Karte beträgt 6,144 TeraFLOPS. Zur Nutzung der Hardware schlagen wir Googles *TensorFlow* [1] vor, das den Übergang von einem normalen PC oder Laptop auf die GPU einfach macht.

Der Beweiser E kann bereits jetzt aus erfolgreichen Beweisen Beispiele von guten (zum Beweis beitragenden) und schlechten (für den Beweis unnötigen) Formeln bzw. Suchentscheidungen generieren. In diesem Projekt soll untersucht werden, wie man dieses Formeln geeignet für Lernen mit einem tiefen neuronales Netz konfiguriert, wie ein neuronales Netz die Klassifikation lernen kann, und wie man das trainierte Netz gut zum Steuern der Beweissuche einsetzen kann.

Das Thema ist auch für die Betreuer z.T. Neuland. Deswegen sollten sich mindestens zwei, gerne auch drei Studenten finden, die das Thema als Team bearbeiten.

Literatur

[8, 2, 10, 7, 1]

Team

2–3 Studenten

Literatur

[1] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Chen Zhifeng, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu

Devin Sanjay Ghemawat, Andrew Harp Ian Goodfellow, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Manjunath Kudlur Lukasz Kaiser, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Paul Tucker Kunal Talwar, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, , and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] C. Goller. Learning Search-Control Heuristics for Automated Deduction Systems with Folding Architecture Networks. In *Proc. of the European Symposium on Artificial Neural Networs (ESANN 99)*, 1999.

[3] Michael Kinyon, Robert Veroff, and Petr Vojtěchovskỳ. Loops with abelian inner mapping groups: An application of automated deduction. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788 of *LNAI*, pages 151–164. Springer, 2013.

[4] William W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010. (acccessed 2016-03-29).

[5] W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE 13 ATP System Competition.

[6] S. Schulz. Analyse und Transformation von Gleichheitsbeweisen. Projektarbeit in Informatik, Fachbereich Informatik, Universität Kaiserslautern, 1993. (German Language).

[7] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[8] S. Schulz, A. Küchler, and C. Goller. Some Experiments on the Applicability of Folding Architecture Networks to Guide Theorem Proving. In D.D. Dankel II, editor, *Proc. of the 10th FLAIRS, Daytona Beach*, pages 377–381. Florida AI Research Society, 1997.

[9] Stephan Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788 of *LNAI*, pages 45–67. Springer, 2013.

[10] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[11] Robert Veroff. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *Journal of Automated Reasoning*, 16(3):223–239, 1996.