Antwortmengenprogrammierung

Event

Nicolas Jezuita
Supervision: Jera Hensel

SS 2017

Zusammenfassung

Antwortmengenprogrammierung ist eine deklarative Form der Programmierung und kann zum Beispiel zum Lösen schwieriger Suchprobleme genutzt werden. Was genau Antwortmengen sind und wie man sie generieren kann, wird in dieser Arbeit erklärt. Zunächst einmal werden Grundlagen geklärt, welche man zum Verständnis benötigt. Darauf aufbauend wird der Algorithmus erklärt der die Antwortmengen angibt. Es wird darauf eingegangen, wie man diese noch weiter anpassen kann, um bei unterschiedlichen Problemen schnell ein Ergebnis zu erzielen. Am Ende wird das neu konfigurierte Programm Cmodels2 mit den Tools Cmodels, smodels, smoddels$_{cc}$, DLV und ASSAT verglichen.

1 Einführung

Antwortmengenprogrammierung ist eine deklarative Form der Programmierung und kann zum Beispiel zum Lösen schwieriger Suchprobleme genutzt werden. In diesem Paper werden die Grundlagern erklärt, die man zum Verständnis und Anwenden von Programmen, welche Antwortmengen generieren, braucht. Dazu zählt, wie eine Regel eines Logik Programms aussieht und aus welchen Teilen sie besteht. Es wird definiert und erklärt was Antwortmengen sind und gezeigt, wie man sie mit Hilfe von *Vervollständigung* genieren kann.
Nachdem alle Grundlagen geschaffen sind, wird der Algorithmus ASP-SAT vorgestellt. Dieser wird dann Schritt für Schritt erläutert indem seine Methoden und Vorgehensweisen erklärt werden. Darüber hinaus werden Zusatzfälle betrachtet, wie zum Beispiel ein *verschachteltes* oder nicht *verschachteltes* Programm. Es wird gezeigt, worauf man bei diesen Fällen achten muss, sodass man effizient eine Antwortmenge generieren kann.
Zum Schluss wird der hier vorgestellte Algorithmus mit den Programmen Cmodels, smodels, smodels$_{cc}$, DLV und ASSAT verglichen. Unter Berücksichtigung der verschiedenen Eigenschaften eines Programms, wie zum Beispiel *verschachtelt*, *nicht verschachtelt*, *groß*, *klein* oder *zufällig generiert*, werden die einzelnen Methoden dann getestet. Dabei lässt sich erkennen, dass man bei den unterschiedlichen Eigenschaften andere Strategien verwenden muss, um ein optimales Ergebnis zu erzielen.

2 Grundlagen

2.1 Begriffserklärung

Ein Programm wird mit dem Symbol Π dargestellt. In der Syntax eines Logik Programms hat eine Regel folgende Form:

Abbildung 1: Notation eines Programms

$$p_0 \leftarrow p_1, ..., p_k, not\ p_{k+1}, ..., not\ p_m, not\ not\ p_{m+1}, ..., not\ not\ p_n$$

($0 \leq k \leq m \leq n$) [2]. Die Symbole p_0,p_1, ..., p_n sind Atome, die entweder den Wahrheitswert *wahr* oder *falsch* annehmen können. Ein Literal ist ein Atom das positiv (x) oder negativ ($\neg$ x) sein kann. Eine Klausel beschreibt eine Disjunktion von Literalen.
Der Pfeil $\leftarrow$ zeigt auf den Kopf der Regel. Der Körper ist alles, was rechts vom Pfeil steht. Kopf und Körper bestehen aus Literalen. Der Kopf ist in diesem Fall p_0 und der Körper alles von p_1 bis *not not* p_n. Die Regel schreibt vor, dass der Kopf erst erfüllt ist, wenn der Körper erfüllt ist. Das Symbol $\bot$ steht für die leere Disjunktion (*falsch*) und wenn es der Kopf einer Regel ist, wird diese auch *Bedingung* genannt. Das Symbol "*not*"wird als "negation as failure operator", also als Negation beim "Fehlschlagen", bezeichnet.

Ein einfaches Beispiel einer Regel ist folgendes:

$$nass \leftarrow Regen$$

Die Bedingung *nass* ist also nur erfüllt, wenn *Regen* erfüllt ist.

Ein Programm kann verschachtelt (nested) oder nicht verschachtelt (nonnested) sein. Ein verschachteltes Programm beinhaltet Atome, welche wie in Beispiel 1 ein *not not* b enthalten. Bei nicht verschachtelten Programmen ist dies nicht der Fall. Ein Beispiel für ein verschachteltes Programm ist:

Abbildung 2: Beispiel 1: ein verschachtelten Programm

$$a \leftarrow not\ not\ b$$

$$d \leftarrow c$$

Ein nicht verschachteltes Programm kann so aussehen:

Abbildung 3: Beispiel 2: ein nicht verschachteltes Programm

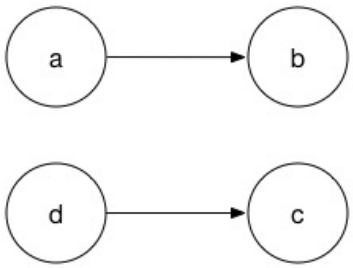
$$a \leftarrow a,\ b$$

$$a \leftarrow c$$

Des Weiteren kann ein Programm fest oder nicht fest sein. Ein Programm ist fest, wenn der zugehörige Abhängigkeitsgraph azyklisch ist. Falls es nicht fest ist, enthält das Programm mindestens eine Schleife, also einen Zykel, der beliebig oft besucht werden kann. Der Abhängigkeitsgraph ist ein gerichteter Graph G. Die Knoten des Graphen sind die Literale des Programms, also wie in Abbildung 4: a, b, c, d. Ist ein Literal Teil einer Regel, existiert eine Kante von dem Knoten, welcher den Kopf der Regel repräsentiert, zu dem

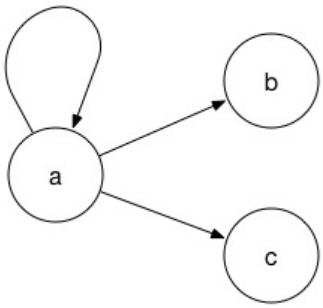
Knoten, des Literals. In diesem Fall sind die Knoten a und d die Köpfe jeweils einer Regel. Es existiert eine Kante von a nach b und von d nach c.

Abbildung 4: Abhängigkeitsgraph von Beispiel 1



Falls ein Programm nicht fest ist, enthält der Abhängigkeitsgraph einen Zykel. Dies wird in Abbildung 5 an dem Knoten a deutlich, der eine Kante zu sich selbst hat.

Abbildung 5: Abhängigkeitsgraph von Beispiel 2



2.2 Antwortmengen für Logik Progamme

Um die Antwortmenge zu erklären, nimmt man als erstes ein Programm Π, das keine Negation in der Formel enthält, sodass man in Beispiel eins k = m = n wählt und somit die Regel aus p_0, ..., p_k besteht. So hat das Programm Π nur eine Antwortmenge, welche abgeschlossen unter Π ist[2]. Das bedeutet das jedes der Atome p_1, ..., p_k *wahr* sein muss damit die Regel erfüllt ist. Um eine Antwortmenge zu bestimmen, definieren wir für einen Zustand S das Redukt auf Π wie folgt:

$$\Pi^S := \{p_0 \leftarrow p_1, ..., p_k \mid p_0 \leftarrow p_1, ..., p_k, not\, p_{k+1}, ..., not\, p_m \in \Pi, \{p_{k+1}, ..., p_m\} \cap S = \emptyset\}$$

Das Redukt Π^S entsteht also aus P in zwei Schritten:

1. Alle Regeln, in denen ein not p mit p ∈ S enthalten ist, werden entfernt.
2. In den verbleibenden Regeln werden alle negativen Literale (mit den zugehörigen not-Ausdrücken) entfernt.

[1]

Definition 2.1 *Seien P ein erweitertes logisches Programm ohne Default-Negation und S ein Zustand. S heißt Antwortmenge (answer set) von P, wenn S eine minimale, unter P geschlossene Menge ist (wobei Minimalität bzgl. Mengeninklusion gemeint ist) [1]*

Die Definition kann man nun erweitern durch die Ergänzung von Default-Negation.

Definition 2.2 *Seien P ein erweitertes logisches Programm und S ein Zustand. S heißt Antwortmenge von P, wenn S Antwortmenge des Reduktes P^S ist.[1]*

Führt man nun die Reduktion von Π aus, also Π^χ, werden die Negationen not p_{k+1},...,not p_m aus dem Körper gestrichen und aus jeder weiteren Regel des Programms, sodass nun folgende Regel vorliegt:

$$p_0 \leftarrow p_1, ..., p_k$$

Wir können sagen, dass χ eine Antwortmenge von Π ist, wenn χ eine Antwortmenge von Π^χ ist. Jede Antwortmenge von Π ist ein Modell von Π. Eine Menge χ eines Programms Π ist genau dann eine Antwortmenge, wenn χ die kleinste abgeschlossene Menge unter Π ist.

Um die Antwortmengen zu erklären werden im Folgenden ein paar Beispiele von Antwortmengen vorgestellt. Man hat folgendes Programm Π gegeben:

$$a \leftarrow a$$

$$b \leftarrow a,\ not\ d$$

$$c \leftarrow a,\ d$$

Nun prüfen wir die Menge {a,c} und bilden die Reduktion $\Pi^{\{a,c\}}$ und erlangen

$$a \leftarrow a$$

$$b \leftarrow a$$

$$c \leftarrow a,\ d$$

Somit ist die Menge {a,c} eine Antwortmenge von Π, da {a,c} eine Antwortmenge von Π wie auch $\Pi^{\{a,c\}}$ ist.

Wenn man nun andere Kombinationen versucht, wird man feststellen, dass {a,c} die einzige Antwortmenge ist. Es kann ebenfalls der Fall sein, dass ein Programm gar keine Antwortmenge hat. Dies wäre bei diesem Programm der Fall:

$$a \leftarrow not\ a$$

Falls man ein nicht festes Programm hat können auch mehrere Antwortmengen existieren. In dem Fall,

$$a \leftarrow b$$

$$b \leftarrow a$$

gibt es einmal die Antwortmenge {a} und {b}.

2.3 Vervollständigung

Vervollständigung (completion) wird auf ein Programm Comp(Π) angewendet.
Dabei ersetzt Vervollständigung die Atome, welche ein not not p_i besitzen mit p_i und wandelt alle not in das Negationssymbol ¬ um. So wird aus dem Beispiel 1:

$$a \equiv b$$

$$b \equiv c$$

Bildet man die Vervollständigung von Beispiel 2 erhält man folgendes:

$$a \equiv (a \wedge b) \vee c$$

Somit werden alle zu a gehörenden Regeln disjunkt. Die Vervollständigung einer Regel mit dem Kopf ⊥ wird als negierte Form umgewandelt. Die Vervollständigung dieses Programms,

$$\bot \leftarrow p_1, ..., p_k, not\ p_{k+1}, ..., not\ p_m, not\ not\ p_{m+1}, ..., not\ not\ p_n$$

wäre also

$$\bigvee_{i=1}^{k} \neg p_i \ \vee \bigvee_{i=k+1}^{m} p_i \ \vee \bigvee_{i=+m1}^{n} \neg p_i$$

[2] Nun können wir mit Hilfe der Vervollständigung eine Antwortmenge generieren, falls sie existiert.

2.4 Antwortmengengenerierung

Um eine Antwortmenge zu generieren können wir die Vervollständigung aus dem letzten Kapitel nutzen.

Theorem 2.1 *Nehme ein Programm* Π. *Wenn* χ *eine Antwortmenge von* Π *ist, dann entscheidet* χ *die Vervollständigung von* Π. *[2]*

Mit dem Theorem 2.1 hat man die Möglichkeit zu testen ob eine Antwortmenge existiert oder nicht. Dies ist genau dann der Fall, wenn die Antwortmenge auch die Vervollständigung des Programmes erfüllt. Betrachten wir nun feste Programme können wir auch die Rückrichtung des Theorems zeigen.

Theorem 2.2 *Nehme ein festes Programm* Π *und eine Menge von Atomen* χ. χ *ist eine Antwortmenge von* Π, *wenn* χ *die Vervollständigung von* Π *erfüllt. [2]*

Nun haben wir die Möglichkeit die Antwortmenge einfach über die Vervollständigung zu bestimmen, solange die Programme fest sind. Betrachtet man nun aber nicht feste Programme, muss man Theorem 2.1 und 2.2 noch erweitern. Falls in einem Programm Π eine Schleife enthalten ist, muss man die Schleifenformeln bestimmen. Eine Schleife kann man wie folgt definieren:

Definition 2.3 *Eine Schleife L eines Programms* Π *ist eine nicht leere Menge von Atomen, so dass für jedes Paar p, p' der Atome in L ein Pfad einer Länge ungleich 0 von p zu p' im Abhängigkeitsgraphen von* Π *existiert.[2]*

In dem Beispiel 2 ist eine Schleife vorhanden. Das kann man in dem Abhängigkeitsgraphen erkennen. Um eine Antwortmenge aus einem nicht festen Programm zu bestimmen benötigt man die Schleifenformel. Dazu definieren wir die Menge aller Schleifenformeln in Π mit R(L). Die Schleifenformel des Beispiels 2 ist c ← a Nun können wir daraus Theorem 2.3 folgern.

Theorem 2.3 *Nehme ein Programm* Π. *Bilde Comp(*Π*). Sei LF(*Π*) die Menge aller Schleifenformeln, zugehörig zu den Schleifen von* Π. *Für jede Menge von Atomen* χ *ist* χ *eine Antwortmenge von* Π *wenn* χ *ein Modell von Comp(*Π*)* ∪ *LF(*Π*)*

Zieht man nun die Schleifenformeln hinzu, kann man aus der Vervollständigung und der Schleifenformel eine Antwortmenge generieren. Das heißt im Beispiel 2 besagt die Schleifen Formel, dass wenn a *wahr* ist, auch c *wahr* ist.

3 Cmodles2

Antwortmengen-Generatoren (answer set solvers) bieten die Möglichkeit, durch einen Erfüllbarkeitsüberprüfe eine Antwortmenge zu erzeugen. Enrico Giunchiglia, Yuliya Lierler und Marco Maratea haben einen Algorithmus entworfen, welcher durch ein paar Anpassungen eine Antwortmenge erzeugen kann. Zur Implementierung des neuen Programms Cmodels2 wurde der bereits existierende Generator CModels genutzt, welcher ein festes Programm übergeben bekommt. Die meisten erfüllbarkeisprüfenden Antwortmengen-Generatoren (ASP-SAT solvers) gehen wie folgt vor:

1. sie generieren Comp(Π)
2. dann starten sie einen Erfüllbarkeitsüberprüfer, welcher dann eine Antwortmenge von der jeweiligen Vervollständigung erzeugt.

Zuvor wurde der Erfüllbarkeitsüberprüfer lediglich als "Black Box"genutzt, was bedeutet, dass man ein Programm übergibt und ein Ergebnis dafür erlangt. Nutzt man den Erfüllbarkeitsüberprüfer jedoch nicht als Black Box, sondern passt ihn etwas an, kann man damit für nicht feste Programme, also Programme mit Schleifen, eine Antwortmenge generieren, ohne dass die Formeln exponentiell größer als das gegebene Programm werden. Mit dem Rücksetzverfahren (Backtracking) kann man die Probleme eleminieren, falls es viele Schleifenformeln gibt und falls es viele Antwortmengen gibt.
Der Algorithmus ASP-SAT generiert eine Antwortmenge. Dabei bekommt er ein Programm Π übergeben und ruft die Methode DLL auf. Die Methode DLL bekommt eine Vervollständigung von Π in KNF, eine Menge S und Π übergeben. Die Vervollständigung von Π in der KNF wird mit Γ beschrieben. Zuerst prüft DLL, ob die Menge Γ leer ist und ruft, wenn dies der Fall ist, die Methode test auf, welche S und Π als Parameter übergeben bekommt. Falls die leere Menge ein Element der Menge Γ ist, gibt DLL False zurück. Falls ein einzelnes Literal in der Menge Γ vorhanden ist, wird die Methode DLL mit den Parametern assign(l.Γ), S ∪ {p} und Π aufgerufen. Ansonsten wird die Methode DLL einmal mit den Parametern assign(p,Γ), S ∪ {p} und Π und einmal mit den Parametern assign(¬p,Γ), S ∪ {p} und Π aufgerufen.
Der Algorithmus wird also so lange ausgeführt, bis die Menge Γ keine Elemente mehr enthält. Als erstes wird überprüft, ob einzelne Literale l noch in Γ enthalten sind. Das bedeutet, dass z.B. in einer Menge {{a},{b,c},{¬b,c}} das Literal {a} entfernt und auf *wahr* gesetzt wird.

Abbildung 6: ASP-SAT Algorithmus zum Generieren von Antwortmengen [2]

function ASP-SAT(Π)
return DLL(CNF(($Comp(\Pi)$), $\emptyset, \Pi$)
end function

function DLL(Γ, S, Π)
  if $\Gamma = \emptyset$ **then return** test(S, Π)
  end if
  if $\Gamma \in \emptyset$ **then return** DLL($assign(p, \Pi), S \cup \{p\}, \Pi$)
  end if
  if $\{l\} \in \Pi$ **then** DLL($assign(l, \Pi), S \cup \{l\}, \Pi$)
  end if
p := ein Atom aus Γ ;
return DLL($assign(p, \Pi), S \cup \{p\}, \Pi$) oder
DLL($assign(\neg p, \Pi), S \cup \{p\}, \Pi$)
end function

Zudem wird jede Klausel, welche ein a enthält, aus Γ entfernt und $\neg$ a aus jeder Klausel in Γ gelöscht. Wenn Γ die Menge {{b,c},{$\neg$b,c}} ist und somit keine einzelnen Literale hat und nicht leer ist wird ein zufälliges Literal z.B. b ausgewählt und auf *wahr* gesetzt. Danach wird es aus Γ entfernt und in S eingefügt, was bedeutet, dass alle Klauseln in denen b aus Γ entfernt werden und $\neg$ b aus jeder Klausel in Γ gelöscht wird (das gilt umgekehrt für $\neg$ b). Wenn Γ keine Elemente mehr hat, wird die Methode test aufgerufen. Die Methode test überprüft, ob in der erstellten Menge mindestens eine Antwortmenge vorhanden ist und gibt dann *wahr* zurück, falls dies der Fall ist, anderen Falls *falsch*. Angenommen P ist die Menge aller Atome des Programms. Falls das Programm nicht verschachtelt ist muss die Methode test lediglich überprüfen, ob S $\cap$ P eine Antwortmenge ist. Dies gilt nach Theorem 3.1

Theorem 3.1 *Nehme ein nicht verschachteltes Programm* Π. *Sei* χ *eine Menge von Atomen, welche Comp(*Π*) erfüllt. Wenn* $\chi \subset \chi$', *dann ist* χ' *keine Antwortmenge von* Π.

Daraus lässt sich schließen, dass durch die Methode test das ASP-SAT bei nicht verschachteltem Programm Π, *wahr* zurück gibt falls eine Antwortmenge existiert und anderen Falls *falsch*.
Betrachtet man nun ein verschachteltes Programm, können zwei Mengen χ und χ' existieren, sodass χ eine Antwortmenge von Π ist und χ' eine Teilmenge von χ und ebenfalls eine Antwortmenge von Π. Daher muss immer wenn die Methode test aufgerufen wird, überprüft werden, ob

1. Die Menge S $\cap$ P
2. und eine Teilmenge von { p: $\neg$ p $\notin$ S, p $\in$ P}

Um den Algorithmus weiter zu optimieren, kann man ihn "lernen"lassen. Das bedeutet, dass man eine Menge R speichern kann, bei der man sicher ist, dass wenn R eine Teilmenge von S ist, die Menge S keine Antwortmenge hat. Die Menge R falsifiziert also Γ. Um eine optimierte Form dieses Mechanismus zu erzeugen, ist es wichtig, dass R so klein wie möglich bleibt. Zum Beispiel können Schleifenformeln dabei helfen, da wenn diese nicht erfüllt sind das Programm sowieso nicht erfüllt wird. Somit kann man versuchen

die Schleifenformeln zu falsifizieren und somit eine Menge zu S' zu erlangen, welche das Programm falsifiziert.

Zusammengefasst kann der ASP-SAT Algorithmus folgendes:

1. er sucht nach der Lösung von Comp(Π) ohne, dass er mehr Variablen bei der Umformung in eine Klausel nimmt als benötigt
2. er löst das Problem in polynomialer Zeit
3. er kann sowohl feste als auch nicht feste Programme lösen.

3.1 Vergleich

In den Benchmark Tests hat im Vergleich mit anderen Erfüllbarkeitsüberprüfern der in diesem Paper vorgestellte Prüfer relativ gut abgeschnitten. Er wurde mit vier verschiedenen Strategien getestet, welche mit verschiedenen Problemen geprüft wurden.

Folgende Strategien wurden für den Test benutzt:

1. ulv: Unit propagation, backtracking enhanced with Learning, and VSIDS heuristic.
2. fbu: Unit propagation enhanced with Failed literal detection, standard Backtracking, and the Unit heuristic.
3. flv: Unit propagation enhanced with Failed literal detection, backtracking enhanced with Learning, and the VSIDS heuristic.
4. flu: Unit propagation enhanced with Failed literal detection, backtracking enhanced with Learning, and the Unit heuristic.

   In den folgenden Tabellen kann man nun Standard-Tests sehen, welche verschiedene Eigenschaften haben.

   Die erste Tabelle zeigt Testläufe mit zufällig erzeugten Programmen. Die Probleme von eins bis zehn sind fest und die von elf bis zwanzig nicht feste. Hier kann man erkennen, dass die ASP-SAT Methode unter Nutzung der verschiedenen Strategien sehr unterschiedliche Ergebnisse erzielt. Dennoch sind diese immer sehr gut. Sehr stark sind hier ebenfalls die Methoden Smodels für die ersten zehn Tests und $\text{smodels}_{\text{cc}}$ für die Tests elf bis fünfzehn.

   Die zweite Tabelle zeigt die Performance bei großen Programmen. Hier sind die Probleme der Zeile einundzwanzig bis sechsundzwanzig fest und die Probleme siebenundzwanzig bis neunundzwanzig nicht fest. Hier sticht die Methode mit der ulv-Strategie heraus. Die festen Probleme wird lediglich von der ASSAT Methode ähnlich gut bewältigt. Bei den nicht festen Problemen, ist wieder die Methode $\text{smodels}_{\text{cc}}$ sehr stark.

   Zusammenfassend kann man sagen, dass die Methode ASP-SAT im Vergleich zu den Erfüllbarkeitsprüfern beim jetzigen Stand der Technik sehr gut abschneidet. Im Vergleich der einzelnen Probleme kann man jedoch gut erkennen, dass die Art der Strategie sehr entscheidend ist. Daher sollte man sich in Zukunft auch noch mehr an den Problemen orientieren und versuchen dafür optimierte Algorithmen zu erstellen.

Abbildung 7: Die erste Tabelle zeigt die Performance bei zufällig generierten Programmen [2]

	PB	#VAR	SMODELS	SMODELS$_{cc}$	ASSAT	DLV	ulv	flv	flu	fbu
1	4	300	1.2	7.23	0.85	2.55	**0.59**	0.8	1.5	1.37
2	4.5	300	**39.97**	TIME	TIME	130.49	TIME	TIME	115.29	40.38
3	5	300	**7.57**	149.37	TIME	26.78	456.22	538.89	17.64	11.32
4	5.5	300	**2.26**	33.12	94.78	7.37	72.83	53.26	4.42	3.59
5	6	300	**1.05**	12.72	22.5	3.26	24.73	21.89	1.83	1.63
6	4	350	4.11	12.6	13.4	49.3	**2.2**	5.74	11.48	8.85
7	4.5	350	**318.1**	TIME	TIME	TIME	TIME	TIME	TIME	384.66
8	5	350	**44.2**	TIME	TIME	147.16	TIME	TIME	134.34	54.07
9	5.5	350	**12.66**	252.11	TIME	32.07	TIME	506.08	20.37	13.61
10	6	350	**3.37**	37.99	174.61	8.76	95.61	104.36	6.05	4.86
11	4	200	3.3	2.02	2.44	32.39	5.34	3.32	1.93	**1.75**
12	4.5	200	6.84	**1.7**	3.28	83.63	6.15	5.82	2.09	1.93
13	5	200	22.8	**2.5**	8.21	82.97	9.82	9.02	3.88	3.33
14	5.5	200	9.42	**1.76**	4.14	39.47	7.5	6.38	2.97	2.85
15	6	200	8.12	**0.85**	1.4	23.93	3.24	2.95	1.25	1.53
16	4	300	298.67	73.64	234.09	TIME	265.43	218.48	41.97	**31.05**
17	4.5	300	TIME	TIME	TIME	TIME	TIME	TIME	190.73	**135.11**
18	5	300	TIME	412.69	TIME	TIME	TIME	TIME	136.67	**99.75**
19	5.5	300	TIME	233.72	TIME	TIME	TIME	TIME	129.29	**78.63**
20	6	300	TIME	191.62	TIME	TIME	TIME	TIME	107.34	**65.83**

Problems (1)–(10) are tight programs being the translation of 3-SAT benchmarks. Problems (11)–(20) are randomly generated logic programs using Lin and Zhao's methodology.

4 Zusammenfassung

In dem Paper wurden die Grundlagen erklärt, welche man benötigt um zu verstehen, wie eine Antwortmenge mit Hilfe von Erfüllbarkeitsprüfern erzielt werden kann. Es wurden Regeln eins Logik Programms erklärt und die Eigenschaften verschachtelt, nicht verschachtelt, fest und nicht fest definiert.
Die Syntax von Logik Programmen wurde erklärt und gezeigt, aus welchen Teilen sie besteht. Damit konnten mit Hilfe von Reduktion die Antwortmengen definiert und anhand von Beispielen erklärt werden. Durch die Vervollständigung ist es möglich, eine Antwortmenge sowohl für feste als auch nicht feste Programme zu generieren. Der Algorithmus ASP-SAT wurde vorgestellt und Schritt für Schritt erklärt. Dieser wurde dann durch Backtracking und der Möglichkeit des Lernens in Form einer Menge R erweitert.
Die vorgestellten Methoden wurden mit anderen Prüfern vom Stand der heutigen Technik verglichen und die Probleme wurden weiter spezifiziert. Durch das Unterteilen in einzelne Gruppen, wurden die Stärken und Schwächen der einzelnen Erfüllbarkeitsprüfer aufgedeckt. Um diese Prüfer weiter zu optimieren, sollte man sich an den verschiedenen Eigenschaften von Programmen, wie zufällig generierten Programmen oder großen Programmen, orientieren. Es wurde im Vergleich gezeigt, dass man mit unterschiedlichen Methoden bei den gegebenen Eigenschaften bessere Ergebnisse erzielen kann. Daher sollte man sich in

Abbildung 8: Die zweite Tabelle zeigt die Performance bei großen Programmen [2]

	PB	#VAR	SMODELS	SMODELS$_{cc}$ASSAT	DLV	ulv	flv	flu	fbu
21	bw*d9	9956+	6.76	7.63	1.72		**1.02**	5.84	2.69	2.75
22	bw*e9	12260	4.3	4.51	4.22		**0.98**	1.91	1.92	1.93
23	bw*e10	13482+	11.15	12.43	2.66		**1.29**	7.51	5.03	4.95
24	4c1000	14955+	22.28	4.95	0.6		**0.48**	37.86	15.41	15.23
25	4c3000	44961+	202.84	1143.13	**2.19**		8.86	369.27	144.12	142.83
26	4c6000	89951+	856.13	TIME	**14.85**		99.50	TIME	583.55	578.98
27	np60c	10742+	242.61	30.81	84.87	361.80	**2.83**	1611.32	44.12	44.11
28	np70c	14632+	557.08	55.31	520.80	798.96	**4.69**	TIME	97.44	97.87
29	np80c	19122+	1001.88	90.59	53.25	1587.60	**7.2**	TIME	195.08	190.49
30	np90c	24212+	2064.61	144.72	1416.24	2807.84	**10.42**	TIME	364.54	357.92
31	np100c	29902+	3573.19	215.37	TIME	TIME	**14.23**	TIME	610.2	608.96
32	np60c	10683+	7.05	3.82			**3.55**	340.86	8.03	7.82
33	np70c	14563+	15.67	**5.92**			10.54	782.69	15.39	14.92
34	np80c	19043+	32.29	**9.01**			15.05	1538.86	23.63	25.94
35	np90c	24123+	53.21	**14.13**			32.19	2918.82	38.75	50.08
36	np100c	29803+	83.11	**14.95**			34.18	TIME	59.15	62.64
37	mutex4	14698+	14.14	5.35	0.54	367.89	**0.46**	28.29	28.3	28.26
38	mutex3	278074+	163.94	**110.27**	MEM		TIME	TIME	TIME	TIME
39	phi3	16930+	3.23	3.04	53.28		**1.43**	55.62	12.15	TIME

Problems (21)–(26) are tight. Problems (27)–(39) are non-tight.

der Zukunft darauf konzentrieren an einer optimierten Form der Erfüllbarkeitsprüfer zu arbeiten, welche diese Methoden kombiniert.

Literatur

[1] G. K.-I. C. Beierle and G. Kern-Isberner. *Methoden wissensbasierter Systeme*. Springer, 2003.

[2] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated reasoning*, 36(4):345–377, 2006.