Bachelor Projekt — Grundlagen der Künstlichen Intelligenz

B. Nebel, T. Engesser, R. Mattmüller, D. Speck
Universität Freiburg
Wintersemester 2019/20
Institut für Informatik

Übungsblatt 1

Aufgabe 1.1 (Tag 1: Aussagenlogische Erfüllbarkeit (SAT))

(a) Mit MiniSat vertraut machen:

MiniSat ist ein SAT-Solver. Das heißt, dass MiniSat eine aussagenlogische Formel als Eingabe nimmt und entscheidet, ob diese Formel erfüllbar ist oder nicht. Falls sie erfüllbar ist, gibt MiniSat zudem eine erfüllende Belegung aus, d.h., ein *Modell* für die eingegebene Formel. Mehr Informationen zu MiniSat stehen auf `http://minisat.se`. Hier kann MiniSat auch heruntergeladen werden. Alternativ ist MiniSat in den Repositories einiger Linux-Distributionen enthalten. Unter Ubuntu zum Beispiel kann MiniSat per `sudo apt install minisat` installiert werden. Nachdem MiniSat installiert ist, erstelle eine Datei namens `modusponens.cnf` und schreibe die folgende SAT-Problembeschreibung im DIMACS-Format hinein:

```
p cnf 2 3
-1 2 0
1 0
2 0
```

Die erste Zeile eines DIMACS-Files beginnt stets mit der Zeile `p cnf numvar numclauses`, wobei `numvar` die Anzahl der aussagenlogischen Variablen und `numvar` die Anzahl der Klauseln angibt. Das `p` steht für "Problem" und `cnf` meint, dass das Problem in Konjunktiver Normalform angegeben ist, d.h., als Konjunktion von Disjunktionen (Klauseln). Die folgenden Zeilen geben die Klauseln an. Jede Zahl zwischen `1` und `numvar` steht für eine Variable. Mit `-Zahl` wird die Negation der Variable `Zahl` notiert. Jede Zeile endet mit einer 0, die sonst ohne weitere Bedeutung ist. Die erste Klausel im obigen Beispiel repräsentiert die Klausel $\neg 1 \vee 2$, d.h., $1 \rightarrow 2$. Die zweite und dritte Klausel bestehen aus nur je einer Variable. Das Problem ist natürlich erfüllbar (Modus Ponens). Dass auch MiniSat dieser Meinung ist, kann mit dem Befehl `minisat modusponens.cnf modusponens.model` überprüft werden. In der Datei `modusponens.model` steht das Ergebnis der Erfüllbarkeitsprüfung, SAT (erfüllbar) oder UNSAT (unerfüllbar). Im Falle von SAT wird auch ein Modell für das Problem ausgegeben. Für das obige Beispiel steht nach der Ausführung in `modusponens.model`:

```
SAT
1 2 0
```

Diese Ausgabe bedeutet, dass das Problem erfüllbar ist (SAT) und $I(1) = True, I(2) = True$ ein Modell, d.h., I ist eine Interpretation, unter der die Klauselmenge wahr ist.

(b) Benutze MiniSat, um zu beweisen, dass aus der Formelmenge $\{a \rightarrow b, \neg b\}$ die Formel $\neg a$ logisch folgt, d.h., $\{a \rightarrow b, \neg b\} \models \neg a$. Erinnere Dich daran, dass $\theta \models \phi$ gdw. $\theta \cup \{\neg\phi\} \models \bot$.

(c) Es sollen beliebige aussagenlogische Formelmengen in das DIMACS-Format übersetzt werden. Implementiere dafür zunächst einen Parser für die folgende Sprache:

$$\phi \to a \,|\, b \,|\, c \,|\, \ldots \,|\, And(\phi,\phi) \,|\, Or(\phi,\phi) \,|\, Impl(\phi,\phi) \,|\, BiImpl(\phi,\phi) \,|\, Not(\phi) \,|\, TOP \,|\, BOT$$

Erweitere den Parser um die Funktion, das Parsing-Ergebnis im DIMACS-Format auszugeben. Dafür muss also zunächst eine KNF erzeugt werden. Beispielsweise sollte die String-Eingabe

```
And(Impl(Not(a), b), And(Not(b), Not(a)))
```

... erst in eine geeignete Datenstruktur überführt werden, etwa einen Syntax-Baum,
...
... dann in die KNF-Darstellung

```
And(Or(a, b), And(Not(b), Not(a)))
```

... und schließlich in jene Darstellung überführt werden:

```
p cnf 2 3
1 2 0
-2 0
-1 0
```

(d) Mit dem Nurse-Scheduling-Problem soll exemplarisch ein NP-vollständiges Problem zunächst aussagenlogisch kodiert und dann gelöst werden. Benutze dafür Deinen Übersetzer, um eine DIMACS-Repräsentation der Formelmenge, die die unten stehende Probleminstanz repräsentiert, zu erstellen. Benutze schließlich MiniSat, um das Problem zu lösen (z.B. um eine Lösung zu finden, oder die Nicht-Existenz einer Lösung festzustellen). Die Probleminstanz stellt sich wie folgt dar:

In einem Krankenhaus sollen Krankenpfleger und -Pflegerinnen auf Schichten am kommenden Wochenende eingeteilt werden. An jedem der beiden Tage gibt es drei Schichten: die Frühschicht, die Spätschicht und die Nachtschicht. Ein(e) Krankenpfleger(in) arbeitet pro Arbeitstag nur eine Schicht. Wer die Nachtschicht arbeitet, soll am nächsten Tag nicht für die Frühschicht eingeteilt werden. In jeder Schicht sollen zwei Krankenpfleger(innen) arbeiten. Es stehen insgesamt sechs PflegerInnen zur Verfügung: Heinz, Frida, Udo, Ira, Nora und Fritz. Nora und Ira verstehen sich nach einem Vorfall letztens nicht mehr gut und sollen nicht gemeinsam für eine Schicht eingeteilt werden. Fritz und Frida sollen immer zusammen arbeiten. Fritz arbeitet niemals Nachts, Nora nur Nachts.

(e) MiniSat gibt im Falle einer erfüllbaren Formel nur ein Modell aus, obwohl es mehrere Modelle geben könnte. Überlege Dir einen systematischen Weg, um mittels wiederholter Aufrufe von MiniSat mit jeweils angepasster Formel alle Modelle auszugeben. Implementiere diesen Weg.

Aufgabe 1.2 (Tag 2: Stabile Modelle)

(a) Stabile Modelle (stable models) verstehen:

In dieser Aufgabe betrachtest Du Mengen von Implikationen der Form

$$a \leftarrow b_1 \wedge \ldots \wedge b_n \wedge \neg c_1 \wedge \ldots \wedge \neg c_m$$

, die als *Programme* bezeichnet werden. Dass die Implikationszeichen hier "verkehrtherum" notiert werden, ist pure Konvention. Jede Implikation in einem Programm heißt auch *Regel*, die aus einem *Kopf* und einem *Körper* besteht. In dem Beispiel ist das Atom a der Kopf und die Konjunktion $b_1 \wedge \ldots \wedge b_n \wedge \neg c_1 \wedge \ldots \wedge \neg c_m$ der Körper der Regel. Eine Regel heißt *Fakt*, falls der Körper leer ist ($n = m = 0$), er wird dann $a \leftarrow \top$ notiert. Eine Regel mit leerem Kopf heißt *Constraint*, und wird als $\bot \leftarrow b_1 \wedge \ldots \wedge b_n \wedge \neg c_1 \wedge \ldots \wedge \neg c_m$ notiert.

Die Ausgabe eines Programms sind gerade seine Modelle. Das Programm $P_1 = \{a \leftarrow b\}$ zum Beispiel hat drei Modelle $I_1 = \{a \mapsto True, b \mapsto True\}, I_2 = \{a \mapsto True, b \mapsto False\}, I_3 = \{a \mapsto False, b \mapsto False\}$. Im Folgenden seien Modelle durch Mengen von Atomen notiert, denen die Modelle den Wahrheitswert $True$ zuordnen. Für das Beispiel gibt es also die drei Modelle $\{a, b\}, \{a\}, \{\}$. Das Beispielprogramm P_1 besagt, dass a gelten soll, falls b gilt. Da $b \notin P_1$, gibt es intuitiv keinen Grund anzunehmen, dass a gilt. Aus Sicht der Logikprogrammierung stellt deshalb nur das Modell $\{\}$ eine Ausgabe des Programmes P_1 dar. Im Programm $P_2 = \{a \leftarrow b, b\}$ gibt es hingegen einen Grund zur Annahme von b (es ist ein Fakt) und von a (als Kopf einer nun anwendbaren Regel). Entsprechend ist $\{a, b\}$ ein gesuchtes Modell für Programm P_2. Generell sollen Modelle von Programmen genau aus jenen Atomen bestehen, die durch das Programm unterstützt (supported) und begründet (founded) sind. Ein Atom in einem Modell M heißt *unterstützt*, wenn es im Kopf einer bezüglich M anwendbaren Regel steht. Eine Regel ist bezüglich M *anwendbar*, falls gilt: Wenn jedes positive Atom im Körper der Regel in M ist und keines der negativen Atome im Körper positiv in M ist, dann ist der Kopf in M. Ein Atom in M heißt *begründet*, falls es sich aus dem bezüglich M reduzierten Programm P^M aus den verbleibenden Fakten und einer (eventuell leeren) Sequenz von Regelanwendungen (per Modus Ponens) ableiten lässt. P^M heißt *Reduktion von* M, und berechnet sich wie folgt: Jede Regel in P, die $\neg b$ im Körper hat, so dass $b \in M$, wird gelöscht (sie ist nicht anwendbar). Aus allen verbleibenden Regeln werden alle negativen Konjunkte gelöscht (sie sind in M bereits erfüllt). Ein Modell M für ein gegebenes Programm heißt *stabiles Modell*, wenn alle Atome in M unterstützt und begründet sind.

Hier zwei zusätzliche Beispiele, die Unterstützung und Begründung verdeutlichen sollen: Gegeben sei $P_3 = \{a \leftarrow \neg b, b \leftarrow \neg a\}$. Das Programm P_3 verfügt über die drei Modelle $\{a\}, \{b\}, \{a, b\}$. Allerdings ist das Modell $\{a, b\}$ nicht unterstützt: Weil $a \in M$, ist die zweite Regel nicht anwendbar, und somit ist b nicht unterstützt—dasselbe gilt für a. Entsprechend überlebt keine der beiden Regeln in P_3 die Reduktion, d.h., $P_3^{\{a,b\}} = \{\}$. Im Falle des Programmes $P_4 = \{a \leftarrow b, b \leftarrow a\}$ sind die Atome in $\{a, b\}$ unterstützt: Beide Regeln bleiben anwendbar und haben die Atome im Kopf. Allerdings gilt $P_4^{\{a,b\}} = P_4$, und in P_4 lässt sich aufgrund fehlender Fakten kein Atom ableiten.

Die Aufgabe besteht nun darin, ein Programm P derart zu modifizieren, dass die Modelle des modifizierten Programmes genau seine stabilen Modelle sind. Im ersten Schritt sollen die Modelle mit nicht-unterstützten Atomen eliminiert werden. Das funktioniert durch die sogenannte Clark's Completion. Im zweiten Schritt werden dem modifizierten Programm sogenannte Loop-Formeln hinzugefügt, die dafür sorgen, dass Modelle mit unbegründeten Atomen eliminiert werden.

(b) Implementation: Wir nehmen fortan an, dass es sich bei den Eingabe-Formeln um Konjunktionen von Implikationen (wie oben beschrieben) handelt. Der Übersetzer aus der vorherigen Aufgabe soll mit der Option gestartet werden können, solche Eingaben derart zu modifizieren, dass der SAT-Solver nur noch Modelle findet, die aus unterstützten Atomen bestehen. Dafür wird aus der geparsten Eingabe zunächst die Clark's Completion (CL) erstellt, bevor diese dann ins DIMACS-Format übersetzt wird.

Die Clark's Completion $cmpl(P)$ eines Programmes P wird wie folgt konstruiert: Für alle Regeln $Impl(Body_1, p), Impl(Body_2, p) \ldots \in P$ ist

$$BiImpl(Or(Body_1, Or(Body_2, \ldots)\ldots), p) \in cmpl(P)$$

(dies umschließt auch alle Fakten—$Impl(TOP, p)$—und auch alle Constraints—$Impl(Body, BOT)$—als spezielle Regeln). Für jedes Atom p, das nicht im Kopf einer Regel in P vorkommt, wird die Formel $BiImpl(p, BOT)$ der Clark's Completion hinzugefügt.

(c) Nimm Dir einen Moment Zeit, um zu verstehen, wieso die Clark's Completion wirklich alle nicht-unterstützten Modelle eliminiert (aber alle anderen beibehält). (Ohne Beweis.)

(d) Wie im obigen Text erläutert, kann auch die Clark's Completion nicht-stabile Modelle haben, weil zwar alle Atome in jedem Modell unterstützt sind, aber nicht unbedingt begründet. Das Problem sind zyklische Regeln wie im obigen Programm P_4, die intuitiv einer "Begründung von außerhalb des Zyklus" bedürfen, um einen Regelkopf begründet ableiten zu können. Um auch unbegründete Modelle zu eliminieren, soll der Übersetzer nach der Clark's Completion zusätzlich sogenannte Loop-Formeln (LF) hinzufügen.

Um die Loop-Formeln zu bestimmen, wird zunächst ein Abhängigkeitsgraph G^P zum Programm P erstellt: Die Knoten des Abhängigkeitsgraphen sind gerade die Atome, die in P vorkommen. Zwei Knoten p, q werden in dem Graphen durch eine gerichtete Kante zwischen p und q miteinander verbunden, wenn es eine Regel $Impl(Body, p) \in P$ gibt, sodass $q \in Body$. Nachdem der Graph G^P erstellt wurde, werden seine stark-verbundenen Komponenten bestimmt, d.h., alle maximalen Teilmengen S der Knoten von G^P, sodass im Graphen für alle $u, v \in S$ ein Pfad von u nach v existiert. Eine Menge von Atomen L heißt *Loop des Programs* P, wenn der Teilgraph von G^P, der durch L induziert ist, stark-zusammenhängend ist. Nun assoziieren wir zwei Mengen mit einem Loop L: $R^+(L) = \{Impl(B, p) \,|\, p \in L, \exists q.q \in B \wedge q \in L\}$ und $R^-(L) = \{Impl(B, p) \,|\, p \in L, \neg\exists q.q \in B \wedge q \in L\}$. $R^+(L)$ enthält also alle Regeln, die sich im Loop L befinden, und $R^-(L)$ alle Regeln außerhalb des Loop. Das Ziel besteht darin, jene Modelle zu eliminieren, die Atome enthalten, die nicht durch Regeln außerhalb des Loops begründet sind. Sind nun also $Impl(Body_{11}, p_1) \ldots Impl(Body_{1k_1}, p_1), \ldots Impl(Body_{n1}, p_n) \ldots Impl(Body_{nk_n}, p_n)$ die Regeln in $R^-(L)$, dann füge die Loop-Formel

$$Impl(Not(Or(Body_{11}, Or(Body_{12}, \ldots)\ldots)), And(Not(p_1), And(Not(p_2), \ldots)\ldots))$$

hinzu.

(e) Nimm Dir einen Moment Zeit, um zu verstehen, wieso durch das Hinzufügen der Loop-Formeln zur Clark's Completion wirklich alle nicht-begründeten Modelle eliminiert werden (und nur diese). (Ohne Beweis.)

Aufgabe 1.3 (Tag 3: Answer-Set-Programming (ASP))

(a) Installiere den ASP-Solver Clingo (`https://potassco.org/`) und mache Dich mit ihm vertraut. Ein sehr gutes, ausführliches Manual von den Clingo-Autoren kann unter `https://sourceforge.net/projects/potassco/files/guide/2.0/` heruntergeladen werden.

(b) Answer-Sets anhand von Beispielen qualitativ mit SAT+CL+LF-Modellen vergleichen: Führe das folgenden Programm sowohl mit Clingo als auch mit dem MiniSat (nach der Clark's Completion und dem Hinzufügen der Loop-Formeln) aus: $P = \{a \leftarrow \neg b, b \leftarrow \neg a\}$. Vergewissere Dich, dass die generierten Modelle von MiniSat mit den von Clingo berechneten Answer-Sets identisch sind. Um alle Answer-Sets von Clingo anzuzeigen, starte Clingo mittels `clingo 0 <file>`.

(c) Kodiere das Nurse-Scheduling-Problem aus der ersten Aufgabe als ASP-Programm und benutze Clingo, um es zu lösen. Führe Gemeinsamkeiten und Unterschiede der SAT- und ASP-Kodierungen auf.

(d) In der vorherigen Aufgabe hast Du ausprobiert, wie ein SAT-Solver genutzt werden kann, um Answer-Sets (stabile Modelle) zu generieren. Ein ASP-Solver kann auch als SAT-Solver verwendet werden. Dafür müssen die nicht-stabilen Modelle hervorgezaubert werden. Interessanterweise funktioniert das, indem man eine einzige sogenannte Choice-Formel $\{p_1; p_2; \ldots; p_n\}.$, die alle Atome p_i aus der ASP-Kodierung des SAT-Problems enthält, einfügt. Damit wird erzwungen, dass der ASP-Solver alle Teilmengen der Atome mal als wahre Fakten annimmt. Probiere dies einmal mit der Eingabe

```
a :- b.
```

aus. Nimm Dir einen Moment Zeit, um zu verstehen, weshalb nun die Answer-Sets genau den klassischen Modellen für $a \leftarrow b$ entsprechen.

Aufgabe 1.4 (Tag 4: Nichtmonotones Schließen)

- Eine zentrale Charakteristik von ASP ist die Unterscheidung zweier Arten von Negation: `not p` und `-p`. Eine Regel der Form

  ```
  a :- not b.
  ```

  kann gelesen werden als "normalerweise gilt a, es sei denn b gilt". Dieses Mittel kann genutzt werden, um Default-Regeln zu notieren. Betrachte dafür das klassische Beispiel über die Flugunfähigkeit von Pinguinen als ASP-Programm:

  ```
  fliegt :- vogel, not -fliegt. -fliegt :- pinguin. vogel. pinguin.
  ```

  Die erste Regel besagt, dass ein Vogel normalerweise fliegt (es sei denn es spricht etwas dagegen, dass er fliegt). Die zweite Regel sagt, dass Pinguine nicht fliegen. Außerdem besagen die Fakten, dass wir es mit einem Vogel, der außerdem ein Pinguin ist, zu tun haben.

  - Führe das Programm über Pinguine einmal mit dem Fakt `pinguin.` und einmal ohne diesen Fakt mit Clingo aus. Notiere die Modelle.
  - Überlege Dir zunächst, wie das ASP-Programm korrekt als aussagenlogische Formeln übersetzt werden kann. Berücksichtige dabei, dass $-fliegt$ eigentlich als ein von $fliegt$ unabhängiges Atom behandelt wird, und dass implizit noch ein zusätzliches Constraint

    ```
    :- fliegt, -fliegt.
    ```

    zum Programm gehört.
  - Benutze als nächstes Dein MiniSat-basiertes Programm mit Clark's Completion und Loop-Formeln aus der zweiten Aufgabe, um bezüglich des Pinguin-Programms dasselbe Verhalten wie Clingo zu erzeugen (wieder einmal mit Fakt *pinguin.* und einmal ohne). Vergleiche das Verhalten auch mit der Ausgabe von MiniSat ohne Clark's Completion und Loop-Formeln.