Homework assignment 2 – Symbolic Systems I – UvA, June 2020

Yke Rusticus

Question 1

We are given a set of propositional formulas Φ and another formula φ. Our task is to decide whether $\Phi \vDash \varphi$ using an algorithm for SAT.

In order to do this, we use the following approach from [1] (page 254): to show that $\Phi \vDash \varphi$, we show that $(\Phi \wedge \neg\varphi)$ is unsatisfiable. In this notation, we take Φ to be the conjunction of its formulas. If it turns out that $(\Phi \wedge \neg\varphi)$ is satisfiable, then $\Phi \nvDash \varphi$.

The first step in solving this task is to translate the problem into a SAT problem. The SAT algorithm only works if the formulas are expressed in conjunctive normal form (CNF), which we cannot assume is the case. Therefore, we need to convert $(\Phi \wedge \neg\varphi)$ to CNF in order to decide its satisfiability. A CNF consists of a conjunction of clauses, which are disjunctions of literals. Simple conversion rules described on pages 253 and 254 of [1] describe how to convert any logical formula into CNF. However, using this approach, the conversion could result in exponential sized CNF formulas. Therefore instead, we will translate the formula into an equisatisfiable formula in CNF (that means, a formula for which satisfiability is preserved)[1]. We do this in our process of translating by replacing each occurrence of some disjunction $(a \vee b)$ with $(a \vee n) \wedge (\neg n \vee b)$, where we introduce a new variable n. The resulting CNF formula will be referred to as Ψ.

The second and final step is then to pass Ψ through our SAT algorithm that will either (i) return a solution, in which case we conclude $\Phi \nvDash \varphi$; or (ii) return no solution, in which case we conclude $\Phi \vDash \varphi$.

Question 2

In this exercise, the setup is the same as in Question 1, except now we will use an algorithm for ASP to perform propositional reasoning.

In this case we must therefore translate our task into a logic program suitable for our ASP algorithm. We may do this with the help of a multitude of sources, describing the syntax and semantics of logic programs in ASP (for instance, see [2] from Chapter 2 onward). We can easily build our answer set program by using Ψ from Question 1: each clause c_j will have a rule "c_j :- 1{l_1, l_2, ...}", in which l_i are literals that contain atoms. We need to make sure each atom a will get a value by using "a ; -a." which assigns either true or false to a. Then a final rule combines all the clauses: "p :- c_1, c_2, ...". To make sure we only have an answer set if p is true we set the constraint: ":- not p". We call the resulting program P, in which these rules are described.

When we pass P through our ASP algorithm, it will tell whether there exists an answer set for this program. If this is the case, then $\Phi \nvDash \varphi$; otherwise $\Phi \vDash \varphi$.

Question 3

The provided default theory (W,D) is:

```
W = p & q
D = {
    q : r / r
    p & r : ~s & q / ~s & q
}
```

In order to come to a solution for this exercise, we will first find extensions of (W,D). We then encode this PDFN default theory into an answer set program P such that the answer sets of P are in direct correspondence to the extensions of (W,D). We then generalize the approach to arbitrary PDFN default theories.

[1] https://en.wikipedia.org/wiki/Equisatisfiability

According to Theorem 6.2 in [3], an extension is the deductive closure of W and a selection of consequents of the available defaults. Using this theorem we can come up with candidate extensions:

```
S1 = p & q
S2 = p & q & r
S3 = p & q & ∼s
S4 = p & q & r & ∼s
```

Subsequently, we can check each of these candidates using the Γ-operator, using Definition 6.2 in [3]. If $\Gamma(S) = S$, then S is an extension. For S_1 we can immediately see that we can draw more conclusions, knowing that p and q are true. Since q is true, and we have no information about r, we can conclude r (i.e. r should be in the extension). This already rules out S_1 and S_3 to be extensions of (W,D). For S_2, r is included, but now we run into a similar problem. Since we know p and r are true, and we have no information about s, we may conclude ∼s is true (i.e. ∼s should be in the extension). This leaves us with S_4, for which we only have to check if it is consistent with the defaults in (W,D). This means that for each default d, we check for each of its justifications B_i whether $S \nvdash \neg B_i$ holds (where $S = S_4$). If this is the case, then $\Gamma(S) = S$ and we call S an extension of (W,D). This is quickly done, and it turns out that S_4 is an extension of (W,D).

The following codeblock corresponds to the ASP program that has as only answer set S_4.

```
1.  % encode W
2.  p. q.
3.
4.  % these rules prevent some warnings and/or errors
5.  p :- p.   q :- q.   r :- r.   s :- s.
6.  -p :- -p. -q :- -q. -r :- -r. -s :- -s.
7.
8.  % encode defaults
9.  r :- q, not -r.
10. -s :- p, r, not s, not -q.  % split the conjunction in the conclusion
11. q :- p, r, not s, not -q.   % into separate rules
```

The answer set of this program is {p, q, r, -s}.

In order to generalize the approach, we will list the necessary steps to build the ASP program such that it works for arbitrary PDFN default theories.

First, each literal in W will be encoded as fact in our ASP program (line 2). Then, for each symbol s that occurs in the default theory we set up rules as in lines 5 and 6, where -s is the negative of s, i.e. ∼s. These rules are to avoid unsafe use of variables. Finally, we encode each default d as follows: the body in ASP consists of the pre-condition of d, followed by each of the justification's counterpart (for s this is -s, for ∼s this is s), preceded by the word "not". If the consequence of the rule is a conjunction of literals, these are split and used as multiple rules with the same body (lines 10 and 11).

The answer sets of this program should correspond to the extensions of any PDFN theory. This approach was tested on various examples in [3] and yielded correct extensions.

Now, if we are given a default theory (W,D) and want to decide whether there is an extension that includes a propositional literal l, we can simply iterate over the answer sets produced by our algorithm and check whether l is in one of them.

Question 4

The following two programs have the same answer sets as long as n, m and u are given the same value in both. An explanation is given below.

P_1:

```
1.  #const n=2.
2.  #const m=3.
3.  #const u=4.
4.
5.  n { item(1..u) } m.
```

P_2:

```
1.  #const n=2.
2.  #const m=3.
3.  #const u=4.
4.
5.  % define elements we will choose from
6.  element(1..u).
7.
8.  % first choice: choices 1 to n need to be different elements
9.  choice1(1..n).
10. choose1(C,E) :- not unchoose1(C,E), element(E), choice1(C).
11. unchoose1(C,E) :- not choose1(C,E), element(E), choice1(C),
12.                    choose1(C,F), element(F), F!=E.
13.
14. % second choice: choices n+1 to m can be any element
15. choice2(n+1..m).
16. choose2(C,E) :- not unchoose2(C,E), element(E), choice2(C).
17. unchoose2(C,E) :- not choose2(C,E), element(E), choice2(C).
18.
19. % combine choice1 and choice2
20. choice(1..u).
21. choose(C,E) :- choose1(C,E).
22. choose(C,E) :- choose2(C,E).
23.
24. % define constraints to filter output answer sets:
25. % each choice in the output must be different
26. :- choose(C,E), choose(D,E), C!=D.
27. % no single choice can contain multiple elements
28. :- choose(C,E), choose(C,V), E!=V.
29. % elements can only occur in increasing order
30. :- choose(C,E), choose(D,F), C<D, E>F.
31.
32. % there cannot be a "gap" in the choices
33. chosen(C) :- choose(C, X).
34. :- not chosen(C), chosen(D), D-1==C, choice(C).
35.
36. item(E) :- choose(C, E).
37. #show item/1.
```

Since n in P_1 means we want to choose at least n items from the range (1..u) and m means we want to choose at most m elements, we define two separate choices in P_2. The first one fills up the choices up till n such that each of the choice opportunities is used (for each choice 1..n we will choose at least one element). This reassures us that our output will be at least of length n. The second choice has more freedom: it can choose any element for each choice index, but it can also choose not to choose any element. This will result in some choices not being used, so out result can be of any length between n and m. Once we have made our choices, the results are combined in lines 20 to 22. Then we define constraints such that any invalid (lines 26 and 28), or double (lines 30, 33 and 34) answer sets will be filtered out. This together results in the program P_2 returning identical sets as P_1, while it is written in basic ASP language.

Question 5

To make the encoding into ASP easier, we first translate both φ_1 and φ_2 into their equisatisfiable CNF formulas ψ_1 and ψ_2, in the same way as in Question 1. So we now want to know if ψ_1 is satisfiable and ψ_2 is unsatisfiable. For ψ_1 we can set up a similar program P as in Question 2, resulting in an answer set only if ψ_1 (and therefore φ_1) is satisfiable. In this case we could add a flag (e.g. "psi1_sat") to our answer set to make analysis easier. Then we want to add to P, part of the program that checks whether is unsatisfiable (i.e. whether it holds for all assignments that ψ_2 is false). We can do this with what is called the saturation technique (see Section 6.3 of [4]). This technique is used to check whether all possible guesses satisfy a certain property, in our case that ψ_2 is false. First, we define the search space of guesses (for assignments in ψ_2) by disjunctive rules: each atom a will be either true or false, e.g. "a ; -a.". Then, we compute whether ψ_2 is true under the given assignment (using the approach from Question 2). If ψ_2 is not satisfied, we saturate the answer set (for this part of the program) to M_{sat}: we set a flag (e.g. "psi2_unsat") to true,

and we set each atom that occurs in ψ_2 to true. If ψ_2 is satisfied, i.e. ψ_2 is true, then the resulting answer set will be a strict subset of M_{sat} and will not contain the flag "psi2_unsat". Subsequently, we can run our program for two logical formulas φ_1 and φ_2, and we can conclude that φ_1 is satisfiable and φ_2 is unsatisfiable if and only if all answer sets contain the flag "psi2_unsat" and at least one contains "psi2_sat".

Using the disjunction in the head to define the search space makes it possible to effectively program the saturation technique into ASP. Without it, we would have to define choose functions for each atom assigning it a truth or false value, or both. So then we have to add contraints to make sure each atom gets exactly one assignment, resulting in inefficient use of code.

References

[1] Stuart J Russell and Peter Norvig. Artificial intelligence: a modern approach., 2016.

[2] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Answer set solving in practice. *Synthesis lectures on artificial intelligence and machine learning*, 6(3):1–238, 2012.

[3] Frank Van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of knowledge representation*. Elsevier, 2008.

[4] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web International Summer School*, pages 40–110. Springer, 2009.