DM552 Programming Languages
Part 2

Jeff Gyldenbrand
Username: jegyl16
E-mail: jegyl16@student.sdu.dk
Supervisor: Youcef Djenouri

January 2, 2018

Contents

1 Specification **3**
1.1 Project description 3

2 Design and Implementation **3**
2.1 The FIM problem 3
2.2 The SAT problem 3
2.3 Prolog programming language 3
2.4 Choice of problem 4

3 Data input and output **4**
3.1 Input section 4
3.2 Output section 5

4 SLD tree **6**

5 Conclusion **7**

6 Appendix **7**

1 Specification

1.1 Project description

This project was about choosing the right problem from two problems in relation to the Prolog programming language. One was the Frequent Itemset Mining problem (FIM) and the other, the boolean satisfiability problem (SAT). Then try Prolog in practice by solving the problem. Lastly to generate a SLD tree. In order to find the right problem, it was important to understand the two. In addition to making the solution, the choice of problem should also be explained thoroughly.

In order to determine which problem is correct in relation to Prolog, we first had to address the two possible problems. So we will explain the FIM and SAT problem, respectively, and then explain the Prolog programming language, and finally explain the choice of our problem, in the next section.

2 Design and Implementation

2.1 The FIM problem

In Frequent Itemset Mining problems, we want to extract frequent patterns among sets of items often in large databases. This could for example be sales data of a supermarket. Here we would like to know if customers buy beers and chips together whether they are likely to buy a third product too.

2.2 The SAT problem

In boolean satisfiability problems, we try to find out whether there exists an interpretation that satisfies a given boolean formula. This means we want to see if the variables in a given boolean formula can be replaced with true or false so that the whole formula can be evaluated to true. In that case, our problem is satisfiable, otherwise unsatisfiable.

2.3 Prolog programming language

Now we have a good idea of what the two problems FIM and SAT are, and must then examine what kind of programming language Prolog is before we can make a decision about which problem is most suitable.

To start with, Prolog is a general purpose language, meaning that it is designed to be used in general for a broad range of application areas. This does not affect the choice of FIM or SAT. But Prolog is based on proportional logic, which is a language that uses logical symbols such as: conjunction, disjunction and negation among others. But in relation to this project, it is enough to mention these three. In addition, Boolean variables are also handled by the language, usually used symbols are p, q, r or x, y, z, or any other symbol.

Prolog consists of three basic constructs. facts, rules and queries. Where facts are always true statements, rules are true or false depending on the situation. queries are just questions we can ask from our facts and rules. See code example below.

```
father(billy, bob).                    % fact: Billy is the father of Bob
father(billy, jane).                   % fact: Billy is also the father of Jane

parent(X,Y) :- father(X,Y).            % rule: X is parent of Y if X is father of Y

parent(billy, Y).                      % query: Lets ask who Billy is the parent of
```

2.4 Choice of problem

Although a solver for a FIM problem could easily be developed in the Prolog programming language, it would make more sense to use a functional programming language like Haskell for this purpose because pattern matching is a cornerstone of this programming language. It is very clear that the SAT problem is based on the same kind of formal logic as Prolog, and therefore its an natural choice to pick the SAT problem for this project. Not only would the code be much longer, but the time complexity of it would be worse, if we were to implement a solver for FIM in Prolog instead of Haskell, and vice versa. In the next section we will explore the SAT problem with Prolog.

3 Data input and output

3.1 Input section

It was allowed to use any module for this project. So instead of developing a solution our self, we make use of the CLPB[1] (Constraint Logic Programming over Boolean variables) library that can do the trick for us. An greater explanation of the CLPB solver will be elaborated in section 4.

First, we will test the CLPB module. We have made two data inputs. One that is SAT with 5 variables and 6 clauses:

$$F1 = (\neg x_1) \wedge (\neg x_2 \vee x_3 \vee x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_4 \vee x_3) \wedge (\neg x_3 \vee \neg x_5)$$

and one that is UNSAT:

$$F2 = (x_1) \wedge (\neg x_1)$$

Prolog can not read logical operators $\wedge$, $\vee$ and $\neg$, and must therefore be replaced, respectively, with $*$, $+$ and $\sim$ furthermore the boolean expression of true and false is, respectively, 1 and 0. We pass F1 as arguments in sat() which is a function in the clpb library that returns true if the given argument is an satisfiable boolean expression. To enumerate concrete solutions, we invoke labeling() on the variables. This assigns truth values to all our variables, and returns all constraints that are satisfied.

```
:- use_module(library(clpb)).

% F1
showSAT(X1,X2,X3,X4,X5) :-
  sat((~X1)*(~X2 + X3 + X1)*(~X1 + ~X2)*(X1 + X2 + ~X4)*(~X4 + X3)*(~X3 + ~X5)),
```

[1] Documentation for CLPB at URL: http://www.swi-prolog.org/pldoc/man?section=clpb

```
6   labeling([X1,X2,X3,X4,X5]).
7
8   % F2
9   showUNSAT(X,X) :-
10    sat((X)*(~X)),
11    labeling([X,X]).
```

3.2 Output section

As seen in the box below, we call showSat with the five variables. As output, we get five solutions that meet SAT. Each solution is separated by semicolon. This is not the nicest output, but it shows us the result.

Output of F1

```
?- showSAT(X1,X2,X3,X4,X5).
X1 = X2, X2 = X3, X3 = X4, X4 = X5, X5 = 0 ;
X1 = X2, X2 = X3, X3 = X4, X4 = 0,
X5 = 1 ;
X1 = X2, X2 = X4, X4 = X5, X5 = 0,
X3 = 1 ;
X1 = X4, X4 = X5, X5 = 0,
X2 = X3, X3 = 1 ;
X1 = X5, X5 = 0,
X2 = X3, X3 = X4, X4 = 1.
```

Let us double check to see if a given solution prompts a true. We try the last one: $(X1 = X5, X5 = 0, X2 = X3, X3 = X4, X4 = 1)$ by replacing them with boolean values: $(0, 1, 1, 1, 0)$ which will return a *true*

```
?- showSAT(0,1,1,1,0).
true.
```

And let us, for the sake of argument, try input boolean values that was not in the solution outputs of F1: $(1, 1, 1, 1, 0)$ which will return a $false$

```
?- showSAT(1,1,1,1,0).
false.
```

As seen in the next box, we call showUNSAT with only one variable and get the output false, because its unsatisfiable no matter which combination of boolean values we provide it.

Output of F2

```
?- showUNSAT(X,X).
false.
```

4 SLD tree

CLPB solver uses Binary Decision Diagram Algorithm[2] (BDD). The BDD algorithm is represented as a rooted, directed, acyclic graph. This means that the algorithm reads the first variable as the root, as seen in Fig:1. In our case, x_1 will be the root. Each child of root or nodes has two children containing the next variable. For example, root x_1 will have two children, both with the variable x_2 valued either true or false, and that will continue until we reach the last variable. Left edges is always assigned false value (or 0), and right edges assigned true (or 1). BDD reads to the left side of the tree first. That is, the first evaluation will be showSAT (0,0,0,0,0), then showSAT (0,0,0,0,1) ... to showSAT(1,1,1,1,1) according to our program. This will compute a total of 2^n evaluations where n = number of variables. In our case, $2^5 = 32$ evaluations. CLPB returns only cases where SAT is met. As seen in Fig:1 we get the five satisfiable solutions in green. Below is a tabular with these five output in order:

	x_1	x_2	x_3	x_4	x_5
1 =	(0,	0,	0,	0,	0),
2 =	(0,	0,	0,	0,	1),
3 =	(0,	0,	1,	0,	0),
4 =	(0,	1,	1,	0,	0),
5 =	(0,	1,	1,	1,	0)

if we compare our SLD tree with the output from showSAT(X1,X2,X3,X4,X5) in section 2.2 we can see that they match. If we where to implement the solver ourself, and not use the CLPB library, we could implement a solver that would evaluate the clauses instead of each variable like we do now, thus we would get a completely different SLD tree.

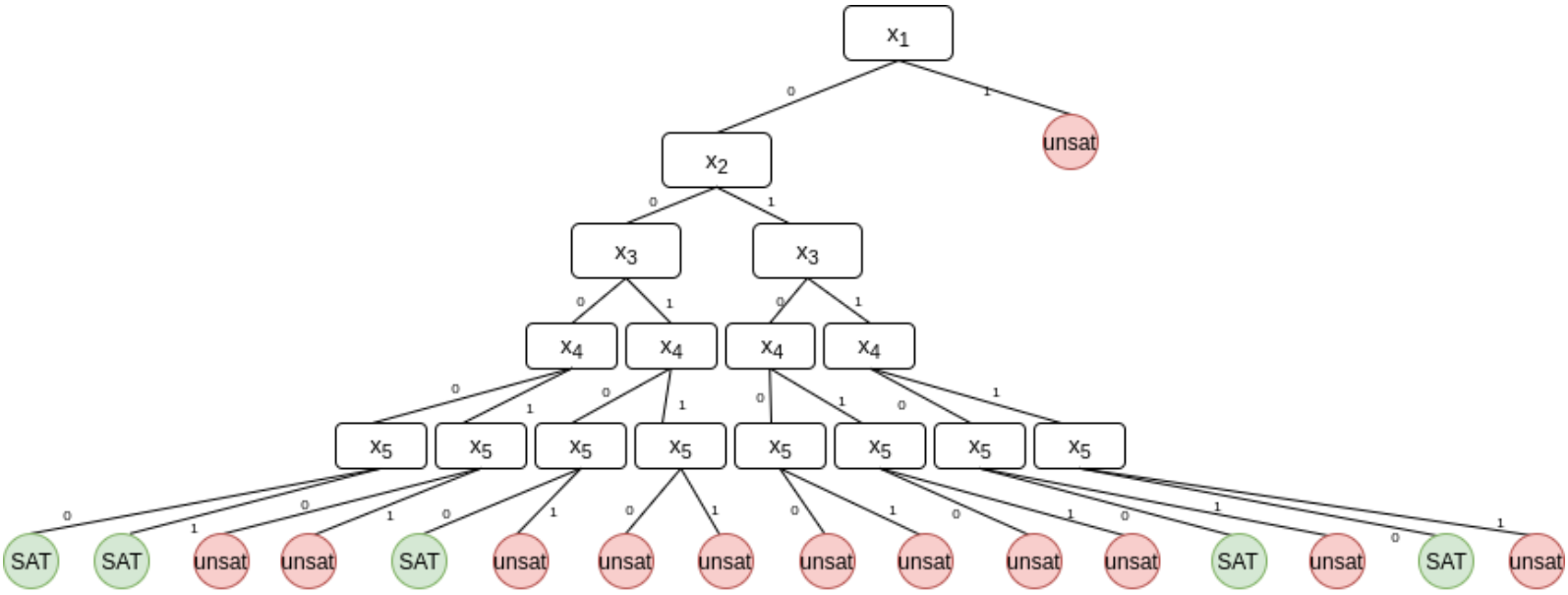

Figure 1: SLD tree of F1-input on CLPB-solver

[2]Paper on CLPB and the use of BDD, at URL: https://www.metalevel.at/swiclpb.pdf

5 Conclusion

We have succeeded in resonating to the correct problem by first describing both FIM, SAT and Prolog, where the choice fell on the SAT problem. We did not make our own SAT solver, but used a library that could solve the problem for us. In addition, we have used data input and received data output, which fulfills the requirement in the task description. Finally, a SLD tree is made based on the SAT solver and our data input. Every requirement is met.

6 Appendix

```
1  :- use_module(library(clpb)).
2
3  % F1
4  showSAT(X1,X2,X3,X4,X5) :-
5    sat((~X1)*(~X2 + X3 + X1)*(~X1 + ~X2)*(X1 + X2 + ~X4)*(~X4 + X3)*(~X3 + ~X5)),
6    labeling([X1,X2,X3,X4,X5]).
7
8  % F2
9  showUNSAT(X,X) :-
10   sat((X)*(~X)),
11   labeling([X,X]).
```