Seminar Report

Applications of Propositional Logic

Samuel Frontull
samuel.frontull@student.uibk.ac.at

24 February 2017

Supervisor: assoc. Prof. Dr. René Thiemann

Abstract

This report is mainly focused on showing techniques of modelling known problems with propositional logic and provides implementations in OCaml. The examples presented are implementations of Ramsey's Theorem, Primality Test and Sudoku Solving – all of them encoded as a tautology/satisfiability question. We have to keep in mind that the methods presented are not the most efficient for solving such problems and soon reach their limits. Nevertheless propositional logic is an interesting tool because it is really close to the human way of thinking and therefore easy to understand.

Contents

1 Introduction

This report is based on the theory discussed in *Handbook of Practical Logic and Automated Reasoning* [2] written by *John Harrison* (mainly chapter 2.7) and shows how to reduce known problems to tautology/satisfiability checking of propositional formulas. Section 2 shows how to prove Ramsey's Theorem by encoding this problem as a propositional formula which will then be tested whether it is a tautology. In Section 3 we will learn how to encode arithmetic operations as propositional formulas and use them to test if a number p is prime. Section 4 presents a way to solve a Sudoku-puzzle with propositional logic and using the SAT-Solver *MiniSAT*. Finally we will see how to use *MiniSAT* also for tautology checking of Ramsey's Theorem and Primality Test, which is much more effective than the naive `tautology` implementation in the source code [1] provided as accompaniment to the textbook [2].

The Sudoku example discussed in Section 4 is taken from the slides [3] of the lecture *Logic in Computer Science* hold by *Aart Midddeldorp* at the University of Innsbruck in 2016. The SAT-Solver `minisat` introduced in Section 4.2 is open-source and free for download[1]. A setup-tutorial of the environment used in Section 2 and 3 can be found in *appendix A*.

2 Ramsey's Theorem

As first example we take the combinatorial problem called *Ramsey's theorem* [2, p. 62]. Consider a group of 6 people. For every pair of them one property holds – either they know each other or they don't. This property can be visualized in an undirected graph with 6 nodes. There is an edge connecting two nodes if they know each other, there is no edge if they do not. An example of such a graph is shown in Figure 1. A simplified version of Ramsey's theorem now states that *in a graph of 6 nodes there's either a completely connected subgraph of size 3 or a completely disconnected subgraph of size 3. Theorem 2.1* extends this assumption to arbitrary integers.

Theorem 2.1 (Ramsey's Theorem [2])**.** *For each $s, t \in \mathbb{N}$ there is some $n \in \mathbb{N}$ such that any graph with n vertices either has a completely connected subgraph of size s or a completely disconnected subgraph of size t. Moreover if the 'Ramsey number' $R(s,t)$ (1) denotes the minimal n for a given s and t we have:*

$$R(s,t) \leq R(s-1,t) + R(s,t-1)^2. \tag{1}$$

Since $R(3,3) \leq 6$, we know that in the graph shown in Figure 1 we have either a completely connected subgraph of size 3 or a completely disconnected subgraph of size 3 by applying *Theorem 2.1.* The subgraph consisting of the nodes $2, 3, 5$ is an example of a completely disconnected subgraph of size 3.

[1] http://minisat.se/Main.html

[2] $R(s,t) = R(t,s); \quad R(n,1) = R(1,n) = 1; \quad R(n,2) = R(2,n) = n$

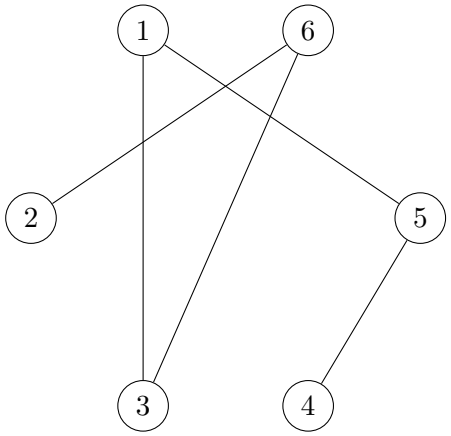

Figure 1: Undirected graph with 6 nodes

2.1 Ramsey's Theorem in Propositional Logic

It is easy to encode *Theorem 2.1* with given values for s, t and n as a propositional logic formula by using an atom for each possible relation between two nodes. p_{xy} $(= p_{yx})$ denotes that the nodes x and y are connected, $\neg p_{xy}$ that they are not.

We need a propositional formula that is satisfied if there is a completely connected or completely disconnected subgraph of size 3. In order to do that we have to model all possible subgraphs of size 3 where all nodes are connected/disconnected to each other. In our case we have

$$2 \cdot \frac{n!}{k! \cdot (n-k)!}^{3} = 2 \cdot \frac{6!}{3! \cdot (6-3)!} = 2 \cdot \frac{6 \cdot 5 \cdot 4}{3 \cdot 2 \cdot 1} = 2 \cdot \frac{5 \cdot 4}{1} = 2 \cdot 20 \tag{2}$$

possibilities. We have the possibilities of 20 completely connected and 20 completely disconnected subgraphs. An example for a completely connected subgraph would be

$$p_{13} \wedge p_{14} \wedge p_{34} \tag{3}$$

and for a completely disconnected one

$$\neg p_{45} \wedge \neg p_{56} \wedge \neg p_{46} \tag{4}$$

as shown in *Figure 2*. Our resulting formulas for all subgraphs of size 3 are shown in equation 5 (completely connected subgraphs) and in equation 6 (completely disconnected subgraphs).

[3] $\frac{n!}{k! \cdot (n-k)!}$ is a formula for sampling without replacement and without permutations

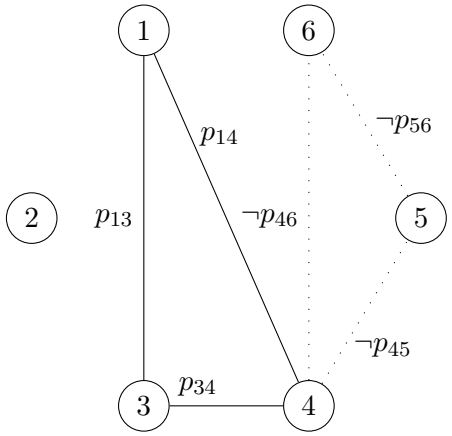

Figure 2: Graph showing the meaning of atom p_{xy}

$$\begin{aligned}\phi =&(p_{12} \wedge p_{13} \wedge p_{23}) \vee (p_{12} \wedge p_{14} \wedge p_{24}) \vee (p_{12} \wedge p_{15} \wedge p_{25}) \vee (p_{12} \wedge p_{16} \wedge p_{26}) \vee \\ &(p_{13} \wedge p_{14} \wedge p_{34}) \vee (p_{13} \wedge p_{15} \wedge p_{35}) \vee (p_{13} \wedge p_{16} \wedge p_{36}) \vee (p_{14} \wedge p_{15} \wedge p_{45}) \vee \\ &(p_{14} \wedge p_{16} \wedge p_{46}) \vee (p_{15} \wedge p_{16} \wedge p_{56}) \vee (p_{23} \wedge p_{24} \wedge p_{34}) \vee (p_{23} \wedge p_{25} \wedge p_{35}) \vee \\ &(p_{23} \wedge p_{26} \wedge p_{36}) \vee (p_{24} \wedge p_{25} \wedge p_{45}) \vee (p_{24} \wedge p_{26} \wedge p_{46}) \vee (p_{25} \wedge p_{26} \wedge p_{56}) \vee \\ &(p_{34} \wedge p_{35} \wedge p_{45}) \vee (p_{34} \wedge p_{36} \wedge p_{46}) \vee (p_{35} \wedge p_{36} \wedge p_{56}) \vee (p_{45} \wedge p_{46} \wedge p_{56})\end{aligned} \tag{5}$$

$$\begin{aligned}\psi =&(\neg p_{12} \wedge \neg p_{13} \wedge \neg p_{23}) \vee (\neg p_{12} \wedge \neg p_{14} \wedge \neg p_{24}) \vee (\neg p_{12} \wedge \neg p_{15} \wedge \neg p_{25}) \vee \\ &(\neg p_{12} \wedge \neg p_{16} \wedge \neg p_{26}) \vee (\neg p_{13} \wedge \neg p_{14} \wedge \neg p_{34}) \vee (\neg p_{13} \wedge \neg p_{15} \wedge \neg p_{35}) \vee \\ &(\neg p_{13} \wedge \neg p_{16} \wedge \neg p_{36}) \vee (\neg p_{14} \wedge \neg p_{15} \wedge \neg p_{45}) \vee (\neg p_{14} \wedge \neg p_{16} \wedge \neg p_{46}) \vee \\ &(\neg p_{15} \wedge \neg p_{16} \wedge \neg p_{56}) \vee (\neg p_{23} \wedge \neg p_{24} \wedge \neg p_{34}) \vee (\neg p_{23} \wedge \neg p_{25} \wedge \neg p_{35}) \vee \\ &(\neg p_{23} \wedge \neg p_{26} \wedge \neg p_{36}) \vee (\neg p_{24} \wedge \neg p_{25} \wedge \neg p_{45}) \vee (\neg p_{24} \wedge \neg p_{26} \wedge \neg p_{46}) \vee \\ &(\neg p_{25} \wedge \neg p_{26} \wedge \neg p_{56}) \vee (\neg p_{34} \wedge \neg p_{35} \wedge \neg p_{45}) \vee (\neg p_{34} \wedge \neg p_{36} \wedge \neg p_{46}) \vee \\ &(\neg p_{35} \wedge \neg p_{36} \wedge \neg p_{56}) \vee (\neg p_{45} \wedge \neg p_{46} \wedge \neg p_{56})\end{aligned} \tag{6}$$

Now we have all we need. If we now put ϕ and ψ together by a disjunction we have a formula that is a tautology if and only if Ramsey's theorem holds for $s = 3$, $t = 3$ and $n = 6$.

$$\phi \vee \psi \text{ a tautology } \iff R(3,3) \leq 6.$$

2.2 Implementation in OCaml

The function in Listing 1 generates the formula $\phi \vee \psi$ explained above which is a tautology if and only if Ramsey's Theorem holds for the given parameters s, t and n. An output example of this function is shown in Listing 2.

```
let ramsey s t n =
  let vertices = 1 -- n in
  let yesgrps = map (allsets 2) (allsets s vertices)
  and nogrps = map (allsets 2) (allsets t vertices) in
  let e[m;n] = Atom(P("p_"^(string_of_int m)^"_"^(string_of_int n))) in
  Or(list_disj (map (list_conj ** map e) yesgrps),
    list_disj (map (list_conj ** map (fun p -> Not(e p))) nogrps));;
```

Listing 1: Function that generates the formula $\phi \vee \psi$

```
# ramsey 3 3 4;;
- : prop formula =
<<(p_1_2 /\ p_1_3 /\ p_2_3 \/
   p_1_2 /\ p_1_4 /\ p_2_4 \/
   p_1_3 /\ p_1_4 /\ p_3_4 \/ p_2_3 /\ p_2_4 /\ p_3_4) \/
  ~p_1_2 /\ ~p_1_3 /\ ~p_2_3 \/
  ~p_1_2 /\ ~p_1_4 /\ ~p_2_4 \/
  ~p_1_3 /\ ~p_1_4 /\ ~p_3_4 \/ ~p_2_3 /\ ~p_2_4 /\ ~p_3_4>>
```

Listing 2: Example output of ramsey function

```
# tautology(ramsey 3 3 5);;
- : bool = false
# tautology(ramsey 3 3 6);;
- : bool = true
```

Listing 3: Tautology checking of a ramsey-formula

The latter example already takes an appreciable time, even slightly larger input numbers can create propositional problems way beyond those that can be solved in reasonable time by the method described [2].

3 Primality test

In this section we will learn how to encode arithmetic operations as propositional formulas that we can use to test if a number p is prime. First we will see how to encode addition of n-bit numbers. We can then multiply n-bit numbers by repeated addition. With this arithmetic operations we are able to generate a formula that is a tautology, precisely if a number p is prime.

3.1 Addition

In computer science we use binary representations of numbers. The usual algorithms for arithmetic on many-digit numbers that we learn in school can be straightforwardly modified for the binary notation. We add from right to left and generate a *carry* of 1 if the sum of the two bits is ≥ 2.

$$\begin{array}{rr} & 1\,0\,1\,1\,0\,0\,1\,1 \\ + & 0\,1\,1\,0\,0\,1\,0\,1 \\ \hline = & 1\,0\,0\,0\,1\,1\,0\,0\,0 \end{array}$$

We have to generate a carry bit if both bits are one which leads us to the formula $x \wedge y$ – this formula corresponds to a *half-carry*. For the *sum-bit* we write 1 if x and y are not equal $x \iff \neg y$ (*half-sum*). If we add two n-bit numbers it can be the case that we have to add three bits if the previous addition generated a carry bit. A *full-adder* takes consideration of this additional carry bit c. We can derive the formulas 7 and 8 from the truth table of a *full-adder*. Listing 4 shows the OCaml code of a *full-adder*.

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$c_{xyz} = x \wedge y \vee (x \vee y) \wedge z \tag{7}$$

$$s_{xyz} = (x \iff \neg y) \iff \neg z \tag{8}$$

```
let halfsum x y = Iff(x,Not y);;

let carry x y z = Or(And(x,y),And(Or(x,y),z));;

let sum x y z = halfsum (halfsum x y) z;;

let fa x y z s c = And(Iff(s,sum x y z),Iff(c,carry x y z));;
```

Listing 4: Full-adder in OCaml

It is easy to extend a *one-bit adder* to an *n-bit adder* by using multiple *full-adders*.

$$\begin{array}{rr} & c_2\ c_1\ c_0 \\ & x_1\ x_0 \\ + & y_1\ y_0 \\ \hline = & c_2\ s_1\ s_0 \end{array}$$

Listing 5 shows the OCaml code that implements the *n-bit adder*. Listing 6 is an optimization of `ripplecarry`, c_0 is dropped here. In Listing 7 we see the output of the `ripplecarry0` function – this formula corresponds to those shown on in (9) and (10) except that the unnecessary bit c_0 is dropped. The resulting formula we see in Listing 7 is satisfied by all number combinations where the *output bits* represents the sum of the numbers given.

$$s_0 = (x_0 \iff \neg y_0) \iff \neg c_0 \qquad c_1 = x_0 \wedge y_0 \vee (x_0 \vee y_0) \wedge c_0 \tag{9}$$

$$s_1 = (x_1 \iff \neg y_1) \iff \neg c_1 \qquad c_2 = x_1 \wedge y_1 \vee (x_1 \vee y_1) \wedge c_1 \tag{10}$$

Table 1: Multiplication of two 3-bit numbers

				A_0B_2	A_0B_1	A_0B_0
+			A_1B_2	A_1B_1	A_1B_0	
+		A_2B_2	A_2B_1	A_2B_0		
=	P_5	P_4	P_3	P_2	P_1	P_0

```
let list_conj l = if l = [] then True else end_itlist mk_and l;;

let conjoin f l = list_conj (map f l);;

let [x; y; out; c] = map mk_index ["X"; "Y"; "OUT"; "C"];;

let ripplecarry x y c out n =
  conjoin (fun i -> fa (x i) (y i) (c i) (out i) (c(i + 1)))
          (0 -- (n - 1));;
```

Listing 5: Ripple-carry in OCaml

```
let ripplecarry0 x y c out n =
  psimplify
   (ripplecarry x y (fun i -> if i = 0 then False else c i) out n);;
```

Listing 6: Optimization of Ripple-carry in OCaml

```
# ripplecarry0 x y c out 2;;
- : prop formula =
<<((OUT_0 <=> X_0 <=> ~Y_0) /\ (C_1 <=> X_0 /\ Y_0)) /\
  (OUT_1 <=> (X_1 <=> ~Y_1) <=> ~C_1) /\
  (C_2 <=> X_1 /\ Y_1 \/ (X_1 \/ Y_1) /\ C_1)>>
```

Listing 7: Example output of a 2-bit adder

3.2 Multiplication

Now that we can add *n-bit* numbers we are not far away from multiplication. Multiplication is basically only repeated addition and that is what we do with propositional logic. Table 1 shows an example of a multiplication of two *3-bit* numbers. Again, we use the same technique we learned in school. We see that two bits can simply be multiplied with a conjunction A_iB_j, the rest is repeated addition, where the rightmost bit is fixed.

Now that we know the pattern, we can generate a formula which is satisfied by all number combinations where the output is the product of the input numbers A and B. We write X_{ij} as abbreviation of the product $A_i \wedge B_j$.

$$P_0 \iff X_{00} \tag{11}$$

$$P_1 \iff X_{01} \iff \neg(X_{10}) \tag{12}$$

$$v_{22} \iff X_{01} \wedge X_{10} \tag{13}$$

$$u_{20} \iff (X_{02} \iff \neg X_{11}) \iff \neg v_{22} \tag{14}$$

$$v_{23} \iff X_{02} \wedge X_{11} \vee (X_{02} \vee X_{11}) \wedge v_{22} \tag{15}$$

$$u_{21} \iff X_{12} \iff \neg v_{23} \tag{16}$$

$$u_{22} \iff X_{12} \wedge v_{23} \tag{17}$$

$$P_2 \iff u_{20} \iff \neg X_{20} \tag{18}$$

$$v_{32} \iff u_{20} \wedge X_{20} \tag{19}$$

$$P_3 \iff (u_{21} \iff \neg X_{21}) \iff \neg v_{32}) \tag{20}$$

$$v_{33} \iff u_{21} \wedge X_{21} \vee (u_{21} \vee X_{21}) \wedge v_{32} \tag{21}$$

$$P_4 \iff (u_{22} \iff \neg X_{22}) \iff \neg v_{33} \tag{22}$$

$$P_5 \iff u_{22} \wedge X_{22} \vee (u_{22} \vee X_{22}) \wedge v_{33} \tag{23}$$

If we conjunct those sub-formulas we have a general formula for the multiplication of two *3-bit* numbers. In Listing 8 we see the OCaml code of the `multiplier` function that generates this formula. Parameter `n` of the `multiplier` function is the number of bits used to encode both factors. The `rippleshift` function is an extension of the `ripplecarry0` where we separate the rightmost bit z from the others.

```
let rippleshift u v c z w n =
  ripplecarry0 u v (fun i -> if i = n then w(n - 1) else c(i + 1))
                   (fun i -> if i = 0 then z else w(i - 1)) n;;

let multiplier x u v out n =
  if n = 1 then And(Iff(out 0,x 0 0),Not(out 1)) else
  psimplify
   (And(Iff(out 0,x 0 0),
        And(rippleshift
              (fun i -> if i = n - 1 then False else x 0 (i + 1))
              (x 1) (v 2) (out 1) (u 2) n,
            if n = 2 then And(Iff(out 2,u 2 0),Iff(out 3,u 2 1)) else
            conjoin (fun k -> rippleshift (u k) (x k) (v(k + 1)) (out k)
                                (if k = n - 1 then fun i -> out(n + i)
                                 else u(k + 1)) n) (2 -- (n - 1))))));;
```

Listing 8: Multiplier formula generator in OCaml

```
# multiplier m u v out 3;;
- : prop formula =
<<(out_0 <=> x_0 /\ y_0) /\
  (((out_1 <=> x_0 /\ y_1 <=> ~(x_1 /\ y_0)) /\
    (v_2_2 <=> (x_0 /\ y_1) /\ x_1 /\ y_0)) /\
   ((u_2_0 <=> (x_0 /\ y_2 <=> ~(x_1 /\ y_1)) <=> ~v_2_2) /\
    (v_2_3 <=>
     (x_0 /\ y_2) /\ x_1 /\ y_1 \/ (x_0 /\ y_2 \/ x_1 /\ y_1) /\ v_2_2)) /\
   (u_2_1 <=> x_1 /\ y_2 <=> ~v_2_3) /\
```

```
  (u_2_2 <=> (x_1 /\ y_2) /\ v_2_3)) /\
 ((out_2 <=> u_2_0 <=> ~(x_2 /\ y_0)) /\
  (v_3_2 <=> u_2_0 /\ x_2 /\ y_0)) /\
 ((out_3 <=> (u_2_1 <=> ~(x_2 /\ y_1)) <=> ~v_3_2) /\
  (v_3_3 <=> u_2_1 /\ x_2 /\ y_1 \/ (u_2_1 \/ x_2 /\ y_1) /\ v_3_2)) /\
 (out_4 <=> (u_2_2 <=> ~(x_2 /\ y_2)) <=> ~v_3_3) /\
 (out_5 <=> u_2_2 /\ x_2 /\ y_2 \/ (u_2_2 \/ x_2 /\ y_2) /\ v_3_3)>>
```

Listing 9: Example output of multiplier function for 3-bit numbers

3.3 Primality as tautology question

We can now encode the question if a specific integer $p > 1$ is prime as tautology question. If a number p is composite and requires n bits to store, it must have factorization with both factors at least 2, hence both $\leq p/2$ and so storable in $n-1$ bits [2]. That's why we call the `multiplier` function with `(n-1)` as last parameter (`n` is the number of bits needed to encode p). By searching only for factors storable in $n-1$ bits we automatically enforce that both factors are > 1 – if one of the factors would be equal to 1 the other one would have to be equal to p which cannot be encoded by using only $n-1$ bits. To assert that p is prime we need to state that for every two $(n-1)$-element sequences of bits their product does not correspond to p.

```
let rec bitlength x = if x = 0 then 0 else 1 + bitlength (x / 2);;

let rec bit n x = if n = 0 then x mod 2 = 1 else bit (n - 1) (x / 2);;

let congruent_to x m n =
  conjoin (fun i -> if bit i m then x i else Not(x i))
          (0 -- (n - 1));;

let prime p =
  let [x; y; out] = map mk_index ["x"; "y"; "out"] in
  let m i j = And(x i,y j)
  and [u; v] = map mk_index2 ["u"; "v"] in
  let n = bitlength p in
  Not(And(multiplier m u v out (n - 1),
      congruent_to out p (max n (2 * n - 2))));;
```

Listing 10: Multiplier formula generator in OCaml

In Listing 11 we can see in the output formula how we can fix the product to a specific number – it's done by conjunction of the binary representation of the number. In this case we conjunct $out\_0 \wedge out\_1 \wedge out\_2 \wedge \neg out\_3$ which corresponds to the binary representation of number 7 (0111). If we are are not able to find a combinations of input numbers that satisfies the formula we know that p is prime – this is the case if the negated formula $(\neg p)$ is a tautology [2, p. 39]. That's why we negate the whole formula. Listing 11 shows call examples where we check if the generated formula is a tautology i.e. the number is prime.

```
# tautology(prime 7);;
- : bool = true
# tautology(prime 11);;
- : bool = true
# tautology(prime 15);;
- : bool = false
```

Listing 11: Example output of prime function tautology check

Here we have the same problem as already seen in Section 2 – a slightly larger input parameter can create a huge problem which cannot be solved in reasonable time by using this method. The `tautology` function can in the worst case take a time exponential in the size of the input formula, since it may need to evaluate the formula on all 2^n valuations of its n atomic propositions [2]. Finding polynomial-time algorithms for such *NP-complete* problems is the outstanding open question in computer science.

4 Sudoku solving

Sudoku is probably the most famous logical puzzle in the world. Solving Sudoku puzzles can be very challenging – but not for us. In this section we will see how to encode a Sudoku as a propositional formula. Satisfiable assignments for this formula gives us a possible solution. The aim is to enter a digit from 1 through 9 in each cell of the grid so that each row, column and region contains only one instance of each digit [5], this means that we have to express the following constraints as a formula:

- There is at least one number in each cell
- There is at most one number in each cell
- A number appears at most once in each row
- A number appears at most once in each column
- A number appears at most once in each $3x3$ sub-grid

4.1 Sudoku in Propositional Logic

We have already seen that we can express with a propositional formula all the conditions for which a problem holds. The same can be done for the Sudoku-problem where we express the rules and the initial state (facts) as a formula. The propositional atom x_{ijd} means that cell at row i and column j contains digit d (shown in Figure 4). In this section we call the set of all possible digits D.

$$D = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \tag{24}$$

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Figure 3: Example Sudoku

x_{11d}	x_{12d}	x_{13d}	x_{14d}	x_{15d}	x_{16d}	x_{17d}	x_{18d}	x_{19d}
x_{21d}	x_{22d}	x_{23d}	x_{24d}	x_{25d}	x_{26d}	x_{27d}	x_{28d}	x_{29d}
x_{31d}	x_{32d}	x_{33d}	x_{34d}	x_{35d}	x_{36d}	x_{37d}	x_{38d}	x_{39d}
x_{41d}	x_{42d}	x_{43d}	x_{44d}	x_{45d}	x_{46d}	x_{47d}	x_{48d}	x_{49d}
x_{51d}	x_{52d}	x_{53d}	x_{54d}	x_{55d}	x_{56d}	x_{57d}	x_{58d}	x_{59d}
x_{61d}	x_{62d}	x_{63d}	x_{64d}	x_{65d}	x_{66d}	x_{67d}	x_{68d}	x_{69d}
x_{71d}	x_{72d}	x_{73d}	x_{74d}	x_{75d}	x_{76d}	x_{77d}	x_{78d}	x_{79d}
x_{81d}	x_{82d}	x_{83d}	x_{84d}	x_{85d}	x_{86d}	x_{87d}	x_{88d}	x_{89d}
x_{91d}	x_{92d}	x_{93d}	x_{94d}	x_{95d}	x_{96d}	x_{97d}	x_{98d}	x_{99d}

Figure 4: Cell labeling

4.1.1 Model the rules

If we want to express that there is *at least* one number in *one* cell we encode:

$$\phi_{ij} = (x_{ij1} \vee x_{ij2} \vee x_{ij3} \vee x_{ij4} \vee x_{ij5} \vee x_{ij6} \vee x_{ij7} \vee x_{ij8} \vee x_{ij9}) \tag{25}$$

To express that there is *at least* one number in *each* cell we conjunct all ϕ_{ij}

$$\phi = \bigwedge_{i,j \in D} \phi_{ij} \tag{26}$$

$$\phi = \phi_{11} \wedge \phi_{12} \wedge \phi_{13} \wedge \phi_{14} \wedge \cdots \wedge \phi_{98} \wedge \phi_{99} \tag{27}$$

In order to say that there is *at most* one digit in each cell (second constraint) we have to exclude that for every possible digit pair x_{ijd} and x_{ijk} (where $d \neq k$) are both true for the same cell.

$$\psi_{ij} = \bigwedge_{\substack{d,k \in D \\ d \neq k}} \neg(x_{ijd} \wedge x_{ijk}) \tag{28}$$

$$\begin{aligned}\psi_{ij} =& \neg(x_{ij1} \wedge x_{ij2}) \wedge \neg(x_{ij1} \wedge x_{ij3}) \wedge \neg(x_{ij1} \wedge x_{ij4}) \wedge \neg(x_{ij1} \wedge x_{ij5}) \wedge \neg(x_{ij1} \wedge x_{ij6}) \wedge \\ & \neg(x_{ij1} \wedge x_{ij7}) \wedge \neg(x_{ij1} \wedge x_{ij8}) \wedge \neg(x_{ij1} \wedge x_{ij9}) \wedge \neg(x_{ij2} \wedge x_{ij3}) \wedge \neg(x_{ij2} \wedge x_{ij4}) \wedge \\ & \neg(x_{ij2} \wedge x_{ij5}) \wedge \neg(x_{ij2} \wedge x_{ij6}) \wedge \neg(x_{ij2} \wedge x_{ij7}) \wedge \neg(x_{ij2} \wedge x_{ij8}) \wedge \neg(x_{ij2} \wedge x_{ij9}) \wedge \\ & \neg(x_{ij3} \wedge x_{ij4}) \wedge \neg(x_{ij3} \wedge x_{ij5}) \wedge \neg(x_{ij3} \wedge x_{ij6}) \wedge \neg(x_{ij3} \wedge x_{ij7}) \wedge \neg(x_{ij3} \wedge x_{ij8}) \wedge \\ & \neg(x_{ij3} \wedge x_{ij9}) \wedge \neg(x_{ij4} \wedge x_{ij5}) \wedge \neg(x_{ij4} \wedge x_{ij6}) \wedge \neg(x_{ij4} \wedge x_{ij7}) \wedge \neg(x_{ij4} \wedge x_{ij8}) \wedge \\ & \neg(x_{ij4} \wedge x_{ij9}) \wedge \neg(x_{ij5} \wedge x_{ij6}) \wedge \neg(x_{ij5} \wedge x_{ij7}) \wedge \neg(x_{ij5} \wedge x_{ij8}) \wedge \neg(x_{ij5} \wedge x_{ij9}) \wedge \\ & \neg(x_{ij6} \wedge x_{ij7}) \wedge \neg(x_{ij6} \wedge x_{ij8}) \wedge \neg(x_{ij6} \wedge x_{ij9}) \wedge \neg(x_{ij7} \wedge x_{ij8}) \wedge \neg(x_{ij7} \wedge x_{ij9}) \wedge \\ & \neg(x_{ij8} \wedge x_{ij9})\end{aligned} \tag{29}$$

To express that there is *at most* one number in *each* cell we conjunct all ψ_{ij}

$$\psi = \bigwedge_{i,j \in D} \psi_{ij} \tag{30}$$

For the third constraint, a number appears at most once in each row, we iterate over each row with same digit. We have the same pattern as in constraint 2 – all row pair combinations have to be excluded to be true at the same time. ω expresses that a digit d appears at most once in each row:

$$\omega_{id} = \bigwedge_{j \in D} \bigwedge_{k=j+1}^{9} \neg(x_{ijd} \wedge x_{ikd}) \tag{31}$$

$$\omega = \bigwedge_{i,d \in D} \omega_{id} \tag{32}$$

Constraint 4, a number appears at most once in each column, is the same as constraint 3 except that this time we iterate over the columns. π expresses that a digit d appears at most once in each column:

$$\pi_{jd} = \bigwedge_{i \in D} \bigwedge_{k=i+1}^{9} \neg(x_{ijd} \wedge x_{kjd}) \tag{33}$$

$$\pi = \bigwedge_{j,d \in D} \omega_{jd} \tag{34}$$

Also for the last rule, a number appears at most once in each $3x3$ sub-grid, we use the *"at-most"-pattern.*

$$G = \{\{1,2,3\},\{4,5,6\},\{7,8,9\}\} \tag{35}$$

$$\lambda_d = \bigwedge_{\substack{(i,j),(k,l) \in I \times J \\ I,J \in G}} \neg(x_{ijd} \wedge x_{kld}) \tag{36}$$

$$\lambda = \bigwedge_{d \in D} \lambda_d \tag{37}$$

4.1.2 Model the facts

The initial setup of the Sudoku shown in figure 3 represented as propositional formula is:

$$\begin{aligned} \theta = & x_{122} \wedge x_{145} \wedge x_{161} \wedge x_{189} \wedge x_{218} \wedge x_{242} \wedge x_{263} \wedge x_{296} \wedge x_{323} \wedge x_{356} \wedge x_{387} \wedge \\ & x_{431} \wedge x_{476} \wedge x_{515} \wedge x_{524} \wedge x_{581} \wedge x_{599} \wedge x_{632} \wedge x_{677} \wedge x_{729} \wedge x_{753} \wedge x_{788} \wedge \\ & x_{812} \wedge x_{848} \wedge x_{864} \wedge x_{897} \wedge x_{921} \wedge x_{949} \wedge x_{967} \wedge x_{986} \end{aligned} \tag{38}$$

There's nothing more to do as conjunct this formula θ with the formulas $\phi, \psi, \omega, \pi, \lambda$ representing the 5 Sudoku-constraints and we have a resulting formula (39) that is

satisfiable if and only if the Sudoku has a solution. A solution for the Sudoku can be read off from the satisfying assignment returned by the SAT solver [5].

$$\theta \wedge \phi \wedge \psi \wedge \omega \wedge \pi \wedge \lambda \tag{39}$$

4.2 SAT Solving

The problems presented in Section 2 and 3 were both encoded as a tautology question. To solve the Sudoku problem it is sufficient to find one satisfying assignment for the formula shown in equation 39. The OCaml code written by *John Harrison* used in Section 2 and 3 is not intended to be efficient and isn't likely to be much use for serious applications [1]. In fact, we would not be able to solve the Sudoku problem in a reasonable time with the given implementation of `satisfiable`. As we can see in Listing 12 the `satisfiable` function is implemented using the `unsatisfiable` function which first checks if the formula is a tautology. For the Sudoku-formula this would mean to make 2^{729} evaluations (one atom for each digit in each cell).

```
let unsatisfiable fm = tautology(Not fm);;

let satisfiable fm = not(unsatisfiable fm);;
```

Listing 12: Implementation of the *satisfiable* function

We need a more efficient procedure to check the satisfiability of the *Sudoku-formula*. That's why we use the open-source SAT solver `minisat`[4]. *MiniSAT* requires a specific input format of the propositional formula given as *CNF*. The OCaml code[5] that generates the formula shown in equation 39 in the required format for the *MiniSAT* solver can be found in appendix C. Listing 13 shows an example output of a `minisat` execution that tries to find a satisfying assignment for a Sudoku formula – we see that `minisat` only needs *0.012s* to find a solution.

```
$ minisat sudoku.txt sudoku_solution.txt
============================[ Problem Statistics ]=============================
|  Number of variables:           999                                         |
|  Number of clauses:           11745                                         |
|  Parse time:                   0.01 s                                       |
|  Eliminated clauses:           0.00 Mb                                      |
|  Simplification time:          0.00 s                                       |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
==============================================================================
restarts              : 1
conflicts             : 8              (667 /sec)
decisions             : 17             (0.00 % random) (1417 /sec)
propagations          : 881            (73417 /sec)
conflict literals     : 29             (6.45 % deleted)
```

[4] http://minisat.se/Main.html
[5] http://cl-informatik.uibk.ac.at/teaching/ws16/lics/material/sudoku.ml

```
Memory used          : 22.00 MB
CPU time             : 0.012 s

SATISFIABLE
```

Listing 13: Example output of minisat call for Sudoku formula

The solution is printed in a separate output-file given as parameter. Listing 14 shows a detail of the solution for the Sudoku shown in Figure 3. Here we see that in cell at *(3,1)* we have digit *1*.

```
-310 311 -312 -313 -314 -315 -316 -317 -318 -319
```

Listing 14: Detail of Sudoku solution

5 Conclusion

This report has given a taste of how certain problems can be reduced to tautology/satisfiability checking of propositional formulas [2]. We also have seen that apparently small problems can produce large formulas. With the naive implementation of `tautology` it cannot be determined in reasonable time if such a formula is a tautology or not. In Section 4 we used the SAT-Solver *MiniSAT* which is highly efficient and solved the Sudoku-puzzle in $0.012s$. It was already mentioned that ϕ is a tautology if and only if $\neg\phi$ is unsatisfiable [2, p. 39]. This means that we can use *MiniSAT* also for our tautology problems described in Sections 2 and 3. We can negate and transform the formula we get as output of `prime` and `ramsey` into an *equisatisfiable* CNF formula by executing `defcnf` (based on *Tseitin's transformation* [4]) as shown in Listing 15.

```
# defcnf(negate(ramsey 4 4 20));;

# defcnf(negate(prime 19));;
```

Listing 15: Output of minisat call for Sudoku formula

We can use the transformed formulas as input of the `minisat` program – if we get *UNSATISFIABLE* as result we know that the original formula is a tautology. As we can see in Listing 16 and 17 we can solve the tautology question for Ramsey's Theorem for $s = 3$, $t = 4$ and $n = 10$ and the Primality Test for number 19 both in less than $0.2s$. This is a big improvement but we cannot forget that also the best SAT-Solver has exponential worst-case complexity. A polynomial-time algorithm for SAT or any other NP-complete problem would give rise to a polynomial-time algorithm for *all* NP-complete problems [2, p. 72].

```
$ minisat prime-19.txt
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:          6733                                         |
|  Number of clauses:             328                                         |
|  Parse time:                   0.00 s                                       |
```

```
|  Eliminated clauses:        0.05 Mb                                    |
|  Simplification time:       0.00 s                                     |
=================================================================================

restarts              : 1
conflicts             : 6              (750 /sec)
decisions             : 6              (0.00 % random) (750 /sec)
propagations          : 110            (13750 /sec)
conflict literals     : 12             (29.41 % deleted)
Memory used           : 23.00 MB
CPU time              : 0.008 s

UNSATISFIABLE
```

Listing 16: Output of minisat call for equisatisfiable CNF-formula of prime 19

```
$ minisat ramsey_3410.txt
============================[ Problem Statistics ]===============================
|                                                                              |
|  Number of variables:          910                                           |
|  Number of clauses:            330                                           |
|  Parse time:                   0.00 s                                        |
|  Eliminated clauses:           0.01 Mb                                       |
|  Simplification time:          0.00 s                                        |
|                                                                              |
============================[ Search Statistics ]================================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
==========================================================================
|       100 |      45      330     1620 |      121      100      9 |  0.000 % |
|       250 |      45      330     1620 |      133      116     10 |  0.000 % |
|       475 |      45      330     1620 |      146      127     10 |  0.000 % |
|       812 |      45      330     1620 |      161      152      9 |  0.000 % |
|      1318 |      45      330     1620 |      177      144      8 |  0.000 % |
|      2077 |      45      330     1620 |      194      150      9 |  0.000 % |
|      3216 |      45      330     1620 |      214      161      8 |  0.000 % |
|      4924 |      45      330     1620 |      235      180      9 |  0.000 % |
|      7486 |      45      330     1620 |      259      173      8 |  0.000 % |
|     11330 |      45      330     1620 |      285      248      8 |  0.000 % |
|     17096 |      45      330     1620 |      313      265      8 |  0.000 % |
|     25745 |      45      330     1620 |      345      226      7 |  0.000 % |
|     38719 |      44      322     1596 |      379      290      7 |  0.110 % |
==========================================================================

restarts              : 127
conflicts             : 40234          (218663 /sec)
decisions             : 46441          (0.00 % random) (252397 /sec)
propagations          : 415948         (2260587 /sec)
conflict literals     : 406932         (20.19 % deleted)
Memory used           : 22.00 MB
CPU time              : 0.184 s

UNSATISFIABLE
```

Listing 17: Output of minisat call for equisatisfiable CNF-formula of ramsey 3 3 6

References

[1] J. Harrison. Code and resources for "Handbook of Practical Logic and Automated Reasoning". `http://www.cl.cam.ac.uk/~jrh13/atp/`, Mar. 2009.

[2] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[3] A. Middeldorp. Lecture slides "Logic in Computer Science". `http://cl-informatik.uibk.ac.at/teaching/ws16/lics/slides/02x1.pdf`, WS 2016.

[4] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[5] T. Weber. A SAT-based Sudoku solver. In *LPAR-12*, pages 11–15, 2006.

A Environment setup

Please take note that the preprocessor installation (steps $2, 3$) is required for OCaml versions *3.10* or above.

1. Install OCaml

```
sudo apt-get install ocaml
```

2. Download and extract preprocessor camlp5

```
wget https://github.com/camlp5/camlp5/archive/rel617.tar.gz
tar -xvf rel617.tar.gz
```

3. Install preprocessor

```
cd camlp5-rel617/
./configure --strict
make world.opt
sudo make install
```

4. Get code and resources

```
cd ..
wget http://www.cl.cam.ac.uk/~jrh13/atp/OCaml.tar.gz
tar -xvf OCaml.tar.gz
cd OCaml/
```

5. Start OCaml

```
make
```

B Propositional Logic in OCaml

```
(* ------------------------------------------------------------------------- *)
(* Recognizing tautologies.                                                  *)
(* ------------------------------------------------------------------------- *)
let tautology fm =
  onallvaluations (eval fm) (fun s -> false) (atoms fm);;

(* ------------------------------------------------------------------------- *)
(* Return the set of propositional variables in a formula.                   *)
(* ------------------------------------------------------------------------- *)
let atoms fm = atom_union (fun a -> [a]) fm;;

(* ------------------------------------------------------------------------- *)
(* Interpretation of formulas.                                               *)
(* ------------------------------------------------------------------------- *)

let rec eval fm v =
  match fm with
    False -> false
  | True -> true
  | Atom(x) -> v(x)
  | Not(p) -> not(eval p v)
  | And(p,q) -> (eval p v) & (eval q v)
  | Or(p,q) -> (eval p v) or (eval q v)
  | Imp(p,q) -> not(eval p v) or (eval q v)
  | Iff(p,q) -> (eval p v) = (eval q v);;

(* ------------------------------------------------------------------------- *)
(* Eval.                                                                     *)
(* ------------------------------------------------------------------------- *)
let rec onallvaluations subfn v ats =
  match ats with
    [] -> subfn v
  | p::ps -> let v' t q = if q = p then t else v(q) in
             onallvaluations subfn (v' false) ps &
             onallvaluations subfn (v' true) ps;;
```

Listing 18: Propositional Logic in OCaml

C Sudoku formula generator for MiniSAT

```
open Format;;
open List;;

type atom = int * int * int;;
type lit = PLit of atom | NLit of atom;;
type clause = lit list;;
type cnf = clause list;;

let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
;;

let rec flat_map f = function
  | [] -> []
  | x :: xs -> f x @ flat_map f xs
;;

let rec fold f d = function
  | [] -> d
  | x :: xs -> f x (fold f d xs)
;;

let show_atom (x, y, z) =
  string_of_int x ^ string_of_int y ^ string_of_int z;;
let show_lit = function
  | PLit a -> show_atom a
  | NLit a -> "-" ^ show_atom a
;;
let rec show_clause c = String.concat " " (map show_lit c) ^ " 0";;
 (* fold (fun l s -> show_lit l ^ " " ^ s) "0";; *)
let rec show_cnf cnf = String.concat "\n" (map show_clause cnf);;
 (*fold (fun c s -> if c = [] then s else show_clause c ^ "\n" ^ s) "";; *)

let dd = [1;2;3;4;5;6;7;8;9];;
let gg = [[1;2;3]; [4;5;6]; [7;8;9]];;
let cc =
  flat_map (fun i -> map (fun d -> map (fun j -> (i, j, d)) dd) dd) dd @
  flat_map (fun j -> map (fun d -> map (fun i -> (i, j, d)) dd) dd) dd @
  flat_map (fun d -> flat_map (fun ii -> map (fun jj ->
    flat_map (fun i -> map (fun j -> (i, j, d)) jj) ii
  ) gg) gg) dd
;;

let at_least_one atoms = [map (fun a -> PLit a) atoms]

let at_most_one xs =
  flat_map (
  fun a -> fold
  (fun b cs -> if a < b then [NLit a; NLit b] :: cs else cs)
  [] xs
```

```
  ) xs
;;

let general_constraints =
  flat_map (fun i -> flat_map (fun j ->
    at_least_one (map (fun d -> (i, j, d)) dd)) dd
  ) dd @
  flat_map (fun aa -> at_most_one aa) cc @
  flat_map (fun i -> flat_map (fun j ->
    at_most_one (map (fun d -> (i, j, d)) dd)) dd
  ) dd
;;

let instance_constraints =
  map (fun a -> [PLit a]) [
    (1,2,2); (1,4,5); (1,6,1); (1,8,9);
    (2,1,8); (2,4,2); (2,6,3); (2,9,6);
    (3,2,3); (3,5,6); (3,8,7);
    (4,3,1); (4,7,6);
    (5,1,5); (5,2,4); (5,8,1); (5,9,9);
    (6,3,2); (6,7,7);
    (7,2,9); (7,5,3); (7,8,8);
    (8,1,2); (8,4,8); (8,6,4); (8,9,7);
    (9,2,1); (9,4,9); (9,6,7); (9,8,6)
  ]
;;

let constraints = general_constraints @ instance_constraints;;

print_string "p cnf ";;
print_int 999;;
print_string " ";;
print_int (List.length constraints);;
print_string "\n";;
print_string (show_cnf constraints);;
```

Listing 19: Sudoku formula generator for MiniSAT in OCaml