Formal Verification CW2

marko mekjavic

November 2021

1 Introduction

These are my answers to the second coursework for the Formal Verification course 2021. The document includes answers to the marked questions, henceforth, it starts with Question 2 rather than Question 1. Due to my limitations of LaTeX drawing skills, I have included some drawings, which help me explain and prove some of the statements in the coursework. All drawings and additional documents can be found in the Appendix.

2 Question 2

Consider the following two expressions:

Expression 1: !(a | b) ? h : !(a == b) ? f : g

Expression 2: !(!a | !b) ? g : (!a && !b) ? h : f

2.1 a)

In order to show manually that the expressions above are equal, I first translated both into their respective Binary Decision Trees. I then performed simple operations with which I changed the order of the layers while preserving the logical construction of the tree. The manual work can be seen in the Appendix 1.

2.2 b)

The above problem was encoded into SMT-LIB format. It can be seen in `question2.smt2` file. The output when running the file with the `Z3` command is:

unsat

3 Question 3

Note, it was explained to me during the lab sessions that a constant in `SMT-LIB` can either be expressed by using the `declare-const` argument or by using the `declare-fun` argument which does not take any arguments - and that this need not be interpreted as a *uninterpreted function*. That is why I did not interpret the examples below as uninterpreted functions but raher as constants.

3.1 a)

Logic chosen:
The logc theory I have chosen for this example is the `QF_UFIDL`.
We are dealing with quantifier free formulas in the expression and hence need to choose the `QF` logic.
Since all the variables are defined as constants on which a subtraction is performed, I have chosen the `IDL` logic because the expression is dealing with difference logic over integers.

Logic meaning:
Do there exist such functions that can assign non-negative value to both, x0 and x1 variables, and such that `x1 + 1 = x0`.

3.2 b)

Logic chosen:
The logic theory I have chosen for this example is the `QF_UFBV`.
We are dealing with quantifier free formulas in the expression and hence we need to choose the `QF` logic.
Since all the variables are defined as size 4 BitVector constants, I have chosen the `BV` logic for representing the BitVector logic as we are dealing with such type in this expression.
Lastly, the two functions `f` and `g` are *uninterpreted functions* and hence I needed to use the `UF` logic as well in order to support the expression.

Logic meaning:
Do there exist such functions `f` and `g` that assign variables v0 and v1 a BitVector of size 4, such that `f(v0) = g(v0)` and that `f(v1) != g(v1)`.

3.3 c)

Logic chosen:
The logic theory I have chosen for this example is the `QF_UFBV`.
We are dealing with quantifier free formulas in the expression and hence we need to choose the `QF` logic.
In our example we have three variables, two are defined as size 32 BitVectors and the other one as a Boolean. But since the expression only operates on the

BitVector variables, I have chosen the `BV` logic for representing the BitVector operations.

Logic meaning:
Do there exist such BitVector variables `x1` and `x2` of size 32, that when the operation of an exclusive *OR* is performed, the first bit is of value 1.

3.4 d)

Logic chosen:
The logic theory I have chosen for this example is the `QF_LIRA`.
We are dealing with quantifier free formulas in the expression and hence we need to choose the `QF` logic.
The operations are basic linear ones so hence I decided to use the *linear* logic, `L`.
Lastly, I have chosen the `IRA` logic because we are dealing with both; *integers* as well as *reals*.

Logic meaning:
Do there exist such non-negative *Real* variables `x0` and `x1`, that `x1` is exactly half the size of `x0`.

3.5 e)

Logic chosen:
The logic theory I have chosen for this example is the `QF_S`.
We are dealing with quantifier free formulas in the expression and hence we need to choose the `QF` logic.
Since the variable is defined as `String` I decided to use the *String* logic `S`.

Logic meaning:
Does there exist a `String` variable `x` such that it's value is `\0\1\2\3\04\00506\7\8\9ABC\\""\t\a\b` and that the length of it is equal to the value of variable `I`.

3.6 f)

Logic chosen:
The logic theory I have chosen for this example is the `ALIA`.
We are dealing with a quantifier in the expression and hence we can't choose the `QF` logic.
Since we are dealing with arrays I have decided to use `A` logic for arrays.
Lastly, since we are comparing integer values - which is a simple linear integer arithmetic, I chose the `LIA` logic.

Logic meaning:
Do there exist and array composed of integers, such that every value in that array is equal to 0.

3.7 g)

Logic chosen:
The logic theory I have chosen for this example is the `ALIA`.
We are dealing with quantifiers in the expression and hence we can't choose the `QF` logic.
Since we are dealing with arrays I have decided to use `A` logic for arrays.
Lastly, since we are comparing integer values - which is a simple linear integer arithmetic, I chose the `LIA` logic.

Logic meaning:
Do there exist such arrays a and b such that for every integer value in array a there exists some value j - specifying the position in array b - such that the two elements in both arrays on the respectable positions, are of the same value.

4 Question 4

Let's first explain the two logical formulas by breaking them down into two logical statements; *Formula 1* and *Formula 2*.

4.1 Formula 1

$$\bigvee_{j=1}^{m} x_{ij} \quad ; \quad \forall i \in [1, 2...n]$$

The formula stated above tells that every pigeon must sit somewhere. That is because x_{ij} is a propositional logical variable that is `True` if and only if pigeon i sits in pigeonhole j.
This can be more easily understood if we break the formula further:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} x_{ij}$$

which states that every pigeon must sit in one of the j pigeonholes.

4.2 Formula 2

$$(i \neq j) \implies (\neg x_{ik} \vee \neg x_{jk}) \; for \; all \; k \in [1, 2, ...m], \; and \; i, j \in [1, ...n]$$

The formula stated above tells that two different pigeons cannot sit in the same pigeonhole. This is even more obvious if we translate the above formula to the following:

$$\bigwedge_{k=1}^{m} \bigwedge_{i,j}^{n} (i = j) \ \vee \ \neg(x_{ik} \wedge x_{jk}) \ ; \ \forall k \in [1, ...m] \ and \ i, j \in [1, ...n]$$

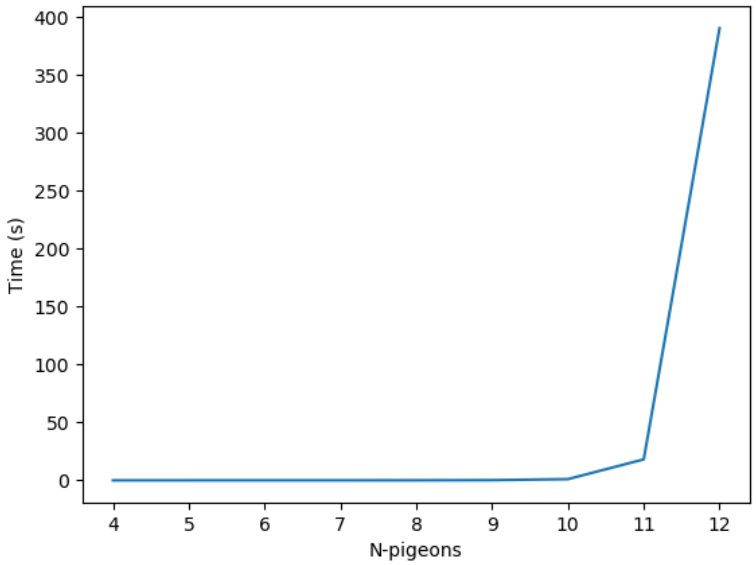
Where we can easily see that for every pigeonhole there must either be the case that pigeons occupying the pigeonhole are the same, or they do not occupy the same pigeonhole.

4.3 Script for generating CNF diagrams in DIMACs format

The script is available in the `question4b.py` file, and the CNF for *n=3* and *m=2* generated by the Python file is available in `question4b.cnf`.
When running the formula, we can see that it is *UNSATISFIABLE* as there are more pigeons than there are pigeonholes.

4.4 Runtimes with respect to n

Bellow is the plotted relationship between the number of pigeons (denoted as N-items on the *x-axis*) and the time needed to compute the *CNF* file generated by the script and ran by the `MiniSAT` programme.



Of course all of the `MiniSAT` returned the *UNSATISFIABLE* result for all the *CNF* files as in every file there were more pigeons then there were available

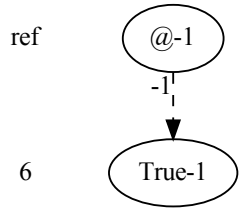
pigeonholes.
What I could observe from the run-times was that they of course increased as we introduced new pigeons to the problem. But what is interesting is that until we reached 10 pigeons, there was not that big of an increase in the CPU-time needed to compute the solution. Running the `MiniSAT` for 10 pigeons was the first time it took the computer more than a second to compute the solution. And after that we started to observe much steeper growth. For calculating solution for 11 pigeons it took the computer approximately 18 seconds, and to compute it for 12 pigeons it took it more than 390 seconds.
What is more, for 13 pigeons the computer is yet to compute a solution - check all possible clauses. For that reason I have denoted the results for 13, 14 and 15 pigeons as *infinity* - this can be seen in the Python file `plotting4c.py`. For that reason I also only plotted results for up to 12 pigeons as I realised it will be more complicated to represent *infinity* on the graph.
This also makes sense, because *the Pigeonhole Principle* is in terms of computational complexity a sub-class of class `TFNP` - the class of total function problems which can be solved in nondeterministic polynomial time.

5 Question 5

The code for this question can be found in `pigeon.py` file!

5.1 a)

In order to construct the `BDD` for the *Pigeon Hole Principle* described above, I first converted both formulas into a `CNF` format that is recognised by the `Python`'s `dd` library. In my code the procedure for both formulas can be easily followed through - see `pigeon.py`.
After each formula was successfully translated, I then conjugated them into a singular formula and checked with the `bdd` object whether the formula is unsatisfied - as it should be!
Indeed, the formula is unsatisfied! The `BDD` that was generated can be seen below:



Note that the example above is for the 3 pigeons! We see that from the root node (denoted with @) there is only one dotted arrow going out, which represents the `False` value - hence the formula cannot be satisfied :)

5.2 b)

When increasing the number of pigeons, the time needed for the programme to compute the BDD increases. As seen in the table below, the time is increasing somewhat normally until `n = 6` and afterwards it increases exponentially. Note that I could not run examples for `n > 8` as the programme started to take too long. For timing the programme I used the recommended linux command `time`.

n	time (s)
3	0.154
4	0.168
5	0.213
6	0.544
7	4.569
8	82.857

The `BDD` on the other hand stays the same in terms of number of nodes - still only the root node as well as the termination node. This is of course expected as the formula cannot be satisfied for any combination of the variables, regardless of n.

5.3 c)

Yes, the variable ordering does affect the run-time. Below you can see the two orderings of variables with their respective computational times written next to them. Note that the examples are for the three pigeons!

`Order 1:`

$$x_0^0 : 0,\ x_1^0 : 1,\ x_0^1 : 2,\ x_1^1 : 3,\ x_0^2 : 4,\ x_1^2 : 5$$

`time = 0.154s`

`Order 2:`

$$x_0^0 : 1,\ x_1^0 : 2,\ x_0^1 : 0,\ x_1^1 : 5,\ x_0^2 : 3,\ x_1^2 : 4$$

`time = 0.372s`

The reason behind why the order of the variable matters and why it affects the run-time is the following:
Given a `BDD`, the size of the graph for that respectable `BDD` depends on the variable order. That is because the algorithm considers variables in order when generating the tree and hence according to the inputted formula, it might matter in what order the algorithm considers the variables as it might be much easier

to first interpret the variables that are nested and then build outward. As the trees might differ in size based on the order, it then follows that building a larger tree will take more time than building a smaller one!
In the `Appendix 2` there are also two pictures, one showing the tree with a more efficient order and the second showing the tree with a less efficient order of variables :)

5.4 d)

Another encoding technique that would require less than *n * (n-1)* variables is where we would represent each pigeon sitting in a hole with a binary encoding - look at `Appendix 3`. For our `n-1` holes we can represent their position with $\log(n-1)$ variables, where their value is either true or falls. This is, because each variable would indicate whether the variable is on the left or on the right side of it. So starting with the first variable, it tells us whether it is on the left half or the right half. Second variable is then responsible for breaking in half both sides of already divided grid (by the first variable of course) and tells us if the hole is located in the left or in the right half. And so on until we come to the level of each hole. Since we are dividing it in half each time, we will need exactly $\log(n-1)$ of such divisions, each corresponding to a variable.
Since we need to specify a hole in which a pigeon sits for every single pigeon, this means that we would have at most $n * \log(n-1)$ variables. That means that a variable would be of the following format:

$$x_{bit}^{pigeon} \;\; pigeon \in [1, ...n] \; and \;\; bit \in [1, ... \; log(n-1)]$$

which shows that each pigeon would need *(n-1)* variables in order to specify his position.
This solution would most certainly increase the scalability opportunity as we would need much less variables to account for when computing the `BDDs`.

6 Question 6

In order to construct the an SMT solver for this particular example, we need to translate the two logical constraints into a form that the SMT solver will understand.
Since there are three shifts-per-day for every day in the week, this adds up to 21 shifts, which need to be covered by the 9 employees we have at our disposal. Hence, we shall define the following two sets:

$$i \in [1, ...9] \;\; = \;\; I \;\; \rightarrow \;\; Employees$$
$$j \in [1, ...21] \;\; = \;\; J \;\; \rightarrow \;\; Shifts$$

Let us now break the two constraints into three logical statements that an SMT solver will understand.

6.1 a)

For every shift there must be exactly two employees designated to it:

$$\forall j \sum_{i \in I} x_{i,j} = 2$$

6.2 b)

No employee is allowed to work on any back-to-back shifts:

$$\forall i, j(\neg x_{i,j} \vee \neg x_{i,j+1}) \; ; \; j \in [1, ...20]$$

Note that here we needed to restrict set J to 20 as we cannot define wrap-arounds. However, this has been addressed in the script by using the `modulo` operation.

6.3 c)

All employees must complete equal number of shifts.

$$\forall i1, \; \forall i2 \quad \sum_{j \in J} x_{i1,j} \; = \; \sum_{j \in J} x_{i2,j}$$

All of these three translated constraints have been implemented in `question6.py` Python script. When running the script we have received the output, that the SMT solver cannot satisfy the problem, which is correct as the shifts cannot be equally distributed amongst the employees.

6.4 Finding a new period!

In order to satisfy the above constraints, let us find a new time period. Let's decrease the period of seven days to only six (Sunday should be a day for the workers!), which translates to 18 shifts. The SMT solver has found a solution and the problem is *satisfied*!
Looking at the above constraints, we can see that in case of 18 shifts it is possible to give each of the employees equal number of shifts, without needing any back-to-back working schedules!

7 Appendix

7.1 Appendix 1

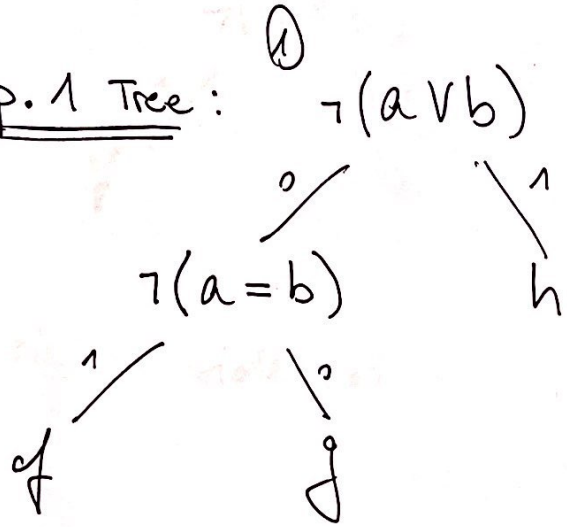
Please look at the `appendix1.pdf` for the added documents! :)

Appendix 1

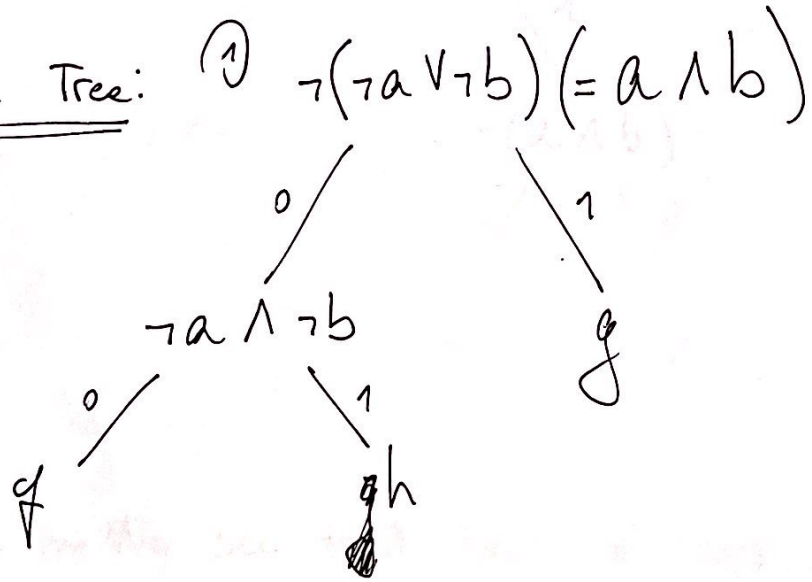
Expression 1 : $\neg(a \vee b) \,?\, h : \neg(a = b) \,?\, f : g$

Expression 2 : $\neg(\neg a \vee \neg b) \,?\, g : (\neg a \wedge \neg b) \,?\, h : f$

Exp. 1 Tree:

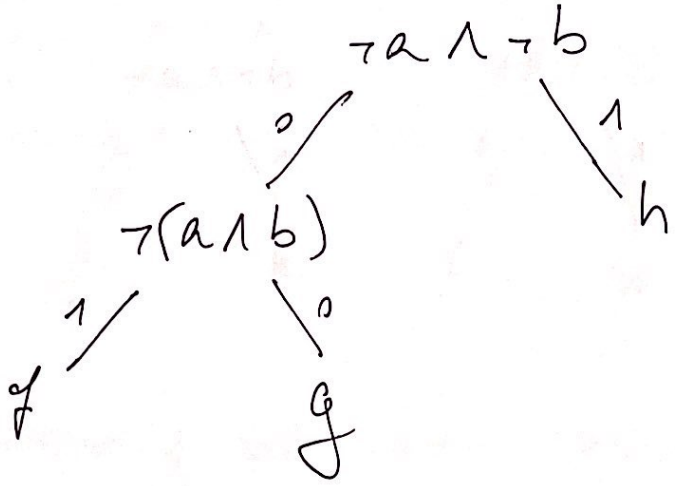


Exp. 2 Tree:

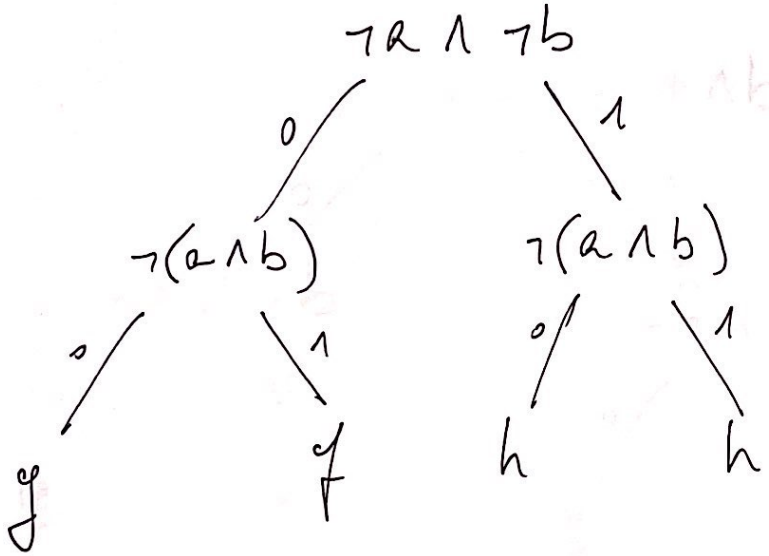


Let's first expand some of the logical trees on Exp. 1 Tree so that we can try and see if there is a way to reconstruct Exp. 1. Tree into that of Exp. 2 Tree.

② $\neg(a=b)$ can be translated to $\neg(a \wedge b)$, and $\neg(a \vee b)$ is broken into $\neg a \wedge \neg b$:

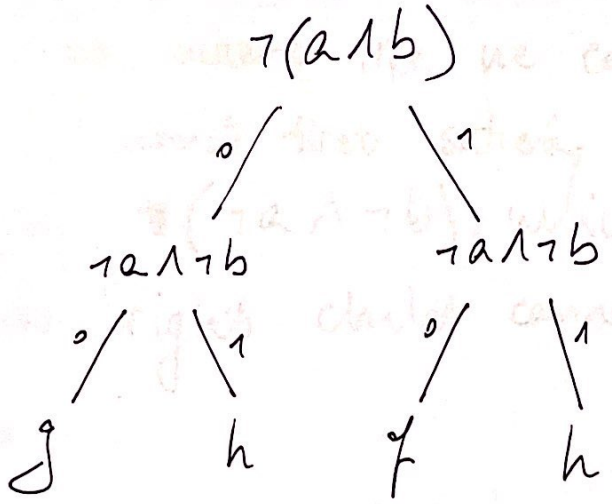


③ right side of the tree can be further expanded:

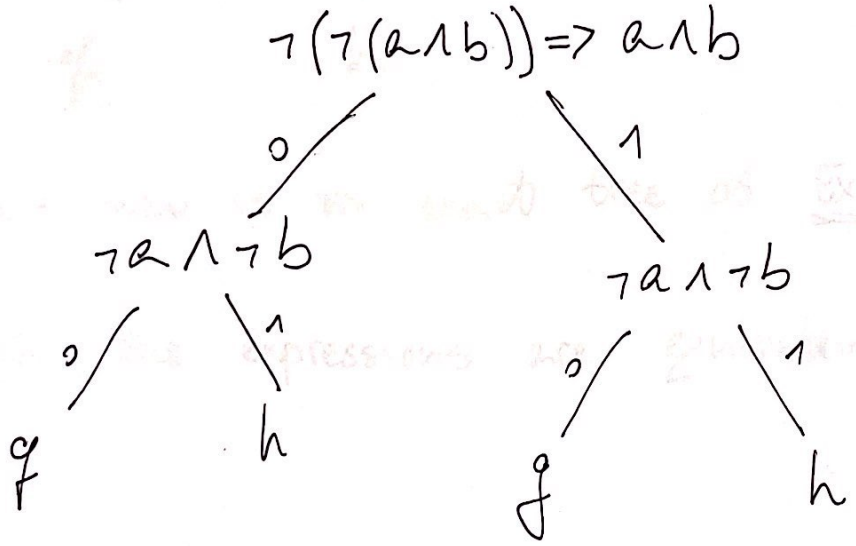


④ ~~since we~~ We see that both childrens are the same. What is more, since h can only be reached when $\neg a \wedge \neg b$ and $g \wedge f$ when not $\neg a \wedge \neg b$, we can switch the order of the nodes:

④



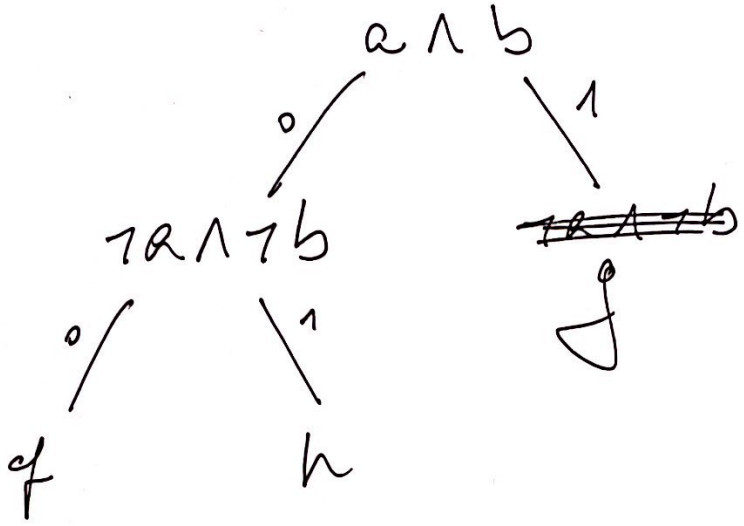
⑤ Comparing our current tree with the one of Exp. 2, we see that they are getting more similar. Let us now negate the root node.



*here we had to swap the child nodes as a consequence of negating the parent node.

⑥ From our current tree we can observe that we cannot first satisfy $a \wedge b$ and then satisfy $\Diamond(\neg a \wedge \neg b)$, which means h of the right child cannot be reached.

Hence:

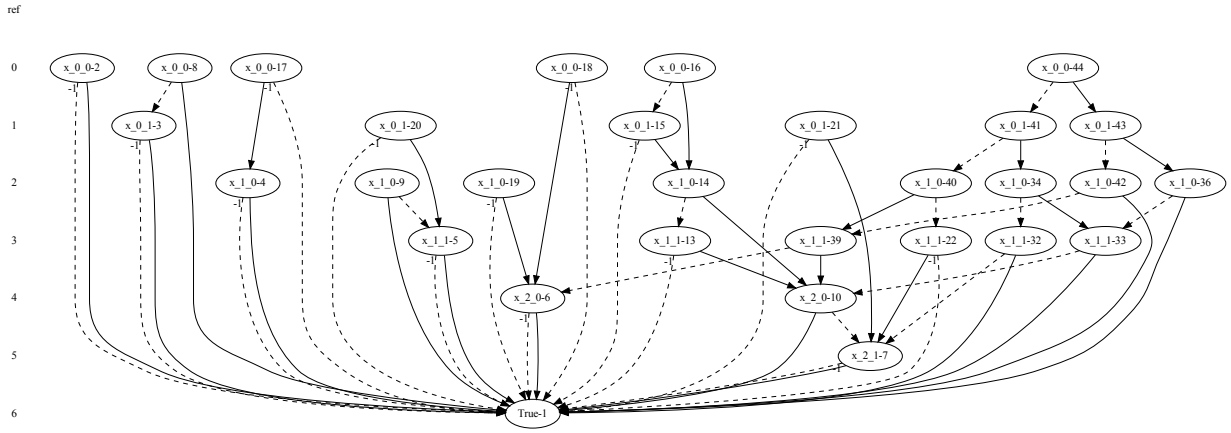


This now is the exact tree of Exp. 2.

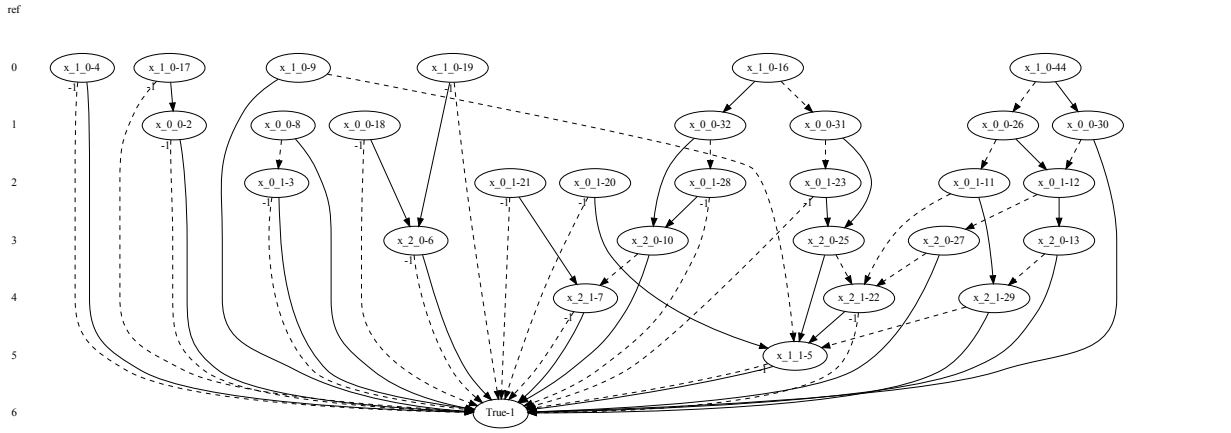
Hence, the expressions are equivalent. ■

7.2 Appendix 2

7.2.1 Before Reordering



7.2.2 After Reordering



7.3 Appendix 3