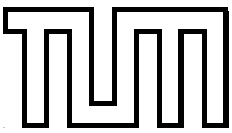


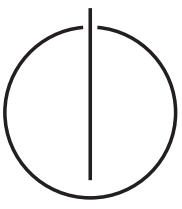
Technische Universität München

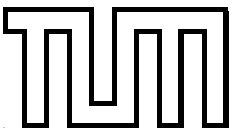
Fakultät für Informatik

Bachelorarbeit in Informatik

An Advanced Solver for Presburger Arithmetic

Moritz Fuchs





Technische Universität München

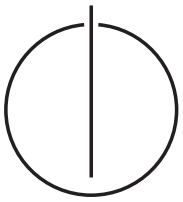
Fakultät für Informatik

Bachelorarbeit in Informatik

An Advanced Solver for Presburger Arithmetic

Ein fortschrittlicher Gleichungslöser für die Presburger Arithmetik

Author:	Moritz Fuchs
Supervisor:	Prof. Dr. Dr. h.c. Javier Esparza
Advisor:	M. Sc. Jan Kretinsky
Date:	August 17, 2011



I assure the single handed composition of this bachelor thesis only supported by declared resources.

Munich, August 15, 2011

. . . . . .
(Moritz Fuchs)

Acknowledgment

I am thankful to my supervisor Prof. Dr. Dr. h.c. Esparza who appointed me this work and always had time for my concerns and problems.
I also want to thank my advisor Jan Kretinsky who helped me especially in creating the grammar for the parser. Apart from that he tested the solver and helped me to correct some parts of this paper.
Andreas Gaiser and Rene Neumann also tested the solver and provided me with very useful feedback and bug reports.

This thesis would not have been possible without the help of my friends. I especially want to thank two of them.

- Fabian Streitel for constantly answering (really stupid) Linux questions and helping me to proof-read this work in a very short time.

- Felix Pascher for helping me to proof-read this work and for being such a great friend for many years now. He always got me back on track when something did not go the way I wanted it to.

Last of all I want to thank my family for their unconditional support. Whatever I did, you always had me covered.

Thank you all very much!

Abstract

Presburger arithmetic is the first-order theory of the natural numbers with addition. This work describes the implementation of a solver for Presburger arithmetic using automata that are labelled by Binary Decision Diagrams (BDDs). The focus lies mainly on user-friendliness.
The paper explains algorithms that are used by the solver with an emphasis on the BDD-parts. Algorithms for a different version of the Presburger arithmetic called bounded Presburger arithmetic are explained as well.
Finally a short insight into the performance of the solver is given to show that the overall performance is good, however the minimization algorithm is identified as a bottleneck.

Zusammenfassung

Presburger Arithmetik ist die Theorie erster Ordnung über den ganzen Zahlen mit Addition als Operation. Diese Arbeit beschäftigt sich mit der Implementierung eines Gleichungslösers für die Presburger Arithmetik, der mit Binären Entscheidungsbäumen (Binary Decision Diagrams - BDDs) beschriftete Automaten nutzt und sich vor allem auf die Benutzerfreundlichkeit konzentriert.
Die Arbeit erklärt die wichtigsten Algorithmen die vom implementierten Gleichungslöser benutzt werden. Das Hauptaugenmerk liegt dabei auf dem BDD-Teil. Algorithmen für eine andere Version der Presburger Arithmetik, der beschränkten Presburger Arithmetik, werden ebenfalls erläutert.
Am Ende der Arbeit wird ein Überblick über die Leistungsfähigkeit der entwickelten Software gegeben um zu zeigen, dass die Gesamtleistung gut ist, der Minimierungs-Algorithmus jedoch als Schwachstelle identifiziert wurde.

Contents

I. Introduction and Theory **1**

1. Introduction **3**
1.1. Presburger arithmetic . . . 3
1.2. Approach . . . 4
1.3. Overview over the work . . . 5

2. Datastructures **7**
2.1. Deterministic/Nondeterministic Finite Automata (DFAs/NFAs) . . . 7
2.2. BDD - Binary Decision Diagram . . . 8
2.3. BDD-labeled NFA/DFA . . . 10

3. Goals of the solver **11**

4. Algorithms **13**
4.1. LogicTree . . . 13
4.1.1. Pushing in negations . . . 13
4.1.2. Simplifying the formula . . . 16
4.2. Automata . . . 19
4.2.1. Union/Intersection . . . 19
4.2.2. Minimization . . . 22
4.2.3. Determination . . . 25
4.2.4. Negation . . . 28
4.3. Solution Space . . . 29
4.3.1. Number of solutions . . . 29
4.3.2. allSat . . . 31
4.3.3. N-Sat . . . 33
4.3.4. Optimized Solutions . . . 34
4.4. Solver . . . 35
4.4.1. Solve a logic tree . . . 35
4.4.2. Solve linear predicates . . . 37

5. Bounded Presburger arithmetic **39**
5.1. Solving a predicate . . . 39

II. Implementation and Conclusion **43**

6. Architecture **45**

6.1. Front end 45
6.1.1. GUI 45
6.1.2. Communication with the Server 48
6.2. Back end 49
6.2.1. Solver Request 49
6.2.2. Lexer/Parser 51
6.2.3. LogicTree 54
6.2.4. Solver 54
6.2.5. Getting solutions 54
6.2.6. Macros 55
6.3. Example 57

7. Performance **59**
7.1. Benchmark 59
7.1.1. Bench01 59
7.1.2. Alternating Quantifiers 60
7.1.3. Multiple predicates 61
7.2. Big factors 62

8. Conclusion **65**

III. Appendix **67**

A. Full grammar for ANTLR **69**

B. Benchmarks and test-cases **71**

C. MeanDifferencesAlg.csv **73**

D. Getting the Solver **75**
D.1. Supported platforms 75

Part I.

Introduction and Theory

1. Introduction

Presburger arithmetic is the first-order theory of the natural numbers with addition[1] . In contrast to the Peano arithmetic[1], which includes addition and multiplication, Presburger arithmetic is still decidable, however its complexity is doubly-exponential[6]. As such, writing a performant solver is a tough task and would go beyond the scope of this work, which is why this implementation of a solver for Presburger arithmetic is not focused on performance.
The main focus will hence lie on the user experience. The user should be able to input a formula without special knowledge in the field of automata theory or Presburger arithmetic. All they need to know is some basic knowledge about the syntax of formulae from Presburger arithmetic. To make the experience as convenient as possible, the actual solver will run on a server, the client in a web browser. That way the user will not need to install any special software apart from a modern web browser.

1.1. Presburger arithmetic

There are several slightly different definitions of the Presburger arithmetic. In this work, the definition from Esparza [4] will be used:
Each formula of Presburger arithmetic is constructed out of an infinitely big set of variables V={x,y,z,...} and the constants 0 and 1. First we need to define a term:

- The constants 0 and 1 are *terms*.
- Every variable $x \in V$ is a *term*.
- Assume t and u are terms, then t+u is a *term*.

An *atomic formula* is an expression $t \leq u$. A *formula* of Presburger arithmetic has to comply with the following rules:

- Every atomic formula is a *formula*.
- If A and B are formulae, then so are $\neg A$, $A \vee B$ and $\exists x : A$.

From this definition a few useful macros can be derived:

- n $\Leftrightarrow$ 1+...+1 (n times) where $n \in \mathbb{N}$
- nx $\Leftrightarrow$ x+...+x (n times) where $n \in \mathbb{N}$ and $x \in V$
- t=u $\Leftrightarrow t \leq u \wedge u \leq t$ where t and u are terms
- $t < u \Leftrightarrow t \leq u \wedge \neg(t = u)$ where t and u are terms

[1] Please see [2] for more information on this topic.

$\forall$,$\wedge$, $\Rightarrow$, as well as free variables are defined as for usual first-order logic.
The interpretation function $I : V \to \mathbb{N}$ is defined as one would expect:

$$
\begin{array}{lll}
I(0) = 0 & & \\
I(1) = 1 & & \\
I(t+u) = I(t) + I(u) & & \\
I \models t \leq u & \text{iff} & I(t) \leq I(u) \\
I \models \neg A & \text{iff} & I \not\models A \\
I \models A \vee B & \text{iff} & I \models A \vee I \models B \\
I \models \exists x : A & \text{iff} & \text{There is a } n > 0 \text{ so that } I[n \backslash x] \models A
\end{array}
$$

If a formula A has n free variables, this leads to a solution space Sol(A) which is a subset of $\mathbb{N}^n$. Since every number from Sol(A) is a n-tuple, we need a fix mapping from the variables to the numbers in the tuple. Therefore we need to define an order over the variables. In this work, the order will be defined by the lexicographical order over the variable names.

Example The formula $x - y = 5$ has a solution space of 2-tuples of numbers, where the first number is substituted for x, the second number is substituted for y. The solution space is then $S = \{(n, n-5) | n \in \mathbb{N} \wedge n \geq 5\} \subseteq \mathbb{N}^2$.
The formula $\exists x : 2x = y$ defines numbers that are divisible by 2. Since the variable x is bound in this formula, the solution space will only contain values for y, which are the even numbers: $S = \{$ y | y is even $\} \subseteq \mathbb{N}$.

1.2. Approach

The usual approach to solve formulae from Presburger arithmetic is to transfer the formula into an automaton, which then recognizes all solutions of the formula [4]. There have been different approaches with different kinds of automata. While Esparza [4] proposes usual Nondeterministic/Deterministic Finite Automata (NFAs/DFAs), other libraries like PresTAF make use of shared automata [2]
My approach will be almost similar to the one described by Esparza in [4] with the difference, that transitions will not be labelled with letters from the alphabet, but rather with Binary Decision Diagrams (BDDs), which then accept or reject the letters from the alphabet. The advantage of this approach is, that less memory is used in the average case, and therefore bigger formulae can be solved with the same amount of memory.

[2] More information on the PresTAF library is available under `http://altarica.labri.fr/forge/projects/altarica/documents`.

1.3. Overview over the work

The work is split into two separate parts.
The first part includes the chapters 2,3, 4 and 5. It will focus on the theoretical aspects of the task.

- Chapter 2 will start by introducing important data structures such as deterministic and non-deterministic finite automata (DFAs and NFAs), Binary decision diagrams (BDDs) and BDD-labelled NFAs and DFAs.
- Chapter 3 will give a concrete goal for the solver using an example.
- Chapter 4 will introduce algorithms which are used by the solver. This includes algorithms to optimize formulae, algorithms for BDD-labelled NFAs and DFAs, algorithms to compute accepting runs of an automaton and finally algorithms to transfer a formula into an automaton.
- Chapter 5 presents algorithms to solve formulae from the bounded Presburger arithmetic, which is a version of the Presburger arithmetic where only limited space is available for each variable.

The second part of the thesis focuses on the implementation of the solver.

- Chapter 6 will explain the front- and back end of the implemented software and show the important parts. In order to give a complete view of the solver, the chapter will close with a big example which will show how a user request is processed.
- Chapter 7 will give an insight into the performance of the implemented solver. It will compare the program to another implementation of a solver for Presburger arithmetic and give an overview of where the weaknesses of the implementation lie.

2. Datastructures

In this section all data structures that will be used in the algorithms will be presented. Additionally all needed operations on these structures will be explained shortly.

2.1. Deterministic/Nondeterministic Finite Automata (DFAs/NFAs)

DFAs/NFAs will form the basic data structure for the solver. Both, a DFA and an NFA are defined as 5-tuple consisting of

- Q - the state space,
- Σ - the alphabet,
- δ - a transition function,
- $q_0 \in Q$ - the initial state,
- $F \subseteq Q$ - the set of final states.

DFA and NFA differ only in the transition function δ. For DFAs, the function is defined as $\delta : Q \times \Sigma \to Q$. The definition for NFAs is $\delta : Q \times \Sigma \to P(Q)$. This means, a letter in a DFA can only lead to one state, whereas a letter in an NFA can lead to multiple states. A run of a DFA on the input $a_0 a_1 ... a_{n-1}$, where for each $0 \leq i \leq n : a_i \in \Sigma$, is defined as $p_0 \to p_1 \to ... \to p_n$ where for each $0 \leq i \leq n-1$: $p_i \in Q \wedge \delta(p_i, a_i) = p_{i+1}$. The run is accepting if $p_n \in F$.
A run of an NFA is defined as for DFAs, but rather than $\delta(p_i, a_i) = p_{i+1}$ we put $p_{i+1} \in \delta(p_i, a_i)$.
L(A) defines the language of an automaton: $L(A) = \{\omega | \omega \in \Sigma^* \wedge$ A has an accepting run on $\omega\}$.

Example Let's consider the automaton A in figure 2.1 with the alphabet $\Sigma = \{0, 1\}$. As one can easily see, the transition function δ is written directly to the edges of the automaton. One can also see, that the function has exactly one target for each state and each letter. Therefore, this is a DFA. The set of final states of the automaton are always marked by a doubly frame. In the example in figure 2.1, the state q_1 is final and all other states are non-final.
The language L(A) is quite easy to recognize. If the first letter is 0, the automaton will always accept the word, else it will always deny it. If we see the accepted word as number encoded over 0 and 1 using the little endian encoding, the automaton recognizes every even number, and therefore L(A) = {y | y is even}.

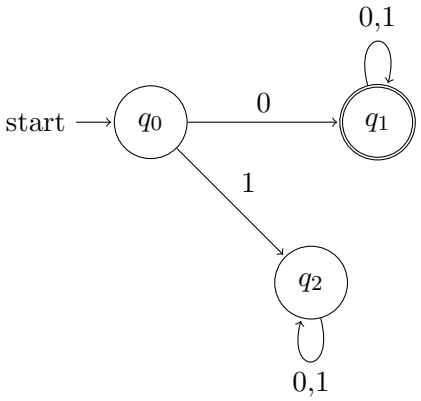


Figure 2.1.: Sample automaton A with L(A) = {y | y is even}

In the following work, we will always use automata that encode numbers using the little endian encoding. Therefore each transition will encode one bit of each variable. This means if we want to encode n numbers, every transition letter has to have n bits. The alphabet-size therefore grows with the number of variables: $|\Sigma| = |\{0,1\}^n| = 2^n$.

Remarks on the language: In the following work the name trap state will describe a state from where every transition leads to itself. Additionally it has to be non-final. That way, if a run reaches this state, it will not be able to leave it again. (Therefore its name trap state)

2.2. BDD - Binary Decision Diagram

Binary Decision Diagrams or short BDDs are used to represent boolean functions as automata with certain properties. The first property is, that a BDD has at most one loop. This loop has to be from the trap state to the trap state. As such, a BDD always has a limited number of accepting runs. The second property describes the length of accepting words. Since BDDs represent boolean functions, all accepting words need to have the same length, because the number of variables in a boolean function is fixed.
Therefore a BDD can be viewed from two different angles. Firstly we can see a BDD as a boolean function, where an input is either accepted or rejected. Secondly we can view a BDD as a set of binary words of equal length, where every word in the set is accepted by the BDD.

Example The following BDD recognizes all inputs that satisfies the formula $x \wedge \neg y$ and therefore the set $S = \{10\}$:

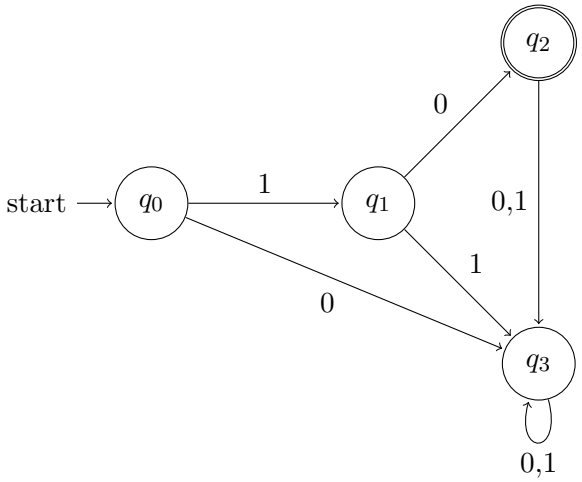


Figure 2.2.: Sample BDD that recognizes the formula $x \wedge \neg y$ and therefore the set $S = \{10\}$.

The following operations on BDDs, where a and b are the BDDs, are used in the algorithms:

- $a \cup b$ / $a \vee b$: Computes a BDD for the union of the accepting runs of both BDDs. So each input that satisfies one of the two BDDs is accepted.
- $a \cap b$ / $a \wedge b$: Computes a BDD for the intersection of the accepting runs of both BDDs. So each input that satisfies both BDDs is accepted.
- $x \in a$: x is an input that is accepted by a.
- $\exists x : a$: Applies an existential quantifier for variable x on the BDD a.
- $\neg a$: negates the BDD a.
- allSatCount(a): counts the number of accepting runs for the BDD a.

As one can see, BDDs can be used as boolean functions with the according logical operations. They can also be used as sets of n-tuples of 0s and 1s, where n is the number of free variables in the function. Each tuple in the set satisfies the binary function.

For more detailed information on BDDs, please see [10],[11] or [12].

2.3. BDD-labeled NFA/DFA

As mentioned in the section on NFAs and DFAs, our automata will always encode numbers using a binary encoding. We also learned, that as the number of variables grows, the alphabet Σ grows. This means that the transition function δ grows. If a traditional lookup table of the form (state, letter, state) is used, this approach will take up a lot of memory when n gets bigger.
This is why we will not use traditional automata, but rather BDD-labelled automata. As we saw before, BDDs can be used to represent boolean functions. Since there is a boolean function for every set of words,where every word has the same length, we can use BDDs to encode the transitions. Let us assume, we want to save a transition from state q_1 to q_2 with the letters 00,10,01. In a traditional implementation we would save (q_1,00,q_2),(q_1,10,q_2) and (q_1,01,q_2) to the lookup table. In our case we will just create a BDD b that accepts the set $S = \{00, 10, 01\}$ and save (q_1,b,q_2).
To sum it up, we alter the lookup table from (state,letter,state) to (state, bdd, state) to get a more compact representation of the transition function.

Remark on the figures of automata: In the following work all NFAs and DFAs will be labelled with BDDs. Since it would not be convenient to put the BDD labels into the figures, we will spare this out and use traditional labels instead. However, please keep in mind that this is only for convenience and the data structure is implemented using BDD labels.

Remark on syntax in the algorithms: In some of the algorithms, a state is known and the algorithm wants to get all transitions, thus all reachable states and the BDDs that label the transitions to these states. This is accomplished using the following line:

$$\text{for all BDD bdd, State s: } \delta(q, bdd) = s \text{ do}$$

3. Goals of the solver

The goal of this work is to implement a solver for Presburger arithmetic which uses BDD-labelled automata as a data structure. The target of this chapter is to give an intuition on how the result of the solver should actually look like.
The solver will take a formula from Presburger arithmetic and compute an automaton which encodes all solutions to this formula.
Let us consider the formula $\exists x : (x + y \leq 4 \wedge x = 2)$ and see which automaton should be computed. The formula states, that there is an x, so that x is equal to two, which is trivial, and that $x + y \leq 4$. Since x has to be 2, the formula can be altered to $\exists x : y \leq 2 \wedge x = 2$. The x is then cancelled out because of the existential quantifier. Therefore the formula is true for each $y \leq 2$. The DFA encoding this set of numbers is shown in figure 3.1.

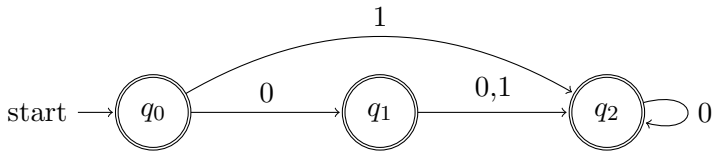


Figure 3.1.: Automaton for $\exists x : x + y \leq 4 \wedge x = 2$. Please note, that the trap state was left out for better visibility. All missing transitions lead to the trap state.

Since the automaton accepts only values for y which are smaller or equal to 2, the solution space will be $S = \{0, 1, 2\}$. Therefore a result of the solver should include a representation of the minimal automaton as well as the solution space. Additionally there should be ways for the user to change the properties of the returned solution space. This includes:

- All solutions: The solver tries to compute all solutions for the formula.
- Optimized solutions: The user should be able to enter a term consisting of free variables from the formula which is then maximized or minimized. So if the user wants to maximize the term 2y in the above example, the maximal value should be 4 with a value of 2 for y.
- Limited number of solutions: The user should be able to limit the number of returned solutions. So if the user requests 2 solutions in the above example, possible valid solutions would be $S = \{0, 1\}$, $S = \{1, 2\}$ and $S = \{0, 2\}$.

Since the solver focusses on the user experience, there has to be the possibility to define macros which can then be used in all formulae of the user. As such the user has to be able to declare a name for a formula and use this name synonymously with the associated formula. The parameters of a macro are the free variables of the associated formula. It

should be possible to substitute a parameter with a term instead of just a variable That way it is possible to define properties of whole terms instead of just variables.

In order to reach the stated goals and get from a text input of the formula to an automaton and actual solutions of the formula, we first need to find algorithms for BDD-labelled automata. This includes union, intersection, existential quantifiers, minimization of automata and determination of automata. Since these algorithms are already available for traditional NFAs and DFAs, we will adapt them for the BDD-labels if possible and focus on the BDD-parts of the adapted algorithms. Apart from algorithms for automata, algorithms to optimize formulae as seen in the above example above will be shown. As the main task is to turn a formula into an automaton, an algorithm for the translation from a formula to an automaton will be shown. Finally algorithms for computing actual solutions from the automaton will be explained.

4. Algorithms

In this chapter the emphasis lies on explaining the algorithms which are used on the way to solving a formula and getting its solution space as well as subsets of it. The first section will explain how a formula is optimized. The algorithms for pushing negations and checking whether a sub-formula can be optimized in some way will be discussed.
The second section will explain the basic operations on automata such as union, intersection, negation, minimization and determination. The focus will lie on how to change existing algorithms to work with BDDs instead of traditional labels on the transitions.
The third section will explain the algorithms which are used to find a subset of the solution space of the formula that suits the properties that the user ordered. This includes counting the number of solutions for an automaton, as well as three different ways of getting the solutions, which are namely AllSat, N-Sat and an algorithm for getting optimized solutions for a certain linear term.
The chapter will close with algorithms to translate a formula in a LogicTree format into an automaton.

4.1. LogicTree

The starting point for the algorithms in this section are trees that look as in Figure 4.1. The nodes are always operators such as $\wedge$, $\vee$, quantifiers or negations. On each leaf of the tree is a predicate of one of the following forms:

$$\sum_{k=0}^{i} a_i x_i = b$$
$$\sum_{k=0}^{i} a_i x_i \neq b$$
$$\sum_{k=0}^{i} a_i x_i \leq b$$

4.1.1. Pushing in negations

This algorithm is used to get rid of as many negations as possible. This is done by applying De Morgan's law exhaustively. De Morgan's law is defined as

$$\neg(a \vee b) = \neg a \wedge \neg b \tag{4.1}$$
$$\neg(a \wedge b) = \neg a \vee \neg b \tag{4.2}$$

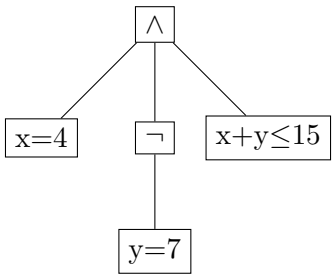


Figure 4.1.: A sample tree for $x = 4 \wedge \neg y = 7 \wedge x + y \leq 15$

As such our first task is to search the tree until we find a negation. Once we have found one, we will try to push it deeper into the tree.

```
1 pushNegations:
2 Input: Tree a
3  while x = a.findNegation() do
4   negate (x.child)
5   remove x from a
6  end;
```

Listing 4.1: Algorithm to push negations down in the LogicTree

The pushNegation algorithm starts the whole process of pushing negations into the tree. It searches the earliest negation and sends a negate-signal to its child. After the child is negated the negation itself is deleted.
The negate algorithm is available in two different forms. One is for nodes, the other one for leafs:

```
1 negate:
2 Input: Node a
3   switch a
4     case AND:
5       a = OR;
6       for all child ∈ a.children do
7         negate(child)
8       end
9     case OR:
10      a = AND;
11      for all child ∈ a.children do
12        negate(child)
13      end
14    case NEGATION:
15      remove a from tree
16    default:
17      insert ¬ before a in the tree
```

Listing 4.2: Negation algorithm for nodes of a tree

If the input of the negate algorithm is a node we have three different possible cases:

- The node is an AND or an OR. In this cases De Morgan's law applies.

- The node is another NEGATION. In this case, the negation is just cancelled out.
- The node is an existential-quantifier. In this case we just put a negation above this node in the tree.

The negate algorithm for leafs is fairly simple. If the leaf is of type equal or not equal, it will just swap it to not equal, resp. equal. If the type of the leaf is less or equal, the type is changed to greater, but since this is not a formula of one of the three forms defined above, it will be transferred back to the less or equal form.

```
1 negate:
2 Input: Formula a
3   if a.TYPE = EQUAL
4     a.TYPE = NOT_EQUAL
5   if a.TYPE = NOT_EQUAL
6     a.TYPE = EQUAL
7   if a.TYPE = LESS_EQUAL
8     a.LEFT_SIDE *= -1;
9     a.RIGHT_SIDE += 1;
10    a.RIGHT_SIDE *= -1;
```

Listing 4.3: Negation algorithm for leafs of a tree

Complexity

Since every node in the tree is met at most once, the complexity is O(n+l) where n is the number of nodes and l the number of leafs of the tree.

Example

Let us take a look at an example to get a better intuition on how this works.

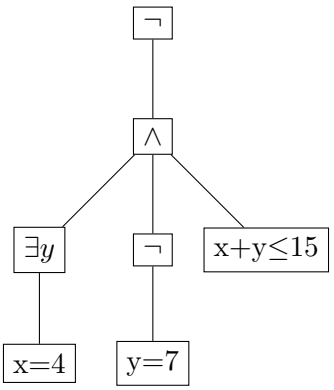


Figure 4.2.: The tree for $\neg(\exists y : x = 4 \wedge \neg y = 7 \wedge x + y \leq 15)$

We will now run the pushNegation algorithm on the tree in Figure 4.2. The algorithm will recognize the negation on top of the tree as the first negation and call negate on its child, which is of type $\wedge$. The $\wedge$ will be turned into an $\vee$ by De Morgan's law. After that each child of the former $\wedge$ will be negated.

The left edge leads to an existential quantifier, which can not be negated. Hence the negation is just put in front of the quantifier.
The second child is a negation. Since the two negations cancel each other out, the second child is just deleted.
The third child is a formula. Since its type is less or equal, lines 6-9 of the negate algorithm for leafs apply.
The complete tree after applying pushNegations looks as in figure 4.3:

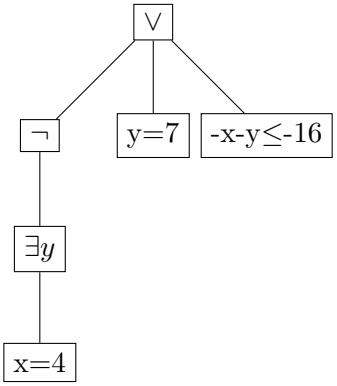


Figure 4.3.: The tree after applying pushNegations.

4.1.2. Simplifying the formula

In this section the goal is to describe an algorithm to recognize and get rid of unnecessary predicates, such as duplicates, or formulae that are already included in other formulae.
The algorithm is split into three parts. First we have to find a predicate to propagate, after that the predicate is pushed up the tree until an $\vee$ or an existential quantifier is reached. Along the way the predicate is already propagated to the sub-trees of the nodes that are met on the way upwards. When the algorithm reaches an $\vee$ or an existential quantifier the predicate is propagated down all sub-trees to the leafs, where the predicates are being compared and possibly changed.
The first part is rather technical and relies on a simple in-order iteration, therefore it will not be covered here. Let us take a look at part two.

```
1 pushUp:
2 Input Node n  Formula f:
3 switch n
4   case AND:
5     for all child ∈ n.children ∧ f not in subtree child do
6       propagate(child , f , AND)
7     end
8     if exists( n.father ) then
9       pushUp(n.father , f)
10  case OR/EXISTS:
11    for all child ∈ n.children ∧ f not in subtree child do
12      propagate(child , f , OR)
13    end
```

Listing 4.4: Algorithm for pushing a formula upwards

Part three is the propagate algorithm, which exists for nodes and for leafs. The version for nodes basically checks if the current node is an $\wedge$ or an $\vee$. If that's true, it continues to propagate the predicate downwards.

```
1 propagate:
2 Input Node n, Formula f, Type t:
3 switch n
4   case AND:
5     foreach child from n.children do
6         propagate(child , f , t)
7     end
8   case OR:
9     foreach child from n.children do
10        propagate(child, f , t)
11    end
```

Listing 4.5: Propagate algorithm, to push a formula to the leafs

The propagate algorithm for two formulae is rather technical and quite large, which is why we spare it out. Table 4.1 shows the basic behaviour of the algorithm. The first two columns represent the types of predicate rec and prop. rec is the predicate that receives the propagation of prop. The third column is the type of the propagation, i.e. the type of the node that sent the propagation for this sub-tree first. In column four and five you can find the relation between the prefixes and the bounds. rec = prop means, that rec and prop have the same variable set with the same prefixes for each variable. rec $\supset$ prop means that each variable from prop exists in rec and has the same prefix in it. The last column contains the action that is performed. True/false means, that the predicate is replaced by true/false, substitute means, that prop is substituted by the bound of prop in rec, so every variable from prop is deleted in rec and the bound of rec is subtracted by the bound of rec b_{rec}. EQUAL means, that the type of the predicate rec is set to EQUAL.

rec	prop	Type	prefix relation	bounds relation	result
$=$	$=$	AND	rec = prop	$b_{rec} = b_{prop}$	true
$=$	$=$	AND	rec $\supset$ prop	-	substitute
$=$	$\neq$	AND	rec = prop	$b_{rec} = b_{prop}$	false
$=$	$\leq$	AND	rec = prop	$b_{rec} > b_{prop}$	false
$\neq$	$=$	AND	rec = prop	$b_{rec} = b_{prop}$	false
$\neq$	$=$	AND	rec $\supset$ prop	-	substitute
$\neq$	$\leq$	AND	rec = prop	$b_{rec} > b_{prop}$	true
$\leq$	$=$	AND	rec = prop	$b_{rec} = b_{prop}$	true
$\leq$	$=$	AND	rec $\supset$ prop	-	substitute
$\leq$	$\leq$	AND	rec = prop	$b_{rec} > b_{prop}$	delete
$\leq$	$\leq$	AND	rec = -prop	$b_{rec} = -b_{prop}$	EQUAL
$\leq$	$\leq$	AND	rec = -prop	$b_{rec} < -b_{prop}$	false
$=$	$\leq$	OR	rec = prop	$b_{rec} =$ b2	delete
$\leq$	$\leq$	OR	rec = prop	$b_{rec} \leq b_{prop}$	delete
$\leq$	$\leq$	OR	rec = -prop	$b_{rec} = -b_{prop}$	true

Table 4.1.: Behavior of the propagate algorithm for two formulae.

Now we can assemble the actual algorithm:

```
1 optimize:
2 Input Tree t:
3
4 changed = true;
5 while changed do
6   while hasNextFormula( t ) do
7     f = nextFormula( t );
8     pushUp (f.father , f);
9   end;
10  if not t.hasChanged()
11    changed = false;
12 end;
```

Listing 4.6: Algorithm to optimize a given LogicTree.

The algorithm will continue to optimize the tree as long as at least one predicate of the tree changes at the previous run. That way we can be sure that optimizations that were done in the previous run, are propagated to every other reachable predicate again.

Complexity Since we have to restart the algorithm every time the tree changed, the algorithm will be in $O(n^2)$, where n is the number of leafs in the tree.

Example Assume we want to optimize the tree in figure 4.4

The algorithm will start to search a predicate to propagate to the rest of the tree. The first one it will find is $x \leq 15$. It will be pushed up and propagated to the rest of the tree with $\wedge$ as operation since we can not go up further. As nothing can be done, nothing happens.

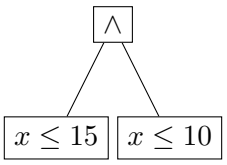


Figure 4.4.: Logic Tree to be optimized

The second predicate to be propagated is $x \leq 10$, which is also propagated with $\wedge$. When it is propagated to $x \leq 15$, the algorithm will recognize that $x \leq 10$ is part of $x \leq 15$ and just delete $x \leq 15$ from the tree since both predicates have to be satisfied. Since the $\wedge$-node now only has one child, it will be removed from the tree as well. The result of the optimization is shown in figure 4.5.

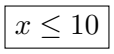


Figure 4.5.: Tree after optimization.

4.2. Automata

In this section the most interesting algorithms for BDD-labelled automata will be discussed. Since most of them rely greatly on algorithms for usual automata, which have been discussed in lots of papers before, the focus will be on the BDD-parts of the algorithms.

4.2.1. Union/Intersection

The pairing-algorithm described by Esparza in [4] plays a major role in developing the algorithm for union/intersection of BDD-labelled automata. It is used for union as well as for intersection and only differs in the requirements for final states. We want to adapt this algorithm for BDD-labelled automata.
Let's first take a look at the pairing algorithm for usual NFAs:

```
1 pairing:
2 Input NFA A = (Q_A, Σ, δ_A, q_{A,0}, F_A) NFA B = (Q_B, Σ, δ_B, q_{B,0}, F_B) Binary
      Operation ⊙
3 Output NFA C = (Q, Σ, δ, q_0, F)
4 Q = ∅, F = ∅
5 q_0 = [q_{A,0}, q_{B,0}]
6 W = {q_0}
7 while W ≠ ∅ do
8   pick [q_1, q2] from W
9   add [q_1, q2] to Q
10  if q_1 ∈ F_A ⊙ q_2 ∈ F_B then
11    add [q_1, q2] to F
12  for all a ∈ Σ do
13    for all q_3 ∈ δ_A(q_1, a), q_4 ∈ δ_B(q_2, a) do
14      if [q_3, q_4] ∉ Q then
15        add [q_3, q_4] to W
16      add ([q_1, q_2], a, [q_3, q_4]) to δ
17    end
18  end
19 end
```

Listing 4.7: Pairing algorithm for usual automata

As we can see, the algorithm relies mainly on an iteration over every letter of the alphabet in the outer loop. This will work since every letter is saved in a plain format in the lookup table. In our case though, we save transitions labelled by binary functions in the BDD-format. Therefore, we can perform operations such as intersection of union on them. This can be used for a pairing algorithm.
Instead of relying on an iteration over the alphabet, we will take the targets of each transition for both states and make the cross product of the two sets. After that we will go through the set of pairs and try to make transitions for them. This is done by computing the intersection of the BDDs leading from q_1 to q_3 in A and q_2 to q_4 in B. If the resulting BDD is not empty, the new transition is added to the new transition table.
The resulting algorithm will look as follows:

```
1 pairingBDD:
2 Input NFA A = (Q_A, Σ, δ_A, q_A,0, F_A) NFA B = (Q_B, Σ, δ_B, q_B,0, F_B) Binary
      Operation ⊙
3 Output NFA C = (Q, Σ, δ, q_0, F)
4 Q = ∅, F = ∅
5 q_0 = [q_A,0 , q_B,0]
6
7 add [q_A,0 , q_B,0] to Q
8 if q_A,0 ∈ F_A ⊙ q_B,0 ∈ F_B then
9   add [q_A,0 , q_B,0] to F
10
11 W = {q_0}
12 while W ≠ ∅ do
13   pick [q_1 , q_2] from W
14   for all [q_3, q_4], [bdd_3, bdd_4] : δ_A(q_1, bdd_3) = q_3 ∧ δ_B(q_2, bdd_4) = q_4 do
15     if bdd_3 ∩ bdd_4 in not empty then
16       if [q_3, q_4] ∉ Q ∧ [q_3, q_4] ∉ W then
17         add [q_3 , q_4] to Q
18         add [q_3 , q_4] to W
19         if q_3 ∈ F_A ⊙ q_4 ∈ F_B then
20           add [q_3 , q_4] to F
21       add ([q_1 , q_2] , bdd_3 ∩ bdd_4 , [q_3 , q_4] to δ
22   end
23 end
24 return (Q, Σ, δ, q_0, F)
```

Listing 4.8: Pairing algorithm for BDD-labelled automata

To Implement Union and Intersection we just insert different operations for $\odot$. For union we insert $\vee$, for intersection $\wedge$.
Since the cross product of the two state sets is finite, the algorithm will terminate.

```
1 intersection:
2 Input NFA A = (Q_A, Σ, δ_A, q_A,0, F_A) NFA B = (Q_B, Σ, δ_B, q_B,0, F_B)
3 Output NFA C = (Q, Σ, δ, q_0, F)
4
5 return pairingBDD (A , B , ∧)
```

Listing 4.9: Intersection for BDD-labelled automata

```
1 union:
2 Input NFA A = (Q_A, Σ, δ_A, q_A,0, F_A) NFA B = (Q_B, Σ, δ_B, q_B,0, F_B)
3 Output NFA C = (Q, Σ, δ, q_0, F)
4
5 return pairingBDD (A , B , ∨)
```

Listing 4.10: Union for BDD-labelled automata

Of course the common implementation of union for NFAs, which is just running both automata side by side can be applied as well. Since the BDD-implementation does not differ at all from the usual implementation we will not cover it here. Further information can be found in the work by Esparza in [4].

Complexity The outer while loop will be run at most $|Q_A| \cdot |Q_B|$ times, the inner for-loop at most $|\Sigma_A| \cdot |\Sigma_B|$. Assuming, that lines 15 to 21 of the pairingBDD algorithm are

in O(1), this makes a total of O($|Q_A| \cdot |Q_B| \cdot |\Sigma_A| \cdot |\Sigma_B|$).

4.2.2. Minimization

The minimization algorithm relies on Hopcroft's algorithm as described for example by Berstel in [9].

```
 1 minimize:
 2 Input DFA A = (Q_A, Σ, δ_A, q_A,0, F_A)
 3 Output DFA B = (Q, Σ, δ, q_0, F)
 4 P = {F_A , Q_A\F_A}
 5 W = ∅
 6 for all a ∈ Σ do
 7   add (min(F_A , Q_A\F_A),a) to W
 8 end
 9 while W ≠ ∅ do
10   pick (A,l) from W
11   for all P_current ∈ P which is split by (A,l) do
12     (P'_current, P''_current) = split(P,A,l)
13     replace P_current by P'_current and P''_current in P
14     for all b ∈ Σ do
15       if (P_current, b) ∈ W then
16         replace (P_current, b) by P'_current and P''_current in W
17       else
18         add(min(P'_current ,P''_current )) to W
19     end
20   end
21 end
22
23 return (Q, Σ, δ, q_0, F)
```

Listing 4.11: Minimization algorithm by Hopcroft.

Since the algorithm was already described in length, for example by Berstel in [9], we will not describe it again, but rather take a closer look at the split algorithm in line 12 as well as the check if a class is splittable in line 11. Those should be the most interesting parts of this algorithm considering the BDD environment.

isSplittable

The isSplittable algorithm takes two classes P and A from the current partition as well as a letter l from the alphabet and checks if the class P can be split by (A,l). To do so, the algorithm has to go through each state of the P and check if, after taking the transition for letter l, we reach the class A or not. If we can find states in P that reach A and states that do not reach A, we return true, else we return false.

```
 1 isSplittable:
 2 Input Class P Class A Letter l
 3 Output true if P can be split by (A,l), false if not
 4
 5 inFound = false
 6 outFound = false
 7 for all State s ∈ P do
 8   State q = δ(s,l)
 9   if q ∈ A then
10     inFound = true
11   else
12     outFound = true
13
14   if inFound ∧ outFound then
15     return true
16 end
17
18 return false
```

Listing 4.12: Algorithm to check whether a class can be split. Please note that line 8 is just a short cut. We have to check the BDDs on the transitions for the state s in order to get q.

This implementation seem to be pretty straightforward. However, line 8 will be quite hard to implement, since in the worst case scenario we would have to check every BDD for this state. When assuming, that we can check a BDD in $O(1)$, this would make it $O(|\Sigma|)$ for each state and $O(|P| \cdot |\Sigma|)$ in total for this algorithm. Since

$$|\Sigma| = 2^n$$

where n is the number of variables, the algorithm will get quite slow for a big number of variables.
Because this check is used in every loop of the algorithm, this will clearly be the bottleneck of the minimize algorithm.

split

The split algorithm is actually pretty much the same algorithm as isSplittable, but instead of just remembering if there exist states that were inside of A or not, we will have to save two separate sets for states that were inside of A resp. outside of A.

```
1 split:
2 Input Class P Class A Letter l
3 Output The splitted class P.
4
5 inside = ∅
6 outside = ∅
7 for all State s ∈ P do
8   State q = δ(s,l)
9   if q ∈ A then
10    add q to inside
11  else
12    add q to outside
13
14 end
15
16 return (inside, outside)
```

Listing 4.13: Algorithm to check whether a class can be split. Please note that line 8 is just a short cut. We have to check the BDDs on the transitions for the state s in order to get q.

Again the bottleneck of the algorithm will be line 8.

In practice it would be advisable to merge the two algorithms or just use the split algorithm. Else some of the BDDs will have to be checked twice, once for checking if the class is splittable and once for the splitting itself. So instead of line 11 and 12 of Listing 4.11 one would write

```
1 for all P_current ∈ P do
2   (P'_current, P''_current) = split(P_current, A, l)
3   if P'_current ≠ ∅ ∧ P''_current ≠ ∅ then
```

to get the best performance out of the algorithm. We decided not to write this directly into the pseudocode description above, since it is an implementation detail and could cause confusion at first reading.

Complexity The theoretical complexity of Hopcroft's Algorithm is well known as $O(|Q||\Sigma|log|Q|)$. Please take a look at the works by Knuth ([13]) or Berstel ([14]) for a justification of the complexity.
However, the complexity of the split algorithm above is interesting. Since we use BDDs as labels for our transitions, we first have to figure out which transition to take with the letter l. In the worst case, this means we have to check every transition of every state in the class P. If we assume, that checking if the letter l is accepted by a BDD is O(1), this gives us an upper bound of $|P| \cdot |\Sigma|$. If we account, that the split algorithm (or the similar isSplittable algorithm) has to be called $|P|$ times for each run of the outer while loop, we can see easily, that this will be the bottleneck of the algorithm.

4.2.3. Determination

The determine algorithm turns an NFA into a DFA. The usual approach to this algorithm is the power-set construction, which can be found in [4].
In this section two approaches to determinate a BDD-labelled NFA will be presented. They both use some kind of power-set-construction, but with different ways of getting there.
Since we have BDD-based transition tables instead of a letter-based transition tables in the original algorithm, we have to find a way of getting the right sets from the power-set of the state-space.

Guessing a letter

The first way of getting the right set is by guessing a letter.

```
 1 determinize:
 2 Input NFA A = (Q_A, Σ, δ_A, q_A,0, F_A)
 3 Output DFA B = (Q, Σ, δ, q_0, F)
 4
 5 W = { {q_A,0} }
 6 Q = { {q_A,0} }
 7 q_0 = {q_A,0}
 8
 9 while W ≠ ∅ do
10   Q_tmp =pick {q_1 , ... , q_n} from W
11   BDD notDone = One-BDD
12   while ¬notDone.isZero() do
13     Letter l = notDone.getOneSat()
14     newState = ∅
15     transition = One-BDD
16     for all q ∈ Q_tmp do
17       add s to newState where ∃b : δ_A(q, b) = s ∧ l ∈ b
18       transition = transition ∩ b
19     end;
20
21     if newState ∉ Q ∧ newState ∉ W then
22       if ∃q ∈ newState : q ∈ F_A then
23         add newState to F
24       add newState to Q
25       add newState to W
26
27     add (Q_tmp , transition , newState) to δ
28
29     notDone = notDone ∩ ¬transition
30   end
31 end
32
33 return (Q, Σ, δ, q_0, F)
```

Listing 4.14: Determination for BDD-labelled automata with letter guessing

The interesting part is from Line 10 to 29. After getting a subset of Q_A we make a new BDD called notDone. This will contain every letter that has not yet been dealt with. Since this is true for every letter at first we have to start with a One-BDD which returns true for every input. Following that we start a loop that goes on until no letter

in notDone is left, which means that notDone is the Zero-BDD.
Next we pick a letter l that is still in notDone and produce the subset for this letter, which means that we go through every state in the current set Q_{tmp} and see which transitions can be taken using our letter l. If a transition can be taken, the target is added to the newState set and the transition-BDD is intersected with the current transition. After finishing the inner loop, the transition-BDD contains all letters that can be taken to get to the newState set.
To finish up we add newState to the new state-space and the worklist, if not done already. Further the transition from Q_{tmp} to newState with the transition-BDD is added to δ.

Complexity The outer while loop will be met at most $|P(Q_A)|$ times, since every set from the power-set construction can only be met once. The inner while loop, which keeps track of the letters that were used already will be met at most Σ times, which would mean that each letter leads to a different set of states. The inner for loop from line 16 to 19 is tougher to calculate. For each state in the current set of states we have to check which transition contains the letter l. Since this is an NFA, we will have to check every transition for every state in the current set. Assuming that checking whether $l \in b$ is in O(1), this makes an upper bound of $|Q_{tmp}| \cdot |\Sigma|$.
In total we get a complexity of $\mathrm{O}(|P(Q_A)| \cdot |\Sigma| \cdot |Q_{tmp}| \cdot |\Sigma|) = \mathrm{O}(|P(Q_A)| \cdot |Q_{tmp}| \cdot |\Sigma|^2)$

Power-set iteration

The second approach does not go through the available letters but rather tries each set in the power-set construction, starting with the biggest.

```
 1 determine:
 2 Input NFA A = (Q_A , Σ , δ_A , q_A,0 , F_A)
 3 Output DFA B = (Q, Σ , δ , q_0 , F)
 4
 5 W = { {q_A,0} }
 6 Q = { {q_A,0} }
 7 q_0 = {q_A,0}
 8
 9 while W ≠ ∅ do
10   Q_tmp =pick {q_1 , ... , q_n} from W
11   BDD restrict = Zero-BDD
12
13   for all Q_sub ∈ P(reachableStates(Q_tmp)) do
14     transition = (⋀_{q∈Q_sub} ⋁_{BDDb:∃s∈Q_tmp:δ_A(s,b)=q} b) ∧ ¬restrict
15     if ¬transition.isZero() then
16       if Q_sub ∉ Q then
17         if ∃q ∈ Q_sub : q ∈ F_A then
18           add Q_sub to F
19         add Q_sub to W
20         add Q_sub to Q
21       add (Q_tmp , transition , Q_sub) to δ
22
23       restrict = restrict ∨ transition
24       if restrict.isOne() then
25         break
26   end
27 end
28
29 return (Q, Σ , δ , q_0 , F)
```

Listing 4.15: Determination for BDD-labelled automata with power-set iteration

The algorithm looks tough at first sight, but the idea behind it is quite simple. For each set S we pull from the worklist, the power-set of all states that are reachable from S is computed. Now we start iterating over each set in the power-set, where we have to make sure to pick the biggest available set first. Let's call this set Q_{sub}. Once we have picked we compute the transition from the current set to this set of reachable states using the formula

$$transition = (\bigwedge_{q \in Q_{sub}} \bigvee_{BDDb:\exists s \in Q_{tmp}:\delta_A(s,b)=q} b) \wedge \neg restrict$$

in line 14. The formula iterates over every state in Q_{sub} and tries to find transitions from Q_{tmp} to this state. These transition-BDDs are then put together using $\vee$. After that we compute the intersection of all those transitions to single states, which gives us the BDD for a transition from the Q_{tmp} to Q_{sub}. If this BDD is not the Zero-BDD, the transition from Q_{tmp} to Q_{sub} will be added to δ.
The restrict-BDD that is added to the end of the formula contains all letters that were taken for Q_{tmp} already. It is used to make sure, that no duplicate transitions can occur

in the resulting automaton. Once this BDD contains all letters, which means that the BDD is the One-BDD, the computation for Q_{tmp} is done and we can go on with the next item in the worklist.

Complexity For the same reason as above, the outer while loop is met at most $|P(Q_A)|$ times. The inner for loop will be met as most $P(Q_A)$ times, since the reachable states from Q_{tmp} could be equal to Q. Inside the loop, line 14 will be quite costly. We first iterate over Q_{sub} which could be equal to Q_A. Following that we look for transitions from states in Q_{tmp} to the current state from the iteration. This will be $|Q_A| \cdot |\Sigma|$. In total we get a complexity of $O(|P(Q_A)| \cdot |P(Q_A)| \cdot |Q_A| \cdot |\Sigma|) = O(2^{2 \cdot |Q_A|} \cdot |\Sigma|)$.

4.2.4. Negation

The implementation of a negation algorithm for BDD-labelled automata is fairly simple. First we will have to make sure that the automaton is deterministic. If it is not we will have to determine it using one of the determine algorithms from section 4.2.3. After that we will just flip the final states, so that each final-state will become a non-final state and each non-final-state will become a final-state. After that the padding closure will be applied to make sure every encoding of solutions is accepted. The padClosure algorithm just propagates final states along 0-transitions. So if a state is non-final, but has a 0-transition to a final state it will be final afterwards.

```
1 negate:
2 Input NFA A = (Q, Σ, δ, q0, F)
3
4 if ¬isDFA(A)
5   determine(A)
6
7 for all q ∈ Q do
8   if q ∈ F then
9     remove q from F
10  else
11    add q to F
12 end
13
14 padClosure(A)
```

Listing 4.16: Negation for BDD-labelled automata

The isDFA part is quiet easy to implement. We have to make sure, that the union of all transitions is the One-BDD, so the BDD that always returns true, while the intersection has to be the Zero-BDD, so a BDD that always returns false. Since the intersection of the current BDD b with any subset of One-BDD\b has to be the empty set we can check this part on the fly inside the inner loop which could safe time in some cases.

```
1 isDFA:
2 Input NFA A = (Q, Σ, δ, q0, F)
3
4 for all q ∈ Q do
5   BDD union = Zero-BDD
6   for all BDD b : ∃x : δ(q,b) = x do
7     if b ∩ union ≠ ∅ then
8       return false
9     union = union ∪ b
10  end
11  if ¬union.isOneBDD() then
12    return false
13 end
14
15 return true
```

Listing 4.17: isDFA for BDD-labelled automata.

Complexity Since the negation algorithm mainly relies on the determine algorithm it will not be covered. However isDFA could be interesting. The for loop is met $|Q|$ times. The inner for loop could be met $|\Sigma|$ times in the worst case. Assuming that the BDD operations have a complexity of O(1), we get a complexity of $O(|Q| \cdot |\Sigma|)$ for isDFA.

4.3. Solution Space

4.3.1. Number of solutions

The purpose of this algorithm is to extract the number of solutions out of a BDD-labelled DFA. The first part of the algorithm will check if the DFA has any loops from where final states are reachable. If this is the case the DFA has an infinite number of solutions. This has one exception, namely a loop in a final state from which no other final state is reachable and which only consists of the padding-letter which is just zeros. If no real loops were found part two of the algorithm comes to action. Since there are no loops outside of the trap state in the automaton, the automaton is a BDD-labelled BDD. As such we can count the number of solutions n as the sum of runs that end at each final state:

$$n = \sum_{s \in F} N_s$$

where F is the set of final states and N_s is the number of unique runs that ends at the state s. At this point we have to be very careful. If one of the incoming states is another final state, and the transition includes the letter 0^x, then this piece of the run can not be counted into N_s since we would otherwise have duplicate solutions and therefore a wrong result. However we still have to remember, that we did not count this piece of the transition for the case that there are transitions leaving the state to other non-final states from where other final states are reachable. For that reason we always have to carry two values, one value if we stop at this state, and one value if we go on.
To illustrate the use of the two values let's have a look at the automaton in figure 4.6.

If we just counted the incoming runs for each final state we would get 4 runs for q_3 and 1 run for q_1 and q_2. In total this would be 6 solutions. However, if we count the solutions

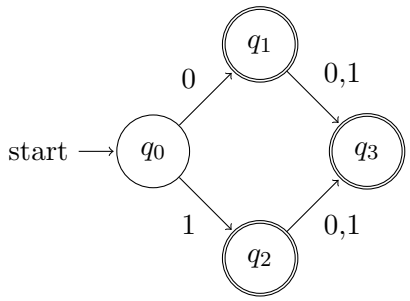


Figure 4.6.: Automaton to illustrate the numSol algorithm

manually we get only 4 of them, namely 0,1,2 and 3.
If we now use the algorithm described below we get 1 solution for q_1 and q_2. Since both incoming transitions to q_3 come from final states and include 0, we get only 2 new solutions instead of 4, which makes it a total of 4 solutions, which is correct.
Listing 4.23 shows a pseudocode implementation of the above described algorithm. It first checks if the automaton has any loops (taking into account the mentioned exception). This algorithm won't be described here, since every cycle detection algorithm known from graph theory can be used. Please check the work by Brent [7] and Floyd [8] for more information on this topic.

```
1 numSol:
2 Input DFA A = (Q, Σ, δ, q0, F)
3 Output: Number of solutions
4
5 if hasLoops(A) then
6   return ∞
7
8 cache = []
9 for all q ∈ F do
10   countPaths(q,cache)
11 end
12 solutions = 0
13 for all q ∈ F do
14   solutions += cache.get(q).getSecond()
15 end
16
17 return solutions
```

Listing 4.18: numSol algorithm which counts the solutions for a BDD-labelled DFA.

```
1 countPaths:
2 Input State q Table from states to 2-tuple of numbers
3 if cache contains q then
4   return
5
6 if q is initial then
7   add [q , <1,1>] to cache
8 else
9   goOn = 0
10  stopHere = 0
11  for all incoming transitions (s,bdd,q) where s ≠ q do
12    if cache does not contain tuple with s then
13      countPaths(s,cache)
14
15    inComingGoOn = cache.get(s).getFirst()
16    inComingStopHere = cache.get(s).getSecond()
17
18    if q is final and s is final then
19      goOn += inComingGoOn · allSatCount(bdd)
20      stopHere += inComingStopHere · allSatCount(bdd\{0})
21    else
22      goOn += inComingGoOn · allSatCount(bdd)
23      stopHere += inComingGoOn · allSatCount(bdd)
24  end
25  add [q,<goOn,stopHere>] to cache
```

Listing 4.19: Algorithm to count all paths to a state.

The algorithm looks quite big at first sight. However, if we take a closer look, it is fairly simple. We start from each final state and check if we have computed the number of unique paths for this state already. If not, we get the solutions for the states of the incoming transitions and multiply it by the number of solutions for the BDD. These numbers are added up to get the actual values for this state. Lines 18 to 20 of the countPaths algorithm solve the problem of the doubly counted solutions described above by leaving out the 0 letter for the stopHere value.

Complexity Every state in the automaton is visited only once. If we assume, that allSatCount is in O(1), we get a complexity of $O(|Q| \cdot |\Sigma|)$

4.3.2. allSat

This algorithm is used to get all solutions out of an automaton. Basically it is just a depth-first algorithm. First the algorithm checks if the automaton has any loops, except loops that are 0-loops where no final state other than the current state are reachable. If a loop exists, then there are infinitely many solutions, so the algorithm can't compute all of them. Therefore it will just return nothing.
If no loops were found, the algorithm will start the depth-first algorithm. Each branch will carry its own temporary solution set that is only valid for this branch. If any final state is met, the temporary solution set will be added to the real solution set. Since there are no loops except 0-letter self-loops in the automaton, the algorithm will terminate.
The main work is done by the allSatWalk algorithm. As argument it takes a state **q**, an integer **add**, a set of tuples of integers **tmpSolution** and a set of tuples of integers

solution. They have the following meaning:

- State **q**: The current state for the depth first algorithm, so the state from where the algorithm will take the next transition.
- Integer **add**: The number to add if a part of the transition is 1. So starting from the top node add will be 1, for states reachable from the initial state it will be 2, then 4 and so on.
- Set **tmpSolution**: Set of Integers that carries the temporary solutions of the current run
- Set **solution**: Set of Integers that carries the actual solutions of the automaton.

```
1 allSat:
2 Input DFA A = (Q, Σ, δ, q0, F)
3 Output: All solutions of this automaton
4
5 if hasLoops(A) then
6   return nothing
7
8 solution = {}
9 allSatWalk(q0 , 1 , {[0,...,0]} , solution)
10
11 return solution
```

Listing 4.20: allSat algorithm for a BDD-labelled DFA.

```
1 allSatWalk:
2 Input State q , Integer add , Set tmpSolution , Set solution
3
4 if q is final then
5   add all [a0 , ... , an] ∈ tmpSolution to solution
6
7 for all Bdd bdd , State s : δ(q,bdd)=s and s ≠ q do
8   newTmpSolution = {}
9   for all [b0 ,...,bn] ∈ bdd do
10    add [a0 + add·b0, ... , an + add·bn] for each [a0 ,...,an] ∈ tmpSolution
          to newTmpSolution
11  end
12
13  allSatWalk( s , 2·add, newTmpSolution, solution);
14 end
```

Listing 4.21: Depth-first allSat algorithm for a BDD-labelled DFA.

Complexity The complexity of the algorithm equals the worst-case complexity of an usual depth-first search. Since every transition and every state have to be considered the complexity is $O(|Q| + |\delta|)$, where $|\delta|$ is the number of edges in the automaton.

4.3.3. N-Sat

The N-Sat algorithm tries to extract n solutions from a DFA. Thereby it takes a different approach than the allSat algorithm. Instead of doing a depth-first search, this algorithm performs a breadth-first search. By doing so, small solutions are preferred over big solutions. Every time it hits a final state it adds the solution for the current branch to the solution set. Once there are n solutions in the solution set, the algorithm will just stop. The worklist will contain 3-tuples of data. The first part will be the state, the second part the current numbers for the run and the third part will be the number which is added depending of the length of the run. So it will add 1 for the first transition, 2 for the 2nd and 2^{n-1} for the n-th transition. While doing the breadth-first search, the algorithm will always keep track of the worklist size. The algorithm will try to keep the worklist-size approximately as big as the number of solutions we want to get.

```
1 nSat:
2 Input  DFA A = (Q, Σ, δ, q0, F) , Integer num
3 Output n solutions to the automaton
4
5 solutions = ∅
6 W = <<q0 , [0,...,0] , 1>>
7 while W ≠ ∅∧|solutions| < num do
8   pick first <q,[a0,...,an] , add> from w
9   if q is final then
10    add [a0,...,an] to solutions
11  if |w| < num then
12    for all Bdd bdd , State s:δ(q,b)=s do
13      for all [b0,...,bn] ∈ bdd do
14        add last [s,[a0+b0·add,...,an+bn·add],2·add] to W
15      end
16    end
17 end
18
19 return solutions
```

Listing 4.22: N-Sat algorithm

The algorithm should not be called if the number of solutions of the automaton is bigger than the number of the solutions we want to get. In this case the algorithm might not terminate.
If there exist less or equally many solutions than we want to get the algorithm will terminate.

Complexity In the worst case, the automaton has exactly one final state and one path from the initial to the final state. From this final state, there is another transition to the initial state. Therefore we have to loop n times to get n solutions. This makes an upper bound of $n \cdot |Q| \cdot |\delta|$, where n is the number of solutions we want to get, $|Q|$ is the size of the state space and $|\delta|$ is the number of transitions in the automaton. Therefore the algorithm is in $O(n \cdot |Q| \cdot |\delta|)$. However the average case complexity will be much better than the worst case complexity.

4.3.4. Optimized Solutions

The algorithm for optimized solutions mainly uses the algorithms that were shown already.
The algorithm will get a term that the user wants to optimize as well as the solution automaton A to the original formula that the user wants to solve. Let's call the term to be optimized P with the variables x_0 to x_k. Every variable x_i that occurs in P has to be a free variable in A. Now we will compute the automaton for the formula $z = P$, where z is a variable that does not occur in A. If we want to minimize P, we will have to compute the automaton $z = -P$ as well since we also have to cover negative values. Let's call these automata B and B_{min}
After that we will construct the intersection of A and B resp. A and B_{min} and call them C resp. C_{min}. Next we define the automata D and D_{min} as the automata where existential quantifiers for all free variables in A have been applied to C resp. C_{min}.
In the last step we will get the optimal solution. The exact algorithm for getting the maximum or minimum value will not be covered in detail here since they are fairly simple. Getting the minimum can be implemented using a breadth-first search for a final state. To get the maximum, an adapted version of the allSat algorithm from section 4.3.2 can be applied.
The optimum for the minimal solution will be min{ getMin(D) , -getMax(D_{min}) }, the optimum for the maximal solution getMax(D).
The algorithm will then return all runs of C or (C_{min} if the optimum is negative) for z = optimum.

```
1 optimalSat:
2 Input  DFA A = (Q, Σ, δ, q0, F) Predicate p Type t ,
3 Integer[] optimal solutions
4
5 DFA B = solve(z = p)
6 if t = Min then
7   DFA Bmin = solve(z = −p)
8
9 DFA C = A ∩ B
10 if t = Min then
11   DFA Cmin = A ∩ Bmin
12
13 DFA D = C
14 if type = Min
15   DFA Dmin = Cmin
16 for all x ∈ free variables in A do
17   D = D.exists(x)
18   if type = Min then
19     Dmin = Dmin.exists(x)
20 end
21
22 if type = Min then
23   optimal = min {getMin(D) , −getMax(Dmin) }
24 else
25   optimal = getMax(D)
26
27 if optimal >= 0 then
28   return runs of C for z = optimal
29 else
30   return runs of Cmin for z = −optimal
```

Listing 4.23: Algorithm to get optimized solutions out of an automaton.

Complexity The complexity of the algorithm relies completely on the complexity of the Presburger arithmetic, which is doubly exponential. For a justification please take a look at the work by Fischer and Rabin [6].

4.4. Solver

4.4.1. Solve a logic tree

This section will describe how to compute the solution automaton to a given logic tree, which was described in section 4.1. The algorithm will start at the top node of the tree and check what kind of node this is.
If it is an existential quantifier or a negation the algorithm calls itself with the child to get the automaton for the child.
In the case of the existential quantifier, all it does is apply an existential quantifier to every BDD-label for the given variable and do a padClosure on the resulting automaton. The padClosure simply looks for non-final states with a 0-transition to a final state and makes those non-final states final. This makes sure every encoding of a solution is accepted.
In the case of the negation, the returned automaton will just be negated using the algo-

rithm described in section 4.2.4.
If the node is an AND or an OR node, we have to check the children in order to decide the behaviour. The children are first put into two separate groups, where the first group are leafs of the tree (which means they have to be predicates) and the second group are nodes (which means they are not predicates).
The automaton for the first group is then computed using the algorithm from section 4.4.2. After doing that, the second group is handled. For each node in the group we start by computing the automaton for the node by calling the algorithm recursively. After that we compute the union (for OR nodes) or intersection (for AND nodes) of the automaton so far with the automaton that we just computed.

```
1 solveAll:
2 Input  node n
3 Output DFA A
4
5 switch n.Type
6   case EXISTS: return exists(solveAll(n.child) , n.var)
7   case NEGATE: return negate(solveAll(n.child))
8   case AND:
9     X = {m | m ∈ n.children∧ m is Predicate}
10    Y = n.children\X
11    if (X ≠ ∅) then
12      A = solvePredicates(X , AND)
13      for all v ∈ Y do
14        A = A ∩ solveAll(v)
15      end
16    else
17      for all v ∈ Y do
18        A = A ∩ solveAll(v)
19      end
20
21    return A
22  case OR:
23    X = {m | m ∈ n.children∧ m is Predicate}
24    Y = n.children\X
25    if (X ≠ ∅) then
26      A = solvePredicates(X , OR)
27      for all v ∈ Y do
28        A = A ∪ solveAll(v)
29      end
30    else
31      for all v ∈ Y do
32        A = A ∪ solveAll(v)
33      end
34
35    return A
```

Listing 4.24: Algorithm to compute the solution automaton to a given logic tree.

Since the complexity completely relies on other algorithms, it will not be covered here.

4.4.2. Solve linear predicates

This algorithm solves arbitrarily many predicates, that are connected with AND or OR all at once. It relies greatly on the algorithm from Boudet and Comon which was first described in [5]. The algorithm in their paper is used to solve formulae of the form

$$\bigwedge_{0 \leq i \leq n} a_{i,1}x_1 + ... + a_{i,m}x_m = b_i$$

We want to extend this algorithm to solve formulae which include $\vee$ for the logic part and $\neq$ and $\leq$ for the numeric part. As such we support the following forms:

$$\bigwedge_{0 \leq i \leq n} a_{i,1}x_1 + ... + a_{i,m}x_m \odot b_i$$

$$\bigvee_{0 \leq i \leq n} a_{i,1}x_1 + ... + a_{i,m}x_m \odot b_i$$

where $\odot$ can be $=$, $\neq$ or $\leq$.

In order to extend the existing algorithm to the cases above, we merge it with the algorithms to solve arbitrary predicates. These algorithms can be found in [4] in the chapter on Presburger arithmetic.

Since the complete algorithm is fairly big and most parts should be known to the reader already, only the interesting parts will be covered here.

In the original algorithm by Boudet and Comon, states were an n-tuple of numbers which represent the residuals of each of the n predicates. If each number in the n-tuple was 0, the state was final. If, after taking a transition , any number in the n-tuple mod 2 was different from 0, this transition had to lead to the trap state since an odd residual can not be resolved any more.

We extend this approach to cover all three operations. If the i-th predicate has the operation $=$, the i-th number of the tuple has to satisfy the properties as in the original algorithm. If the operation is $\neq$, the i-th number in the tuple has to be different from 0 and if the operation is $\leq$, the i-th number of the tuple has to be greater or equal to 0.

This covers all operations that are linked by $\wedge$. If we also want to cover $\vee$, we have to change the condition for final states. In the AND case every number in the n-tuple had to satisfy the condition given by the predicate operation in order to become final. In the OR case only one of the numbers in the n-tuple has to satisfy the given condition. If any predicate has the operation $=$ and runs into a case where it would go to the trap state in the AND case, it will be marked as not solvable. This part of the formula will just be ignored after taking that transition.

Example In this example we want to solve the formula $x = 3 \wedge y \leq 5$. The starting point will be 3,5. From there, the transitions 10 and 11 go to 1,2 since $\lfloor \frac{3-1}{2} \rfloor = 1$, $\lfloor \frac{5-1}{2} \rfloor = 2$ and $\lfloor \frac{5-0}{2} \rfloor = 2$. 00 and 01 leads to the trap state, because 3-0 mod $2 = 1$, so this formula can not be solved any more after taking that transition. The other transitions should be self-explaining.

The states 0,1 and 0,0 are final states since they both satisfy the conditions given by the predicates, which are equal to 0 for the first number and greater or equal to 0 for the second number.

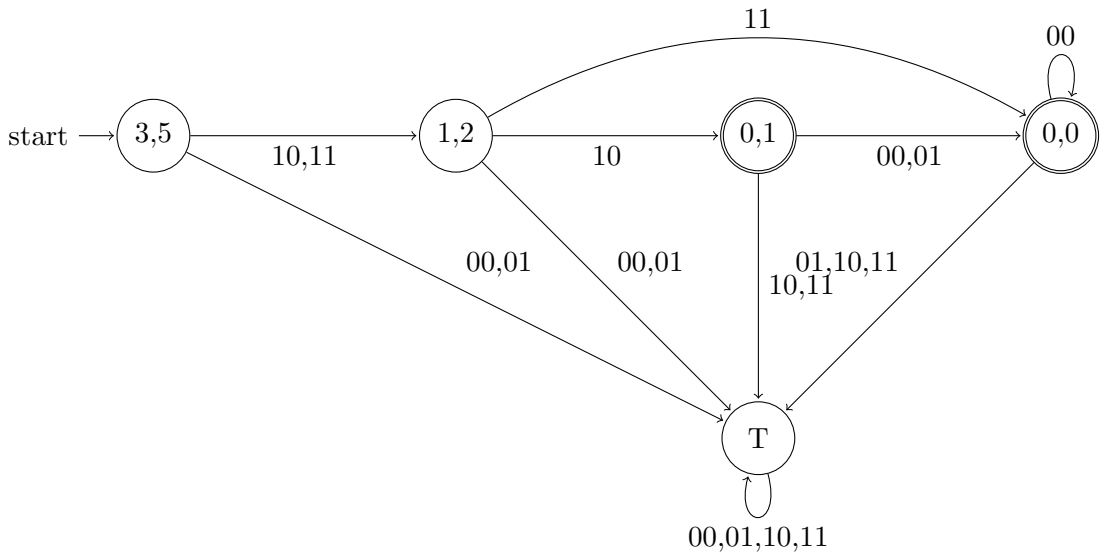


Figure 4.7.: Solution automaton for x = 3 ∧ y ≤ 5

5. Bounded Presburger arithmetic

Bounded Presburger arithmetic is a version of the usual Presburger arithmetic, where there is only limited space for each variable. As such, there are only finitely many solutions for each formula. Therefore BDDs are sufficient to capture the full complexity of the bounded Presburger arithmetic and will be used as the data structure. The aim of this chapter is to outline a method to solve formulae in the context of bounded Presburger arithmetic.
The solver works almost as for usual Presburger arithmetic. Conjunction of two formulae turns into intersection of two BDDs, disjunction of two formulae are turned into union of two automata and so on. The only difference lies within solving the predicates. When we solved conjunction/disjunctions of predicates, we solved them all at once as described in 4.4.2. Although a similar algorithm can be found in this case, the predicates will be solved separately, since it's easier to understand. To implement con-/disjunction, the results are intersected/unioned in the end.
When using BDDs to solve formulae we have to consider, that the labels of the BDD are limited to 0 and 1. Therefore we have to introduce a separate variable for each bit of every variable. So assume the variables x and y are used in the predicate, and the number of bits is limited to 32, the variable space of the BDD would be $\{x_0, y_0, x_1.y_1, ..., x_{31}, y_{31}\}$. This has to be considered when existential quantifiers are applied to the BDD. Instead of applying the quantifier for e.g. x we will have to apply the quantifier for $q_0, ..., q_{31}$ (for 32 Bit).
Since union, intersection, existential quantifier and negation are already available for BDDs, it suffices to give an algorithm to solve a predicate.

5.1. Solving a predicate

Predicates of the form

$$\sum_{i=1}^{m} a_i \cdot x_i = b \tag{5.1}$$

or

$$\sum_{i=1}^{m} a_i \cdot x_i \leq b \tag{5.2}$$

will be covered. All other operations such as $\neq$ or $<$ can be implemented using the above forms, conjunctions and negations.
The algorithm works in n steps where n is the number of available bits for each variable. Each step will add m levels to the BDD, where m is the number of variables in the predicate. Every step starts with a number of labelled nodes, where the labels represent the current remaining value for b. At the start, there will be one node with the bound of the predicate as label.
For every step, the algorithm will go through all variables of the predicate. Of course a

fixed order on the variables is needed for this to work. For each variable, one level will be added to the BDD. From each node in the precedent level, two transitions are taken, one for 0 and another one for 1. The resulting new nodes have labels as follows: For the 1-transition, the prefix for the current variable is subtracted from the label of the current node in the precedent level. For the 0-transition, the label of the current node in the precedent level will just be copied. The nodes with these labels will then be added to the current level. Please note, that nodes on a level have to be unique. So if a node with label l exists already and the algorithm tries to add a new node with label l, the existing node will be returned.
When the end of a step is reached, this means that the algorithm has computed the possible outcomes for one bit of each variable. Since the next bit has double value, the labels of the lowest level have to be halved. Here two cases have to be considered.
In the first case, the operation of the predicate we want to solve is $=$. In this case the label has to be an even number. If it is, the label will just be halved. If not, the predicate can not be solved any more. Therefore the current node will be deleted and all incoming transition lead to the trap node of the BDD.
The second case is the operation $\leq$. In this case we just take the floor of the halved label as motivated in [4].
After halving the nodes, a merge operation is applied, so that all nodes on the same level that have similar labels are merged into one node.
After that the algorithm will try to optimize the added levels of the BDD. This means, if a transition carries both, 0 and 1, the precedent node can be just skipped and therefore deleted.
When the algorithm is done with every step, which means all levels for every variable have been computed, the last level is checked again. Once more we have to consider two cases: $=$ and $\leq$.
In the $=$-case, all nodes on the bottom level which have label 0 will be deleted and all incoming transitions lead to the accepting node. All other nodes will be deleted as well with their transitions being led to the trap node.
In the $\leq$-case all nodes will be deleted, too. The transitions to nodes with label ≥ 0 will be led to the accepting node, all other to the trap node.

```
1 solvePredicate
2 Input: Integer bits Variables v Integer prefixes Integer bound Operation
      op
3 Output: BDD b
4 node = add Node with label bound on Level 0 to b
5
6 currentLevel= {node}
7 for i=0;i<n; i++ do
8   for all var ∈ v do
9     newCurrentLevel  = {}
10    for all node ∈ currentLevel do
11      level = level of n
12      l = label of n
13      newLabel = l - prefix of var
14      newNode = add Node with label newLabel on Level level+1 to b
15      add transition (node , 1 , newNode) to b
16      add newNode to newCurrentLevel
17      newNode = add Node with label l on Level level+1 to b
18      add transition (node , 0 , newNode) to b
19      add newNode to newCurrentLevel
20    end
21    currentLevel = newCurrentLevel
22  end
23
24  for all node ∈ currentLevel do
25    l = label for node
26    if  l is odd ∧ op is EQUAL then
27      delete node and lead all incoming transition to the trap state
28    else
29      change label of node to ⌊label/2⌋
30  end
31
32  optimize b
33
34 end
35
36 for each node ∈ currentLevel do
37  if op = EQUAL then
38    if label of node = 0 then
39      delete node and lead all incoming transitions to accepting state
40    else
41      delete node and lead all incoming transitions to trap state
42  else
43    if label of node ≥ 0 then
44      delete node and lead all incoming transitions to accepting state
45    else
46      delete node and lead all incoming transitions to trap state
47 end
48
49 optimize b
50
51 return b
```

Listing 5.1: Algorithm for solving predicates in the bounded Presburger arithmetic. Please note, that duplicate nodes won't be added on a level, so if a node with the same label is added twice to the BDD, there will be only one node with this label on this level.

This approach solves only one predicate at a time. However, it should be feasible to extend the approach to solve multiple predicates which are assembled by con-/disjunction using the method from [5] which is also used in section 4.4.2.

Part II.

Implementation and Conclusion

6. Architecture

In this chapter the overall architecture of the software is described. It will start by explaining the front end, which is the graphical user interface as well as the communication to the server. After that the back end, which includes the grammar for the parser, the optimizations of the formulae, the actual solver as well as getting a solution space that the user ordered will be explained in detail.

6.1. Front end

The front end is build as a web application, using HTML,CSS and JavaScript. Therefore an up-to-date CSS3-enabled web browser is needed to use the front end. Strong JavaScript performance is recommended.

6.1.1. GUI

The goal of the GUI is to provide a simple way for the user to communicate with the solver. The user should be able to work with the GUI without reading a manual before. As such the GUI is held very simplistic. A screen shot is shown in figure 6.1.
On top of the screen you can see the tab-bar. When the interface is launched, there are two tabs. The first tab is a tab for a formula that the user wants to enter, the second is the macro manager where the user can enter macros to use in his formulae. Right next to the macro-tab is the +-button which adds a new formula tab.

The formula tab

When a formula tab is opened, the input screen is shown. It consists of a text area in the middle of the screen. Above it are a some buttons to help the user to enter his formula into the text area. Each button is labelled with its function, so the $\exists$-button enters the syntax for an existential quantifier, $\wedge$ enters the syntax for $\wedge$ and so on.
Besides the buttons, there is also a select menu, which lists all existing macros that the user entered before in the macro tab. By clicking on any one of them the chosen macro is entered to the text area.
Below the text area is a radio button menu where the user can specify properties of the solution space. There are four possibilities:

- Get n solutions: The user can specify a number of solutions, that he wants to get.
- Get all solutions: All solutions of the formula are computed if possible.
- Maximize a term: The user can enter a term, which uses free variables from the formula above. The solver then tries to find solutions which maximize the term.

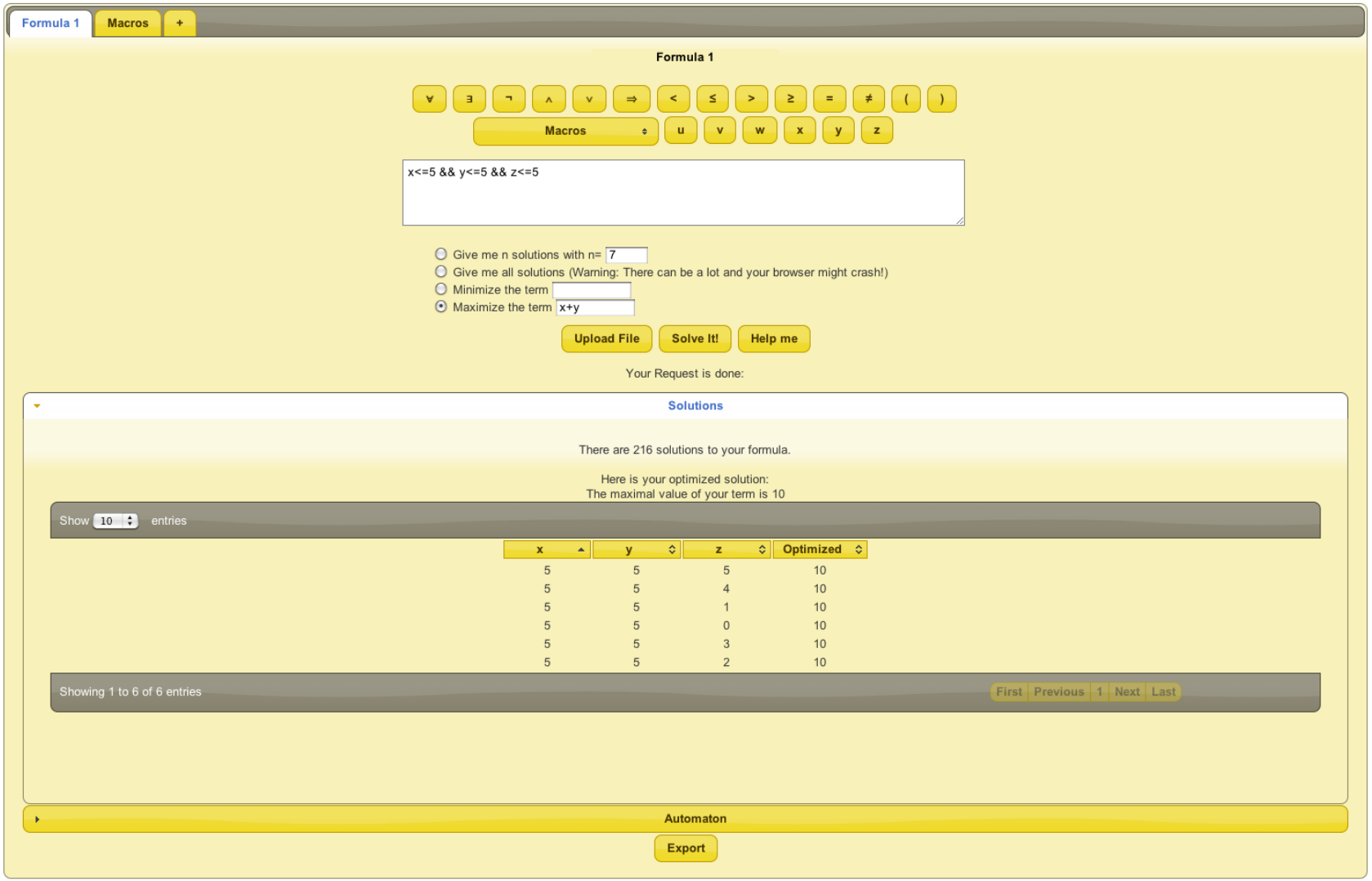


Figure 6.1.: Screen shot of the GUI of the solver with the formula input on the top, and the results on the bottom. In this case the formula $x \leq 5 \wedge y \leq 5 \wedge z \leq 5$ is solved. In addition, the term x+y is being maximized in the solution section on the bottom.

- Minimize a term: The user can enter a term, which uses free variables from the formula above. The solver then tries to find solutions which minimize the term.

Finally there are three buttons:

- Upload a File: Opens a dialogue to upload a file with formulae. For each formula in the file a tab is opened.
- Solve it!: Tries to solve the formula.
- Help me: Opens an overlay which contains a simplified grammar of the formulae that the user can enter, as well as some simple examples.

After pushing the solve button, the formula is sent to the server which tries to solve it. Please see chapter 6.1.2 for details on the communication with the server.
If the server does not succeed with solving the formula an error message is shown to the user. If a syntax error was observed, the assumed source of the error is also marked in the text area.
If the server succeeds in solving the formula, the results are shown below in an accordion, which is shown in figure 6.1 on the bottom of the screen shot. Initially the solutions part of the accordion is shown where the user can watch the computed solutions. By clicking on the name of the columns, the solutions can be ordered.
The automaton part of the accordion shows the minimal DFA for the formula. The BDD-labels of the DFA will be transformed into logical formulae in conjunctive normal form (CNF).

The macro window

When opening the macro window first, there is only one input area and some buttons. The input area is used to enter a new macro. When clicking the save button right next to the input area the macro is sent to the server. If there are any difficulties, an error will be shown. If another macro with the same name already exists, the user will be asked whether the old macro should be overwritten or not.
Apart from the send button there are a few more:

- Add Macro: Adds a new input area to input another macro;
- Clear: Deletes all macros for a 'fresh start';
- Upload File: Opens a dialogue to upload a text file that contains macros. For each macro in the text file a new input area will be added and filled with the macro from the file;
- Export: Exports all macros into a text file.

All macros will be saved within the users session. Therefore all macros will be available for the session life cycle time, which depends on the server settings. The default setting is set to 30 minutes.

request	meaning	other required parameters
new	add a new formula to the queue	equation, clientId, solType, numSol,term
status	check the status of a given request	id
test	check the syntax of a given formula	equation, clientId, solType, term
testMacro	test if the given macro has the right syntax	macro
clearMacros	deletes all macros in the users session	-
getMacros	return all macros in the users session	-
addMacros	adds a macro to the MacroManager if possible	macro

Table 6.1.: Different request types and the reaction by the server.

6.1.2. Communication with the Server

All communication with the server is done using asynchronous HTTP requests, commonly known as AJAX (asynchronous Javascript and XML). Using this technique prevents reloading the web page at any point and reduces traffic on the web server since for every request only necessary data is sent. The server will return JSON-Strings with the data that has been requested.
Each AJAX call from the client has a request parameter which determines what kind of request this is. Table 6.1 shows possible parameters. The last column contains the other parameters that are required for this type of request. The meaning of these parameters will be described next.

- **equation**: Contains the formula from the text area;
- **clientId**: The ID of the tab on the client-side;
- **solType**: The type of solution space that the user wants, i.e. N for n solutions , All for all solutions , Min for the minimization of a term and Max for the maximization of a term;
- **numSol**: if solType is N, numSol specifies the number of solutions;
- **term**: if solType is Min or Max, term specifies the term to maximize/minimize;
- **macro**: contains the macro that the user wants to add.

6.2. Back end

The back end of the software relies greatly on the worker pattern. Every time a new request from a user comes in, a SolverRequest, which is described in section 6.2.1 is generated and lined up into a queue.
The worker system consists of two workers. The first worker takes SolverRequests from the queue and handles them using the software shown in figure 6.2. While handling the input, the solver will update the SolverRequest every time new information is available. This way, the latest data is always available in the SolverRequest. After solving the formula, the worker will create a new Thread which then generates the image of the automaton, the JSON-representation of the automaton, the file for the export function and so on. That way costly operations such as creating the image will not delay the worker.
The second worker monitors the first worker. If the first worker does not give a notify in a certain timespan (which is 30 seconds by default), it will get killed. The user will then get notified, that the solver ran into a time-out.
Please note, that the timespan for the time-out is just a rough estimate. In the worst case, the time-out might take up to 1.999... times the timespan to get recognized. This way computation time can be saved for the second worker, while the target, which is the recognition of crashes of the system is still achieved.
The solver used in the first worker consists of four separate parts, namely the lexer, the parser, the LogicTree optimization and the actual solver which solves the formula and gets the solutions. Figure 6.2 shows how these components play together. In the following all parts of the solver will be explained.

6.2.1. Solver Request

The SolverRequest is the most important object in the process from the text input to the finished request. The object contains all information about the request. A new SolverRequest is generated each time a new request reaches the server. After that it is put into a queue where it waits to be handled.
The SolverRequest contains the following information:

- id: The server-side id of the request;
- clientId: The client-side id of the request, i.e. the tab number;
- equation: The formula to be solved;
- sessionId: The session id of the user;
- done: Flag indicating whether the request is done or not;
- timeout: Flag indicating whether the request ran into a time-out;
- exception: Contains the error description if any occurred;
- solType: Contains the requested solution type;
- solTerm: Contains the term that the used wants to optimize;

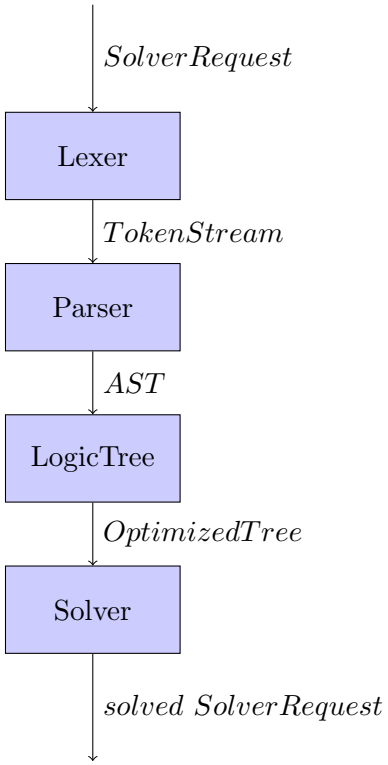


Figure 6.2.: Rough architecture of the program which is run within the first worker, including all steps needed to parse and optimize a formula.

- numSol: Contains the number of solutions that the user wants to get;
- macroManager: Contains the macro manager which is used to keep track over the macros for this user;
- solutionSpace: Contains the solutions to the solved formula;
- optimizedSolution: Contains the optimized solutions for the solved formula;
- automaton: Contains a string representation of the automaton with CNF-formulae on the transitions, in the dot format[1];
- numberOfSolutions: Contains the number of available solutions to the solved formula;
- isTautology: Flag to indicate whether the formula is a tautology;
- isOxymoron: Flag to indicate whether the formula is an oxymoron;

[1]Please visit `http://www.graphviz.org/` for more information on the dot format.

6.2.2. Lexer/Parser

The transformation of a formula in text-format to a formula in tree-format is done by the lexer and the parser. They both rely on ANTLR[2], which generates a lexer and parser from a given grammar. As the lexer and parser itself are implemented by ANTLR, this work will only discuss a slightly simplified grammar.

The grammar

First we will define the lexer rules, which are used to define a TokenStream that is then analysed by the parser.
The first rule defines a quantifier, which is either 'A' (for ∀)or 'E' (for ∃) followed be a arbitrary number of variables and ending with ':'. The second rule defines the beginning of a macro call which is basically the macros name followed be an opening bracket. The name of a macro has to start with an upper case letter followed by arbitrary many lower case letters.
The third rule defines a variable. The basis for a variable is a single lower case letter which can be followed by arbitrarily many array definitions written in square brackets, e.g. x[1] or x[a+1]. The variables inside an array definition are used for the 'forand' and 'foror' rules which will be described later.
The fourth rule defines an integer which is just one or more numbers.

```
1 QUANT
2     : 'A'VAR+':'
3     | 'E'VAR+':'
4 ;
5
6 MACRO   :   ('A'..'Z')('a'..'z')*'(';
7
8 VAR   :   ('a'..'z')('['  (('a'..'z')('+' INT)? | INT)  ']')*;
9
10 INT  : ('0'..'9')+;
```

Listing 6.1: Lexer rules for the grammar

The next target is to define rules for linear predicates which can also be stacked, e.g. $x \leq y \leq z + 4 = 20$.
For that to work we first need to define linear terms:

[2]Please see `http://www.antlr.org/` for more information on ANTLR.

```
1 linmon
2    : '+'? INT VAR VAR? -> ^(PLUS["+"]  INT VAR VAR?)
3    | '-'  INT VAR VAR?-> ^(MINUS["-"] INT VAR VAR?)
4    | '+'? INT  -> ^(PLUS["+"]  INT )
5    | '-'  INT  -> ^(MINUS["-"] INT )
6    | '+'? VAR VAR?     -> ^(PLUS["+"] INT["1"] VAR VAR?)
7    | '-'  VAR VAR?     -> ^(MINUS["-"] INT["1"] VAR VAR?)
8 ;
9
10 sgnlinmon
11   : '+' INT VAR VAR? -> ^(PLUS["+"]  INT VAR VAR?)
12   | '-'  INT VAR VAR?-> ^(MINUS["-"] INT VAR VAR?)
13   | '+' INT  -> ^(PLUS["+"]  INT )
14   | '-'  INT  -> ^(MINUS["-"] INT )
15   | '+' VAR VAR?     -> ^(PLUS["+"] INT["1"] VAR VAR?)
16   | '-'  VAR VAR?     -> ^(MINUS["-"] INT["1"] VAR VAR?)
17 ;
18
19 comp
20   : '==' -> ^(EQ["eq"])
21   | '!=' -> ^(NEQ["neq"])
22   | '>=' -> ^(GEQ["geq"])
23   | '<=' -> ^(LEQ["leq"])
24   | '>'  -> ^(GT["gt"])
25   | '<'  -> ^(LT["lt"])
26 ;
```

Listing 6.2: Parser rules for comparing operations and parts of linear terms.

linmon and sgnlinomn basically define the same structure, whereas linmon does not need the + sign in front if it is positive. This is the case because linmon will later be the first part of each linear term. The left side of both rules defines the structure that will be matched. The second variable is again just for the 'forand' and 'foror' functions. The right side then defines the output of the rule, which is basically PLUS or MINUS followed by a number and between 0 and 2 variables depending on the input.
Apart from that comparing operations are defined by the comp rule.

Now defining linear terms and linear predicates is very easy. A linear term is just a linmon followed by arbitrarily many sgnlinmon and a linear predicate is a linear term followed arbitrarily many alternating comparing operations and linear terms.Every linear predicate has to end with a linear term.

```
1 linearpred
2   : linearterm (comp linearterm)+ -> ^(PRED linearterm (comp linearterm)
        +)
3 ;
4
5 linearterm
6   : linmon (sgnlinmon)*
7 ;
```

Listing 6.3: Parser rules for linear terms and linear predicates

Finally we are ready to define rules for actual formulae.

```
1 formula
2   :   linearpred
3   | '!' expr -> ^(NEG["not"] expr)
4   | '&&' ('[' VAR '=' num ".." num ']')+ expr -> ^(FORAND (VAR num num)
      + expr)
5   | '||' ('[' VAR '=' num ".." num ']')+ expr -> ^(FOROR (VAR num num)+
      expr)
6   | QUANT expr          -> ^( QUANT expr)
7   | MACRO linearterm (',' linearterm)* ')' -> ^(MACRO (PRED linearterm)
      +)
8 ;
9
10 expr
11   : pre
12   | pre ("&&" expr)+ -> ^(AND["and"] pre expr+
13   | pre ("||" expr)+ -> ^(OR["or"] pre expr+
14   | pre "->" expr -> ^(IMPL["impl"] pre expr
15   | pre "<->" expr -> ^(BIIMPL["biimpl"] pre expr
16 ;
17
18 pre
19   : '('expr')'->^(expr)
20   | formula -> ^(formula)
21 ;
22
23 f
24   : expr EOF -> ^(expr)
25 ;
```

Listing 6.4: Parser rules for linear terms and linear predicates

The starting rule for the process of parsing a formula will be the rule f, which is defined as an expression followed by the end of input (EOF). The expression is the tricky part, which is why there is only a simplified version of it in this paper. The actual version is quite technical and can be seen in the full version of the grammar in appendix A.
The expr rule always starts with a pre, which will be described in a second. Following the pre can be arbitrarily many $\wedge$ or $\vee$ instances , an implication/bi-implication or just nothing.
The pre rule plays a major role in keeping track of the parenthesis. If it finds an opening bracket in the beginning and a closing bracket in the end it will call the expr rule on the part between the brackets. Else the input has to match the formula rule.
The most interesting part of the formula rule is in line 4 and 5. They define the 'forand' and 'foror' rules mentioned earlier. The forand rule starts with a '&&' and is followed with at least one variable definition over a range. This range is defined in square brackets as follows: [i=a..b], where i can be is any variable and a/b can be either a number or another variable defined in a forand (possibly even this one). Following the definitions has to be an expression. The expression will then be initialised with any combination of values for the variables.

Example The formula

$$\&\&[i = 1..2][j = i..3]\ x[i] <= j$$

is short for

$$x[1] <= 1\ \&\&\ x[1] <= 2\ \&\&\ x[1] <= 3\ \&\&\ x[2] <= 2\ \&\&\ x[2] <= 3$$

The 'foror' rule in line 5 will do just the same but with $\vee$ instead of $\wedge$.
The other parts of the formula rule define macro calls, quantifies, negations as well as just plain linear predicates and should be pretty easy to understand. One important aspect to note is, that the arguments for a macro call are not just simple variables, but linear terms. That makes it possible to define properties of whole linear terms instead of just variables.

6.2.3. LogicTree

The LogicTree part of the architecture is easy. The resulting tree from the parser is translated into an equivalent tree in the LogicTree format, which is nearly the same as the AST format returned by the parser. It only differs in the fact that macros, forand, foror ,$\forall$ quantifiers, implications, bi-implications and stacked linear predicates are automatically resolved. This leads to a tree which only contains $\wedge$,$\vee$,$\exists$ quantifiers, $\neg$ and linear predicates. This makes it easier to do optimizations on the tree.
Optimization is then done by the algorithms described in the sections 4.1.1 and 4.1.2.

6.2.4. Solver

The solver does the main work of the program. It computes the minimal automaton for the given LogicTree, which was computed in the step before. Hereby it uses the algorithms described in the sections 4.4.1 and 4.4.2.
Since chapter 4.2.3 proposes two different ways of implementing determination of automata, the algorithm that suits our needs best had to be chosen. To find out which one is the better fit, both algorithms were implemented and benchmarked on existing test cases as well as the actual benchmarks for the software. The test-cases were considered to get a bigger number of formulae. The mean difference between the letter-guessing method and the power-set method was 27ms in favour for the letter-guessing method. Both algorithms are still contained in the code and can be switched using the DET_ALG constant, which is defined in the PresburgerAutomaton class.
Instructions on how to get the benchmarks and test-cases can be found in the appendix B. The complete data from the benchmarks and test-cases can be found in appendix C.

Remarks on the bounded Presburger arithmetic Unfortunately the algorithm for bounded Presburger arithmetic could not be implemented in this solver, since no access to the BDD structure of the used library could be gained. The implementation of the described algorithm with a different library will be left open.

6.2.5. Getting solutions

In order to pick the right algorithm and provide the user with information about the number of solutions to their formula, the program will start by calculating the number

of available solutions and save it to the SolverRequest.
To get the appropriate solutions the program will then check the SolverRequest to get the type of the solution space that the user ordered. After that it will run one of the three described algorithms from chapter 4.3. The mapping from the type to the algorithm with regard to the number of available solutions is shown in table6.2.5.

Type	Number of solutions	Algorithm
N	request:numSol $<$ result:numSol	N-Sat from chapter 4.3.3
N	request:numSol $\geq$ result:numSol	allSat from chapter 4.3.2
All	result:numSol $\in \mathbb{N}$	allSat from chapter 4.3.2
All	result:numSol $\notin \mathbb{N}$	-
Min,Max	result:numSol $\in \mathbb{N}$	optimize from chapter 4.3.4
Min,Max	result:numSol $\notin \mathbb{N}$	-

Table 6.2.: Mapping from the ordered type of solution to appropriate algorithm.

6.2.6. Macros

Since macros were only mentioned shortly in the section on the LogicTree, they will be explained a bit more detailed here. Every time an user inputs a macro definition, the software will compute the LogicTree for this formula and save it in the MacroManager which is unique for each user. The software will also check if the parameters on the left hand side of the definition are equal to the free variables in the computed LogicTree. If not an error will be sent to the user.
When a macro is called in a formula, the software will replace the macro by the LogicTree for the macro, which is saved in the MacroManager. The free variables will be replaced by the terms that the user entered as parameters. If any variable in these terms equals a bound variable in the LogicTree for the macro, the bound variable will just be renamed. This way the user can call macros with arbitrary linear terms to define properties of these terms.

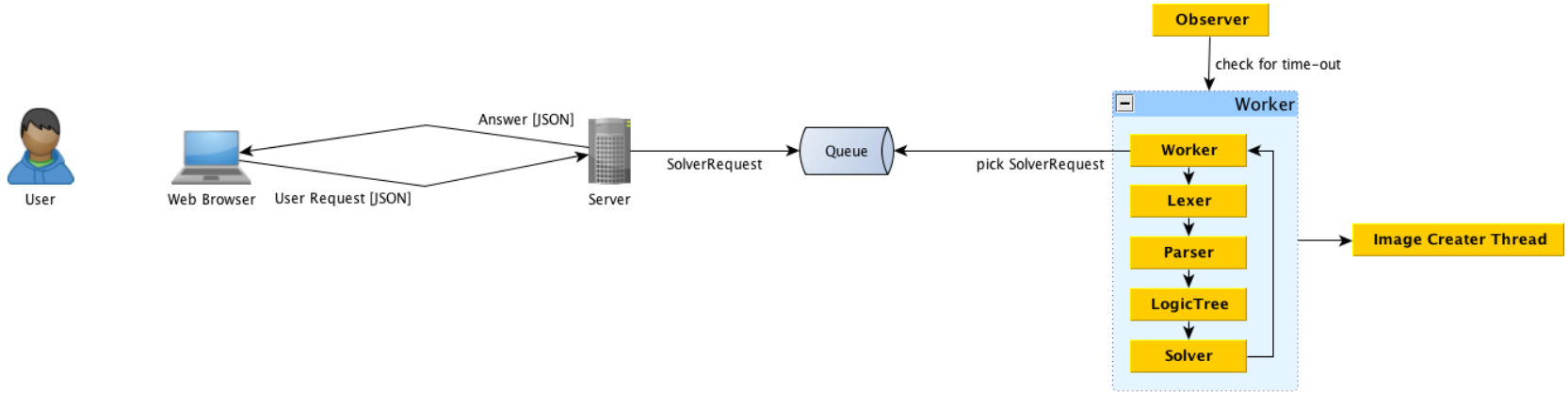


Figure 6.3.: Overview over the most important components of the software. The user sends a request using a web browser. The web browser then communicates with the server using JSON-encoded data. The server enqueues a new SolverRequest into the queue. The worker then takes the requests one by one and handles them using the Lexer,Parser,LogicTree and the actual solver. When done, it creates a new Thread which then creates the image of the automaton as well as some files for the export function of the GUI. Meanwhile the observer makes sure, that the worker does not get stuck in any computation. If a time limit is exceeded, the worker will get killed and replaced by a new one. The user of the formula that caused the time-out will get an time-out error.
While the SolverRequest is handled, the web browser will ask for the status of the request regularly and report the status back to the user. Once the request is done, the results are shown in the GUI.

6.3. Example

In this section a complete run of the program will be explained. The program will start with an user input and return the finished request in the end.
Let us assume, the server gets a new request with the formula $\exists x : (x + y \leq 4 \wedge x = 2)$ and the order to get all solutions of the formula.
The lexer and parser will parse the formula and pass the result to the LogicTree, which then will create a tree from the AST output of the parser. The initial LogicTree will look as shown in figure 6.4.

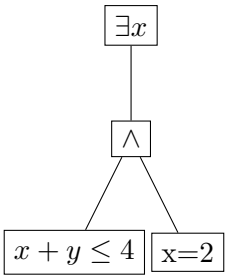


Figure 6.4.: Syntax tree for $\exists x : (x + y \leq 4 \wedge x = 2)$

The first optimization, which is pushing in negations, will not bring any improvement since there are no negations in the tree. The second optimization, which is propagating formulae across the tree will improve the tree. When $x = 2$ is propagated to $x + y \leq 4$, the algorithm will substitute x by 2. The optimized tree will then look as in figure 6.5.

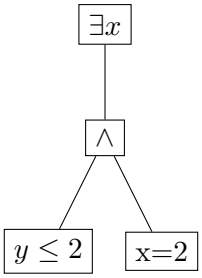


Figure 6.5.: Optimized tree for $\exists x : (x + y \leq 4 \wedge x = 2)$

Next, the software will create an automaton for $y \leq 2 \wedge x = 2$. The resulting automaton is shown in figure 6.6.

The last step to solve the formula is the top-node in the tree which is $\exists x$. To apply the existential quantifier on the automaton, the program just applies the existential quantifier on every transition BDD. After that, the automaton will be determined and minimized to create the final minimal automaton. The result is shown in figure 6.7.

To finish the request, the solver now computes the numbers of solutions of the automaton. In this case 3 will be saved in the SolverRequest. Since the automaton has finitely many solutions, the allSat algorithm can be run on it. The algorithm will return $\{0, 1, 2\}$, which will then be saved into the SolverRequest as well. Finally the SolverRequest is re-

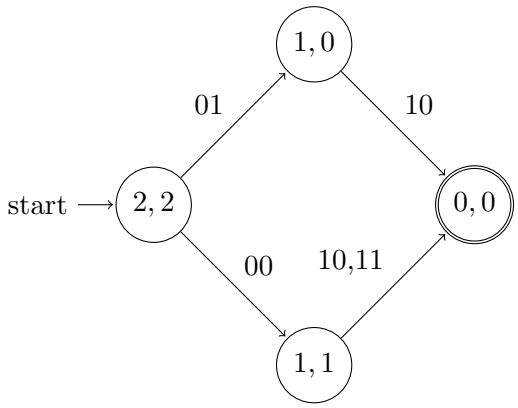


Figure 6.6.: Automaton for $y \leq 2 \wedge x = 2$. Please note, that the trap state was left out for better visibility. All missing transitions lead to the trap state.

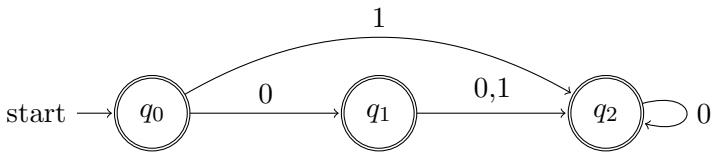


Figure 6.7.: Automaton for $\exists x : x + y \leq 4 \wedge x = 2$. Please note, that the trap state was left out for better visibility. All missing transitions lead to the trap state.

turned and marked as done.
The Thread started by the first worker will now create the image for the automaton and save the export files for the user to download.

7. Performance

7.1. Benchmark

In this part of the thesis the strengths and weaknesses of the described approach to solve formulae are analysed. Most of the used benchmarks originate from [3] . Benchmarks coming from this source can be easily recognized by the prefix "Bench" and a number after that.
We will not discuss every benchmark individually but rather pick out the interesting ones. Results of each benchmark will be reported in table 7.3 at the end of this chapter.
Most benchmarks consist of three values. The first one, which is named Compare in the legend is the value from [3], NonMin corresponds to the times without minimizing the automaton at any stage, whereas Min is the time the program took to compute the minimal automaton for the formula.
All Benchmarks were done using an Intel Core i7 Quad-Core CPU at 2.3GHz and a memory limit of 64 MB.

7.1.1. Bench01

In this benchmark a 4-tuple consisting of numbers divisible by 11, 7 , 5 and 3 is computed . This is equivalent to the formula $\exists x : 11x = v \wedge \exists x : 7x = w \wedge \exists x : 5x = y \wedge \exists x : 3x = z$.

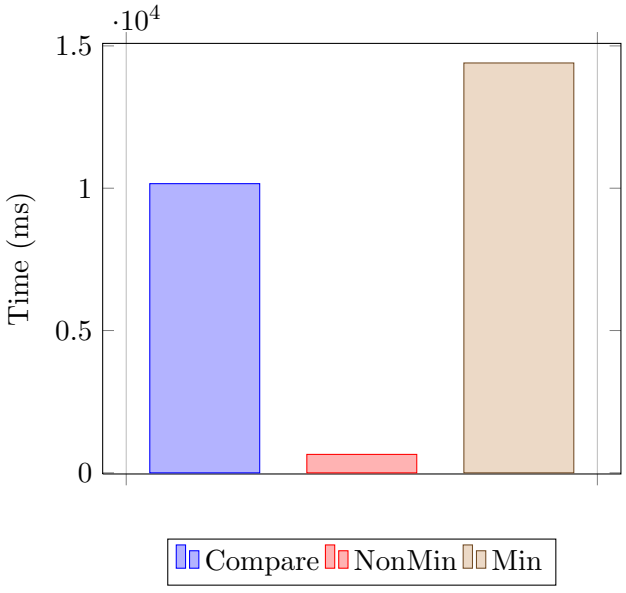


Figure 7.1.: Results for Bench01

This benchmark is particularly interesting since we have to compute automata for each of the 4 variables divisible by 11,7,5 and 3 separately and then compute the intersection of these automata. From figure 7.1 we see, that the minimize-algorithm is a bottleneck of the program. If we do not minimize the automaton, the program takes under a second, if we minimize the automaton, the program takes roughly 11 seconds.

7.1.2. Alternating Quantifiers

In this benchmark we want to analyse the behaviour of the solver for formulae with multiple alternating quantifiers. The challenge in this task lies within the negations that appear in front of every existential quantifier. The tested class of formulae looks as follows:

$$\forall x_0 \exists x_1 ... \forall x_{n-2} \exists x_{n-1} : x_0 - x_1 + ... + x_{n-1} + x_n = 0 \tag{7.1}$$

When this class of formulae is translated into a solvable formula, it will look as follows:

$$\neg\exists x_0 \neg\exists x_1 ... \neg\exists x_{n-2} \neg\exists x_{n-1} : x_0 - x_1 + ... + x_{n-1} + x_n = 0 \tag{7.2}$$

Figure 7.2 shows the results for different n.

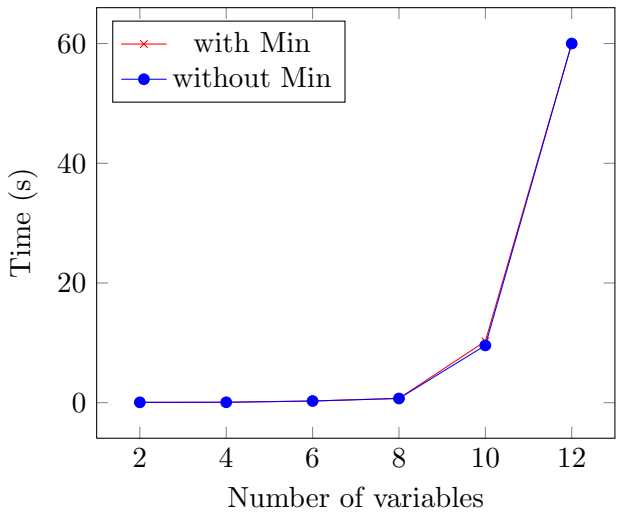


Figure 7.2.: Results for alternation quantifiers

Please note, that a score of 60 does not mean, that the algorithm finished after 60 seconds. A score of 60 just means, that the benchmark ran into a time-out.
As we can see the performance for both, the program with and without minimization does a good job for up to 10 variables. When the variable number exceeds 10 variables, the program is not able to compute the solution within 60 seconds. We can see, that though the minimize algorithm is usually a bottleneck, the minimization of automata makes sense in this case, since the automaton is constantly negated, which results in a blow-up of the automaton. In some cases, the solver with minimization even outperformed the solver without any minimization.

7.1.3. Multiple predicates

In this benchmark we want test how fast the solver can solve the following linear equation system:

$$\begin{aligned} x_0 + x_1 &\leq 1 \\ x_1 + x_2 &\leq 1 \\ &\dots \\ x_{n-2} + x_{n-1} &\leq 1 \end{aligned}$$

The equation system can have different numbers of solutions depending on the number of variables n in the formula. However as one can see quickly, the LES can be solved using a three-state-automaton. The initial state has to be final since 0,...,0 is a valid solution. The transition to the second final state checks, whether all conditions for the first bit of every variable are fulfilled. If so, all following bits have to be 0 since all equations have to be smaller or equal to 1. The third state is the trap state, which just captures all that do not lead to the second final state. Figure 7.3 shows a sample solution for n = 3.

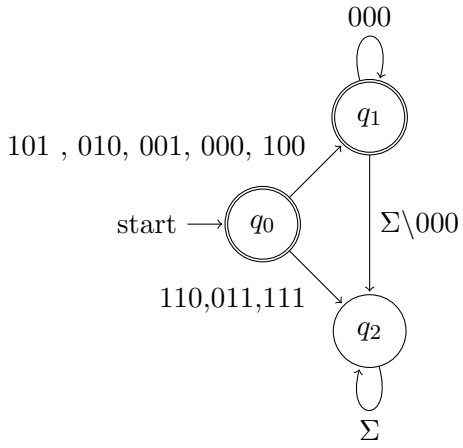


Figure 7.3.: Sample solution for the LES, with n=3.

Figure 7.4 shows the results for different n.

As you can see from the chart in figure 7.4 , the runtime explodes starting at 8 variables when we minimize the automaton. This shows, that the minimization algorithm has some trouble when dealing with big state spaces. Table 7.1 shows the size of the automaton for the different numbers of variables n before the solver minimizes it.

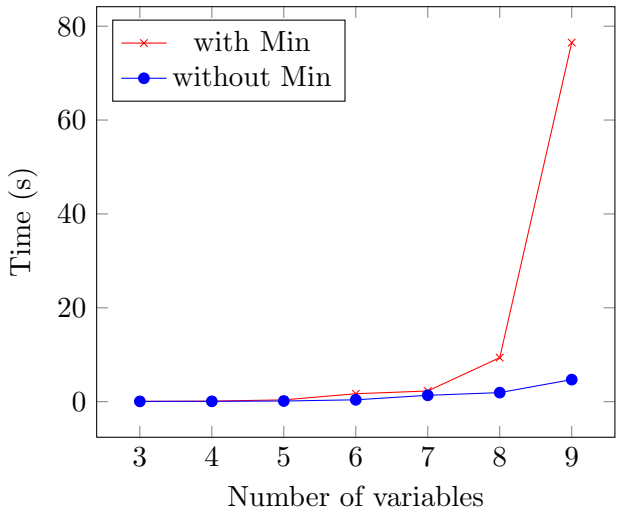


Figure 7.4.: Performance for different numbers of variables.

Number of variables n	Automaton size
3	8
4	18
5	42
6	99
7	236
8	565
9	1354

Table 7.1.: Automaton sizes for different numbers of variables n in the LES Benchmark without minimizing the automaton. When minimizing, the automaton size is always equal to 3.

7.2. Big factors

In this benchmark the behaviour of the program with big factors in a predicate is tested. To do this, the following predicate is solved by the solver:

$$10^n \cdot x - (10^n + 1) \cdot y = 0$$

The predicate has infinitely many solutions for each n, namely

$$x = (10^n + 1) \cdot s$$
$$y = 10^n \cdot s$$
$$s \in \mathbb{N}$$

Since the distance between x's and y's grows with n, the automaton also grows with n. Figure 7.5 shows the runtimes for different values for n. The according automaton sizes can be seen in table 7.2. Please note, that a time of 60 indicates a time-out and not a

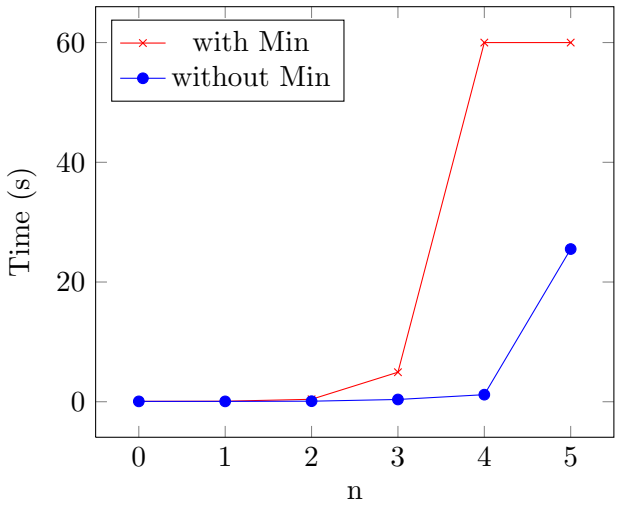


Figure 7.5.: Performance for different instantiations of n.

run time of 60 seconds.

Value for n	Automaton size
0	3
1	21
2	201
3	2001
4	20001
5	200001
...	...
n	$2 \cdot 10^n + 1$

Table 7.2.: Automaton sizes for different n.

This benchmark is particularly interesting since the process without minimization does return the same automaton as the process with minimization. This means, that this benchmark does not only test the performance for predicates with big factors, but also the performance of the minimization algorithm for automata that are already minimal. From the chart in figure 7.5 one can see, that the algorithm does fairly well up to 2001 states. At the next step which has 20001 states , the time-out limit is reached already.

Name	Time without minimization [s]	Automaton size without minimization	Time with minimization [s]	Automaton size with minimization	Comparison from [3] [s]
Bench01	0.706	1261	16.651	1155	10.16
Bench02	3.303	23587	- (Time-out)	-	8.23
Bench03	0.685	1507	31.961	1507	10.13
Bench04	1.644	20452	- (Time-out)	-	2.08
Bench05	1.736	144403	- (Time-out)	-	22.58
Bench08	0.621	11112	- (Time-out	-	2.13
Bench09	1.178	9158	- (Time-out)	-	2.42
Bench10	0.005	106	0.057	105	0.23
Bench11	0.001	4	0.001	3	0.19
Bench12	0.001	5	0.004	5	0.19
Bench13	0.012	316	0.512	211	0.31
Bench14	0.053	340	0.604	236	0.85

Table 7.3.: Run times for all benchmarks from [3]. Bench06 and Bench07 were not used since the solver is not able to solve them. Bench06 was too complex and did not finish. Bench07 had too many variables to even compute the alphabet for the automaton.

8. Conclusion

The aim of this thesis was to develop an user-friendly and easy to use solver for the Presburger arithmetic. As a data structure for this solver, BDD-labelled NFAs/DFAs were used. As such, the first goal was to examine algorithms for BDDs-labelled automata. The first part of the work proposed algorithms for these automata, such as union, intersection and minimization. Additionally algorithms for optimizing formulae, converting formulae into automata and extracting accepting words from automata were explained and discussed.
To round up the theoretical part of the work, a closer look at the bounded Presburger arithmetic and algorithms to solve it was taken. Unfortunately, the algorithms could not be implemented since no direct access to the BDD data structure of the used library could be established. Therefore the implementation of algorithms for bounded Presburger arithmetic will be left open for future works.

The second part of the thesis covered the architecture of the solver. The chapter explained how the front- and back end of the software communicate using Asynchronous HTTP Requests (AJAX) with JSON-encoded data. Further the chapter explained how the algorithms from the first part of the work were used to build the actual solver. Apart from that, a slightly simplified version of the grammar was explained in detail. To end the chapter, a complete run of the solver was given to show how the different parts of the software work together.
The last chapter of the thesis focused on the performance of the implemented software. It was shown, that the overall performance was good, especially when it came to solving con - or disjunctions of predicates. However the minimization algorithm was a bottleneck of the program. Whenever the size of the state space of the automaton grew over a few thousand states, the runtime of the minimization algorithm exploded. Finding a better implementation of this algorithm will be left open for future works on this topic.

Part III.

Appendix

A. Full grammar for ANTLR

```
grammar Presburger;

options {
output=AST;
backtrack=true;
memoize=true;
}

tokens {
  PLUS;
  MINUS;
  NEG;
  AND;
  OR;
  IMP;
  BIIMP;
  ALL;
  EX;
  EQ;
  NEQ;
  GEQ;
  LEQ;
  GT;
  LT;
  L;
  R;
  PRED;
  FORAND;
  FOROR;
  CONSTNUM;
  MACRO;
}



formula
   :   linearpred
   | '!' expr  -> ^(NEG["not"] expr)
   | '&&' ('[' VAR '=' num '.' '.' num ']')+ expr -> ^(FORAND (VAR num num)+ expr)
   | '||' ('[' VAR '=' num '.' '.' num ']')+ expr -> ^(FOROR (VAR num num)+ expr)
   | QUANT expr           -> ^( QUANT expr)
   | MACRO linearterm (',' linearterm)* ')' -> ^(MACRO (PRED linearterm)+)
   ;

expr
    : (s=pre -> pre)(('&&' expr)+ ->
     ^(AND["and"] $s expr+)
     | ('||' expr)+ -> ^(OR["and"] $s expr+)
     | ('->' expr) -> ^(IMP["imp"] $s expr)
     | ('<->' expr) -> ^(BIIMP["biimp"] $s expr)
     )?
    ;

pre
    : '('expr')'->^(expr)
    | formula -> ^(formula);
```

```
f : expr EOF -> ^(expr);

linearpred
  : linearterm (comp linearterm)+ -> ^(PRED linearterm (comp linearterm)+)
  ;

linearterm
  : linmon (sgnlinmon)*
  ;

con : '&&'  -> ^(AND["and"])
    | '||'  -> ^(OR["or"])
    | '->'  -> ^(IMP["implies"])
    | '<->'  -> ^(BIIMP["implies"])
    ;

linmon
  : '+'? INT VAR VAR? -> ^(PLUS["+"]  INT VAR VAR?)
  | '-'  INT VAR VAR?-> ^(MINUS["-"] INT VAR VAR?)
  | '+'? INT  -> ^(PLUS["+"]  INT )
  | '-'  INT  -> ^(MINUS["-"] INT )
  | '+'? VAR VAR?    -> ^(PLUS["+"] INT["1"] VAR VAR?)
  | '-'  VAR VAR?    -> ^(MINUS["-"] INT["1"] VAR VAR?)
  ;

sgnlinmon
  : '+' INT VAR VAR? -> ^(PLUS["+"]  INT VAR VAR?)
  | '-'  INT VAR VAR?-> ^(MINUS["-"] INT VAR VAR?)
  | '+' INT  -> ^(PLUS["+"]  INT )
  | '-'  INT  -> ^(MINUS["-"] INT )
  | '+' VAR VAR?    -> ^(PLUS["+"] INT["1"] VAR VAR?)
  | '-'  VAR VAR?    -> ^(MINUS["-"] INT["1"] VAR VAR?)
  ;

cons
  : '+'? INT -> ^(PLUS["+"]  INT )
  | '-'  INT -> ^(MINUS["-"] INT )
  ;

comp : '==' -> ^(EQ["eq"])
     | '!=' -> ^(NEQ["neq"])
     | '>=' -> ^(GEQ["geq"])
     | '<=' -> ^(LEQ["leq"])
     | '>'  -> ^(GT["gt"])
     | '<'  -> ^(LT["lt"])
     ;

num : VAR | INT;

QUANT
     : 'A'VAR+':'
     | 'E'VAR+':'
     ;

MACRO  :  ('A'..'Z')('a'..'z')*'(';

VAR  :  ('a'..'z')('[' (('a'..'z')('+' INT)? | INT) ']')*;

INT : ('0'..'9')+;

WS : (' '|'\n'|'\t'|'\r')+ { skip(); };
```

B. Benchmarks and test-cases

In order to get the code for the benchmarks and test-cases, please check out the code as described in appendix D. The code for the benchmarks is located in the package BA.Solver.Test under the name PresburgerSolverBenchmark.java. The code for the test-cases is in the same package under the name PresburgerSolverTest.java.

C. MeanDifferencesAlg.csv

Test-name	LetterGuessing [s]	Powerset [s]	Difference [s]
testEqual	0.084	0.083	0.001
testSmallerThan	0.004	0.003	0.001
testBiggerThan	0.006	0.005	0.001
testLessOrEqual	0.005	0.004	0.001
testBiggerOrEqual	0.003	0.003	0
testAnd	0.009	0.009	0
testOr	0.021	0.016	0.005
testBigOr	0.018	0.019	-0.001
testExists	0.009	0.009	0
testNegOr	0.008	0.009	-0.001
testAll	0.007	0.007	0
testForAll	0.733	0.583	0.15
testForOr	2.806	2.311	0.495
testCancelOut	0.003	0.001	0.002
testOptimizeEqual	0.002	0.002	0
testBrakets	0.002	0.002	0
testAndOr	0.004	0.003	0.001
testOrCancel	0.001	0.001	0
testAndCancelDuplicates	0.002	0.001	0.001
testAndCancelFalse	0.003	0.001	0.002
testMultiExists	0.017	0.011	0.006
testMultipleAnd	0.037	0.018	0.019
testOrDifferentTypes	0.001	0.002	-0.001
testOrDifferentTypes2	0	0.002	-0.002
testOrDifferentTypes3	0	0.001	-0.001
testTautology	0	0.001	-0.001
testDivisibleBy2	0	0.002	-0.002
testUnequal	0.002	0.002	0
altaricBench01	0.729	1.914	-1.185
altaricBench02	3.3	3.148	0.152
altaricBench03	0.713	0.75	-0.037
altaricBench04	1.637	1.48	0.157
altaricBench05	1.796	2.11	-0.314
altaricBench08	0.614	0.526	0.088
altaricBench09	1.186	1.256	-0.07
altaricBench10	0.006	0.006	0
altaricBench11	0.001	0.002	-0.001
altaricBench12	0	0	0
altaricBench13	0.014	0.581	-0.567
altaricBench14	0.036	0.038	-0.002
			-0.027575

D. Getting the Solver

In this section instructions on how to get and run the software will be given. To download the solver please visit `http://code.google.com/p/presburger-solver/downloads/list`. You will find up to two downloads. The first download is called PLN.zip. It contains the solver as it was when this work was handed in. The other download - if existent - contains the latest version of the solver. After extracting the archive, please open a Terminal and change the directory to the extracted folder. To run the software type

```
java -jar launch.jar
```

in the Terminal window.

Please use the $-Xm\_m$ command to increase the memory-limit of the solver. The solver will now be accessible on port 8080.
The solver is licensed under the GPL 3.0 license. As such you can also download the code from the Google Code project. Please visit `http://code.google.com/p/presburger-solver/source/checkout` for instructions on how to check out the code.

D.1. Supported platforms

The software was developed under Mac OS X 10.6 and tested under Gnome 3. All other platforms were not tested and are therefore not supported. However it should run under any Linux-based system. To make it work under Windows, some work on the code will be needed. Especially the file operations, as well as the path to the dot-program would have to be changed.

Bibliography

[1] Mojżesz Presburger, 1929, "Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt" in Comptes Rendus du I congrès de Mathématiciens des Pays Slaves. Warszawa: 92–101.

[2] Handbook of Mathematical Logic (Ed. J. Barwise). Amsterdam, Netherlands: North-Holland, pp. 1133-1142, 1977.

[3] `http://altarica.labri.fr/forge/projects/3/wiki/PresTAF`

[4] Javier Esparza, Jörg Kreiker: Automaten und Formale Sprachen, WS 2010/11

[5] Alexandre Boudet, Hubert Comon: Diophantine Equations, Presburger Arithmetic and Finite Automata

[6] M. J. Fischer and M.O. Rabin. Super-exponential complexity of Presburger arithmetic. In R.M. Karp, edition, Complexity of Computation, SIAM-AMS Proceedings, Vol. 7 pages 27 42, 1974.

[7] An Improved Monte Carlo Factorization Algorithm, Richard P. Brent, 1980.

[8] Nondeterministic Algorithms , Robert W. Floyd , 1967

[9] Minimization of Automata , Jean Berstel, Luc Boasson, Olivier Carton, Isabelle Fagnot , 2010

[10] Donald E. Knuth: The Art of Computer Programming - Combinatorial Algorithms, Part 1, Addison-Wesley 2011, 202-280, ISBN 0-201-03804-8

[11] Ingo Wegener: Branching Programs and Binary Decision Diagrams, SIAM Monographs on Discrete Mathematics and Applications 4, ISBN 0-89871-458-3

[12] Sheldon B. Akers: Binary Decision Diagrams, IEEE Transactions on Computers, C-27(6):509-516, Juni 1978.

[13] T. Knuutila. Re-describing an algorithm by Hopcroft. Theoretical Computer Science, 250(1-2):333-363, 2001.

[14] Jean Berstel und Olivier Carton: On the Complexity of Hopcroft's State Minimization Algorithm, Lecture Notes in Computer Science, 2005, Volume 3317/2005, 35-44