Comparison of Solving Techniques for Non-Linear Sparse Equations over Finite Fields with Application in Cryptanalysis

Thorsten Ernst Schilling

June 2, 2008

Master Thesis

Department of Informatics
University of Bergen
Norway

Acknowledgements/Danksagung

At first I would like to thank my family for their persistent support during my studies and in all other circumstances.

Furthermore I would like to thank Igor Semaev who motivated me during the whole thesis and was available for all questions to the topic of the thesis.

I thank also my friends for several liters of coffee, sensible discussions and miscellaneous kinds of distraction.

Als erstes möchte ich mich bei meiner Familie für ihre ständige Unterstützung während meines Studiums und in allen anderen Lebenslagen bedanken. Durch sie wurde diese Masterarbeit erst möglich.

Des weiteren danke ich Igor Semaev, der mich von Anfang an motiviert hat und mir für alle Fragen zum Thema der Arbeit bereit stand.

Ich bedanke mich ebenfalls bei meinen Freunden, für den ein oder anderen Liter Kaffee, sinnvolle Diskussionen und verschiedene Arten der Ablenkung.

Contents

1 Introduction

Let $\mathbb{F}_q$ be a finite field of q elements and $X = \{x_1, x_2, \ldots, x_n\}$ be variables where $x_i \in \mathbb{F}_q$. We consider the following system of equations F

$$f_1(X_1) = 0,\ f_2(X_2) = 0,\ \ldots,\ f_m(X_m) = 0. \tag{1.1}$$

In this system all f_i are polynomials over $\mathbb{F}_q$ and for every $1 \leq i \leq m$ it holds that $X_i \subseteq X$. Such an equation system is called l-sparse if for all $1 \leq i \leq m : |X_i| \leq l$.

The content of the following master thesis presents three different approaches to solve an instance of (1.1).

The research in this field is motivated by the cryptanalysis done on symmetric block and stream ciphers in the last years. By trying to break a cipher one can obtain for different systems a system (1.1) and by obtaining the solution to it possibly reveal an inner state or even the key to the crypto system itself. This kind of approach to break a cipher belongs to the group of algebraic attacks and has the advantage that a relatively small number of plaintext ciphertext pairs is needed to obtain a solution.

Solving this kind of equation system belongs to the class of NP-complete problems, since it exists a many-to-one reducibility to the popular SAT-problem. Therefore there is little hope to find an algorithm which does this task in polynomial time, thus efficient.

Recent attacks of this kind were often done by applying the theory of Gröbner basis to the system. The root to the family of these algorithms is the so called Buchberger's Algorithm which, simply spoken, tries to obtain another, simpler to solve equation system from the input instance. The related algorithms have often an unattractive run-time behavior. That means that the cost of calculating a Gröbner basis often exceeds the cost of a brute force attack.

Another approach, especially attractive for systems (1.1) of characteristic 2, are so called SAT-solving algorithms. The algorithms of this family try to obtain a solution by sophisticated guessing heuristics, fast propagation and conflict resolution tactics. Despite the fact, that their roots range from the 1960's, they are currently the most successful, spoken in terms of speed, to find a solution to (1.1) over $\mathbb{F}_2$, due to extensive research over the last 40 years.

Lately a new method was developed from the need of a fast solving algorithm. This is the group of the Gluing/Agreeing Algorithms which were the main matter during the work of this thesis. They also use a backtracking strategy: guess and determine. However Gluing and Agreeing their selves are more general approaches than Clause Resolution and Unit Clause propagation, which are main components of SAT-solvers. This is probably reason why expected complexity bounds on Gluing/Agreeing algorithms are so low in comparison with the worst case theoretical estimates provided by SAT-solving methods. The thesis was aimed to compare all above methods in practice.

The work on this thesis gives a summary of the Gluing/Agreeing techniques, as well as a reference implementation of this methods. Furthermore widely used SAT-solving techniques are explained and a short insight to the theory of Gröbner basis is given. To demonstrate the application of this techniques two ciphers are presented along with an explanation how to obtain a system (1.1) for them. By the reference implementation obtained experimental results are presented in comparison to results of an up-to-date SAT-solver, called *minisat*. In the last part of this thesis further improvements to the Gluing/Agreeing techniques are presented. Finally the results and cognitions obtained during the work on this thesis are discussed.

2 Gluing and Agreeing

In the following chapter algorithms, originally developed by Håvard Raddum, Igor Semaev and independently discovered by A.D. Zakrevskii and I.V. Vasilkova, see [ZV00, Rad04, Sem05, RS06, Sem07, RS07], are presented. They belong to the group of the Gluing/Agreeing Algorithms whose aim is to find a solution to an equation system over a finite field. The main work during this thesis was to implement the ideas beyond this algorithms and to find possibilities to speed them up, either algorithmically or by implementation.

The fundament of the algorithms builds the Gluing Algorithm, and its tree search variant from the family of the backtracking algorithms, presented at first. Afterwards the main matter will be the Agreeing Algorithms which are polynomial algorithms to check if a partial solution produced from a Gluing Algorithm is correct and to reduce the number of possible solutions to an equation system.

2.1 Basic Definitions

To give an alternative representation for an equation from (1.1) we introduce the definition of the *symbol.*

Definition 2.1 (Symbol) *A symbol of an equation $f_i(X_i) = 0$ is a tuple (X_i, V_i), where X_i is a set of variables in which f_i is defined and $V_i = \{v_1, v_2, \dots, v_k\}$ is a set of satisfying assignments of f_i.*

Equipped with this definition we can express (1.1) as a set of symbols

$$E = \{S_1, S_2, \dots, S_m\} = \{(X_1, V_1), (X_2, V_2), \dots, (X_m, V_m)\}, \tag{2.1}$$

where every symbol S_i is referring to an equation $f_i(X_i) = 0$.

Definition 2.2 (Landau Notation [Knu76]) *To estimate the complexity of algorithms we use the Landau notation which defines the following three sets of functions:*

1. *$\mathcal{O}(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants C and n_0 with $|g(n)| \leq Cf(n)$ for all $n \geq n_0$.*
2. *$\Theta(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants C, C', and n_0 with $Cf(n) \leq g(n) \leq C'f(n)$ for all $n \geq n_0$.*

2.2 The Gluing Algorithm

2.2.1 Algorithmic Description

Let us consider two symbols

$$(X_1, V_1), (X_2, V_2) \tag{2.2}$$

as our input to the algorithm. We will first demonstrate how to obtain all common solutions to that pair of symbols, denoted as a set of vectors U. If all common solutions to (2.2) are stored as vectors defined in $X_1 \cup X_2$ we can create a new symbol $(X_1 \cup X_2, U)$ which can be seen as the *glued representation* of (2.2) and therefore be substituted. To apply the method one defines for the pair of symbols the set of variables $Z = X_1 \cup X_2$ and $Y = X_1 \cap X_2$. Then one computes the set U of Z-vectors by

$$U = \{(a_1, b, a_2) | (a_1, b) \in V_1 \text{ and } (b, a_2) \in V_2\}$$

where a_i are $(X_i \setminus Y)$-vectors and b is an Y-vector, that is a projection of a vector to variables Y. One can see, that the size of the outcome of this computation, namely the size of U is in $\mathcal{O}(q^{|X_1 \cup X_2|})$. The overall complexity of the operation is

$$\mathcal{O}(|U| + |V_1| + |V_2|)$$

where $|V_1| + |V_2|$ can be considered as the sorting of the vectors through algorithms like the bucket sort [CL04].

The single gluing operation is denoted by

$$(Z, U) = (X_1, V_1) \circ (X_2, V_2),$$

where $Z = X_1 \cup X_2$. And by

$$a = b \circ c$$

is denoted, that a is the combination of the vectors b and c.

To solve the equation system (2.1) one can apply this procedure repetitively to the problem instance in the form of the following algorithm.

Algorithm 1 Gluing Algorithm

```
1: procedure GLUING(E)
2:     $(Z, U) \leftarrow (X_1, V_1)$
3:     $k \leftarrow 2$
4:     while $k \leq m$ do
5:         $(Z, U) \leftarrow (Z, U) \circ (X_k, V_k)$
6:         $k \leftarrow k + 1$
7:     end while
8:     return $(Z, U)$
9: end procedure
```

It is obvious, that this algorithm returns all possible solutions to the equation system. It should be remarked, that the size of the memory used by the algorithm is equal to the time the algorithm runs since the gluing algorithm is mostly dependent on finding common solutions and storing them back to (Z, U).

Example To get a better understanding, here is an example of the procedure. In this case the algorithm yields an unique solution, which is of course not the general case. For the most single gluing steps the outcome will be a set of possible solutions, therefore the final computation step may contain a set of solutions instead of a single one.

Let (2.1) consist of three symbols S_1, S_2, S_3 in five variables $X = \{x_1, x_2, x_3, x_4, x_5\}$

S_1	x_2	x_4
a_1	1	0
a_2	1	1
a_3	0	0

,

S_2	x_1	x_3	x_4
b_1	0	1	0
b_2	1	1	0
b_3	1	0	1
b_4	0	0	0

,

S_3	x_1	x_4	x_5
c_1	1	1	1
c_2	0	1	0

,

then the gluing of the first two equation leads us to the equation system

$S_1 \circ S_2$	x_1	x_2	x_3	x_4
d_1	1	1	0	1
d_2	0	1	1	0
d_3	1	1	1	0
d_4	0	1	0	0
d_5	0	0	1	0
d_6	1	0	1	0
d_7	0	0	0	0

,

S_3	x_1	x_4	x_5
c_1	1	1	1
c_2	0	1	0

.

The last gluing operation yields

$S_1 \circ S_2 \circ S_3$	x_1	x_2	x_3	x_4	x_5
e_1	1	1	0	1	1

,

where one can see that the vector $a_2 \circ b_3 \circ c_1 = e_1 = (1, 1, 0, 1, 1)$ is the only solution to the system. Another order of the equations would result in a different intermediate solution, but the outcome is the same.

2.2.2 Gluing1 Algorithm

Since the memory requirement of the gluing algorithm is the same as the running time, namely exponential, here another version of the algorithm, the Gluing1 Algorithm is presented. The asymptotic running time of the algorithm is the same, but it requires only $poly(n)$ bits memory. In general it can be seen as a tree search [Wei94] through the possible gluings of equations. One defines a rooted search tree T with the root $\emptyset$.

With M_i are vectors of length n denoted which keep track of a partial solution for every tree depth $1 \leq d \leq m$. The set of this vectors is denoted by $M_{1,2,\ldots,m}$. The level, or current depth of the tree, where the algorithm is operating is denoted by d.

Through the tree search the algorithm keeps track of which assignments have already been tried with a set of integer variables $s_i, 1 \leq d \leq m$, denoted by $s_{1,2,\ldots,m}$. Every s_i is specific for a symbol S_i and points to the next not yet tested vector in the ordered set $V(S_i) = V_i$.

Now every possible gluing is tried sequentially from the root up to $d = m$ until a solution is found, which is M_m. The algorithm aborts if $d < 1$. In that case no initial gluing is any longer possible and all vectors of S_1 have been tried and $NOSOLUTION$ is returned to announce that there exists no solution for the equation system.

Algorithm 2 Gluing1 Algorithm

```
1: procedure GLUING1(E)
2:     $d \leftarrow 1$
3:     $M_{1,2,\dots,m} \leftarrow (-_1, -_2, \dots, -_n)$        ▷ Empty vector with size of number of variables
4:     $s_{1,2,\dots,m} \leftarrow 1$
5:     while $d \leq m$ and $d > 0$ do
6:         if $\exists i \geq s_d : \exists v_i \in V_d : v_i \circ M_d$ then        ▷ First glueable vector of $V_d$
7:             $M_{d+1} \leftarrow v_i \circ M_d$
8:             $s_d \leftarrow s_d + 1$        ▷ Keep track which vector was already used
9:             $d \leftarrow d + 1$
10:        else
11:            $s_d \leftarrow 1$        ▷ Reset all evaluated vectors from this level
12:            $d \leftarrow d - 1$
13:        end if
14:    end while
15:    if $d = m$ then
16:        return $M_m$
17:    else
18:        return NOSLUTION
19:    end if
20: end procedure
```

Example According to our example we step through a search tree. Assume the equations S_1, S_2, S_3 in that order. The symbol S_1 is located at $d = 1$, at $d = 2$ we have the assignments of S_2 and at $d = 3$ the assignments of S_3. Now we define $M = \{M_0, M_1, M_2, M_3\}$ which is a set of vectors of length n and stands for the intermediate partial solutions at every depth in the tree. At the root all M_i's are empty, denoted $(-, -, -, -, -)$.

$$M_{0,1,2,3} = (-, -, -, -, -).$$

Now we apply, according to the order of vectors in S_1, the first vector to our model. That results in an intermediate solution

$$M_1 = (-, 1, -, 0, -).$$

The next step is to find a possible extension to the model. This step can be seen as an attempt to find a possible gluing between two vectors a_i and b_j. Whenever a gluing is possible the model is extended, if no gluing is possible to the partial solution we have to go one step back in the search tree. In our case we find that the gluing $a_1 \circ b_1$ is possible. That results in the model

$$M_2 = (0, 1, 1, 0, -).$$

Now no further gluing to a vector from S_3 is possible. That implies, that the guess was wrong and so the algorithm tries another vector from S_2. The gluings $a_1 \circ b_2$ and $a_1 \circ b_4$ are also valid, but neither of them gives a intermediate result which can be further extended with an assignment of S_3. Therefore the algorithm goes one step back and tries the next vector from S_1. The result of $a_2 \circ b_3$ yields

$$M_2 = (1, 1, 0, 1, -)$$

which can be extended with c_1 to

$$M_3 = (1, 1, 0, 1, 1)$$

and so gives a final solution. Therefore the search tree of our example after termination by finding a solution would look like figure (2.2.2).

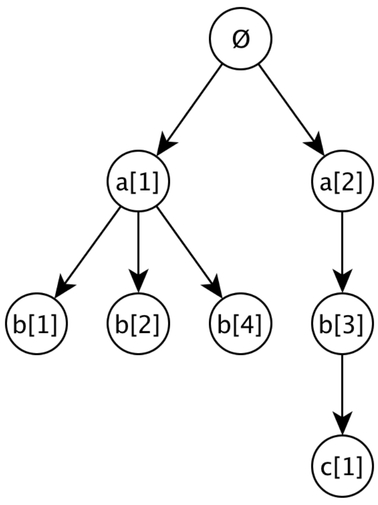

Figure 2.1: Search Tree

2.2.3 Expected Complexity of the Gluing Algorithm

Here we will give the mathematical expectation of the complexity of the Gluing Algorithm.

Equiprobable distribution on instances (1.1), each instance has the same probability, is assumed. That is, given the sequence of natural numbers m and $l_1, \ldots, l_m \leq l$, equations in (1.1) are generated independently. The particular equation $f_i(X_i) = 0$ is determined by the subset X_i of size l_i taken uniformly at random from the set of all possible l_i-subsets of X, that is with the probability $\binom{n}{l_i}^{-1}$, and the mapping (polynomial) f_i taken with the equal probability $q^{-q^{l_i}}$ from the set of all possible mappings to $\mathbb{F}_q$ defined on l_i-tuples over $\mathbb{F}_q$ (the set of polynomials of degree $\leq q-1$ in each of l_i variables).

With reference to [Sem05] let be

$$\gamma_0 = g(\alpha_0)$$

be the maximum of

$$g(\alpha) = f(z_\alpha) - \alpha + \alpha\ ln\ \alpha - \frac{\alpha\ ln\ q}{l}, \alpha > 0$$

with

$$f(z) = ln(e^z + q^{-1} - 1) - \alpha\ ln(z)$$

as a real valued function in a real valued variable z for a positive number α. By z_α the only positive root of the equation

$$\frac{\partial f}{\partial z} = 0$$

is denoted.

Theorem 2.3 *Let ϵ be any positive real number and $l \geq 3$ and $q \geq 2$ be fixed natural numbers when n tends to infinity. Then the mathematical expectation of the complexity of the Gluing Algorithm is*

$$\mathcal{O}(qe^{\gamma_0} + \epsilon)^n + poly(n)m)$$

operations, where $\gamma_0 = -\frac{ln\ q}{l} - (q^{\frac{1}{l}} - 1)ln\left(\frac{1-q^{-1}}{1-q^{-\frac{1}{l}}}\right)$ and $poly(n)$ is a polynomial in n.

as stated and proven by Igor Semaev in [Sem05].

2.2.4 Gluing2 Algorithm

In order to find a way for even faster gluing the following lemma as stated and proven in [Sem05] is presented.

Lemma 2.4 *Let $q \geq 2$ and $l \geq 3$ be natural numbers. Then for $\alpha > 0$ the function $g(\alpha)$ has just one maximum value.*

$$g(\alpha_0) = -\frac{ln\ q}{l} - (q^{\frac{1}{l}} - 1) ln \left(\frac{1 - q^{-1}}{1 - q^{-\frac{1}{l}}} \right)$$

and $\alpha_0 < l/2$.

This lemma implies that there exists just one real number $\alpha_1 > 0$ such that

$$g(\alpha_1) = g(2\alpha_1)$$

and $\alpha_1 < \alpha_0 < 2\alpha_1 \leq l$. So one finds natural numbers k_1 and k_2 such that

$$\frac{(k_1 - 1)l}{n} < \alpha_1 \leq \frac{k_1 l}{n}$$

and

$$\frac{(k_1 + k_2 - 1)l}{n} < 2\alpha_1 \leq \frac{(k_1 + k_2)l}{n}.$$

Let us consider two subsystems of equations (1.1):

$$f_1(X_1) = 0, f_2(X_2) = 0, \ldots, f_{k_1}(X_{k_1}) = 0$$

and

$$f_{k_1+1}(X_{k_1+1}) = 0, f_{k_1+2}(X_{k_1+2}) = 0, \ldots, f_{k_1+k_2}(X_{k_1+k_2}) = 0$$

in the form (2.1). Let $X' = X_1 \cup X_2 \ldots \cup X_{k_1}$ and V' be the set of all solutions to the first subsystem in X'-vectors. Similarly let $X'' = X_{k_1+1} \cup X_{k_1+2} \cup \ldots \cup X_{k_1+k_2}$ and V'' be the set of all solutions to the second subsystem in X''-vectors.

Algorithm 3 Gluing2 Algorithm

```
1:  procedure GLUING2(E)
2:      Apply the Gluing Algorithm to find $(X', V')$ and $(X'', V'')$
3:      $(Z, U) \leftarrow (X', V') \circ (X'', V'')$
4:      $k \leftarrow k_1 + k_2 + 1$
5:      while $k \leq m$ do
6:          $(Z, U) \leftarrow (Z, U) \circ (X_k, V_k)$
7:          $k \leftarrow k + 1$
8:      end while
9:      return $(Z, U)$
10: end procedure
```

2.2.5 Expected Complexity of the Gluing2 Algorithm

Theorem 2.5 *Let ϵ be any positive real number and $l \geq 3$ and $q \geq 2$ be fixed natural numbers when n tends to infinity. Then the mathematical expectation of the complexity of the Gluing2 Algorithm is*

$$\mathcal{O}(qe^{g(\alpha_1)} + \epsilon)^n + poly(n)m)$$

operations.

as stated and proven by Igor Semaev in [Sem05].

2.2.6 Complexity Comparison

Now one can compare the above stated complexity expectation values of the Gluing Algorithm and the Gluing2 Algorithm based on the probabilistic model described in [Sem05].

The following table shows a comparison between worst-case values for the l-SAT problem (see chapter 3) taken from [Iwa04] to the mentioned expectation values.

l	3	4	5	6
worst-case	1.324^n	1.474^n	1.569^n	1.637^n
Gluing1, expectation	1.262^n	1.355^n	1.425^n	1.479^n
Gluing2, expectation	1.238^n	1.326^n	1.393^n	1.446^n

It should be remarked that the huge difference between the values from [Iwa04] and the values from the algorithms presented here are never the less caused by taking worst-case values in comparison to expectation values and because the average instances of the l-SAT problem and that of (1.1) are different.

2.3 The Agreeing Procedure

The following section describes the Agreeing Algorithm [RS06, Sem07] which is a way to eliminate in a system of equations 1.1 vectors that are not suitable to a solution to the whole equations system.

2.3.1 Algorithmic Description

For symbols (2.2) one defines the set of variables $Y = X_1 \cap X_2$. $V_{1,2}$ contains all subvectors of V_2 to the variables in Y and $V_{2,1}$ contains all Y-subvectors of V_2.

If $V_{1,2} = V_{2,1}$ the symbols are called agreeing. If $V_{1,2} \neq V_{2,1}$ we apply a procedure which is called Agreeing. Agreeing means, that we delete from the set V_1 all vectors whose Y-subvectors are not occuring in in $V_{2,1}$ and vice versa we delete from the set V_2 all vectors whose Y-subvectors are not occurring in $V_{1,2}$. In other words, we make the sets $V_{i,j}$ equal and let in V_1, respectively V_2 only the vectors occur which have a reference in $V_{1,2} \cap V_{2,1}$.

The vectors we deleted from the sets can obviously not occur in a solution to both symbols, since they have no appropriate counterpart in the other symbol.

Now we want to express this procedure in an algorithmic way. By $V_i(Y)$ we denote the set of Y-subvectors of V_i. Similarly by $a_i(Y)$ we denote a single vector projection to the set of variables Y of the vector a_i.

Algorithm 4 Agreeing Procedure

```
1:  procedure AGREE($(X_1, V_1), (X_2, V_2)$)
2:      $Y \leftarrow X_1 \cap X_2$
3:      $V_{1,2} \leftarrow V_1(Y)$
4:      $V_{2,1} \leftarrow V_2(Y)$
5:      if $V_{2,1} = V_{1,2}$ then
6:          return
7:      else
8:          $V_{agree} = V_{2,1} \cap V_{1,2}$
9:          for $\forall a \in V_1$ do
10:             if $a(Y) \notin V_{agree}$ then
11:                 $V_1 \leftarrow V_1 \setminus a$
12:             end if
13:         end for
14:         for $\forall a \in V_2$ do
15:             if $a(Y) \notin V_{agree}$ then
16:                 $V_2 \leftarrow V_2 \setminus a$
17:             end if
18:         end for
19:     end if
20: end procedure
```

Example To illustrate that procedure we show here an example to the equation system of symbols from section 2.2.1. We consider here the symbols S_1, S_3.

S_1	x_2	x_4
a_1	1	0
a_2	1	1
a_3	0	0

,

S_3	x_1	x_4	x_5
c_1	1	1	1
c_2	0	1	0

.

In order to agree these two symbols we create first the set

$$Y = X_1 \cap X_3 = \{x_4\}.$$

This leads to the sets $V_{1,3}$ and to $V_{3,1}$. For a better understanding here the symbols which contain only the projections of the vectors are shown first.

$S_1(Y)$	x_4
$a_1(Y)$	0
$a_2(Y)$	1
$a_3(Y)$	0

,

$S_3(Y)$	x_4
$c_1(Y)$	1
$c_2(Y)$	1

.

Then

$$V_{1,3} = \{(1), (0)\} \text{ and } V_{3,1} = \{(1)\}.$$

One sees immediately that $V_{1,3} \neq V_{3,1}$ and so we calculate $V_{agree} = V_{1,3} \cap V_{3,1} = \{(1)\}$. All vectors which have a projection to Y which is not contained in V_{agree} will be deleted from the sets V_1, V_3. In our example the vectors a_1 and a_3 are deleted from the symbol S_1 and the result of the equation system is

S_1	x_2	x_4
a_2	1	1

,

S_3	x_1	x_4	x_5
c_1	1	1	1
c_2	0	1	0

.

It should be remarked that if the case $V_{i,j} \cap V_{j,i} = \emptyset$ occurs and $X_i \cap X_j \neq \emptyset$, then there exists no solution for such a system of equations. This fact becomes crucial later when we combine both strategies, Gluing and Agreeing. Also quite important is the fact, that the Agreeing does not always produce a single solution or a reduction in the number of vectors as one can see in trying to agree the symbols S_1 and S_2.

2.3.2 Upper Bound Complexity of the Agreeing Procedure

To analyze the complexity of one Agreeing step lets take again a look to the algorithmic structure. Let us assume that we can do the set intersection in $\mathcal{O}(1)$, e.g. by binary operations on bitsets. To calculate the subvectors for the sets $V_{1,2}, V_{2,1}$ we need $\mathcal{O}(|V_1| + |V_2|)$, again by binary operations on bitsets. Since we want to get an upper bound we discard the case that $V_{1,2} = V_{2,1}$ and take a closer look what happens if $V_{1,2} \neq V_{2,1}$. We have to determine for every assignment $a \in V_{i,j}$ if it occurs in $V_{j,i}$, too. This could be done for example by a hash table lookup, therefore possible in $\mathcal{O}(1)$. To determine that for all $a \in V_1$ and all $a \in V_2$ we get again the bound $\mathcal{O}(|V_1| + |V_2|)$. This sums up to

$$\mathcal{O}(1 + 1 + 2(|V_1| + |V_2|)) = \mathcal{O}(|V_1| + |V_2|)$$

for a single Agreeing operation on two symbols.

2.4 The Agreeing1 Algorithm

2.4.1 Algorithmic Description

Now we often run in the situation that two symbols S_i, S_j are agreed, but symbols S_i, S_t and/or S_j, S_t are not after applying the Agreeing Procedure. To propagate now the changes made to the symbols S_i, S_j and to eliminate more solutions to our equation system we use the following approach. The Agreeing1 Algorithm is therefore a way to propagate information obtained by Agreeing about our equation system (2.1) through the whole system.

Algorithm 5 Agreeing1 Algorithm

```
1: procedure AGREEING1(E)
2:     while $S_i, S_j \in E$ which are not agreeing do
3:         AGREE($S_i, S_j$)
4:     end while
5: end procedure
```

Here now a slightly modified version of the lemma that the outcome of the Agreeing1 Algorithm does not depend on the order of the pairwise agreeing is presented. For the original proof one should refer to [RS06] where also was shown, that the Agreeing1 Algorithm produces a maximal agreed set.

Given the set of symbols (2.1) related to the initial system of equations (1.1), we consider a set of subsymbols $(X_i, U_i) \subseteq (X_i, V_i)$ meaning that $U_i \subseteq V_i$ for all $1 \leq i \leq m$. Such a set of subsymbols is called a maximal agreed set of subsymbols if the symbols (X_i, U_i) pairwise agree and for any sets U_i'

$$U_i \subseteq U_i' \subseteq V_i$$

with $U_i \subset U_i'$ for at least one i, the set of subsymbols $(X_i, U_i'), 1 \leq i \leq m$ does not agree.

Lemma 2.6 *The maximal agreed set of subsymbols is unique.*

Proof 2.7 *Assume there are two maximal agreed sets of subsymbols: $(X_i, U_i), 1 \leq i \leq m$ and $(X_i, U_i'), 1 \leq i \leq m$. Then one constructs a new set of subsymbols $(X_i, U_i \cup U_i'), 1 \leq i \leq m$. The latter subsymbols pairwise agree. That is only possible if $U_i = U_i', 1 \leq i \leq m$. The statement is therefore proved.* □

Therefore, if the maximal agreed set of subsymbols is unique and the Agreeing1 Algorithm produces a maximal agreed set, the outcome of the algorithm does not depend on the order of the pairwise agreeing.

2.4.2 Upper Bound Complexity of the Agreeing1 Algorithm

We now try to approximate the upper bound complexity and recall that a single Agreeing operation takes $\mathcal{O}(|V_1| + |V_2|)$ operations. Lets assume, that our equation system is l-sparse with m equations in n variables over $\mathbb{F}_q$. So we have at most q^l assignments per symbol. Every single agreeing step on two symbols involves investigating all possible assignments and deleting some of them. As one should delete at most mq^l of them, this results in an overall complexity for the Agreeing1 Algorithm of

$$\mathcal{O}(m^3 q^{2l})$$

operations.

2.5 The Agreeing2 Algorithm

2.5.1 Algorithmic Description

The Agreeing2 Algorithm, also referred to as the Full Agreeing Algorithm is another way to propagate the information of a single Agreeing step to the whole equation system. Instead of working with the equations itself and agreeing pairwise every symbol we do some beforehand calculations and use this information to create a graph which distributes our information obtained by Agreeing. The method here presented refers to [RS07] in a modified version for our problem instance. With $X_{i,j}$ we denote the set $X_i \cap X_j$.

Algorithm 6 Agreeing2 Precomputation

```
1: procedure AGREEING2PRECOMPUTATION(E)
2:     for each $S_i, S_j \in E$ do
3:         if $|X_{i,j}| > 0$ then
4:             for each $b$ of length $|X_{i,j}|$ do
5:                 Store $\{V_{i,j}(b); V_{j,i}(b)\}$
6:             end for
7:         end if
8:     end for
9:     return List of all tuples $\{V_{i,j}(b); V_{j,i}(b)\}$
10: end procedure
```

In the precomputation the algorithm creates for every pair of symbols tuples $\{V_{i,j}(b); V_{j,i}(b)\}$ if the set $X_{i,j}$ is not empty. The list $V_{i,j}(b)$ consists of the addresses of the assignments a of V_i whose projection to $X_{i,j}$ is b. Similarly the list $V_{j,i}(b)$ contains the addresses of assignments a of V_j whose projection to $X_{i,j}$ is b. This is done for every $|X_{i,j}|$-bit b. Additionally the address of an assignment in the tuples gets a field to mark them. If an address is marked it is considered to be deleted. That means that a list $V_{i,j}(b)$ in which all assignment addresses a are marked is considered as an empty list. A tuple t in which exactly one list, either $V_{i,j}(b)$ or $V_{j,i}(b)$ is empty is called one-sided empty.

Algorithm 7 Agreeing2 Algorithm

```
1: procedure AGREEING2(E)
2:     $T \leftarrow$ AGREEING2PRECOMPUTATION($E$)
3:     while exists a tuple $t \in T$, which is one-sided empty do
4:         for each address $a$ in $t$ which is not yet marked do
5:             for each tuple $u$ in which an address of $a$ exists do
6:                 Mark $a$ in $u$
7:             end for
8:         end for
9:     end while
10:    if All tuples empty then
11:        return FALSE
12:    else
13:        return TRUE
14:    end if
15: end procedure
```

That means that the algorithm at first precomputes the list T of $\{V_{i,j}(b); V_{j,i}(b)\}$ tuples. Then it steps through all tuples which got one-sided empty. The algorithm propagates the information that the assignment a is not agreeing to the rest of the equation system, since it has in at least one symbol no counterpart. The algorithm stops if there exists no more one-sided empty tuple. That is either all assignments which are still present agree to the equation system, or the equation system has no solution. In the first case the Agreeing2 Algorithm returns $TRUE$ as an indication that the system is in an agreed state. In the second case $FALSE$ is returned to indicate that there is no common solution to all symbols.

An important condition for this method to work properly is that the system has to be connected. Assume E is connected. That is for any X_i, X_j there is a path $X_i = X_{i_1}, X_{i_2} \dots X_{i_t} = X_j$, where $X_{i_k} \cap X_{i_{k+1}} \neq \emptyset$ for all $1 \leq k \leq t-1$.

Example Let us consider the example from section 2.2.1. At first we preprocess our equation system and create the required list of tuples in equal projections b to the set $X_{i,j}$ for every pair of symbols.

$$\{a_1, a_3; b_1, b_2, b_4\}, \{a_1, a_3; \emptyset\}, \{a_2; b_3\}, \{a_2; c_1, c_2\}, \{b_3; c_1\}, \{b_1, b_4; \emptyset\}, \{b_2; \emptyset\}, \{\emptyset; c_2\}$$

Note that the tuple $\{a_1, a_3; \emptyset\}$ here implies, that the vectors a_1, a_3 have no counterpart for $X_{1,3} = \{x_4\}$ in the symbol S_3. And the tuples $\{b_1, b_4; \emptyset\}, \{b_2; \emptyset\}$ indicate, that there exists no equation projection in S_3 for $X_{2,3} = \{x_1, x_4\}$ in b_1, b_2, b_4. Similar $\{\emptyset; c_2\}$ gives us the result, that there is no vector b_i in S_2 which has the same projection on $X_{2,3} = \{x_1, x_4\}$ as c_2.

This precomputation now lets us continue with the main algorithm. We start with the first one-sided empty tuple $\{a_1, a_3, \emptyset\}$ and propagate the information to the other tuples. We get

$$\{\overline{a_1}, \overline{a_3}; b_1, b_2, b_4\}, \{\overline{a_1}, \overline{a_3}; \emptyset\}, \{a_2; b_3\}, \{a_2; c_1, c_2\}, \{b_3; c_1\}, \{b_1, b_4; \emptyset\}, \{b_2; \emptyset\}, \{\emptyset; c_2\}$$

where $\overline{x}$ denotes a marked assignment. Further on going with receiving $\{\overline{a_1}, \overline{a_3}; b_1, b_2, b_4\}$ as one-sided empty tuple we get

$$\{\overline{a_1}, \overline{a_3}; \overline{b_1}, \overline{b_2}, \overline{b_4}\}, \{\overline{a_1}, \overline{a_3}; \emptyset\}, \{a_2; b_3\}, \{a_2; c_1, c_2\}, \{b_3; c_1\}, \{\overline{b_1}, \overline{b_4}; \emptyset\}, \{\overline{b_2}; \emptyset\}, \{\emptyset; c_2\}$$

finally by resuming with $\{\emptyset; c_2\}$

$$\{\overline{a_1}, \overline{a_3}; \overline{b_1}, \overline{b_2}, \overline{b_4}\}, \{\overline{a_1}, \overline{a_3}; \emptyset\}, \{a_2; b_3\}, \{a_2; c_1, \overline{c_2}\}, \{b_3; c_1\}, \{\overline{b_1}, \overline{b_4}; \emptyset\}, \{\overline{b_2}; \emptyset\}, \{\emptyset; \overline{c_2}\}$$

which leads us to our overall resulting equation system after Agreeing2 of

S_1	x_2	x_4
a_2	1	1

S_2	x_1	x_3	x_4
b_3	1	0	1

S_3	x_1	x_4	x_5
c_1	1	1	1

in an agreed state.

Introducing a Guess To the Agreeing2 Structure Assume that the equation system is pairwise agreed from the beginning. We should introduce a guess to the structure and use it to check if the guess was correct. This can in general be done in a very easy way.

Given symbols $S_i = (X_i, V_i)$ a guess in variables Y, denoted by $g(Y)$, is compared to all projections $a(Y \cap X_i)$ of V_i if $X_i \cap Y \geq 1$. If $g(Y \cap X_i) \neq a_i(Y \cap X_i)$ one marks a.

After this one should mark them in the appropriate tuples and save the tuples as starting point if they get one sided empty. If the tuples get both sided empty there is nothing more to do with them since they cannot propagate any more information.

One can run now the Agreeing2 Algorithm and check if the result is either TRUE or FALSE. In case the system is consistent to the guess, the output of the Agreeing2 Algorithm is TRUE, otherwise FALSE.

2.6 The Gluing-Agreeing Algorithm

2.6.1 Algorithmic Description

Up to this point we obtained two different approaches. Gluing and Agreeing of equations. From now on we will combine the two strategies to introduce our qualified guess obtained by the Gluing Algorithm into our structure of Agreeing2 and check if the guess is correct. This is done by continuously updating our equation system due to agreeing our intermediate result with the rest of the equations.

As input to the algorithm we take the system (2.1).

Algorithm 8 Gluing-Agreeing Algorithm

```
1: procedure Gluing-Agreeing(E)
2:     $(Z, U) \leftarrow (X_1, V_1)$
3:     $k \leftarrow 2$
4:     while $k \leq m$ do
5:         $s \leftarrow k$
6:         while $s \leq m$ do
7:             Agree($(Z, U), (X_s, V_s)$)
8:             $s \leftarrow s + 1$
9:         end while
10:        $(Z, U) \leftarrow (Z, U) \circ (X_k, V_k)$
11:        $k \leftarrow k + 1$
12:    end while
13:    return $(Z, U)$
14: end procedure
```

2.6.2 Expected Complexity of the Gluing-Agreeing Algorithm

The expected complexity of the Gluing-Agreeing is the same like for the Gluing-Agreeing1 algorithm, which utilizes the algorithmic structure above as a tree search and uses only polynomial memory. Let $(X(1), U'_1)$ be the symbol (X_1, V_1) after $m-1$ agreeings with the symbols (X_i, V_i), where $1 < i \leq m$. For any $1 \leq k < m$ let $(X(k+1), U'_{k+1})$ denote the symbol $(X(k), U'_k)$. The complexity of the algorithm is then

$$\mathcal{O}(m(\Sigma_{k=1}^{m-1}|U'_k| + 1))$$

operations with $\mathbb{F}_q$-vectors of length at most n, where q and l are fixed and n or m may grow as showed in [Sem07].

We compare this values to the worst case scenario here again as for the Gluing1 and Gluing2 algorithms in section (2.2.6).

l	3	4	5	6
worst-case	1.324^n	1.474^n	1.569^n	1.637^n
Agreeing-Gluing1, expectation	1.113^n	1.205^n	1.276^n	1.334^n

2.7 The Gluing-Agreeing2 Algorithm

2.7.1 Algorithmic Description

In order to utilize the agreeing we use in this approach the Agreeing2 Algorithm since it has a better runtime behavior due to the preprocessing steps which make repeating assignment comparisons unnecessary. Furthermore it propagates the knowledge about not agreed vectors more efficient due to its structure in comparison to the Agreeing1 Algorithm which simply tries for every step if there are any not agreed equation pairs left. As mentioned before is the tree search the desirable solution, since its memory requirements are polynomial and it has the same asymptotic run time than the plain approach.

The core structure of the Gluing-Agreeing2 Algorithm is similar to the Gluing1 Algorithm, but we introduce a new variable f which indicates the tree depth at which we check our solution obtained so far by the Gluing Algorithm with the Agreeing2 Algorithm.

Algorithm 9 Gluing-Agreeing2 Algorithm

```
1: procedure GLUING-AGREEING2(E)
2:     $d \leftarrow 1$
3:     $M_{0,1,\ldots,m} \leftarrow (-_1, -_2, \ldots, -_n)$
4:     $s_{1,2,\ldots,m} \leftarrow 1$
5:     while $d \leq m$ and $d > 0$ do
6:         if $\exists i \geq s_d : \exists v_i \in V_d : v_i \circ M_d$ then
7:             $M_{d+1} \leftarrow v_i \circ M_d$
8:             if $d = f$ then          ▷ Start at tree depth $f$ the Agreeing2 and check the result
9:                 if not AGREEING2($M_{d+1}$) then
10:                    $s_d \leftarrow s_d + 1$
11:                    goto 6                  ▷ If Agreeing2 failed resume with the next vector
12:                end if
13:            end if
14:            $s_d \leftarrow s_d + 1$
15:            $d \leftarrow d + 1$
16:        else
17:            $s_d \leftarrow 1$
18:            $d \leftarrow d - 1$
19:        end if
20:    end while
21:    if $d = m$ then
22:        return $M_m$
23:    else
24:        return NOSLUTION
25:    end if
26: end procedure
```

2.8 Sorting Equations

In order to keep the number of new variables arising through the tree search small one should somehow sort the equations to get a low magnitude in $|X(i)| = |X_1 \cup X_2 \cup \ldots \cup X_i|$ before starting the computation. For example consider a 4-sparse equation system with the sets X_i of variables

$$\{1, 2, 3, 4\}, \{5, 6, 7, 8\}, \{3, 4, 7, 8\}$$

in that order. If one would start the Gluing Algorithm on that order it is obvious, that in the Gluing step for the first symbol and the second step all combinations have to be tried in the worst case. Since we have a number of q^4 satisfying assignments in the worst case it would result in the number of q^8 possible gluings.

If we consider a better ordering like

$$\{1,2,3,4\},\{3,4,7,8\},\{5,6,7,8\}$$

it would yield a potential of q^6 solutions for the first gluing step since two variables are already defined through the first assignment chosen. This fact makes it pretty obvious how important a good sorting for a fast gluing is.

Here a simple sorting approach is presented, which has an upper bound of $\mathcal{O}(m^2)$. The algorithm takes as the input a list E of symbols and sorts them in the way, that for every i the locally lowest $|X(i)|$ is archived iteratively. There exist of course other, more efficient possibilities to sort, but in the face of the practicability of this algorithm and the polynomial running time it is still senseful to use this approach.

Algorithm 10 Simple Sorting Algorithm

```
1:  procedure SORT(E)
2:      $n \leftarrow |X(E)|$                         ▷ Store the number of variables
3:      $T_1 \leftarrow S_1$
4:      $T_2 \leftarrow S_2$
5:      for each $S_i, S_j \in E$ do      ▷ Find pair $S_i, S_j$ with the smallest $|X(S_i) \cup X(S_j)|$
6:          if $|X(S_i) \cup X(S_j)| < |X(T_1) \cup X(T_2)|$ then
7:              $T_1 \leftarrow S_i$
8:              $T_2 \leftarrow S_j$
9:          end if
10:     end for
11:     $E \leftarrow E \setminus \{T_1, T_2\}$
12:     $R[1] \leftarrow T_1$              ▷ Result list becomes the first two symbols as first elements
13:     $R[2] \leftarrow T_2$
14:     while $|E| > 0$ do
15:         $s \leftarrow n$
16:         for each $S_i \in E$ do
17:             if $|X(R) \cup X(S_i)| < s$ then
18:                 $s \leftarrow |X(R) \cup X(S_i)|$
19:                 $e \leftarrow S_i$
20:             end if
21:         end for
22:         $E \leftarrow E \setminus e$
23:         Append $e$ at R                ▷ Append iteratively the locally smallest equation
24:     end while
25: end procedure
```

2.9 Implementation

During the work on this master thesis the program "fastglue2" developed. The program itself is a result of working with the algorithms from this chapter, and only one in a row while finding a efficient way to implement the methods. In this section the implementation of the program "fastglue2" is described. It uses the Gluing-Agreeing2 Algorithm to find a solution to a non-linear equation system over $\mathbb{F}_2$.

The main goal during the development of this program was to keep the implementation as easy as possible but at the same time retaining performance. In the early states of experimenting with different implementations it turned out that the only sensible way is to go the tree search way, since only polynomial memory is required. This had to be iterative and not recursive due to technical reasons and this immediately affected the speed of the program.

Also easy was the decision to write the solver exclusively for $\mathbb{F}_2$. Here is a strong competitor available, namely minisat see [ES04] and [ES03] which uses a filed version of the DPLL algorithm (described in chapter 3).

The programming language used to implement the program is C++. This has different reasons. Firstly it is fast. Since it is in comparison to other high level programming languages (for example to Java) quite hardware oriented and gives the control of the memory to the programmer. On the other hand it is object oriented and it is possible to produce a good readable code and a lot of well developed libraries are available. One could argue, that a implementation in C could increase the speed even more, but a test implementation in C showed that the increasing unreadability outweighed the speed advantage.

2.9.1 Code Notation

In this section the following pseudo code notation is used and throughout the rest of the document whenever to written program code is referred.

- A C/C++ internal datatype is announced by **datatype**, for example **int**, **float** or **short int**.
- Predefined C/C++ classes, for example from the std/stl-lib or the boost library are denoted by verbatim text, e.g. `boost::dynamic_bitset` or `std::vector`.
- Types and classes which are written or defined in the context of the work for this thesis or for other projects which are mentioned and are crucial parts of the functioning of this programs are denoted by ***Classname***. For example ***Equation*** or ***Assignment***.
- Template classes are like in C++ denoted with the datatype and the template parameter in brackets (<>), like `boost::dynamic_bitset` <**unsigned long long int**>.
- Blank types, for example for the definition of templates are denoted italic, e.g. *BlankType*.

2.9.2 Memory Representation

General Considerations The first problem in implementing the algorithms was to find a reasonable representation in the memory of a standard personal computer with a x86 architecture. The representation should be easy to handle and also perform well in the comparison of single vectors. First recall the input data of the program.

An equation, or symbol S_i is a tuple (X_i, V_i), where X_i is a set of indices of the variables in which it is defined and V_i is a list of vectors which make the equation satisfiable. This could in a real problem for example be the symbol

$$S_j \quad = \quad (\{2,5,9\}, \{a_1 = (1,1,0), a_2 = (0,0,1), a_3 = (1,0,1), a_4 = (1,1,1)\}),$$

where the equation system over $\mathbb{F}_2$ is 3-sparse and the number of variables is $n = 10$. The simple approach is to represent the X_j as a set of integer values and the satisfying vectors in that case as C++ vectors (or arrays) of length $|X_j|$ of boolean values.

Let us assume that the algorithm reached the point where we have to find out wether or not an intermediate result $M_k = (0, -, 1, -, 1, -, -, -, 0, -)$ and the vector a_1 are glueable.

The first task is that we have to know the set intersection between the variables in which model M_k is so far defined and X_j. It is obviously possible to calculate that beforehand, since the ordering of the symbols in the search tree does not change during the computation and can therefore be stored for each depth of the tree in advance in the variable $ModInt_k$, where k is the tree depth (see 2.9.10). So this value is accessible in $\Theta(1)$.

The next problem is the comparison of the intermediate solution M_k to the vector a_1. In the above mentioned scenario we have to compare the subvectors $a_1[x_5, x_9]$ and $M_k[x_5, x_9]$ for equality. Since the positions 5 and 9 are not the real positions of the desired indices in the vector (or array) a_1 we have to perform additional calculations to form the vector $a_1[x_5, x_9] = a_1[2, 3] = a_1' = (1, 0)$ in order to compare it with $M_k[x_5, x_9] = M_k[5, 9] = (1, 0)$. A possible solution here would be for example a hash table which allows the lookup of correct indices in a_1 for the variables x_5 and x_9. Nevertheless would every single access to an index equal a single operation, so a comparison of a vector to a intermediate solution would take $\mathcal{O}(l)$.

To avoid this behavior and since we are working in $\mathbb{F}_2$ the decision was obvious to choose a construct like a stl `std::bitset` [MS95]. Since this data structure is not very flexible, e.g. needs its size specified at compile time, the decision was made to use the `boost::dynamic_bitset` [Kar05]. Here during the runtime of the program the programmer can specify through variables the size of the bitset and resize it.

The comparison procedure now depends on the size of the bitset blocks and the number of variables n. Let us assume that our processor has 64 bit registers and the problem instance has n=10 variables. The current calculation step is as above and we want again compare vector a_1 to the model in order to find out if it is gluable or not. The vector a_1 consists now of two bitsets. The first one represents the values of the vector, the second one a mask. That represents the variable names. In the memory we would have a representation, bitwise, such as

$$\begin{aligned} a_1 &= (0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0) \\ a_1^m &= (0,1,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0) \end{aligned}$$

where a_1^m represents the mask. The advantage is that we can put our whole a_1 in one **int** variable since its size is usually 32 bit on a 64 bit x86 architecture; the same holds for a_1^m. Similar to our vectors we have to represent the model as a bitset of that kind and give it a mask, which indicates $X(k) = X_1 \cup X_2 \cup \ldots \cup X_k$. In other words, the variables which are through previous gluings already set in the model and have to be considered in the calculation. The model would now be

$$\begin{aligned} M_k &= (0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0) \\ M_k^m &= (1,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0) \end{aligned}$$

along with its mask M_k^m. To find out if a_1 is gluable with M_k it fits to apply the following algorithm with the input $M = (M_k, M_k^m)$ and $A = (a_j, a_j^m)$ where $\otimes$ denotes a bitwise "and" and $\oplus$ a bitwise "xor" operation.

Algorithm 11 Model Assignment Comparison

```
1: procedure GLUABLE(M, A)
2:     if not A.is_agreeing then
3:         return FALSE
4:     end if
5:     $T \leftarrow M_k$
6:     $T \leftarrow T \oplus a_j$
7:     $T \leftarrow T \otimes M_k^m$
8:     $T \leftarrow T \otimes a_k^m$
9:     if $T > 0$ then
10:        return FALSE
11:    else
12:        return TRUE
13:    end if
14: end procedure
```

The important point is that the comparison no longer depends on the sparsity and the number of variables in the first place. It depends more on the possible blocksize of the bitset. In a ”big” real world example with $n = 128, l = 10$ and a block with the datatype **unsigned long long int** of 64 bit the comparison takes always constant 8 steps. That are the bitwise ”xor” and ”and” operations and the determination if the outcome is > 0. One does not need to determine the subvectors $a_j[X(k)]$ and $M_k[X(j)]$. If the assignment is for some reason already not in an agreed state it is beforehand rejected.

In the algorithm above one might ask, why use a temporary variable T to calculate the result of the comparison. In the implementation itself the variable is **static**. The consequence is that only on the first call of the function space is allocated for the variable. Until the end of the program this space is neither deallocated nor reallocated. Since we have to store somewhere the result of our comparison it would be very inconvenient to allocate for every comparison the space of the size of the model again. Therefore, through the **static** declaration in the function this is done once and one can use that allocated memory over and over again.

One should also note, that all masks in that example are calculated beforehand and are accessible in $\Theta(1)$.

2.9.3 Class Definitions

An outline of the class definitions of the primitive datatypes, together with explanations is given in this paragraph. That means all logically important parts of the classes are mentioned and for clarity all too technical details, e.g. ”setter/getter methods”, are omitted in this documentation. Let us start with the most basic datatype, the assignment, which models a satisfying vector of an specific equation.

class ***Assignment*** : `boost::dynamic_bitset` <*BlockType*>	
Equation*	get_parent_equation()
`boost::dynamic_bitset` <*BlockType*>*	get_equation_projection(***Equation**** e)
`boost::dynamic_bitset` <*BlockType*>*	get_mask()
`std::vector` <***ProjectionContainer****>*	get_projection_containers()
bool	is_agreeing

Figure 2.2: Class Assignment

This class is a straightforward implementation of the strategy mentioned above. The ***Assignment*** class is inherited from the `boost::dynamic_bitset` <*BlockType*> and has therefore all its methods and operators, especially the "and" and "xor" operators which are used later on for the calculations. It contains additionally the function get_equation_projection(***Equation****
e) in order to determine the projection of the assignment to a given equation. This is done by referring to the address of the ***Equation*** object and storing beforehand (see below). The last function, get_projection_containers(), returns a vector of pointers to ***ProjectionContainer*** objects in which this assignment occurs. This is used later for the Agreeing2 Procedure, see below. The is_agreeing variable of the type **bool** indicates if the assignment is in an agreed or disagreed state to the current state of the calculation and is for performance reasons directly accessible.

class ***Equation***	
Branch*	get_parent_branch()
`std::vector` <**unsigned int**>*	get_variables()
`std::vector` <***Assignment****>*	get_assignments()
unsigned int	num_agreeing_assignments

Figure 2.3: Class Equation

The ***Equation*** class reflects a symbol and contains therefore the variables and the assignments. Both collections are from the type `std::vector` <*Type*> to ensure that all information is stored back-to-back. The num_agreeing_assignments variable of type **unsigned int** indicates the number of assignments in an agreed state which are left in an agreeing state in the equation at the current state. If num_agreeing_assignments equals 0 it is clear that a contradiction of the current state to the solution of the equation system occurred and appropriate actions must be taken. Moreover the ***Equation*** class contains a pointer to a ***Branch*** object, which can be accessed, in order to determine at which depth of the tree the symbol occurs.

class ***Model*** : `boost::dynamic_bitset` <*BlockType*>	
`boost::dynamic_bitset` <*BlockType*>*	get_mask()

Figure 2.4: Class Model

The ***Model*** class is, like the ***Assignment***, just a inheritance of the `boost::dynamic_bitset` <*BlockType*> to have the necessary operators and the storing strategy mentioned above. It holds the mask information ready for access.

2.9.4 Tree Representation

Since the program implements a tree-search, or a backtracking algorithm, we introduce here the parts crucial for this algorithm.

class ***Branch***	
Equation*	get_equation()
Model*	get_model()
`std::vector` <**unsigned int**>*	get_variables()
AssignmentsToModelIterator*	get_assignments_current()
AssignmentsToModelIterator*	get_assignments_end()
void	reset_iterators()

Figure 2.5: Class Branch

The ***Branch*** class represents literally the branch of the tree. At every depth of the tree position we have an equation through sorting and a model which stands for our (partial) intermediate solution. Moreover we have at every tree depth i a set of variables $X(i)$ which is accessible through the method get_variables(). The methods get_assignments_current() and get_assignments_end() are responsible for returning iterators to assignments which are appropriate for gluing to the current model. The ***AssignmentsToModelIterator*** type is explained in section 2.9.8. If during the tree search a step back is performed, which means that we found a partial solution is not applicable, these iterators are reset through the method reset_iterators().

class ***Tree*** : `vector` <***Branch****>	
bool	has_next()
Branch*	current()
Branch*	next()
Branch*	last()
void	forward()
void	back()
unsigned int	pos

Figure 2.6: Class Tree

The ***Tree*** class represents the structure for the backtracking algorithm. The function has_next() indicates if we are at the end of our tree search or if we have to continue with normal operation. The function current() returns a pointer to the current ***Branch*** object; the same holds for next() and last() respectively for the next and the last ***Branch*** object. The forward() and backward() routines are setting the state of the tree to one step back or one step forward. The internal variable pos indicates the current tree depth.

2.9.5 Full Agreeing Representation

struct ***ProjectionContainer*** : std::vector <***Assignment****>	
ProjectionTuple*	parent
unsigned int	num_agreeing_assignments

Figure 2.7: ProjectionContainer struct

If we take a look at the structure of the Agreeing2 we are working with tuples $\{A, B\}$ of pairwise agreeing subvectors of some equations. The ***ProjectionContainer*** class now represents one side of this tuple. The pointers to the assignments referring to one side are stored in the form of a `std::vector`. Additionally a counter of how many agreed assignments are currently present in that side of the tuple is available in the structure.

struct ***ProjectionTuple*** : std::pair <***ProjectionContainer****, ***ProjectionContainer****>

Figure 2.8: ProjectionTuple struct

Tuples of the form $\{A, B\}$ now are nothing else than a pair from the single containers above, here modelled as a `std::pair`.

struct ***FullAgreeingStructure***	
bool	run_agreeing2(***Model**** m)
void	undo()

Figure 2.9: FullAgreeingStructure class

The ***FullAgreeingStructure*** structure holds the information to run the Agreeing2 to the current instance on with a given model. Therefore the function run_agreeing2 has as only parameter a pointer to a model. Furthermore to undo changes, for example if a wrong guess was introduced to the equation system, the function undo() is offered. It sets the problem instance to the last known state, in particular the assignments which were set in the last call of run_agreeing2 to agreeing again.

2.9.6 Sorting

During the implementation of the program we developed several possibilities to sort the input instances. But since the sorting takes in comparison to the actual solving process a relatively short time even with a slow algorithm, most of the effort was done by me in enhancing the solving procedures. So a quite intuitive algorithm, presented below, was implemented but as one can see in the experiments section (**??**) it fits the needs to give reasonable results.

Let us assume, that the input E to the procedure is a `std::vector` of ***Equation**** objects and n is the number of variables in the equation system. The return value of the procedure is a `std::vector` of ***Equation**** objects in the best sorting with respect to the algorithmic structure. The operator $X(S_i)$ returns here the set of variables to an equation S_i.

Algorithm 12 Sorting Procedure

```
 1: procedure SORT($E$, $n$)
 2:     $R \leftarrow (E[0])$                ▷ The result list becomes $E[0]$ as first element
 3:     while $|E| > 0$ do
 4:         $size_{new} \leftarrow n + 1$
 5:         for $S \in E$ do
 6:             if $|\bigcup_i X(R(i)) \cup X(S)| < size_{new}$ then
 7:                 $S_{new} \leftarrow S$
 8:                 $size_{new} \leftarrow |\bigcup_i X(R(i)) \cup X(S)|$
 9:             end if
10:         end for
11:         $E \leftarrow E \setminus S_{new}$
12:         Append $S_{new}$ at $R$
13:     end while
14:     return $R$
15: end procedure
```

As one can see is to the previously determined result list R always the equation appended in which $\bigcup_i |X(i)|$ has the lowest growth.

2.9.7 The Agreeing2 Procedure

In the Agreeing2 Procedure we have to handle the set of assignments organized as tuples. A guess has to be introduced in the system and the outcome has to be determined. There exist again two possibilities of how to handle the algorithm, the iterative and the recursive way. In my implementation I choose the iterative way for speed reasons. The ***FullAgreeingStructure*** holds as private variables a vector of tuples of the datatype `std::vector` <***ProjectionTuple****>, here denoted as *pt_vector*. The second private structure hold by ***FullAgreeingStructure*** is a `std::vector` <***Assignment****>, denoted as *init_vector*, which holds all assignments which are suitable for Agreeing while introducing the guess. A third structure, for the iterative processing, here called *queue* and a fourth structure which keeps track of the empty tuples, here called *empty_tuples* are both of the type `std::vector` <***ProjectionTuple****>. The guess introduction as well as the Agreeing2 process are then handled as described below, where M is the introduced model. The functions DEC and INC are helper functions which decrease or increase an integer value. The variable *undo_assignments* keeps track of changed assignments. It is used to determine later which assignments were altered and have to set to an agreeing state back in case the *undo*() function is called.

Algorithm 13 Agreeing2 Procedure

```
procedure RUN_AGREEING2($M$)
    for $a \in init\_vector$ do          ▷ Introduce the guess in our set of predefined assignments
        if not GLUABLE(M,A) then
            $a.is\_agreeing \leftarrow$ FALSE
            DEC($a.get\_parent\_equation().num\_agreeing\_assignments$)
            for $pc \in a.get\_projection\_containers()$ do
                DEC($pc.num\_agreeing\_assignments$)
                if $pc.parent$ got one sided empty then
                    Put $pc.parent$ on $queue$
                end if
                if $pc.parent$ got both sided empty then
                    Put $pc.parent$ on $empty\_tuples$
                end if
            end for
            Put $a$ on $undo\_assignments$
        end if
    end for
    while $queue$ is not empty do          ▷ Run iteratively agreeing2 on the tuples
        $t \leftarrow queue.pop()$
        $p \leftarrow$ ProjectionContainer which is not empty of $t$
        for $a \in p$ do
            if $a.is\_agreeing$ then
                $a.is\_agreeing \leftarrow$ FALSE
                DEC($a.get\_parent\_equation().num\_agreeing\_assignments$)
                for $pc \in a.get\_projection\_containers()$ do
                    DEC($pc.num\_agreeing\_assignments$)
                    if $pc.parent$ got one sided empty then
                        Put $pc.parent$ on $queue$
                    end if
                end for
            end if
            Put $a$ on $undo\_assignments$
            if $a.get\_parent\_equation().num\_agreeing\_assignments = 0$ then
                return FALSE
            end if
        end for
        Put $p$ on $empty\_tuples$
    end while
    if $|empty\_tuples| = |pt\_vector|$ then
        return FALSE
    else
        return TRUE
    end if
end procedure
```

The undo() function for the Agreeing2, called whenever a revisit of the branch at point d occurs is done as follows. Remember that we stored all assignments which were set in the last run of the Agreeing2 Procedure in *undo_assignments*. The preparation of the Agreeing2 Algorithm, means the creation of the tuples is done in the preparation steps of the main program.

Algorithm 14 Undo Procedure

```
1: procedure UNDO
2:     for $a \in undo\_assignments$ do
3:         $a.is\_agreeing \leftarrow$ TRUE
4:         INC($a.get\_parent\_equation().num\_agreeing\_assignments$)
5:         for $pc \in a.get\_projection\_containers()$ do
6:             INC(pc.num_agreeing_assignments)
7:         end for
8:     end for
9: end procedure
```

2.9.8 The Solving Procedure

It is necessary to explain first the function of the type `boost::filter_iterator` in order to explain the implementation principle of the solving procedure. The solving procedure implements the Gluing Procedure which calls itself the Agreeing2 Procedure at a given depth d. The `boost::filter_iterator` is a class template from the Boost C++ Library [DA08] which becomes in my implementation the important role to find the assignments which are suitable for Gluing instead of determining them by sorting. It takes as template parameter a predicate function which determines assignments to skip. That means that while iterating through the assignments of an ***Equation*** object only those which are in an agreed state are returned. The following type of the iterator is created through a step forward in the tree. As template parameter it receives the function assignment_model_equality, the implementation of the Model Assignment Comparison Algorithm which determines which assignment should be rejected during the iterating process. It should be remarked here that only on a step forward is a new iterator created with the current model and the current vector of assignments as parameter.

```
boost::filter_iterator<
    std::binder2nd<assignment_model_equality<Assignment*, Model*> >,
    std::vector<Assignment*>::iterator >
```

Figure 2.10: Filter Iterator Type

Should the case occur, that the backtracking algorithm jumps back to a branch which holds an iterato, it is received through the function get_assignments_current() and the procedure of choosing the next possible Gluing is resumed from the last known point. Only a jump over the branch back that holds the iterators results in a deletion of them, since it can be assumed that the model has been altered and the assignment_model_equality function has also to be altered in the template parameter of ***Model****. Because these kind of iterators comes as pair to determine the end of the sequence we have to generate a second iterator which is simply a pointer to the end of the sequence.

The reason for the use of this construction is that sorting the assignments would take with a method like bucket sort constant $\Theta(2^{l-1})$ on the average. With the method of the `boost::filter_iterator`'s we are still in $\mathcal{O}(2^{l-1})$, but very often we are below this upper bound since it can be assumed that the resulting path of glueable assignments is somewhere between the beginning and the end of all assignments. Moreover, if we would like to sort the assignments, for example ascending by a numerical value or to handle a hash table the problem arises of how to reflect this value in the memory. To distinguish vectors which are defined in n different variables for $\mathbb{F}_2$ one needs at least 2^n different hash values, which would in that case immediately represent the vectors itself.

The parameter T represents the beforehand generated ***Tree*** object. The return type of the algorithm is a pointer to the final model which holds the result of the calculation. If the equation system is not satisfiable, e.g. there exists at some point a contradiction in the system, the function ***Tree***::back() is called for the position 0 at some point, an error occurs and the program quits.

In the algorithmic description below the Agreeing2 structure is denoted by FA and d is the beforehand determined treedepth at which the Agreeing2 should have been applied.

Algorithm 15 Solving Procedure

```
1: procedure SOLVE($T$)
2:     while $T.has\_next()$ do
3:         if $T.pos = d$ then
4:             FA.undo()
5:         end if
6:         $asc \leftarrow T.current().get\_assignments\_current()$
7:         $ase \leftarrow T.current().get\_assignments\_end()$
8:         if $asc \neq ase$ then
9:             $T.next().get\_model() \leftarrow T.current().get\_model() \circ *asc$
10:            $asc \leftarrow asc + 1$
11:            if $T.pos = d$ then
12:                if not FA.RUN_AGREEING2($T.next().get\_model()$) then
13:                    goto 2
14:                end if
15:            end if
16:            $T.forward()$
17:        else
18:            $T.back()$
19:        end if
20:    end while
21:    return $T.last().get\_model()$
22: end procedure
```

In the algorithm the line 9 applies the assignment to the current model and stores the result in the next branch as the model. If the Agreeing2 at line 12 yields a FALSE result, then we know that a wrong guess occurred and we jump back to the begin of our while loop. Since we incremented the iterator for the current assignments at line 10 we will receive the next vector for the Gluing and we can proceed.

Only if the algorithm runs over the tree depth d, the changes from the Agreeing2 are undone. That has the consequence that in the further gluing procedure it may occur that assignments are already set to a disagreed state, therefore by the filter_iterator dismissed.

2.9.9 The Main Program

The main program simply assembles all steps together. At first the data is read of a file and a set of preprocessing steps is started. Afterwards the solving procedure starts and the result of the procedure is printed out.

Algorithm 16 Main Program

```
1: procedure MAIN($T$)
2:     Read equation system $E$ from file
3:     SORT($E$)
4:     PREPARE($E$)
5:     print SOLVE($E$)
6: end procedure
```

2.9.10 Preprocessing

Since memory is considered here to be cheap and CPU time as expensive the implementation should tend in the direction to precalculate as much as possible beforehand. The preprocessing steps in the implementation do not give an exponential speedup but a quite remarkable polynomial speedup instead. This might be not important for the theoretical bound of the algorithms, but is obvious while running the program. Moreover slow memory operations and repetitively allocation of memory are avoided.

Sorting Although the sorting procedure is considered to be a separate step of the calculation it is factual calculated beforehand.

Set Preprocessing One point at which time can be saved is the preprocessing of set operations. It is a fact that during the tree walk the order of the equations is not changing, so we can preprocess most of the sets along our tree for every branch. The first is for every branch the $X(i)$ in form of a n-length bitmask which indicates the variables set at the given tree depth of the branch. Secondly the masks of the ***Model*** objects are pre calculated.

Agreeing2 Tuples Since we point out one d at which the Agreeing2 Algorithm should run, we can calculate statically all our tuples from the beginning on and let the general structure unchanged during the calculation.

Assignment Occurrences in Tuples In order to speed up the Agreeing2 Algorithm and to keep track in which tuple a specific assignment occurs, every assignments gets a list of addresses of tuples attached. This gives the opportunity to find in $\mathcal{O}(1)$ all tuples in which an assignment a occurs.

Assignment Projections Whenever two equations S_i, S_j have the property $X(S_i) \cap X(S_j) \neq \emptyset$ for every assignment of them the associated projection is calculated and stored in the assignment object. This gives in the preprocessing of the occurrences in the tuples an advantage as well as for further development of the routines.

Assignment Counters Instead of keeping track of the number of agreeing assignments in an equation or in an tuple by counting, the structures have a member variable which indicates the value. This makes it unnecessary to count always when this value is required. This fact is for example important during the Agreeing2 procedure.

Parental Pointers Every assignment object owns a pointer to its parental equation, that is to the equation where it belongs to. This makes it possible to directly decrement or increment the number of agreeing assignments in the assignment counters.

3 SAT Solving Techniques

In the following chapter different complete SAT solving techniques are presented. Complete means here that the outcome is always reliable in comparison to heuristic algorithms where the correctness of the outcome is determined with some probability.

Our focused problem instances are l-sparse algebraic equations over finite fields, so section 3.2 deals with the transformation of a given instance from this set to the set of SAT problem instances.

Most of the SAT solving techniques are based on the ideas behind the DP/DPLL backtrack search algorithm, therefore it is explained first in section 3.4 and then different improvements to this method are explained.

3.1 Basic Definitions

Definition 3.1 (Satisfiability Problem) *The decision version of the satisfiability problem is defined by*

$$SAT = \{\phi \mid \phi \text{ is a satisfiable boolean formula in CNF}\}.$$

For each fixed $l \geq 1$ the restriction of the SAT problem is

$$l\text{-}SAT = \{\phi \mid \phi \text{ is a satisfiable boolean formula in } l\text{-CNF}\}.$$

The task for a SAT solving algorithm is now to find out wether a given formula ϕ belongs to the set SAT or respectively to the set l-SAT, which would yield a satsifiying assignment for the given ϕ.

The two most common ways to express such a formula are the conjunctvie and disjunctive normal form.

Definition 3.2 (Conjunctive Normal Form Formula (CNF-Formula)) *A conjunctive normal form formula ϕ with n binary variables $x_1, x_2, \ldots, x_n$ is the conjunction of m clauses $C_1, C_2, \ldots, C_m$ of which each clause is the disjunction of one or more literals, where a literal is the affirmative*[1] *occurrence of a variable or as its negation.*

Definition 3.3 (Disjunctive Normal Form Formula (DNF-Formula)) *A disjunctive normal form formula ϕ with n binary variables $x_1, x_2, \ldots, x_n$ is the disjunction of m clauses $C_1, C_2, \ldots, C_m$ of which each clause is the conjunction of one or more literals, where a literal is affirmative the occurrence of a variable or as its negation.*

3.2 Conversion to SAT

Next it will be describe an easy and intuitive way how a sparse algebraic equation over a finite field of characteristic 2 can be transformed to an instance of the SAT problem. Consider the equation system (1.1). For every equation f_i one creates a truth table T for all possible assignments. One tests now all possible assignments if they satisfy f_i. If they satisfy the equation they are removed from T.

[1]The occurence of a variable without negation is called an *affirmative occurence.*

Every truth assignment which still resides in T represents now a disjunction of the involved variables and every $true$ assignment stands for a negated variable and every $false$ assignment stands for a affirmative variable. The concatenation of the clauses gives us the equivalent solvable conjunctive normal form formula for f_i.

It is easy to see, why this transformation works. If one considers all satisfying assignments of $f_i(X_i) = 0$, the formula could be equally described as the disjunction of this assignments, which group themselves their variables as conjunction. This would yield a DNF formula. And converting a DNF formula to a CNF formula is easily done by eliminating all satisfying assignments from a truth table and grouping them together like mentioned above.

This transformation is due to a fixed l computable in $O(m2^l)$, so has a polynomial complexity with respect to a fixed l.

Example Consider the equation

$$f(x_1, x_3, x_4) = x_1x_3 \oplus x_4 = 0$$

so the satisfying assignments are

$$\{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\}.$$

Applied the step of deletion of this assignments to the whole truth table T of the variables x_1, x_3, x_4 one obtains

$$\{(0,0,1),(0,1,1),(1,0,1),(1,1,0)\}$$

which can be read in conjunctive normal form as

$$\phi = (x_1 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_3} \vee \overline{x_4}) \wedge (\overline{x_1} \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4)$$

Every assignment to this boolean formula ϕ is satisfying if and only if it is satisfying for f_i.

3.3 General Structure of a SAT-Algorithm

Here is a short summary about the general procedure of finding a satisfying assignment to a given formula ϕ given in order to establish the terms of a backtracking algorithm.

Starting with an empty assignment to a formula ϕ a backtracking algorithm traverses the search space of assignments in order to find a satisfying one. In doing so it maintains an implicated search tree. Each branch in this search tree can be seen as a point of decision by either local heuristics or by the result of a further branch. The depth of the branch in the tree is referred to as the decision level. The algorithm steps iteratively through the different decision levels according to the following steps:

1. Extend the current assignment. The main purpose here is to find a most appropriate new variable assignment for an unassigned variable. This has the aim to explore new regions of the search space. The algorithm terminates if all clauses get satisfied or no more possible new variable assignment to an unbound variable can be done. In the last case the remaining search space got empty and the given formula is unsatisfiable.

2. Propagate the assignment to the formula. In this state the solving algorithm derives implications from a given assignment, also referred to as the deduction process. During this process conflicts in so called conflict-clauses may arise. This conflict-clause would be an unsatisfied clause in terms of the current assignment. The assignment is therefore a contradicting assignment.

3. Undo the last decision made if a conflict arrises. This is called backtracking and allows the algorithm to start at step 1. to explore the search space further in another direction.

The complexity of this steps depends on how decisions are made and of course the problem instance itself.

3.4 DP and DPLL

DP In this section it follows the description of the Davis-Putnam algorithm[DP60]. The complete DP algorithm has the intention to proof a formula of quantification theory but uses generally techniques which can be used to solve SAT problem instances.

Basic DP Algorithm Rules Let ϕ be a formula in conjunctive normal form so the core of the algorithm uses the following rules.

1. *Rule for the Elimination of One-Literal Clauses:*
   a) If ϕ contains a one-literal clause x_i, that is a clause which contains only one variable, and contains also its negation as one-literal clause, namely $\overline{x_i}$, then ϕ has no satisfying assignment.
   b) If a) does not apply and if ϕ contains a one-literal clause x_i, then one may modify ϕ by striking out all clauses that contain x_i affirmatively and delete all occurrences of $\overline{x_i}$ from the remaining clauses.
   c) If a) does not apply and if ϕ contains a one-literal clause $\overline{x_i}$, then one may modify ϕ by striking out all clauses that contain $\overline{x_i}$ and delete all occurrences of x_i from the remaining clauses.
   d) In cases b) and c), if the modified ϕ gets empty, then ϕ has a satisfying assignment.

2. *Affirmative-Negative Rule.* If a variable x_i is only occurring as affirmative literal or if x_i is only occurring as its negation, then one may delete all clauses which contain x_i. (If ϕ' is empty it is satisfiable.)

3. *Rule of Eliminating Atomic Formulas*[2]. Let ϕ be put into the form $(A \vee x_i) \wedge (B \vee \overline{x_i}) \wedge R$, where A, B and R are formulas, and free of x_i, then ϕ is satisfiable if and only if $(A \vee B) \wedge R$ is satisfiable. (To put ϕ in to the form mentioned above one can group together the clauses containing x_i and then factor out x_i to obtain the expression $(A \vee x_i)$. The same procedure with respect to $\overline{x_i}$ is applicable to the expression $(B \wedge \overline{x_i})$. The remaining clauses are then grouped into R.)

Proof 3.4 *To verify that the single rules of the DP algorithm are correct it is shown one by one that applying them to a initial instance ϕ leads to a satisfiable transformation ϕ' if and only if the initial instance was satisfiable.*
Rule 1) The case (a) is obvious, $x_i \wedge \overline{x_i} = 0$, so the system is not satisfiable. Case (b) is justified with the observation that for every $\phi = x_i \wedge A$, where x_i is a one-literal clause, ϕ is solvable if and only if $x_i = 1$. Since we are working in CNF this yields automatically that every clause which contains x_i becomes true and can be striked out. In every clause which contains $\overline{x_i}$ it can be dismissed, since this literal will become false, so can be striked out. Case (c) is analog to case (b). Case (d) reduces to the observation that if all literals and clauses deleted from ϕ so x_i occurred in all clauses and ϕ is satisfiable.

[2]The term „Atomic Formula" is used in the original paper [DP60] and is describing an expression $p(p_1, p_2, \ldots, p_i)$ if p is a predicate symbol and $p_1, p_2, \ldots, p_i$ are terms. With respect to our problem instances (CNF formulas) this terminus is not applicable but is used to preserve the rule name.

Rule 2) Let x_i occur in $\phi = (A \wedge R)$ only affirmatively, where A is the conjunction of clauses containing x_i and R be x_i-free. Since $A = 1$ for $x_i = 1$ the satisfiability depends only on R, which means $(A \wedge R) \Leftrightarrow R$. The justification of the rule is similar to the case if x_i occurs only as its negation.
Rule 3) The formula $\phi = (A \vee x_i) \wedge (B \vee \overline{x_i}) \wedge R$ might be satisfiable if $x_i = 1$ or $x_i = 0$. For the first case that means $(A \wedge R) = 1$ and for the second case $(B \wedge R) = 1$. Since one case must be true to keep ϕ satisfiable that results ins $\phi' = (A \wedge R) \vee (B \wedge R) \Leftrightarrow (A \vee B) \wedge R$. □

Examples To illustrate the basic rules of the DP algorithm here some reformulated example instances of the original publication are presented. Consider the following formulas in CNF:

1. $\phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2}) \wedge \overline{x_1} \wedge x_3$
   By rule 1 there are two one-literal clauses ($\overline{x_1}$ and x_3) we can eliminate. After eliminating $\overline{x_1}$ we get $\phi' = (x_2 \vee \overline{x_3}) \wedge \overline{x_2} \wedge x_3$ and eliminating x_3 leads us to $\phi'' = x_2 \wedge \overline{x_2}$ which is a contradiction and ϕ is therefore not satisfiable.

2. $\phi = (x_1 \vee x_2) \wedge \overline{x_2} \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$
   By eliminating the one-literal occurrence of $\overline{x_2}$ we proceed with $\phi' = x_1 \wedge (\overline{x_1} \vee \overline{x_3})$ which in turn yields by rule 3 $\phi'' = \overline{x_3}$, so ϕ is satisfiable.

3. $\phi = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3})$
   One can observe that the variable x_3 occurs only as its negation which results by rule 2 in $\phi' = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2)$. Applying rule 3 to ϕ' one can obtain the form $\phi'' = x_1 \vee \overline{x_1}$ which leads to $\phi'' = 1$, so ϕ is satisfiable.

DPLL The only difference in the DPLL algorithm[DLL62] which can be called an extension or modification to the DP algorithm is the exchange of rule 3 to the following rule:

3*. *Splitting Rule.* Let the given formula ϕ be put in the form

$$\phi' = (A \vee x_i) \wedge (B \vee \overline{x_i}) \wedge R,$$

where A, B, R do not depend on x_i. So ϕ is satisfiable if and only if $(A \wedge R)$ or $(B \wedge R)$ is satisfiable.

The proof of correctness is obviously the same as for rule 3.

The reason for the new *Splitting Rule* is described mostly technical since the *Rule of Eliminating Atomic Formulas* easily increased the number and length of the clauses. It is also stated in [DLL62] that the observation of many duplicated, thus redundant, clauses after performing rule 3 was made.

The Algorithm Now we are ready to assemble everything together and to obtain the algorithmic structure of the DPLL algorithm. By convention let ϕ be our input formula in CNF and $\phi(x_i)$ the formula with the variable x_i set to 1 (similar for $\phi(\overline{x_i})$, where as $\overline{x_i} = 1$). A monotone literal is called a variable which occurs only affirmatively or as its negation. The algorithm returns whenever an input formula is satisfiable SATISFIABLE and if not UNSATISFIABLE. To apply rule 3 it is stated to choose a variable, and both, original DP and original DPLL are choosing the first variable from the first clause of minimal length.

Algorithm 17 DPLL algorithm

```
procedure DPLL($\phi$)
    while $\phi$ contains an one-literal clause do                ▷ Rule 1
        if $\phi$ has an empty clause then
            return UNSATISFIABLE
        end if
        $v \leftarrow$ any one-literal clause
        $\phi \leftarrow \phi(v)$
    end while
    while $\phi$ contains a monotone literal do                   ▷ Rule 2
        $v \leftarrow$ any monotone literal
        $\phi \leftarrow \phi(v)$
    end while
    if $\phi$ is empty then
        return SATISFIABLE
    end if
    $x \leftarrow$ choose a variable in $\phi$                    ▷ Rule 3*
    if DPLL($\phi(x)$) returns SATISFIABLE then
        return SATISFIABLE
    end if
    if DPLL($\phi(\overline{x})$) returns SATISFIABLE then
        return SATISFIABLE
    end if
    return UNSATISFIABLE
end procedure
```

Obtaining Satisfying Assignment On the first look it might not seem that the DPLL algorithm also yields a satisfying assignment for the given instance ϕ, but implicitly it is generated through the algorithm rules. By applying rule 1 to the given formula one knows, that the one-literal clause, e.g. $C = (x_i)$, must have the value 1, since ϕ is a disjunction of clauses. By applying the rule 2 the value of the literal can be assumed to be 1. Whenever the third rule is applied a value is guessed and if the outcome is SATISFIABLE it can be assumed as a right guess.

3.5 Algorithmical Improvements

While the DPLL algorithm is the most common used general structure of SAT-solvers like MiniSAT [ES03, ES04, EB05], Chaff[MMZ+01] and is still used as the basic foundation for more sophisticated algorithms like GRASP[MSS96], many ideas for the improvement algorithmically were introduced in this solvers to speed up the process of solving of a Boolean formula.

The following sections should give an overview about the different techniques used in modern SAT-solvers.

The set of clauses given by the instance is called a clause database. The stage, that is the depth in the decision tree at which a value for a variable x_i was chosen, is denoted by $\delta(x_i)$. Then $x_i = v$ @ d means the variable x_i got assigned the value v at tree depth d.

3.6 Branching Heuristics

One important part in the DPLL algorithm, or in backtracking algorithms also called for our problem BCP (Boolean constraint propagation), is the fact that it recursively propagates knowledge obtained and chooses successively new variables to assign them with a value not yet tried in that branch of the tree. This process of choosing a variable and assigning it a value is called a branching heuristic which differs from algorithm to algorithm. Here a short overview of two different branching heuristics is given as presented in [MS99].

3.6.1 MOM's Heuristic

One of the most well-known and utilized branching heuristics is the Maximum Occurrences of clauses of Minimum size (MOM's) heuristic [DABC93, ZM88, Pre96, Fre95].

Let $f^*(l)$ be the number or occurrences of a literal l in the smallest non-satisfied clauses. It is widely accepted that a good variable to select is one that maximizes the function

$$[f^*(x) + f^*(\overline{x})] * 2^k + f^*(x) * f^*(\overline{x}). \tag{3.1}$$

Intuitively, preference is given to variables x with a large number of clauses in x or in $\overline{x}$ (assuming k is chosen to be sufficiently large), and also to variables with a large number of clauses in both x and $\overline{x}$. Several variations of MOM's heuristic have been proposed in the past with heuristic functions related to but different from (3.1). A detailed description of MOM's heuristics can be found in [Fre95]. We should also note that in general we may also be interested in taking into account not only the smallest clauses, but also clauses of larger sizes.

3.6.2 Jeroslow-Wang Heuristic

Two branching heuristics were proposed by Jeroslow and Wang in [JW90], and are also analyzed in [Bar95, BS97]. For a given literal l let us compute:

$$J(l) = \sum_{l \in \omega \wedge \omega \in \phi} 2^{-|\omega|} \tag{3.2}$$

The one-sided Jeroslow-Wang (JW-OS) branching heuristic selects the assignment that satisfies the literal with the largest value $J(l)$. The two-sided Jeroslow-Wang (JW-TS) heuristic identifies the variable x with the largest sum $J(x) + J(\overline{x})$, and assigns to x value true, if $J(x) \geq J(\overline{x})$, and value false otherwise.

For another comparison of branching heuristics and how to "fool" some of them one should refer to [Ouy98].

3.7 Conflict Induced Clauses

The term *conflict-induced clause* is defined through the work on the algorithm GRASP[MSS96] by M. Silva and K. Sakallah. The method describes a way of dynamic learning during a backtracking process in order to prune a large space of the search-tree. In simple terms it is one among other algorithmical methods to extend the clause database to avoid wrong guesses already made. The description here is made with the help of examples of the original presentation in [MSS96].

If a conflict arrises during the backtracking process we naturally do not want to run in the same situation again. To avoid this case we can create a conflict-induced clause. Let us assume we have the following clause database

$$\begin{aligned}
\omega_1 &= (\overline{x_1} \vee x_2) \\
\omega_2 &= (\overline{x_1} \vee x_3 \vee x_9) \\
\omega_3 &= (\overline{x_2} \vee \overline{x_3} \vee x_4) \\
\omega_4 &= (\overline{x_4} \vee x_5 \vee x_{10}) \\
\omega_5 &= (\overline{x_4} \vee x_6 \vee x_{11}) \\
\omega_6 &= (\overline{x_5} \vee \overline{x_6}) \\
\omega_7 &= (x_1 \vee x_7 \vee \overline{x_{12}}) \\
\omega_8 &= (x_1 \vee x_8) \\
\omega_9 &= (\overline{x_7} \vee \overline{x_8} \vee \overline{x_{13}})
\end{aligned}$$

and a partial assignment

$$\{x_9 = 0 \;@\; 1, x_{10} = 0 \;@\; 3, x_{11} = 0 \;@\; 3, x_{12} = 1 \;@\; 2, x_{13} = 1 \;@\; 2\}$$

Now we are in our computation at $d = 6$ and we try the value for $x_1 = 1 \;@\; 6$. This would yield us the following partial implication graph $I = (V, E)$ for this tree depth which is defined as follows. Let the assignment of a variable x_i be implied due to a clause $\omega = (x_1, \dots, x_k)$ where the antecedent assignment of x_i, denoted as $A(x_i)$, is defined as the set of assignments to variables other than x_i with literals in ω. For example, the antecedent assignments of x_1, x_2 and x_3 due to the clause $\omega = (x_1 \vee x_2 \vee \overline{x_3})$ are, respectively, $A(x_1) = \{x_2 = 0, x_3 = 1\}, A(x_2) = \{x_1 = 0, x_3 = 1\}, A(x_3) = \{x_1 = 0, x_2 = 0\}$.

Definition 3.5 (Implication Graph[MSS96]) *An implication graph $I = (V, E)$ is defined by the following rules:*

1. *Each vertex in I corresponds to a variable assignment $x = v(x)$.*
2. *The predecessors of vertex $x = v(x)$ in I are the antecedent assignments $A(x)$ corresponding to the unit clause ω that led to the implication of x. The directed edges from the vertices in $A(x)$ to vertex $x = v(x)$ are all labeled with ω. Vertices that have no predecessors correspond to decision assignments.*
3. *Special conflict vertices are added to I to indicate the occurrence of conflicts. The predecessors of a conflict vertex κ correspond to variable assignments that force a clause ω to become unsatisfied and are viewed as the antecedent assignments $A(\kappa)$. The directed edges from the vertices in $A(\kappa)$ to κ are all labeled with ω.*

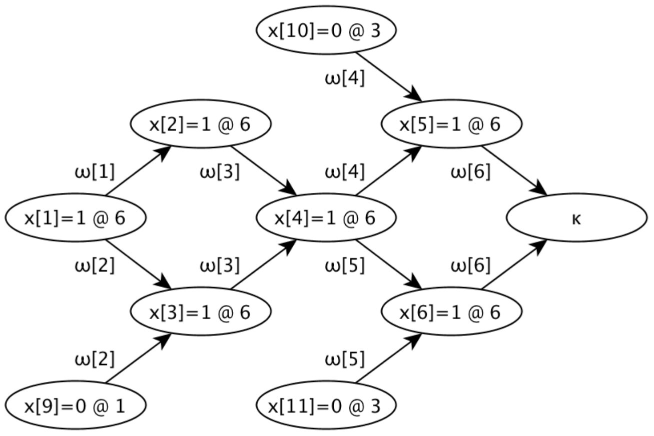

Figure 3.1: Implication Graph

As one can see this new assignment of x_1 leads to the contradiction κ. If one interprets this implication graph it is obvious that from the outermost nodes, namely the decisions, the assignment

$$A_C(\kappa) = \{x_1 = 1 \ @ \ 6, x_9 = 0 \ @ \ 1, x_{10} = 0 \ @ \ 3, x_{11} = 0 \ @ \ 3\}$$

is the reason for this contradiction during the computation. In terms of a conjunction

$$\zeta_C = (x_1 \wedge \overline{x_9} \wedge \overline{x_{10}} \wedge \overline{x_{11}})$$

led to the conflict. In order to prevent the algorithm to use again this partial assignment one could now insert a new clause in the clause database which is the negation of the reason, namely

$$\omega_C(\kappa) = (\overline{x_1} \vee x_9 \vee x_{10} \vee x_{11})$$

By inserting this new implicate in the clause database we prevent the algorithm at any depth of the search to step into the same conflicting assignment again. Remark that only variables are included into the conflict clause whose decisions are made at the accordant depth d or at depths $< d$. This is justified by the fact that only this variable assignments led to the conflict.

Now we will give a set of rules to construct such an assignment. One can determine the decision level of an implied variable x with its antecedents by

$$\delta(x) = max\{\delta(y)|(y, v(y)) \in A(x)\}$$

and x denotes either κ or a variable that is assigned at the current decision level. At first we split $A(x)$ into two sets

$$\Delta(x) = \{(y, v(y)) \in A(x)|\delta(y) < \delta(x)\}$$
$$\Sigma(x) = \{(y, v(y)) \in A(x)|\delta(y) = \delta(x)\}$$

The conflicting assignment $A_C(\kappa)$ can now be determined by the recursive formula

$$A_C(x) = \left\{ \begin{array}{ll} (x, v(x)) & \text{if } A(x) = \emptyset \\ \Delta(x) \cup \left[\bigcup_{(y,v(y)) \in \Sigma(x)} A_C(y) \right] & \text{otherwise} \end{array} \right\}$$

starting with $x = \kappa$. The conflict-induced clause for $A(\kappa)$ is now determined by

$$\omega_C(\kappa) = \bigvee_{(x,v(x)) \in A_C(\kappa)} x^{v(x)}$$

where $x^0 \equiv x$ and $x^1 \equiv \overline{x}$.

By introducing successively these clauses to the clause database one can prune large spaces of the search tree but makes the problem instance at the same time also more complicated, means that it becomes larger. This can lead to a problem if it grows to big. Therefore most SAT solvers are supporting limits for the introduction of this clauses and are from time to time deleting them.

One example for this deleting behavior is given as the *Rule of Decaying* in Chaff[MMZ+01] where every conflict-induced clause gets a specific value which is reduced by some rule every time the clause database is updated. The value is increased in case a conflict arrises through this clause. If the value drops beyond a specific threshold the conflict-induced clause is removed from the clause database.

3.8 Non-Chronological Backtracking

General backtracking algorithms like DPLL work chronological. The intention is to learn dynamically from errors made and to jump a branch back in the search tree (either recursively or iteratively) to try another variable assignment in case the predecessor failed in both ways. If that happened one knows, that at a previous depth a wrong guess was made and this satisfies the backtracking of one step. Chronological is in that case that on any problem occurred, namely a conflict arrised, it is assumed that the last decision made is the most probable causing this problem. Therefore the last branch is taken and the assignment made is tried in its opposite way. The algorithmical improvement is to analyze at what tree depth the wrong guess was made.

If we take again a look at our previous example from 3.7 we can determine that inserting the conflict induced clause would immediately yield a unit clause and that the variable x_1 is determined to be $x_1 = 0$ @ 6, since we are still at tree depth $d = 6$. That gives the following implication graph.

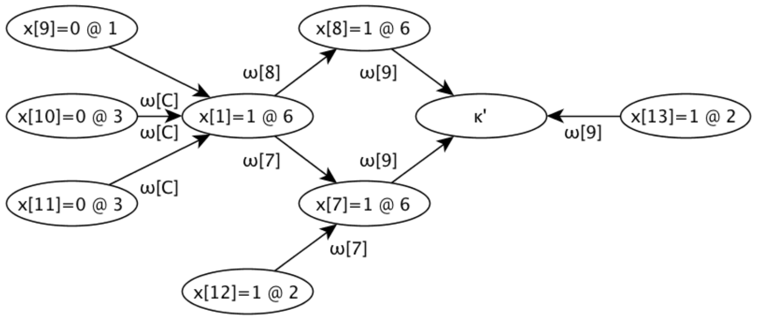

Figure 3.2: Implication Graph II

Obviously another conflict κ' occurred. Where the conflicting assignment is

$$A_C(\kappa') = \{x_9 = 0 \text{ @ } 1, x_{10} = 0 \text{ @ } 3, x_{11} = 0 \text{ @ } 3, x_{12} = 1 \text{ @ } 2, x_{13} = 1 \text{ @ } 2\}$$

along with the conflict-induced clause

$$\omega(\kappa') = (x_9 \vee x_{10} \vee x_{11} \vee \overline{x_{12}} \vee \overline{x_{13}})$$

We can already conclude from the fact that in $\omega_C(\kappa')$ are only occurring variables that are not assigned or determined at our current decision level $d = 6$ that the reason for the conflict can only be related to a previous branch at a depth $< d - 1$. A chronological backtracking engine would now jump one step back and eventually waste time by investigating futile branches. To avoid this one can determine the backtrack level β by

$$\beta = max\{\delta(x)|(x, v(x)) \in A_C(\kappa')\}$$

If however $\beta = d - 1$ then it is obvious that we jump chronologically. In the case that $\beta < d - 1$ we have a non chronological back-jump and go back several decisions.

In our case $\beta = 3$ and thus the following figure illustrates that behavior for the given example.

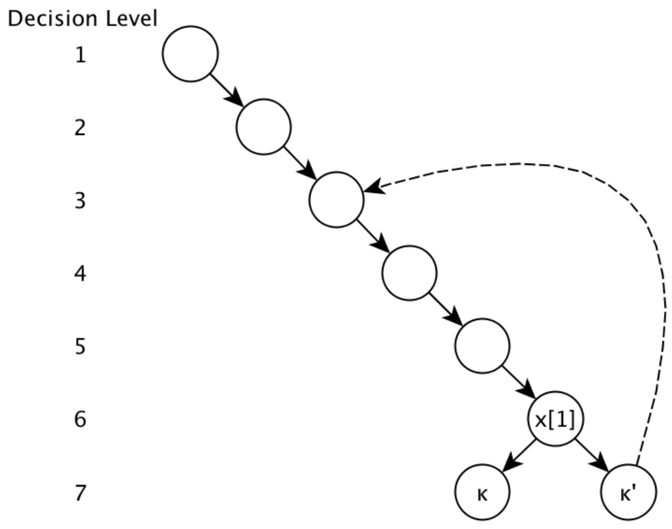

Figure 3.3: Decision Tree

A justification of the method can be found in [GCE93].

3.9 Watched Literals

One key factor for a SAT solver is an efficient BCP engine since the general backtracking algorithm spends the most time in jumping forth and back in its tree structure. One strategy to do this is by so called watched literals[MMZ$^+$01].

Every time a decision is made a BCP algorithm searches through the clauses for one-literal implications. This progress stops if no new one-literal implication can be found. The goal is to determine and visit those clauses which became newly empty. One intuitive approach would be to keep a counter for the false assignments per clause and to step through the list of clauses in order to find one which contains a literal that the current assignment sets to 0.

But it is not necessary to visit a clause if $1, 2, \ldots, N - 1$ variables are set to zero, if N is the number of literals in the clause. The only important moment is to visit a clause is if the counter of the number of variables assigned to 0 changes from $N - 2$ to $N - 1$, which would yield an implication.

In order to realize that one can create so called watched literals per clause. That means one just observes two variables per clause and one needs only to visit the clause if one of this variables gets assigned 0. If the clause is visited the following conditions must hold:

1. The clause is not implied. That means that there is one literal left which is not yet assigned to 0. The one watched literal gets now replaced by the other which is not assigned 0.
2. The clause is implied. Follow the one-literal rule and follow the implication from the other watched literal.

Example Consider the following clause

$$\omega = (\underline{x_1} \vee \underline{x_2} \vee x_3 \vee \overline{x_4}),$$

where $\underline{x_i}$ denotes a watched literal and we have $N = 4$ literals. At some tree depth d_i becomes $x_4 = 1$ @ d_i. Nothing has to be done in that case, since we only visit the clause in the case one of the watched literals gets assigned 0. Now $x_1 = 0$ @ $d_i + 1$ and by our strategy of watched literals we have to visit the clause. The condition holds that the clause is not yet implied. That is that we have still 2 free variables in the clause not assigned to 0 which are x_2 and x_3. Thus we replace the status of being watched of x_1 by x_3. That results in

$$\omega = (x_1 \vee \underline{x_2} \vee \underline{x_3} \vee \overline{x_4}).$$

If now either x_2 or x_3 gets at some point assigned 0 we know that the clause becomes implied, which would force the other variable to be 1. That is if x_2 gets assigned 0 we know that x_3 has to be 1 and vice versa. So no counter has to be maintained and number of visits to a clause is reduced.

This process may speedup the BCP process tremendous since only clauses are visited where the chance is given that they really became one-literal through the last decision or the last implication which was passed down. Furthermore should be remarked that with this technique the unassigning process of an variable for a backtracking jump can be done in constant time.

4 Gröbner Basis Algorithms

The subject to the following chapter is the Gröbner basis and algorithms related to it. The roots of this method can be found in investigating the question inherited by following geometrical definition.

Definition 4.1 *Let k be a field and let $f_1, \ldots, f_m$ be polynomials in $k[x_1, \ldots, x_n]$. Then we set*

$$V(f_1, \ldots, f_2) = \{(a_1, \ldots, a_n) \in k^n : f_i(a_1, \ldots, a_n) = 0 \text{ for all } 1 \leq i \leq m\}. \tag{4.1}$$

*We call $V(f_1, \ldots, f_m)$ the **affine variety** defined by $f_1, \ldots, f_m$.*

As one can see (4.1) is the set of all solutions of the system of equations $f_1(X_1) = 0, \ldots, f_m(X_m) = 0$. If the condition holds that the size of all sets of variables $X_i, 1 \leq i \leq m$ is $\leq l$ we have an equivalent equation system to (1.1).

With the following definitions and algorithms we establish a method how to solve this equations in an algebraic way.

4.1 Basic Definitions and Lemmas

Definition 4.2 *A subset $I \subset k[x_1, \ldots, x_n]$ is an **ideal** if it satifies:*

1. *$0 \in I$.*
2. *If $f, g \in I$, then $f + g \in I$.*
3. *If $f \in I$ and $h \in k[x_1, \ldots, x_n]$, then $hf \in I$.*

Definition 4.3 *Let $f_1, \ldots, f_m$ be polynomials in $k[x_1, \ldots, x_n]$. Then we set*

$$\langle f_1, \ldots, f_m \rangle = \left\{ \sum_{i=1}^{m} h_i f_i : h_1, \ldots h_m \in k[x_1, \ldots, x_n] \right\}. \tag{4.2}$$

Lemma 4.4 *If $f_1, \ldots, f_m \in k[x_1, \ldots, x_n]$, then $\langle f_1, \ldots, f_s \rangle$ is an ideal of $k[x_1, \ldots, x_n]$. We will call $\langle f_1, \ldots, f_m \rangle \in k[x_1, \ldots, x_n]$ the **ideal generated by** $f_1, \ldots, f_m$. The latter is called basis of the ideal.*

Proof 4.5 *The proof that (4.2) is an ideal can be found in [CLO92].*

An ideal can be seen as all polynomial consequences of the initial equation system. For example for $h_i \in k[x_1, \ldots, x_n], 1 \leq i \leq m$ we can obtain

$$h_1 f_1 + h_2 f_2 + \ldots + h_m f_m$$

which is exactly an element of (4.2) and it is zero on (4.1).

In order now to utilize (4.1) we state the following lemmas, both proved in [CLO92].

Lemma 4.6 *If $f_1, \ldots, f_m$ and $g_1, \ldots, g_t$ are bases of the same ideal in $k[x_1, \ldots, x_n]$, so that $\langle f_1, \ldots, f_m \rangle = \langle g_1, \ldots, g_t \rangle$, then $V(f_1, \ldots, f_m) = V(g_1, \ldots, g_t)$.*

4.2 Monomial Orderings and Division Algorithm

Definition 4.7 *A* ***monomial ordering*** *on $k[x_1, \dots, x_n]$ is any relation $\geq$ on $\mathbb{Z}^n_{\geq 0}$, or equivalently, any relation on the set of monomials x^α, $\alpha \in \mathbb{Z}^n_{\geq 0}$, satisfying:*

1. *$\geq$ is a total (or linear) ordering on $\mathbb{Z}^n_{\geq 0}$.*
2. *If $\alpha \geq \beta$ and $\gamma \in \mathbb{Z}^n_{\geq 0}$, then $\alpha + \gamma \geq \beta + \gamma$.*
3. *$\geq$ is a well-ordering on $\mathbb{Z}^n_{\geq 0}$. This means that every nonempty subset of $\mathbb{Z}^n_{\geq 0}$ has a smallest element under $\geq$.*

Definition 4.8 (Lexicographic Order) *Let $\alpha = (\alpha_1, \dots, \alpha_n)$, and $\beta = (\beta_1, \dots, \beta_n) \in \mathbb{Z}^n_{\geq 0}$. We say $\alpha \geq_{lex} \beta$ if, in the vector difference $\alpha - \beta \in \mathbb{Z}^n$, the left-most nonzero entry is positive. We will write $x^\alpha \geq_{lex} x^\beta$ if $\alpha \geq_{lex} \beta$.*

It should be mentioned, that there exist several other orderings, for example the *Graded Reverse Lex Order*[CLO92].

Definition 4.9 *Let $f = \sum_\alpha a_\alpha x^\alpha$ be a nonzero polynomial in $k[x_1, \dots, x_n]$ and let $\geq$ be a monomial order.*

1. *The* ***multidegree*** *of f is*
$$multideg(f) = max(\alpha \in \mathbb{Z}^n_{\geq 0} : a_\alpha \neq 0)$$
where the maximum is taken with respect to $>$.
2. *The* ***leading coefficient*** *of f is*
$$LC(f) = a_{multideg(f)} \in k.$$
3. *The* ***leading monomial*** *of f is*
$$LM(f) = x^{multideg(f)}$$
with its coefficient 1.
4. *The* ***leading term*** *of f is*
$$LT(f) = LC(f)LM(f).$$

4.3 Gröbner Basis and Reduced Gröbner Basis

Definition 4.10 *Let $I \subset k[x_1, \dots, x_n]$ be an ideal other than $\{0\}$.*

1. *We denote by $LT(I)$ the set of leading terms of elements of I. Thus,*
$$LT(I) = \{cx^\alpha : \text{ there exists } f \in I \text{ with } LT(f) = cx^\alpha\}.$$
2. *We denote by $\langle LT(I) \rangle$ the ideal generated by the elements of $LT(I)$.*

Definition 4.11 *Fix a monomial order. A finite subset $G = \{g_1, \dots, g_t\}$ of an ideal I is said to be a* ***Gröbner basis*** *(or* ***standard basis****) if*
$$\langle LT(g_1), \dots, LT(g_t) \rangle = \langle LT(I) \rangle.$$

Definition 4.12 *A* ***reduced Gröbner basis*** *for a polynomial ideal I is a Gröbner basis G for I such that:*

1. *$LC(p) = 1$ for all $p \in G$.*
2. *For all $p \in G$ no monomial of p lies in $\langle LT(G - \{p\}) \rangle$.*

4.4 Buchberger's Algorithm

In order to compute a Gröbner basis we state the following algorithm whose proof for correctness can be found in [CLO92].

Definition 4.13 *Let $f, g \in k[x_1, \ldots, x_n]$ be nonzero polynomials.*

1. *If multideg$(f) = \alpha$ and multideg$(g) = \beta$, then let $\gamma = (\gamma_1, \ldots, \gamma_n)$, where $\gamma_i = max(\alpha_i, \beta_i)$ for each i. We call x^γ the least common multiple of LM(f) and LM(g), written $x^\gamma = LCM(LM(f), LM(g))$.*
2. *The **S-polynomial** of f and g is the combination*
$$S(f,g) = \frac{x^\gamma}{LT(f)} f - \frac{x^\gamma}{LT(g)} g.$$
3. *$\overline{f}^G$ denotes the remainder (see [CLO92]) on division of f by the ordered m-tuple $G = (f_1, \ldots, f_m)$.*

The input E to the algorithm are polynomials $f_1(X_1), \ldots, f_m(X_m)$, the output G the resulting Gröbner basis.

Algorithm 18 Buchberger's Algorithm

```
procedure BUCHBERGER(E)
    $G \leftarrow E$
    repeat
        $G' \leftarrow G$
        for each pair $p, q, p \neq q$ in $G'$ do
            $S \leftarrow \overline{S(p,q)}^{G'}$
            if $S \neq 0$ then
                $G \leftarrow G \cup \{S\}$
            end if
        end for
    until $G = G'$
    return $G$
end procedure
```

4.5 Properties of a Gröbner Basis

Here the key properties of a Gröbner basis are presented which enable us to solve our equation system (1.1). This is at first the definition of an elimination ideal. In order to utilize it the *Elimination Theorem* is stated. It enables us to find an ordering, with respect to the lex order, for eliminating variables from our equation system.

Definition 4.14 *Given $I = \langle f_1, \ldots, f_m \rangle \in k[x_1, \ldots, x_n]$, the kth **elimination ideal** I_k is the ideal of $k[x_{k+1}, \ldots, x_n]$ defined by*

$$I_k = I \cap k[x_{k+1}, \ldots, x_n].$$

Thus I_k consists of all consequences of $f_1, \ldots, f_m$ which eliminate the variables $x_1, \ldots, x_k$.

Theorem 4.15 (Elimination Theorem) *Let $I \subset k[x_1, \ldots, x_n]$ be an ideal and let G be a Gröbner basis of I with respect to lex order where $x_1 > x_2 > \ldots > x_n$. Then for every $0 \leq k \leq n$, the set*

$$G_k = G \cap k[x_{k+1}, \ldots, x_n]$$

is a Gröbner basis of the kth elimination ideal I_k.

Proof 4.16 *See [CLO92]*

4.6 Solving the Equation System

This gives us a method for solving the equation system (1.1). We know from lemma 4.6 that instead of obtaining a solution to our initial equation system we can obtain a solution from our Gröbner basis since both are equivalent. This gives us the advantage to use theorem 4.15. By calculation all solutions to one equation of the Gröbner basis we can extend our solution to all other variables if the Gröbner basis of I was generated with respect to the lex order. That gives an easy and convenient way to solve the whole system of equations.

4.7 Complexity of the Solving Procedure

The complexity of the computation mostly depends on the time which is spend by calculating the Gröbner basis. In case of m Boolean equations in n variables of algebraic degree d one can find that the running time of the Buchberger's Algorithm and its variants for $d = 2$ is $\mathcal{O}(1.7^n)$. If d raises, that is $d \geq 3$ the cost of calculating a Gröbner Basis already exceeds the cost of a brute force attack to obtain the solution, that is 2^n [YCC04].

4.8 Improvements for Calculation a Gröbner basis

There exist different improvements to calculate a Gröbner Basis which are there the XL algorithm [CKPS00] and the F4/F5 algorithms [Fau99, Fau02] algorithms. For the XL algorithm it was shown, that it is a redundant version of the F4 algorithm. The F4 and F5 algorithms use different methods to improve the performance mostly based on matrix operations.

5 Application

In this chapter two ciphers are as examples described. The first is the well known DES [oCoST99] cipher which is in its form as 3-DES still widely used in various kinds of applications. The second one is Trivium [CP05], a minimized stream cipher.

5.1 DES

The DES (Data Encrypt Standard) cipher is a widespread algorithm used in many applications. It is not longer used in its original form since by its structure the key length of single DES with 56 bits is considered to be insecure.

Description of the Cipher DES works as a symmetrical block cipher where the plaintext is substituted and permuted by a feistel scheme in 16 iterations. The block size is 64 bits as well as the key size where the effective key size is only 56 bits, the rest is dedicated to parity checking.

The algorithm consists of 16 identical stages, so called rounds. Moreover the algorithm consists of an initial permutation and a final permutation. The initial permutation is here denoted by IP, the reverse operation by $-IP$. Just as well as the final permutation is denoted by FP and its reverse operation by $-FP$. Before every round one 64 bit block is divided into two 32 bit half blocks and processed alternately by a round function F.

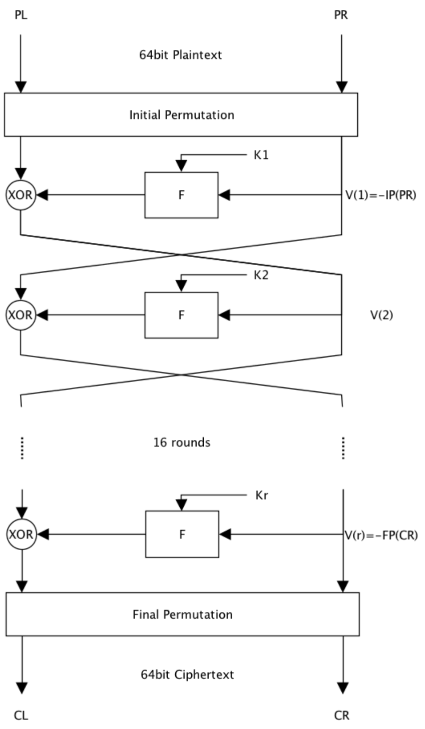

Figure 5.1: DES Structure

For every round 16 different sub keys are determined for the round functions F. From the initial 64 key bits are 56 taken by the initial permutation $P1$, the rest is for parity checking or are discarded. This 58 bits are split up into two blocks of size 28 bits and then by $P2$ are 24 bits from the left and 24 bits from the right half chosen and permuted to assemble a 48 bit sub key. This process continues for every sub key and "<<<" denotes that the input key is rotated specifically for every stage of the key schedule.

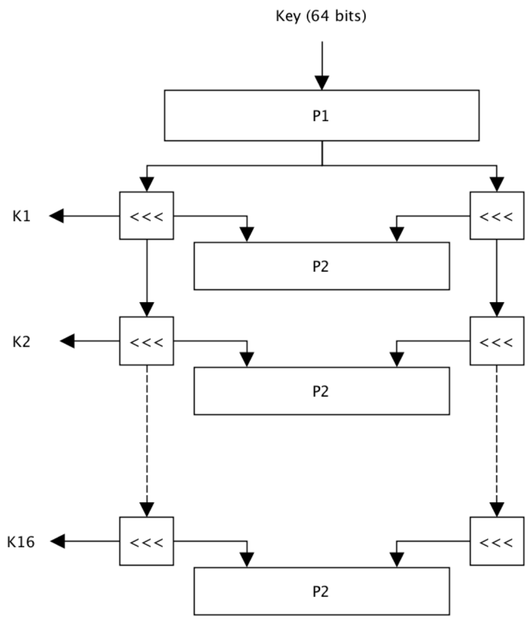

Figure 5.2: DES Key Schedule

Equipped with this sub keys for every round the function F first expands every half 32 bit block to 48 bits by duplicating some bits. The result of this expansion is then processed by a bitwise XOR with the sub key obtained by the key schedule. Afterwards the 48 bit outcome is split up into 6 bit blocks and processed by 8 S-boxes which substitute the input by a defined scheme to 4 bit blocks. The outcome is then again permuted.

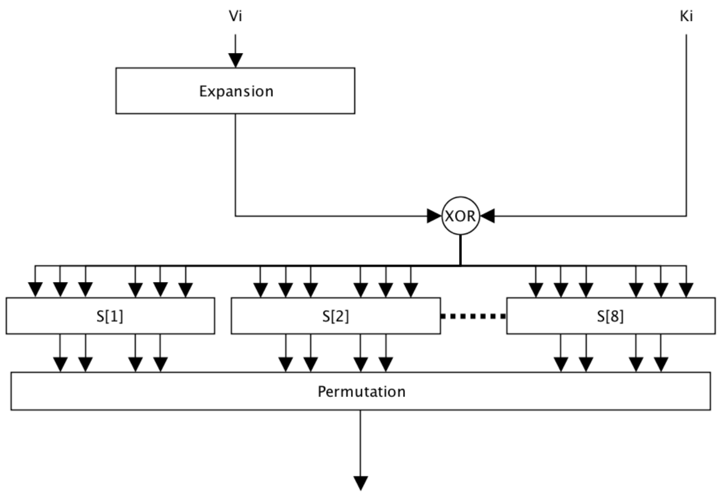

Figure 5.3: DES Round Function

Obtaining an Equation System The generation of an equation system in ANF works as follows as described in [RS06].

Each bit in the ouput of a DES S-box can be expressed as a function of its six inputs and so defines an equation. The four equations coming from the same S-box share all variables input to the S-box, so they can be glued immediately. In doing so one obtains an equation for each S-box in every round. The general form of the equation from the jth S-box in round i is

$$V_j^{(i-1)} \oplus V_j^{(i+1)} = S_j[V_j^{(i)} \oplus K_j^i]$$

where $V_j^{(i-1)}$ and $V_j^{(i+1)}$ are two four-bit strings and $V_j^{(i)}$ and K_j^i are two six-bit strings. K_j^i are the six bits of round key i going into S-box j, determined by the key schedule. The bits in $V_j^{(i-1)}$ and $V_j^{(i+1)}$ are taken from the input to the previous and the next round. Determined by the permutation in the output of the round function, they represent the ouput of S-box j in round i. $V_j^{(i)}$ are the six bits of expanded input to round i going into S-box j.

When the equation comes from an S-box in one of the two first or the two last rounds some of the $V^{(\cdot)}$-values will be constants from the plaintext or the ciphertext. Adding up the number of bits in the general equation we see that no equation contains more than 20 variables. In the second and the second last round the equations contain 16 variables each since $V^{(1)}$ and $V^{(r)}$ comes from the plaintext and ciphertext. In the first and the last round the equations contain only 10 variables each.

The general equation defines a four-bit condition to be satisfied. If an equation contains a variables only 2^{a-4} of the 2^a configurations will satisfy the equation, so the largest configuration lists in the system will contain 2^{16} configurations.

The description above is using only one plaintext/ciphertext pair, but can easily be extended. To build a system using several plaintext/ciphertext pairs, the $V^{(i)}$-variables will have to be diffrent fo reach plaintext/ciphertext pair used, but the key variables remain the same across all equations.

5.2 Trivium

The Trivium stream cipher was developed with the aims on simplicity. The authors say itself that they do not recommend it in a productive environment [CP05]. Nevertheless makes this simplicity the Trivium cipher an interesting study object and should be mentioned as an example how to generate a set of non-linear equations over $\mathbb{F}_2$ in order to break the cipher.

Description of the Cipher Trivium is a hardware oriented stream cipher and was originally an exercise how far a stream cipher could be simplified in order not to sacrifice its security. It is designed to generate up to 2^{64} bits of key stream and consists of 3 NLFSRs, has a 80-bit secret key and a 80 bit initialization vector. The inner state is described by 288 bits.

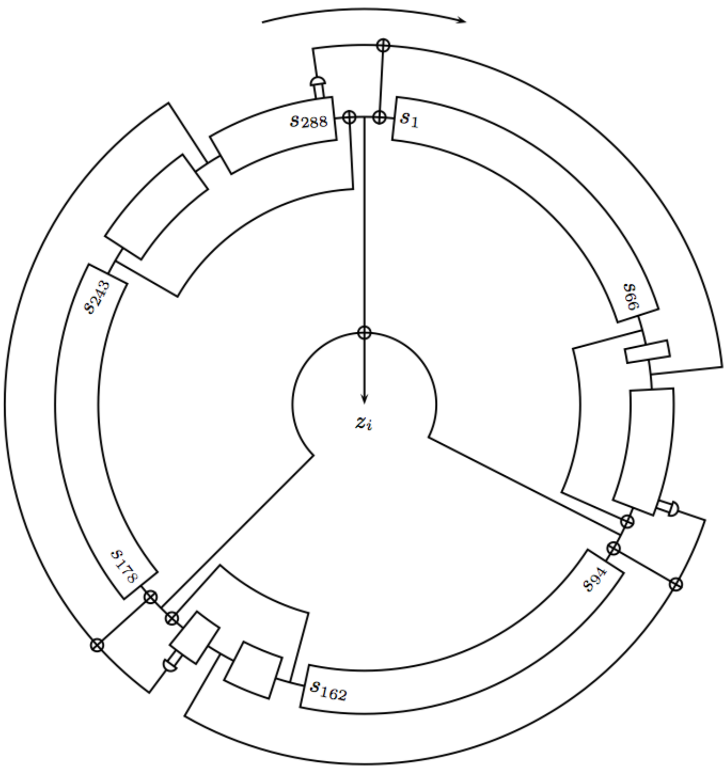

Figure 5.4: Trivium [CP05]

In general a sequence of length $N \leq 2^{64}$ is generated by using iterative 15 specific bits of the internal state to generate the key stream bit and for updating the internal state. The generation of the key stream bit can be expressed algorithmically as follows.

Algorithm 19 Trivium key stream generation

```
1: for $i \leftarrow 1, N$ do
2:     $t_1 \leftarrow s_{66} \oplus s_{93}$
3:     $t_2 \leftarrow s_{162} \oplus s_{177}$
4:     $t_3 \leftarrow s_{243} \oplus s_{288}$
5:     $z_i \leftarrow t_1 \oplus t_2 \oplus t_3$          ▷ Key stream bit
6:     $t_1 \leftarrow t_1 \oplus s_{91}s_{92} \oplus s_{171}$
7:     $t_2 \leftarrow t_2 \oplus s_{175}s_{176} \oplus s_{264}$
8:     $t_3 \leftarrow t_3 \oplus s_{286}s_{287} \oplus s_{69}$
9:     $(s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$
10:    $(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$
11:    $(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$
12: end for
```

This algorithm can be equally expressed as recursive equations

$$
\begin{aligned}
w_n &= y_{n-66} \oplus y_{n-110}y_{n-109} \oplus w_{n-69} \\
x_n &= w_{n-66} \oplus w_{n-93} \oplus w_{n-92}w_{n-91} \oplus x_{n-78} \\
y_n &= x_{n-69} \oplus x_{n-84} \oplus x_{n-83}x_{n-82} \oplus y_{n-87} \\
z_n &= y_{n-66} \oplus y_{n-111} \oplus w_{n-66} \oplus w_{n-93} \oplus x_{n-69} \oplus x_{n-84}
\end{aligned}
$$

where the initial configuration is

$$\begin{aligned}(w_{-1}, w_{-2}, ..., w_{-93}) &\equiv (s_1, s_2, ..., s_{93})\\(x_{-1}, x_{-2}, ..., x_{-84}) &\equiv (s_{94}, s_{95}, ..., s_{177})\\(y_{-1}, y_{-2}, ..., y_{-111}) &\equiv (s_{178}, s_{179}, ..., s_{288})\end{aligned}$$

The key setup of Trivium is described by the following algorithm.

Algorithm 20 Trivium key and IV setup

```
1: $(s_1, s_2, \ldots, s_{93}) \leftarrow (K_1, \ldots, K_{80}, 0, \ldots, 0)$
2: $(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (IV_1, \ldots, IV_{80}, 0, \ldots, 0)$
3: $(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (0, \ldots, 0, 1, 1, 1)$
4: for $i \leftarrow 1, 4 * 288$ do
5:     $t_1 \leftarrow s_{66} \oplus s_{91}s_{92} \oplus s_{93} \oplus s_{171}$
6:     $t_2 \leftarrow s_{162} \oplus s_{175}s_{176} \oplus s_{177} \oplus s_{264}$
7:     $t_3 \leftarrow s_{243} \oplus s_{286}s_{287} \oplus s_{288} \oplus s_{69}$
8:     $(s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$
9:     $(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$
10:    $(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$
11: end for
```

Obtaining an Equation System The first observation from the above definitions is that the key stream is obtained linearly from the state registers, so the key stream is correlated with the state register bits in the following way

$$\begin{aligned}z_i &\equiv t_1 \oplus t_2 \oplus t_3\\&\equiv s_{66} \oplus s_{93} \oplus s_{162} \oplus s_{177} \oplus s_{243} \oplus s_{288}\end{aligned}$$

That implies that the bits $(z_i, z_{i+1}, \ldots, z_{i+65})$ are all linear combinations of the state bits.

If we have 288 variables, namely the initial state register bits $s_1, s_2, \ldots, s_{288}$ and our key stream bits $z_0, z_1, \ldots, z_n$ we can obtain after 66 clocks the following equation system yields only by looking only at the key stream algorithm.

$$\begin{aligned}z_0 &= y_{-66} \oplus y_{-111} \oplus w_{-66} \oplus w_{-93} \oplus x_{-69} \oplus x_{-84}\\z_1 &= y_{-65} \oplus y_{-110} \oplus w_{-65} \oplus w_{-92} \oplus x_{-68} \oplus x_{-83}\\&\vdots\\z_{65} &= y_{-1} \oplus y_{-45} \oplus w_{-1} \oplus w_{-27} \oplus x_{-3} \oplus x_{-18}\end{aligned}$$

which is a linear equation system with 66 equations in 198 variables. But the original content of the state registers is still 288 so we have to obtain more equations. There are in general two ways for the example of Trivium to realize that.

The first way is obvious and given by the recursive equation system of the cipher and would let the degree of equations grow. That means after 66 clock cycles we have our linear equation system. Further 66 clock cycles we have through the new register bits 66 more equations of degree 2. After the next 66 cycles we have new equations of degree 4, then 8 etc.

The second way would be to increase the number of variables by introducing 3 new variables w_n, x_n, y_n for the state update function and their equations as above to compute them.

This approach would yield after k clocks $3k$ equations of degree 2 in $3k$ variables and k linear equations in 288 unknowns.

At this point it should be remarked that there exists another possibility to interpret equation systems which are partial linear, namely as MRHS equations (Multiple Right Hand Side Equations). For further reading about MRHS equations and systems and the Gluing and Agreeing Algorithm one should consult [RS07].

6 Experimental Results

The following experimental results were obtained by using the program "fastglue2" developed during the work on this master thesis. For a detailed program description refer to section 2.9 and for a description of the experimental environment used to Appendix B.

6.1 Gluing Algorithm

The following diagram shows times obtained by solving sample instances with different values for sparsity and different number of variables. In each case $n = m$. Here the Gluing Algorithm in its tree search version was used exclusively without sorting. All values are average values obtained by 10 sample runs on randomly generated instances. The values are measured in seconds.

6.1.1 Pure Gluing Algorithm Unsorted

l	$n = m = 32$	$n = m = 48$	$n = m = 64$	$n = m = 80$	$n = m = 96$
3	0,0040986	0,0585903	35,3578084	535,1290413	7515,71725
4	0,0298945	4,8946569	514,66195	-	-
5	0,4916236	73,297138	8291,282	-	-
6	0,6016082	582,8163838	-	-	-
7	5,8607115	3863,0677	-	-	-
8	70,302769	-	-	-	-

Figure 6.1: Pure Gluing Times (unsorted)

6.1.2 Pure Gluing Algorithm Sorted

l	$n = m = 32$	$n = m = 48$	$n = m = 64$	$n = m = 80$	$n = m = 96$
3	0,002299	0,0034992	0,0096976	0,0195962	0,4248356
4	0,0033988	0,0111979	0,0288949	0,2650592	3,4559803
5	0,0045986	0,1436768	4,6506932	15,135003	253,278232
6	0,0302942	1,029543	32,049681	1450,41318	7661,973333
7	0,0894861	9,879398	232,70408	10701,0725	-
8	0,5925087	55,095221	2367,911	-	-

Figure 6.2: Pure Gluing Times (sorted)

6.1.3 Gluing Tree Depth Histograms

The following graphs show two runs of "fastglue2" on the same instance with $l = 4, m = n = 56$. The abscissa indicates the tree depth values. On the ordinate the number of visits of the algorithm for a specific depth in the tree is plotted. Booth instances use the same input file, but in figure (6.4) the input data is unsorted and processed in that form and in figure (6.3) the data is sorted beforehand.

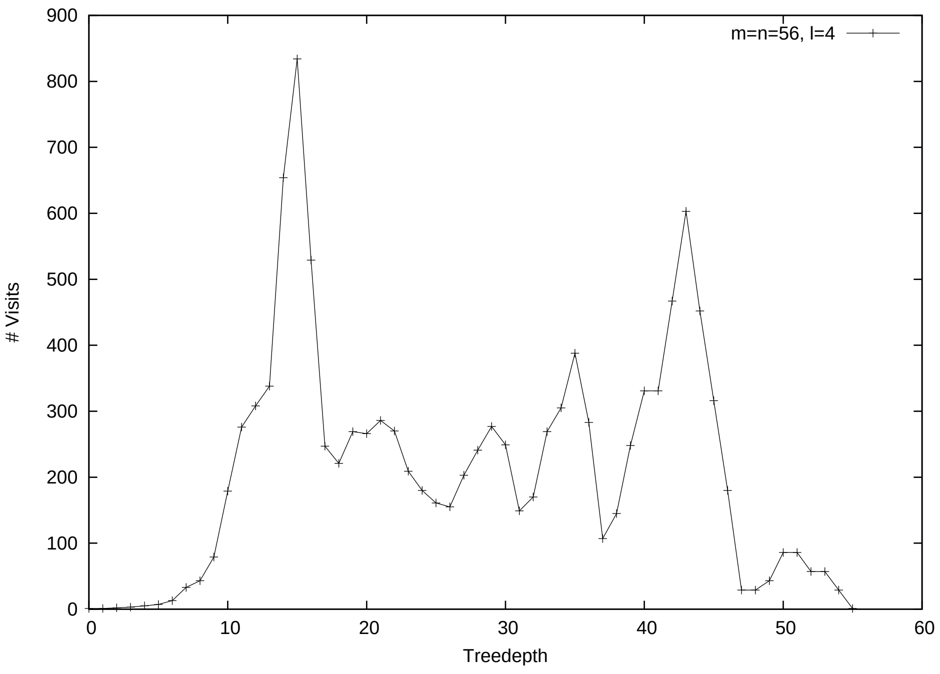

Figure 6.3: Tree depth histogram for an example $m = n = 56, l = 4$ sorted

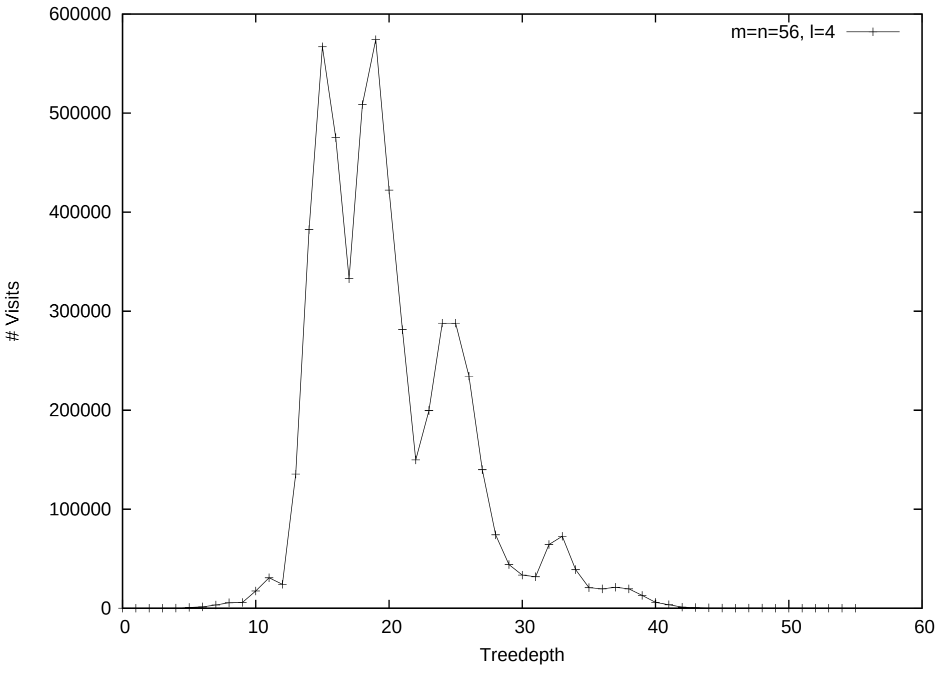

Figure 6.4: Tree depth histogram for an example $m = n = 56, l = 4$ unsorted

6.2 Sorting

The sorting is one point to speed the process of Gluing up. That is because the size of the Gluing or the elements of U_i in $(X_1, V_1) \circ \ldots \circ (X_i, V_i) = (X(i), U_i)$ is $|U_i| = 2^{|X(i)|-i}$ on the average, so the complexity of the Gluing is

$$2^{\max\limits_i |X(i)|-i},$$

with reference to [Sem05].

From the experiments of section 6.1.2 the following values for $|X(i)| - i$ in the average and for the maximum through the ordering are taken. First unsorted and then sorted.

6.2.1 Average $|X(i)| - i$ Unsorted

l	$n = m = 32$	$n = m = 48$	$n = m = 64$	$n = m = 80$	$n = m = 96$
3	6,9	9,6	12,9	15,9	18,7
4	9,2	12,9	17,5	21,2	25,2
5	10,5	15,1	20,4	25,1	30,3
6	11,6	17,1	21,8	27,9	32,67
7	12,4	18,3	24,3	30,25	-
8	13	19,1	24,67	-	-

Figure 6.5: Average $|X(i)| - i$ Unsorted

6.2.2 Maximum $|X(i)| - i$ Unsorted

l	$n = m = 32$	$n = m = 48$	$n = m = 64$	$n = m = 80$	$n = m = 96$
3	12,5	17,3	22,2	27,9	32,3
4	16,1	21,8	29,4	35,5	41,7
5	17,4	25,2	33,6	41,9	50,5
6	20	28,6	36	46,7	52
7	21,3	30,6	40,5	50,5	-
8	22,4	32,3	42,67	-	-

Figure 6.6: Maximum $|X(i)| - i$ Unsorted

6.2.3 Average $|X(i)| - i$ Sorted

l	$n = m = 32$	$n = m = 48$	$n = m = 64$	$n = m = 80$	$n = m = 96$
3	2,4	2,1	3,1	3,2	3,9
4	4,8	6,4	8,1	9,1	11,7
5	6,4	8,7	11,5	13,8	16,3
6	7,9	11,5	14,1	17,5	19,67
7	9	13,2	16,5	19,75	-
8	10,2	14,2	17,78	-	

Figure 6.7: Average $|X(i)| - i$ Sorted

6.2.4 Maximum $|X(i)| - i$ Sorted

l	$n = m = 32$	$n = m = 48$	$n = m = 64$	$n = m = 80$	$n = m = 96$
3	4,9	5,2	7,2	7,8	9
4	7,8	10,4	13,2	14,8	18,7
5	10,2	13,7	17,9	21,4	25,5
6	12,7	18	21,5	27,3	31,33
7	14,4	20,4	25,4	30,5	-
8	15,9	21,9	27,56	-	-

Figure 6.8: Average $|X(i)| - i$ Sorted

6.3 Gluing-Agreeing2 Algorithm

6.3.1 Number of Tuples

The following tables show the number of tuples of the Agreeing2 structure and the initial tuples to agree for different instances of the problem.

l	$n = m = 32$	$n = m = 48$	$n = m = 64$	$n = m = 80$	$n = m = 96$
3	280	396	556	713	808
4	461	718	981	1211	1479
5	776	1210	1531	1996	2290
6	1152	1809	2186	2852	3378
7	1772	2488	3168	4001	4672
8	2673	3533	4326	5174	6403

Figure 6.9: Number of tuples in the Agreeing2 structure

6.3.2 Comment on Magma

While experimenting with magma[Mag08], a program that implements algorithms to produce a Gröbner basis, it became obvious that it can, on the considered input, not compete with minisat and fastglue2 and is therefore in the experiments discarded.

6.3.3 Comparison to Minisat

In this section selected times of the Gluing-Agreeing2 with sorting, as implemented in fastglue2, and minisat[ES03] for different instances of the problem are compared. The minisat parameters are here the following:

- **Restarts** Minisat employs a so called *restart*-mechanism to escape futile parts of the search tree. If a branch exceeds a certain limit of conflicts the search is restarted with possibly different parameters to the branching heuristic [ES03]. This number counts the number of restarts.

- **Decisions** The number of decisions made according to minisats branching heuristic. See section 3.6.

- **Conflicts** Number of conflicts during the solving procedure, see section 3.4

- **Propagations** A propagation occurs if a constraint is found in a watcher list during propagation of unit information, see section 3.4, *Rule for the Elimination of One-Literal Clauses* and section 3.9.

Furthermore are the tree depth histograms for the selected examples of fastglue2 presented.

Example 1

fastglue2	
$m = n$	32
l	8
d	3
Average $\|X(i)\| - i$	11
Maximum $\|X(i)\| - i$	17
Number of tuples	2478
Initial tuples to agree	512
Guesses from Gluing	3635
Time spent Gluing	0,032235s
Time spent Agreeing2	6,77658s
Solve time	6,82778s
minisat	
Restarts	8
Conflicts	4201
Decisions	4784
Propagations	25785
Solve time	0.116548s

Figure 6.10: Values for Example 1

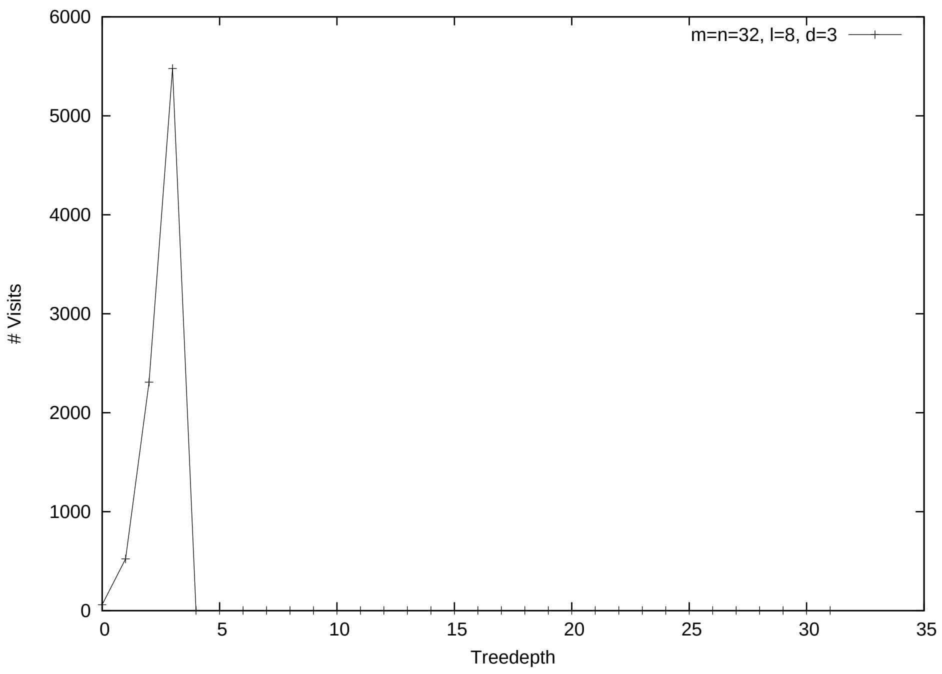

Figure 6.11: Tree depth histogram for Example 1

Example 2

fastglue2	
$m = n$	48
l	7
d	5
Average $\lvert X(i)\rvert - i$	13
Maximum $\lvert X(i)\rvert - i$	21
Number of tuples	2276
Initial tuples to agree	320
Guesses from Gluing	49453
Time spent Gluing	0,385979s
Time spent Agreeing2	63.8821s
Solve time	64.4678s
minisat	
Restarts	8
Conflicts	4627
Decisions	5555
Propagations	40085
Solve time	0.098998s

Figure 6.12: Values for Example 2

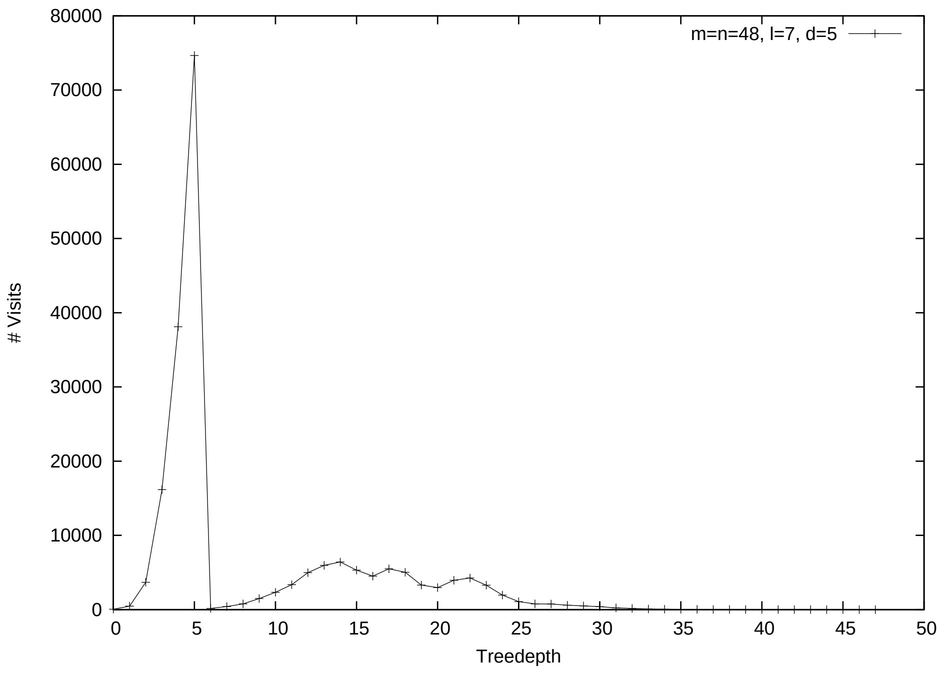

Figure 6.13: Tree depth histogram for Example 2

Example 3

fastglue2	
$m = n$	64
l	7
d	8
Average $\lvert X(i)\rvert - i$	14
Maximum $\lvert X(i)\rvert - i$	20
Number of tuples	2221
Initial tuples to agree	192
Guesses from Gluing	108966
Time spent Gluing	0,469531s
Time spent Agreeing2	76,409s
Solve time	77,2692s
minisat	
Restarts	9
Conflicts	5210
Decisions	6116
Propagations	64483
Solve time	0.100011s

Figure 6.14: Values for Example 3

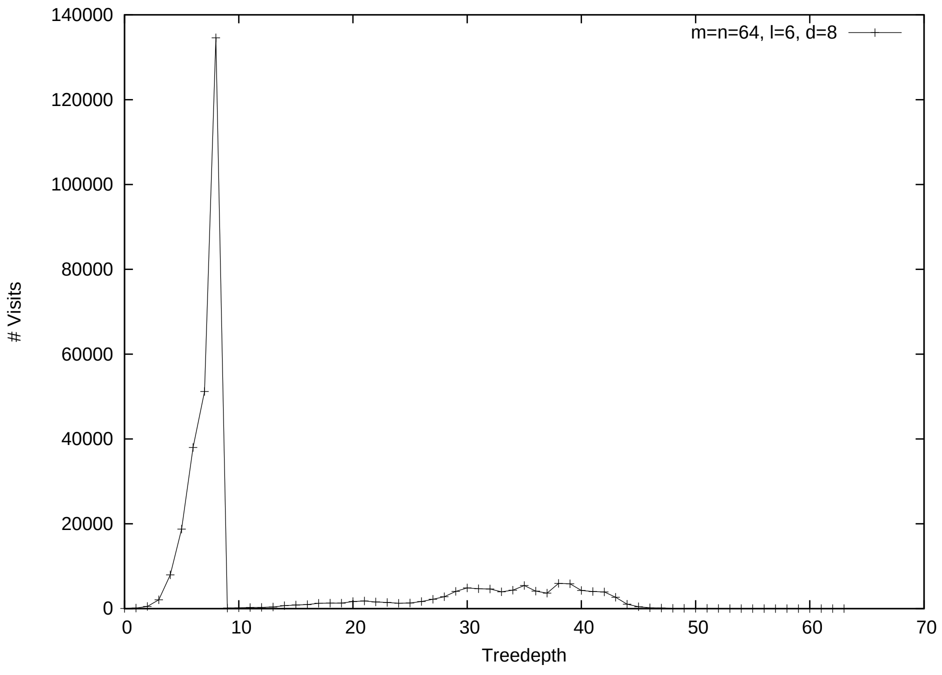

Figure 6.15: Tree depth histogram for Example 3

7 Algorithmic and Implementation Improvements to the Gluing and Agreeing

The following techniques to improve the procedures and to enhance the running time of the algorithms are discussed and developed while creating this masterthesis. They might not be all considered to become a part of the algorithms, but some of them might find their way into the methods if they show a enhancing behavior to the running time. This methods presented might also interfere, which means that they partially may express the same algorithm in another way.

7.1 Edge Removal

This techniques aim is to decrease the running time of the agreeing algorithm. The running time is already improved through the development of the Agreeing2 algorithm from the Agreeing1 algorithm by introducing a more sophisticated propagation of not agreed assignments, but this can still be enhanced. In order to do this we try to reduce the number of tuples and do not from every pair (S_i, S_j) of equations create a new set of tuples for the full agreeing structure, see [Sem08].

For this reason let us see our equation system as a graph $G = (V, E)$, where the set of equations is the set of vertices V and the set of edges is given by

$$E = \{(S_i, S_j) | X(S_i) \cap X(S_j) \neq \emptyset\}$$

That means whenever a given pair of equations has a nonempty intersection in their variables we connect them through an edge. The Agreeing2 procedure can now be seen as an information exchange between the equations S_i and S_j about the agreeing state of their assignments.

As one can easy see is in the Agreeing2 algorithm intended that whenever an edge between two equations exists a set of tuples is generated and introduced to the full agreeing structure. To reduce this number of tuples one can utilize the following lemma.

Lemma 7.1 *Let $G = (V, E)$ be a graph with the equations vertices $V = \{S_1, S_2, \ldots, S_m\}$ and the edges*

$$E = \{(S_i, S_j) | X(S_i) \cap X(S_j) \neq \emptyset\}$$

Then one can remove edges (S_i, S_j) if the following conditions hold:

1. *There exists a path from S_i to S_j, denoted by $p(S_i, S_j)$. That means there exists a set $P = \{S_1, S_2, ..., S_l\}$, where $S_1 = S_i$ and $S_l = S_j$ and for all S_k, S_{k+1} it holds that $(S_k, S_{k+1}) \in E(G)$.*
2. *For the path $p(S_i, S_j)$ and the collection P it holds, that for all $k : X_{i,j} \subseteq X_{k,k+1}$*

and G is still a connected graph after the removal of (S_i, S_j).

Proof 7.2 *Assume that we have two graphs G and G'. G' is the modified graph after the removal of the edges, G before. The outcome of Agreeing1 on G' is different than the outcome of G. W.l.o.g. this implies that in G' an assignment a_j of an equation S_j is still in the agreeing state, which is in G deleted. We know that a_j is in an disagreeing state to some set of variables*

$X_{i,j}$ and some equation S_i. Furthermore we know that there exists a path $p(S_i, S_j)$ with the criterion that $X_{i,j} \subseteq X_{k,k+1}$ for all k along the way. Therefore we have some chain

$$X_{i,j} \subseteq X_{i,k}, X_{k,k+1}, \ldots, X_{k+l-1,k+l}, X_{k+l,j}$$

and by the structure of the Agreeing2 algorithm has to run through pairs of equations (S_x, S_y) until all are in an agreeing state. So S_j would be disagreeing to some S_{k+l} in the subset $X_{k+l,j}$ for which $X_{i,j} \subseteq X_{k+l,j}$ which leads to a contradiction and proves the lemma. □

7.2 Implied Equations

Another enhancement discussed during the further development of the above methods is to add so called *implied equations*. The goal is to insert in a given ordering of equations new ones, which are implied from equations given at a later point in the instance.

Consider an ordering o_1 of the equations and a point for the Agreeing2 algorithm d. This ordering was obtained through sorting equations by a specific pattern and let us assume that this is the best ordering in terms of the size of $X(i)$ we can achieve.

Now it can occur that some $S_i = (X_i, V_i)$ is missing a combination in j variables $Y = \{x_{k_1}, x_{k_2}, \ldots, x_{k_j}\} \subset X_i$ in its assignments. $V_i(Y)$ is therefore not a full binary table. If it holds that the variables $Y \subset X(d)$ we can implicate an equation and insert it before some branch $\leq d$ in order to make the Gluing more precise and to exclude some possible solutions. The method works as follows.

1. After obtaining a sorting of the equations and specifying a senseful d for the point of Agreeing2 calculate the set $X(d) = X(1) \cup \ldots \cup X(d)$.
2. Let $Y = X_i \cap X(d)$ for some symbol $S_i = (X_i, V_i)$. Test if $V_i(Y)$ is full.
3. If $V_i(Y)$ is not full create a new symbol $S_i = (Y, V_i(Y)), i > d$.
4. Insert S_i at some position $< d$ and increment d.

As one can see does this operation not increase $|X(d)|$, but gives the opportunity to obtain information of equations which are behind the point of Agreeing2 and with some probability $|X(S_i) \cap X(d)| > 2$ and we gain information about variables which are implied.

7.3 Parallelization

As one can find different approaches for parallelization of the SAT problem in [SV05] and for a complete survey in [Sin06] the consideration to find a way how the Gluing and the Agreeing approach could gain performance from parallelization techniques is an evident idea. The general structure of the problem instance offers an easy way to find two distinct working partitions which can be processed in parallel. If we consider the equation system (2.1) one easy approach is after the preprocessing to split the first equation into a set of symbols. This could be for example if we want to split $S_1 = (X_1, V_1)$ into k_1 distinct symbols be a resulting set of symbols

$$S_{1,1} = (X_1, V_1^1), S_{1,2} = (X_1, V_1^2), \ldots, S_{1,k_1} = (X_1, V_1^{k_1}) \tag{7.1}$$

where $V_1^i \subset V_1$ and $|V_1^i| \approx \frac{|V_1|}{k_1}$. Equipped with this newly generated equations we can distribute the instance them over k_1 worker nodes in the form of k_1 different instances:

$$\begin{aligned} I_1 &= S_{1,1}, S_2, \ldots, S_m \\ I_2 &= S_{1,2}, S_2, \ldots, S_m \\ &\vdots \\ I_k &= S_{1,k_1}, S_2, \ldots, S_m \end{aligned}$$

The advantage of this approach is, that the nodes need no or only very few communication. The only communication task to do here is to deploy the instances to the worker nodes and to fetch the result. There is no communication which could slow down the process needed. This simple approach reflects the method of fixing a given set of variables. If more parallel resources are available one could either increase k or if not possible split up S_2 into $S_{2,1}, S_{2,2}, \dots, S_{2,k_2}$ and to create $k_1 k_2$ instances.

Another point, this time for shared memory optimization is the Agreeing2 Algorithm. Since this algorithm is bound to one specific equation system and needs to be very fast there is no sense in parallelizing it to multi processes, but it could be senseful to parallelize it in the task to a multithreaded operation. Propagating the information of the empty tuples across the whole Agreeing2 structure is a process which could easily implemented in shared memory parallel mode with OpenMP[Boa08] for example. In using parallel loop techniques the graph traversal of the Agreeing2 could start at different points simultaneously and traverse the it starting from different one-sided empty tuples.

The last point for parallelization is to take the Implicated Equations approach and share found implicated equations with all worker nodes. Only equations obtained by (7.1) may not be distributed since they are modified through the parallelization and therefore to dismiss.

7.4 Watched Assignments

As in the SAT solving techniques it could be applicable to introduce so called watched assignments. The technique should work similar for the Agreeing2 algorithm such that not every tuple is visited if a modification is made there but only if there is one watched assignment changing its state to not agreeing. In comparison to the method from the SAT algorithms (see 3.9) there are no implications to make if there is only one assignment left in an agreed state so there is only need for one watched assignment instead of two. This could improve the running time of the Agreeing2 algorithm.

8 Summary and Conclusions

In this thesis three approaches were presented to solve a system (1.1) with application in crypt-analysis. The first approach are the Gluing Agreeing strategies (short GA strategies) and the second the SAT-solving techniques. In the end a short introduction to Gröbner basis algorithms was given. The GA strategies and the Gröbner basis algorithms are designed to obtain all solutions to an equation system in contrast to the SAT-techniques, which are in general designed to yield only one solution.

During the work on this thesis the main focus was the implementation of the Gluing and Agreeing strategies and to understand how they are related with the problem instance and could be improved to make the solving procedures faster.

While considering the experiments from section 6.3.3 one can conclude that the SAT-solving programs are still the most successful to solve l-sparse equation systems over finite fields, at least in this region of parameters. They are well developed, a wide range of literature as well as a lot of research is done on them. The DPLL algorithm was mentioned first in 1960 and all further work bases on this technique.

Nevertheless, the GA strategies can become a serious competitor for them. If one compares the values for the time spent in the Gluing and the time spent in the Agreeing2 Algorithm, then one sees that the Agreeing2 Algorithm is consuming remarkably more computing time. With the fact that this algorithm has a polynomial complexity in m and n, with respect to a fixed l, it is reasonable to assume that there exist possibilities to reduce its running time tremendously. It should be remarked, that currently no new methods are known which could improve SAT-solving algorithms.

Moreover the Agreeing2 Algorithm is nothing else than a graph representation of information on the equation system. So it is natural to assume that there exist methods to improve the complexity of the information propagation through the graph. For instance, that is the strategy mentioned in chapter 7, namely the Edge Removal. Furthermore it could be found a better memory representation and probably there exist already algorithmical solutions to use.

The approach of implicated equations should result in fewer guesses. So the whole algorithm works faster. If one compares the Gluing times from section 6.3.3 it becomes clear that with a fast verification of the generated guesses the GA strategies are on a competitive basis with the SAT-solving strategies.

In spite of very low complexity expectations the GA family algorithms are still behind SAT-solving programs. The theoretical bounds are valid under assumption that main parameters of the problem may grow. However the exploration of instances with a high number in m, n, l is very difficult. For both, SAT and Gluing/Agreeing techniques one can for large m, n, l only give approximations how they will behave if this values raise, since calculations are very time intensive for sufficiently large instances. To resolve this problem it should be considered to apply the parallelization techniques mentioned in this thesis.

Furthermore should be in the implementational part some changes be considered. While creating the reference implementation "fastglue2" the program was built upon widely used libraries, which may be not perfectly suitable for the application. Implementational primitives should be reconsidered and a machine depend improvement could be applicable.

To summarize the results can it be said that all three solving techniques are not yet ready to take the challenge of solving huge, complex instances (1.1) yielded by a nowadays used cipher on a single workstation. The the SAT-solving techniques and the GA strategies are the most probable candidates to achieve this task. For the GA strategies there exist some reasonable approaches for improvement not yet examined in a practical implementation.

A Program Sources

A.1 Test Instance Generator

A.1.1 InstanceGenerator.py

```python
#!/usr/bin/python
import random
import sys
import math
from copy import deepcopy

class Sorter:
    @staticmethod
    def flatten(x):
        result = []
        for el in x:
            if hasattr(el, "__iter__") and not isinstance(el, basestring):
                result.extend(Sorter.flatten(el))
            else:
                result.append(el)
        return result

    # Calculate max |X(i)|-i for every step
    @staticmethod
    def sizesXiSum(l):
        sizes = []
        for i in range(0, len(l)):
            sizes.append(len(set(Sorter.flatten([s.getVariables() for s in l[:i]])))-i)
        return sizes

    @staticmethod
    def stats(l):
        sizes = Sorter.sizesXiSum(l)
        # Sum up positive sizes
        sizesSum = 0
        nSum = 0
        for s in sizes:
            if s > 0:
                sizesSum += s
                nSum += 1

        print "c Maximum |X(i)|-i: " + str(max(sizes))
        print "c Average |X(i)|-i: " +str(sizesSum/nSum)

    @staticmethod
    def sort(sList,n):
        X = deepcopy(sList)
        sortedList = [X[0]]
        X.remove(sortedList[0])

        while len(X) > 0:
            # Find element which yields smallest |X(i)|-i
            XiMi = n+1
            for Xi in X:
                setLength = len(set(Sorter.flatten([s.getVariables() for s in sortedList] + [Xi.getVariables()])))
                if setLength < XiMi:
                    minXi = Xi
                    XiMi = setLength

            sortedList.append(minXi)
            X.remove(minXi)

        return sortedList

class Symbol:
    def __init__(self, variables, vectors):
        self.__variables = variables
        self.__vectors = vectors

    def getVariables(self):
        return self.__variables

    def getVectors(self):
        return self.__vectors

    def __str__(self):
        result = str(self.__variables) + "; "
        for i in range(0, len(self.__vectors)-1):
            result += str(self.__vectors[i])+": "
        result += str(self.__vectors[len(self.__vectors)-1])
        return result
```

```

class SortedSet(list):
    def __str__(self):
        result = ""
        for i in self:
            result += str(i)+" "
        return result

class Vector(list):
    def __str__(self):
        result = ""
        for i in self:
            result += str(i)+" "
        return result

class InstanceGenerator:
    def __init__(self, numVariables, numSymbols, sparsity, solvable, strict, noFixedVariables):
        self.__sparsity = sparsity
        self.__index = numSymbols
        self.__assignments = []
        self.__variables = [x for x in range(numVariables)]
        self.__generateAssignments()
        self.__solvable = solvable
        self.__strict = strict
        self.__noFixedVariables = noFixedVariables

        if self.__solvable:
            self.__solution = [random.choice([0,1]) for x in range(numVariables)]

    def __iter__(self):
        return self

    def getSolution(self):
        return self.__solution

    def next(self):
        if self.__index == 0:
            raise StopIteration
        self.__index = self.__index-1

        if self.__strict:
            numVars = self.__sparsity
        else:
            numVars = random.randint(3,self.__sparsity)

        vars = SortedSet(random.sample(self.__variables, numVars))
        vars.sort()
        vecs = random.sample(self.__assignments[numVars-1], 2**(numVars-1))

        if self.__solvable:
            trueVector = Vector([self.__solution[v] for v in vars])
            if not trueVector in vecs:
                vecs[random.randint(0,len(vecs)-1)] = trueVector

        if self.__noFixedVariables:
            vecs = self.__eliminateFixedVariables(vecs, vars)

        result = Symbol(vars,vecs)
        return result

    def getFixedAssignment(self):
        vars = SortedSet(random.sample(self.__variables, 1))
        vars.sort()
        vecs = [Vector([self.__solution[v] for v in vars])]
        return Symbol(vars, vecs)

    def getFixedAssignment(self, n):
        vars = SortedSet([n])
        vecs = [self.__solution[n]]
        return Symbol(vars, vecs)

    def __generateAssignments(self):
        bin = lambda n: n>0 and bin(n>>1)+[n&1] or []
        fill = lambda l, s: [0 for x in range(s-len(l))]+l

        for i in range(1, self.__sparsity+1):
            self.__assignments.append([])
            for j in range(2**i):
                self.__assignments[i-1].append(Vector(fill(bin(j),i)))

    def __eliminateFixedVariables(self, vecs, vars):
            leftAssignments = deepcopy(self.__assignments[len(vars)-1])
            for v in vecs:
                leftAssignments.remove(v)

            sum = [0 for i in range(0, len(vars))]
            for v in vecs:
                sum = [sum[i] + v[i] for i in range(0, len(vars))]

            while(0 in sum or len(vars)+1 in sum):
                vecs = random.sample(self.__assignments[len(vars)-1], 2**(len(vars)-1))
                if self.__solvable:
                    trueVector = Vector([self.__solution[v] for v in vars])
                    if not trueVector in vecs:
                        vecs[random.randint(0,len(vecs)-1)] = trueVector

                sum = [0 for i in range(0, len(vars))]
                for v in vecs:
                    sum = [sum[i] + v[i] for i in range(0, len(vars))]
            return vecs

if __name__ == "__main__":
    if len(sys.argv) < 4:
        sys.stderr.write("Usage: " + sys.argv[0] + " <#Variables> <#Symbols> <#Sparsity> [-sort] [-fl] [-nf] [-nsol] [-fixed=<int>]\n")
        sys.exit(1)

    if "-sort" in sys.argv:
        sortFlag = True
    else:
        sortFlag = False

    if "-nf" in sys.argv:
        noFixedVariables = True
    else:
        noFixedVariables = False

    if "-fl" in sys.argv:
        fixedLength = True
    else:
        fixedLength = False

    if "-nsol" in sys.argv:
        solvableFlag = False
    else:
        solvableFlag = True


    numFixedVariables = 0
    for argument in sys.argv:
        if "-fixed=" in argument:
            (arg, num) = argument.split('=')
            numFixedVariables = int(num)

    numVariables = int(sys.argv[1])
    numSymbols = int(sys.argv[2])
    sparsity = int(sys.argv[3])

    if sparsity < 3 and not fixedLength:
        sys.stderr.write("If no fixed length sparsity has to be at least 3\n")
        sys.exit(1)

    g = InstanceGenerator(numVariables, numSymbols, sparsity, solvableFlag, fixedLength, noFixedVariables)
    symbolList = [symbol for symbol in g]

    print "c Number of Variables: "+sys.argv[1]
    print "c Number of Equations: "+sys.argv[2]

    print "c Sparsity: "+sys.argv[3]

    if numFixedVariables:
        print "c Number of fixed Variables: " + str(numFixedVariables)

    if sortFlag:
        sortedList = Sorter.sort(symbolList, numVariables)
        print "c Sorted:"
        Sorter.stats(sortedList)
    else:
        sortedList = symbolList
        print "c Unsorted:"
        Sorter.stats(symbolList)

    if solvableFlag:
        print "c Solution: " + str(g.getSolution())
    else:
        print "c Not neccessarily solvable"

    # If variables are sorted insert at the front variables which are not part of the intersection of symbols from the beginning
    # Otherwise append random fixed variables at the end
    if numFixedVariables:
        if not sortFlag:
            for i in range(0, numFixedVariables):
                sortedList.append(g.getFixedAssignment())
        else:
            i = 0
            varUnion = set([])
            fixedVars = set([])

            while len(fixedVars) < numFixedVariables:
                if i == 0:
                    fixedVars = set(sortedList[0].getVariables())
                else:
                    varUnion = varUnion.union(set(sortedList[i].getVariables()))
                    fixedVars = fixedVars.union(set(sortedList[i+1].getVariables()) - varUnion)
                i = i+1

            while len(fixedVars) > numFixedVariables:
                fixedVars.pop()

            sortedList = [g.getFixedAssignment(i) for i in fixedVars] + sortedList

    # Prefix
    print "p anf " + str(numVariables) + " " + str(numSymbols)
    for symbol in sortedList:
        print symbol

    sys.exit(0)
```

A.1.2 Eq2DimacsCNF.py

```
#!/usr/bin/python
import sys

class ANFtoCNFConverter:
    # Example clause in ANF:
    # [[1,2,3],[[1,0,1],[0,0,1],[0,1,1],[1,1,1]]]
    # to CNF:
    # [[1,2,3],[[0,0,0],[0,1,0],[1,0,0],[1,1,0]]]
    @staticmethod
    def convert(symbol):
        cnfAssignments = ANFtoCNFConverter.__generateAssignments(len(symbol[0]))
        for anfAssignment in symbol[1]:
            cnfAssignments.remove(anfAssignment)
        symbol[1] = cnfAssignments
        return symbol

    @staticmethod
    def __generateAssignments(num_variables):
        bin = lambda n: n>0 and bin(n>>1)+[n&1] or []
        fill = lambda l, s: [0 for x in range(s-len(l))]+l
        assignments = []

        for j in range(2**num_variables):
            assignments.append(fill(bin(j),num_variables))
        return assignments

def recur_map2(fun, data):
    if hasattr(data, "__iter__"):
        return [recur_map2(fun, elem) for elem in data]
    else:
        return fun(data)

if __name__=="__main__":
    f=open(sys.argv[1], 'r')

    dimacsClauses = []
    maxVariable = -1

    for line in f:
        if line[0] == '#' or line[0] == 'c':
            sys.stdout.write(line)
            continue
        if line[0] == 'p':
            (p, cnf, numVariables, numEquations) = line.split(" ")
            continue

        (strVariables, strClauses) = line.split(";")

        variables = strVariables.split(" ")
        variables.remove('')
        variables = map(int, variables)

        clauses = strClauses.split(":")
        clauses = [c.split(" ") for c in clauses]

        for c in clauses:
            for l in c:
                if l == '' or l == '\n':
                    c.remove(l)

        clauses = recur_map2(int, clauses)

        symbol = [variables, clauses]

        cnfSymbol = ANFtoCNFConverter.convert(symbol)

        dimacsBlock = []
        for c in cnfSymbol[1]:
            dimacsClause = ""

            for i in range(0, len(cnfSymbol[0])):
                minus = ''
                if c[i] == 1:
                    minus = '-'
                dimacsClause += minus + str(cnfSymbol[0][i]+1) + " "
            dimacsClause += "0"
            dimacsBlock.append(dimacsClause)
        dimacsClauses += dimacsBlock

    sys.stdout.write("p cnf " + numVariables + " " + str(len(dimacsClauses)) + "\n")
    for c in dimacsClauses:
        sys.stdout.write(c+"\n")
```

A.2 fastglue2

A.2.1 Assignment.h

```
#ifndef ASSIGNMENT_H_
#define ASSIGNMENT_H_
#include <map>
#include <boost/dynamic_bitset.hpp>
#include "Model.h"

typedef boost::dynamic_bitset<unsigned long long int> Projection;
class Equation;
class ProjectionContainer;

/** The equality operator template.
 *
 */
template <class T, class U>
struct assignment_model_equality : public std::binary_function<T, U, bool> {
    bool operator()(const T x, const U y) const {
        //If the assignment isn't agreeing at all, return immediatly false
        if (!(*x).is_agreeing) {
            return false;
        }
        /* The old one had too may calls to constructor of
        dynamic_bitset<unsigned long long int>()
        Therefore:
        a_m = assignment mask
        m_m = model mask
        p = projection
        m = model
        (a_m AND m_m AND p) XOR (a_m AND m_m AND m) here substituted by
        (p XOR m) AND a_m AND m_m
        The old method creates overall at least 3 new objects
        #define MASK ((*(*x).get_model_mask() & *(*y).get_mask()))
                    return (MASK & *(*x).get_model_projection()) == (MASK & (*y));
        #undef MASK
        The new method exactly 1
        boost::dynamic_bitset<unsigned long long int> result(*(*x).get_model_projection());
        result ^= (*y);
        result &= *(*x).get_model_mask();
        result &= *(*y).get_mask();
        return !result.any();
        And even better method without any renewed allocation of memory */
        static boost::dynamic_bitset<unsigned long long int> compare_field;
        compare_field = *(*x).get_model_projection();
        compare_field ^= (*y);
        compare_field &= *(*x).get_mask();
        compare_field &= *(*y).get_mask();
        return !compare_field.any();
    }
};

class Assignment : public boost::dynamic_bitset<unsigned long long int> {
public:
    Assignment() : is_agreeing(true) {};
    Assignment(unsigned int size, bool value) :
            boost::dynamic_bitset<unsigned long long int>(size, value),
            is_agreeing(true) {};
    Assignment(unsigned int size, unsigned long value) :
            boost::dynamic_bitset<unsigned long long int>(size, value),
            is_agreeing(true) {};
    virtual ~Assignment() {};
    /** Set the parent equation */
    inline void set_parent_equation(Equation* e) {
        m_parent_equation = e;
    };
    /** Get the parent equation */
    inline Equation* get_parent_equation(void) {
        return m_parent_equation;
    };
    /** Set the projection to a specified Equation */
    inline void add_equation_projection(Equation* e, Projection p) {
        m_equation_projections.insert(make_pair(e,p));
    };
    /** Fetch a projection to a specific Equation
     * Only used in the preprocessing for the full agreeing so far.
     */
    inline Projection* get_equation_projection(Equation* e) {
        return &m_equation_projections[e];
    }
    /** Set the projection to a model */
    inline void set_model_projection(Projection& p) {
        m_model_projection = p;
    };
    /** Set the mask to the model */
    inline void set_mask(boost::dynamic_bitset<unsigned long long int>& m) {
        m_model_mask = m;
    };
    //Get the mask to the model
    inline boost::dynamic_bitset<unsigned long long int>* get_mask() {
        return &m_model_mask;
    };
    /** Get the projection to a model */
    inline Projection* get_model_projection() {
        return &m_model_projection;
    };
    /** Get the projection containers the assignments is in */
    inline std::vector<ProjectionContainer*>* get_projection_containers() {
        return &m_projection_containers;
    };
    /** The agreeing flag */
    bool is_agreeing;
private:
    /** Projections to specific equations */
    std::map<Equation*, Projection> m_equation_projections;
    /** Projection to a model */
    Projection m_model_projection;
    /** Adress of the Equation which the Assignment belongs to */
    Equation* m_parent_equation;
    /** Mask all variables which are not set in that assignment */
    boost::dynamic_bitset<unsigned long long int> m_model_mask;
    /** Projection containers which the assignment belongs to */
    std::vector<ProjectionContainer*> m_projection_containers;
};

#endif /*ASSIGNMENT_H_*/
```

A.2.2 Model.h

```
#ifndef MODEL_H_
#define MODEL_H_

#include <boost/dynamic_bitset.hpp>

class Model : public boost::dynamic_bitset<unsigned long long int> {
public:
    Model() {};
    ~Model() {};
    inline boost::dynamic_bitset<unsigned long long int>* get_mask() {
        return &m_mask;
    };
private:
    boost::dynamic_bitset<unsigned long long int> m_mask;
};

#endif /*MODEL_H_*/
```

A.2.3 Main.cpp

```
#define NDEBUG

#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <boost/foreach.hpp>
#include <boost/tokenizer.hpp>
#include <getopt.h>
#include <stdio.h>
#include "Stats.h"
#include "Equation.h"
#include "Assignment.h"
#include "Model.h"
#include "Branch.h"
#include "Tree.h"
#include "Solver.h"
#include "Sorter.h"

#define __FreeBSD__ 1

using std::cerr;
using std::cout;
using std::endl;

Tree Solver::m_tree;
unsigned int Solver::m_num_variables;
unsigned int Solver::m_num_equations;
unsigned int Solver::m_average_sparsity;
int Solver::m_fa_position;
FullAgreeingStructure Solver::m_fa_structure;

static void usage(char* s_exec);
static inline void read_from_file(std::ifstream &file, EquationVector& e, int& num_equations, int& num_variables);
static inline double cpu_time(void);

int main(int argc, char** argv) {
        /** Program flags options */
        int sort_mode = 0;
        int verbose_mode = 0;
        int fa_position = -1;
        std::ifstream infile;
        std::fstream histo_out;
        std::fstream stats_out;
        std::fstream solution_out;

        static struct option longopts[] = {
                        {"i", required_argument, NULL,  'i'},
                        {"s",  no_argument,  &sort_mode,  1},
                        {"v", no_argument,  &verbose_mode, 1},
                        {"d", required_argument, NULL,  'd'},
                        {"hout", required_argument, NULL,  'h'},
                        {"sout", required_argument, NULL,  'o'},
                        {"sol", required_argument, NULL,  'l'},
                        {NULL,  0,    NULL,  0}
        };

        char ch;
        while((ch = getopt_long_only(argc, argv, "svi:d:h:o:l:", longopts, NULL)) != -1){
                switch(ch){
                case 'i':
                        infile.open(optarg, std::fstream::in);
                        if(!infile.good()){
                                cerr<<"Error opening in file!"<<endl;
                                exit(1);
                        }
                        break;
                case 'd':
                        fa_position = atoi(optarg);
                        break;
                case 'h':
                        histo_out.open(optarg, std::fstream::app | std::fstream::out);
                        if(!histo_out.good()){
                                cerr<<"Error opening histogram file!"<<endl;
                                exit(1);
                        }
                        break;
                case 'o':
                        stats_out.open(optarg, std::fstream::app | std::fstream::out);
                        if(!stats_out.good()){
                                cerr<<"Error opening statistic file!"<<endl;
                                exit(1);
                        }
                        break;
                case 'l':
                        solution_out.open(optarg, std::fstream::app | std::fstream::out);
                        if(!solution_out.good()){
                                cerr<<"Error opening solution file!"<<endl;
                                exit(1);
                        }
                        break;
                case 0:
                        break;
                default:
                        usage(argv[0]);
                        exit(1);
                        break;
                }
        }

        int num_variables = 0;
    int num_equations = 0;
    EquationVector equation_vector;

    /** Timing */
    double start_time = 0;
    double parse_time = 0;
    double sorting_time = 0;
    double preparation_time = 0;
    double solve_time = 0;

    start_time = cpu_time();
    /** Read equation system from file */
    read_from_file(infile, equation_vector, num_variables, num_equations);
    parse_time = cpu_time() - start_time;

    if(verbose_mode) cout<<"Parse time: "<<parse_time<<endl;

    int unsorted_avg_xi;
    int unsorted_max_xi;
    Sorter::stats(&equation_vector, unsorted_avg_xi, unsorted_max_xi);
    /** Sort if applicable */
    if(sort_mode){
        Sorter::sort(&equation_vector, num_variables);
    }
    int sorted_avg_xi;
    int sorted_max_xi;
    Sorter::stats(&equation_vector, sorted_avg_xi, sorted_max_xi);
    sorting_time = cpu_time() - parse_time;

    if(verbose_mode){
        if(sort_mode){
                cout<<"Sorting time: "<<sorting_time<<" "<<", |X(i)|-i unsorted avg: "<<unsorted_avg_xi<<" max: "<<unsorted_max_xi
                       <<", sorted avg: "<<sorted_avg_xi<<" max: "<<sorted_max_xi<<endl;
        }else{
                cout<<"|X(i)|-i unsorted avg: "<<unsorted_avg_xi<<" max: "<<unsorted_max_xi<<endl;
        }
    }

    Stats.sorting_time = sorting_time;
    Stats.unsorted_avg_xi = unsorted_avg_xi;
    Stats.unsorted_max_xi = unsorted_max_xi;
    Stats.sorted_avg_xi = sorted_avg_xi;
    Stats.sorted_max_xi = sorted_max_xi;

    /** Set the position for the full agreeing */
    Solver::set_fa_position(fa_position);
    Solver::prepare(num_variables, num_equations, &equation_vector);
    preparation_time = cpu_time() - sorting_time;
    if(verbose_mode){
        cout<<"Perparation time: "<<preparation_time<<endl;
        cout<<"Number of tuples: "<<Stats.tuples<<endl;
        cout<<"Initial assignments to agree: "<<Stats.initial_assignments<<endl;
    }
    Stats.preparation_time = preparation_time;

    Model *res = Solver::solve();
    solve_time = cpu_time() - preparation_time;
    Stats.solve_time = solve_time;
    if(verbose_mode){
        cout<<"Guesses produced by gluing: "<<Stats.guesses_produced<<endl;
        cout<<"Time gluing: "<<Stats.time_gluing<<endl;
        cout<<"Time agreeing2: "<<Stats.time_agreeing2<<endl;
        cout<<"Solve time: "<<solve_time<<endl;
        cout<<"Overall time: "<<cpu_time()<<endl;
    }

    Stats.overall_time = cpu_time();

    if(histo_out.good()) Stats.print_histo(histo_out);
    if(stats_out.good()) Stats.print_stats(stats_out);
    if(solution_out.good()) solution_out<<*res<<endl;
    return EXIT_SUCCESS;
}

void usage(char* s_exec) {
    cerr<<"Usage: "<<s_exec<<" <equation file>"<<endl;
}

void read_from_file(std::ifstream &file, EquationVector& e, int& num_variables, int& num_equations) {
    std::string s_line;
    typedef boost::tokenizer<boost::char_separator<char> > tokenizer;
    boost::char_separator<char> eq_sep(";");
    boost::char_separator<char> var_sep(" ");
    boost::char_separator<char> ass_sep(":");
    boost::char_separator<char> x_sep(" ");

    while (getline(file, s_line)) {
        if (s_line[0] == 'c') {
            //Comment line
            continue;
        } else if (s_line[0] == 'p') {
            //Parameter line
            tokenizer p_tok(s_line);
            tokenizer::iterator p_tok_it = p_tok.begin();
            ++p_tok_it;
            ++p_tok_it;
            num_variables = atoi((*p_tok_it).c_str());
            ++p_tok_it;
            num_equations = atoi((*p_tok_it).c_str());
        } else {
            //Equation
            tokenizer eq_tok(s_line, eq_sep);
            std::vector<std::string> equation;
            BOOST_FOREACH(std::string s, eq_tok) {
                equation.push_back(s);
            }

            //Variables
            std::vector<std::string> variables;
            tokenizer var_tok(equation[0], var_sep);
            BOOST_FOREACH(std::string s, var_tok) {
                variables.push_back(s);
            }

            //Assignments
            std::vector<std::string> assignments;
            tokenizer ass_tok(equation[1], ass_sep);
            BOOST_FOREACH(std::string s, ass_tok) {
                assignments.push_back(s);
            }

            //Treat Variables
            std::vector<unsigned int> vars;
            BOOST_FOREACH(std::string str_variable, variables) {
                vars.push_back(atoi(str_variable.c_str()));
            }

            //Treat Assignments
            std::vector<Assignment*> ass;
            BOOST_FOREACH(std::string str_assignment, assignments) {
                tokenizer x_tok(str_assignment, x_sep);
                Assignment* a = new Assignment();
                BOOST_FOREACH(std::string str_var, x_tok) {
                    a->push_back(atoi(str_var.c_str()) == 1 ? true : false);
                }
                ass.push_back(a);
            }

            //Create new equation
            Equation *eq = new Equation(vars, ass);
            e.push_back(eq);
        }
    }
}
```

A.2.4 Tree.h

```
#ifndef TREE_H_
#define TREE_H_

#include <vector>
#include "Branch.h"

class Tree : public std::vector<Branch*> {
public:
    Tree() : pos(0) {};
    inline bool has_next(void) {
        return pos == this->size()-1 ? false : true;
    };
    inline void forward(void) {
        ++pos;
    };
    inline Branch* current(void) {
        return this->at(pos);
    };
    inline Branch* next(void) {
        return this->at(pos+1);
    };
    inline Branch* last(void) {
        return this->at(this->size()-1);
    };
    inline void back(void);
    inline void repeat_from(unsigned int d);
    unsigned int pos;
};

void Tree::back(void) {
    //Reset iterators
    this->at(pos)->reset_iterators();
    this->at(pos)->get_model()->reset();
    //And go one step back
    --pos;
}

void Tree::repeat_from(unsigned int d) {
    while (pos > d) {
        back();
    }

    --(*this->at(pos)->get_assignments_current());
}

#endif /*TREE_H_*/
```

A.2.5 Branch.h

```
#ifndef BRANCH_H_
#define BRANCH_H_

#include <vector>
#include <boost/dynamic_bitset.hpp>
#include "Equation.h"

class Branch {
public:
    Branch(int num_variables, Equation* eq, std::vector<unsigned int>& variables) :
            m_variables(variables),
            m_equation(eq),
            m_current_assignment(NULL),
            m_end_assignments(NULL) {
        m_model.resize(num_variables, false);
        m_equation->set_parent_branch(this);
    };
    virtual ~Branch() {};
    inline std::vector<unsigned int>* get_variables() {
        return &m_variables;
    };
    inline Model* get_model() {
        return &m_model;
    };
    inline Equation* get_equation() {
        return m_equation;
    }
    inline AssignmentsToModelIterator* get_assignments_current();
    inline AssignmentsToModelIterator* get_assignments_end() {
        return m_end_assignments;
    };
    inline void reset_iterators(void) {
        delete m_current_assignment;
        delete m_end_assignments;
        m_current_assignment = NULL;
        m_end_assignments = NULL;
    };
private:
    //Model of the current Branch
    Model m_model;
    //The variables involved to this branch
    std::vector<unsigned int> m_variables;
    //The current equation
    Equation* m_equation;
    AssignmentsToModelIterator* m_current_assignment;
    AssignmentsToModelIterator* m_end_assignments;
};

AssignmentsToModelIterator* Branch::get_assignments_current() {
    if (m_current_assignment == NULL) {
        m_current_assignment =
            new boost::filter_iterator<std::binder2nd<assignment_model_equality<Assignment*, Model*> >, std::vector<Assignment*>::iterator>
            (std::bind2nd(assignment_model_equality<Assignment*, Model*>(), &m_model),
             m_equation->get_assignments()->begin(),
             m_equation->get_assignments()->end());
        m_end_assignments =
            new boost::filter_iterator<std::binder2nd<assignment_model_equality<Assignment*, Model*> >, std::vector<Assignment*>::iterator>
            (std::bind2nd(assignment_model_equality<Assignment*, Model*>(), &m_model),
             m_equation->get_assignments()->end(),
             m_equation->get_assignments()->end());

    }
    return m_current_assignment;
}

#endif /*BRANCH_H_*/
```

A.2.6 Equation.h

```
#ifndef EQUATION_H_
#define EQUATION_H_

#include <vector>
#include <iostream>
#include <boost/iterator/filter_iterator.hpp>
#include "Assignment.h"

class Branch;

typedef boost::filter_iterator<std::binder2nd<assignment_model_equality<Assignment*, Model*> >, std::vector<Assignment*>::iterator>
      AssignmentsToModelIterator;

class Equation {
public:
    Equation(std::vector<unsigned int> variables, std::vector<Assignment*> assignments) :
            m_variables(variables),
            m_assignments(assignments) {
        BOOST_FOREACH(Assignment* a, m_assignments) {
            a->set_parent_equation(this);
        }
        num_agreeing_assignments = assignments.size();
    };
    Equation() {};
    ~Equation() {};
    inline void set_parent_branch(Branch* b) {
        m_parent_branch = b;
    };
    inline Branch* get_parent_branch(Branch* b) {
        return m_parent_branch;
    };
    inline std::vector<unsigned int>* get_variables() {
        return &m_variables;
    };
    inline std::vector<Assignment*>* get_assignments() {
        return &m_assignments;
    };
    friend std::ostream& operator<<(std::ostream& out, const Equation &e);
    unsigned int num_agreeing_assignments;
private:
    std::vector<unsigned int> m_variables;
    std::vector<Assignment*> m_assignments;
    Branch* m_parent_branch;
};

std::ostream& operator<<(std::ostream& out, const Equation &e) {
    out<<"{[";
    std::copy(e.m_variables.begin(), e.m_variables.end(), std::ostream_iterator<int>(out, " "));
    out<<"],";
    for (std::vector<Assignment*>::const_iterator it = e.m_assignments.begin(); it != e.m_assignments.end(); ++it) {
        if ((*it)->is_agreeing) {
            out<<"[";
            out<<*(*it);
            out<<"]";
        }
    }
    out<<"}||="<<e.num_agreeing_assignments;
    return out;
}

#endif /*EQUATION_H_*/
```

A.2.7 FullAgreeingStructure.h

```
#ifndef FULLAGREEINGGRAPH_H_
#define FULLAGREEINGGRAPH_H_

#include <string>
#include <vector>
#include <map>
#include <iostream>
#include <set>
#include <queue>
#include <boost/foreach.hpp>
#include <boost/dynamic_bitset.hpp>
#include "Assignment.h"
#include "Equation.h"
#include "Model.h"
#include "Branch.h"

using std::cerr;
using std::cout;
using std::endl;
using std::make_pair;

#define PRINTTUPLES \
        cerr<<endl; \
        BOOST_FOREACH(ProjectionTuple *p, m_tuples){ \
                cerr<<"|{"; \
                BOOST_FOREACH(Assignment *a, *p->first){ \
                        if(a->is_agreeing()) cerr<<*a->get_model_projection()<<","; \
                } \
                cerr<<"}|="<<p->first->num_agreeing_assignments<<", |{"; \
                BOOST_FOREACH(Assignment *a, *p->second){ \
                        if(a->is_agreeing()) cerr<<*a->get_model_projection()<<","; \
                } \
                cerr<<"}|="<<p->second->num_agreeing_assignments<<endl; \
        } \
        cerr<<endl;

struct ProjectionContainer;
//The Projections tuple
struct ProjectionTuple : public std::pair<ProjectionContainer*, ProjectionContainer*>{

};

//Container for the projections
struct ProjectionContainer : public std::set<Assignment*> {
        ProjectionTuple *parent;
        unsigned int num_agreeing_assignments;
    };

class FullAgreeingStructure {
public:
    FullAgreeingStructure() {};
    virtual ~FullAgreeingStructure() {};
    inline void init(std::vector<unsigned int> * initial_variables, std::vector<Equation*> *eqv);
    inline void add_equation_pair(Equation* e1, Equation* e2);
    inline bool run_agreeing2(Model *m);
    inline void undo();
    inline unsigned int get_num_tuples(){return m_tuples.size();}
    inline unsigned int get_num_initial_assignments(){return m_initial_agreeing_assignments.size();}
private:
    //Tuples of equal projections
    std::vector<ProjectionTuple*> m_tuples;
    //For the depth first search
    std::vector<ProjectionTuple*> m_processing_queue;
    //Assignments for the initial agreeing in the beginning
    std::set<Assignment*> m_initial_agreeing_assignments;
    //All assignments to undo after a run
    std::vector<Assignment*> m_undo_assignments;
    //Keep track about the empty tuples, allocated beforehand
    std::vector<ProjectionTuple*> m_empty_tuples;
};

bool FullAgreeingStructure::run_agreeing2(Model *m) {
        //cerr<<endl;
    for (std::set<Assignment*>::iterator as_it = m_initial_agreeing_assignments.begin();
         as_it != m_initial_agreeing_assignments.end();
         ++as_it) {
            static boost::dynamic_bitset<unsigned long long int> compare_field;
            compare_field = *(*as_it)->get_model_projection();
            compare_field ^= *m;
            compare_field &= *m->get_mask();
            compare_field &= *(*as_it)->get_mask();
            if (compare_field.any()) {
                //cerr<<**as_it<<endl;
                (*as_it)->is_agreeing = false;

                --(*as_it)->get_parent_equation()->num_agreeing_assignments;
                for (std::vector<ProjectionContainer*>::iterator pc_it = (*as_it)->get_projection_containers()->begin();
                         pc_it != (*as_it)->get_projection_containers()->end();
                         ++pc_it) {
                    --(*pc_it)->num_agreeing_assignments;

                    //Check if one side got empty, if yes put it to the processing queue, if booth sides got empty add it to empty tuples
                    if ((*pc_it)->parent->first->num_agreeing_assignments == 0 ^
                            (*pc_it)->parent->second->num_agreeing_assignments == 0) {
                        m_processing_queue.push_back((*pc_it)->parent);
                    }
                    if (((*pc_it)->parent->first->num_agreeing_assignments == 0) &&
                            ((*pc_it)->parent->second->num_agreeing_assignments == 0)) {
                        m_empty_tuples.push_back((*pc_it)->parent);
                    }
                }

                m_undo_assignments.push_back((*as_it));

                if ((*as_it)->get_parent_equation()->num_agreeing_assignments == 0) {
                    return false;
                }
            }
        }

    //Treat all one sided empty tuples
    while (m_processing_queue.size() > 0) {
        ProjectionTuple *t = m_processing_queue.back();
        m_processing_queue.pop_back();

        //Decide which side to treat (side which is not empty)
        ProjectionContainer *p = t->first->num_agreeing_assignments != 0 ? t->first : t->second;

        //Set every assignment of a one sided empty tuple to not agreeing (if not already done)
        for (std::set<Assignment*>::iterator as_it = p->begin(); as_it != p->end(); ++as_it) {
                if ((*as_it)->is_agreeing) {
                    (*as_it)->is_agreeing = false;

                    --(*as_it)->get_parent_equation()->num_agreeing_assignments;
                    for (std::vector<ProjectionContainer*>::iterator pc_it = (*as_it)->get_projection_containers()->begin();
                         pc_it != (*as_it)->get_projection_containers()->end();
                         ++pc_it) {
                        --(*pc_it)->num_agreeing_assignments;

                        //Check if the projection container got one sided empty, if yes insert into the processing queue
                        if ((*pc_it) == p) continue; //Continue if it is the current projection contianer
                        if ((*pc_it)->parent->first->num_agreeing_assignments == 0 ^
                                (*pc_it)->parent->second->num_agreeing_assignments == 0) {
                            m_processing_queue.push_back((*pc_it)->parent);
                        }
                    }

                    m_undo_assignments.push_back((*as_it));

                    if ((*as_it)->get_parent_equation()->num_agreeing_assignments == 0) {
                        return false;
                    }
                }
            }

        //The tuple is now on both sides empty, insert it into the empty_tules set
        m_empty_tuples.push_back(t);
    }

    return m_empty_tuples.size() == m_tuples.size() ? false : true;
}

inline void FullAgreeingStructure::undo() {
    //Reset assignments
    for (std::vector<Assignment*>::iterator as_it = m_undo_assignments.begin();
         as_it != m_undo_assignments.end();
         ++as_it) {
        (*as_it)->is_agreeing = true;
        ++(*as_it)->get_parent_equation()->num_agreeing_assignments;
        for (std::vector<ProjectionContainer*>::iterator pc_it = (*as_it)->get_projection_containers()->begin();
                 pc_it != (*as_it)->get_projection_containers()->end();
                 ++pc_it) {
            ++(*pc_it)->num_agreeing_assignments;
        }
    }
    m_undo_assignments.clear();
    m_empty_tuples.clear();
    m_processing_queue.clear();
}

//Initialize the graph structure, neccessary to set for example m_variables_occurences and m_assignment_occurences also
//on which variables the initial agreeing will operate
void FullAgreeingStructure::init(std::vector<unsigned int> *initial_variables, std::vector<Equation*> *eqv) {
    //Create a copy of initial_variables to be able to modify it
    std::set<unsigned int> work_initial_variables(initial_variables->begin(), initial_variables->end());

    //In order to introduce the whole guess fetch as long equations to introduce until we have no variables left in "initial_variables"
    while (work_initial_variables.size() > 0) {
        unsigned int max_intersection_size = 0;
        std::set<unsigned int> max_intersection;
        Equation *max_intersection_equation = 0;
        BOOST_FOREACH(Equation* eq, *eqv) {
            std::set<unsigned int> intersection;
            std::insert_iterator<std::set<unsigned int> > ins_intersection(intersection, intersection.begin());
            set_intersection(work_initial_variables.begin(), work_initial_variables.end(), eq->get_variables()->begin(), eq->get_variables()
                ->end(), ins_intersection);

            if (intersection.size() > max_intersection_size) {
                max_intersection_equation = eq;
                max_intersection_size = intersection.size();
                max_intersection = intersection;
            }
        }
        BOOST_FOREACH(Assignment* as, *max_intersection_equation->get_assignments()) {
            m_initial_agreeing_assignments.insert(as);
        }

        BOOST_FOREACH(unsigned int var, max_intersection) {
            work_initial_variables.erase(var);
        }
    }
}

//Dublicates are already avoided already in Solver.h
void FullAgreeingStructure::add_equation_pair(Equation* e1, Equation* e2) {
    //Get pairwise equal projections from e1 to e2 from e1
    std::multimap<unsigned long int, Assignment*> e1_value_assignment_table;
    std::multimap<unsigned long int, Assignment*> e2_value_assignment_table;
    std::set<unsigned long int> e1_value_set;
    std::set<unsigned long int> e2_value_set;

    //Get common projection values
    BOOST_FOREACH(Assignment* as, *e1->get_assignments()) {
        e1_value_assignment_table.insert(make_pair(as->get_equation_projection(e2)->to_ulong(), as));
        e1_value_set.insert(as->get_equation_projection(e2)->to_ulong());
    }
    BOOST_FOREACH(Assignment* as, *e2->get_assignments()) {
        e2_value_assignment_table.insert(make_pair(as->get_equation_projection(e1)->to_ulong(), as));
        e2_value_set.insert(as->get_equation_projection(e1)->to_ulong());
    }

    std::vector<unsigned long int> common_projection_values;
    std::insert_iterator<std::vector<unsigned long int> > ins_common_projections(common_projection_values,
            common_projection_values.begin());
    set_intersection(e1_value_set.begin(), e1_value_set.end(),
                     e2_value_set.begin(), e2_value_set.end(),
                     ins_common_projections);

    //For every common projection value create a Tuple of the assignments
    //During creation of m_variable_occurences keep track of the uniqueness of m_variable_occurences pairs
    BOOST_FOREACH(unsigned long int projection_value, common_projection_values) {
        ProjectionTuple *tuple = new ProjectionTuple();
        ProjectionContainer *tuple_left = new ProjectionContainer();
        ProjectionContainer *tuple_right = new ProjectionContainer();
        //Set the parent tuple for the container and initialize the number of assignments
        tuple_left->parent = tuple;
        tuple_right->parent = tuple;

        //Treat the left side (eq1)
        std::multimap<unsigned long int, Assignment*>::iterator left_it = e1_value_assignment_table.find(projection_value);
        std::multimap<unsigned long int, Assignment*>::iterator left_end = e1_value_assignment_table.upper_bound(projection_value);
        while (left_it != left_end) {
            tuple_left->insert(left_it->second);
            left_it->second->get_projection_containers()->push_back(tuple_left);
            ++left_it;
        }

        //Treat the right side (eq2)
        std::multimap<unsigned long int, Assignment*>::iterator right_it = e2_value_assignment_table.find(projection_value);
        std::multimap<unsigned long int, Assignment*>::iterator right_end = e2_value_assignment_table.upper_bound(projection_value);
        while (right_it != right_end) {
            tuple_right->insert(right_it->second);
            right_it->second->get_projection_containers()->push_back(tuple_right);
            ++right_it;
        }

        //Set the number of assignments in the ProjectionTuples
        tuple_left->num_agreeing_assignments = tuple_left->size();
        tuple_right->num_agreeing_assignments = tuple_right->size();

        tuple->first = tuple_left;
        tuple->second = tuple_right;
        m_tuples.push_back(tuple);
    }
}

#endif /*FULLAGREEINGGRAPH_H_*/
```

A.2.8 Sorter.h

```
#ifndef SORTER_H_
#define SORTER_H_

#include <set>

class Sorter {
public:
    inline static void sort(EquationVector *equation_vector, int num_variables);
    inline static void stats(EquationVector *equation_vector, int &avg_xi, int &max_xi);
};

/* Archieve minimal |X(i)|−i to given number of equations */
void Sorter::sort(EquationVector *equation_vector, int num_variables) {
    EquationVector *result_eqv = new EquationVector();
    std::set<unsigned int> xi;

    //Start with the first equation
    result_eqv−>push_back(equation_vector−>at(0));
    equation_vector−>erase(equation_vector−>begin());
    std::insert_iterator<std::set<unsigned int> > ins_xi(xi, xi.begin());
    std::set_union(xi.begin(), xi.end(), result_eqv−>at(0)−>get_variables()−>begin(), result_eqv−>at(0)−>get_variables()−>end(), ins_xi);


    //Try every equation in order to find the smallest xi
    while (equation_vector−>size() != 0) {
        Equation* minimal_growth_equation;
        int new_xi = num_variables+1;

        BOOST_FOREACH(Equation* eq, *equation_vector) {
            std::set<unsigned int> current_xi;
            std::insert_iterator<std::set<unsigned int> > ins_current_xi(current_xi, current_xi.begin());
            std::set_union(xi.begin(), xi.end(), eq−>get_variables()−>begin(), eq−>get_variables()−>end(), ins_current_xi);

            if ((int)current_xi.size() < new_xi) {
                new_xi = current_xi.size();
                minimal_growth_equation = eq;
            }
        }

        std::insert_iterator<std::set<unsigned int> > ins_xi(xi, xi.begin());
        std::set_union(xi.begin(), xi.end(), minimal_growth_equation−>get_variables()−>begin(), minimal_growth_equation−>get_variables()−>
            end(), ins_xi);

        result_eqv−>push_back(minimal_growth_equation);
        equation_vector−>erase(find(equation_vector−>begin(), equation_vector−>end(), minimal_growth_equation));
    }

    equation_vector−>clear();
    BOOST_FOREACH(Equation *eq, *result_eqv) {
        equation_vector−>push_back(eq);
    }

    delete result_eqv;
}

void Sorter::stats(EquationVector *equation_vector, int &avg_xi, int &max_xi) {
    avg_xi = 0;
    max_xi = −1;

    int i = 0;

    std::set<unsigned int> xi;
    BOOST_FOREACH(Equation *eq, *equation_vector) {
        std::insert_iterator<std::set<unsigned int> > ins_xi(xi, xi.begin());
        std::set_union(xi.begin(), xi.end(), eq−>get_variables()−>begin(), eq−>get_variables()−>end(), ins_xi);

        //Sum up only positive grow values
        avg_xi += xi.size() − i > 0 ? xi.size() − i : 0;
        if ((int)xi.size() − i > max_xi) {
            max_xi = xi.size() − i;
        }
        ++i;
    }
    avg_xi /= i;
}
#endif /*SORTER_H_*/
```

A.2.9 Solver.h

```
#ifndef SOLVER_H_
#define SOLVER_H_

#include <boost/foreach.hpp>
#include <iostream>
#include <vector>
#include "Equation.h"
#include "Assignment.h"
#include "Tree.h"
#include "Branch.h"
```

```
#include "Model.h"
#include "Assignment.h"
#include "FullAgreeingStructure.h"


typedef std::vector<Equation*> EquationVector;

/** Main class which coordinates the solving.
 * This class is responsible for the solving of the given instance. At first the function
 * prepare() should be called with suitable values for num_variables, num_equations and an
 * EquationVector to initialize the FullAgreeingStructure and to start the before hand
 * calculations of set intersections and other task which can be done before hand. After
 * that the function solve() can be executed, which returns a pointer to the resulting
 * Model if a solution is found. If the routine does not find any solution it exits with
 * an error.
 */
class Solver {
public:
    /** An empty constructor.
     * This constructor can be empty, since all member functions and member variables
     * are static and at no time a instance of the class is generated.
     */
    Solver();

    /** Function to prepare the equation system and perform before hand calculations.
     * Different preparations are done in that procedure. At first ...
     * @param num_variables Number of variables in the equation system.
     * @param num_equations Number of equations in the equation system.
     * @param eqv A pointer to the vector of Equation pointers.
     * @return A pointer to the resulting model.
     * @see Equation
     * @see Model
     */
    inline static void prepare(int num_variables, int num_equations, EquationVector* eqv);

    /** The Main solving routine.
     *
     */
    inline static Model* solve();

    /** Set the desired value for the agreeing2 algorithm */
    inline static void set_fa_position(int d){
        m_fa_position = d;
    }

    /** Fetch the treedepth for the full agreeing */
    inline static int get_fa_position(){
        return m_fa_position;
    }
    /** An empty destructor.
     * This destructor can be empty, since all member functions and member variables
     * are static and at no time a instance of the class is generated.
     */
    virtual ~Solver();
private:

    /** The tree structure for the Gluing Algorithm.
     * This static variable holds the tree structure for the Gluing Algorithm. It will be
     * generated through the prepare function and will not be changed during the whole
     * computation.
     * @see Tree
     * @see prepare
     */
    static Tree m_tree;

    /** The Full Agreeing structure.
     * Holds the Full Agreeing structure generated by the prepare function. The general
     * structure will not be altered during the whole computation.
     * @see FullAgreeingStructure
     * @see prepare
     */
    static FullAgreeingStructure m_fa_structure;

    /** Number of variables */
    static unsigned int m_num_variables;

    /** Number of equations */
    static unsigned int m_num_equations;

    /** The average sparsity */
    static unsigned int m_average_sparsity;

    /** Integer value at which point the Full Agreeing procedure should be applied */
    static int m_fa_position;
};

Model* Solver::solve() {
        Stats.timer_gluing_start();
    while (m_tree.has_next()) {
        //cerr<<m_tree.pos<<" ";
        ++Stats.depth_histo[m_tree.pos];
        //In case we are at the point of full agreeing undo previous changes
        if ((int)m_tree.pos == m_fa_position) {
                Stats.timer_gluing_stop();
                Stats.timer_agreeing2_start();
            m_fa_structure.undo();
            Stats.timer_agreeing2_stop();
            Stats.timer_gluing_start();
        }

        Model* current_model = m_tree.current()->get_model();
        Model* next_model = m_tree.next()->get_model();

        /* Fetch iterators to the assignments which are fitting to the current model from
         * the last point read */
        AssignmentsToModelIterator *current_assignment =
                m_tree.current()->get_assignments_current();
        AssignmentsToModelIterator *end_assignments =
                m_tree.current()->get_assignments_end();

        if (*current_assignment != *end_assignments) {
            //Copy the old model (without the mask)
            *next_model |= *current_model;
            //Apply the current assignment to the next model
            *next_model |= *(*(*current_assignment))->get_model_projection();
            //Increment filter iterator
            ++(*current_assignment);
            //The full agreeing
            if ((int)m_tree.pos == m_fa_position) {
                ++Stats.guesses_produced;

                Stats.timer_gluing_stop();
                Stats.timer_agreeing2_start();
                bool agreeing2_result = m_fa_structure.run_agreeing2(next_model);
                Stats.timer_agreeing2_stop();
                Stats.timer_gluing_start();

                if (!agreeing2_result) {
                    //Try the next guess
                    next_model->reset();
                    continue;
                }
            }
            m_tree.forward();
        } else {
            m_tree.back();
        }
    }
    return m_tree.last()->get_model();
}

void Solver::prepare(int num_variables, int num_equations, EquationVector* eqv) {
    //Since we have only model
    BOOST_FOREACH(Equation* e, *eqv) {
        BOOST_FOREACH(Assignment* a, *e->get_assignments()) {
            Projection model_p(num_variables);
            boost::dynamic_bitset<unsigned long long int> model_m(num_variables);
            for (std::vector<unsigned int>::iterator v_it = e->get_variables()->begin();
                    v_it != e->get_variables()->end(); ++v_it) {
                //Get the position/index of the variable in the vector by
                //subtracting begin() from the current vector iterator (address)
                model_p[*v_it] = (*a)[v_it - e->get_variables()->begin()];
                model_m[*v_it] = true;
            }
            a->set_model_projection(model_p);
            a->set_mask(model_m);
        }
    }

    //Calculate projections to other Equations
    //First find intersections
    std::vector<std::vector<Equation*> > equations_to_variables(num_variables);
    BOOST_FOREACH(Equation* e, *eqv) {
        BOOST_FOREACH(int var, *e->get_variables()) {
            equations_to_variables[var].push_back(e);
        }
    }

    //Second calculate for every variable pair (e1,e2) where e1 != e2
    //their projections and add that equation pair to
    //the full agreeing graph, iff there is a variable interseciton
    typedef std::pair<Equation*, Equation*> EquationPair;
    std::set<EquationPair> pairs_treated;

    BOOST_FOREACH(std::vector<Equation*> eq_var, equations_to_variables) {
        BOOST_FOREACH(Equation* eq1, eq_var) {
            BOOST_FOREACH(Equation* eq2, eq_var) {
                //Keep track of duplicates and unneccessary information
                if (eq1 == eq2) continue;
                EquationPair eq_pair1(eq1, eq2);
                EquationPair eq_pair2(eq2, eq1);
                if (pairs_treated.find(eq_pair1) == pairs_treated.end() &&
                        pairs_treated.find(eq_pair2) == pairs_treated.end()) {

                    //Calculate set intersection
                    std::vector<unsigned int> var_intersection;
                    std::insert_iterator<std::vector<unsigned int> >
                    var_intersection_inserter(var_intersection, var_intersection.begin());
                    set_intersection(eq1->get_variables()->begin(),
                                     eq1->get_variables()->end(),
                                     eq2->get_variables()->begin(),
                                     eq2->get_variables()->end(),
                                     var_intersection_inserter);

                    /* If we have a variable interseciton calculate the indices and append
                     * equation projections */
                    if (var_intersection.size() >= 1) {
                        //Calculate indices
                        std::vector<unsigned int> eq1_indices;
                        std::vector<unsigned int> eq2_indices;
                        BOOST_FOREACH(unsigned int var, var_intersection) {
                            eq1_indices.push_back(find(eq1->get_variables()->begin(),
                                                       eq1->get_variables()->end(), var) -
                                                  eq1->get_variables()->begin());
                            eq2_indices.push_back(find(eq2->get_variables()->begin(),
                                                       eq2->get_variables()->end(), var) -
                                                  eq2->get_variables()->begin());
                        }

                        //For each assignment calculate projection and append in equation 1
                        BOOST_FOREACH(Assignment* a, *eq1->get_assignments()) {
                            Projection p;
                            BOOST_FOREACH(unsigned int i, eq1_indices) {
                                p.push_back((*a)[i]);
                            }
                            a->add_equation_projection(eq2, p);
                        }

                        //For each assignment calculate projection and append in equation 2
                        BOOST_FOREACH(Assignment* a, *eq2->get_assignments()) {
                            Projection p;
                            BOOST_FOREACH(unsigned int i, eq2_indices) {
                                p.push_back((*a)[i]);
                            }
                            a->add_equation_projection(eq1, p);
                        }

                        //Insert into the full agreeing graph
                        m_fa_structure.add_equation_pair(eq1, eq2);
                    }
                }
                //Insert to avoid duplicates
                pairs_treated.insert(eq_pair1);
                pairs_treated.insert(eq_pair2);
            }
        }
    }

    //Create the search tree and sum up variables/equation to get average sparsity
    std::vector<unsigned int> variables_so_far;
    unsigned int sum_variables_per_equation = 0;
    BOOST_FOREACH(Equation* eq, *eqv) {
        Branch* branch = new Branch(num_variables, eq, variables_so_far);
        branch->get_model()->resize(num_variables, false);
        branch->get_model()->get_mask()->resize(num_variables, false);
        //Mask variables which are not yet used.
        BOOST_FOREACH(unsigned int var, variables_so_far) {
            (*branch->get_model()->get_mask())[var] = true;
        }

        //Push branch back to tree
        m_tree.push_back(branch);

        //Fill variables_so_far to keep track of which variables are already involed and to create the model mask
        std::vector<unsigned int> new_variables_so_far;
        std::insert_iterator<std::vector<unsigned int> >
        ins_new_variables_so_far(new_variables_so_far, new_variables_so_far.begin());
        set_union(variables_so_far.begin(),
                  variables_so_far.end(),
                  eq->get_variables()->begin(),
                  eq->get_variables()->end(),
                  ins_new_variables_so_far);
        variables_so_far = new_variables_so_far;
        sum_variables_per_equation += eq->get_variables()->size();
    }

    //At the end of the tree insert a new blank branch to store the result in
    Branch *b = new Branch(num_variables, new Equation(), variables_so_far);
    b->get_model()->get_mask()->resize(num_variables, true);
    m_tree.push_back(b);

    //Calculate the average sparsity of the equations and store the number of variables
    m_average_sparsity = sum_variables_per_equation / eqv->size();
    m_num_equations = eqv->size();
    m_num_variables = num_variables;

    //Initialize full agreeing
    if(m_fa_position >= 0){
        m_fa_structure.init(m_tree[m_fa_position+1]->get_variables(), eqv);
        Stats.initial_assignments = m_fa_structure.get_num_initial_assignments();
        Stats.tuples = m_fa_structure.get_num_tuples();
    }

    //Initialize Statistics
    Stats.init(eqv->size());
    Stats.n = num_variables;
    Stats.m = eqv->size();
    Stats.l = m_average_sparsity;
    Stats.d = m_fa_position;
}

```

```
#endif /*SOLVER_H_*/
```

A.2.10 Stats.h

```
#ifndef SOLVER_H_
#define SOLVER_H_

#include <boost/foreach.hpp>
#include <iostream>
#include <vector>
#include "Equation.h"
#include "Assignment.h"
#include "Tree.h"
#include "Branch.h"
#include "Model.h"
#include "Assignment.h"
#include "FullAgreeingStructure.h"


typedef std::vector<Equation*> EquationVector;

/** Main class which coordinates the solving.
 * This class is responsible for the solving of the given instance. At first the function
 * prepare() should be called with suitable values for num_variables, num_equations and an
 * EquationVector to initialize the FullAgreeingStructure and to start the before hand
 * calculations of set intersections and other task which can be done before hand. After
 * that the function solve() can be executed, which returns a pointer to the resulting
 * Model if a solution is found. If the routine does not find any solution it exits with
 * an error.
 */
class Solver {
public:
    /** An empty constructor.
     * This constructor can be empty, since all member functions and member variables
     * are static and at no time a instance of the class is generated.
     */
    Solver();

    /** Function to prepare the equation system and perform before hand calculations.
     * Different preparations are done in that procedure. At first ...
     * @param num_variables Number of variables in the equation system.
     * @param num_equations Number of equations in the equation system.
     * @param eqv A pointer to the vector of Equation pointers.
     * @return A pointer to the resulting model.
     * @see Equation
     * @see Model
     */
    inline static void prepare(int num_variables, int num_equations, EquationVector* eqv);

    /** The Main solving routine.
     *
     */
    inline static Model* solve();

    /** Set the desired value for the agreeing2 algorithm */
    inline static void set_fa_position(int d){
        m_fa_position = d;
    }

    /** Fetch the treedepth for the full agreeing */
    inline static int get_fa_position(){
        return m_fa_position;
    }
    /** An empty destructor.
     * This destructor can be empty, since all member functions and member variables
     * are static and at no time a instance of the class is generated.
     */
    virtual ~Solver();
private:

    /** The tree structure for the Gluing Algorithm.
     * This static variable holds the tree structure for the Gluing Algorithm. It will be
     * generated through the prepare function and will not be changed during the whole
     * computation.
     * @see Tree
     * @see prepare
     */
    static Tree m_tree;

    /** The Full Agreeing structure.
     * Holds the Full Agreeing structure generated by the prepare function. The general
     * structure will not be altered during the whole computation.
     * @see FullAgreeingStructure
     * @see prepare
     */
    static FullAgreeingStructure m_fa_structure;

    /** Number of variables */
    static unsigned int m_num_variables;

    /** Number of equations */
    static unsigned int m_num_equations;

```

```cpp
    /** The average sparsity */
    static unsigned int m_average_sparsity;

    /** Integer value at which point the Full Agreeing procedure should be applied */
    static int m_fa_position;
};

Model* Solver::solve() {
        Stats.timer_gluing_start();
    while (m_tree.has_next()) {
        //cerr<<m_tree.pos<<" ";
        ++Stats.depth_histo[m_tree.pos];
        //In case we are at the point of full agreeing undo previous changes
        if ((int)m_tree.pos == m_fa_position) {
                Stats.timer_gluing_stop();
                Stats.timer_agreeing2_start();
            m_fa_structure.undo();
            Stats.timer_agreeing2_stop();
            Stats.timer_gluing_start();
        }

        Model* current_model = m_tree.current()->get_model();
        Model* next_model = m_tree.next()->get_model();

        /* Fetch iterators to the assignments which are fitting to the current model from
         * the last point read */
        AssignmentsToModelIterator *current_assignment =
                m_tree.current()->get_assignments_current();
        AssignmentsToModelIterator *end_assignments =
                m_tree.current()->get_assignments_end();

        if (*current_assignment != *end_assignments) {
            //Copy the old model (without the mask)
            *next_model |= *current_model;
            //Apply the current assignment to the next model
            *next_model |= *(*(*current_assignment))->get_model_projection();
            //Increment filter iterator
            ++(*current_assignment);
            //The full agreeing
            if ((int)m_tree.pos == m_fa_position) {
                ++Stats.guesses_produced;

                Stats.timer_gluing_stop();
                Stats.timer_agreeing2_start();
                bool agreeing2_result = m_fa_structure.run_agreeing2(next_model);
                Stats.timer_agreeing2_stop();
                Stats.timer_gluing_start();

                if (!agreeing2_result) {
                    //Try the next guess
                    next_model->reset();
                    continue;
                }
            }
            m_tree.forward();
        } else {
            m_tree.back();
        }
    }
    return m_tree.last()->get_model();
}

void Solver::prepare(int num_variables, int num_equations, EquationVector* eqv) {
    //Since we have only model
    BOOST_FOREACH(Equation* e, *eqv) {
        BOOST_FOREACH(Assignment* a, *e->get_assignments()) {
            Projection model_p(num_variables);
            boost::dynamic_bitset<unsigned long long int> model_m(num_variables);
            for (std::vector<unsigned int>::iterator v_it = e->get_variables()->begin();
                    v_it != e->get_variables()->end(); ++v_it) {
                //Get the position/index of the variable in the vector by
                //subtracting begin() from the current vector iterator (address)
                model_p[*v_it] = (*a)[v_it - e->get_variables()->begin()];
                model_m[*v_it] = true;
            }
            a->set_model_projection(model_p);
            a->set_mask(model_m);
        }
    }

    //Calculate projections to other Equations
    //First find intersections
    std::vector<std::vector<Equation*> > equations_to_variables(num_variables);
    BOOST_FOREACH(Equation* e, *eqv) {
        BOOST_FOREACH(int var, *e->get_variables()) {
            equations_to_variables[var].push_back(e);
        }
    }

    //Second calculate for every variable pair (e1,e2) where e1 != e2
    //their projections and add that equation pair to
    //the full agreeing graph, iff there is a variable interseciton
    typedef std::pair<Equation*, Equation*> EquationPair;
    std::set<EquationPair> pairs_treated;

    BOOST_FOREACH(std::vector<Equation*> eq_var, equations_to_variables) {
        BOOST_FOREACH(Equation* eq1, eq_var) {
            BOOST_FOREACH(Equation* eq2, eq_var) {
                //Keep track of duplicates and unneccessary information
                if (eq1 == eq2) continue;
                EquationPair eq_pair1(eq1, eq2);
                EquationPair eq_pair2(eq2, eq1);
                if (pairs_treated.find(eq_pair1) == pairs_treated.end() &&
                        pairs_treated.find(eq_pair2) == pairs_treated.end()) {

                    //Calculate set intersection
                    std::vector<unsigned int> var_intersection;
                    std::insert_iterator<std::vector<unsigned int> >
                    var_intersection_inserter(var_intersection, var_intersection.begin());
                    set_intersection(eq1->get_variables()->begin(),
                                     eq1->get_variables()->end(),
                                     eq2->get_variables()->begin(),
                                     eq2->get_variables()->end(),
                                     var_intersection_inserter);

                    /* If we have a variable interseciton calculate the indices and append
                     * equation projections */
                    if (var_intersection.size() >= 1) {
                        //Calculate indices
                        std::vector<unsigned int> eq1_indices;
                        std::vector<unsigned int> eq2_indices;
                        BOOST_FOREACH(unsigned int var, var_intersection) {
                            eq1_indices.push_back(find(eq1->get_variables()->begin(),
                                                       eq1->get_variables()->end(), var) -
                                                  eq1->get_variables()->begin());
                            eq2_indices.push_back(find(eq2->get_variables()->begin(),
                                                       eq2->get_variables()->end(), var) -
                                                  eq2->get_variables()->begin());
                        }

                        //For each assignment calculate projection and append in equation 1
                        BOOST_FOREACH(Assignment* a, *eq1->get_assignments()) {
                            Projection p;
                            BOOST_FOREACH(unsigned int i, eq1_indices) {
                                p.push_back((*a)[i]);
                            }
                            a->add_equation_projection(eq2, p);
                        }

                        //For each assignment calculate projection and append in equation 2
                        BOOST_FOREACH(Assignment* a, *eq2->get_assignments()) {
                            Projection p;
                            BOOST_FOREACH(unsigned int i, eq2_indices) {
                                p.push_back((*a)[i]);
                            }
                            a->add_equation_projection(eq1, p);
                        }

                        //Insert into the full agreeing graph
                        m_fa_structure.add_equation_pair(eq1, eq2);
                    }
                }
                //Insert to avoid duplicates
                pairs_treated.insert(eq_pair1);
                pairs_treated.insert(eq_pair2);
            }
        }
    }

    //Create the search tree and sum up variables/equation to get average sparsity
    std::vector<unsigned int> variables_so_far;
    unsigned int sum_variables_per_equation = 0;
    BOOST_FOREACH(Equation* eq, *eqv) {
        Branch* branch = new Branch(num_variables, eq, variables_so_far);
        branch->get_model()->resize(num_variables, false);
        branch->get_model()->get_mask()->resize(num_variables, false);
        //Mask variables which are not yet used.
        BOOST_FOREACH(unsigned int var, variables_so_far) {
            (*branch->get_model()->get_mask())[var] = true;
        }

        //Push branch back to tree
        m_tree.push_back(branch);

        //Fill variables_so_far to keep track of which variables are already involed and to create the model mask
        std::vector<unsigned int> new_variables_so_far;
        std::insert_iterator<std::vector<unsigned int> >
        ins_new_variables_so_far(new_variables_so_far, new_variables_so_far.begin());
        set_union(variables_so_far.begin(),
                  variables_so_far.end(),
                  eq->get_variables()->begin(),
                  eq->get_variables()->end(),
                  ins_new_variables_so_far);
        variables_so_far = new_variables_so_far;
        sum_variables_per_equation += eq->get_variables()->size();
    }

    //At the end of the tree insert a new blank branch to store the result in
    Branch *b = new Branch(num_variables, new Equation(), variables_so_far);
    b->get_model()->get_mask()->resize(num_variables, true);
    m_tree.push_back(b);

    //Calculate the average sparsity of the equations and store the number of variables
    m_average_sparsity = sum_variables_per_equation / eqv->size();
    m_num_equations = eqv->size();
    m_num_variables = num_variables;

    //Initialize full agreeing
    if(m_fa_position >= 0){
        m_fa_structure.init(m_tree[m_fa_position+1]->get_variables(), eqv);
        Stats.initial_assignments = m_fa_structure.get_num_initial_assignments();
        Stats.tuples = m_fa_structure.get_num_tuples();
    }

    //Initialize Statistics
    Stats.init(eqv->size());
    Stats.n = num_variables;
    Stats.m = eqv->size();
    Stats.l = m_average_sparsity;
    Stats.d = m_fa_position;
}

#endif /*SOLVER_H_*/
```

B Experimental Environment

B.1 Generating Random Instances

In order to run the experiments there is a need to generate several sample random instances. The "InstanceGenerator" itself is a python program as one can see in A.1.1. The program takes default 3 parameters, namely the number of variables, the number of symbols and the sparsity. Moreover it has the following optional parameters:

Parameter	Description
-nf	No fixed variables
-sort	Sort the instance in terms of $X(i)$
-fl	Fixed length
-fixed=n	Fix n variables
-nsol	Do not guarantee solvability

Figure B.1: InstanceGenerator.py Parameters

If the "-nf" flag is set it is assured that the instance contains no fixed variables in one symbol. That means all vectors of the instance are summed up (integer) and it is checked if the resulting vector contains an element which is either 0 or 2^{l-1}. If that happened a new set of assignments is chosen at random. If the sort flag is set the instance get sorted and the "-fl" flag guarantees that all equations contain l variables. With the "-fixed" flag one can specify the number of fixed variables in the instance and if the "-nsol" flag is set it is not guaranteed that the instance is solvable, means its outcome is undetermined.

Sample Random Instance As a sample for the output consider the command

```
InstanceGenerator.py 10 10 3 -fixed=2 -nf -sort -fl
```

in which we want to generate 10 symbols in 10 variables where 2 variables are fixed and the rest not and the size of X_i is fixed to 3. Furthermore the equation system should be sorted. This would result in the following output:

```
c Number of Variables: 10
c Number of Equations: 10
c Sparsity: 3
c Number of fixed Variables: 2
c Sorted:
c Maximum |X(i)|-i: 2
c Average |X(i)|-i: 1
c Solution: [1, 1, 0, 0, 0, 0, 1, 1, 1, 0]
p anf 10 10
3 ; 0
5 ; 0
1 3 5 ; 1 0 1 : 1 0 0 : 0 0 0 : 1 1 1
3 5 6 ; 0 0 0 : 1 0 0 : 0 1 0 : 0 0 1
```

```
0 5 6 ; 0 0 1 : 1 1 1 : 1 0 1 : 0 0 0
2 3 5 ; 1 1 0 : 0 0 0 : 1 0 1 : 0 1 1
0 1 8 ; 1 0 0 : 1 1 1 : 0 0 0 : 0 0 1
3 7 8 ; 1 1 0 : 0 1 1 : 1 1 1 : 0 0 1
1 6 7 ; 0 1 0 : 1 0 1 : 1 1 1 : 0 1 1
0 2 7 ; 0 1 0 : 1 0 1 : 1 1 0 : 0 0 1
7 8 9 ; 0 1 0 : 1 0 1 : 0 0 0 : 1 1 0
0 4 8 ; 1 0 1 : 0 0 1 : 1 1 0 : 0 1 0
```

which is further referred to as ANF form.

B.2 Converting to SAT

The process of converting an instance in the given format above is described in 3.2. The tool used for this procedure is called ”Eq2DimacsCNF.py” and presented in A.1.2. The process of conversion works exactly as described, except that for tests with fixed variables there is a modification in the process since a fixed variable in the ANF form would result in $2^{l-1} - 1$ clauses. This was considered as an ”unfair” disadvantage for sat solvers and therefore avoided. Instead there are inserted single clauses as like a symbol with only one vector.

The output of the above example instance would result in the following file in CNF form:

```
c Number of Variables: 10
c Number of Equations: 10
c Sparsity: 3
c Number of fixed Variables: 2
c Sorted:
c Maximum |X(i)|-i: 2
c Average |X(i)|-i: 1
c Solution: [1, 1, 0, 0, 0, 0, 1, 1, 1, 0]
p cnf 10 42
-4 0
-6 0
2 4 -6 0
2 -4 6 0
2 -4 -6 0
-2 -4 6 0
4 -6 -7 0
-4 6 -7 0
-4 -6 7 0
-4 -6 -7 0
1 -6 7 0
1 -6 -7 0
-1 6 7 0
-1 -6 7 0
3 4 -6 0
3 -4 6 0
-3 4 6 0
-3 -4 -6 0
1 -2 9 0
1 -2 -9 0
-1 2 -9 0
```

```
-1 -2 9 0
4 8 9 0
4 -8 9 0
-4 8 9 0
-4 8 -9 0
2 7 8 0
2 7 -8 0
-2 7 8 0
-2 -7 8 0
1 3 8 0
1 -3 -8 0
-1 3 8 0
-1 -3 -8 0
8 9 -10 0
8 -9 -10 0
-8 9 10 0
-8 -9 -10 0
1 5 9 0
1 -5 -9 0
-1 5 9 0
-1 -5 -9 0
```

B.3 Compiler

As compiler to translate the programs it was used the Intel compiler [DKK+99] in the version 10.0.

B.4 OS & CPU

While running the experiments it was made use of the IBM e1350 cluster of the Parallab which is dedicated to run sequential jobs. The key stats are the following:

- 86 e326 nodes
- 172 AMD/opteron 250 (2.4 Ghz) processors (2 cpus per node)
- 258 Gigabyte memory (on average 3 Gigabyte per node)
- 6880 Gigabyte disk (80 Gigabyte per node)
- Linux operating system (Redhat)
- Gigabit Ethernet on all 86 nodes, and a low latency SCI/Dolphin interconnect on 25 nodes

More information about the system can be obtained at www.parallab.uib.no.

Bibliography

[Bar95] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI–I–95–2–003, Max-Planck-Institut für Informatik, 1995. Available from: `citeseer.ist.psu.edu/article/barth95davisputnam.html`.

[Boa08] OpenMP Architecture Review Board. Openmp c and c++ application program interface [online]. 2008. Available from: `www.openmp.org`.

[BS97] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997. Available from: `citeseer.ist.psu.edu/article/bayardo97using.html`.

[CKPS00] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *EUROCRYPT*, pages 392–407, 2000.

[CL04] Edward Corwin and Antonette Logar. Sorting in linear time - variations on the bucket sort. *J. Comput. Small Coll.*, 20(1):197–202, 2004.

[CLO92] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra (Undergraduate Texts in Mathematics)*. Springer, 1992.

[CP05] Christophe De Canniere and Bart Preneel. Trivium specifications. Technical report, European Network of Excellence for Cryptology (Ecrypt), 2005. Available from: `http://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf`.

[DA08] Beman Dawes and David Abrahams. Boost c++ libraries [online]. 3 2008. Available from: `http://www.boost.org` [cited June 2, 2008].

[DABC93] Olivier Dubois, Pascal André, Yacine Boufkhad, and Jacques Carlier. SAT versus UNSAT. *Second DIMACS Implementation Challenge*, 1993.

[DKK$^+$99] Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, and David Sehr. An overview of the Intel IA-64 compiler. *Intel Technology Journal*, 1(Q4):15, 1999. Available from: `citeseer.ist.psu.edu/dulong99overview.html`.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. *Theory and Applications of Satisfiability Testing*, pages 61–75, 2005. Available from: `http://dx.doi.org/10.1007/11499107\_5`.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver [ver 1.2] [online]. 2003. Available from: `http://citeseer.ist.psu.edu/een03extensible.html`.

[ES04] Niklasn Eén and Niklas Sörensson. An extensible sat-solver. *Theory and Applications of Satisfiability Testing*, pages 502–518, 2004. Available from: `http://www.springerlink.com/content/x9uavq4vpvqntt23`.

[Fau99] Jean Charles Faugère. A new efficient algorithm for computing gröbner bases (f4). *Journal of Pure and Applied Algebra*, 139(1):61–88, June 1999.

[Fau02] Jean Charles Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f5). In *ISSAC '02: Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 75–83, New York, NY, USA, 2002. ACM.

[Fre95] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms.* PhD thesis, University of Pennsylvania, Philadelphia, 1995. Available from: `citeseer.ist.psu.edu/freeman95improvement.html`.

[GCE93] Matthew L. Ginsberg, James M. Crawford, and David W. Etherington. Dynamic backtracking. *Journal of AI Research*, 1:25–46, 1993. Available from: `citeseer.ist.psu.edu/ginsberg96dynamic.html`.

[Iwa04] Kazuo Iwama. Worst-case upper bounds for ksat (column: Algorithmics). *Bulletin of the EATCS*, 82:61–71, 2004.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1:167–187, 1990.

[Kar05] Björn Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost.* Addison Wesley Professional, 2005.

[Knu76] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, 1976.

[Mag08] Magma computational algebra system [online]. May 2008. Available from: `http://magma.maths.usyd.edu.au/magma/` [cited 31.05.2008].

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001. Available from: `citeseer.ist.psu.edu/moskewicz01chaff.html`.

[MS95] David R. Musser and Atul Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[MS99] Joao P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA '99: Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, pages 62–74, London, UK, 1999. Springer-Verlag.

[MSS96] Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, 1996. Available from: `citeseer.ist.psu.edu/marques-silva96grasp.html`.

[oCoST99] U.S. Department of Commerce and National Institute of Standards & Technology. Data encryption standard (des). *Federal Information Processing Standards Publication*, 1999.

[Ouy98] Ming Ouyang. How good are branching rules in dpll? *Discrete Applied Mathematics*, 89(1-3):281–286, 1998.

[Pre96] D. Pretolani. Efficiency and stability of hypergraph sat algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:479–498, 1996.

[Rad04] Håvard Raddum. Solving non-linear sparse equation systems over gf(2) using graphs. *University of Bergen*, 2004.

[RS06] Håvard Raddum and Igor Semaev. New technique for solving sparse equation systems. *Cryptology ePrint Archive*, 475, 2006. Available from: `http://eprint.iacr.org/`.

[RS07] Håvard Raddum and Igor Semaev. Solving mrhs linear equations. *Extended abstract in Proceeding of WCC'07, Versailles, France*, pages 323–332, April 2007.

[Sem05] Igor Semaev. On solving sparse algebraic equations over finite fields. extended abstract. *Design, Codes and Cryptography*, 2005. Available from: `http://dx.doi.org/10.1007/S10623-008-9182-x`.

[Sem07] Igor Semaev. On solving sparse algebraic equations over finite fields ii. Cryptology ePrint Archive, Report 2007/280, 2007. Available from: `http://eprint.iacr.org`.

[Sem08] Igor Semaev. Solving sparse boolean equations with circuit lattices. *submitted*, 2008.

[Sin06] Daniel Singer. *Parallel Resolution of the Satisfiability Problem: A Survey*, chapter 5. Wiley Interscience, October 2006. Available from: `http://lita.sciences.univ-metz.fr/~singer/SatReview.pdf`.

[SV05] Daniel Singer and Alain Vagner. Parallel resolution of the satisfiability problem (sat) with openmp and mpi. In *PPAM*, pages 380–388, 2005.

[Wei94] Mark Allen Weiss. *Data structures and algorithm analysis in C++*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[YCC04] Bo-Yin Yang, Jiun-Ming Chen, and Nicolas Courtois. On asymptotic security estimates in xl and gröbner bases-related algebraic cryptanalysis. In *ICICS*, pages 401–413, 2004.

[ZM88] Ramin Zabih and David A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *AAAI*, pages 155–160, 1988.

[ZV00] A. Zakrevskii and I. Vasilkova. Reduction of large systems of logical equations. *4th Int. Workshop on Boolean Problems, Freiberg University*, pages 21–22, 2000.