3MCard

A Lookahead Cardinality Solver

Martijn van Lambalgen
Master thesis in Computer Science

Parallel and Distributed Systems group
Department of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, The Netherlands

Supervisors: dr. Hans van Maaren and ir. Marijn Heule

3MCard

A Lookahead Cardinality Solver

Martijn van Lambalgen
Master thesis in Computer Science

Parallel and Distributed Systems group
Department of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, The Netherlands

Supervisors: dr. Hans van Maaren and ir. Marijn Heule

Committee:
ir. M.J.H. Heule
dr. H. van Maaren
prof.dr.ir. H.J. Sips
prof.dr. C. Witteveen

October 21, 2006

Contents

List of Tables

List of Algorithms

List of Figures

Preface

This thesis is the result of my master's project for the Parallel and Distributed Systems group. I did the research however at the Algorithms group. During the first year of the Master program, I got interested in the satisfiability problem. On the one hand it was the simplicity of assigning 1's and 0's to variables. On the other hand it was the huge power it gives you when applying this on all kinds of complicated problems. I was given the opportunity to work on a relatively new subject. A subject of which the usefulness was not clear, but for which the prospects were positive. A level of solving methods, which was almost unexplored in the literature: cardinality solving. I am grateful that I was given the chance to contribute to this field and that I was given enough freedom to choose my own direction.

Several weeks ago, my supervisor asked me to think of a name for my project. Someone once said to me that choosing a name for a project is fun, and it was vital for his success on that project. Without a name he had no idea where to start. I partly agree with him. Choosing a name is fun. However, I'm glad that it is not as vital for me as it was for him.

I'd like to compare the past year with walking a marathon. It takes you a lot of energy, you're happy when you've reached the finish, but you always know that you should have done better. Therefore I'd like to name my project after the $3M$ marathon in Leiden, the city where I live. I will use the name 3MCard.

This all wouldn't have been possible without the support of many people. Therefore I would like to thank everone who has supported me in my study and specifically during the last year. First of all, I want to thank my supervisors. They were always full of advice and ideas, and they critiqued my work when needed. Their trust in the project was important for my motivation during the year. Furthermore I would like to thank my parents for giving moral support and for always being there when I needed them. Finally I want to thank all of my friends who showed interest in my research.

Martijn van Lambalgen

Delft, October 21, 2006

Chapter 1

Introduction

Have you ever tried to put 5 pigeons in 4 holes without putting two pigeons in the same hole? This is a well known problem which is called the pigeonhole problem.

> *Can you put a number of pigeons in separate holes, if the number of holes is smaller than the number of pigeons?*

Clearly this is not possible. It is not difficult to understand, but how do you prove it? Even that is not difficult. There are many ways to achieve this, ranging from trying all different possible combinations, till using mathematical induction. Viewing this problem as a logic problem, it becomes possible to use a general solving method. There exists a hierarchy of solving methods which differ in the level of reasoning. On one of the lower levels we have the Boolean satisfiability (or SAT) problem. The SAT problem is a decision problem, which tries to give an answer to the question whether a solution exists for a given logic problem.

On higher levels in the hierarchy we have for example the pseudo-Boolean (or PB) problem and Integer Linear Programming (or ILP). Both methods use higher level constraints to define and reason with a logic problem. Reasoning is done with Boolean variables and Integer variables respectively, where both methods can additionally define an optimization function. This addition makes the PB problem and the ILP optimization problems instead of decision problems. Higher level solving methods are also possible, but because they are not interesting for our purposes, they won't be discussed here. The three methods which were mentioned will be explained to give an idea of the different efficiencies.

As mentioned before, the SAT problem is a decision problem. It deals with deciding whether a giving logic problem has a solution or not. Satisfiability solving (or SAT solving) involves finding a solution for a logic problem or proving that no solution exists. The basis for most modern satisfiabil-

ity solvers is the DPLL algorithm, designed by Davis, Putnam, Loveland and Logemann [6, 7]. Researchers have been working on this algorithm for the past forty years, but in the last ten years we have seen a significant growth in the use of SAT solvers [22]. Examples of modern state-of-the-art SAT solvers are March, Minisat and zChaff [13, 11, 12]. Applications lie for example in solving scheduling problems and in integrated circuit design. Recently SAT solving is also increasingly being used in a variety of disciplines such as software verification and bounded model checking.

Modern solvers use the conjunctive normal form (CNF) to define SAT problems. It means that the problem is defined as a conjunction of a number of *clauses*. Clauses are a disjunction of a number of Boolean variables or their negations, where a variable can only have the value `true` or `false`, often represented by respectively 1 and 0. A literal is either a positive or a negative occurrence of a variable, often displayed as x_i or $\neg x_i$. An example of a SAT problem is: $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_4) \wedge (\neg x_1 \vee x_4 \vee x_5)$, where the $\wedge$ is the logical AND between the clauses and $\vee$ is the logical OR between literals within a clause. A clause is satisfied when at least one literal is satisfied, and a formula is satisfied when all clauses are satisfied. SAT solving means deciding whether an assignment of variables exists that satisfies the given CNF. The SAT problem mentioned above is satisfiable, because $\{x_1, \neg x_2, x_3, x_4 \neg x_5\}$ is a valid assignment, which satisfies all clauses.

In recent years a new type of solvers has emerged, the pseudo-Boolean (PB) solvers. These solvers accept a higher level representation of logic problems and allow for an optimization function to be defined. This basically changes the decision problem into an optimization problem. The main advantage of the higher level representation, is that logic problems can be defined more efficiently. A PB problem consists of a number of PB constraints, which must all be satisfied to solve the problem. If an optimization function is defined, the solver must additionally optimize this to solve the problem. A PB constraint is an inequality $a_1 l_1 + a_2 l_2 + \ldots + a_n l_n \geq k$, where l_i can be both the Boolean variable x_i and its negation $\neg x_i$. Again variables can only have the values `true` and `false`. The coefficients a_i and the right-hand side k can have any integer value. Note that when all a_i and k are equal, the constraint reduces to a CNF clause. To satisfy this constraint, the left-hand side (LHS) must be greater than or equal to the right-hand side (RHS). In general, for one PB constraint an exponential number of SAT clauses would be required [9]. It is however also possible to make such a translation more efficient by introducing *dummy* variables. Using dummy variables, it becomes possible to create translations which require less constraints. For example, Warners [20] created a translation which only requires a linear number of CNF clauses. Examples of PB solvers are PBS [1], OPBDP [4] and Pueblo [18].

An even more general solving method is Integer Linear Programming, or ILP. ILP generalizes the PB constraints further by letting the vari-

ables have any integer value, not just Boolean values. As with PB, an optimization function can be defined. An ILP constraint is an inequality $a_1 l_1 + a_2 l_2 + \ldots + a_n l_n \geq k$, where the literals l_i can have any integer value. The coefficients a_i and the RHS k can have any rational value.

A special case of a PB constraint is a *cardinality* constraint. A cardinality constraints is a PB constraint where all coefficients are equal to 1. A RHS of k means that at least k literals in the constraint must be satisfied. A cardinality constraint with a RHS of 1 reduces to a CNF clause. In the literature cardinality constraints are also referred to as Extended clauses [3] or TL clauses [21]. We will use only the term cardinality constraint.

Because cardinality constraints are a special case of PB constraints, and because they are still more general than CNF clauses, we can place cardinality constraints between SAT and PB if sorted on level of generalization.

1. ILP: coefficients and right-hand side can have any Rational value, all variables must be integer
   e.g. $3x_1 + 5x_2 + 17x_3 \geq 123$
2. PB: coefficients and right-hand side are integer, variables are Boolean
   e.g. $2x_1 + 1x_2 + 3x_3 \geq 3$
3. CARD: coefficients are 1, right-hand side is integer and variables are Boolean
   e.g. $x_1 + x_2 + x_3 + x_4 \geq 2$
4. SAT: coefficients and right-hand side are 1, variables are Boolean
   e.g. $x_1 + x_2 + x_3 \geq 1$

From these four expression languages, ILP can most efficiently define logic problems. However, this does not mean that it is best to define all problems with ILP. The higher level language is more efficient, but it also needs more complex reasoning, which may require more time. Therefore, if you can define a given problem equally efficient with a lower level language, or in other words, if you can define the problem with an equal number of lower level constraints, then this is preferred. Generally one prefers the lowest level language with which you can solve the problem as quick as possible.

It appears that many real world problems can be naturally defined with cardinality constraints. Think of scheduling problems for example: constraints like "at least three people from a team are required for job X", or "No more than five hours may be spent for a number of tasks" are quite common. H. Zhang, D. Li and H. Shen created a SAT based scheduler for tournament schedules [21]. They used cardinality constraints to define a scheduling problem. Examples of such constraints are: *no team can play*

more than one game per day or *no three home games or road games on consecutive days are allowed*. They created a solver that can handle a mix of CNF clauses and cardinality constraints. They reported that their solver outperformed ordinary satisfiability solvers and one of the best PB boolean solvers on the scheduling problems.

Based on the idea that the straightforward translation of many logic problems involve many cardinality constraints, we decided to create a pure cardinality solver, which is aimed specifically at cardinality constraints and therefore handles both CNF clauses and cardinality constraints in the same way. Such a solver has the advantage of a more elegant problem representation and the ability to reason more efficiently with this representation than SAT solvers. Also, it does not have the extra complexity of dealing with coefficients, for which special datastructures would be needed. As with the PB problem, it is possible to define an optimization function, though this is not required.

Previous research on cardinality solving focused on the theoretical aspects or on a modular implementation. P. Barth [3] gave an algorithm to detect redundancy and he created an efficient algorithm to translate PB constraints into cardinality constraints. H. Zhang, D. Li and H. Shen created a SAT based scheduler for tournament schedules. They used cardinality constraints to define schedules.

The questions we try to answer in this thesis are:

- *Is a solver which is purely based on cardinality constraints useful?*
- *What are the main advantages of such a solver*
- *Under what circumstances can we recommend a cardinality solver instead of a solver based on another solving method?*

The remainder of this thesis is structures as follows. First, in chapter 2 the DPLL algorithm will be explained. Also we discuss the different advantages and drawbacks of several types of solvers and we make a choice between these types to base our solver on. In chapter 3 we will give the architecture of a cardinality solver, based on the chosen type. Also the most important differences with SAT solvers are highlighted. In chapter 4 a new algorithm is explained that can be used to filter cardinality constraints from a series of CNF clauses. This algorithm can be used to create cardinality benchmarks from existing SAT benchmarks. Chapter 5 shows how to translate PB constraints into cardinality constraints. This algorithm can be used to create cardinality benchmarks from existing PB benchmarks. Chapter 6 gives an overview of the benchmarks that will be used to test new techniques on our solver. One of these techniques is the diff-function in chapter 7. An important aspect is the addition of a resolution strategy, which is explained

in chapter 8. In chapter 9 the results are given, and finally in chapter 10 we come to a conclusion and discuss future work.

Chapter 2

DPLL

In this section the well-known DPLL algorithm will be explained. The DPLL algorithm is the basis for most modern satisfiability solvers and we will use this algorithm to explain the main differences between two types of solvers that are most common. We will base our solver on one of these two types and discuss the theoretical advantages and disadvantes. Finally a choice between the two types will be made.

There are several categories of solvers: Lookahead solvers, conflict-driven solvers and local search solvers, of which the first two generally use the well-known DPLL search algorithm, constructed by Davis, Putnam, Logemann and Loveland [6, 7]. Local search solvers are usually extremely fast when it is known that many solutions exist and the search space doesn't need to be checked completely for a solution. A disadvantage of these solvers is that because they don't search the whole search space, they can't guarantee that a formula is unsatisfiable when no solution was found.

We aim for a cardinality solver that *can* return both `SAT` and `UNSAT` as an answer and therefore we will only consider the lookahead and conflict-driven solvers. The details of the algorithm used by these types of solvers are given later in this section.

Before we can explain the DPLL algorithm we first need to explain the concept of unit constraints and unit propagation.

2.1 Unit constraints and unit propagation

Suppose you have the following constraint:

$$x_1 + x_2 + \neg x_3 + x_4 + \neg x_5 \geq 3$$

During the search for a satisfiable solution, one might set both x_1 and x_4 to `false`. Then the constraint reduces to:

$$x_2 + \neg x_3 + \neg x_5 \geq 3$$

There is now only one way to satisfy this constraint: All three literals must be satisfied. When there is only one way to solve a constraint, we call it a *unit constraint*. For cardinality constraints this can easily be seen by checking whether the following holds:

$$length = RHS$$

After having set these three literals, we want to use this knowledge also in other constraints in the formula. Therefore we want to *propagate* the unit constraint. This means that we will update all constraints containing either one of those literals, or their complements. This process is called *unit propagation*. Note that during unit propagation constraints are updated, which may again result in new unit constraints. Therefore this algorithm will be executed iteratively until no new unit constraints can be found.

2.2 The basic algorithm

The DPLL algorithm can be written as a recursive function or as an iterative one. In algorithm 1 an iterative version of the basic DPLL algorithm is shown, as given in [22]. This version can help to explain the main differences between lookahead and conflict driven solvers.

Before the DPLL algorithm is called, it is possible to start with a preprocessing phase, to *clean up* the formula. Many solvers use preprocessing in one form or another. This is not required however.

The DPLL algorithm starts with unit propagation. It is possible that the original formula already contains unit constraints, and by propagating them, the formula may be reduced considerably. If this does not already lead to an answer, the branching is started. One variable at a time is selected through GetBranchVariable() and given a value. As this may lead to new unit constraints, the method IterativeUnitPropagation() is called each time a variable is set. The result is either a conflict, *or* a satisfiable solution, *or* an undetermined status. In case of a satisfiable solution we are done. As long as no conflicts occur we can continue setting variables. If a conflict occurs, one or more variables have a wrong value and we need to backtrack until the last decision that resulted in this conflict. This *backtrack level* is decided in the method AnalyzeConflict(), which may also further analyze the conflict. This process is continued until either a solution has been found, or all assignments have been checked and appear to lead to conflicts. When all assignments lead to a conflict the problem is unsatisfiable.

2.3 Lookahead vs. Conflict driven

In algorithm 1 you can clearly point out the differences between the lookahead solvers and the conflict driven solvers. Lookahead solvers focus mainly

Algorithm 1 DPLL

```
1: F^0 := IterativeUnitPropagation(F)
2: if F^0 = ∅ then
3:   return "satisfiable"
4: else if F^0 contains a conflicting constraint then
5:   return "unsatisfiable"
6: end if
7: while TRUE do
8:   x := GetBranchVariable()
9:   while TRUE do
10:    F^{i+1} := IterativeUnitPropagation(F^1)
11:    if F^{i+1} contains conflicting constraint then
12:      blevel := AnalyzeConflict()
13:      if blevel = -1 then
14:        return "unsatisfiable"
15:      else
16:        backtrack(F^{blevel})
17:      end if
18:    else if F^{i+1} = ∅ then
19:      return "satisfiable"
20:    else
21:      break
22:    end if
23:  end while
24: end while
```

on the selecting procedure of the next branch variable. This happens in the method GetBranchVariable(). Conflict driven solvers focus more on the analysis of conflicts. They try to make sure that certain parts of the search space where conflicts occurred won't be checked twice. This happens in the method AnalyzeConflict().

2.3.1 Lookahead solvers

A lookahead solver will test (a part of) all variables and it *looks* what happens when that variable would be selected. For each variable both values (`true` and `false`) are tried. The result is examined and any conflicts on both values mean that that variable is not the one that will lead to a satisfiable solution (given the current assignment). Important are the *failed literals*. These are literals that lead to conflicts when used. Therefore they can be set directly to the other value. When these literals have been propagated, the search space can be reduced significantly. After the lookahead phase, a choice must be made between the variables that didn't lead to a conflict. Usually the variable that leads to the greatest reduction is selected. However, the definition of *reduction* is not unambiguous here. Different solvers use different strategies. The challenge is to find a rule for valuing a reduction, that leads to the fastest decision between *satisfiable* and *unsatisfiable*. A difficulty in constructing a cardinality lookahead solver is, that you must decide which clauses are more important. CNF only consists of clauses with a right-hand side equal to 1. Cardinality constraints have variable right-hand sides. The problem of valuing the different clauses therefore becomes in a way a two-dimensional problem.

2.3.2 Conflict driven solvers

A conflict-driven solver doesn't use such an advanced algorithm for selecting a branch variable. It starts with a variable and sees what happens. As soon as a conflict arises, this conflict will be analyzed: What decision led to the conflict? What combination of variables gave rise to this? A modern conflict driven solver will try to prevent such situations in the future and adds a *conflict clause*. This conflict clause will discover such a situation as soon as it occurs, so the solver doesn't need to continue in this situation until the original conflict occurs again. Because during the solving process many conflicts arise, also many conflict clauses can be added. The challenge is to add only those conflict clauses that are most useful and will often prevent searching parts of the search space that don't contain solutions.

Conflict clauses are generated by performing resolution on the clauses that caused the conflict. If a cardinality solver is constructed as a conflict-driven solver, this might lead to an extra complexity, because, as Dixon et al [9] already mentioned, resolution on cardinality constraints can result

in PB constraints. As mentioned before, PB constraints can be translated into cardinality constraints, but translation is not trivial and it will take a considerable amount of time when used often. On the other hand, most PB solvers use the conflict driven algorithm, so apparently a conflict driven approach has a positive effect on the higher level constraints.

2.4 Choice

Probably both types of solvers give opportunities to create a very good cardinality solver. Based on the fact that conflict driven solvers probably need many translations of PB constraints to cardinality constraints, and because it is not known whether this will become a bottleneck, we choose the lookahead type to construct a cardinality solver.

Chapter 3

Architecture

In this chapter we give the basic structure of our cardinality solver. Here we focus only on the framework of the solver, so no heuristics will be used yet. In section 3.1 we start with the DPLL algorithm. The differences between a SAT solver and a cardinality solver lie mainly in the way the constraints are maintained and updated internally. The algorithms for this are discussed in section 3.2. In section 3.3 we will explain the lookahead algorithm.

3.1 DPLL

The DPLL algorithm walks through the search tree of possible assignments by branching on variables and checking through iterative unit propagation whether conflicts occur because of wrong branches. The process of branching on variables, iteratively propagating unit constraints and backtracking when necessary, continues until a solution has been found or until it is clear that no solution exists. A solution is found when every variable has been assigned a value and no conflicts occurred. When every possible assignment results in a conflict, the formula is unsatisfiable. Algorithm 2 shows a recursive version of this.

The iterative unit propagation iteratively updates the formula by processing known facts until either a conflict is found, or until there are no more unit constraints (see algorithm 3). In every iteration unvalued literals from the unit constraints are set and all constraints are updated. This means that any constraint that contains a literal, that was satisfied, must be *shortened*. Both the length and the RHS side are decreased by 1. Any constraint that contains a literal that was falsified must be *strengthened*. Only the length is decreased by 1. This may result in a conflict, after which the unit propagation is terminated.

Algorithm 2 DPLL($\mathcal{F}$)

```
1: F := IterativeUnitPropagation(F)
2: if F = ∅ then
3:    return "satisfiable"
4: else if F contains conflicting constraint then
5:    return "unsatisfiable"
6: end if
7: x := GetBranchVariable()
8: if DPLL(F ∪ x) = "satisfiable" then
9:    return "satisfiable"
10: else
11:    return DPLL(F ∪ ¬x)
12: end if
```

Algorithm 3 IterativeUnitPropagation($\mathcal{F}$)

```
1: while F doesn't contain conflicts and a unit constraint u exists do
2:    for all unvalued literals l_i in u do
3:       satisfy l_i
4:       shortenConstraints(F, l_i)
5:       strengthenConstraints(F, ¬l_i)
6:    end for
7: end while
8: return F
```

3.2 Support for cardinality constraints

In the solver a database must maintained, containing all constraints. Each time variables are set or unset, all constraints need to be updated. Managing the constraint database for a cardinality solver is not the same as for a SAT solver. With a SAT solver it is possible to just remove any clause containing a literal that has been satisfied. With a cardinality solver, this is no longer possible, because often multiple literals must be satisfied before the constraint is satisfied. This is determined by the RHS of the constraint. Compare the following constraints:

CNF	cardinality
$a \vee b \vee c \vee d$	$a + b + c + d \geq 2$

Suppose you have set a to `true`. Now the CNF clause is satisfied and can be removed. However, the cardinality constraints is not yet satisfied. It is one step closer, but it can't be removed yet.

A similar difference appears when falsifying a literal. With a SAT solver it was possible to check for a conflict by looking whether the clause was empty. With a cardinality solver, this is no longer possible, because conflicts can also occur if there are still multiple literals. Compare the following constraints:

CNF	cardinality
$a \vee b \vee c \vee d$	$a + b + c + d \geq 3$

Suppose you have set a and b to `false`. Now the CNF clause becomes just shorter, but because it is not empty, there can be no conflict. The cardinality constraint however, does result in a conflict, because at least three literals need to be satisfied, which is no longer possible.

Because of these differences, the algorithms used for updating the constraint database must be generalized, so that they can deal with cardinality constraints.

3.2.1 Generalized update algorithms

For each constraint that contains the satisfied literal, both the length and the RHS decrease by 1. This may reduce the RHS to 0, which means the constraint is satisfied. The pseudo code for this is shown in algorithm 4. For each constraint that contains the complement of the satisfied literal, only the length decreases by 1. This may reduce the length to either the RHS, or smaller than the RHS. In the first case the constraint has become unit. In the latter case, it is no longer possible to satisfy the constraint, which means there's a conflict. The pseudo code for this is shown in algorithm 5.

To recognize the status of a constraint, it is necessary to see when a cardinality constraint is satisfied, so it can be removed. Also, we need a way to see whether a constraint is a unit constraint, or whether a constraint

Algorithm 4 shortenConstraints($\mathcal{F}$, l_i)

```
1: for all constraints c in F containing l_i do
2:    length(c) := length(c) - 1
3:    rhs(c) := rhs(c) - 1
4:    if isSatisfied(c) = true then
5:       remove c from F
6:    end if
7: end for
```

Algorithm 5 strengthenConstraints($\mathcal{F}$, $\neg l_i$)

```
1: for all constraints c in F containing ¬l_i do
2:    length(c) := length(c) - 1
3:    U := isUnit(constraint)
4:    if U = true then
5:       mark c as unit constraint
6:    else if U = conflict then
7:       return
8:    end if
9: end for
```

contains a conflict. We need to implement the methods isSatisfied and isUnit from algorithm 4 and 5.

For a fast implementation, we will maintain two values for each constraint, which can be used to distinguish between these situations. First we will use the RHS `rhs` for constraint c, under partial assignment P. Second we will use the number of unvalued literals `unvalued` for constraint c, under partial assignment P.

$$rhs(c, P) = rhs(c, \emptyset) - \sum_{\{P \cap c\}} l_i$$
$$unvalued(c, P) = |U| \quad \text{with } U = \{l_i | l_i \notin P\}$$

The RHS for c can be calculated as the original RHS minus the sum of all valued literals in c. The number of unvalued literals is calculated as the number of literals that are not in the partial assignment P. With these two values every status a constraint can get is easily recognizable.

- Satisfied constraints can simply be identified by checking whether the following holds:
$$rhs(c, P) \leq 0$$
- Unit constraints can be identified by checking whether the number of unvalued literals is equal to the value of the RHS.
$$unvalued(c, P) = rhs(c, P)$$

- Conflicts occur when there are less unvalued literals than the value of the RHS.

$$unvalued(c, P) < rhs(c, P)$$

The resulting pseudo code for the methods isSatisfied and isUnit are shown in algorithm 6 and 7.

Algorithm 6 isSatisfied(constraint)

```
1: return rhs(constraint) ≤ 0
```

Algorithm 7 isUnit(constraint)

```
1: if unvalued(constraint) = rhs(constraint) then
2:    return true
3: else if unvalued(constraint) < rhs(constraint) then
4:    return conflict
5: else
6:    return false
7: end if
```

3.3 The lookahead algorithm

The method GetBranchVariable() in algorithm 2 is implemented in a lookahead solver by the lookahead method (see algorithm 8). For some or all free variables x a lookahead will be performed. First x is set to `true` and through unit propagation the resulting formula is obtained. The same is done when setting x to `false`. If both resulting formulas contain conflicts, the current situation is unsatisfiable. The answer "`unsatisfiable`" is returned. If only one of the two formulas contains a conflict, the other formula is selected as the new formula. This means that the literal which resulted in a conflict is a failed literal. In all other situations a value must be given to the usefulness of the variable. This is done with the diff function, which will be discussed in chapter 7. When all variables have been checked, the variable with the highest value for both polarities is returned, which is decided by the MixDiff function. This is the variable that will be used in the DPLL for the next branch.

Algorithm 8 GetBranchVariable_LOOKAHEAD($\mathcal{F}$)

repeat
  for all free variables in $\mathcal{F}$ do **do**
    $\mathcal{F}' :=$ IterativeUnitPropagation($\mathcal{F} \cup x_i$)
    $\mathcal{F}'' :=$ IterativeUnitPropagation($\mathcal{F} \cup \neg x_i$)
    if $\mathcal{F}'$ contains empty constraint and $\mathcal{F}''$ contains empty constraint **then**
      return "`unsatisfiable`"
    else if F' contains empty constraint **then**
      $\mathcal{F} := \mathcal{F}''$
    else if F'' contains empty constraint **then**
      $\mathcal{F} := \mathcal{F}'$
    else
      $H(x_i) :=$ MixDiff(diff($\mathcal{F}$,$\mathcal{F}'$), diff($\mathcal{F}$,$\mathcal{F}''$))
    end if
  end for
until no new literals are detected
return x_i with highest $H(x_i)$

Chapter 4

From SAT to cardinality

Frequently, SAT benchmarks contain a series of CNF clauses that are equivalent to a cardinality constraint. Because (until now) no cardinality solvers have been designed, there are also no cardinality benchmarks. Problems where the natural structure contains cardinality constraints are therefore often written as SAT benchmarks. We need to find a way to filter the cardinality constraints from a set of CNF clauses. Deciding whether a set of cardinality constraints is equivalent to a set of CNF clauses is an NP-complete problem. Finding an optimal translation is even harder. Therefore we will not try to make the translation optimal. We created an algorithm that can syntactically recognize cardinality constraints from a list of CNF clauses. This algorithm is also be NP-complete. Although this will generally lead tot non-optimal translations, we think that this will already be beneficial. With this algorithm it becomes possible to use existing CNF benchmarks, and benefit from the advantages that the higher level of reasoning gives.

4.1 Domination

Before we will start with the translation, we need to explain the concept of domination, as this will be needed in our algorithm and also later in this thesis.

Domination of one constraint over another constraint means that satisfying the first constraint automatically also satisfies the second constraint and unsatisfying the second constraint also automatically leads to the unsatisfiability of the first constraint. The first constraint is said to *dominate* the second constraint, which means that the second constraint is redundant and can be left out of the problem without affecting the satisfiability of the problem. In the literature domination is also called *subsumption* or *redundancy*. We will use the term *domination*, because this term has already been used in combination with cardinality constraints [3]. First the basics of domination are explained.

Look at the following example:

$$a + b + c \geq 1$$
$$a + b + c + d + e \geq 1$$

In this case the literals from the first clause are a subset of the literals in the second clause. The first clause is said to dominate the second clause, which means that the second clause can be removed without any consequences for the satisfiability of the problem. If the original problem was satisfiable, then the clause $a \vee b \vee c$ must contain at least literal that is `true`. Therefore the second clause will also be true. If the original problem was unsatisfiable, then removing a clause would only be a problem if this clause was the only unsatisfiable clause. Because this would mean that all five literals should be `false`, the first clause also needs to be unsatisfiable. Therefore removing the second clause doesn't change the satisfiability in this case. The first clause is now called the *dominating clause* and the second clause is called the *dominated clause*. With cardinality constraints, checking domination by looking if the literals are a subset of another constraint, doesn't work. This can be seen from the following example:

$$a + b + c \geq 2$$
$$a + b + c + d + e \geq 3$$

With *regular* domination, the only requirement is that the literals from one clause are a subset of the literals from another clause. Satisfying the first clause then automatically leads to satisfaction of the second clause. In the example above this does not work however (satisfy for example literals a and b). The reason for this is that with cardinality constraints the RHS's can differ.

P. Barth [3] solved this problem and shows that the following holds:

$$L \geq d \text{ dominates } L' \geq d' \text{ iff } |L \backslash L'| \leq d - d'.$$

where $L \geq d$ is a constraint with literals L and RHS d. This means that a constraint only dominates another constraint if the difference between their RHS's is at least as great as the number of literals that are in the first constraint, but not in the second constraint.

An example of this is the following:

$$a + b + c + d + e \geq 4$$
$$a + b + c + f \geq 2$$

The first constraint dominates the second constraint, because the number of literals that are in the first constraint, but not in the second constraint is 2 (d and e), and this is the same as the difference in RHS's. Note that it is not necessary anymore that the literals in the first constraint are a subset from the literals in the second constraint.

4.2 Translation of cardinality to CNF

To syntactically recognize a cardinality constraint from a series of CNF clauses, we first need to know how cardinality constraints are translated into CNF clauses. We need to find out which CNF clauses are together equivalent to a cardinality constraint. Multiple encodings are known, some of which introduce dummy variables to reduce the number of resulting constraints. See for example [19]. We will only consider the most straightforward encoding for translating cardinality constraints to CNF clauses, as we expect that this encoding will occur more frequently.

A cardinality constraint of length N and with RHS K means that at most $N - K$ literals may be falsified. In other words, In every combination of $N-K+1$ literals, at least 1 literal must be satisfied. Now, we can translate a cardinality constraint just by creating a CNF clause for every combination of $N - K + 1$ literals. For example, the translation of $a + b + c + d + e \geq 4$ to CNF consists of the following clauses:

$$\begin{array}{lll} a \vee b & b \vee c & c \vee d \\ a \vee c & b \vee d & c \vee e \\ a \vee d & b \vee e & d \vee e \\ a \vee e & & \end{array}$$

Another example is the constraint $a+b+c+d+e \geq 3$, which results in the following translation:

$$\begin{array}{ll} a \vee b \vee c & a \vee d \vee e \\ a \vee b \vee d & b \vee c \vee d \\ a \vee b \vee e & b \vee c \vee e \\ a \vee c \vee d & b \vee d \vee e \\ a \vee c \vee e & c \vee d \vee e \end{array}$$

Any `N,M-CARD` constraint (a constraint of length N with RHS M) results after translation in CNF clauses of length N-M+1. For example, all `N,N-1-CARD` constraints result in CNF clauses of length 2. Note that this means that all constraints for which $N-M$ is equal, result in CNF clauses of the same length. This is an important fact to keep in mind when searching CNF clauses that are syntactically equivalent to a cardinality constraint.

The number of clauses needed for a translation depends on the number of literals L and on the RHS k and is calculated as:

$$\#clauses = \binom{L}{L-k+1}$$

In table 4.1 the number of necessary clauses is given. Each row gives the cardinality constraints that results in clauses with the same length, where each column contains constraints with the same RHS.

Table 4.1: Number of CNF clauses required for a `N,M-CARD` constraint

		right-hand side					
		1	2	3	4	5	6
length	2	1					
	3	1	3				
	4	1	4	6			
	5	1	5	10	10		
	6	1	6	15	20	15	
	7	1	7	21	35	35	21

Another important fact to notice is that every cardinality constraint `N,M-CARD` consists of several smaller cardinality constraints. Look at the cardinality constraint $a+b+c+d \geq 3$ which consists of the following CNF clauses:

$$\begin{array}{ll} a \vee b & b \vee c \\ a \vee c & b \vee d \\ a \vee d & c \vee d \end{array}$$

Notice that all these clauses also occur in the list of CNF clauses that are equivalent to $a+b+c+d+e \geq 4$. The same holds for $a+b+c+e \geq 3$, $a+b+d+e \geq 3$, $a+c+d+e \geq 3$ and for $b+c+d+e \geq 3$. Generally, with straightforward syntax, a cardinality constraint `N,M-CARD` consists of a set of N `N-1,M-1-CARD` constraints.

4.3 Cardinality graph

We want to search CNF clauses that are somehow connected and together are equivalent to a cardinality constraint. Therefore we create a cardinality graph.

Definition 4.3.1 (cardinality graph)**.** *Given a CNF formula, the cardinality graph is an undirected graph, where each CNF clause represents a node and where two nodes are connected by an edge iff they have the same length and have at least one literal in common.*

Suppose we have the following CNF:

$$\begin{array}{ll} a \vee b & a \vee d \\ a \vee c & b \vee d \\ b \vee c & \end{array}$$

For example, the first two clauses are connected in the cardinality graph, because they both contain the literal a. Figure 4.1 shows the cardinality graph for this cnf.

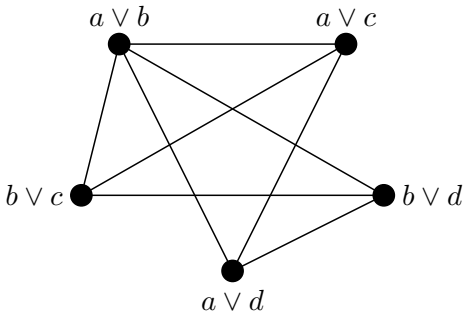

Figure 4.1: Cardinality graph

Two cardinality constraints can be filtered from the CNF: $a + b + c \geq 2$ and $a + b + d \geq 2$. Notice that both constraints use the clause $a \vee b$. Any CNF clause can be part of multiple cardinality constraints.

4.4 Paths to cardinality constraints

In the cardinality graph it is possible to create a path that passes all CNF clauses that are necessary to form one cardinality constraint (see figure 4.2). We say that this is *a path to a cardinality constraint.*

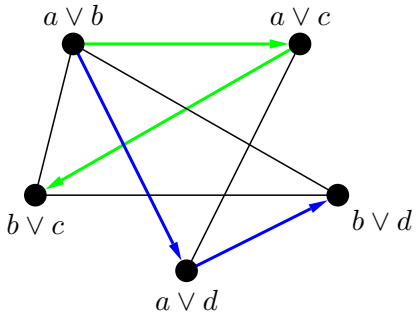

Figure 4.2: Cardinality graph with paths to cardinality constraints

For both cardinality constraints one possible path is shown. However, it is clear that in this situation several different paths can be constructed to one specific cardinality constraint. All three clauses that are used can serve as starting point for example. Although this situation is clear, we need to be sure that such a path always exists for longer constraints.

Theorem 4.4.1 (Paths to cardinality constraints)**.** *Suppose you have a CNF formula $\mathcal{F}$, and its corresponding cardinality graph $\mathcal{G}$. Suppose there is a set $\mathcal{S}$ of CNF clauses that are together equivalent to a cardinality constraint. Then there always exists a path in $\mathcal{G}$ that leads to this cardinality constraint.*

Proof:
To prove this, we will construct one specific type of path that can always be followed. Remember that every `N,M-CARD` constraint consists of several `N-1,M-1-CARD` constraints. The difference between the `N,M-CARD` constraint and one of the `N-1,M-1-CARD` constraints is that the latter has one literal less. This means that all CNF clauses that need to be added to the `N-1,M-1-CARD` constraint, will contain this same literal and are therefore connected in the cardinality graph. This also means that in the cardinality graph there will always be a path along all these additional clauses. So, if you have already created a path to a `N-1,M-1-CARD` constraint, and you know that the constraint can be extended to a `N,M-CARD` constraint, then there also exists an extension of the path which leads to `N,M-CARD`.

For the same reason there is also always a path from a `N-2,M-2-CARD` constraint to a `N-1,M-1-CARD` constraint. This reasoning can be extended till `N-M+1,1-CARD`, which is a single clause, and can be seen as the starting clause. □.

In figure 4.3 such a path is drawn for $a + b + c + d + e \geq 4$. After the starting clause, clauses are visited that lead to a `3,2-CARD` constraint. As soon as this constraint has been found, the path is continued to form a `4,3-CARD` constraint and after that even a `5,4-CARD` constraint. The length of a path (the number of nodes) is equal to the number of CNF clauses needed for the constraint. The different lengths for various constraints can be found in table 4.1.

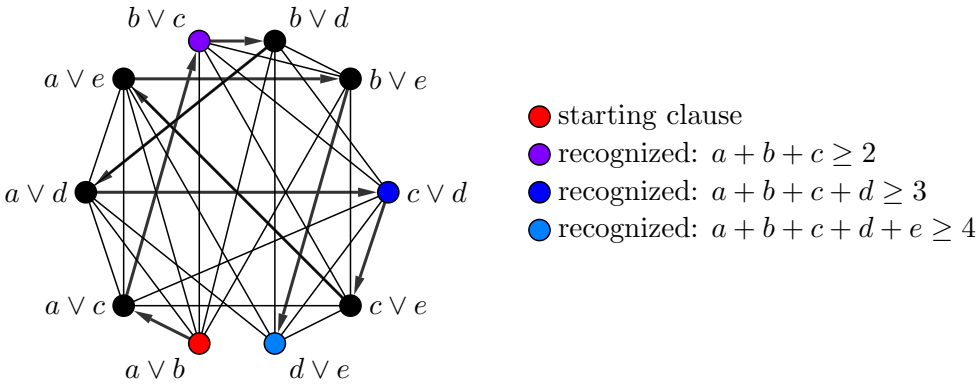

Figure 4.3: Cardinality graph showing a path to a cardinality constraint

If you would start with every CNF clause and follow all possible paths, then we are certain that all possible cardinality constraints will be found. Although this strategy would work, it also means that every cardinality constraint will be recognized many times, and that many paths will be created that don't lead to the recognition of a cardinality constraint. Because the overhead of walking all paths is far too great, we need some optimizations.

4.5 Optimization

To reduce the overhead of walking dead-end paths, and walking multiple paths that lead to the same cardinality constraint, we need optimizations. Ideally we want to create a situation for which every cardinality constraint is recognized through exactly one path, and where no paths are walked that don't lead to new cardinality constraints. We try to approach this situation by adding a few restrictions for searching the cardinality graph.

4.5.1 Specific paths

First, suppose the cardinality graph contains 10 clauses that are necessary for a `5,4-CARD` constraint. It is possible to create a random path through these clauses, but this would mean that there are 10! possible paths. The first restriction therefore is to only use the type of path that was described in the proof of theorem 4.4.1. This means that paths must start with searching a `N,2-CARD` constraint. From this point in the path, a literal is selected through which the path is continued. This means that additional clauses on the path must contain this literal, and may not introduce literals that weren't already found in previous clauses. See for example figure 4.4. When, at a certain point you have recognized the constraint $a+b+c \geq 2$, there are several clauses to continue the path. There are two different literals involved: d and e. For both literals there are several possibilities to complete the path and find the constraint $a+b+c+d \geq 2$ or $a+b+c+e \geq 2$. For each literal however, it is enough to try one path. If this path doesn't lead to the new constraint, then the other possible paths won't either. So, for every literal through which a path can be extended, only one path is tried.
In summary:

- Paths start with one clause, which can be seen as a `N,1-CARD` constraint. From this point paths are only extended through one literal until a cardinality constraint has been found that is one longer. For every literal, through which the path can be extended, only one path is tried.

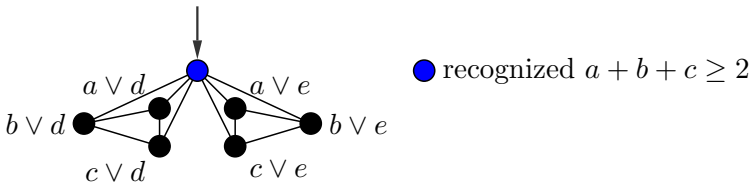

Figure 4.4: Possibilities to continue the path after a recognition

4.5.2 Starting points

It is still possible now to start with any clause used for the cardinality constraint. If a formula contains clauses necessary to form the constraint $a + b + c + d \geq 3$, then you can use $a \vee b$ for example as a starting point. After having recognized the cardinality constraint, and no other constraint can be found with $a \vee b$ as starting point, then we don't want any of the clauses that were used for the found cardinality constraints to be used as new starting point.

It is however possible that there are other cardinality constraints, which use some of these clauses. Such a constraint cannot be found by using the 'overlapping' clauses as starting point. However, because this constraint would also contain new clauses, it will still be found through these clauses. For example if there is a constraint $c + d + e \geq 2$, then the clauses $c \vee e$ and $d \vee e$ could serve as starting point.

Problems arise when these two clauses have also been marked not to be used as starting point. If at an earlier point the cardinality constraint $c + d + e + f \geq 3$ has been found, and $e \vee f$ was the starting point, then the constraint $c + d + e \geq 2$ won't be found and none of the necessary clauses will be used as starting point again. In this case you will never be able to find it again.

Now notice the fact that this only happens when the earlier found constraint contains the literals c,d and e. Remember the domination theory as explained in section 4.1. A cardinality constraint $L \geq d$ dominates another cardinality constraint $L' \geq d'$ if and only if $|L \setminus L'| \leq d - d'$ holds. The number of literals that appear in the earlier found constraint, but not in the constraint that cannot be found anymore is at least 1. In this example it was only the literal f. Because we are using only clauses of the same length, the difference between the RHS side and the number of literals is the same for both constraints. Therefore the differences in RHS's depend on the differences in number of literals, which means that $|L \setminus L'|$ is exactly equal to $d - d'$. Also if you found the constraint $c + d + e + f + g + h \geq 5$,

this constraint dominates the constraint $c + d + e \geq 2$ that cannot be found anymore.

So, if a constraint cannot be found anymore, because none of the necessary clauses are used as starting point anymore, then this constraint must be dominated by other constraints, and it doesn't matter anymore that it cannot be found.
To summarize:

- If a path leads to a cardinality constraint, then none of the clauses on this path may be used as a starting point for new paths.

4.5.3 Literal sequence

With the current restrictions it is still possible that a cardinality constraint is being found multiple times. The sequence in which literals are added to extend the path can differ. For example if a cardinality constraint $a + b + c + d + e \geq 4$ can be found, then it is possible to first create a path for $a + b + c \geq 2$, but it is also possible to start the path with $a + d + e \geq 2$. In both cases you can also find the `5,4-CARD` constraint. See also figure 4.5.

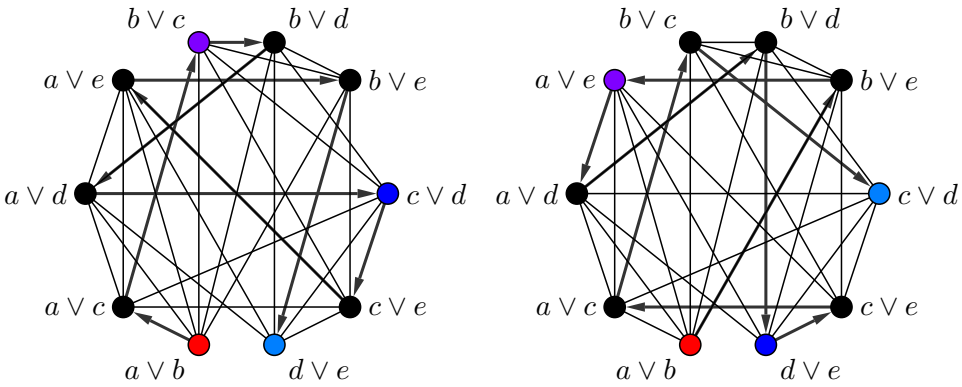

Figure 4.5: Multiple paths to the same cardinality constraint

This redundancy can be prevented by using the knowledge of previous found cardinality constraints. If at a certain moment you have found the constraint $a + b + c \geq 2$ and it is possible to extend the path both through d and e. If, through literal d you find $a + b + c + d + e \geq 4$, then it is not useful to try to extend $a + b + c \geq 2$ also through e. This can be prevented by disallowing to extend a path through literals that are also part of a constraint found through another extension of the path.
In summary:

- A path can only be extended through a certain literal, if this literal

doesn't also appear in cardinality constraints that have already been found through other extensions of this path.

4.5.4 Adding cardinality constraint

The last optimization deals with the moment when recognized cardinality constraints are added. If you add a cardinality constraint as soon as you find one, you will need useless redundancy checks every time you find a longer cardinality constraint on the same path, because longer constraints on the same path are redundant. To prevent this, we will only add a cardinality constraint if no longer cardinality constraints on the same path can be found. To summarize:

- Recognized cardinality constraints may only be added when it is certain that no longer cardinality constraints can be found on the same path. If there are longer cardinality constraints, then we don't add it at all.

With the optimizations mentioned, it becomes possible to walk through a large number of CNF clauses and filter all cardinality constraints. In the next section the pseudo code for the obtained algorithm is given.

4.6 Algorithm

Algorithm 9 gives the pseudo code for the cardinality recognition algorithm.

In this algorithm one clause is visited in every call to RECOGNIZE. First it is checked whether the current series of visited nodes form a cardinality constraint. If this is the case, then a list is created of literals through which the path can be extended. For each literal in this list one path is tried. This is done by calling the recognize method recursively with the updated list of visited clauses, and with the new connected clause and the added literal. If this call results in a longer cardinality constraint being found, then all literals in this constraint are added to the list totalResult, which contains all literals that are part of a longer cardinality constraint. These literals are also removed from the extensionLits list, so no new paths can be created through these literals. After all paths have been tried, the totalResult list is checked. If it is empty, this means that there were no longer cardinality constraint, and thus this constraint is the longest and must be added. The clause that was last visited is stamped, so it won't be used again as starting point. Finally the totalResult list is returned. If you have just visited a clause that doesn't lead to a new cardinality constraint, then this means that you are trying to extend the path through an already chosen literal. You must try to find one other clause that contains this added literal and use it to extend the path through another recursive call to recognize. If this

Algorithm 9 RECOGNIZE(visitedClauses, lastClause, addedLit)

```
1: visistedClauses = visitedClauses ∪ lastClause
2: complete = isCardinality();
3: if complete=true then
4:    for all clauses clause connected to lastClause do
5:       if clause contains a new lit then
6:          add lit to extensionLits
7:       end if
8:    end for
9:    for all literals lit in extensionLits do
10:      find connectedClause, containing lit and connected to lastClause
11:      result = RECOGNIZE(visitedClauses, connectedClause, lit)
12:      if result != null then
13:         for all literals flit in result do
14:            add flit to totalResult
15:            remove flit from extensionLits
16:         end for
17:      end if
18:   end for
19:   if totalResult = null then
20:      ADD(recognized cardinality constraint)
21:   end if
22:   STAMP(lastClause)
23:   return totalResult
24: else
25:   find new connectedClause containing addedLit
26:   if connectedClause = null then
27:      return null
28:   end if
29:   result = RECOGNIZE(visitedClauses, connectedClause, addedLit)
30:   if result != null then
31:      STAMP(lastClause)
32:   end if
33:   return result
34: end if
```

leads to a cardinality constraint being recognized, the current clause is being stamped. Then the result is returned

This recognize method must be called for any CNF clause that is not stamped yet.

Chapter 5

From PB to cardinality

Rewriting PB constraints to cardinality constraints potentially requires an exponential number of additional constraints. P. Barth gives a translation algorithm that minimizes the overhead needed [3]. His algorithm does not introduce additional variables, so the search space is not increased after the translation. Because of this the time needed for the translation grows exponentially with the size of the PB constraint. It should be noted that Warners [20] gave a linear-time algorithm, which *does* increase the search space. Because we don't want this, we will use the algorithm of Barth.

In short, the translation means that you must construct the set of all strongest cardinality constraints of all strict reductions of the PB constraints, and then remove the constraints that are dominated by others.

A *reduction* of a PB constraint means eliminating a literal. So, for example a reduction of $5x_1 + 4x_2 + 3x_3 + 2x_4 + 1x_5 \geq 11$ would be $5x_1 + 4x_2 + 3x_3 + 2x_4 \geq 10$ or $5x_1 + 4x_2 + 3 + 2x_4 + 1x_5 \geq 6$. If the original PB constraint can be satisfied, then also the reduced constraints can be satisfied.

A *strict reduction* of a PB constraint means that when eliminating a literal, you keep the RHS as high as possible. The strict reduction of $5x_1 + 4x_2 + 3x_3 + 2x_4 + 1x_5 \geq 11$, where the literal x_3 is removed, would then be $5x_1 + 4x_2 + 2x_4 + 1x_5 \geq 8$

The *strongest cardinality constraints* of a PB constraint can be constructed by updating the RHS to the minimum number of literals that need to satisfied, and then removing the coefficients. For example, the strongest cardinality constraint of the PB constraint $5x_1 + 4x_2 + 2x_4 + 1x_5 \geq 8$ is $x_1 + x_2 + x_4 + x_5 \geq 2$, because at least two literals need to be satisfied in the PB constraint.

Now, take for example the following PB constraint:

$$5x_1 + 4x_2 + 3x_3 + 2x_4 + 1x_5 \geq 11$$

The set of strongest cardinality constraints of all strict reductions is:

$x_1 + x_2 + x_3 + x_4 + x_5 \geq 3$ $x_1 + x_2 + x_3 + x_4 \geq 3$ $x_1 + x_2 + x_3 + x_5 \geq 2$
$x_1 + x_2 + x_4 + x_5 \geq 2$ $x_1 + x_3 + x_4 + x_5 \geq 2$ $x_2 + x_3 + x_4 + x_5 \geq 2$
$x_1 + x_2 + x_3 \geq 2$ $x_1 + x_2 + x_4 \geq 2$ $x_1 + x_2 + x_5 \geq 2$
$x_1 + x_3 + x_4 \geq 2$ $x_1 + x_3 + x_5 \geq 1$ $x_1 + x_4 + x_5 \geq 1$
$x_2 + x_3 + x_4 \geq 1$ $x_2 + x_3 + x_5 \geq 1$ $x_2 + x_4 + x_5 \geq 1$
$x_3 + x_4 + x_5 \geq 1$ $x_1 + x_2 \geq 1$ $x_1 + x_3 \geq 1$
$x_1 + x_4 \geq 1$ $x_1 + x_5 \geq 1$ $x_2 + x_3 \geq 1$
$x_2 + x_4 \geq 1$ $x_2 + x_5 \geq 1$ $x_3 + x_4 \geq 1$
$x_1 \geq 1$

After removing the redundant constraints, only 3 cardinality constraints remain, which are equivalent to the PB constraint:

$$x_1 + x_2 + x_3 + x_4 \geq 3 \quad x_1 + x_2 + x_5 \geq 2 \quad x_1 \geq 1$$

Another example is this PB constraint:

$$4x_1 + 2x_2 + 1x_3 - 4x_4 - 2x_5 - 1x_6 + 9x_7 \geq 1$$

This leads to the following set of strongest cardinality constraints of all strict reductions:

$x_1 + x_2 + x_3 + \neg x_4 + \neg x_5 + \neg x_6 + x_7 \geq 1$ $x_1 + x_2 + x_3 + \neg x_4 + \neg x_5 + x_7 \geq 1$
$x_1 + x_2 + x_3 + \neg x_4 + \neg x_6 + x_7 \geq 1$ $x_1 + x_2 + \neg x_4 + \neg x_5 + \neg x_6 + x_7 \geq 1$
$x_1 + x_2 + x_3 + \neg x_5 + \neg x_6 + \neg x_7 \geq 1$ $x_1 + x_3 + \neg x_4 + x_5 + x_6 + x_7 \geq 1$
$x_2 + x_3 + \neg x_4 + \neg x_5 + \neg x_6 + x_7 \geq 1$ $x_1 + x_2 + x_3 + \neg x_4 + x_7 \geq 1$
$x_1 + x_2 + x_3 + \neg x_5 + x_7 \geq 1$ $x_1 + x_2 + x_3 + \neg x_6 + x_7 \geq 1$
$x_1 + x_2 + \neg x_4 + \neg x_5 + x_7 \geq 1$ $x_1 + x_2 + \neg x_4 + \neg x_6 + x_7 \geq 1$
$x_1 + x_2 + \neg x_5 + \neg x_6 + x_7 \geq 1$ $x_1 + x_3 + \neg x_4 + \neg x_5 + x_7 \geq 1$
$x_1 + x_3 + \neg x_4 + \neg x_6 + x_7 \geq 1$ $x_1 + x_3 + \neg x_5 + \neg x_6 + x_7 \geq 1$
$x_1 + \neg x_4 + \neg x_5 + \neg x_6 + x_7 \geq 1$ $x_2 + x_3 + \neg x_4 + \neg x_5 + x_7 \geq 1$
$x_2 + x_3 + \neg x_4 + \neg x_6 + x_7 \geq 1$ $x_2 + \neg x_4 + \neg x_5 + \neg x_6 + x_7 \geq 1$
$x_3 + \neg x_4 + \neg x_5 + \neg x_6 + x_7 \geq 1$ $x_1 + x_2 + x_3 + x_7 \geq 1$
$x_1 + x_2 + \neg x_4 + x_7 \geq 1$ $x_1 + x_2 + \neg x_5 + x_7 \geq 1$
$x_1 + x_2 + \neg x_6 + x_7 \geq 1$ $x_1 + x_3 + \neg x_4 + x_7 \geq 1$
$x_1 + x_3 + \neg x_5 + x_7 \geq 1$ $x_1 + \neg x_4 + \neg x_5 + x_7 \geq 1$
$x_1 + \neg x_4 + \neg x_6 + x_7 \geq 1$ $x_1 + \neg x_5 + \neg x_6 + x_7 \geq 1$
$x_2 + x_3 + \neg x_4 + x_7 \geq 1$ $x_2 + \neg x_4 + \neg x_5 + x_7 \geq 1$
$x_2 + \neg x_4 + \neg x_6 + x_7 \geq 1$ $x_3 + \neg x_4 + \neg x_5 + x_7 \geq 1$
$\neg x_4 + \neg x_5 + \neg x_6 + x_7 \geq 1$ $x_1 + \neg x_4 + x_7 \geq 1$

After removing the redundant constraints, 11 cardinality constraints remain, which are equivalent to the original PB constraint:

$x_7 + \neg x_4 + x_1 \geq 1$ $x_7 + \neg x_4 + \neg x_5 + x_2 \geq 1$ $x_7 + x_1 + \neg x_5 + x_2 \geq 1$
$x_7 + \neg x_4 + \neg x_5 + \neg x_6 \geq 1$ $x_7 + x_1 + \neg x_5 + \neg x_6 \geq 1$ $x_7 + \neg x_4 + x_2 + \neg x_6 \geq 1$
$x_7 + x_1 + x_2 + \neg x_6 \geq 1$ $x_7 + \neg x_4 + \neg x_5 + x_3 \geq 1$ $x_7 + x_1 + \neg x_5 + x_3 \geq 1$
$x_7 + \neg x_4 + x_2 + x_3 \geq 1$ $x_7 + x_1 + x_2 + x_3 \geq 1$

Because many cardinality constraints are being created during the translation, many redundancy checks are needed as well. Barth already pointed out that these redundancy checks may become very expensive as the set of cardinality constraints grows. He proved that each cardinality constraint only needs to be checked for redundancy against two specific other constraints. The overhead of the redundancy checks is thereby minimized to an acceptable level. The pseudo code for the translation is given in algorithm 11, which uses the method getBeta, given in algorithm 10.

The method getBeta is used to get the RHS's of the two constraints against which redundancy must be checked. Barth proved that redundancy can easily be checked just by looking at the RHS's of these two constraints. In this algorithm, a PB constraint is written as cL, where c are the coefficients and L the literals.

The method transform actually translates the PB constraint to a number of non-redundant cardinality constraints. As parameters it uses the PB constraint (which is split into two parts), the RHS and the RHS of one of the specific constraints.

To translate a PB constraint, you must call transform$(cL, \emptyset, d, 0)$.

Algorithm 10 GETBETA($cL \geq d$)

1: i := 1
2: $\beta := 0$
3: sum := 0
4: prevsum := 0
5: **while** $sum < d$ **do**
6: $cL = c_l + c'L'$
7: prevsum := sum
8: sum := sum + c_l
9: $\beta := \beta + 1$
10: $cL := c'L'$
11: **end while**
12: **while** $cL = c_l L_l + c'L'$ **do**
13: $cL := c'L'$
14: **end while**
15: **if** $prevsum \geq d - c_l$ **then**
16: $\beta'' := \beta - 1$
17: **else**
18: $\beta'' := \beta$
19: **end if**
20: **return** (β, β'')

Algorithm 11 TRANSFORM(cL, $\hat{c}\hat{L}$, D, β')

1: $S := \emptyset$
2: (β, β'') = GETBETA($cL + \hat{c}\hat{L} \geq d$)
3: **if** $\beta' - 1 \neq \beta \wedge \beta'' \neq \beta$ **then**
4: (* not redundant, so add this clause to the output set *)
5: $S := S \cup \{L + \hat{L} \geq \beta\}$
6: **end if**
7: (* As long as cL is not empty and there are valid strict reductions *)
8: **while** $c'L' + c_l L_l = cL \wedge d - c_l \geq 1$ **do**
9: $S := S \cup$ TRANSFORM($c'L'$,$\hat{c}\hat{L}$,$d - c_l$,β)
10: $\hat{c}\hat{L} := \hat{c}\hat{L} + c_l L_l$ (* forbid further reduction on L_l *)
11: $cL := c'L'$
12: **end while**
13: **return** S

Chapter 6

Benchmarks

We choose four different categories of benchmarks through which different parts of our solver can be tested. First we will create random instances with cardinality constraints. These instances will contain no structure and therefore techniques like resolution and a diff-function become very important. The variable that is used to branch on is extremely important now.

The second category will be the pigeonholes. The pigeonhole problem is a very simple problem to solve. However, for satisfiability solvers they appear to be quite difficult. Buss and Turan [5] proved that resolution proofs for the pigeonholes are exponential in size. Cardinality resolution becomes very important in solving these benchmarks.

The third category consists of several benchmarks from the SAT competition [24]. These benchmarks are mainly used to test the cardinality recognition algorithm and to compare our solver with existing SAT solvers.

The last category consists of PB benchmarks from the PB competition 2005 [25]. Since last year, there is a separate competition for PB solvers. Therefore there are also a lot of PB benchmarks available, some of which contain only cardinality constraints. With these benchmarks we can also compare the performance of our solver with the performance of existing PB solvers.

6.1 Random benchmarks

Clauses of random benchmarks are generated by randomly choosing variables. Usually the clause lengths for a given benchmark are kept constant. A random 3SAT benchmarks has for example only randomly generated clauses of length 3. The ratio between the number of variables and the number of clauses appears to be crucial for creating difficult benchmarks. If many variables and few clauses are used, the benchmarks is often easily satisfiable. On the other hand, if few variables and many clauses are used, the benchmark is often unsatisfiable. For a certain ratio, the chance of having a satisfiable

benchmark is 50%. We call this the phase transition. Benchmarks on the phase transition are generally very difficult to solve.

Finding the phase transition for random k-SAT benchmarks has been a point of interest for a long time [15]. Many researchers have been investigating the problem of finding the exact value and new articles about this problem still appear quite often. The main reason for this interest is that knowing the phase transition value makes it possible to create extremely difficult benchmarks.

To our knowledge, no-one has ever investigated the phase transitions for random cardinality benchmarks. Therefore we will try to approximate these phase transitions ourselves. We will indicate a benchmark with random constraints of length N and a RHS of M as `N,M-rand`. To approximate these phase transitions, we chose a reasonable number of variables, and created random instances with a varying number of constraints. All instances were solved and the solve times are plotted, where satisfiable and unsatisfiable instances were plotted with different colors. The resulting plots all show a similar characteristic structure, starting with a block of satisfiable dots that's slowly rising. Then a block of blue dots that starts on top of the first block and lowers slowly. See for example the plot of `7,3-rand` in figure 6.1. The other plots can be found in Appendix A. Note that this is not interesting for `N,N-1-rand`, because these benchmarks are linearly solvable. This can be explained from the fact that after translating these benchmarks to SAT, only clauses of length 2 remain, and 2-SAT is solvable in linear time [2]. The plot for `7,3-rand` is shown in figure 6.1.

We can now create a table for the phase transitions. See table 6.1. It should be noted however that these values are not extremely accurate, as we didn't perform an extensive statistical analysis. However, for our purpose it suits to use these values.

Table 6.1: Phase transitions for cardinality constraints

		right-hand side					
		1	2	3	4	5	6
length	3	4.27	-				
	4	9.93	1.36	-			
	5	21.2	2.88	0.66	-		
	6	43.4	5.57	1.32	0.40	-	
	7	87.8	10.4	2.42	0.75	0.26	-

By using these phase transitions as variable-constraint ratio, we can create difficult benchmarks that can be used for testing purposes. As the variation in difficulty for *satisfiable* instances appears to be very large, we will only use *unsatisfiable* instances on the phase transition.

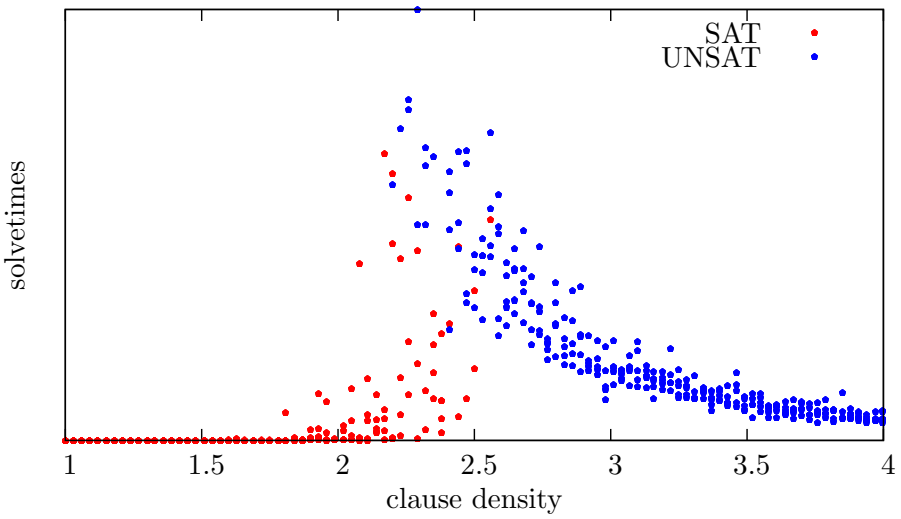

Figure 6.1: Phase transition for `7,3-rand`

6.2 Pigeonholes

The pigeonhole problem is an easy problem that says that you can't put $n+1$ pigeons into n holes if no two pigeons may share the same hole. It is easy to prove this with for example induction. However, for SAT solvers the problem often appears to be more difficult. You cannot define the concept of symmetry into a SAT problem, and therefore the SAT solver needs to try every possible combination of pigeons and holes to reach this conclusion. The problem is defined with two types of constraints. The first type says that every pigeon must be put in at least one hole. The second type says that no two pigeons can share the same hole.

6.3 Benchmarks from the SAT competition

For a long time already, almost every year, a competition for SAT solvers is organized. The performance of many state-of-the-art SAT solvers is tested on a wide range of benchmarks. Generally, SAT solvers are developing fast, and every year, benchmarks are solved that were far beyond reach in the previous year.

We think that several benchmarks that are used for the SAT competition could better be solved by a cardinality solver, because they naturally contain a considerable amount of cardinality constraints. An example of such

benchmarks is again the pigeonhole problem (see section 6.2). For this category we will use the same problems, but now we will use the CNF version. Another example is the well-known Hanoi problem. The problem consists of a tower with a number of disks on it. All disks have different sizes and every disk is smaller than the ones it lies on. The objective is to transfer the disks to one of two other towers, moving one disk at a time, and never putting a disk on a smaller one. All benchmarks can be found through SATLIB[1].

6.4 Benchmarks from the PB competition

PB solvers are developing rapidly in the last few years. Since last year there is also a separate competition for PB solvers, and therefore PB benchmarks have been created as well. The benchmarks that were used last year can be divided in optimization problems and decision problems. Because we are creating a solver for decision problems, we will use only the first category. We should note however, that it is possible to convert an optimization problem into a decision problem. An incremental or binary search can then be used to search for a solution with an optimal value for the optimization function.

In the category without an optimization function, there are many sets of both satisfiable and unsatisfiable benchmarks. There are two sets of benchmarks which already contain only cardinality constraints and CNF clauses. These sets are suitable for our solver to test with. The other sets all contain PB constraints (and CNF clauses). In chapter 5 we explained a way to translate these PB constraints to cardinality constraints. The two sets mentioned above contain benchmarks that were created by Fadi Aloul. They are about FPGA (field programmable gate array) switch-boxes. One set (fpga) contains 36 satisfiable instances, and the other set (chnl) contains 21 unsatisfiable instances[2].

After using the translation algorithm on the PB benchmarks from the competition, it appeared that the translation resulted in most cases in an extremely increased number of constraints. Even though most benchmarks contain less than 10% PB constraints, the translation resulted in a total number of constraints which was between 2 and 200 times higher. It can hardly be expected that our solver will be competitive in such cases. It is interesting to note that after translating the PB constraints to cardinality constraints, all resulting constraints had a RHS equal to 1, so the translation resulted only in CNF clauses. To be able to compare our solver with PB solvers, we will use the two sets with a minimum increase in cardinality constraints. These sets are 'elf.rf' and 'ooo.rf'. Both sets contain 5 unsatisfiable benchmarks.

[1] All SATLIB benchmarks can be found through: `http://www.satlib.org`

[2] An explanation of the encoding for these instances can be found here: `http://www.eecs.umich.edu/~faloul/benchmarks.html`

6.5 Statistics

This section gives some statistics about the benchmarks that were chosen in this chapter.

Table 6.2: Statistics about the pigeonhole benchmarks

	CNF		Cardinality			
	#clauses	#lits	#constr	#lits	#N,1-CARD (length)	#N,N-1-CARD (length)
hole 6	133	294	13	84	7 (6)	6 (7)
hole 7	204	448	15	112	8 (7)	7 (8)
hole 8	297	648	17	144	9 (8)	8 (9)
hole 9	415	900	19	180	10 (9)	9 (10)
hole 10	561	1210	21	229	11 (10)	10 (11)

Table 6.3: Statistics about the Hanoi benchmarks

	CNF		Cardinality			
	#clauses	#lits	#constr	#lits	#N,1-CARD	#N,N-1-CARD
hanoi 4	4934	9432	1953	6100	1665	258
hanoi 5	14468	23404	5755	18552	5049	673

Table 6.4: Statistics about the longmult benchmarks

	CNF			Cardinality			
	#clauses	#vars	#lits	#constr	#lits	#N,1-CARD	#N,N-1-CARD
longmult0	1206	437	2872				
longmult1	2335	791	3689	1608	3685	3681	4
longmult2	3525	1164	6334	2701	6318	2693	8
longmult3	4767	1555	9099	3849	9075	3837	12
longmult4	6069	1966	12004	5057	11972	5041	16
longmult5	7431	2397	15049	6325	15009	6305	20
longmult6	8853	2848	18234	7653	18186	7629	24
longmult7	10335	3319	21559	9041	21503	9013	28
longmult8	11877	3810	25024	10489	24960	10457	32
longmult10	15141	4852	32374	13565	32294	13525	40
longmult12	18645	5974	40284	16881	40188	16833	48

Table 6.5: Statistics about the chnl benchmarks

	#constr	#vars	#lits	#N,1-CARD (length)	#N,N-1-CARD (length)
chnl10-11	42	220	440	22 (10)	20 (11)
chnl10-15	50	300	600	30 (10)	20 (15)
chnl10-20	60	400	800	40 (10)	20 (20)
chnl15-16	62	480	960	32 (15)	30 (16)
chnl15-20	70	600	1200	40 (15)	30 (20)
chnl15-25	80	750	1500	50 (15)	30 (25)
chnl20-21	82	840	1680	42 (20)	40 (21)
chnl20-25	90	1000	2000	50 (20)	40 (25)
chnl20-30	100	1200	2400	60 (20)	40 (30)
chnl30-31	122	1860	3720	62 (30)	60 (31)
chnl30-35	130	2100	4200	70 (30)	60 (35)
chnl30-40	140	2400	4800	80 (30)	60 (40)
chnl35-36	142	2520	5040	72 (35)	70 (36)
chnl35-40	150	2800	5600	80 (35)	70 (40)
chnl35-45	160	3150	6300	90 (35)	70 (45)
chnl40-41	162	3280	6560	82 (40)	80 (41)
chnl40-45	170	3600	7200	90 (40)	80 (45)
chnl40-50	180	4000	8000	100 (40)	80 (50)
chnl50-51	202	5100	10200	102 (50)	100 (51)
chnl50-55	210	5500	11000	110 (50)	100 (55)
chnl50-60	220	6000	12000	120 (50)	100 (60)

Table 6.6: Statistics about the fpga benchmarks

	#constr	#vars	#lits	#N,1-CARD	#N,N-1-CARD
fpga10-8	106	120	600	88	18
fpga10-9	118	135	720	99	19
fpga10-10	130	150	850	110	20
fpga11-9	128	149	797	108	20
fpga11-10	141	165	935	120	21
fpga11-11	154	182	1095	132	22
fpga12-10	152	180	1020	130	22
fpga12-11	166	198	1188	143	23
fpga12-12	180	216	1368	156	24
fpga13-11	178	215	1293	154	24
fpga13-12	193	234	1482	168	25
fpga13-13	208	254	1697	182	26
fpga14-12	206	252	1596	180	26
fpga14-13	222	273	1820	195	27
fpga14-14	238	294	2058	210	28
fpga15-13	236	293	1957	208	28
fpga15-14	253	315	2205	224	29
fpga15-15	270	338	2483	240	30
fpga20-18	416	540	4500	378	38
fpga20-19	438	570	4940	399	39
fpga20-20	460	600	5400	420	40
fpga25-23	646	863	8637	598	48
fpga25-24	673	900	9300	624	49
fpga25-25	700	938	10013	650	50
fpga30-28	926	1260	14700	868	58
fpga30-29	958	1305	15660	899	59
fpga30-30	990	1350	16650	930	60
fpga35-33	1256	1733	23117	1188	68
fpga35-34	1293	1785	24395	1224	69
fpga35-35	1330	1838	25743	1250	70
fpga40-38	1636	2280	34200	1558	78
fpga40-39	1678	2340	35880	1599	79
fpga40-40	1720	2400	37600	1640	80
fpga45-43	2066	2903	48397	1978	88
fpga45-44	2113	2970	50490	2024	89
fpga45-45	2160	3038	52673	2070	90

Table 6.7: Statistics about the ooo.rf benchmarks

	#constr	#vars	#lits	#N,1-CARD	#N,N-1-CARD
ooo.rf6	8981	1804	32085	8981	0
ooo.rf7	22925	3736	97800	22925	0
ooo.rf8	36844	6832	147438	36844	0
ooo.rf9	56152	11476	212327	56152	0
ooo.rf10	87750	18069	332236	87750	0

Table 6.8: Statistics about the elf.rf benchmarks

	#constr	#vars	#lits	#N,1-CARD	#N,N-1-CARD
elf.rf6	226	67	682	226	0
elf.rf7	8363	1328	35415	8363	0
elf.rf8	44502	6059	198144	44502	0
elf.rf9	173644	20445	895682	173644	0
elf.rf10	460322	55066	2419130	460322	0

Chapter 7

The diff-function

The lookahead procedure must decide which variable will be instantiated next. To make a decision, a lookahead is performed on each variable. With the *diff-function*, a value can be given to the usefulness of setting a variable to `true` or `false`. This way, each variable will be given two values. If there are no variables which always result in a conflict, a new branch variable must be chosen. This is done by comparing the diff-values for all variables. The variable with the highest diff-values is chosen as branch variable.

The challenge is to find an effective formula for the diff-function. The diff-function can depend for example on the number of variables that are valued during the lookahead. Another option is to count the constraints that are handled. This last option can also be extended by giving different types of constraints different weights. Because (to our knowledge) lookahead solvers have never been constructed for cardinality constraints, it is not known what approaches will generally lead to the best results. We can only take SAT solvers and PB solvers as an example, or come up with something new.

To test different strategies, we need a number of difficult benchmarks with as little structure as possible. For this purpose we will use random benchmarks as explained in section 6.1.

7.1 Variables vs. clauses

There are two potentially interesting strategies to decide the usefulness of a variable in the lookahead. The first strategy is to use the number of *implied* variables as an indicator. The second strategy is to use the number of reduced constraints.

Implied variables are variables that are forced to a certain value, after having set another variable. For example, if the formula contains the constraint $a + b + c \geq 2$, and in the lookahead you set a to `false`, then both b and c are forced to `true`. In the lookahead we set only one variable, but be-

cause of unit propagation, multiple variables are implied. It can be expected that when the number of implied variables is high, the lookahead variable is a good branching variable.

It can also be expected that variables which lead to many reduced constraints are good branching variables. If the formula becomes smaller, then every next step can be performed faster.

7.1.1 Implied variables vs. reduced constraints

To test the importance of both strategies, we will use the 3,1-rand benchmarks. In the lookahead there will be only constraints of type 3,1-CARD and they are all reduced to constraints of type 2,1-CARD, therefore we do not need to differentiate between the types of reduced constraints. A variable x will be used to indicate the importance of the number of reduced constraints. It is variated on the interval $[0 : 1]$ with steps of 0.02. When $x = 0$, this means that only the number of implied variables is used as an indication. When $x = 1$, this means that only the number of reduced constraints is used as an indication. To get a good view of the changing solve times, we will use 100 unsatisfiable instances of `3,1-rand`. For each value of x the total solve time will be used. Figure 7.1 shows the resulting graph.

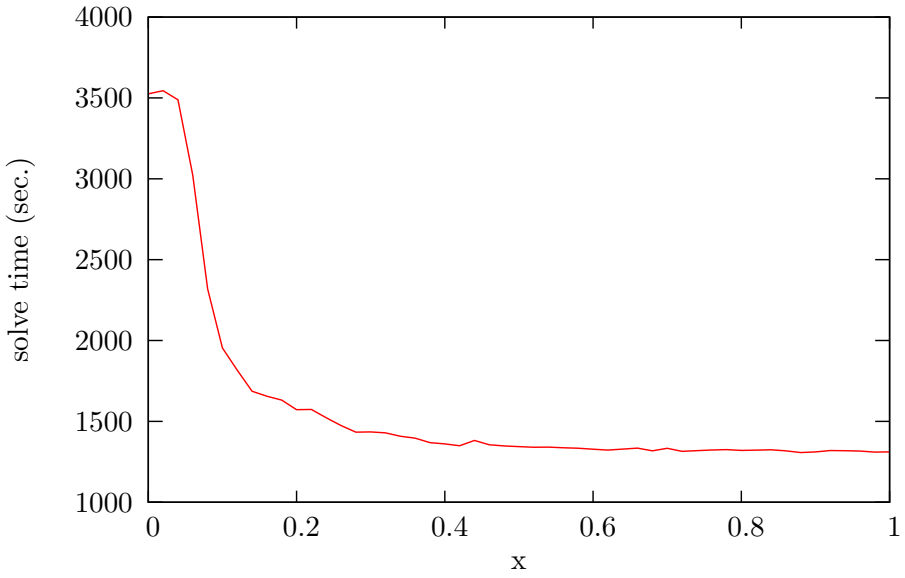

Figure 7.1: Solve times for experiment 'implied variables vs. reduced constraints'

The graph shows that the number of implied variables is a very bad indicator, as this leads to a combined solve time of more than 3500 seconds.

Table 7.1: Clause weights

clause length	weight
2	1.0
3	0.2
4	0.05
5	0.01
6	0.003
7	0.00049
8	0.00011

The more weight you give to the number of reduced constraints as indicator, the lower the combined solve time becomes. Though there is no clear optimum, it seems highly unlikely that the solve times will decrease further if x is given values higher than 1 (such that the number of implied variables is used negatively). For this reason we will use only the number of reduced constraints as an indication to base the branching variable on.

7.2 Deciding on the weights

After having performed the first experiment, we decided to base the lookahead diff-function on the number of reduced constraints. As we expect that it is more important to have many short constraints in the lookahead then it is to have long constraints, we want to give different weights to the different constraints. This conclusion was also drawn by Oliver Kullmann [16]. He showed that CNF clauses get rapidly less important when they become longer.

Let's say γ_i gives the weight for the reduction from a clause of length $i+1$ to a clause of length i. Kullmann says the optimal weights are then calculated as follows:

$$\gamma_2 = 1,\ \gamma_3 = 0.2,\ \gamma_4 = 0.05,\ \gamma_5 = 0.01,\ \gamma_6 = 0.003$$
$$\gamma_k = 20.4514 \cdot 0.218673^k \quad \text{for} \quad k \geq 7$$

The value for each next weight is roughly 5 times lower. The first few clause weights are shown in table 7.1.

The reason that short clauses are important, is most likely that they are almost unit. As soon as a clause becomes unit, you can eliminate a variable from the formula, which is very important when trying to decrease the problem.

For cardinality constraints, deciding the weights becomes a two-dimensional problem, because we don't just have to deal with the length of the constraint, but also with the effect of different RHS's. We need to find the

optimal weights to fill in table 7.2 and also extract a rule from it to estimate the weights for longer constraints, and those with a higher RHS. In this table, weights are given to the resulting constraint, so a weight for `N,M-CARD` actually means a reduction from `N+1,M-CARD` to `N,M-CARD`.

The absolute value of the weights is not important. Only the relative values matter. Therefore you can always start by fixing one weight.

Table 7.2: Initial constraint weights

		right-hand side				
		1	2	3	4	5
length	2	1.0				
	3	·	·			
	4	·	·	·		
	5	·	·	·	·	
	6	·	·	·	·	·

7.2.1 Finding the optimal weights

Now we need to fill in the table with optimal weights. The idea is to choose benchmarks, such that every time only one new weight will be investigated. We will variate this weight and see what happens to the solve times. Afterwards we can decide the optimal value for that weight. This can be done for all weights. Two directions in the table are particularly interesting. First we need to find out whether the constraints are more important if they become shorter. Second, we need to find out whether the constraints are more important if the RHS is larger.

Reductions from 4,1CARD to 3,1CARD

We start with the decision of the optimal value for the weight of `3,1-CARD` constraints. These are ordinary CNF constraints, and therefore we expect the value to be 0.2, like the one Kullmann found to be optimal. We calculate this optimal value to be sure that the weights aren't different because we use another solver. We will use 100 `4,1-rand` benchmarks. In the lookahead procedure, `4,1-CARD` constraints will reduce to `3,1-CARD` constraints, and they can reduce further to `2,1-CARD` constraints. This way only two weights are used, of which one was already fixed to 1.0. Only unsatisfiable instances are used, because solve times for satisfiable instances variate far too much and won't lead to an accurate mean solve time. The weight will be variated on the interval $[0.04 : 1]$. Until 0.3 the steps are 0.02, and around 0.2 steps of 0.01 are used, because this is were we expect to find the optimal value.

Table 7.3: First weight

		right-hand side			
		1	2	3	4
length	2	1.0			
	3	0.21			
	4				

For each weight all benchmarks are solved and the total solve time is used as a measure to decide the optimal weight.

The resulting solve times are displayed in the figure 7.2.

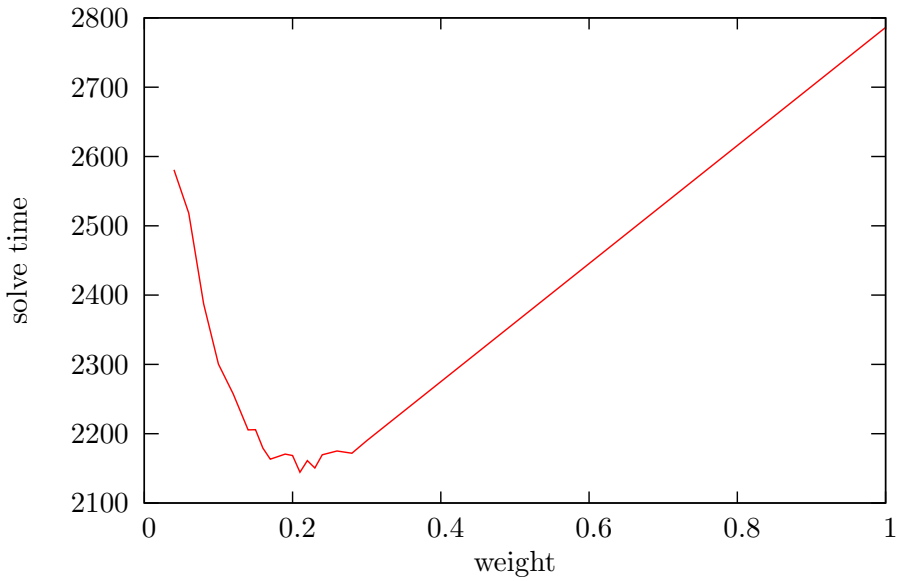

Figure 7.2: Solve times for experiment 'Reductions from `4,1-CARD` to `3,1-CARD`'

Around weight 0.2 the graph is not very smooth, but this is probably because of the smaller steps that were used. The global differences are clear. A very low weight (< 0.1) leads to bad performance. As the weight increases towards 1, performance is also going down. 0.21 seems to be optimal.

Reductions from 4,2-CARD to 3,2-CARD

Now, we want to find the optimal value for the weight of reduction from `4,2-CARD` to `3,2-CARD` constraints. Therefore we will use 100 unsatisfiable `4,2-rand` benchmarks. In the lookahead procedure, `4,2-CARD` constraints

Table 7.4: Second weight

		right-hand side			
		1	2	3	4
length	2	1.0			
	3	0.21	2.1		
	4				

will reduce to `3,2-CARD` constraints. When literals are satisfied in the lookahead, `3,1-CARD` constraints are created, which will reduce to `2,1-CARD` constraints. Therefore, only two weights will be used, of which one was already fixed to 1.0. The weight will be variated on the interval $[0.3 : 22]$. Until 4.0 steps of 0.1 are used, and steps of 1.0 until 22.
The resulting solve times are displayed in figure 7.3.

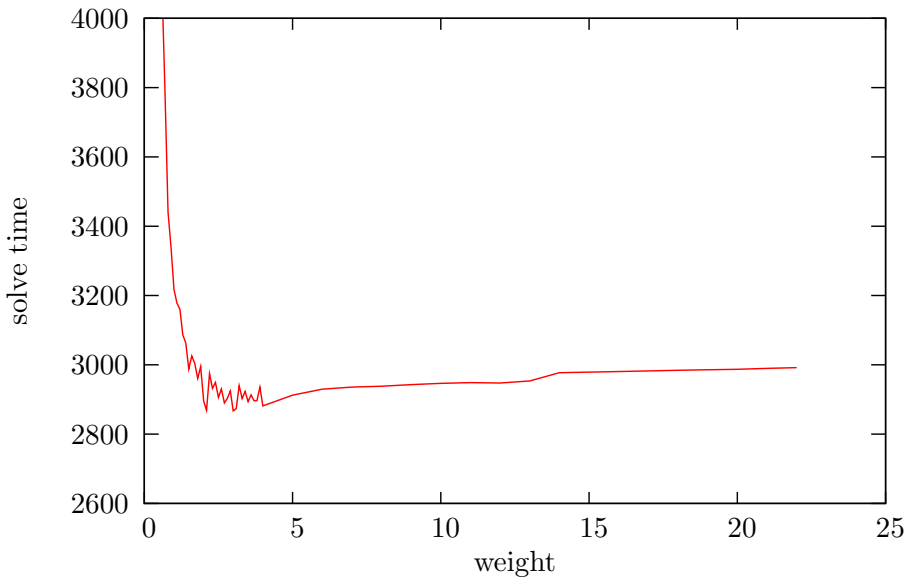

Figure 7.3: Solve times for experiment 'Reductions from `4,2-CARD` to `3,2-CARD`'

Looking at this graph, it is difficult to give an optimal value for reductions to `3,2-CARD` constraints. It seems that the optimal value is somewhere between 2.0 and 3.0, but the differences in solve times are small. Also when you keep increasing the weight, solve times don't go up significantly. For now, we will use 2.1 as the optimal value, but in the next experiment, we will try both 2.1 and 20 as optimal values, and see whether this will have any influence on the performance when deciding the next weight.

Reductions from 5,3-CARD to 4,3-CARD

Now, we are going to look at the optimal value for reductions from `5,3-CARD` to `4,3-CARD` constraints. As discussed in the previous experiment, we will do this both with a weight of 2.1 and with a weight of 20 for the reductions to `3,2-CARD` constraints. 100 unsatisfiable `5,3-rand` benchmarks will be used. In the case that weight 2.1 will be used for `3,2-CARD`, the value for `4,3-CARD` are chosen on the interval $[2.1 : 7]$, using steps of 0.1. In the case that 20 is used, the interval $[16 : 36]$ is used, with steps of 2.
The resulting solve times (when 2.1 is used as reduction weight for `3,2-CARD`) are shown in figure 7.4. The solve times (when 20 is used as reduction weight for `3,2-CARD`) are shown in figure 7.5.

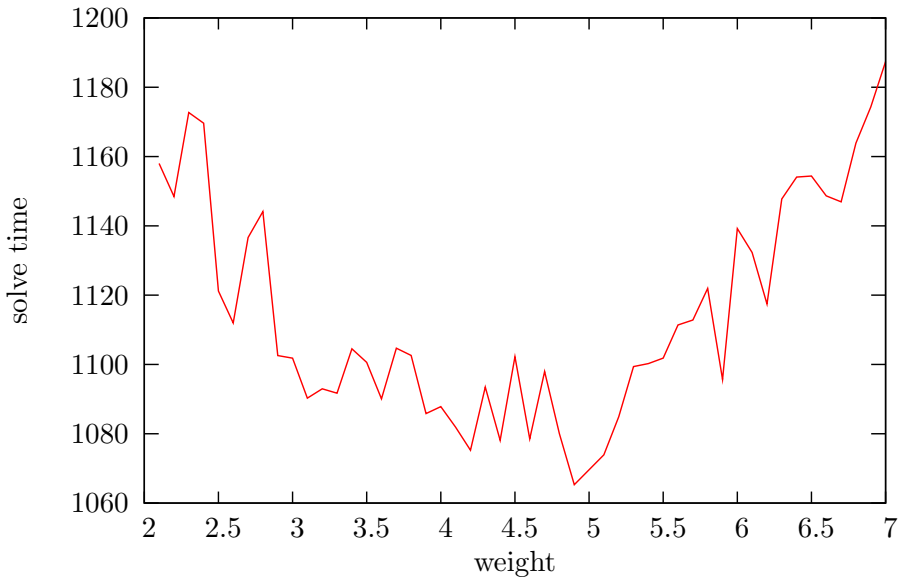

Figure 7.4: Solve times for first experiment 'Reductions from `5,3-CARD` to `4,3-CARD`'

In the first graph you can see that solve times get as low as 1065 seconds (with a weight of 4.9). In the second graph the lowest point is 1075 seconds. This means that it makes almost no difference whether you use 2.1 as optimal value for reductions to `3,2-CARD` constraints, or 20. This makes it very difficult to empirically decide what weights are best. Perhaps, five of six weights further these small differences do have significant consequences. This gives us the idea to search for a logical *rule* to fill the table, rather then deciding all optimal weights empirically.

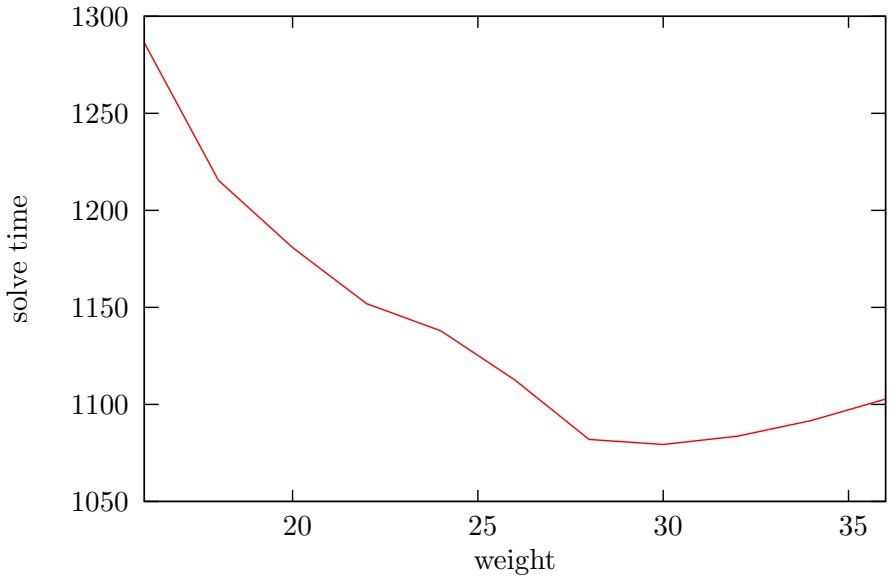

Figure 7.5: Solve times for second experiment 'Reductions from `5,3-CARD` to `4,3-CARD`'

Table 7.5: Third weight

		right-hand side			
		1	2	3	4
length	2	1.0			
	3	0.21	2.1		
	4			4.9	

7.2.2 Finding an optimal rule

We need to find a rule through which we can fill in the weights table. It seems logical to base such a rule on the translation to CNF clauses. We already have Kullmann's reduction weights for CNF clauses. We don't know however, whether these weights are based only on the resulting constraint, or also on the constraint that was *removed*. This is an important distinction, because cardinality reductions don't result in an equal number of CNF clauses being removed and created (see table 4.1 for these numbers. We will therefore try two possible rules to fill in the table. The first rule assumes that the weights are based on both removing one clause and creating another clause. The second rule will assume that only the resulting clause is important.

Rule 1

Kullmann has already given optimal values for reductions from each CNF clause of length $n+1$ to a CNF clause of length n. As we know that each cardinality constraint can be translated to a number of CNF clauses, the weight for a reduction from a given `N,M-CARD` can be calculated as the weight of the corresponding CNF clauses that are created, minus the weight of the corresponding CNF clauses that are removed.

We assume that the weights that Kullmann gave are weights for both removing and creating a CNF clause, so we need to find out the weights for only adding a CNF clause. Let's say a `2,1-CARD` constraint is called γ_2^*, a `3,1-CARD` constraint γ_3^*, a 4,1-CARD constraint γ_4^* and so on. The weight for a reduction from `3,1-CARD` to `2,1-CARD` was fixed to 1.0. This is therefore $\gamma_2^* - \gamma_3^*$. For the same reason, $\gamma_3^* - \gamma_4^*$ is 0.2 and $\gamma_4^* - \gamma_5^*$ is 0.05 and so on. The value for γ_2^* can now be calculated by taking the sum of all reduction weights, because this results in $\gamma_2^* - \gamma_3^* + \gamma_3^* - \gamma_4^* + \gamma_4^* - \gamma_5^* + \gamma_5^* - \gamma_6^* \ldots = \gamma_2^*$. The value for each next clause can be calculated by taking the same sum, but leaving out the first element. Now we know the values for each clause, rather that the value for a reduction:

γ^*	value
γ_2^*	1.26362
γ_3^*	0.26362
γ_4^*	0.06362
γ_5^*	0.01362
γ_6^*	0.00362
γ_7^*	0.00062
γ_8^*	0.00013
$\vdots$	$\vdots$

The last thing we need to know is the number of CNF clauses that are needed for each cardinality constraint. According to the translation that was explained in section 4.2 a cardinality constraint `N,M-CARD` results in all combinations of $N - M + 1$ literals. This comprises a number of $\binom{N}{M-1}$ clauses. Table 7.6 shows these numbers.

Now we can calculate the weight for a reduction from `N+1-M-CARD` to `N,M-CARD` as:

$$\binom{N}{M-1} \cdot \gamma^*_{(N-M+1)} - \binom{N+1}{M-1} \cdot \gamma^*_{(N-M+2)}$$

In table 7.7 the reduction weights are filled in according to this rule.

Table 7.6: Required number of CNF clauses per cardinality constraint

		right-hand side						
		1	2	3	4	5	6	
length	2	1						
	3	1	3					
	4	1	4	6				
	5	1	5	10	10			
	6	1	6	15	20	15		
	7	1	7	21	35	35	21	
	⋮	⋮					⋮	⋱

Table 7.7: Reduction weights for rule 1

		right-hand side					
		1	2	3	4	5	
length	2	1.0					
	3	0.2	2.736				
	4	0.05	0.736	4.946			
	5	0.01	0.236	1.682	7.364		
	6	0.003	0.056	0.668	3.046	9.637	
	⋮	⋮				⋮	⋱

Rule 2

The second rule is also based on the translation to CNF. However, in this case we assume that only the resulting constraint is important, and that the *removed* constraint has no weight. If a cardinality constraint is equivalent to N CNF clauses, then the idea is to give a reduction to this cardinality constraint the same weight as the sum of all reductions to those N CNF clauses. For example, when reducing the cardinality constraint `6,3-CARD` to `5,3-CARD`, we create 10 new CNF clauses of length 3, so this reduction is valued $10 * 0.2 = 2.0$.

Now we can calculate the weight for a reduction from `N+1,M-CARD` to `N,M-CARD` as:

$$\binom{N}{M-1} \cdot (\gamma^*_{(N-M+1)} - \gamma(N-M+2)^*)$$

In table 7.8 the reduction weights are filled in according to this rule.

Table 7.8: Reduction weights for rule 2

		right-hand side					
		1	2	3	4	5	
length	2	1.0					
	3	0.2	3.0				
	4	0.05	0.8	6.0			
	5	0.01	0.25	2.0	10.0		
	6	0.003	0.06	0.75	4.0	15.0	
	⋮	⋮				⋮	⋱

Results

These two rules will be compared with the case in which no weights are used. To test the different rules, we will use random benchmarks of length 7 on the phase transition, where we create different sets for different RHS's. For each set 100 unsatisfiable benchmarks are used and the total of the solve times is used as measure. Table 7.9 shows the results.

Table 7.9: Results of test with diff-function

	`7,1-rand`	`7,2-rand`	`7,3-rand`	`7,4-rand`	`7,5-rand`
Rule 1	**2354.91**	1181.73	1505.83	**3534.52**	**3163.54**
Rule 2	2355.66	**1163.36**	**1505.82**	3598.48	3352.90
No weights	2913.14	2398.87	1507.14	7273.16	4758.34

From these results it is clear that both rules lead to better performance than the situation where no weights are used, although it is odd that for `7,3-rand` there seems to be no difference. For `7,1-rand` there is of course no difference between the two rules, because the same weights were used. For `7,2-rand` rule 2 performs about 2% better than rule 1. The first rule however seems to be the best for both `7,4-rand` and `7,5-rand`, where the differences are 2% and 6%. This is also where the largest differences are in the tables with the weights. For the first rule, the weights on the diagonal are growing slightly faster than linear. For the second rule, these weights are growing exponentially. This is clearly too fast. Based on these results, we choose the first rule for our solver. However, we must keep in mind that the differences are small, and that there's probably a rule that is far better than the two that we have tried.

Chapter 8

Cardinality resolution

In logic, the resolution rule is a rule of inference that takes two clauses and produces a new clause, the resolvent, that is implied by them [26]. In propositional logic you need two clauses with complementary literals. The resolvent can be constructed by taking together all literals from both clauses, except for the complementary literals. For example, the clauses

$$a + b + c \geq 1$$
$$\neg a + d + e \geq 1$$

can be used to form this resolvent:

$$b + c + d + e \geq 1$$

If, during solving, you created a partial assignment where b, c and d are falsified, then this resolvent results in also setting e to satisfy this clause. That this is necessary can also be seen from the original two clauses, but only after setting a and finding out that the second clause has only one literal left. Adding a resolvent therefore saves you one step.

Resolution on cardinality constraints is also possible and it is almost as easy. Resolving two cardinality constraints is done by adding them. The constraints

$$a + b + c+ \geq 2$$
$$\neg a + d + e \geq 2$$

lead to this resolvent:

$$a + \neg a + b + c + d + e \geq 4$$

which, after simplifying, results in:

$$b + c + d + e \geq 3$$

An important advantage is that you can also perform resolution on multiple complementary literals. This is theoretically also possible with CNF

clauses, but it will always result in tautological resolvents. Compare the following two situations:

$$\begin{array}{l} a+b+c \geq 1 \\ \neg a+\neg b+d \geq 1 \\ \hline c+d \geq 0 \end{array} \qquad \begin{array}{l} a+b+c \geq 2 \\ \neg a+\neg b+d \geq 2 \\ \hline c+d \geq 2 \end{array}$$

When using two CNF clauses (as in the left situation), the resolvent is trivially satisfied, as any assignment will result in a value larger than or equal to 0. However, when using two cardinality constraints with RHS's larger than 1 (as in the right situation), the resolvent becomes very useful. You can immediately set two variables.

Generally resolvents can help speed up the process of deciding satisfiability. However, not all resolvents are always useful. If you add all possible resolvents to a formula, then the number of constraints can grow exponentially, resulting in much overhead needed to manage the constraint database. Therefore we need some kind of rule to add only those resolvents that we expect to be beneficial. We will try three different strategies for adding resolvents and test their performance on several benchmarks.

8.1 Least occurrence resolution

To avoid the problem that adding all resolvents may result in a massive increase of the number of constraints, we decided to *stamp* constraints after they have been used for resolution. For example after performing resolution on all constraints containing the variable x_1 (either positive or negative), we will stamp these constraints and not use them again for resolution on other variables.

However, we can only guarantee that such a method terminates if in every resolution step more constraints are stamped than the number of resolvents that are added. Table 8.1 shows the relative number of constraints that are added or stamped, depending on the number of constraints in which a variable appears positively or negatively. For example, if a variable x_1 appears positively in 4 constraints, and negatively in 2 constraints, then this will result in 8 new resolvents. while only 6 constaints are stamped. This results in an increase of 2 of the number of *unstamped* constraints.

When a variable appears either only once positive or only once negative, less resolvents are added than stamped. In these cases we can guarantee that the number of unstamped constraints converges to 0 and that the process of adding new resolvents will end in a reasonable time.

Therefore, for each variable we will count the number of positive and negative occurrences before performing resolution. We will start with the variable that appears least, because else it is possible that many constraints

Table 8.1: Relative number of added constraints after resolution

		# positive			
		1	2	3	4
# negative	1	-1	-1	-1	-1
	2	-1	0	1	2
	3	-1	1	3	5
	4	-1	2	5	8
	5	-1	3	7	11

are stamped, through which resolution possibilities with few constraints are maybe not possible anymore. Look at the following example:

$$a + b + c \geq 1$$
$$\neg a + d + e \geq 1$$
$$\neg c + f + g \geq 1$$
$$\neg c + h + i \geq 1$$

If we wouldn't start with the least occurring variable a, but with c, the only two resolvents can be added:

$$a + b + f + g \geq 1$$
$$a + b + h + i \geq 1$$

However, if we do start with the least occurring variable, after adding the resolvent for a, we can also still add the resolvents for c:

$$b + c + d + e \geq 1$$
$$b + d + e + f + g \geq 1$$
$$b + d + e + h + i \geq 1$$

.
We call this type of resolution *least occurrence resolution*.

After some tests, it appeared that it was not always beneficial to add resolvents for a variable that occurs either positively in many constraints, or negatively in many constraints. Therefore we decided to test by what number of occurrences the results were still positive. Table 8.2 shows the results of this test.

The threshold gives the number of constraints in which the variable appears either positive or negative (where the complement appears in exactly one constraint). Only if a variable appears once positive and a certain number of times (less than the threshold) negative, resolution will be performed on this variable. `NONE` means that no resolution is performed at all.

The differences are largest for the pigeonholes. It seems to matter considerably whether resolvents are added or not. The reason why this is so, is

Table 8.2: Results of test with least-occurrence resolution

Threshold	benchmarks	# total	# solved	# resolvents	solve time
NONE	pigeonholes (2-30)	29	9	-	148.83
	longmult (0-7)	8	8	-	**532.26**
	4,2-rand	10	10	-	**218.72**
	5,2-rand	10	10	-	**19.42**
	6,2-rand	10	10	-	**23.87**
	7,2-rand	10	10	-	**118.39**
	hanoi4	1	1	-	**75.00**
1	pigeonholes (2-30)	29	29	899	**447.38**
	longmult (0-7)	8	8	154	535.24
	4,2-rand	10	10	138	223.15
	5,2-rand	10	10	0	**19.42**
	6,2-rand	10	10	0	**23.87**
	7,2-rand	10	10	0	**118.39**
	hanoi4	1	1	0	**75.00**
2	pigeonholes (2-30)	29	29	899	**447.38**
	longmult (0-7)	8	8	672	537.54
	4,2-rand	10	10	790	248.77
	5,2-rand	10	10	0	**19.42**
	6,2-rand	10	10	0	**23.87**
	7,2-rand	10	10	0	**118.39**
	hanoi4	1	1	0	**75.00**
100	pigeonholes (2-30)	29	29	899	**447.38**
	longmult (0-7)	8	8	1008	541.17
	4,2-rand	10	10	5896	648.77
	5,2-rand	10	10	63	19.98
	6,2-rand	10	10	0	**23.87**
	7,2-rand	10	10	0	**118.39**
	hanoi4	1	1	122	83.22

explained in section 8.1.1. In case of the `4,2-rand` benchmarks, solvetimes go a little up when more resolvents are added. Using a threshold of 100 seems to have negative results on especially `4,2-rand` and hanoi. It seems that a threshold of 1 has the best overall performance.

The reason why a threshold of 1 appears to be best, is probably because in this situation you can often remove the source constraints, because of redundancy. See theorem 8.1.1.

Theorem 8.1.1. *Suppose* $a + x_1 + x_2 + \ldots + x_n \geq n$ *and* $\neg a + y_1 + y_2 + \ldots + y_m \geq m$ *are two* `N,N-1-CARD` *constraints with resolvent* $x_1 + \ldots + x_n + y_1 + \ldots + y_m \geq n + m - 1$. *If variable a doesn't appear anywhere else in the formula, then after adding the resolvent, both source constraints can be removed.*

To prove this theorem, we need to show that both source constraints are redundant. Therefore we must make sure that both source constraints are satisfiable when the resolvent is satisfiable. Because the resolvent is also a `N,N-1-CARD` constraint, it is only satisfied when either all $n + m$ literals are satisfied, or when at most one of its literals is falsified. In the first case all x_i and y_i are satisfied, resulting in n satisfied literals in the first constraints and m satisfied literals in the second constraint, satisfying both. In the latter case either one x_i is falsified, or one y_i is satisfied. If the first situation, variable a must be set to `true` to satisfy the first constraint. In the latter situation, variable a must be set to `false` to satisfy the second constraint. Now, we know in all situations that if the resolvent is satisfiable, that in all cases also the source constraints are satisfiable, which makes them redundant. □

Because of this we choose to perform least occurrence resolution only when a variable appears exactly once positive and once negative. In section 8.1.1 it is explained why this works so well on the pigeonhole problems.

8.1.1 Least occurrence resolution on pigeonholes

In this section we will explain why the least occurrence resolution works so well on the pigeonhole problems. As an example we will use pigeonhole 3, but the same reasoning applies on any other pigeonhole. The question is whether you can put $3 + 1$ pigeons in 3 holes, such that every pigeon gets its own hole. One variable will be used for every hole-pigeon combination. Variable $x_{i,j}$ means that pigeon i is put in hole j. If a variable is true, this means that the corresponding hole-pigeon combination applies. There are two types of constraints: One type of constraints says that every pigeon must be put in at least one hole, and the other type says that at most one pigeon can be put in a hole. The cardinality constraints are displayed below. The second type of constraints have been adjusted to normal form.

1. $x_{1,1} + x_{1,2} + x_{1,3} \geq 1$
2. $x_{2,1} + x_{2,2} + x_{2,3} \geq 1$
3. $x_{3,1} + x_{3,2} + x_{3,3} \geq 1$
4. $x_{4,1} + x_{4,2} + x_{4,3} \geq 1$
5. $\neg x_{1,1} + \neg x_{2,1} + \neg x_{3,1} + \neg x_{4,1} \geq 3$
6. $\neg x_{1,2} + \neg x_{2,2} + \neg x_{3,2} + \neg x_{4,2} \geq 3$
7. $\neg x_{1,3} + \neg x_{2,3} + \neg x_{3,3} + \neg x_{4,3} \geq 3$

Now we will show each step of the least occurrence resolution algorithm. In each step we give the variable that appears least `min_var`, the resolvent and a list of the constraints that were used for resolution and have been stamped.

Step 1:	`min_var`:		$x_{1,1}$
	resolvent:	8.	$x_{1,2} + x_{1,3} + \neg x_{2,1} + \neg x_{3,1} + \neg x_{4,1} \geq 3$
	stamped:		1, 5
Step 2:	`min_var`:		$x_{1,2}$
	resolvent:	9.	$x_{1,3} + \neg x_{2,1} + \neg x_{2,2} + \neg x_{3,1} + \neg x_{3,2} + \neg x_{4,1} + \neg x_{4,2} \geq 5$
	stamped:		6, 8
Step 3:	`min_var`:		$x_{1,3}$
	resolvent:	10.	$\neg x_{2,1} + \neg x_{2,2} + \neg x_{2,3} + \neg x_{3,1} + \neg x_{3,2} + \neg x_{3,3} + \neg x_{4,1} + \neg x_{4,2} + \neg x_{4,3} \geq 7$
	stamped:		7, 9
Step 4:	`min_var`:		$x_{2,1}$
	resolvent:	11.	$\neg x_{3,1} + \neg x_{3,2} + \neg x_{3,3} + \neg x_{4,1} + \neg x_{4,2} + \neg x_{4,3} \geq 5$
	stamped:		3, 10
Step 5:	`min_var`:		$x_{3,1}$
	resolvent:	12.	$\neg x_{4,1} + \neg x_{4,2} + \neg x_{4,3} \geq 3$
	stamped:		4, 11
Step 6:	`min_var`:		$x_{4,1}$
	resolvent:	13.	$0 \geq 1$
	stamped:		4, 12

Initially every variable appears once positive and once negative in the formula. After adding resolvents, variables will never appear twice or more, because source constraints are stamped and not counted anymore. In every step a resolvent is created where the previous resolvent is used as one of the source constraints. This way the new resolvents become longer, as new literals are added. In step 3 the resolvent contains all active literals and can't become any longer. Now resolution is performed together with the original constraints. In these resolution steps every time 3 literals are removed, while

the RHS decreases only 2. In the last step this results in a conflict, because of an empty resolvent.

The same reasoning can be used for larger pigeonhole problems. n steps are needed to create a resolvent of length n^2, and then n steps are needed to reduce the resolvents until a conflict occurs. See also table 8.3. Recognizing a variable for which least-occurrence resolution is possible, is not difficult. In our solver, we already keep lists of clauses in which a literal appears. Therefore we can just check the number of occurrences for each variable, until we find a variable which appears exactly once positive and once negative. In the worst case, this will take n steps.

Table 8.3: Number of required steps and resolvents for pigeonholes

pigeonhole	# resolvents	max res. length
n=2	4	4
3	6	9
4	8	16
5	10	25
6	12	36
7	14	49
8	16	64
9	18	81
10	20	100

8.2 Stringency resolution

Some solvers just add any possible resolvent and add a limit if the constraint database is growing too fast. See for example Satz [23] which adds all resolvents until a certain limit, and after that only the short resolvents. We tried a similar approach. However, we don't use the resolvent length to decide whether it is important enough. As we saw in chapter 7, the importance of a clause is decided both by it's length and it's RHS. Therefore we will use the *stringency* of a constraint to decide whether the resolvent is important enough to add it. The stringency of a constraint decides how difficult it is to satisfy that constraint. For example, in a `8,7-CARD` constraint at most one literal may be falsified. It is difficult to satisfy this constraint, but this also means that such a constraint will help to find conflicts faster. Therefore we only want to add resolvents which are stringent. As a measure for the stringency we will use the weights from chapter 7, which decide the importance of constraints.

The results of this test are shown in table 8.4. In the first column

the used stringency threshold is shown. `NONE` means that no resolution is performed at all. A threshold of 0.0 means that any constraint is allowed. So, every possible resolvent is added. In the third column the number of solved instances is shown. The last column shows the total solving time.

Notice that for every used threshold the same number of benchmarks were solved, except for the pigeonholes, which seem to benefit from adding all possible resolvents. Besides this benefit, it seems clear that adding stringency resolvents has a devastating effect on the performance. Less resolvents result in better the solve times. Also notice that in the case of the pigeonholes the solve time for solving all 29 benchmarks is slightly higher than we found after adding least-occurrence resolvents. Because only the pigeonholes seem to benefit from this type of resolution, and because the pigeonholes can also be solved with least-occurrence resolution, we conclude that stringency resolution is not a beneficial strategy.

8.3 Increasing the number of potential unit propagations

An important advantage of resolution on cardinality constraints over resolution on CNF clauses, is that it can result in more potential unit propagations. When resolution is performed on clauses with equal literals, this will result in a PB resolvent. However, in some situations, this does not matter and we can reduce it to a cardinality constraint.

Look at the following cardinality constraints:

$$\begin{aligned} a + b + c &\geq 1 \\ a + \neg b + d &\geq 1 \end{aligned}$$

which results in the following resolvent:

$$2a + c + d \geq 1$$

In this case you can remove the coefficient without doing any harm. In fact, you have created a new possibility for unit propagation. If the variables c and d are both falsified, this resolvent shows that you must set a. The original two clauses don't lead to that conclusion.

We differentiate between three situations in which similar results can be obtained:

1. If some coefficients of the resolvents are 2, then they can safely be removed if the RHS equals 1. An example of this situation is:

$$\begin{array}{l} a + b + c \geq 1 \\ \neg a + \neg b + c + d \geq 2 \\ \hline 2c + d \geq 1 \\ c + d \geq 1 \end{array}$$

Table 8.4: Results of test with stringency resolution

Threshold	benchmarks	# total	# solved	# resolvents	solve time
NONE	pigeonholes	29	9	0	148.92
	longmult (0-7)	8	8	0	**534.71**
	4,2-rand	10	10	0	**219.28**
	5,2-rand	10	10	0	**19.42**
	6,2-rand	10	10	0	**23.93**
	7,2-rand	10	10	0	**118.47**
	hanoi4	1	1	0	**75.08**
2.0	pigeonholes	29	9	2	148.94
	longmult (0-7)	8	8	110	535.16
	4,2-rand	10	10	0	**219.28**
	5,2-rand	10	10	0	**19.42**
	6,2-rand	10	10	0	**23.93**
	7,2-rand	10	10	0	**118.47**
	hanoi4	1	1	249	86.06
1.0	pigeonholes	29	9	18	148.97
	longmult (0-7)	8	8	16870	632.81
	4,2-rand	10	10	1	219.32
	5,2-rand	10	10	0	**19.42**
	6,2-rand	10	10	0	**23.93**
	7,2-rand	10	10	0	**118.47**
	hanoi4	1	1	739	107.89
0.2	pigeonholes	29	9	60	199.30
	longmult (0-7)	8	8	24045	776.99
	4,2-rand	10	10	6571	565.78
	5,2-rand	10	10	0	**19.42**
	6,2-rand	10	10	0	**23.93**
	7,2-rand	10	10	0	**118.47**
	hanoi4	1	1	975	140.03
0.1	pigeonholes	29	9	60	199.30
	longmult (0-7)	8	8	24126	780.32
	4,2-rand	10	10	7907	742.81
	5,2-rand	10	10	0	**19.42**
	6,2-rand	10	10	0	**23.93**
	7,2-rand	10	10	0	**118.47**
	hanoi4	1	1	981	145.04
0.0	pigeonholes (2-30)	29	29	901	**456.88**
	longmult (0-7)	8	8	37133	1133.71
	4,2-rand	10	10	9696	1389.64
	5,2-rand	10	10	2793	70.64
	6,2-rand	10	10	1944	56.08
	7,2-rand	10	10	1686	176.56
	hanoi4	1	1	1748	460.94

This situation can be recognized by checking whether the constraints contain equal literals and whether the RHS of the resulting resolvent equals 1.

2. If all coefficients are equal to 2, then you can just divide the RHS by 2 and remove the coefficients. Although such a situation will probably happen seldom, its results are very good. An example of this is:

$$\begin{array}{l} a + b + c + d \geq 2 \\ \neg a + \neg b + c + d \geq 2 \\ \hline 2c + 2d \geq 2 \\ c + d \geq 1 \end{array}$$

   This situation can be recognized by checking whether there are no literals that appear only in one of the source constraints, but not in the other.

3. There is one situation in which the some coefficients (but not necessarily all) may be 2 with a RHS larger than 1. If all coefficients together are equal to the RHS, the resolvent is clearly a unit constraint. It actually is a PB constraint, but it is easily recognizable and therefore we added this case. An example of this is:

$$\begin{array}{l} a + b + c \geq 2 \\ \neg a + \neg b + c + d \geq 3 \\ \hline 2c + d \geq 3 \\ c + d \geq 2 \end{array}$$

   This situation can be recognized by counting the equal literals and the literals that appear only in one of the source constraints, and checking whether this equals the RHS of the resolvent.

We have implemented this by checking for any potential resolvent whether one of these three situations applies. In those cases the resolvent is added. Checking for these types of resolvents is done iteratively, because added resolvents may result in new opportunities for resolution. The results of the tests are shown in table 8.5. The first column shows the used benchmarks. The second and third column show the total number of instances and the number of solved instances. The next three columns show the number of resolvents for each of the three types. The last column shows the total solving time.

Notice that the second and third type of resolvents never occur. Longmult, `5,1-rand`, `5,2-rand`, `6,1-rand` and `6,2-rand` benchmarks result in higher solve times because of the added resolvents. Only the `3,1-rand` benchmarks and the hanoi benchmark seems to benefit a little from the added resolvents. These results are a little strange, because in [13] it was

Table 8.5: Results of test with extra unitpropagation resolution

Benchmarks	# total	# solved	# res(1)	# res(2)	# res(3)	solve time
pigeonholes (2-30)	29	9	-	-	-	**148.83**
longmult (0-7)	8	8	-	-	-	**532.26**
`3,1-rand`	50	50	-	-	-	562.32
`4,1-rand`	50	50	-	-	-	1204.43
`4,2-rand`	50	50	-	-	-	**1439.01**
`5,1-rand`	50	50	-	-	-	**191.44**
`5,2-rand`	50	50	-	-	-	**95.21**
`6,1-rand`	50	50	-	-	-	**544.23**
`6,2-rand`	10	10	-	-	-	**133.26**
hanoi4	1	1	-	-	-	75.14
pigeonholes (2-30)	29	9	0	0	0	149.40
longmult (0-7)	8	8	2128	0	0	590.43
`3,1-rand`	50	50	5404	0	0	**531.40**
`4,1-rand`	50	50	411	0	0	**1200.35**
`4,2-rand`	50	50	0	0	0	**1439.01**
`5,1-rand`	50	50	649	0	0	204.74
`5,2-rand`	50	50	3	0	0	95.38
`6,1-rand`	50	50	493	0	0	616.33
`6,2-rand`	10	10	566	0	0	136.79
hanoi4	1	1	436	0	0	**67.94**

shown that adding the first type of resolvents generally results in 10% to 15% better performance of March_dl on 3SAT benchmarks. Also other benchmarks were solved faster. Except for one benchmark, solve times didn't become higher.

It seems that especially the longer cardinality constraints are influenced negatively by the resolvents. Because there are only a few cases where adding these resolvents seems to be beneficial, we will not use this type of resolution in our solver.

Chapter 9

Results

In this chapter we will discuss the results of our solver. However, first we need to explain in which way we implemented our solver, and how we used the results from previous chapters to optimize its performance. After we explained this, we will give a short overview of the solvers with which we will compare our solver. We will also explain our choice for the set of benchmarks used for testing. In the next section we will give the results of the selected solvers on all relevant benchmarks. These results will be shortly discussed afterwards.

9.1 Solver implementation

In our implementation, we used the basic architecture as discussed in chapter 3. This architecture uses the DPLL algorithm and a lookahead procedure to select the next branch variable. In this chapter we also explained the way in which cardinality clauses must be managed. We used this architecture as a starting point for our solver.

Additionally, we added all techniques discussed in previous chapters. Most techniques are used in a preprocessing phase, which will be executed before the DPLL algorithm. The preprocessing phase will start with a short clean-up of the formula. All redundant constraints are removed, using the algorithm from section 4.1. A call to the `IterativeUnitPropagation` method will shrink the formula as far as possible at this moment.

Next, the recognition algorithm from chapter 4 is used to syntactically filter cardinality constraints from the CNF clauses in the formula. If new cardinality constraints are found, this results in many redundant constraints. Therefore, again the domination algorithm is used to remove all redundancy.

The last step in the preprocessing phase is resolution, as discussed in chapter 8. In this chapter three different strategies were proposed, of which only the first seemed beneficial. We have implemented this strategy. Resolution is only performed when the resolution variable appears only in the

two source constraints and nowhere else.

After this preprocessing phase, the DPLL algorithm is started. During each iteration the lookahead procedure is called to find failed literals and to choose the next branch variable. As discussed in chapter 7, branching variables are selected through the diff function. This function is implemented by using the stringency of all reduced constraints, as explained in section 7.2.

The solver which resulted after these changes in the implementation is used in this chapter to obtain all results. Because we didn't test our choice of weights on a broad set of benchmarks, we decided add another version of our solver, which doesn't use weights. We will call this version 3MCard-ww.

9.2 Solvers

Because we have some doubts of the usefulness of different weights compared to no weights at all, we will use two versions of our solver. The first version 3MCard, is the version as explained in the section 9.1. The second version, 3MCard-ww, has one difference, which is that this version won't use any weights.

For comparison, we will use both SAT solvers and PB solvers. All selected solvers are state-of-the-art solvers, most of which competed in the SAT Competition [1]
As SAT solvers we have chosen:

- `Satz`[2]. Created by Chu Min Li. We will use version 2.15.
- `Minisat`[3]. Created by Niklas Éen and Niklas Sörensson. We will use version 1.13.
- `March`[4]. Created by Marijn Heule and Hans van Maaren. We will use the March_dl version.

As PB solvers we have chosen:

- `Galena`[5]. Created by Donald Chai and Andreas Kuehlmann.
- `Pueblo`[6]. Created by Hossein Sheini and Karem Sakallah. We will use version 1.4.

[1] See `www.satcompetition.org`
[2] `www.laria.u-picardie.fr/~cli/EnglishPage.html`
[3] `www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`
[4] `www.st.ewi.tudelft.nl/sat/march_dl.php`
[5] `www.eecs.berkeley.edu/~donald/code/galena/`
[6] `www.eecs.umich.edu/\%7Ehsheini/pueblo/`

- `Minisat+`[7]. Created by Niklas Éen and Niklas Sörensson. This solver translates PB constraints through several translation algorithms back to CNF clauses, after which SAT solving is performed.

9.3 Comparison

All benchmarks will be run on a Pentium III with 733MHz and 256MB RAM, running Mandrake Linux version 9.3. If a solve time in one of the tables is printed *italicized*, this means that this instance couldn't be solved within the given time limit. For all random benchmarks, a time limit of *600* seconds will be used. For all other benchmarks the time limit is *1200* seconds. If a solve time is printed **bold**, this means that that solver solved the benchmark faster than the other solvers. If there is no solve time, but the word *Error*, this means that the solver didn't return the right answer or terminated otherwise within the time limit.

9.3.1 Benchmarks from the PB Competition

Table 9.1: Comparison of solvers on chnl

	3MCard	3MCard-ww	Galena	Pueblo	Minisat+
chnl10-11	0.14	0.15	**0.0**	0.01	51.80
chnl10-15	0.35	0.34	**0.0**	**0.0**	251.41
chnl10-20	0.74	0.74	**0.01**	0.02	231.00
chnl15-16	1.51	1.49	**0.01**	**0.01**	*>1200*
chnl15-20	2.93	2.90	**0.01**	0.02	*>1200*
chnl15-25	5.39	5.42	**0.0**	0.04	*>1200*
chnl20-21	8.31	8.24	**0.01**	0.03	*>1200*
chnl20-25	14.56	13.87	**0.01**	0.05	*>1200*
chnl20-30	23.33	23.18	**0.01**	0.07	*>1200*
chnl30-31	92.96	92.36	**0.03**	0.15	*>1200*
chnl30-35	133.63	132.48	**0.03**	0.22	*>1200*
chnl30-40	196.23	195.48	**0.04**	0.44	*>1200*
chnl35-36	234.45	232.21	**0.04**	0.26	*>1200*
chnl35-40	321.00	319.28	**0.04**	0.36	*>1200*
chnl35-45	451.09	456.87	**0.06**	0.45	*>1200*
chnl40-41	526.33	524.19	**0.05**	0.55	*>1200*
chnl40-45	700.23	695.72	**0.05**	0.62	*>1200*
chnl40-50	958.94	956.13	**0.08**	6.96	*>1200*
chnl50-51	*>1200*	*>1200*	**0.1**	1.31	*>1200*
chnl50-55	*>1200*	*>1200*	**0.11**	1.9	*>1200*
chnl50-60	*>1200*	*>1200*	**0.13**	2.26	*>1200*

[7] `www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`

Table 9.2: Comparison of solvers on fpga

	3MCard	3MCard-ww	Galena	Pueblo	Minisat+
fpga10-8	0.01	**0.00**	104.12	0.03	0.02
fpga10-9	0.02	0.02	0.13	0.03	**0.01**
fpga10-10	0.02	0.02	0.24	0.05	**0.01**
fpga11-9	0.02	0.02	*>1200*	0.03	**0.01**
fpga11-10	**0.02**	0.03	0.35	0.04	**0.02**
fpga11-11	0.04	0.03	0.14	0.03	**0.01**
fpga12-10	0.03	0.03	*>1200*	0.04	**0.02**
fpga12-11	0.04	0.04	0.09	0.07	**0.02**
fpga12-12	0.05	0.05	0.14	0.04	**0.02**
fpga13-11	0.05	0.04	*>1200*	0.03	**0.02**
fpga13-12	0.06	0.04	0.2	0.07	**0.02**
fpga13-13	0.07	0.07	0.15	**0.05**	0.09
fpga14-12	0.06	0.04	*>1200*	0.06	**0.01**
fpga14-13	0.08	0.07	0.33	0.05	**0.03**
fpga14-14	0.10	0.09	1.33	0.04	**0.02**
fpga15-13	0.09	0.08	*>1200*	0.06	**0.02**
fpga15-14	0.10	0.09	2.36	0.05	**0.04**
fpga15-15	0.15	0.12	0.48	**0.05**	**0.05**
fpga20-18	0.35	0.32	*>1200*	**0.11**	0.26
fpga20-19	0.42	0.38	43.74	**0.19**	0.4
fpga20-20	0.49	0.45	22.79	**0.12**	**0.12**
fpga25-23	1.05	0.93	*>1200*	**0.21**	35.74
fpga25-24	1.18	1.06	*>1200*	**0.27**	0.54
fpga25-25	1.37	1.24	*>1200*	**0.24**	0.36
fpga30-28	2.54	**2.28**	*>1200*	737.44	27.69
fpga30-29	2.89	2.59	*>1200*	0.55	**0.20**
fpga30-30	3.17	2.86	214.7	**0.48**	3.33
fpga35-33	5.49	4.90	*>1200*	**0.81**	*>1200*
fpga35-34	6.01	5.41	*>1200*	**1.43**	371.12
fpga35-35	6.66	6.03	177.57	**1.05**	198.89
fpga40-38	10.59	**9.35**	*>1200*	400.56	*>1200*
fpga40-39	11.60	10.31	*>1200*	**6.51**	716.76
fpga40-40	12.55	11.28	*>1200*	**2.34**	*>1200*
fpga45-43	19.06	16.87	*Error*	**16.68**	*>1200*
fpga45-44	20.56	**18.31**	*>1200*	36.57	*>1200*
fpga45-45	22.16	19.91	*>1200*	4.31	**1.33**

3MCard seems to perform reasonably well on the fpga benchmarks (figure 9.2), although Pueblo is slightly better. However, on the other three categories (figure 9.1, 9.3 and 9.4) the PB solvers outperform our solver. The fact that the PB solvers are better on the chnl benchmark shows that the reasoning we use is far from optimal. The chnl benchmark is a cardinality benchmark with only cardinality constraints. Therefore a cardinality solver

Table 9.3: Comparison of solvers on ooo.rf

	3MCard	3MCard-ww	Galena	Pueblo	Minisat+
ooo.rf6	14.83	*>1200*	**0.05**	0.32	0.12
ooo.rf7	*>1200*	*>1200*	**0.1**	1.54	0.33
ooo.rf8	*>1200*	*>1200*	**0.18**	5.55	1.29
ooo.rf9	*>1200*	*>1200*	**0.29**	29.62	6.24
ooo.rf10	*>1200*	*>1200*	**0.47**	191.14	33.18

Table 9.4: Comparison of solvers on elf.rf

	3MCard	3MCard-ww	Galena	Pueblo	Minisat+
elf.rf6	0.02	0.02	**0.0**	0.01	**0.00**
elf.rf7	15.41	15.51	**0.03**	0.15	0.10
elf.rf8	*>1200*	*>1200*	**0.12**	1.61	0.48
elf.rf9	*>1200*	*>1200*	**0.53**	14.2	2.09
elf.rf10	*>1200*	*>1200*	**1.47**	164.42	14.32

should be able to perform better on this benchmark than a PB solver. The fact that Minisat+ can even solve the ooo.rf and elf.rf benchmarks without problems (despite the translation to CNF) strengthens this conclusion.

9.4 CNF benchmarks

Table 9.5: Comparison of solvers on hanoi

	3MCard	3MCard-ww	satz	Minisat	March	Galena	Pueblo	Minisat+
hanoi4	75.58	42.03	22.91	**0.17**	24.87	4.72	0.91	0.40
hanoi5	*>1200*	*>1200*	*>1200*	59.1	*>1200*	301.25	39.52	**35.41**

Table 9.6: Comparison of solvers on longmult

	3MCard	3MCard-ww	satz	Minisat	March	Galena	Pueblo	Minisat+
longmult8	*>1200*	*>1200*	1010.35	**46.23**	93.60	107.07	94.81	51.42
longmult10	*>1200*	*>1200*	*>1200*	**153.17**	317.14	305.47	517.0	210.25
longmult12	*>1200*	*>1200*	*>1200*	**213.28**	675.68	318.45	551.07	246.3

The results of our solver on hanoi (figure 9.5) and longmult (figure 9.6) again show that better reasoning is required to solve these benchmarks. The

PB solvers and one of the SAT solvers can solve both hanoi benchmarks in reasonable time. Also, the other solvers seem to have no problems with the longmult benchmarks, while 3MCard cannot even solve the first of these benchmarks.

9.5 Pigeonhole benchmarks

Table 9.7: Comparison of solvers on pigeonhole (CNF)

	3MCard	3MCard-ww	satz	Minisat	March	Galena	Pueblo	Minisat+
hole10	**0.16**	**0.16**	137.29	100.10	387.21	37.3	*Error*	65.56
hole11	**0.27**	**0.27**	*>1200*	*>1200*	*>1200*	307.16	*Error*	*>1200*
hole12	**0.44**	**0.44**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole13	**0.70**	**0.70**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole14	**1.07**	**1.07**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole15	**1.61**	**1.61**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole16	**2.33**	**2.33**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole17	**3.32**	**3.32**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole18	**4.61**	**4.61**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole19	**6.37**	**6.37**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole20	**8.61**	**8.61**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole21	**11.45**	**11.45**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole22	**15.05**	**15.05**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole23	**19.59**	**19.59**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole24	**25.21**	**25.21**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole25	**32.02**	**32.02**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole26	**40.40**	**40.40**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole27	**50.61**	**50.61**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole28	**62.78**	**62.78**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole29	**77.19**	**77.19**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*
hole30	**94.54**	**94.54**	*>1200*	*>1200*	*>1200*	*>1200*	*Error*	*>1200*

Table 9.7 shows the true power of our recognition algorithm. 3MCard can solve all pigeonhole benchmarks, where most solvers already fail on pigeonhole 12. The results in table 9.8 show that when cardinality constraints are used, Galena outperforms 3MCard. Even though our solver only needs $2n$ resolution steps to solve the pigeonholes, Galena seems to know the answer instantly. This shows that the implementation of our resolution algorithm should be optimized. Note that Pueblo returned SAT in all cases, which is obviously wrong.

Table 9.8: Comparison of solvers on pigeonhole (CARD)

	3MCard	3MCard-ww	Galena	Pueblo	Minisat+
hole10	0.14	0.14	**0.0**	*Error*	95.76
hole11	0.24	0.24	**0.0**	*Error*	*>1200*
hole12	0.40	0.40	**0.0**	*Error*	*>1200*
hole13	0.65	0.65	**0.01**	*Error*	*>1200*
hole14	1.01	1.00	**0.0**	*Error*	*>1200*
hole15	1.50	1.50	**0.01**	*Error*	*>1200*
hole16	2.20	2.20	**0.0**	*Error*	*>1200*
hole17	3.14	3.14	**0.01**	*Error*	*>1200*
hole18	4.41	4.41	**0.0**	*Error*	*>1200*
hole19	6.10	6.10	**0.0**	*Error*	*>1200*
hole20	8.26	8.26	**0.0**	*Error*	*>1200*
hole21	11.02	11.02	**0.01**	*Error*	*>1200*
hole22	14.53	14.53	**0.01**	*Error*	*>1200*
hole23	18.95	18.95	**0.01**	*Error*	*>1200*
hole24	24.44	24.44	**0.01**	*Error*	*>1200*
hole25	31.37	31.37	**0.01**	*Error*	*>1200*
hole26	39.30	39.30	**0.01**	*Error*	*>1200*
hole27	49.29	49.29	**0.01**	*Error*	*>1200*
hole28	61.26	61.26	**0.01**	*Error*	*>1200*
hole29	75.04	75.04	**0.01**	*Error*	*>1200*
hole30	92.36	92.36	**0.01**	*Error*	*>1200*

9.6 Random cardinality benchmarks

Table 9.9: Comparison of solvers on `3,1-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`3,1-rand 1`	13.23	10.44	3.08	12.63	**1.32**	*>600*	113.57	11.82
`3,1-rand 2`	11.11	9.57	3.37	18.00	**1.46**	*>600*	99.33	18.97
`3,1-rand 3`	12.08	9.77	3.02	9.53	**1.29**	464.02	39.4	12.64
`3,1-rand 4`	11.01	8.73	3.04	14.05	**1.36**	*>600*	68.18	9.95
`3,1-rand 5`	6.03	4.79	1.56	4.90	**0.77**	112.94	28.35	4.69
`3,1-rand 6`	6.48	5.16	1.63	3.13	**0.70**	44.38	12.53	2.76
`3,1-rand 7`	18.76	15.12	5.17	17.50	**2.12**	*>600*	125.25	23.21
`3,1-rand 8`	15.09	12.09	5.92	18.21	**1.62**	*>600*	112.62	19.70
`3,1-rand 9`	12.31	9.84	3.64	13.14	**1.38**	596.76	96.38	9.94
`3,1-rand 10`	10.78	8.62	2.28	12.30	**1.27**	584.75	52.22	7.89

Table 9.10: Comparison of solvers on `4,1-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`4,1-rand 1`	21.55	21.85	**16.62**	22.88	30.63	*>600*	119.46	25.51
`4,1-rand 2`	30.42	28.83	**27.35**	40.47	34.15	*>600*	215.61	57.30
`4,1-rand 3`	**19.29**	19.74	20.70	29.71	27.00	*>600*	110.69	31.26
`4,1-rand 4`	21.18	19.66	23.69	23.27	**19.73**	*>600*	153.22	26.34
`4,1-rand 5`	27.67	27.06	**22.40**	47.99	28.01	*>600*	170.79	39.30
`4,1-rand 6`	26.72	26.84	**26.50**	45.18	29.33	*>600*	190.9	43.54
`4,1-rand 7`	19.21	19.67	**12.49**	24.85	28.01	*>600*	124.71	23.71
`4,1-rand 8`	21.48	21.61	**15.68**	28.53	29.33	*>600*	110.24	34.58
`4,1-rand 9`	24.26	22.99	25.91	47.07	**16.20**	*>600*	186.84	39.55
`4,1-rand 10`	28.05	28.86	**20.73**	49.33	27.01	*>600*	242.04	51.79

Table 9.11: Comparison of solvers on `4,2-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`4,2-rand 1`	9.12	11.87	4.87	6.69	**2.26**	316.05	155.61	38.09
`4,2-rand 2`	9.47	8.98	5.21	9.56	**2.23**	327.59	123.1	32.29
`4,2-rand 3`	23.38	23.45	11.91	26.71	**5.62**	*>600*	318.68	103.00
`4,2-rand 4`	10.01	11.78	4.45	7.95	**1.76**	124 93	79.84	23.17
`4,2-rand 5`	31.55	34.82	13.50	37.63	**6.92**	*>600*	450.16	151.45
`4,2-rand 6`	53.42	45.23	17.24	63.57	**8.96**	*>600*	263.27	*>600*
`4,2-rand 7`	40.72	36.61	15.95	46.23	**7.13**	*>600*	423.96	191.94
`4,2-rand 8`	10.70	9.45	4.84	9.15	**2.02**	268.09	84.36	27.12
`4,2-rand 9`	15.40	17.62	7.37	8.12	**2.33**	252.65	88.81	32.62
`4,2-rand 10`	19.46	16.79	6.28	8.21	**3.08**	237.68	110.45	48.99

Table 9.12: Comparison of solvers on `5,1-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`5,1-rand 1`	4.57	4.86	3.79	**3.00**	11.58	21.32	7.89	3.58
`5,1-rand 2`	4.83	5.12	4.17	**2.95**	13.01	17.18	6.67	3.55
`5,1-rand 3`	4.37	4.65	4.49	3.12	11.32	14.06	7.66	**3.06**
`5,1-rand 4`	4.92	5.47	4.63	3.63	12.30	25.73	11.75	**3.52**
`5,1-rand 5`	4.46	5.37	4.33	3.42	12.22	21.34	10.19	**3.26**
`5,1-rand 6`	4.65	5.28	5.19	**3.22**	13.61	20.88	9.08	3.65
`5,1-rand 7`	4.85	5.09	4.69	4.36	12.60	23.23	10.22	**3.64**
`5,1-rand 8`	4.29	4.62	4.51	**2.62**	11.01	14.39	8.42	2.86
`5,1-rand 9`	4.45	5.21	4.64	3.57	12.55	31.83	11.68	**3.42**
`5,1-rand 10`	4.97	5.14	4.18	**3.27**	12.81	25.08	14.78	3.37

Table 9.13: Comparison of solvers on `5,2-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`5,2-rand 1`	2.46	**2.45**	6.13	2.99	3.82	48.16	24.72	13.61
`5,2-rand 2`	**1.96**	2.04	5.08	2.68	2.77	22.48	19.66	12.51
`5,2-rand 3`	**1.60**	2.25	3.83	2.78	2.36	27.92	22.84	14.65
`5,2-rand 4`	**1.04**	1.13	3.98	1.46	1.97	15.01	6.74	5.95
`5,2-rand 5`	2.54	**2.21**	5.26	3.65	2.48	21.05	26.33	13.63
`5,2-rand 6`	**1.87**	1.91	3.56	2.51	2.76	25.39	15.67	14.09
`5,2-rand 7`	1.59	1.65	3.01	**1.29**	1.96	10.11	13.09	6.45
`5,2-rand 8`	**2.57**	3.05	3.19	2.84	3.78	50.22	18.5	13.76
`5,2-rand 9`	2.07	**1.97**	4.48	2.09	2.31	20.83	17.27	9.47
`5,2-rand 10`	1.77	**1.74**	3.43	2.11	2.05	19.19	14.13	8.49

Table 9.14: Comparison of solvers on `5,3-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`5,3-rand 1`	3.8	8.40	1.92	1.69	**0.93**	37.32	17.83	5.75
`5,3-rand 2`	38.41	62.70	16.96	21.36	**7.20**	*>600*	381.47	89.42
`5,3-rand 3`	4.96	4.37	2.78	1.88	**1.44**	172.99	26.58	11.33
`5,3-rand 4`	2.08	2.55	1.39	1.24	**0.78**	9.04	33.94	6.21
`5,3-rand 5`	11.68	20.15	5.55	8.11	**2.91**	214.14	160.51	48.75
`5,3-rand 6`	24.74	33.92	12.67	14.50	**5.98**	563.71	536.49	87.26
`5,3-rand 7`	9.69	17.24	4.39	1.89	**1.65**	37.2	34.72	10.46
`5,3-rand 8`	15.88	16.91	6.34	12.42	**2.59**	160.37	104.7	47.00
`5,3-rand 9`	19.95	25.69	12.80	9.82	**4.43**	566.93	173.48	38.66
`5,3-rand 10`	14.98	21.78	7.92	9.74	**2.98**	180.25	229.27	60.59

Table 9.15: Comparison of solvers on `6,1-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`6,1-rand 1`	13.63	14.52	12.60	**5.63**	53.97	27.63	24.13	5.91
`6,1-rand 2`	13.49	14.99	11.67	**5.50**	54.36	24.37	17.03	8.35
`6,1-rand 3`	12.95	14.31	12.30	**6.13**	57.95	27.75	18.05	6.76
`6,1-rand 4`	13.66	15.13	11.54	6.73	55.54	31.03	24.16	**6.08**
`6,1-rand 5`	13.91	14.73	11.89	**7.33**	53.09	33.77	24.67	8.12
`6,1-rand 6`	14.22	15.39	11.72	6.58	57.98	23.01	22.10	**6.36**
`6,1-rand 7`	13.29	14.30	11.55	6.71	49.02	25.09	24.58	**6.05**
`6,1-rand 8`	13.29	14.59	11.13	**6.28**	52.09	28.64	24.18	6.73
`6,1-rand 9`	13.72	15.52	11.37	7.54	57.88	28.14	17.04	**7.10**
`6,1-rand 10`	13.39	14.63	11.05	7.08	52.15	23.16	23.72	**6.67**

Table 9.16: Comparison of solvers on `6,2-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`6,2-rand 1`	**2.50**	3.06	9.99	5.72	8.12	23.16	23.06	34.17
`6,2-rand 2`	**2.28**	2.55	8.52	3.46	8.23	21.39	14.80	18.30
`6,2-rand 3`	**2.18**	2.89	11.44	3.80	7.79	26.01	22.24	26.52
`6,2-rand 4`	**2.49**	3.30	11.24	4.07	9.68	32.57	17.70	26.80
`6,2-rand 5`	**2.44**	3.79	12.32	4.99	9.79	22.42	17.89	27.09
`6,2-rand 6`	**2.40**	3.66	12.97	4.85	9.24	24.31	25.04	32.03
`6,2-rand 7`	**2.53**	3.59	11.80	4.56	9.09	25.17	20.51	24.57
`6,2-rand 8`	**2.78**	3.86	12.78	4.98	9.22	40.43	20.69	24.74
`6,2-rand 9`	**2.05**	2.61	12.40	2.95	7.86	13.92	11.60	18.81
`6,2-rand 10`	**2.27**	3.00	10.26	4.49	9.45	13.77	17.33	21.28

Table 9.17: Comparison of solvers on `6,3-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`6,3-rand 1`	**13.25**	20.77	235.88	39.48	23.61	*>600*	184.98	125.25
`6,3-rand 2`	**7.11**	9.54	41.16	17.17	9.38	133.21	71.56	46.84
`6,3-rand 3`	**5.41**	9.34	30.83	8.53	7.88	70.45	52.17	30.19
`6,3-rand 4`	**9.02**	9.79	76.23	15.48	15.12	259.71	80.17	64.82
`6,3-rand 5`	**15.30**	18.52	102.73	42.72	30.63	*>600*	274.59	221.46
`6,3-rand 6`	**5.22**	6.34	100.49	55.52	23.59	432.95	280.18	193.82
`6,3-rand 7`	**11.03**	15.59	56.49	38.28	18.25	*>600*	40.18	117.76
`6,3-rand 8`	**13.29**	25.35	116.300	48.00	38.80	*>600*	394.49	240.31
`6,3-rand 9`	13.72	22.54	31.03	**6.54**	7.06	66.92	46.03	30.21
`6,3-rand 10`	**13.39**	19.38	69.30	25.44	17.75	581.97	329.89	126.47

Table 9.18: Comparison of solvers on `6,4-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`6,4-rand 1`	4.16	7.37	2.75	1.52	**1.13**	5.63	15.01	5.47
`6,4-rand 2`	0.91	2.14	1.01	0.26	0.42	**0.12**	0.45	1.23
`6,4-rand 3`	1.61	11.86	1.33	0.94	0.68	**0.23**	2.59	4.45
`6,4-rand 4`	6.21	12.25	3.17	2.72	**1.11**	17.19	39.28	8.77
`6,4-rand 5`	5.84	11.78	3.37	1.89	**1.45**	10.17	47.34	10.32
`6,4-rand 6`	4.60	8.70	3.13	2.01	**1.35**	67.39	50.68	9.95
`6,4-rand 7`	20.07	104.08	9.72	10.34	**3.83**	587.15	92.29	38.26
`6,4-rand 8`	4.69	5.89	2.14	1.79	**0.82**	1.59	20.03	5.75
`6,4-rand 9`	10.49	11.85	4.79	2.85	**1.66**	7.32	34.88	8.74
`6,4-rand 10`	5.84	10.38	3.77	1.43	1.29	**0.52**	19.20	7.12

Table 9.19: Comparison of solvers on `7,1-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`7,1-rand 1`	49.55	54.84	32.90	17.11	295.71	46.69	52.64	**16.11**
`7,1-rand 2`	50.94	55.88	34.90	**16.33**	302.41	48.15	71.71	18.91
`7,1-rand 3`	50.02	53.67	33.02	17.27	285.49	45.27	51.46	**14.70**
`7,1-rand 4`	50.27	55.09	34.53	**16.52**	316.56	44.66	54.65	18.51
`7,1-rand 5`	50.96	55.22	34.78	**16.42**	326.08	49.45	72.85	17.00
`7,1-rand 6`	49.74	54.23	35.97	17.56	314.24	44.70	**9.01**	14.17
`7,1-rand 7`	48.61	53.08	33.53	**16.36**	312.80	41.30	47.61	18.63
`7,1-rand 8`	49.68	54.07	32.64	17.43	308.78	42.30	26.18	**16.41**
`7,1-rand 9`	50.11	54.36	34.70	16.70	315.12	43.19	52.00	**14.82**
`7,1-rand 10`	50.11	55.64	34.54	**16.95**	295.12	53.57	72.76	18.61

Table 9.20: Comparison of solvers on `7,2-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`7,2-rand 1`	**12.88**	20.48	102.97	33.98	125.55	208.90	99.11	345.87
`7,2-rand 2`	**11.99**	21.65	85.10	34.65	133.83	305.75	74.08	222.99
`7,2-rand 3`	**12.29**	20.88	57.73	37.38	105.82	220.69	74.53	297.86
`7,2-rand 4`	**10.59**	17.99	54.09	30.99	105.04	137.76	101.08	241.13
`7,2-rand 5`	**11.83**	20.07	61.03	26.94	99.84	165.71	110.00	213.76
`7,2-rand 6`	**12.06**	17.83	52.73	41.22	99.71	170.11	88.84	310.05
`7,2-rand 7`	**14.09**	22.86	59.57	42.95	102.57	254.86	107.79	379.21
`7,2-rand 8`	**11.53**	19.01	69.40	33.23	103.71	230.64	106.20	358.98
`7,2-rand 9`	**13.29**	21.65	69.07	38.69	109.27	171.52	109.66	245.68
`7,2-rand 10`	**12.00**	17.60	76.14	39.11	118.77	144.57	78.21	289.51

Table 9.21: Comparison of solvers on `7,3-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`7,3-rand 1`	**52.59**	113.70	*>600*	545.05	264.52	*>600*	*>600*	*>600*
`7,3-rand 2`	**62.45**	148.01	*>600*	522.42	327.80	*>600*	*>600*	*>600*
`7,3-rand 3`	**66.54**	119.45	*>600*	474.27	346.91	*>600*	*>600*	*>600*
`7,3-rand 4`	**49.34**	105.22	*>600*	384.21	239.95	*>600*	*>600*	*>600*
`7,3-rand 5`	**76.07**	186.67	*>600*	*>600*	382.08	*>600*	*>600*	*>600*
`7,3-rand 6`	**56.46**	86.90	*>600*	297.75	262.10	*>600*	*>600*	*>600*
`7,3-rand 7`	**60.22**	117.34	*>600*	540.77	343.16	*>600*	*>600*	*>600*
`7,3-rand 8`	**72.29**	196.45	*>600*	*>600*	404.27	*>600*	*>600*	*>600*
`7,3-rand 9`	**59.18**	128.86	*>600*	*>600*	352.19	*>600*	*>600*	*>600*
`7,3-rand 10`	**57.19**	122.40	*>600*	518.94	346.47	*>600*	*>600*	*>600*

Table 9.22: Comparison of solvers on `7,4-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`7,4-rand 1`	**21.48**	48.42	335.66	46.25	43.28	*>600*	443.92	198.63
`7,4-rand 2`	**5.36**	16.23	176.15	11.63	11.59	108.85	81.32	52.71
`7,4-rand 3`	**7.13**	8.94	179.53	9.60	10.06	47.60	38.89	31.92
`7,4-rand 4`	**14.78**	28.12	203.58	36.98	25.79	258.87	160.31	114.41
`7,4-rand 5`	**20.02**	34.14	422.11	57.81	50.55	328.79	419.45	202.90
`7,4-rand 6`	**21.71**	43.63	448.31	38.07	34.57	*>600*	272.51	193.73
`7,4-rand 7`	**9.48**	16.12	180.95	20.68	21.21	87.48	129.96	107.61
`7,4-rand 8`	**24.42**	55.92	319.31	69.59	49.91	*>600*	376.24	170.71
`7,4-rand 9`	**23.19**	29.56	434.33	70.25	51.56	*>600*	63.73	186.48
`7,4-rand 10`	**25.24**	39.68	*>600*	42.89	44.61	*>600*	311.42	143.85

Table 9.23: Comparison of solvers on `7,5-rand`

	3MCard	3MCard-ww	Satz	Minisat	March	Galena	Pueblo	Minisat+
`7,5-rand 1`	21.25	46.50	8.89	7.02	3.56	**0.29**	207.51	27.03
`7,5-rand 2`	24.75	38.48	9.14	3.27	3.80	**0.07**	8.58	11.23
`7,5-rand 3`	89.15	223.52	41.17	70.14	**14.71**	160.12	539.70	372.12
`7,5-rand 4`	28.96	64.86	14.54	12.04	5.04	**0.92**	48.00	52.54
`7,5-rand 5`	17.28	23.83	11.27	4.83	4.65	**0.54**	8.64	16.97
`7,5-rand 6`	20.66	50.45	14.33	6.18	5.00	**0.76**	48.68	28.84
`7,5-rand 7`	24.14	67.95	10.61	6.30	3.31	**0.52**	5.08	21.39
`7,5-rand 8`	48.14	98.59	25.59	44.75	**11.38**	82.83	214.72	142.69
`7,5-rand 9`	85.50	127.37	31.10	52.03	12.13	**0.31**	104.69	498.70
`7,5-rand 10`	21.52	17.14	10.61	8.21	3.45	**0.03**	65.98	21.28

As can be expected, the SAT solvers perform better on the CNF benchmarks. March is faster on `3,1-rand` (table 9.9), Satz on `6,1-rand` (table 9.15) and Minisat and Minisat+ perform best on `5,1-rand`, `6,1-rand` and `7,1-rand` (table 9.12, 9.15 and 9.19). In none of these cases 3MCard performs very bad, but the difference with the best solvers (except for `4,1-rand`) is large.

March, and also other SAT solvers, perform also very good on the `N,N-2-rand` benchmarks (see table 9.11, 9.14, 9.18 and 9.23). This can be explained from the fact that after translation these benchmarks results in clauses of length 3, which are generally easier for SAT solvers than longer clauses.

3MCard performs best on all other benchmarks (see table 9.13, 9.16, 9.17, 9.20, 9.21 and 9.22). The differences with other solvers seem to become larger for longer cardinality constraints. For example, most solvers cannot even solve the `7,3-rand` benchmarks.

Another important result is the fact that for most benchmarks 3MCard seems to perform better than the 3MCard version which doesn't use weights.

Chapter 10

Conclusions and Future Work

In the introduction we asked ourselves whether a pure cardinality solver is useful. It is impossible to answer this question positively based only on the results of one study. However, we did find several positive results which indicate a potential future for cardinality solvers. Chapter 9 shows several situations in which our solver already outperforms both SAT solvers and PB solvers. Apart from these situations, we found several theoretical advantages, which can be exploited further in the future.

We will discuss the different subjects in this thesis one by one to explain these advantages.

10.1 Recognizing cardinality constraints from CNF

We have created an algorithm with which we can syntactically *recognize* cardinality constraints from a set of CNF clauses. With this algorithm it becomes possible to bring clauses from a SAT problem to a higher level and benefit from the advantages of the cardinality representation. The algorithm uses a straightforward translation, which does not use additional variables. This means that the search space remains the same. Recognition of cardinality constraints is an expensive process, even if you don't search for an optimal translation. However, our algorithm efficiently walks through the set of clauses while trying to recognize new constraints. We have implemented the algorithm as preprocessing in our solver, which enabled us to solve the SAT problems just as easy as their cardinality counterparts.

> *A recognition algorithm for cardinality constraints has been created, which can translate CNF clauses into cardinality constraints. This algorithm can be used for solving SAT benchmarks which can more efficiently be defined as cardinality problems.*

10.2 Compactness

An important advantage of defining a problem as a cardinality problem instead of a SAT problem is the fact that less literals are needed. This was clearly shown with the statistics in section 6.5. Especially the pigeonholes benefit from cardinality constraints, but the hanoi problems can be reduced significantly as well. Note that with our recognition algorithm it is possible that the cardinality constraints which are recognized show considerable overlap. If this happens, it is possible that the translation requires more literals than the original SAT problem. In these situations the recognition algorithm results in a translation which is far from optimal. In general however, if cardinality constraints can be recognized, then there exists a cardinality representation which is more efficient than the SAT representation.

> *The compactness of many problems can be reduced significantly by defining them with cardinality constraints instead of with CNF clauses. Generally size is reduced most when cardinality constraints have little overlap.*

10.3 Diff-function

We tried to find a function which chooses the most effective variable in the lookahead to branch on. This function has to be based on the results of the lookahead on each variable. It is difficult to tell when a variable is *effective* and many strategies are possible to decide when a variable is a more effective branch variable than other variables. We tried to use both the number of satisfied literals and the number of reduced constraints as an indication. It appeared that only the reduced constraints served as a good indication. We also differentiated between the type of constraints, by giving higher values to shorter constraints and constraints with a higher right-hand side. Trying to decide the optimal weights empirically didn't work, because the differences were too small. Based on the translation to CNF clauses, we came up with a rule which gives a value to all different cardinality constraints. Chapter 9 shows that these weights become more useful for longer cardinality constraints. In these cases performance is far better than the situation where no diff-function is used. Also, on benchmarks with a RHS which is at least 2 smaller than the length, but langer than 1, our solver outperforms all other solvers. It is clear that SAT solvers perform better on the CNF constraints. That SAT solvers also perform good on constraints with higher right-hand sides might be explained by the fact that the translation results in short CNF clauses which are solved relatively easy.

> *The diff-function can decide which variables to branch on by counting the weights of all reduced cardinality constraints. The*

results show that this leads to increased performance for constraints with a right-hand side which is at least 2 *smaller than its length, but higher than* 1*.*

10.4 Resolution

We tried several resolution strategies, which aimed to add only the useful resolvents. The first strategy only adds resolvents when the variable on which resolution is performed doesn't appear in other constraints. This strategy appears to be useful for many benchmarks, which couldn't be solved without this strategy. The second strategy meant adding any resolvent with a minimum stringency, where the stringency is the weight of a constraint, which was also used for the diff-function. This strategy didn't result in increased performance and was therefore abandoned. The last strategy was to add resolvents which result in potentially more unit propagations. In several situations this resulted in an increase of performance, while in other situations performance went down. Because we couldn't find an overall benefit, we abandoned this strategy as well. However, we expect that this strategy can be optimized, such that in situations where performance is currently going down, the solve times can be kept equal.

One beneficial resolution strategy was created, which appears crucial in solving many benchmarks. This strategy adds resolvents only the resolution variable doesn't appear in other constraints.

10.5 Future work

In this thesis we focused on the usefulness of a pure cardinality solver. Many areas still remain to be investigated. Additionally, we feel that several of the subjects that we did focus on can be further explored. We will discuss some of these subjects, and explain the possibilities for future work.

10.5.1 Recognition algorithm

First of all it seems interesting to further investigate the possibilities of our recognition algorithm. Currently we only used CNF clauses for recognition, but it is possible to include existing cardinality constraints as well. Take the following example:

$$\begin{array}{cc} x_1 + x_2 + x_3 \geq 2 & x_1 + x_4 \geq 1 \\ x_2 + x_4 \geq 1 & x_3 + x_4 \geq 1 \end{array}$$

Though the first constraint is not a CNF clause, it must be used together with the other clauses to form the longer constraint $x_1 + x_2 + x_3 + x_4 \geq 3$.

It is interesting to find out whether such situations ever occur, and whether the algorithm will slow down because of such an addition.

Next, we can expand the algorithm to check for *missing* literals. If for example 10 constraints are needed to form a cardinality constraint, but in the last constraint a literal is missing, then it might be beneficial to just add this literal, so that the clauses can be exchanged for the cardinality constraint. Take for example this set of clauses:

$$\begin{array}{ll} x_1 + x_2 + x_3 \geq 1 & x_1 + x_2 + x_4 \geq 1 \\ x_1 + x_2 + x_5 \geq 1 & x_1 + x_3 + x_4 \geq 1 \\ x_1 + x_3 + x_5 \geq 1 & x_1 + x_4 + x_5 \geq 1 \\ x_2 + x_3 + x_4 \geq 1 & x_2 + x_3 + x_5 \geq 1 \\ x_2 + x_4 + x_5 \geq 1 & x_3 + x_4 \geq 1 \end{array}$$

Our current algorithm will not recognize a cardinality constraint in this set. However, it is possible to just *add* the last literal, as this doesn't change the satisfiability of the problem.

Next, we would like to use some kind of domination algorithm, which can check the redundancy of one constraint against two or more constraints. Take for example these constraints:

$$\begin{array}{l} x_1 + x_2 + x_3 + x_4 \geq 2 \\ \neg x_1 + x_5 + x_6 \geq 2 \\ x_2 + x_3 + x_5 \geq 1 \end{array}$$

The first two constraints are together dominant over the third constraint, but with the domination algorithm that we're currently using, this redundancy can't be found.

Finally, research can also be done on the question whether the recognition algorithm might be useful during solving. Constraints are changing continuously, and this might also result in new possibilities for recognizing cardinality constraints.

10.5.2 Full lookahead vs. partial lookahead

Until now, we have used full lookahead in our solver. This means that in the lookahead all unvalued variables are tested. Clearly, not all variables are equally interesting. It is interesting to found out in what ways we can reduce the set of variables that are used in the lookahead.

10.5.3 Double lookahead

Another way to further shrink the set of interesting variables is to perform a double lookahead on (a part of) the lookahead variables. After setting one variable, a second variable can be set, which gives a better view of the consequences of using the first variable as branch variable. Obviously

a double lookahead results in considerably more work, so it becomes even more important here not to use the whole set of variables.

10.5.4 Resolution

We performed resolution only during preprocessing. It is interesting to see whether inserting resolvents during solving can help performance.

Bibliography

[1] F. Aloul, A. Ramani, I. Markov & K. Sakallah. PBS: A Backtrack Search Pseudo Boolean Solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT'2002)*, 2002.

[2] B. Aspvall, M. Plass, and R. Tarjan. A Linear-time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. *Information Processing Letters*, 8: 121-123, 1979.

[3] P. Barth. Linear 0-1 Inequalities and Extended Clauses. *Logic Programming and Automated Reasoning: International Conference LPAR '93; St Petersburg, Russia: Proceedings, July 1993.*

[4] P. Barth (1995). A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo Boolean Optimization. *Research Report MPI-I-95-2-003*, Max-Plack-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, Januari 1995.

[5] S. Buss, G. Turan. Resolution Proofs of Generalized Pigeonhole Principles. *Theoretical Computer Science*, 62(3):311-317, 1988.

[6] M. Davis, G. Logemann and D. Loveland. A Machine program for theorem-proving. *Communications of the ACM*, 5:394-397, 1962.

[7] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the Association for Computing Machinery*, 7:201-215, 1960.

[8] Dixon, H. E., & Ginsberg, M. L. Combining satisfiability techniques from AI and OR. *Knowledge Engrg. Rev., 15*, 31-45, 2000.

[9] H. E. Dixon, M. L. Ginsberg & A. J. Parkes. Generalizing Boolean Satisfiability I: Background and Survey of Existing Work. *Journal of Artificial Intelligence Research 21 (2004)* 193-243.

[10] N. Eén, N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation, 2 (2006)* 1-26.

[11] N. Eén & N. Sörensson. Minisat - A SAT Solver with Conflict-Clause Minimization. *The International Conference on Theory and Applications of Satisfiability Testing (2005)*

[12] Z. Fu, Y. Mahajan, S. Malik. New Features of SAT'04 version of zChaff. *The International Conference on Theory and Applications of Satisfiability Testing (2004)*

[13] M. Heule, H. van Maaren. March_dl: Adding Adaptive Heuristics and a new Branching Strategy. *Journal on Satisfiability, Boolean Modeling and Computation* 2 (2006), pp. 47-59.

[14] Guignard, M. & Spielberg, K. (1981). Logical reduction methods in zero-one programming. *Operations Research, 29.*

[15] S. Mertens, M. Mézard, R. Zecchina. Threshold Values of Random K-SAT from the Cavity Method. *Random Structures and Algorithms* 28, 3 (2006): 340-373.

[16] O. Kullmann, Investigating the behaviour of a SAT solver on random formulas. Submitted to *Annals of Mathematics and Artificial Intelligence* (2002).

[17] M. W. P. Savelsbergh. Preprocessing and probing for mixed integer programming problems. ORSA *Journal on Computing, 6,* 445-454, 1994.

[18] H. M. Sheini and K. A. Sakahallah. Pueblo: A Hybrid Pseudo-Boolean SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2: 163-183, 2006.

[19] C. Sinz. Towards an Optimal CNF encoding of Boolean Cardinality Constraints. In *Proc. of 11th Intl. Conf. on Principles and Practices of Constraint Programming (CP 2005)*, pages 827-831, Sitges, Spain, October 2005.

[20] J. P. Warners. A Linear-time transformation of linear inequalities into conjunctive normal form. In *Information Processing Letters*, 68(2), pp. 63-69, 1998.

[21] L. Zhang, D. Li & H.Shen. A SAT Based Scheduler for Tournament Schedules. *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing*, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings.

[22] L. Zhang, S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *Proc. CAV'02, number 2404 in LNCS*, pages 17-36. Springer, 2002.

[23] Satz: http://www.laria.u-picardie.fr/~cli/EnglishPage.html

[24] SAT 2005 Competition: http://www.satcompetition.org/2005/

[25] PB 2005 Competition: http://www.cril.univ-artois.fr/PB05/

[26] Resolution (logic):
http://en.wikipedia.org/wiki/Resolution_(logic)

Appendix A

Phase transition images

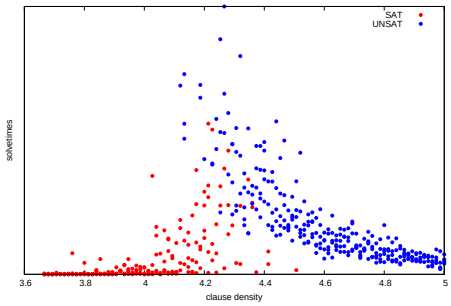

Figure A.1: Phase transition for 3,1-rand

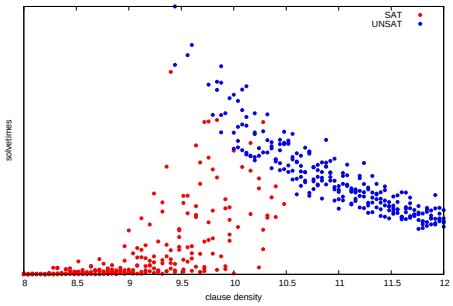

Figure A.2: Phase transition for 4,1-rand

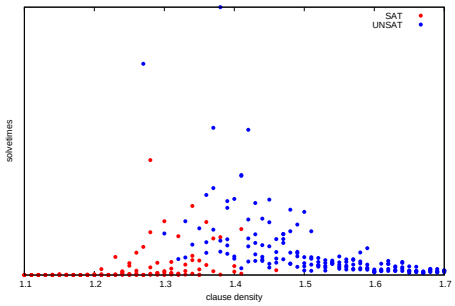

Figure A.3: Phase transition for 4,2-rand

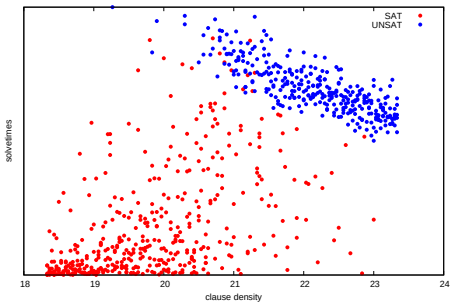

Figure A.4: Phase transition for 5,1-rand

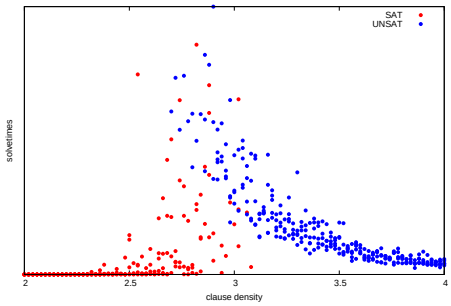

Figure A.5: Phase transition for 5,2-rand

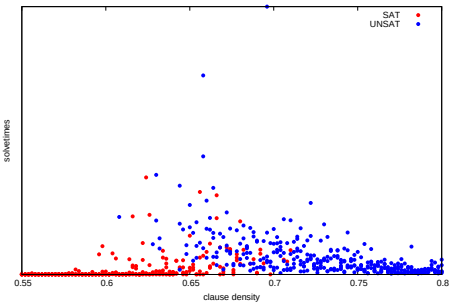

Figure A.6: Phase transition for 5,3-rand

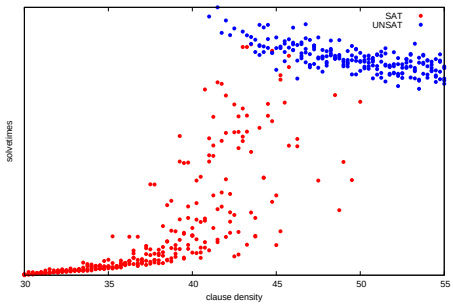

Figure A.7: Phase transition for 6,1-rand

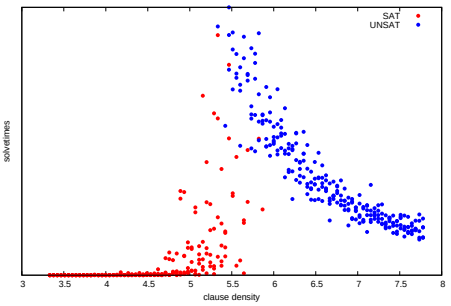

Figure A.8: Phase transition for 6,2-rand

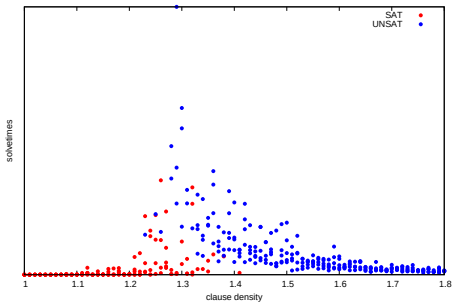

Figure A.9: Phase transition for 6,3-rand

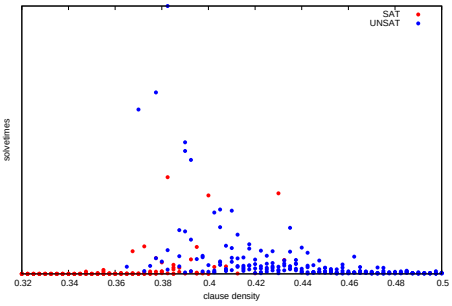

Figure A.10: Phase transition for 6,4-rand

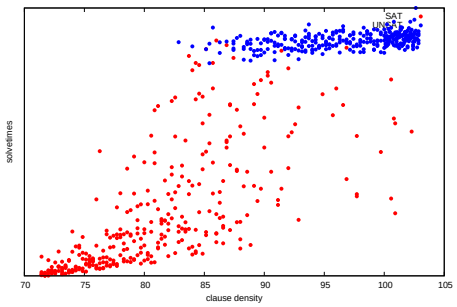

Figure A.11: Phase transition for 7,1-rand

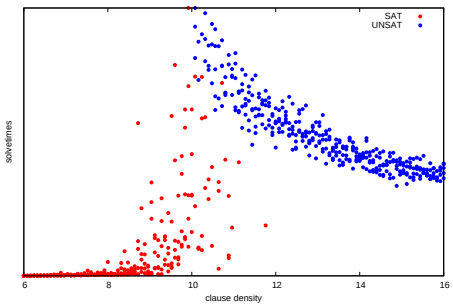

Figure A.12: Phase transition for 7,2-rand

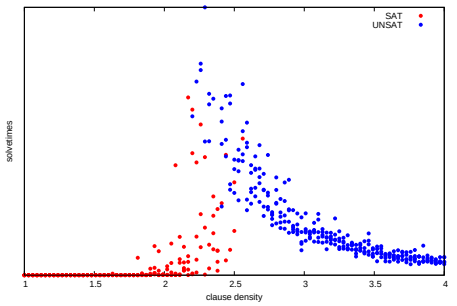

Figure A.13: Phase transition for 7,3-rand

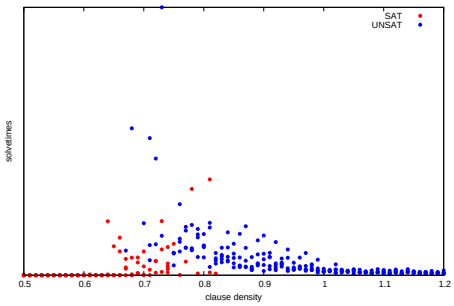

Figure A.14: Phase transition for 7,4-rand

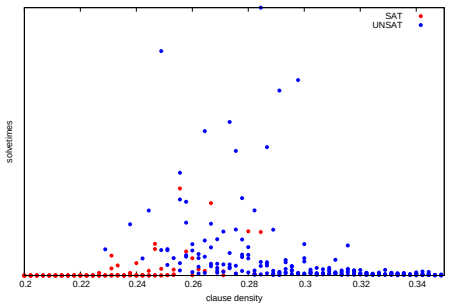

Figure A.15: Phase transition for 7,5-rand