SDP-based Max-2-Sat Decomposition

Thomas Frederik Lotgering

SDP-based Max-2-Sat Decomposition

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Thomas Frederik Lotgering
born in Rotterdam, the Netherlands

Algorithmics Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

SDP-based Max-2-Sat Decomposition

Author: Thomas Frederik Lotgering
Student id: 1183052
Email: tlotgering@gmail.com

Abstract

The Max-Sat problem has been intensively studied during the past few decades. Semi-definite programming based approximation algorithms provide good approximation ratios and polynomial runtime solutions to this problem. Unfortunately the high degree of their polynomial runtime prevents their application to problems with a large number of variables. We've investigated the possibility of decomposing Max-Sat problems to reduce the number of variables in the problems supplied to SDP solvers. We've considered several forms of decomposition and evaluated these empirically. The methods we investigated were able to approach the lower bounds of SDP within 2% but do not provide performance guarantees and cannot compete with conventional local search methods.

Thesis Committee:

Chair:	Prof. Dr. C. Witteveen, Faculty EEMCS, TU Delft
University supervisor:	Dr. H. van Maaren, Faculty EEMCS, TU Delft
Committee Member:	Dr. J. Fokkink, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

The Boolean satisfiability problem, or SAT for short, is the problem of finding an assignment to the variables of a Boolean formula such that the formula is satisfied. A Boolean variable can be either true or false, and a Boolean formula consists of Boolean variables and logical operators. Using the $\vee$ symbol to denote logical disjunction ("or"), the $\wedge$ symbol to denote logical conjunction ("and") and the $\neg$ symbol to denote negation, an example of a Boolean formula is:

$$(x \vee y) \wedge (y \vee z) \wedge (\neg y \vee \neg z) \wedge (\neg x \vee \neg y)$$

This formula is in "conjunctive normal form" (or "CNF"), because it is a conjunction of disjunctions. In such a formula, every disjunction is called a clause. A Boolean formula is said to be satisfied if it evaluates to true. For a formula in CNF, this requires that every individual clause evaluates to true. For the previous example, this could be achieved by the following truth assignment: x = true, y = false, z = true.

Now consider the following example of a formula with two variables only:

$$(x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$$

This formula cannot be satisfied, because it requires that both x and y are true, and that x is not true or y is not true, which is a contradiction. The next question one might ask is, if a formula cannot be satisfied, through what variable assignment can we satisfy as many clauses as possible, and how many clauses are we then able to satisfy? This is called the Maximum-satisfiability problem, or Max-Sat. For the previous formula, the maximum number of clauses that can be satisfied is 3.

Although simple for such a small example, it is very hard to solve this problem for more complicated formulas. For a formula with n variables, there are 2^n possible true/false assignments, so if a problem has 69 variables, there are 2^{69} possible assignments. The universe is roughly 2^{69} milliseconds old [1] so if one were to try one assignment every millisecond, one should've started at the dawn of time to have been able to try all assignments for such a problem. Since there are many practical problems that require Max-Sat problems like this to be solved, and simply trying all 2^n variable assignments takes too long, we'd like a more efficient *algorithm* for solving this problem. Unfortunately, it can be proven that the

Satisfiability problem is "NP-complete" which means that even though it is very simple to verify if a solution is correct (it is easy to test if an assignment satisfies a formula) it is very likely to be impossible to create an algorithm that is able to find the *optimal* solution for *any* given formula in reasonable time, and it can also be proven that the Maximum-Satisfiability problem is "NP-hard" meaning it is at least as hard.

There does exist an algorithm for Max-Sat however that gives a solution that is guaranteed, for any given formula, to be *almost* optimal and is able to find an answer in reasonable time. The time it takes for this algorithm to solve a problem with n variables corresponds to $n^{3\frac{1}{2}}$ (instead of 2^n for trying all assignments, which is a huge improvement)! Unfortunately $n^{3\frac{1}{2}}$ still grows too fast to be practical for large problems. This algorithm is based on "Semidefinite programming", or "SDP".

1.0.1 Research Motivation

SDP is the only natural Max-Sat algorithm providing an upper bound and lower bound, and is able to provide excellent approximation guarantees and guaranteed polynomial solve time as well as insensitivity to the number of clauses (in the Max-2-Sat case), but despite these very useful properties it is not a popular method. We hoped to find a way to increase the relevance of SDP. Much research has been put, and is still ongoing, into improving SDP solvers – mostly by researchers with a background in applied mathematics. We hoped that by approaching the problem from "another angle" as it were, by treating SDP as a "black box" and applying computer science techniques, we might uncover useful new approaches.

1.0.2 Research Questions

To what extent is semidefinite programming relevant to solving maximum-satisfiability problems compared to current alternatives?

Is it possible to enhance the relevance of semidefinite programming in solving maximum-satisfiability problems by decomposing those problems, while treating semidefinite programming as black-box?

1.0.3 Report Outline

This chapter, as well as chapters 2 and 3 serve as introduction. Chapter 2 describes the maximum-satisfiability problem and some of the algorithms used to solve it. Chapter 3 describes semidefinite programming, how it can be used to solve maximum-satisfiability problems, and the software used to solve SDP problems.

Chapter 4 introduces two decomposition methods we've investigated. Chapter 5 describes the first method in more detail, and presents experimental results and a conclusion regarding that method. Chapter 6 describes the second method in more detail, and describes the various variations on that method that we've experimented with. Chapter 7 describes various algorithms for solving problems decomposed using the methods described in chapter 6, and chapter 8 describes the impact that various methods of hyper-plane rounding have when using these algorithms.

Chapter 9 describes the empirical evaluation methods we've used to evaluate SDP and our decomposition methods. Chapter 10 presents the results of this empirical evaluation regarding the decomposition method, and chapter 11 compares our results to alternative max-sat solvers. The conclusion is presented in chapter 12.

Chapter 2

Maximum Satisfiability Problem

This chapter describes the maximum-satisfiability problem and some of the algorithms used to solve it.

2.1 Definition

The Maximum Satisfiability problem can be defined as follows:

Given an instance consisting of a set U of Boolean variables, and a collection C of disjunctive clauses of literals, where a literal is a variable or a negated variable in U. The problem is to find a truth assignment for U such that the number of clauses satisfied by the truth assignment is maximized.

Additionally, we can define the *weighted* Maximum Satisfiability problem as:

Given an instance consisting of a set U of Boolean variables, a collection C of disjunctive clauses of literals, and a set W of weights for every clause in C. The problem is to find a truth assignment for U such that the sum of the weights of all clauses satisfied by the truth assignment is maximized.

There is also the (weighted) Maximum k-Satisfiability problem, which is identical to the (weighted) Maximum Satisfiability problem, except all clauses in C can consist of at most k literals.

The (weighted) Maximum (k-) Satisfiability problem is NP-hard (non-deterministic polynomial-time hard).

The algorithms that try to solve such problems can be grouped in 3 categories. The complete algorithms (solvers) always give an optimal solution, but can (depending on the problem) take a (very) long time to do so. The incomplete algorithms return an answer that's often very good, and only need a limited amount of time to do so, but are unable to guarantee that the solution is optimal or even close to optimal. Finally the approximation algorithms are able to return an answer that is guaranteed to be close to optimal, and can (guarantee to) do so in reasonable time.

2.2 Complete Solvers

An example of a good complete solver is MSUnCore2. We've used this solver to gather optimal solutions to some of our benchmarks, and for comparing the various types of solvers since its behavior is typical for most complete Max-Sat solvers.

2.2.1 MSUnCore description

MSUnCore (Maximum Satisfiability with UNsatisfiable COREs) is an unsatisfiability-based Max-Sat solver. The unsatisfiability-based solver approach consists of identifying unsatisfiable sub-formulas, relaxing each clause in each unsatisfiable sub-formula, and adding a new constraint requiring exactly one relaxation variable to be relaxed in each unsatisfiable sub-formula. MSUnCore uses PicoSAT, a Conflict-Driven Clause Learning SAT solver, for the iterative identification of unsatisfiable sub-formulas [2].

Unsatisfiability based solvers use the ability of (certain) SAT solvers to provide a certificate of unsatisfiability in the form of an unsatisfiable core (sub-formula) to solve the Max-SAT problem optimally. Unsatisfiability-based algorithms start by using an underlying SAT solver to solve the problem. If the solver returns SATISFIABLE, the SAT solution is returned as the Max-SAT solution. If the solver returns UNSATISFIABLE, the algorithm will relax the clauses in the unsatisfiable core returned by the SAT solver by adding relaxation variables which, when set to true, satisfy the clause. This process is repeated until the problem is satisfied. The resulting solution can be shown to be optimal, since each relaxed clause eliminates at most one UNSAT core [3].

2.2.2 MSUnCore Performance

The performance of MSUnCore2 is determined by the problem's complexity, rather than the number of variables. For Max-2-Sat problems, complexity increases with density (number of clauses / number of variables) and therefore MSUnCore2 can solve low-density problems with a (fairly) high number of variables very rapidly but MSUnCore2 struggles with high-density problems, even running out of memory (8GB) at 60 variables and density 3.0, though it is sometimes still able to solve problems with 60 variables at density 3.5 in under a second (see Figure 2.1 and Figure 2.2).

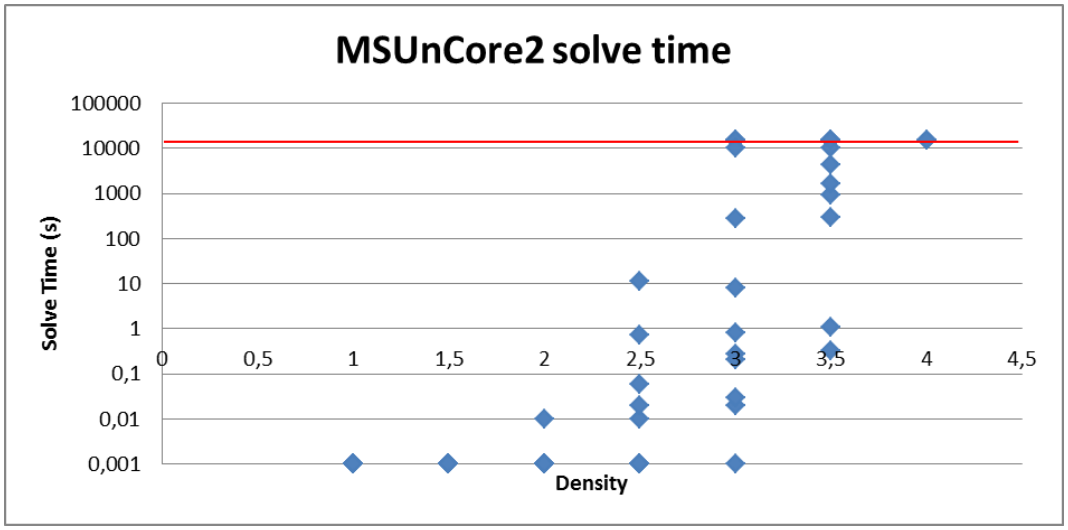


Figure 2.1: MSUnCore solve times per density for N=60.

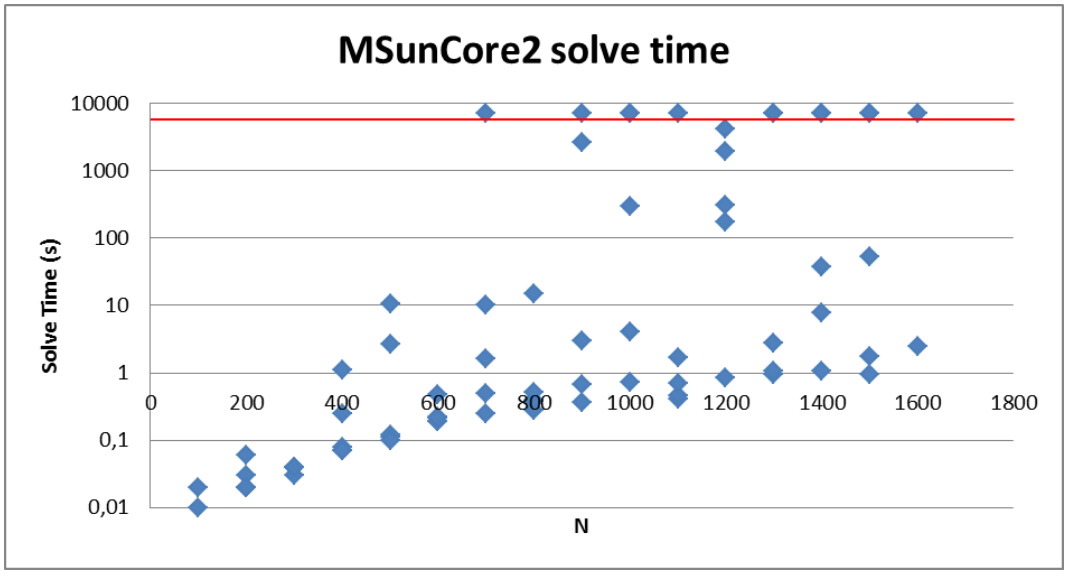


Figure 2.2: MSUnCore solve times per N for density 2.5.

2.3 Incomplete Solvers

A local search solver was used for comparison as well. The solver used was the UBCSAT implementation of Scaling and Probabilistic Smoothing (SAPS).

2.3.1 SAPS Description

SAPS is a Stochastic Local Search (SLS) algorithm. An SLS solver is incomplete: it does not guarantee that its solution is optimal, but it can often find a good solution effectively.

The general strategy of an SLS solver is to start with an initial complete assignment of truth values to all variables in the given formula and in each step flip the truth assignment of one variable, with the purpose of minimizing an objective function: typically the number of clauses unsatisfied by the variable assignment.

SAPS uses an SLS method called Dynamic Local Search (DLS). DLS strategies are based on the idea of modifying the evaluation function in order to prevent the search from getting stuck in local minima. They typically associate weights with the clauses of the given formula, which are modified during the search process, and the objective becomes to minimize the total weight, rather than simply the number of falsified clauses. There are several variants of this scheme, including the Exponentiated Sub-Gradient method (ESG), which SAPS is closely related to.

ESG for SAT works as follows: The search is initialized by randomly chosen truth values for all propositional variables in the input formula, F, and by setting the weight associated with each clause in F to one. Then, in each iteration, a weighted search phase followed by a weight update is performed. During the weighted search phase a series of greedy variable flips are performed on variables selected at random from the set of all variables that appear in currently unsatisfied clauses. The clause weights are updated after the search phase based on their satisfaction status. The algorithm terminates when a solution is found or a maximal number of iterations (determined by the cutoff) have been performed. Although not running in polynomial time generally, cutoff parameters can therefore be set to do so.

SAPS differs from ESG in the way it implements weight updates: SAPS performs the computationally expensive weight smoothing probabilistically and less frequently than ESG, leading to a reduction in the time complexity of the weight update procedure without increasing the number of variable flips required for solving a given SAT instance [4].

2.3.2 SAPS Performance

The quality of SAPS' solutions, and the time required to arrive at them, is primarily determined by its cutoff parameter, which determines the number of iterations it has available to search for a solution.

Since SAPS can only guarantee optimality when it is able to satisfy a problem formula, on unsatisfiable instances it will keep searching for a better solution until a specified cutoff (for example: 100.000) on the number of rounds is reached. For this reason it is hard to get an accurate estimate of the minimal amount of time required by SAPS to arrive at the solutions it finds, if the problem is unsatisfiable. For a cutoff of 1.000.000, solve time is in the order of seconds.

Figure 2.3 shows how SAPS' solution quality is improved with an increase in cutoff. It also shows that small problems are solved optimally, and larger problems are solved near-optimally.

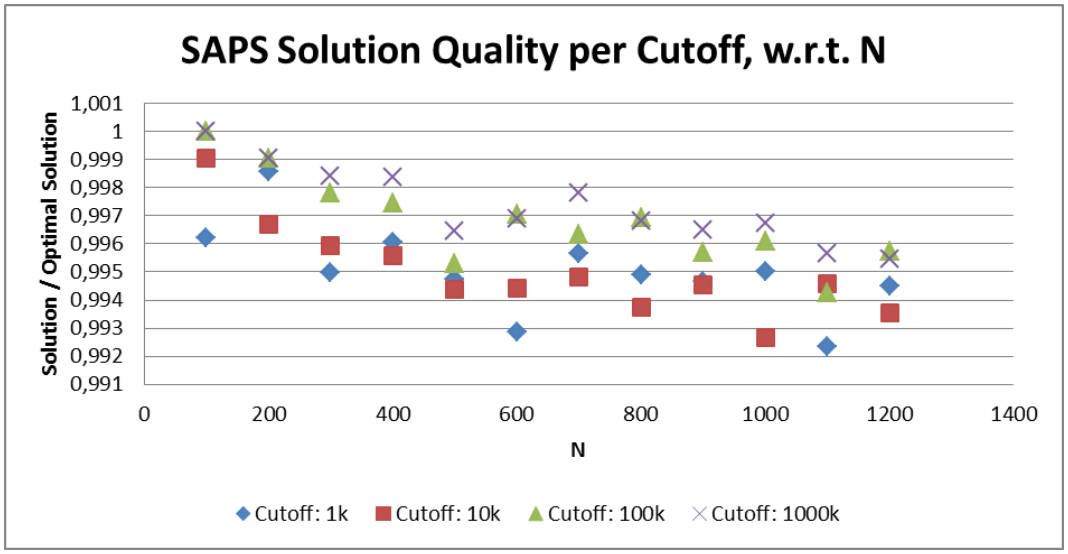


Figure 2.3: Averaged SAPS Solution Quality. Uniform random problems, density 2.5.

2.4 Approximation Algorithms

Approximation algorithms guarantee their solution to be within a certain ratio of the optimal solution. A trivial Max-k-Sat approximation algorithm is the following:

Randomized MAX-SAT

Choose a random assignment r.
Return r.

This remarkably simple algorithm has an expected performance ratio of $1 - 2^{-k}$. For Max-2-Sat, this gives an approximation ratio of .75. An approximation algorithm for Max-2-Sat based on Semidefinite programming with a superior approximation ratio will be discussed in detail in the next chapter.

Chapter 3

SDP

This chapter describes semidefinite programming, how it can be used to solve maximum-satisfiability problems, and the software used to solve SDP problems.

Semidefinite programming is a class of convex optimization problems with a symmetric matrix variable where a linear objective function is optimized under linear constraints, and under the constraint that the matrix variable is positive semidefinite. These problems can be solved efficiently using interior point methods [5].

3.1 Definition SDP

Let $\mathcal{S}^n$ be the space of real symmetric $n \times n$ matrices. A matrix A is positive semidefinite when:

$$y^T Ay \geq 0 \; for\; all\; y \in \Re^n$$

Here, y is a non-zero real vector of dimension n and y^T is its transpose. Let $\mathcal{S}^n_+$ be the set of positive semidefinite matrices, and let $A \succeq 0$ denote that $A \in \mathcal{S}^n_+$.

The standard form of an SDP problem is:

$\max C \bullet X$
subject to:
$A_i X = b_i$ for $i = 1 \ldots m$
$X \succeq 0$

In this definition X is a variable and $X \in \mathcal{S}^n$, and this definition uses the scalar product (where $tr(M)$ is the trace of the square matrix M, which is the sum of the elements on its main diagonal):

$$U \bullet V = tr(U^T V)$$

Every SDP problem of this form has an equivalent dual problem of the form:

$\min b^T y$

subject to:

$$Z = \sum_{i=1}^{m} y_i A_i - C$$
$$Z \succeq 0$$

However, SDP problems are often defined in terms of linear expressions on the dot products of vectors. This is possible because for any positive semidefinite matrix X there exists [6] a set of vectors $B = \{b_1, \ldots, b_n\}$ such that:

$$X_{i,j} = b_i \cdot b_j$$
$$i.e. : X = B^T B$$

Such a matrix B can be computed efficiently, for example by using an *incomplete* Cholesky decomposition in $O(n^3)$ [7] (incomplete because the matrix X is constrained to be positive semidefinite, instead of positive definite which is required for Cholesky decomposition) or by using the singular value decomposition. Matrix B has dimensions $n \times r$, where $1 \leq r \leq n$ is the *rank* of X.

3.2 SDP Encodings for Max-2-Sat

To solve a Max-Sat problem using SDP requires the Max-Sat problem to be *encoded* as an SDP problem.

We've chosen to restrict our research to the Max-2-Sat problem (which unlike 2-Satisfiability is still an NP-hard problem [8]), rather than the more general Max-Sat problem, because it allows a more elegant SDP encoding.

There are several encodings for Max-2-Sat. The earliest efficient SDP encoding for Max-2-sat was contributed by Goemans and Williamson in 1995 which had an approximation ratio of 0.87856 [9], which was then improved upon by adding triangle inequalities to obtain a better approximation ratio of 0.931 by [10]. We based our work on the original encoding without triangle inequalities.

3.2.1 Goemans-Williamson encoding

Goemans and Williamson define their Max-2-Sat SDP encoding as follows (see [9]). *For every Boolean variable x_i, introduce a variable $y_i \in \{-1,1\}$, as well as a homogenous variable $y_0 \in \{-1,1\}$ whose function is to determine whether -1 or 1 corresponds to true. A variable x_i is true if $y_i = y_0$ and false otherwise. Given a Boolean formula Φ, define the value $v(\Phi) = 1$ if the formula is true and 0 if the formula is false.* This can be implemented for each clause using the following quadratic formulae:

$$v(x_i) = \frac{1 + y_0 y_i}{2}$$
$$v(\overline{x_i}) = 1 - v(x_i) = \frac{1 - y_0 y_i}{2}$$

	$\frac{1}{4}(\mathbf{3} \pm_{\mathbf{x}} \mathbf{M_{0,x}} \pm_{\mathbf{y}} \mathbf{M_{0,y}} \mp_{\mathbf{x}y} \mathbf{M_{x,y}})$
$(\overline{\mathbf{x}} \vee \overline{\mathbf{y}})$	$\frac{1}{4}(\mathbf{3} - \mathbf{M_{0,x}} - \mathbf{M_{0,y}} - \mathbf{M_{x,y}})$
$(\overline{\mathbf{x}} \vee \mathbf{y})$	$\frac{1}{4}(\mathbf{3} - \mathbf{M_{0,x}} + \mathbf{M_{0,y}} + \mathbf{M_{x,y}})$
$(\mathbf{x} \vee \overline{\mathbf{y}})$	$\frac{1}{4}(\mathbf{3} + \mathbf{M_{0,x}} - \mathbf{M_{0,y}} + \mathbf{M_{x,y}})$
$(\mathbf{x} \vee \mathbf{y})$	$\frac{1}{4}(\mathbf{3} + \mathbf{M_{0,x}} + \mathbf{M_{0,y}} - \mathbf{M_{x,y}})$

Table 3.1: Clause formulas.

$$v(x_i \vee x_j) = 1 - v(\overline{x_i} \wedge \overline{x_j}) = 1 - \frac{1 - y_0 y_i}{2} * \frac{1 - y_0 y_j}{2} = \frac{1}{4}(3 + y_0 y_i + y_0 y_j - y_0^2 y_i y_j)$$

Clauses with negated terms can be translated similarly, such that any clause can be described using

$$v(C) = \frac{1}{4} * (3 \pm_i y_o y_i \pm_j y_0 y_j \mp_{ij} y_i y_j)$$

Where $\pm_i, \pm_j$ are $+$ if variable i resp. j occur positively in the clause, and $--$ otherwise; $\mp_{ij}$ is $-$ if i and j occur both positively or negatively, and $+$ otherwise (see Table 3.1).

This way a (weighted) Max-2-Sat problem Φ can be encoded as the following integer quadratic program:

Maximize $\sum_{C_j \in \Phi} w_j v(C_j)$
subject to:
$y_i \in \{1, 1\}$

Which can be expressed equivalently using symmetric matrices as:

Maximize $\sum_{C_j \in \Phi} w_j * \frac{1}{4} * (3 \pm_i M_{0,i} \pm_j M_{0,j} \mp_{ij} M_{i,j})$
subhect to:
$rank(X) = 1$, $diag(M) = 1^{n+1}$

The problem space of this problem consists of an exponential number of points and solving it is NP-hard since it equals the Max-2-Sat problem. However, it can be relaxed to an SDP relaxation by replacing the $rank(X) = 1$ constraint with the constraint that $X \succeq 0$:

Maximize $\sum_{C_j \in \Phi} w_j * \frac{1}{4} * (3 \pm_i M_{0,i} \pm_j M_{0,j} \mp_{ij} M_{i,j})$
subhect to:
$X \succeq 0$, $diag(M) = 1^{n+1}$

The problem space of this SDP relaxation is convex [6] and can, because it is an SDP relaxation, be solved efficiently using interior point methods.

The value of the objective function of this SDP relaxation provides an *upper bound* on the optimal value of Φ which can (in combination with lower bounds) be used by a branch & bound algorithm to solve Φ exactly. The solution of the SDP relaxation can also be used

to derive a variable assignment to the Boolean variables and a corresponding *lower bound* for Φ using *hyper-plane rounding*. It can be shown that the expected total weight of the satisfied clauses is at least 0.87856 times the optimal value of the SDP relaxation [9], and thus the quality of both upper and lower bounds is guaranteed to fall within that margin.

For the integer quadratic programming problem, the value of a Boolean variable x_i is defined using the homogenous variable y_0 as x_i is true if $y_i = y_0$ and false otherwise. A similar comparison can be made for the solution to the SDP relaxation. Given the decomposition $X = B^T B$, where Matrix B has dimensions $n \times r$, where $1 \leq r \leq n$ is the *rank* of X, then the i^{th} row B_i of B corresponds to the variable y_i and each row corresponds to a pole on an r dimensional hyper-sphere. Using a random unit-length vector $\bar{r}$ of dimension r, the value of Boolean variable x_i can be defined as x_i is true if $sign(B_i \cdot \bar{r}) = sign(B_0 \cdot \bar{r})$ and false otherwise. In other words, x_i is true if and only if y_i and y_0 lie on the same side of the hyperplane defined by $\bar{r}$. A single step of the hyper-plane rounding algorithm consists of rounding all poles in this manner using a single random pole. This random step should be repeated several times, with every iteration yielding a lower bound, to arrive at the final lower bound, which is the maximum of all found lower bounds.

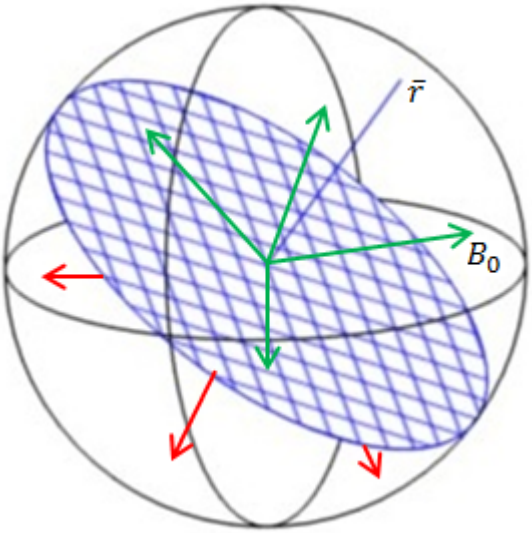
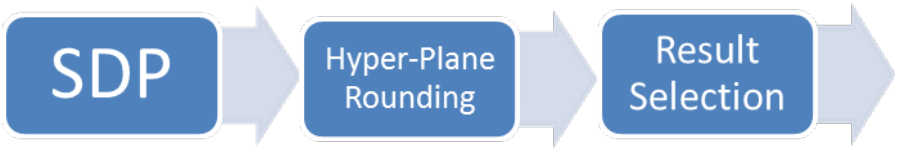


Figure 3.1: Illustration of hyper-plane rounding if r=3. Green arrows indicate variables set to true, red indicates variables set to false.

The process of deriving a lower bound using an SDP relaxation can be summarized using the following flow diagram. First, the problem is encoded as SDP relaxation, which is then solved using an SDP solver and the resulting matrix is decomposed using, for example, Cholesky- or Singular-Value-Decomposition. Then, for a fixed number of iterations, lower bounds are derived by evaluating the formula with assignments created by rounding the rows of the decomposed matrix. Finally, the maximum of the found lower bounds is selected as solution. The runtime complexity of this approach is $O(n^{3\frac{1}{2}} + Tn + Tm)$, where T is the number of hyper-plane roundings, n the number of variables, and m the number of clauses.



3.3 SDP Solvers

Almost all modern SDP solvers are based on interior point methods. Interior point algorithms solve convex optimization problems by traversing the interior of the feasible region (in contrast to the well-known simplex algorithm for linear (convex) optimization, which is not applied to SDP, which traverses the edges of the feasible region). Essentially the algorithm repeatedly chooses a direction to move toward, and distance to move in that direction, such that it converges toward the optimal solution. Convergence is guaranteed to be polynomially bounded in the dimension and accuracy of the solution. There are several variants of interior point methods, including over 20 different ways to compute the search direction [11].

We've investigated and performed experiments with the following three SDP solvers. They are all based on interior point methods and they all converge in $O(\sqrt{n}\log e)$ iterations [5]. For each solver, each iteration involves computing a Schur complement matrix and its Cholesky factorization [12] which takes $O(n^3)$ [7] and dominates the complexity of each iteration, thus bringing the total complexity of all solvers up to $O\left(\sqrt{n} * n^3\right) = O(n^{3\frac{1}{2}})$.

3.3.1 SeDuMi

SeDuMi is a popular solver based on the primal-dual algorithm [13]. The pseudo-code of a generic primal-dual algorithm is given below, where a tuple (x, λ, s) determines a search direction, and (system) is a set of matrix equations based on the problem and a specific search direction type.

Primal-dual algorithm:

Given (x^0, λ^0, s^0) with $\left(x^0, s^0\right) > 0$
Set $k \leftarrow 0$ and $\mu^0 = \left(x^0\right)^T s^0 / n$
repeat
  Choose σ_k and r^k
  Solve (system) with $(x, \lambda, s) = (x^k, \lambda^k, s^k)$ and $(\mu, \sigma, r) = (\mu_k, \sigma_k, r^k)$ to obtain $(\triangle x^k, \triangle \lambda^k, \triangle s^k)$
  Choose step length $\alpha_k \in (0, 1]$ and set $\left(x^{k+1}, \lambda^{k+1}, s^{k+1}\right) \leftarrow \left(x^k, \lambda^k, s^k\right) + \alpha_k (\triangle x^k, \triangle \lambda^k, \triangle s^k)$
  Set $\mu_{k+1} \leftarrow \left(x^{k+1}\right)^T s^{k+1} / n$ and $k \leftarrow k+1$
until some termination test is satisfied

3.3.2 SDPT3

SDPT3 employs a predictor-corrector infeasible primal-dual path-following method. In each iteration, it first computes a *predictor* search direction aimed at decreasing the duality

gap as much as possible, followed by a *corrector* step which is intended to keep the iterates close to the central path. [12]

The pseudo-code of a predictor-corrector algorithm is as follows, where a tuple (x, λ, s) determines a search direction, and (system) is a set of matrix equations based on the problem and a specific search direction type.

Predictor-corrector algorithm:
Given (x^0, λ^0, s^0) with $\left(x^0, s^0\right) > 0$, $\|Xs - \mu e\|_2 \leq \beta$
Set $k \leftarrow 0$ and $\mu^0 = \left(x^0\right)^T s^0 / n$
repeat
Set $(x\lambda, s) = (x^k, \lambda^k, s^k)$ and $(\mu, \sigma, r) = (\mu_k, 0, 0)$
Solve (system) and set $(u, w, v) \leftarrow (\triangle x, \triangle\lambda, \triangle s)$ to obtain $(\triangle x^k, \triangle\lambda^k, \triangle s^k)$
Choose step length α_k as the largest $\alpha \in (0, 1]$ such that $(x, \lambda, s) + \alpha(u, w, v)$ has $(x, s) > 0, \|Xs - \mu e\|_2 \leq \beta$
Set $(x, \lambda, s) \leftarrow (x, \lambda, s) + \alpha_k(u, w, v)$ and $(\mu, \sigma, r) \leftarrow (\mu_k, (1 - \alpha_k), 0)$
Solve (system) and set $\left(x^{k+1}, \lambda^{k+1}, s^{k+1}\right) \leftarrow (x, \lambda, s) + \alpha_k(\triangle x, ?\lambda, ?s)$
Set $\mu_{k+1} \leftarrow \left(x^{k+1}\right)^T s^{k+1} / n$ and $k \leftarrow k+1$
until some termination test is satisfied

3.3.3 DSDP

The DSDP solver is an implementation of a dual scaling algorithm [14]. Unlike primal-dual interior point methods such as the previous two solvers, this algorithm uses only the dual solution to generate a step direction. It initially computes search directions by solving the Schur complement equations by using the conjugate residual method, and then switches to the Cholesky method when the conditioning of the matrix worsens [15]. The conjugate residual method is an iterative method that can be applied efficiently to sparse systems that are too large to be handled by direct methods such as the Cholesky decomposition [16].

3.4 Performance SDP Encoding

SDP is a relevant approach because it has several attractive features. First of all, it is guaranteed to terminate in polynomial time. This is what makes it attractive as approximation algorithm for an NP-hard problem. Additionally, it is not affected significantly by the number of clauses or density, in either effectiveness or efficiency. However its runtime complexity $O(n^{3\frac{1}{2}})$ is of a sufficiently high degree to make this approach infeasible on current hardware for larger problems.

These properties are illustrated by the following results. Figure 3.2 shows that the solve time of Sedumi and SDPT3 does not depend on problem density, and DSDP depends on density only sub-linearly. Figure 3.3 shows that SDPT3 and DSDP significantly outperform Sedumi, but that Sedumi (Figure 3.3) as well as SDPT3 and DSDP (Figure 3.4) have solve times that correspond to a polynomial of degree less than 4.

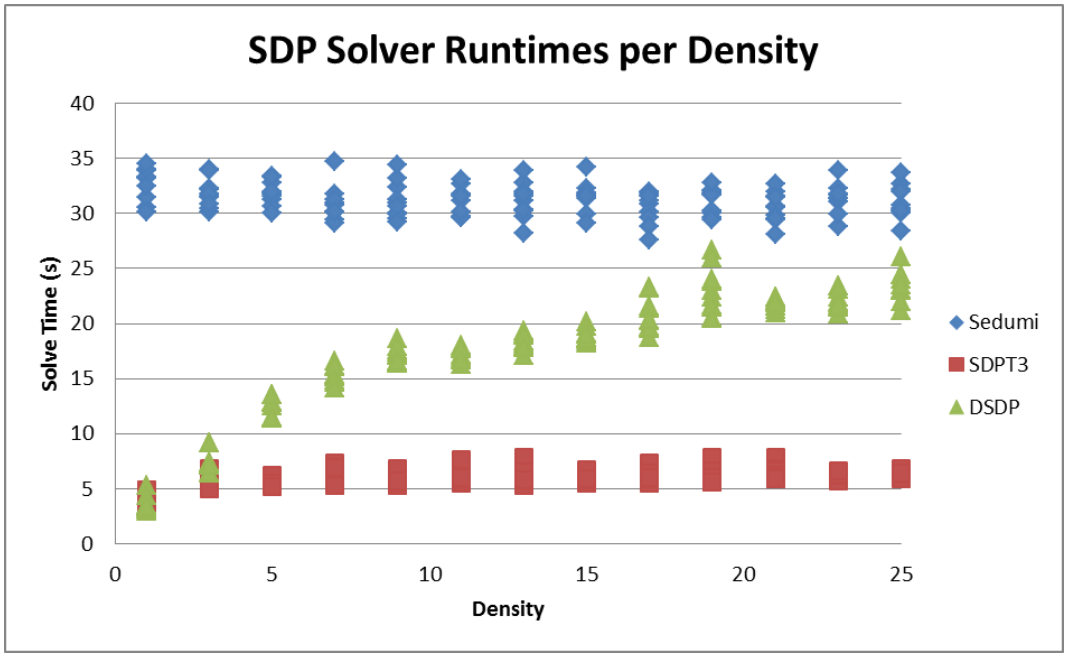


Figure 3.2: Solve time of various SDP solvers w.r.t. max-sat density. Uniform random problems, N=400.

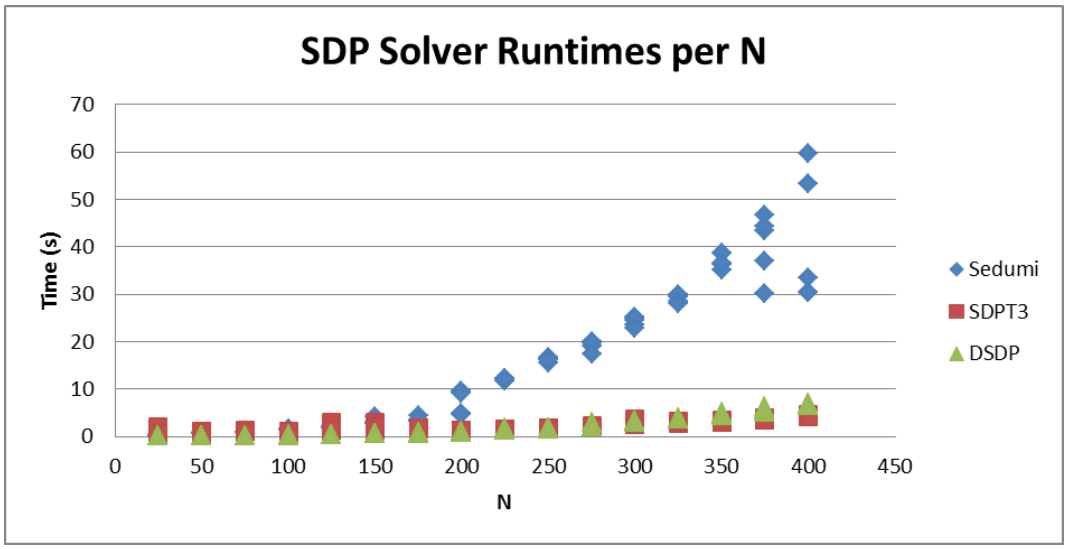


Figure 3.3: SDP solve times by number of variables N. Uniform random problems with density = 4.0.

3.5 Matlab Implementation

All experiments in our research relating to SDP were performed using Matlab, and with limited numerical accuracy. Although not particularly troublesome for our work, it did

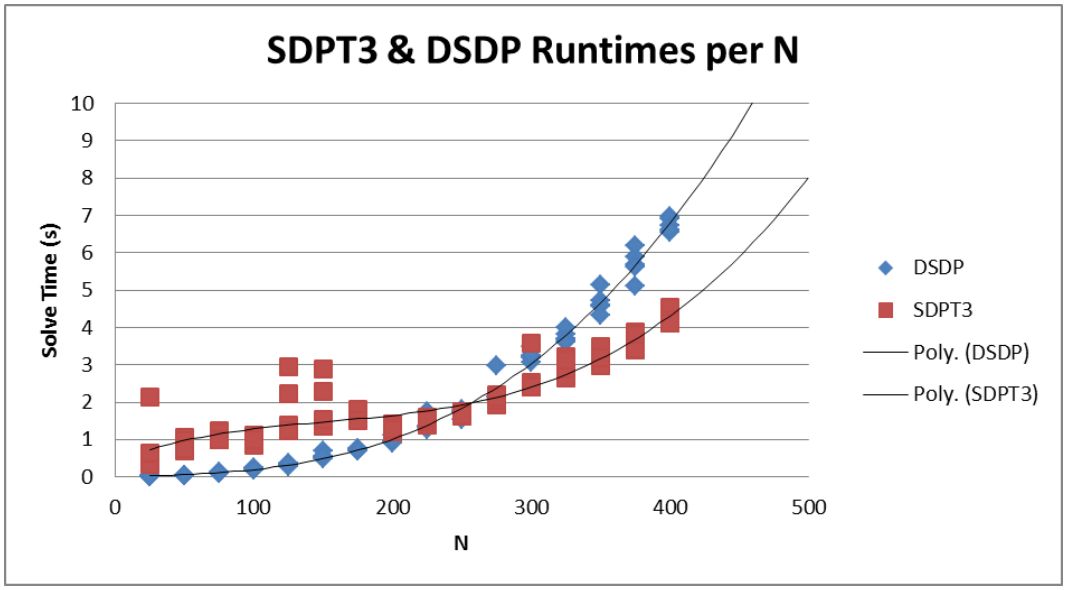


Figure 3.4: Solve time and 4th degree polynomial w.r.t. N. Uniform random problems with density 4.0.

entail some consequences. In particular the decomposition $X = B^T B$ of the matrix resulting from SDP, as part of the hyper-plane rounding procedure, has limited accuracy, such that the values of $B^T B$ deviate from X by less than $1e^{-7}$ and, more importantly, the number of columns of B (which correspond to the rank of X) is inflated with many columns containing only negligible values (less than $1e^{-5}$). The inflated rank was often close to or as high as the number of variables, while the "true" rank (the number of columns with significant values) was much lower, typically 2 or 3.

Chapter 4

Max-2-Sat Decomposition

This chapter introduces two decomposition methods we've investigated.

Our motivation for applying decomposition is based on two observations. First, that the runtime of SDP solvers is predominantly determined by the number of variables and therefore by reducing the size of the problems supplied to an SDP solver, we can hope to accelerate the process. Second, the intuition that for many Max-Sat instances, it is possible to assign a number of variables without over-constraining the problem, so it is possible to assign these early on and still achieve reasonable results.

Since Max-Sat is an NP-hard problem, it is generally impossible to solve small parts of the problem such that their solutions can be combined to optimally solve the whole problem. Our intention was therefore to sacrifice (guaranteed) solution quality in favor of speed.

As the results will show, for some problems and some decompositions the solution quality penalty is high, but in other cases it provides a significant speedup at a cost of only a few unsatisfied clauses. When reading the remainder of this chapter, the reader may notice that the expected speedup provided by decomposition into only two parts is low, and might wonder why one would bother decomposing at all for such low speedup. Please keep in mind that the intention behind these decompositions was to ultimately arrive at some recursive or iterative scheme that would decompose a problem into several smaller parts, thus allowing for greater speedup.

We've investigated two kinds of decomposition: clause-based and variable-based decomposition.

4.1 Clause-Based Decomposition

Given a CNF formula Φ consisting of the conjunction of clauses $C_1 \wedge \cdots \wedge C_m$ we create a dichotomy of clauses $\Phi_1 = C_1 \wedge \cdots \wedge C_{m/2}$ and $\Phi_2 = C_{m/2+1} \wedge \cdots \wedge C_m$.

We can then calculate an *upper bound* for Φ as the sum of the upper bounds of Φ_1, Φ_2 which are obtained using an SDP solver.

The analysis and results of this approach are presented in chapter 5.

4.2 Variable-Based Decomposition

Given a problem as CNF formula Φ consisting of variables $U = u_1 \dots u_n$ and m clauses, we can create two (or more) subsets of variables $U_1, U_2 \subset U$, such that $U_1 \cup U_2 = U$. For example:

$\mathbf{u_1}$	$\dots$	$\mathbf{u_{n/2}}$	$\mathbf{u_{n/2+1}}$	$\dots$	$\mathbf{u_n}$
U_1			U_2		

Given such a separation of variables, we create two corresponding problems Φ_1, Φ_2 as follows:

$$\Phi_i = \bigwedge_{(x \vee y) \in \Phi} \begin{cases} (x \vee y) \ \textit{if } x \in U_i \textit{ and } y \in U_i \\ (x \vee c) \ \textit{if } x \in U_i \textit{ and } y \notin U_i \\ \emptyset \ \textit{otherwise} \end{cases}$$

In this definition $(x \vee c)$ is a clause which contained a term consisting of a variable that has been separated into a different sub-problem, and c is a constant for which we will assume $c = \frac{1}{2}$. Clearly we're taking some liberties with the notation when we write $(x \vee \frac{1}{2})$ in a Boolean formula. In this notation $(x \vee \frac{1}{2})$ is to be understood as the choice between either satisfying the clause $(x \vee y)$ by satisfying x or not satisfying the clause with x and instead receiving half the value of the clause. What this expresses is the choice between satisfying the clause with the variable that is part of this sub-problem, or deferring the satisfaction of the clause to the term y which depends on a variable that is not a part of this sub-problem, and c is a value that expresses *the a priori expectation* that it will be satisfied by that term.

This form of decomposition can be implemented without relying on solver-specific techniques by using the following equivalence to substitute a (weighted) clause for those clauses with a separated variable ($\frac{1}{2}(x)$ means the unary clause (x) is assigned half its original value).

$$\frac{1}{2}(x) \equiv \frac{1}{2} + \frac{1}{2}(x) = (x \vee \frac{1}{2})$$

After decomposing a problem in this manner, solving both Φ_1, Φ_2 creates a full variable assignment (since $U_1 \cup U_2 = U$) and corresponding *lower bound* for Φ.

The reason for decomposing the problem in this manner is to create smaller (in terms of number of variables) parts, which can be more efficiently solved by an SDP solver. The technique for reducing the problem size by separating variables can be performed with any (weighted) Max-Sat solver, however, though it reduces the number of variables it typically increases the density, and therefore this approach cannot be applied effectively using a complete solver like MSUnCore, whose runtime is determined by density rather than number of variables.

4.3 Variable-Based Decomposition for SDP

This approach can be implemented in the Goemans-Williamson encoding in an easy manner by fixing the separated variables at 0. This does not imply that the variable is assigned a value of false, but rather that the assignment of the variable is deferred. If a variable $u_y \in [-1, 1]$ were to be fixed at value 0, the formula for a clause consisting of a variable u_x

$\pm_{\mathbf{x}}\ \mathbf{M_{0,x}}$	$\frac{\mathbf{1}}{\mathbf{4}}(\mathbf{3} \pm_{\mathbf{x}}\ \mathbf{M_{0,x}})$
$-\mathbf{1}$	$\mathbf{1/2}$
$\mathbf{1}$	$\mathbf{1}$

Table 4.1: Value of clause with one 0'd variable.

and u_y will be:

$$\frac{1}{4}(3 \pm_x M_{0,x} \pm_y M_{0,y} \mp_{xy} M_{x,y}) = \frac{1}{4}(3 \pm_x M_{0,x} \pm_y 0 \mp_{xy} 0) = \frac{1}{4}(3 \pm_x M_{0,x})$$

This (literally) removes u_y from the equation and therefore u_y can be removed from the symmetric variable matrix, reducing its dimension by 1 per fixed variable.

A clause with a single variable "0'd" will have a value in $\{1/2, 1\}$ (see Table 4.1). If the clause is satisfied by u_x it gets value 1 as intended. If the clause is not satisfied however, using this encoding, it is still assigned a value > 0, which symbolizes the potential for the clause to be satisfied later on by the "0'd" variable. Therefore, when maximizing the sum of clause values, satisfying this clause with u_x will be less important.

Clauses where both variables get fixed to 0 will receive a value $\frac{1}{4}(3 \pm_x 0 \pm_y 0 \mp_{xy} 0) = 3/4$. These clauses can be removed from or ignored by the objective function, since constants do not affect its solution and the value of the objective function is not used to compute the value of the lower bounds.

Since the runtime $T(n)$ of the SDP solver is determined (almost) exclusively by the number of variables, and this runtime conforms to a 3^{th} degree polynomial, solving Max-Sat problems with a high number of variables is infeasible in practice. By splitting up the problem into separate parts, the runtime can be reduced, at the cost of solution quality. The potential speedup $\frac{T(n)}{2*T(n/2)}$ of solve time $T(n) = c*(n+1)^{3\frac{1}{2}}$ gained by such a decomposition is limited:

$$speedup = \frac{(n+1)^{3\frac{1}{2}}}{2*(n/2+1)^{3\frac{1}{2}}}$$

$$highest\ possible\ speedup = \lim_{n\to\infty} \frac{(n+1)^{3\frac{1}{2}}}{2*(n/2+1)^{3\frac{1}{2}}} = 2^{\frac{5}{2}} \approx 5.65$$

The severity of the impact on the solution quality and the speedup realized by applying various forms of this kind of variable-based decomposition is investigated in the main body of this report.

Chapter 5

Clause-Based Decomposition

This chapter presents our analysis of clause-based decomposition, as well as experimental results and a conclusion regarding that method.

5.1 Restrictions on Density due to Expected Speedup

The solve-time used by an SDP solver is polynomial in the number of variables, and can be approximated with the following function:

$$T = c * (N)^{3\frac{1}{2}}$$

N.B.: $T = c(N+1)^{3\frac{1}{2}}$ *would be more accurate since it accounts for the homogenizing variable.*

If the problem is separated into two equally large (in terms of variables) parts, then the total solve time can be expressed as:

$$T_{split} = 2 * c * (xN)^{3\frac{1}{2}}$$

If we hope to improve the solve-time (so $T_{split} < T$) then the average relative size of the parts, x, must be smaller than the following value:

$$T_{split} < T$$

$$2 * c * (xN)^{3\frac{1}{2}} < c * (N)^{3\frac{1}{2}}$$

$$2 * x^{3\frac{1}{2}} < 1$$

$$x < \frac{1}{2^{\frac{2}{7}}} \approx \mathbf{0,8203}$$

For a Max-2-Sat problem with N variables where the variables are uniformly distributed among the clauses, a variable has a chance of $2/N$ of occurring in any given clause. If the problem has density D (density is the number of clauses divided by number of variables,

so the number of clauses C is $N * D$), then a selection S of half the clauses ($|S| = \frac{ND}{2}$) can be expected to have the following number of variables (because every variable has a chance $1 - \frac{2}{N}$ of not occurring in a given clause, and $\left(1 - \frac{2}{N}\right)^{\frac{ND}{2}}$ of not occurring in any of the selected clauses, so $N\left(1 - \frac{2}{N}\right)^{\frac{ND}{2}}$ variables can be expected not to occur in the selected clauses, assuming they are uniformly distributed):

$$N - N\left(1 - \frac{2}{N}\right)^{\frac{ND}{2}}$$

Therefore, to be able to improve on the solve-time of an SDP solver by using clause-based decomposition, the following must hold:

$$N - N\left(1 - \frac{2}{N}\right)^{\frac{ND}{2}} < \frac{1}{2^{\frac{2}{7}}}N$$

$$1 - \left(1 - \frac{2}{N}\right)^{\frac{ND}{2}} < \frac{1}{2^{\frac{2}{7}}}$$

$$D < \frac{2\ln\left(1 - \frac{1}{2^{\frac{2}{7}}}\right)}{\ln\left(1 - \frac{2}{N}\right)N}$$

Using this function we can determine the maximum density for which this approach can work, for a problem with N variables.

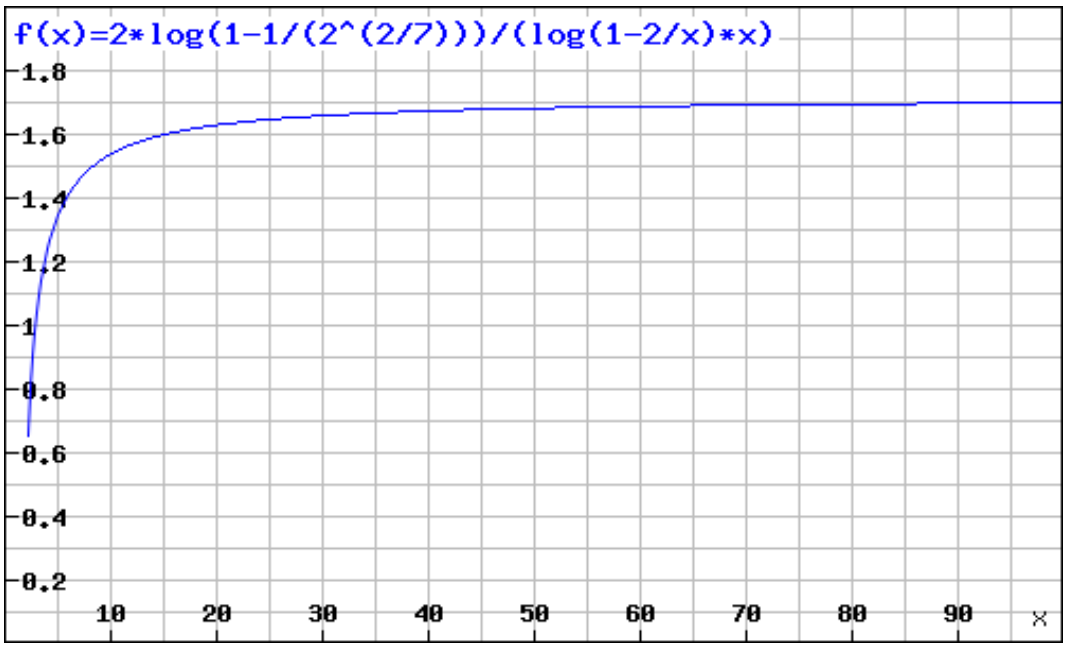


Figure 5.1: 8: X-axis: number of variables. Y-axis: maximum allowed density.

It follows that, for problems with variables uniformly distributed among its clauses, the limit to the density (as $N \to \infty$) is $-\log\left(1-2^{-2/7}\right) \approx \mathbf{1.716}$.

It should immediately be noted that for such low densities, complete solvers like MSUnCore are able to solve Max-2-Sat problems exactly in a matter of seconds, for large numbers of variables.

5.2 Evaluation

The following results are based on a benchmark with 100 problems, each with 60 variables and 90 clauses (density: 1.5) which were selected out of several thousands for being highly unsatisfiable (= 85 clauses satisfiable).

5.3 Benchmark Results

The following graphs illustrate the results of this method of decomposition.

5.3.1 Upper Bound Quality

Figure 5.2 shows that the values of upper bounds for the individual parts are distributed according to a Gaussian distribution and never go below 50% of the number of clauses. Figure 5.3 shows that the resulting upper bounds (the sum of the individual parts) always exceeds the trivial upper bound that is the total number of clauses. It also shows, without decomposition, SDP is barely able to find non-trivial upper bounds for this type of problem at such a low density.

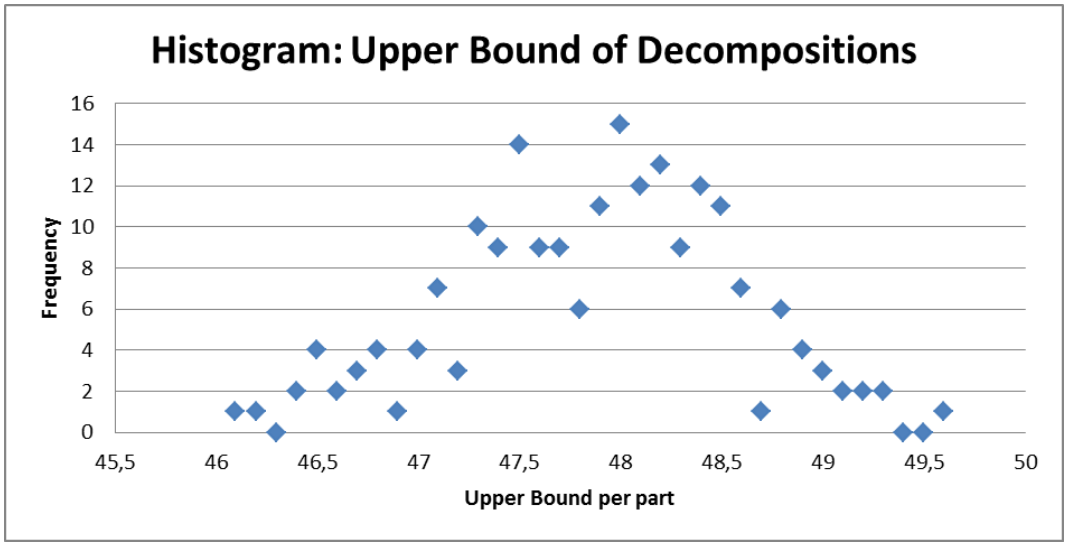


Figure 5.2: Upper Bounds for a benchmark with 90 clauses, density 1.5.

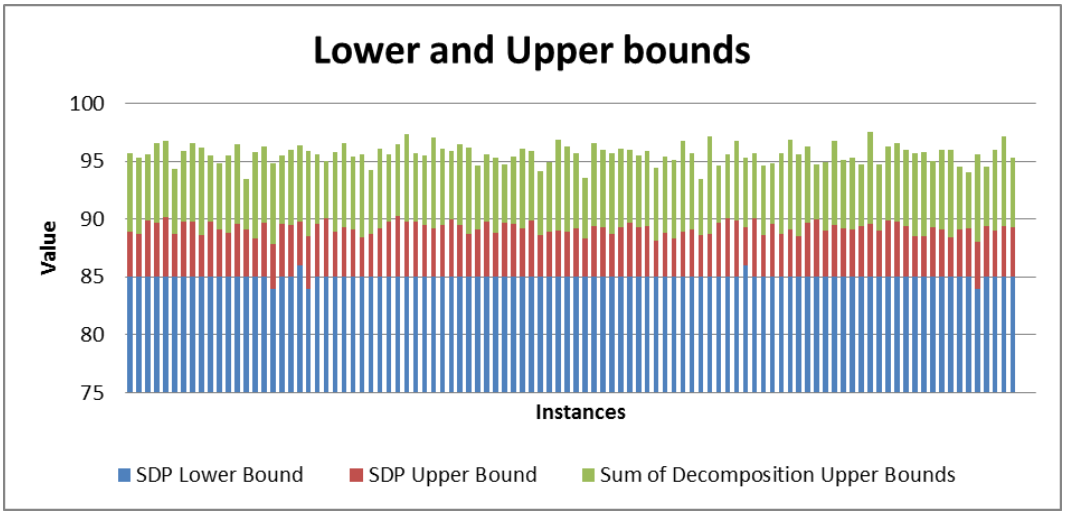


Figure 5.3: Results for a benchmark of 100 problems with 90 clauses (density 1.5) with weight 1.0.

5.3.2 Solve Time

Figure 5.4 shows that the number of variables of each part of the decomposition is distributed according to a Gaussian distribution with a mean value around 75% of the total number of variables (for a problem at density 1.5). Figure 5.5 shows that this method failed to solve the problem faster than without decomposition in all but one instance.

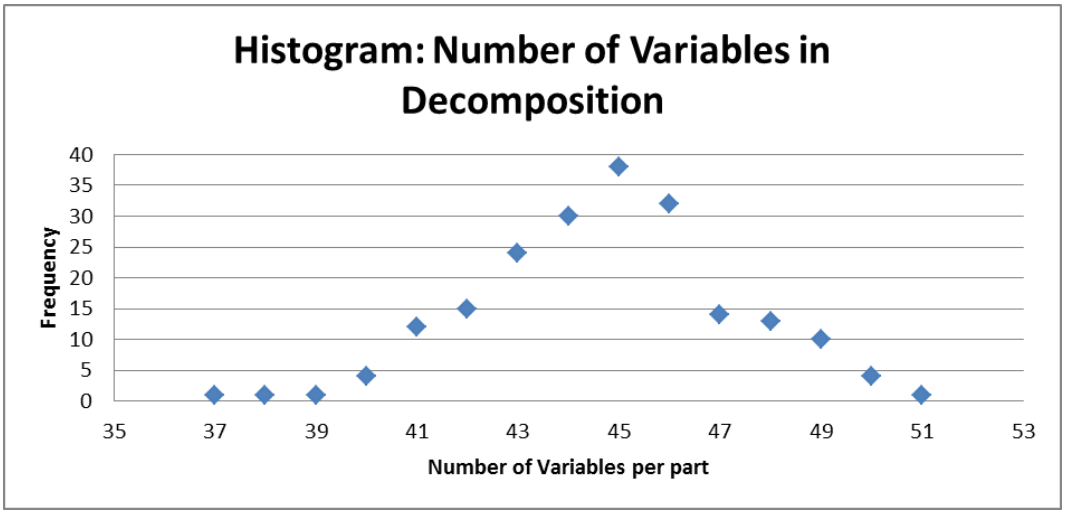


Figure 5.4: Number of Variables for 50% clause decomposition of problems with 60 variables, density 1.5.

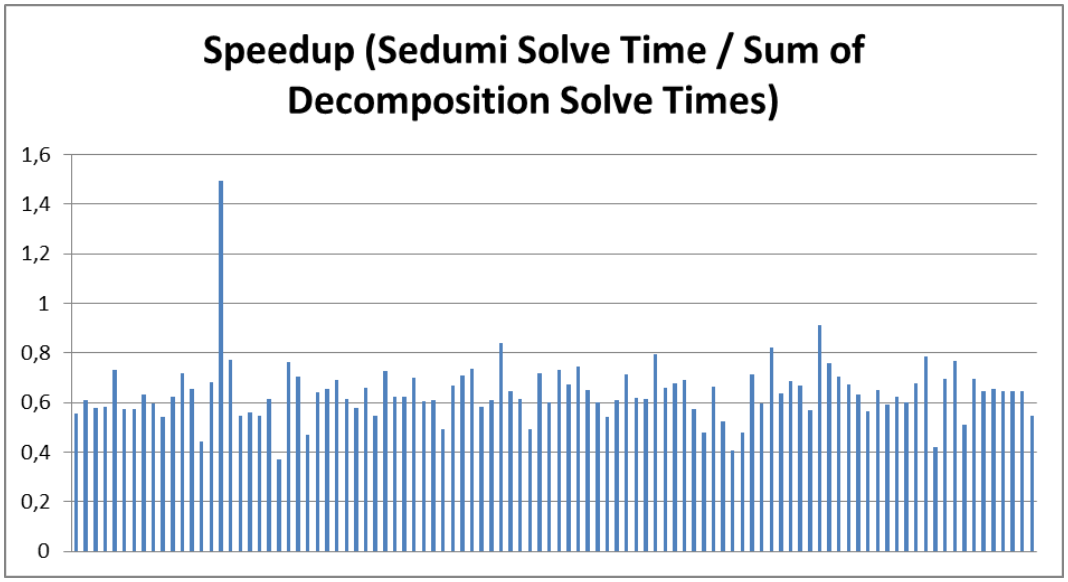


Figure 5.5: Speedup for a benchmark with 60 variables, density 1.5.

5.3.3 Test Results: Single Problem

The following results are based on 100 decompositions of the first problem of the previously mentioned benchmark. These results show that the results presented in the previous two sections are relatively stable with respect to the decomposition.

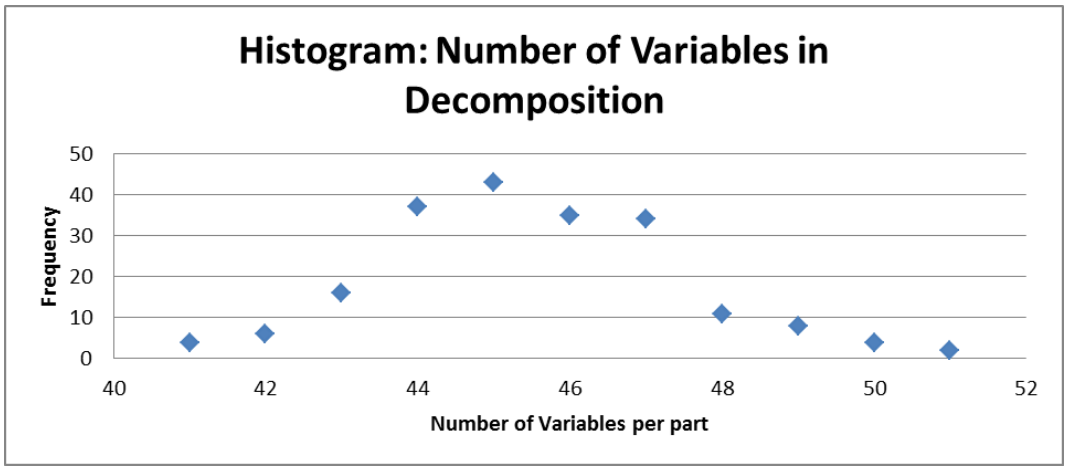


Figure 5.6: Number of Variables for 100 random 50% clause decompositions of a problem with 60 variables, density 1.5.

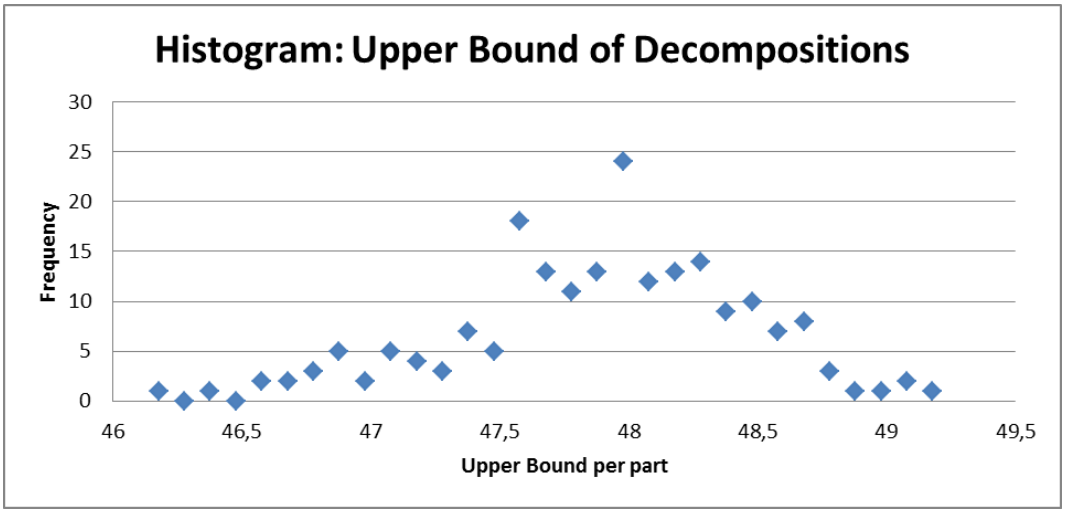


Figure 5.7: Upper Bounds for 100 decompositions of a problem with 90 clauses, density 1.5.

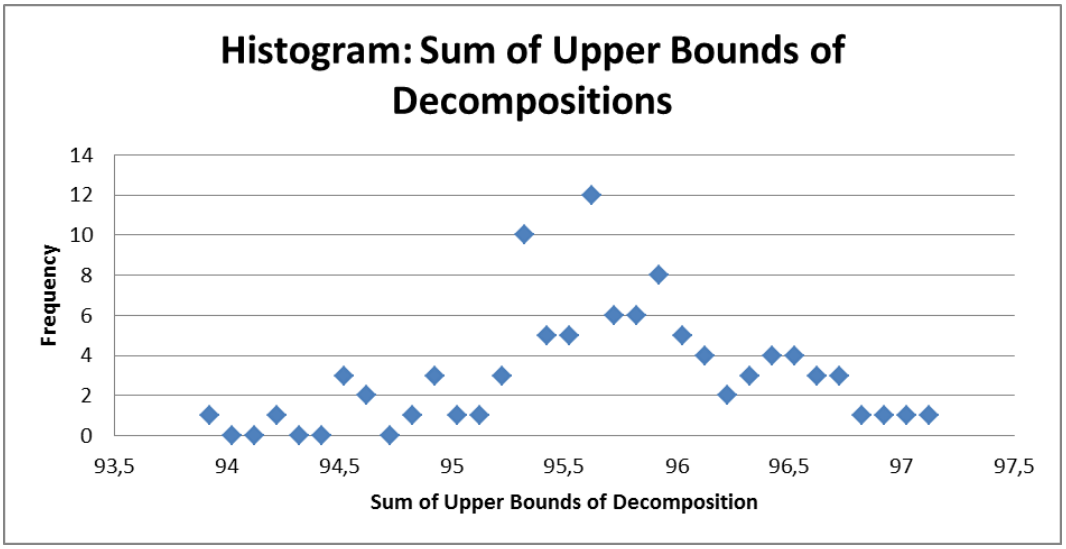


Figure 5.8: Combined upper Bounds for 100 decompositions of a problem with 90 clauses, density 1.5.

5.4 Conclusion

Expected speedup for random problems is less than 1 (no speedup) at densities greater than 1.716. Indeed, for a benchmark with 60 variables and density 1.5, the average measured speedup out of 100 problems is 0.64, so it cost more time. Therefore this method will only achieve speedup for a very small density interval and even then the speedup is low.

The upper bounds achieved using this method consistently exceed the number of clauses by a significant amount (both for 100 different problems, and for 100 decompositions of a single problem). At these low densities, even SDP *without decomposition* barely manages to find useful upper bounds.

All these results are based on unstructured problems (but artificially selected to have a large number of unsatisfiable clauses, given the low density). It is possible that better results can be achieved for structured problems.

Based on these results we decided that this method of decomposition was not viable and did not warrant further investigation.

Chapter 6

Variable-Based Decomposition Methods

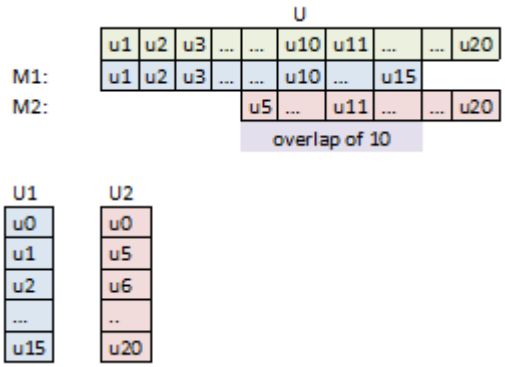
This chapter describes the various variations on the variable-based decomposition method described in section 4.2 that we've experimented with.

6.1 Decomposition into more than two parts

Theoretically this method of decomposition could involve any number of parts, with the potential speed increase growing as the problem is decomposed into more (smaller) parts. Unfortunately our results showed that the speedup gained by decomposing into more than 2 parts declined rapidly. Decomposing into 4 parts is less than 5 times faster than a decomposition into 2 parts, which is insufficient to justify an observed decline in quality of the results. Therefore, our research focused on decomposition into just two parts, and we'll assume the number of parts of a decomposition is 2 for the remainder of this chapter.

6.2 Decomposition with Overlap

The variable decompositions do not necessarily have to be disjoint. When a variable occurs in more than one segment, we say these segments overlap. Overlap provides an additional connection between the segments of a decomposition, which potentially improves the overall performance.
For example, consider the following illustration, of a problem with 20 variables and an overlap of 10.



6.3 Decomposition Heuristics

Several heuristics can be used to decide how to separate the variables. This section lists the most obvious choices.

6.3.1 Static

Let $r \in [0,1]$ be the decomposition ratio, and let $o = \frac{|overlap|}{2}$.
This is the most basic decomposition, where the first $r * n + o$ variables become part of the first problem, and the last $(1-r) * n + o$ variables become part of the second part of the decomposition.

6.3.2 Random

This decomposition is like the Static decomposition, but with a random permutation applied to the variable selection.

6.3.3 Minimum Cut

This decomposition is based on an incidence graph based on a Max-2-Sat problem, where there is one node for every variable, and an edge between any two nodes if their corresponding variables occur together in a clause, with the number of such clauses as the weight of the edge. This variable decomposition is then equivalent to the cut that minimizes the sum of the weights of the edges spanning the cut.
This will typically, although not necessarily, result in a decomposition of $n-1$ variables for one part, and 1 variable for the second. Decomposition into such unequally large problems does not provide significant speedup. A better, similar decomposition heuristic would be the SparsestCut. SparsestCut finds a balance between an evenly sized cut and the minimum cut. Given the same incidence graph as described above the SparsestCut is the cut $(S, \overline{S})$, where $|S| \leq \frac{n}{2}$ and where $E(S, \overline{S})$ is the sum of the weight of the edges spanning the cut, that minimizes $\frac{E(S,\overline{S})}{|S|}$. However, finding a SparsestCut is NP-hard, and even though there exists

an $\sqrt{\log n}$ approximation of SparsestCut, which can be computed in $O(n^2)$ [17], we did not implement this method.

6.4 Ordering Based Decomposition Heuristics

The following decomposition heuristics define a (partial) ordering of the variables by assigning each variable u_i a heuristic value h_i. Given such an ordering of variables, a decomposition can be constructed that maximizes and minimizes the sum of heuristic value of the variables in the first and second part of the decomposition respectively, called "Most". For example, "Most Degree" selects the variables with the highest degree for the second part. Alternatively the sum value can be divided more evenly by alternatingly assigning variables to part one and two (called "Zip"), or using a partition problem approximation algorithm for an almost equal distribution.
The following are some examples of heuristics which we've experimented with.

6.4.1 Most/Zip Flipped

Under this heuristic the $(1-r)*n+\frac{o}{2}$ most flipped (by a local search solver) variables are used for the second part of the decomposition.

$$h_i = flips_i$$

6.4.2 Most/Zip Balance

$$h_i = balance\,(i) = \sum_{c=\left(t_i \vee t_j\right),\ t_i=u_i} w_c - \sum_{c=\left(t_i \vee t_j\right),\ t_i=\overline{u_i}} w_c$$

6.4.3 Most/Zip Degree

$$h_i = \sum_{c=\left(t_i \vee t_j\right)} w_c$$

6.4.4 Most/Zip Sign Degree

We define the "Sign Degree", which represents the degree of a node in an incidence graph consisting only of edges that go "against" the balance of both its variables as follows:

$$h_i = \sum_{c=\left(t_i \vee t_j\right)} \begin{cases} w_c \; if \quad t_i = u_i \leftrightarrow balance\,(i) < 0 \; and \;\; t_j = u_j \leftrightarrow balance\,(j) < 0 \\ 0 \; otherwise \end{cases}$$

6.5 Contra-Balanced Decomposition

A question to ask is whether $1/2$ is the optimal value to assign the potentially satisfiable clauses. Perhaps a solver could benefit from a more accurate estimate of the value of such a clause, based on the probability of it getting satisfied in another sub-problem.

If we define a variable's balance as the sum of the weights of the clauses where a variable occurs positively minus the sum of the weights of the clauses where it occurs as a negated literal, then a literal that corresponds with the balance is more likely to satisfy a clause than a literal that opposes it.
For example, if the balance of y is $+4$, then y is likely to be assigned the value $true$, and $(x \vee y)$ is likely to be satisfied by y. Therefore if a decomposition separates x and y, satisfying $(x \vee y)$ using x has a lower priority, which can be reflected using the weight assigned to the removed term. Instead of replacing $(x \vee y)$ with $(x \vee \frac{1}{2})$, we might replace it with $(x \vee \beta)$, with $\beta > \frac{1}{2}$, thereby reducing the relative importance of x.
Generally, when given a clause $(t_1 \vee t_2)$ with weight w, where $t_2 = u$ or $t_2 = \overline{u}$, and u is to be removed, let $c = \begin{cases} -1 \; if \; t_2 = \overline{u} \\ 1 \; if \; t_2 = u \end{cases}$ and $\beta = \frac{c * balance(v)}{c_b} + 0.5$ where $c_b = 2 * \max_i(|balance(u_i)|) + 1$, then we substitute $(t_1 \vee t_2)$ with $(t_1 \vee t_1)$ with weight $(1-\beta)w$, which corresponds to $(t_1 \vee \beta)$.

$$(1-\beta)(x) \equiv (x \vee \beta)$$

If t_2 corresponds to its balance then $c * balance(v) > 0$ and $\beta > 0.5$, so $(1-\beta) < 0.5$ and the clause becomes less important to satisfy. If t_2 does not correspond to its balance then $c * balance(v) < 0$ and $\beta < 0.5$, so $(1-\beta) > 0.5$ and the clause becomes more important. Since $|c * balance(v)| < \frac{1}{2} c_b$ we know that $0 < \beta < 1$, so the clause never disappears entirely or becomes worth more than it actually is.

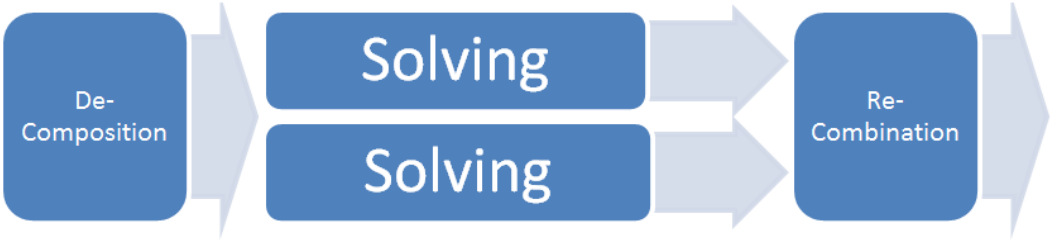
Chapter 7

Solving Methods

Using the variable-based decompositions described in the previous chapters, several approaches can be taken to solve the original problem, which are presented in this chapter.

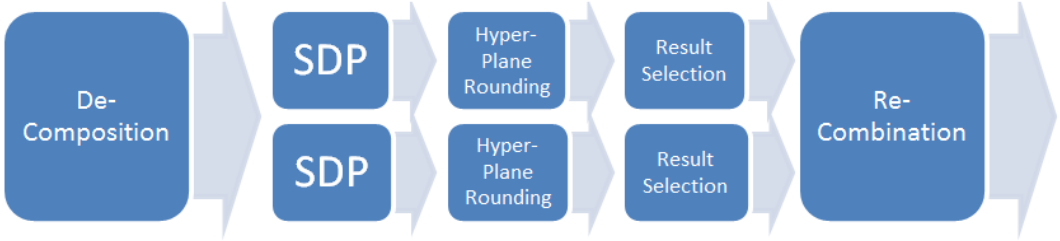
7.1 Independent

The simplest approach is to solve both problems separately, and afterwards merge the results. In the case of decompositions with overlap, those variables in the intersection of multiple segments will be assigned several times; their value is selected from these arbitrarily, e.g.: whatever it is assigned last.



This approach is solver-independent: it can be implemented using SDP or other weighted Max-Sat solvers.

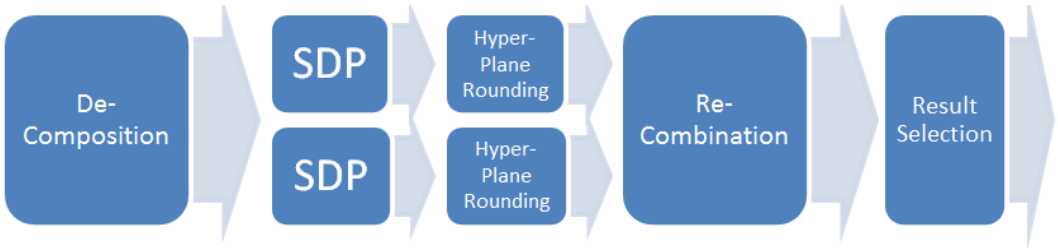
In the case of SDP, the underlying process is as follows:



Both parts of the decomposition are solved independently and produce a single variable assignment for all variables per part, which when combined give a single variable assignment for all variables in the original problem, and therefore a single resulting lower-bound.

7.2 Delayed Result Selection

A less naive approach specific to SDP based on the previous method recombines the partial assignments resulting from hyper-plane rounding before choosing the assignment that satisfies the most clauses.



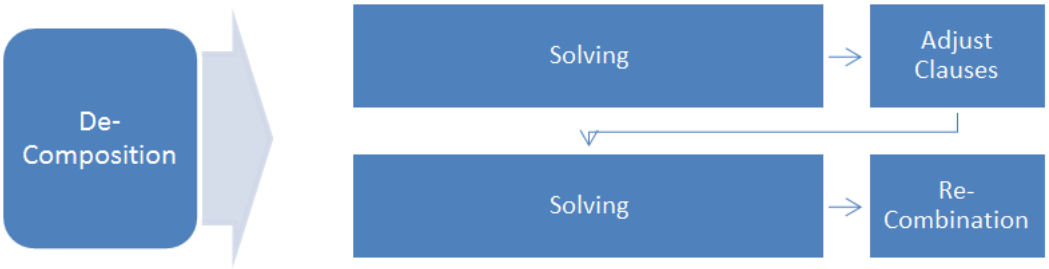
This approach has the benefit that result selection can be deferred until the very end, allowing the evaluation and selection of the variable assignments resulting from hyper-plane rounding to take into account a full variable assignment of the original problem before any selection is made.

If hyper-plane rounding is repeated T times for each part of the decomposition, and there are P parts, then there are T^P possible combinations to evaluate. Since we mostly keep $P = 2$ this is not necessarily prohibitively expensive. Alternatively, a limited number of combinations can be selected randomly. This reduces the number of evaluations that need to be performed, but can select equally good lower bounds with good probability.

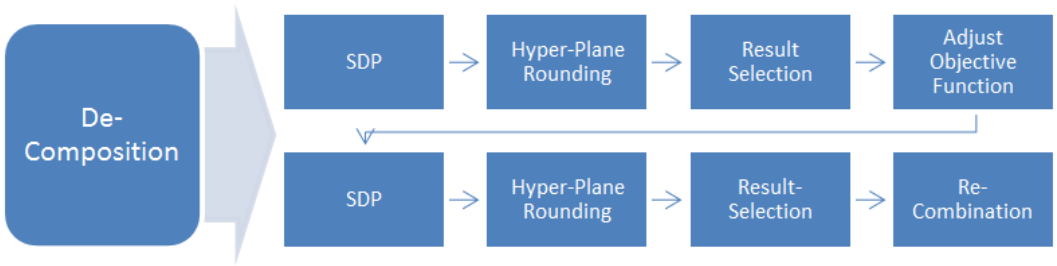
7.3 Sequential

Contrary to the previous two methods the results from solving one part can be used to help solve subsequent parts. If a clause is satisfied by an already assigned variable, it is of no benefit to satisfy this clause again later with variables from another segment, hence these clauses can be removed from the target function. Likewise, clauses that contain, but have not been satisfied by, previously assigned variables will have to be satisfied by variables in following segments, if at all. Hence such clauses become unary clauses.

This approach is solver-independent.



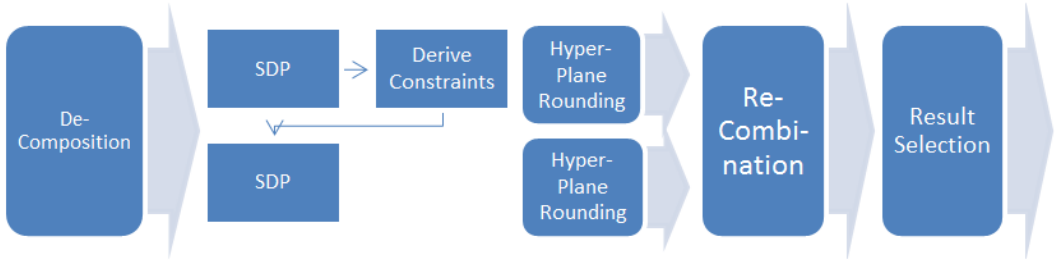
If this approach is applied using SDP, it looks as follows:



A downside to this approach is that hyper-plane rounding and result selection must be performed on the first part, before the second can be solved. This ultimately restricts the solution-space searched by the result selection, compared to the delayed result selection approach. Another downside is that this approach does not lend itself to parallelization as well as one might like, unlike the previous approaches.

7.4 B-Sequential

This method is an alternative to the Sequential approach that does not commit to a selection of a variable assignment from hyper-plane rounding of intermediate results, but instead uses the un-rounded results from a solution matrix to constrain subsequent problems, by modifying their objective-function. The benefit of this is that it can combine sequential constraints with delayed result selection.



To illustrate, consider the example illustrated in section 6.2. Then the variable matrix X of the second part is constrained conceptually as follows:

X2 with results from M1:

$X2 \leftarrow B1$

b0·b0	b0·b5	b0·b6	b0·...	b0·v20
b5·b0	b5·b5	b5·b6	b5·...	b0·v20
b6·b0	b6·b5	b6·b6	b6·...	b0·v20
...·b0	...·b5	...·b6	.....	...v20
v20·b0	v20·b5	v20·b6	v20...	v20v20

These conceptual constraints are then implemented in the following manner:

1. Constraints of the form $X_{x,y} = b_i \cdot b_j$ are implemented as follows:

   $X_{x,y} = b_i \cdot b_j$ (the dot product is at this time a known constant)

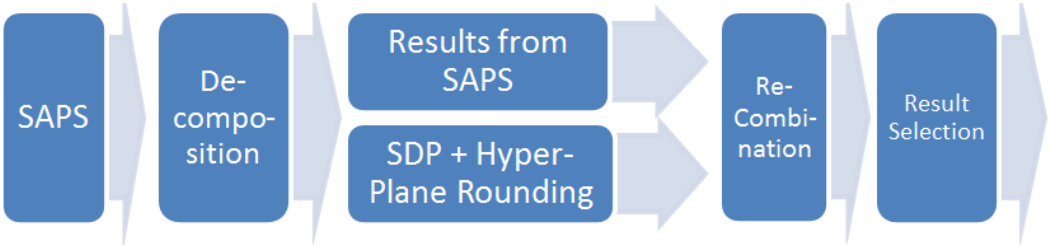
2. The partially constrained variables of the form $X_{x,y} = v_i \cdot b_j$ can not be implemented without rounding B1 and therefore have no additional constraints.

3. The remaining variables are of the form $X_{x,y} = v_i v_j$ and are not additionally constrained.

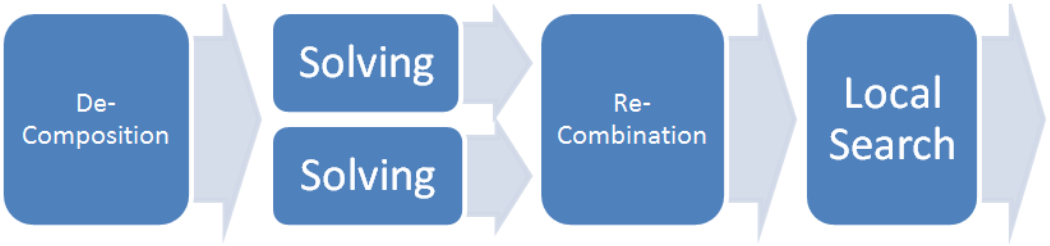
This solve method is only different from Delayed Result Selection solving when the decomposition has overlap.

7.5 Hybrid Sequential

Since other types of solvers are better for some/several types of problems multiple kinds of solvers can be combined, and although not guaranteed, doing so might combine the desirable properties of both solvers. For example, a local search solver's result could be used as input to an SDP solver operating on the hardest subset of the problem, determined by the most flipped variables by the local search solver, while using the truth assignments determined by the local search solver for the remaining variables.



Another, more effective, alternative would be to use the lower bound derived following decomposition as input to a local search solver:



The assumption that would justify both these approaches is that there are some problems where an SDP solver might outperform a local search solver.

Chapter 8

Rounding Methods

When solving parts of a decomposition using SDP, hyper-plane rounding must be applied to derive a variable assignment. Application of an SDP solver results in a matrix S, such that $S = B * B^T$. Matrix B is found by applying the Singular Value Decomposition function to S. For a decomposed part containing $\frac{n}{2}$ variables, B is a matrix of dimension $\left(\frac{n}{2}+1\right) \times r$, where r is the rank of S. Each row of B corresponds to a variable, where the first row, henceforth referred to as B_0, corresponds to the homogenous variable.

Hyper-plane rounding uses a plane through the origin, defined by a vector through the origin, to define a cut between the rows belonging to each variable (which are poles on an r dimensional sphere). Several methods for hyper-plane rounding are available, and their effectiveness depends on the solving method.

8.1 Uniform Random

This rounding method uses a random vector $\bar{r}$ of dimension r and length 1 to define a random plane through the origin.

8.2 The B_0 Vector

This rounding method uses the vector B_0 to define a plane through the origin. The downside of this approach is that it is deterministic and repeated rounding using this method is therefore pointless. However, this pole has a high probability of giving a good lower bound.

8.3 Around B_0

This rounding method uses a random vector near B_0 to define a plane through the origin to provide the advantages of the previous two methods. Selection of a random vector "around" B_0 is achieved as follows. This multiplication rotates B_0 toward random vector y over a random angle θ, where the angle is distributed uniformly between 0 and a chosen predetermined maximum angle β.

1: Let $\bar{r}$ be a random vector of dimension r and length 1, and let α be a uniform random value in $[0,1]$.
2: Let $\theta = \alpha * \beta$
3: Create an orthonormal basis for $\Re^r$ using a basis for the span of B_0, y as first two components (where the null function gives an orthonormal basis for the null space of its argument):

$$Basis = \left[\begin{array}{cc} B_0 & \frac{y-(B_0 \cdot y)B_0}{|y-(B_0 \cdot y)B_0|} \end{array} \quad null\left(\left[\begin{array}{cc} B_0 & y \end{array} \right]^T \right) \right]$$

4: Let R be an $r \times r$ dimensional identity matrix. Set the upper left corner $R_{(1:2,1:2)}$ to $\left[\begin{array}{cc} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{array} \right]$
5: Let $M_{ROT} = Basis * R * Basis^{-1}$
6: Then the vector used to define the hyper-plane is given by $M_{ROT} * B_0$

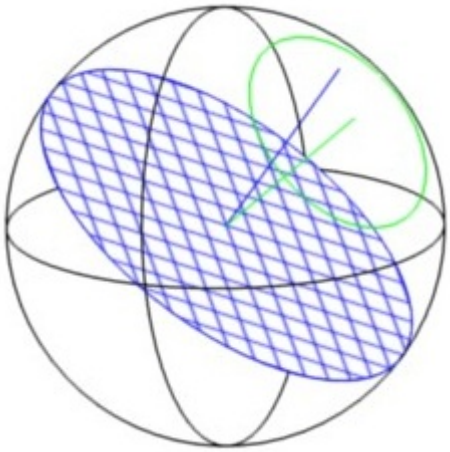


Figure 8.1: "Around B_0" rounding. The green line represents B_0, the blue line $\bar{r}$.

Chapter 9

Evaluation

The nature of the presented approaches means that it can be shown theoretically to yield very poor results for specific decompositions of specific problems, and the expected performance of the highly randomized algorithms depends heavily on the problems it is applied to. Since we're primarily interested in the potential for practical application, we've chosen to evaluate this approach empirically using a diverse set of benchmarks. A brief description of the used benchmarks is presented in this chapter.

9.0.1 Random

For this benchmark, we generate Max-k-Sat problems of a given variable count n and density d in CNF by generating $m = nd$ clauses, each with $k = 2$ variables u_i, u_j where i, j are drawn randomly (uniform) from $[1, n]$ excluding those where $i = j$, and each negated with probability .

9.0.2 Non-Uniform Random

These problems are similar to those in the random benchmark, except the occurrence of variables and the sign of their terms are explicitly non-uniform.

For every variable $u_i \in U$, randomly (uniform) select an occurrence $o_i \in [0, 1)$.
Let the probability p_i of a variable being selected for any given clause be given by $o_i / \sum_j o_j$.
Choose the probability of a variable occurring positively be chosen uniform-randomly as $s_i \in [0, 1)$.
Then construct $m = nd$ clauses, consisting of 2 terms drawn from the probability distributions defined by p_i and s_i.

9.0.3 Graph-2-Colorability ("G2C")

For this benchmark, a graph-2-colorability problem is created based on a random graph, and encoded to Max-2-Sat as follows:

Given a graph $G = (V,E)$, where $N = |V|$, for every vertex $i \in V$ introduce the variables $u_i \rightarrow i = red$, $u_{i+N} \rightarrow i = blue$ and the clauses $(u_i \vee u_{i+N})$ and $(\neg u_i \vee \neg u_{i+N})$ with weight 1.
For every edge (i, j) in the graph, introduce a clause $(\neg u_i \vee \neg u_j)$ and a clause $(\neg u_{i+N} \vee \neg u_{j+N})$ both with weight $2N+1$.

9.0.4 Alternative Graph-2-Colorability ("NewG2C")

This is an alternative Graph-2-Colorability encoding, where $u_i = true$ represents the assignment of one color (e.g. red) to vertex i and $u_i = false$ represents the assignment of the other color to that vertex.

Given a graph $G = (V,E)$, for every edge (i, j), introduce a clause $(\neg u_i \vee \neg u_j)$ and a clause $(u_i \vee u_j)$.

9.0.5 Triangle Packing

This method encodes a vertex-disjoint triangle packing problem as Max-2-Sat. Given a graph $G = (V,E)$, let $T = T_1 \ldots T_K$ be the set of all triangles in the graph. Then the problem of finding the maximum number of triangles can be encoded as follows:

For every triangle $T_i \in T$, introduce a variable u_i and a unary clause (u_i) with weight 1.
For every pair of triangles $i, j \in T$, if i, j share a vertex, introduce a clause $(\neg u_i \vee \neg u_j)$ with weight $K+1$.

9.0.6 Split

This benchmark consists of problems that arise after variable-based decomposition of a Random problem. Given a random Max-2-Sat problem consisting of variables $u_1 \ldots u_n$ and clauses $C = c_1 \ldots c_m$, a "Split" problem consists of the variables $v_{half} = u_1 \ldots u_{n/2}$ and the following clauses:

For every clause $(t_i \vee t_j) \in C (t_i = u_i or \neg u_i)$, where $i, j < \frac{n}{2}$, add a clause $(t_i \vee t_j)$ with weight 2.
For every clause $(t_i \vee t_j) \in C$ where for one variable (assuming i w.l.o.g.) $i < \frac{n}{2}$, add a clause (t_i) with weight 1.

9.0.7 Model RB ("FRB") [18]

Problems in this benchmark are supposed to be "very hard", and are generated as follows:

1: First generate n disjoint sets of boolean variables, each of which has cardinalityn^α (where $\alpha > 0$ is a constant), and then for every variable x, generate a unit clause x, and for every two variables x and y in the same set, generate a 2-clause $\neg x \vee \neg y$.

2: Randomly select two different disjoint sets and then generate without repetitions $pn^{2\alpha}$ clauses of the form $\neg x \vee \neg z$ where x and z are two variables selected at random from these two sets respectively (where $0 < p < 1$ is a constant);
3: Run Step 2 (with repetitions) for another $rn \ln n - 1$ times (where $r > 0$ is a constant);
4: For every unit clause, we have a weight equal to 1, and for every 2-clause, we have a weight equal to$i$(where i= the total number of unit clauses + 1).

9.0.8 Dimacs_Mod

A MaxCut-based crafted benchmark used in the 2009 MaxSat evaluation.

9.0.9 Spinglass

A MaxCut-based crafted benchmark used in the 2009 MaxSat evaluation.

Chapter 10

Results

This chapter shows the results of the experiments we've performed on the previously mentioned benchmarks, for variable-based decomposition.

10.1 Basic Decomposition Results

The basic approach of variable based decomposition works, and generally finds lower bounds within 94% of those found by SDP without decomposition.

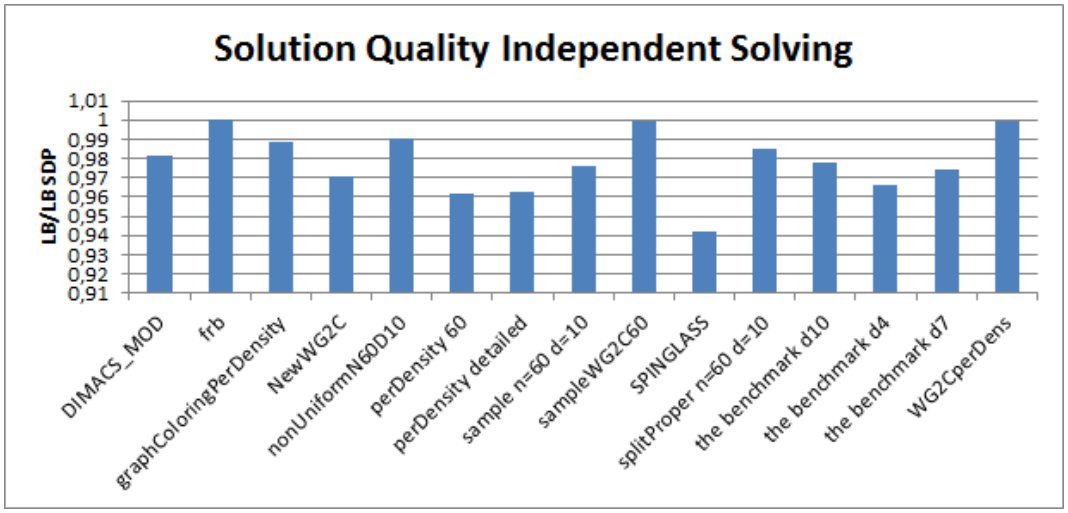


Figure 10.1: Lower bounds achieved using decomposition divided by lower bounds achieved by SDP without decomposition.

The speedup achieved by a decomposition into two parts depends on the number of variables but approaches a factor 5 as the number of variables grows, as expected. For low numbers of variables, speedup is negligible, in part due to the SDP solver's startup costs.

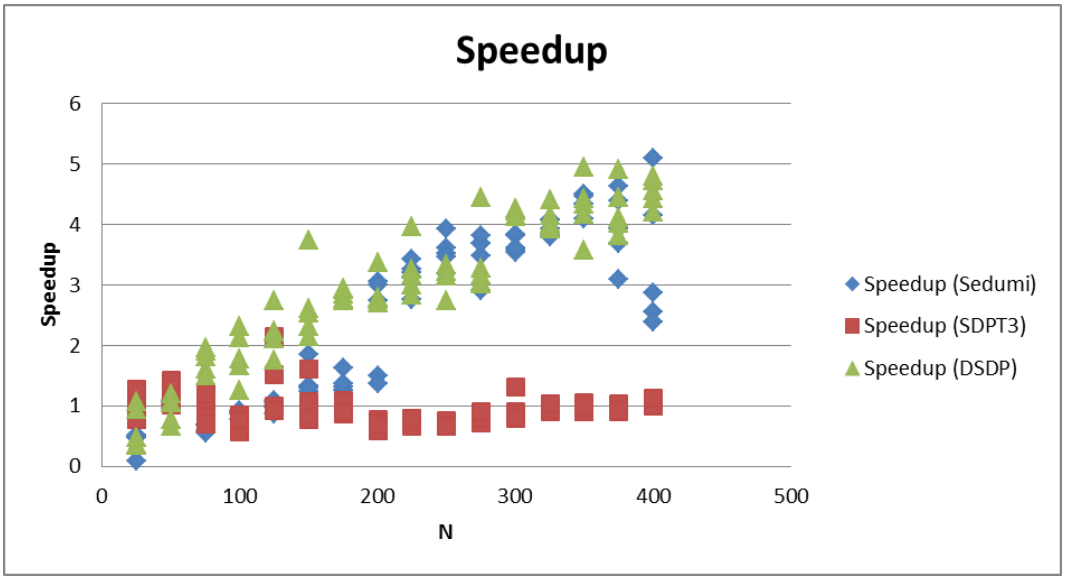


Figure 10.2: Speedup of decomposition with respect to SDP without decomposition. Uniform random problems with density 4.0.

10.2 Solve Method Results

This section presents our results for the various solve methods. Delayed result selection turns out to be the most effective method.

10.2.1 Delayed Result Selection

The delayed result selection approach is very effective at improving the quality of lower bounds. Without this approach, the lower bounds are within 5% of that found by SDP without decomposition, and with this approach the gap is reduced to less than 2% on average.

We did not measure the time it took to perform the evaluation of combinations of partial assignments, because we did not implement this with the goal of maximum efficiency. This may seem like an oversight, but we know that state of the art local-search solvers are able to evaluate a Boolean formula given a truth assignment very efficiently and performing the T^2 evaluations required by this approach using an efficient implementation should be doable in negligible time.

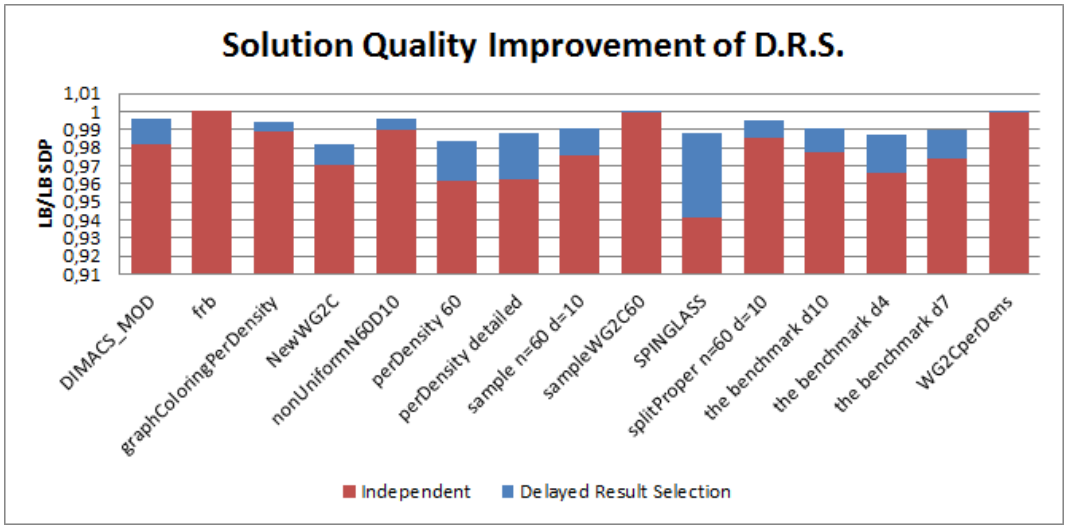


Figure 10.3: Improvement of lower bounds when using delayed result selection.

10.2.2 Scalability of Delayed Result Selection

Figure 10.4 shows that the solution quality of delayed result selection is stable with respect to N (for a benchmark of uniformly random problems, with up to 3000 variables) and is not directly negatively impacted by the number of variables. Similarly, this approach is not negatively impacted by an increase in density, as shown in Figure 10.5, which shows that the solution quality actually improves as problems get denser, including very high density problems.

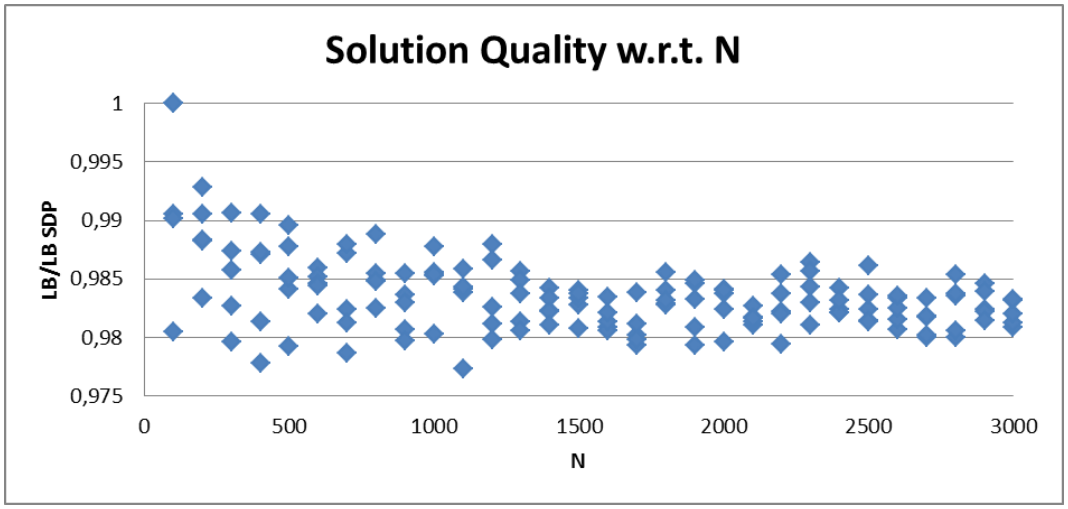


Figure 10.4: Relative solution quality of delayed result selection w.r.t. N. Uniform random problems, density = 2.5.

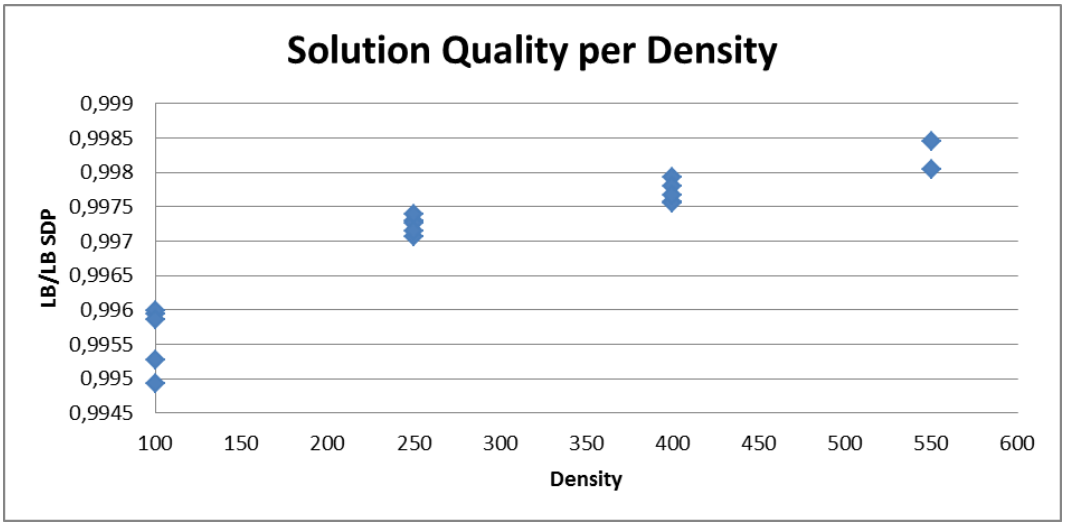


Figure 10.5: Relative solution quality of delayed result selection w.r.t. Density. Uniform random problems, N=2000.

10.2.3 Sequential

The quality of the sequential solving approach depends heavily on the relative size of the decomposed parts. If a problem is decomposed into two disjoint parts with an equal number of variables (both containing 50% of the variables) it is not significantly better than solving the same parts independently, and worse than solving them using delayed result selection. This approach improves as the relative size of the first part, whose results are used when solving the second part, gets smaller. However, because for any ratio other than 0.5 there is a part that receives more than 50% of the variables, such an uneven reduces the speedup. As the ratio approaches 0 (or 1), speedup becomes negligible, and even though the lower bound quality for such a ratio is superior to the lower bound quality found using independent solving, it is still inferior to the quality of the delayed result selection approach.

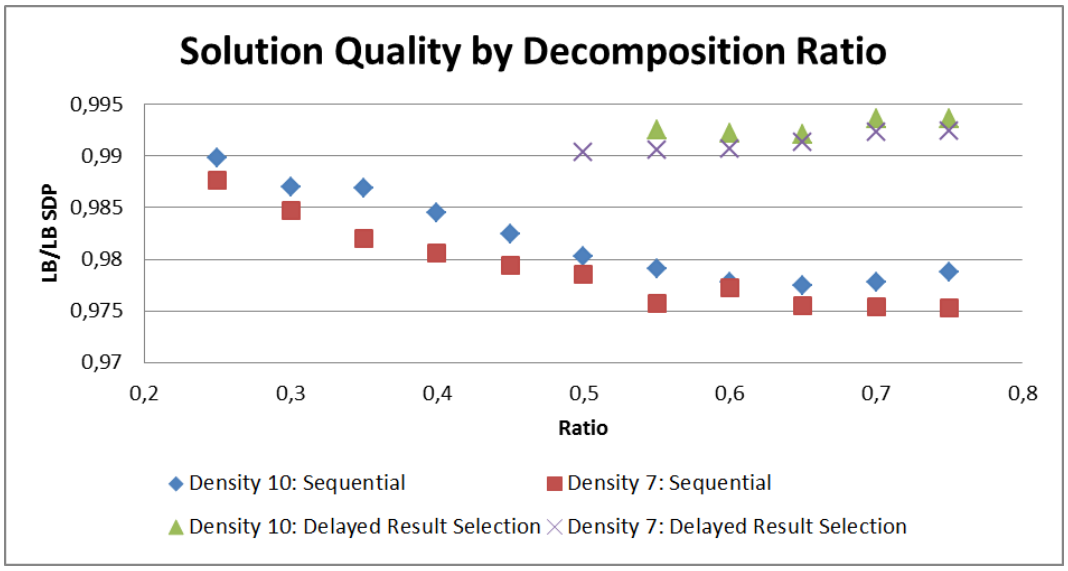


Figure 10.6: Solution quality w.r.t. decomposition ratio. Uniform random problems, N=60, D=7/D=10.

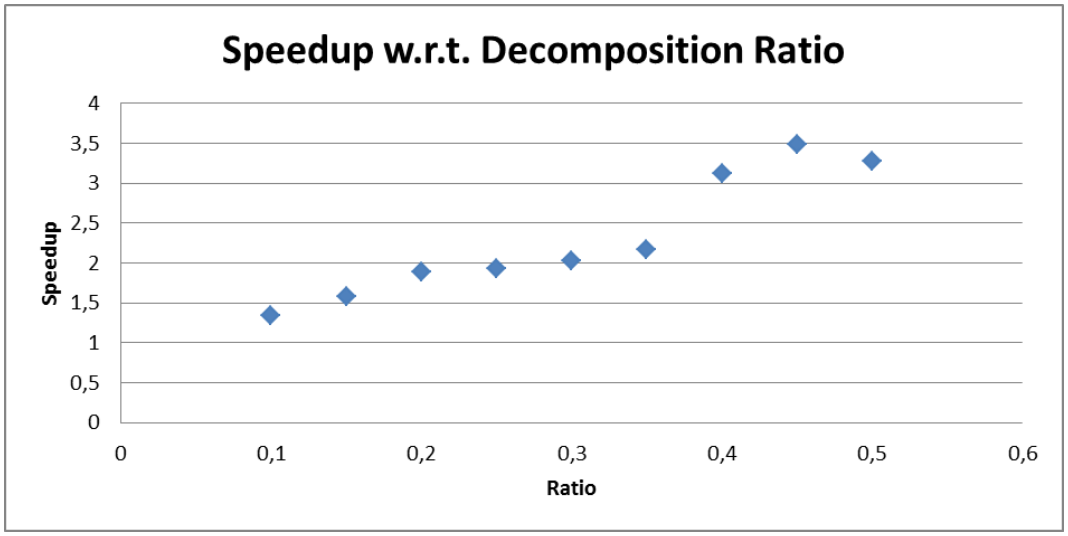


Figure 10.7: Speedup w.r.t decomposition ratio. Uniform random problems, N=2500, D=4. Solver: SDPT3.

10.2.4 B-Sequential

The B-sequential approach does not appear to significantly influence the quality of the derived lower bounds (for better or worse) when compared to the similar approach of delayed result selection with overlapping variables.

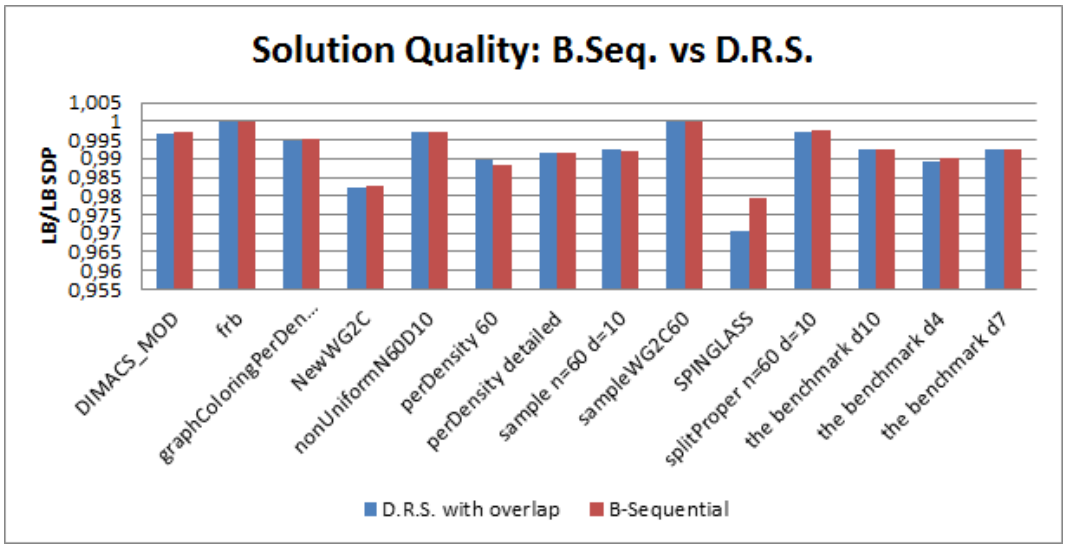


Figure 10.8: Solution quality of B-sequential solving compared to delayed result selection with similar overlap.

10.2.5 Hybrid

We were unable to find a benchmark where SDP consistently achieved better results than the local search solver, and as a result neither of the hybrid approaches proved to be useful.

10.3 Decomposition Method Results

10.3.1 Overlap

Decomposing a problem into parts with a limited overlap, variables that occur in both parts, gives a marginal improvement to solution quality. This improvement comes at a cost, since larger SDP problems have to be solved and thus overlap size implies a tradeoff between speedup and solution quality. For a problem with 60 variables, an overlap of 10 (two parts, both consisting of 35 variables) the improvement to the lower bound quality is less than a 1 percentage point increase in number of clauses satisfied. However, for such an overlap, when combined with the delayed result selection solving approach the lower bounds are within 1% of that found without decomposition.

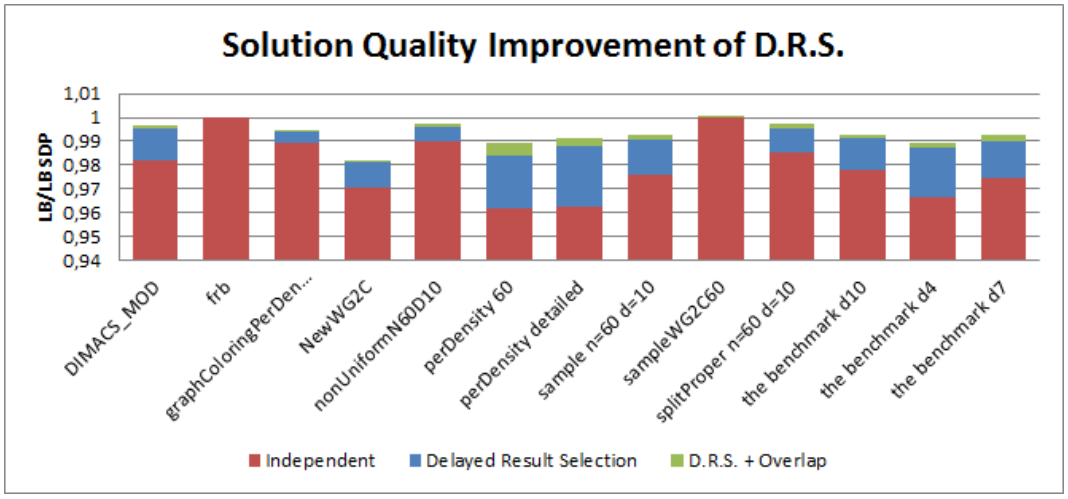


Figure 10.9: Improvement of lowerbounds when using overlap.

10.3.2 Decomposition Heuristics

Using the minimum cut of a problem's incidence graph usually resulted in the removal of a single variable, so decomposed parts of size $N-1$ and 1, which provided a small and insignificant speedup although often at no cost to the solution quality. This would suggest that a decomposition such as Sparsest Cut, which balances the number of vertices in the cuts as well as the cutsize could be effective, however we did not investigate this further because other results (see appendix) showed no strong correlation between cut size and solution quality for evenly sized decompositions.

We've experimented with the various ordering based decomposition heuristics, but as the following chart indicates, none of those reliably improved on a simple random decomposition. The chart shows the quality of the lower bound for various decomposition methods, when solved with delayed result selection, relative to the lower bound obtained without decomposition for two benchmarks: one structured and one uniform random benchmark.

We've attempted to find a correlation between the quality of lower bounds and several metrics that characterized random decompositions, the results of which are presented in the appendix, but none of those provided sufficient correlation to suggest they might be used to construct superior decompositions.

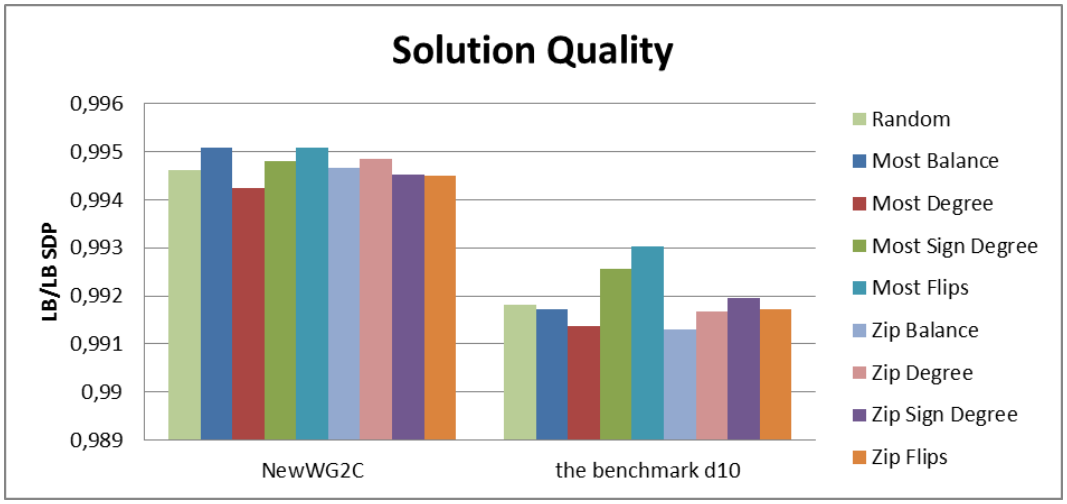


Figure 10.10: Solution quality comparison of various decomposition heuristics for two benchmarks.

10.3.3 Contra-Balanced Decomposition

This method of decomposition, which does not change the variables that are selected but only adjusts the values of the clauses in the decomposition based on the probability of certain terms becoming satisfied, shows some significant improvements on our random benchmarks, but does not appear to work (and sometimes even reduces the solution quality) for harder structured benchmarks.

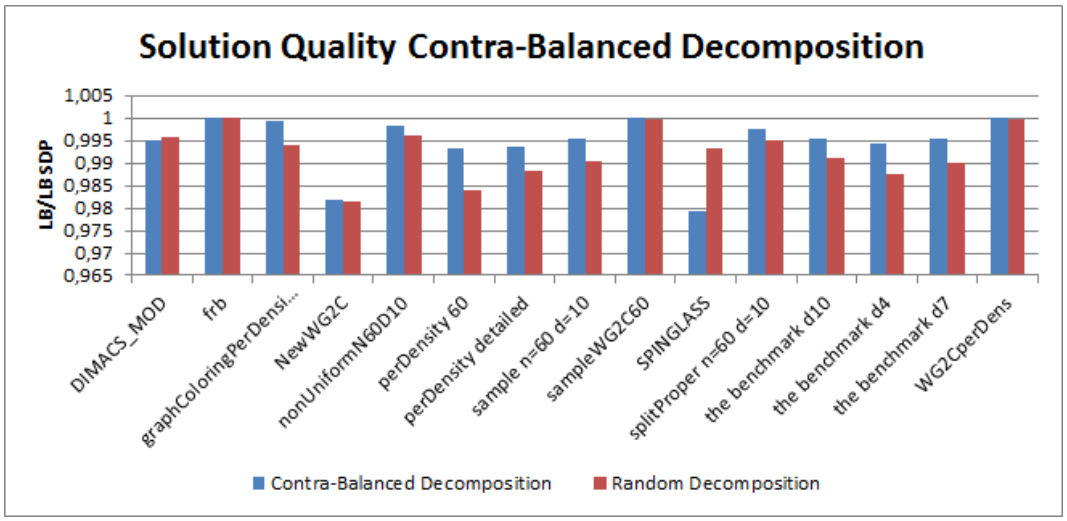


Figure 10.11: Solution quality comparison of contra-balanced decomposition with random decomposition.

10.4 Rounding Method Results

We tried two methods of hyper-plane rounding. Random and "around B_0" as described in chapter 8. The results of these rounding methods are presented here.

10.4.1 Without decomposition:

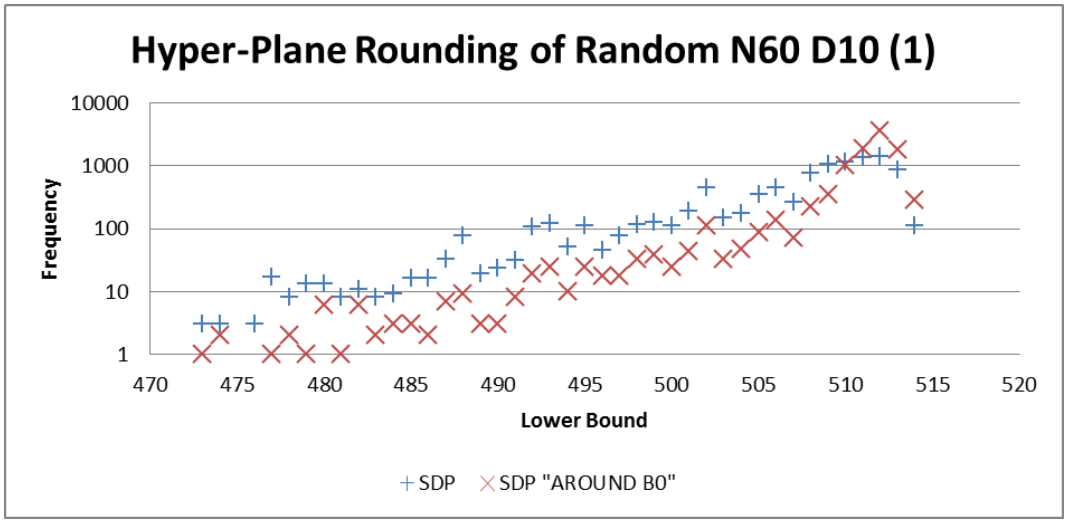


Figure 10.12: Histogram for 10.000 Hyper-Plane roundings for a single instance, rotation up to 108°.

On certain benchmarks "around B_0" rounding has a slightly higher chance of finding a variable assignment that achieves a high lower bound, as seen in Figure 10.12 which shows that the best lower bounds are found with higher frequency, but it does not find better lower bounds than random rounding, when given a sufficiently large number of roundings.

On other benchmarks, the frequency of finding lower bounds of a certain quality is almost identical between the two methods (see, for example, Figure 10.13). This difference can be explained by considering the variability of the assignments resulting from hyper-plane rounding. If we define "hamming variability" as a population's average hamming distance (number of different Boolean variables) to the median (per variable) assignment, we can see that "Around B_0" rounding results in lower variability on some benchmarks, as shown in Table 10.1. This is likely due to the increased similarity of the variable assignments found by hyper-plane rounding with more similar planes. Furthermore, these differences in hamming variability corresponds to the differences in frequency of resulting lower bounds.

These results suggest that using a hyper-plane defined by a pole in the vicinity of B_0 is preferable, because it finds good lower bounds with higher frequency on some benchmarks, and shows no difference on others.

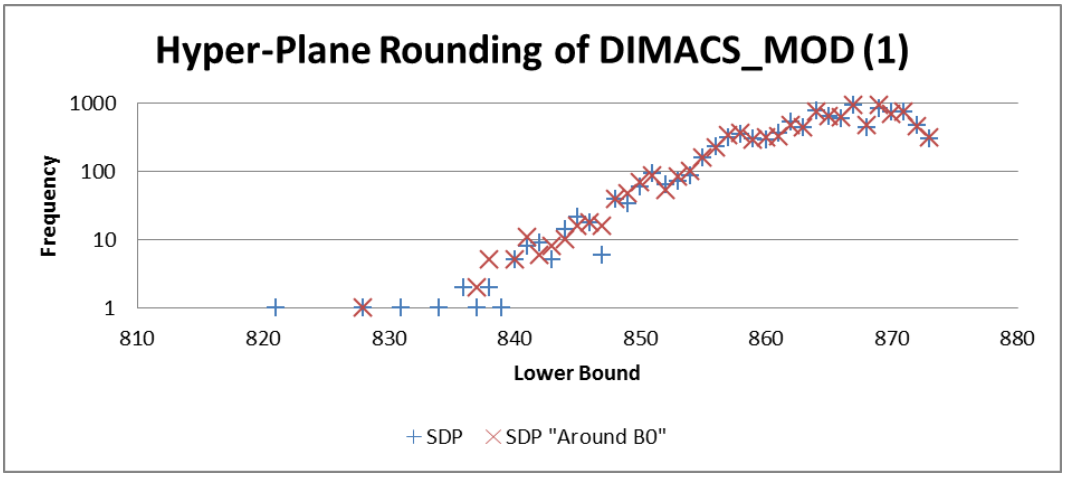


Figure 10.13: Histogram for 10.000 Hyper-Plane roundings for a single instance, rotation up to 108°.

Benchmark	**SDP**	**SDP "AROUND B0"**
Random N60 D10	15,0576	5,6002
DIMACS_MOD	20,8706	21,5858
frb	24,072	3,34
Spinglass	20,36667	6,59333
NewWG2C	19,6856	19,7906

Table 10.1: "Hamming Variability" of rounding methods

10.4.2 With Decomposition:

A similar difference in hamming variability, between these two methods of rounding, can be found when applying decomposition and delayed result selection. On some benchmarks, the variability of assignments is reduced when using "around B_0" rounding (see Table 10.2), and this results in an increased frequency of specific resulting lower bounds on corresponding problems (see Figure 10.14 and Figure 10.15).

Benchmark	**Delayed Result Selection**	**"AROUND B0" + D.R.S.**
Random N60 D10	11,28	4,71
DIMACS_MOD	20,75	20,90
frb	0	0
Spinglass	28,27	26,97
NewWG2C	20,04	19,84

Table 10.2: "Hamming Variability" of rounding methods

When applying decomposition, this reduced variation of variable assignments when us-

ing "around B_0" rounding results in a smaller (less varied) set of possible combinations of overall variable assignments when using delayed result selection recombination. Unlike the case of SDP without decomposition, where there was an increased frequency of "good" lower bounds, after decomposition the distribution of lower bounds more closely resembles a normal distribution, and the reduced variability results in an increased frequency of the median lower bounds, which results in worse overall performance compared to other rounding methods for the same decomposition – on benchmarks displaying reduced variability.

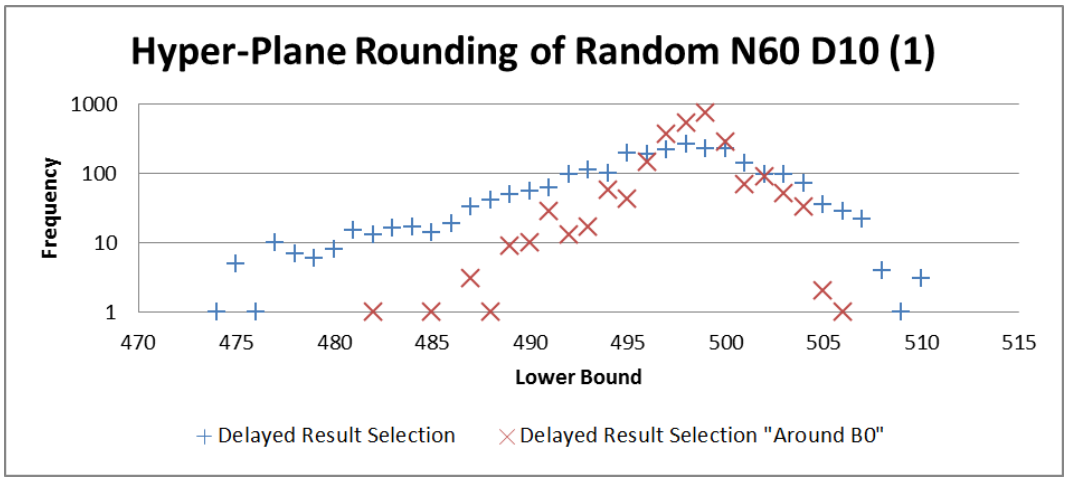


Figure 10.14: Histogram for 50 * 50 Hyper-Plane roundings for a single decomposition of a single instance

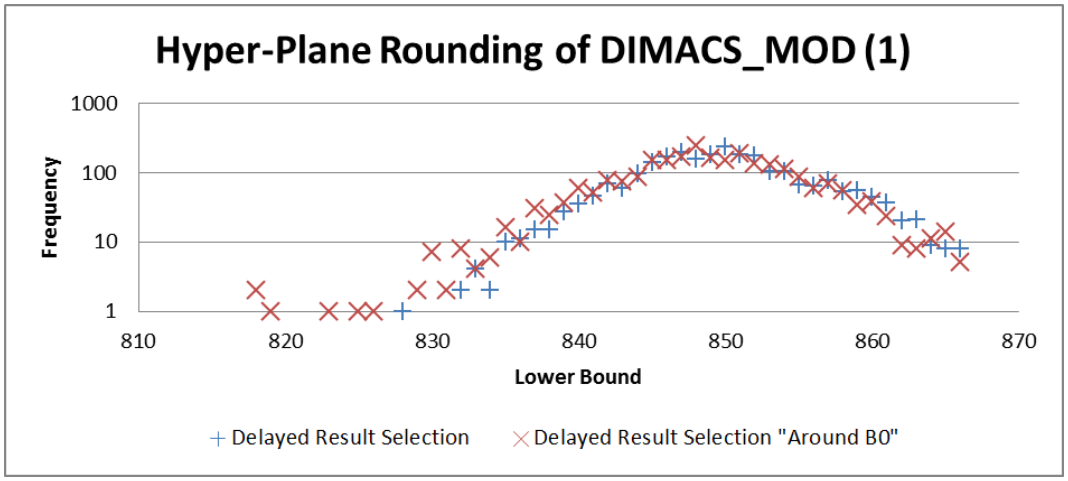


Figure 10.15: Histogram for 50 * 50 Hyper-Plane roundings for a single decomposition of a single instance

Figure 10.16 and Figure 10.17 show the solution quality of decomposition with uniform

random rounding and delayed result selection versus the solution quality of decomposition with "around B_0" rounding and delayed result selection for various angles of rotation, for a benchmark of 100 uniform random problems. A small angle means less rotation and planes defined by poles closer in the vicinity of B_0 are used, which will therefore exhibit less variation. The highest variability was found at 108° with a slight decline for larger angles, which corresponds to the best average solution quality (also at 108°), and the results suggest a strong correlation between variability and solution quality (when using delayed result selection on decomposed problems) for this unstructured benchmark. At 108° "around B_0" rounding achieves roughly the same, but slightly lower, solution quality as random rounding (as seen in Figure 10.16). This angle corresponds roughly to a 90° rotation which, when considering symmetry, means the entire hypersphere is covered, although the random vectors are distributed differently (more densely near B_0) compared to the uniformly random rounding method.

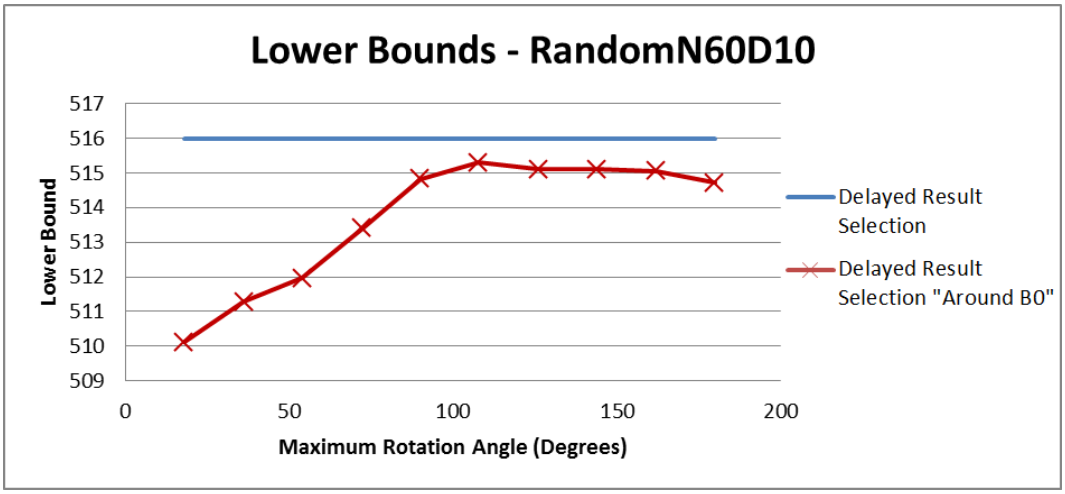


Figure 10.16: Solution quality by rotation angle. "Random N60D10 benchmark", all using same decomposition

These results suggest that for the "delayed result selection" method (of solving a problem while using decomposition) to work, it requires a diverse, varied set of variable assignments per part. That way, the Cartesian product of these assignments (which are good solutions to their respective parts) has a reasonable chance of containing an assignment that results in a high lower bound, and thus of finding a good solution to the overall problem.

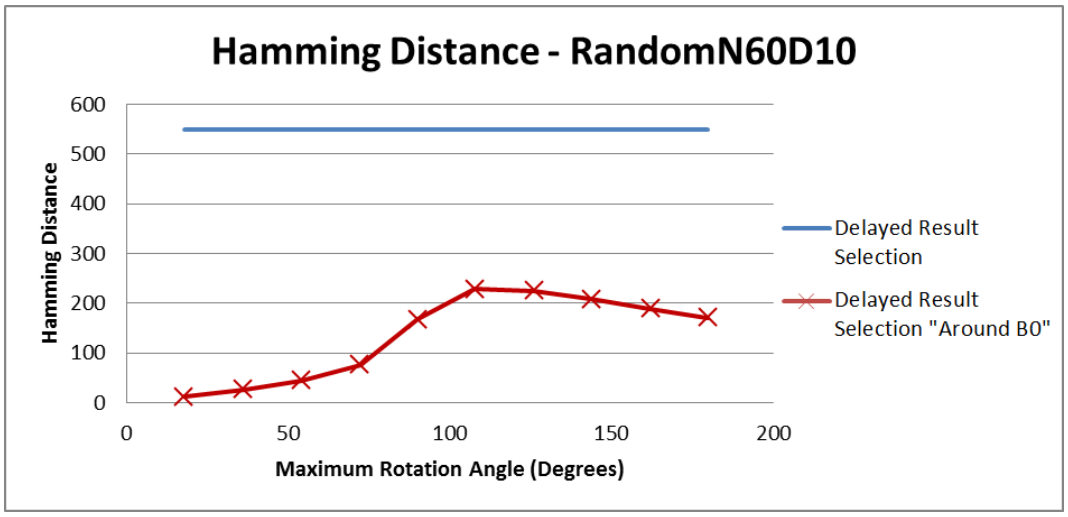


Figure 10.17: Variation of solutions by rotation angle. “Random N60D10” benchmark, all using same decomposition

Chapter 11

Comparison with other Max-Sat solvers

The performance of the SDP solvers and the decomposition and rounding methods was compared to complete and local-search Max-Sat solvers, the results of which are presented in this chapter.

The complete solver, MSUnCore2, is able to solve some problems very quickly, quicker than any of the alternatives that we tried, but is unable to solve many problems that have a higher density, large number of variables, or simply happen to be very difficult. For those problems which it is able to solve efficiently, a complete solver such as MSUnCore2 is clearly the preferred method, since it is fast and provides an optimal solution.

For those problems that a complete solver is unable to solve, SDP is the only viable alternative for deriving upper bounds. Unfortunately, our decomposition techniques were unable to contribute to letting SDP find upper bounds more efficiently.

The local search solver SAPS can solve problems very rapidly, and finds remarkably good lower bounds for every benchmark we've tried. Those problems that a complete solver is able to solve, the local search solver was often able to solve optimally or near optimally. For all other problems, it was usually able to find lower bounds that were very slightly better, on average, than the lower bounds found by SDP (when using a modest number of hyper-plane roundings), and was able to do so much faster. Presumably there exist instances where a local search solver is not able to achieve such good results, but we were unable to construct those.

It is therefore no surprise that after decomposition, our lower bounds (which were very unlikely to be superior to those provided by SDP) are not better than those found by SAPS. Although we were able to find those lower bounds faster than SDP without decomposition, solve times using this method are not competitive with those of the SAPS local search solver either. Furthermore, even though the lower bounds achieved following decomposition generally come within 2% of those found by SDP (without decomposition), this 2% error rate is large compared to the difference between SDP and local search, and sufficiently large to not make this method worthwhile, even if it could achieve greater speedup.

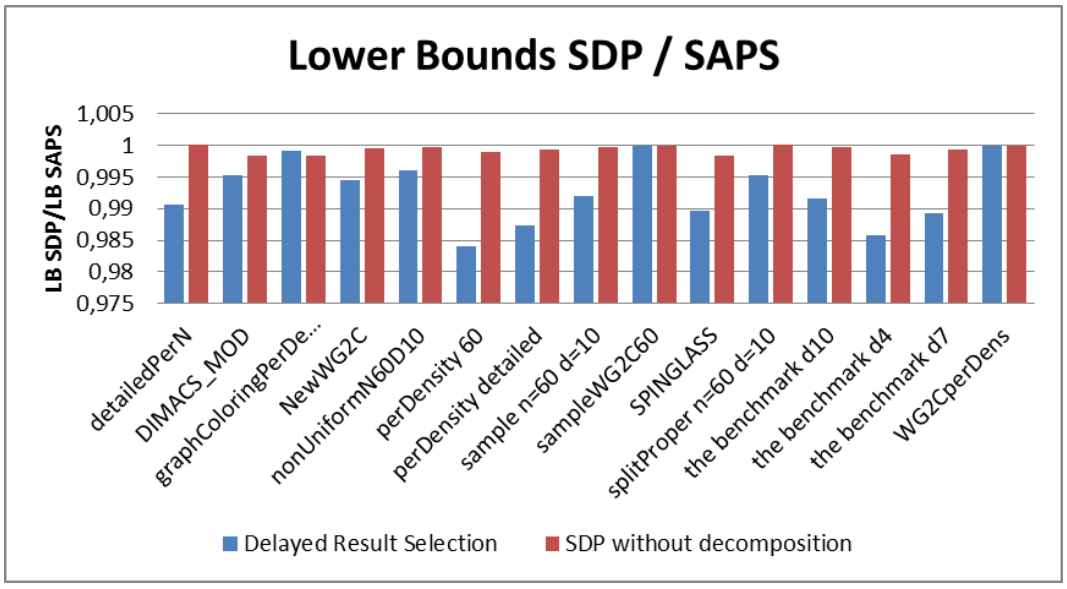


Figure 11.1: Comparison of SDP, decomposed SDP and SAPS lower bounds.

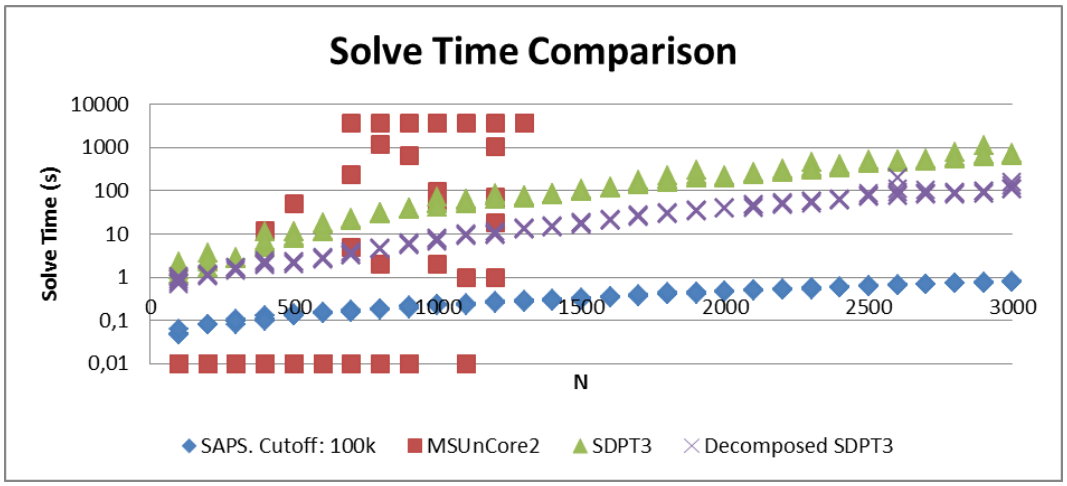


Figure 11.2: Solve times w.r.t. N for various solvers. Uniform random problems of density 2.5.

Chapter 12

Conclusion

SDP is still a relevant approach, as it is able to provide upper bounds and performance guarantees for problems that complete solvers are unable to solve efficiently. It can solve problems of up to problem sizes of several thousand variables on a regular workstation. It is outperformed, however, by local search algorithms when it comes to finding lower bounds.

Decomposition in the manner described in this report allows SDP solvers to solve problems with a high number of variables more rapidly and can find lower bounds that are close to those found by SDP without decomposition. However by doing so, we can no longer guarantee the quality of those lower bounds, and solving problems in this manner does not provide useful upper bounds. Using SDP to solve decomposed problems in this manner can not compete with conventional incomplete local search algorithms in either solve time or solution quality.

Bibliography

[1] L. Knox, N. Christensen, and C. Skordis, "The age of the universe and the cosmological constant determined from cosmic microwave background anisotropy measurements," *The Astrophysical Journal Letters*, vol. 563, p. L95, 2001.

[2] J. Marques-Silva, "The msuncore max-sat solver," *SAT 2009 competitive events booklet*, p. 151, 2009.

[3] Z. Fu and S. Malik, "On solving the partial max-sat problem," *Theory and Applications of Satisfiability Testing*, pp. 252–265, 2006.

[4] F. Hutter, D. Tompkins, and H. Hoos, "Scaling and probabilistic smoothing: Efficient dynamic local search for sat," *Principles and Practice of Constraint Programming*, pp. 241–249, 2006.

[5] F. Potra and S. Wright, "Interior-point methods," *Journal of Computational and Applied Mathematics*, vol. 124, pp. 281–302, 2000.

[6] M. Anjos, "Semidefinite optimization approaches for satisfiability and maximum-satisfiability problems," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 1, pp. 1–47, 2005.

[7] F. Alizadeh, J. Haeberly, and M. Overton, "Primal-dual interior-point methods for semidefinite programming: convergence rates, stability and numerical results," *SIAM Journal on Optimization*, vol. 8, no. 3, pp. 746–768, 1998.

[8] M. Krom, "The decision problem for a class of first-order formulas in which all disjunctions are binary," *Mathematical Logic Quarterly*, vol. 13, pp. 15–20, 1967.

[9] M. Goemans and D. Williamson, "Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming," *Journal of the ACM (JACM)*, vol. 42, no. 6, pp. 1115–1145, 1995.

[10] U. Feige and M. Goemans, "Approximating the value of two power proof systems, with applications to max 2sat and max dicut," pp. 182–189, Theory of Computing and Systems, 1995. Proceedings., Third Israel Symposium on the, 1995.

[11] M. Todd, "A study of search directions in primal-dual interior-point methods for semidefinite programming," *Optimization methods and software*, vol. 11, pp. 1–46, 1999.

[12] R. Ttnc, K. Toh, and M. Todd, "Solving semidefinite-quadratic-linear programs using sdpt3," *Mathematical programming*, vol. 95, no. 2, pp. 189–217, 2003.

[13] J. Sturm, "Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones," *Optimization methods and software*, vol. 11, pp. 625–653, 1999.

[14] S. Benson, Y. Ye, and X. Zhang, "Solving large-scale sparse semidefinite programs for combinatorial optimization," *SIAM Journal on Optimization*, vol. 10, no. 2, pp. 443–461, 2000.

[15] H. Mittelmann, "The state-of-the-art in conic optimization software," 2010.

[16] Y. Saad, *Iterative methods for sparse linear systems*. Society for Industrial Mathematics, 2003.

[17] S. Arora, E. Hazan, and S. Kale, "O (? log n) approximation to sparsest cut in o (n2) time," pp. 238–247, Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on, IEEE, 2004.

[18] K. Xu, "Weighted max-2-sat benchmarks with hidden optimum solutions."

Appendix A

Results

A.1 Solution Quality Distribution for 100 Random Decompositions

These results show the frequency at which certain lower bounds were attained, when applying independent solving to 100 random decompositions of a single problem.

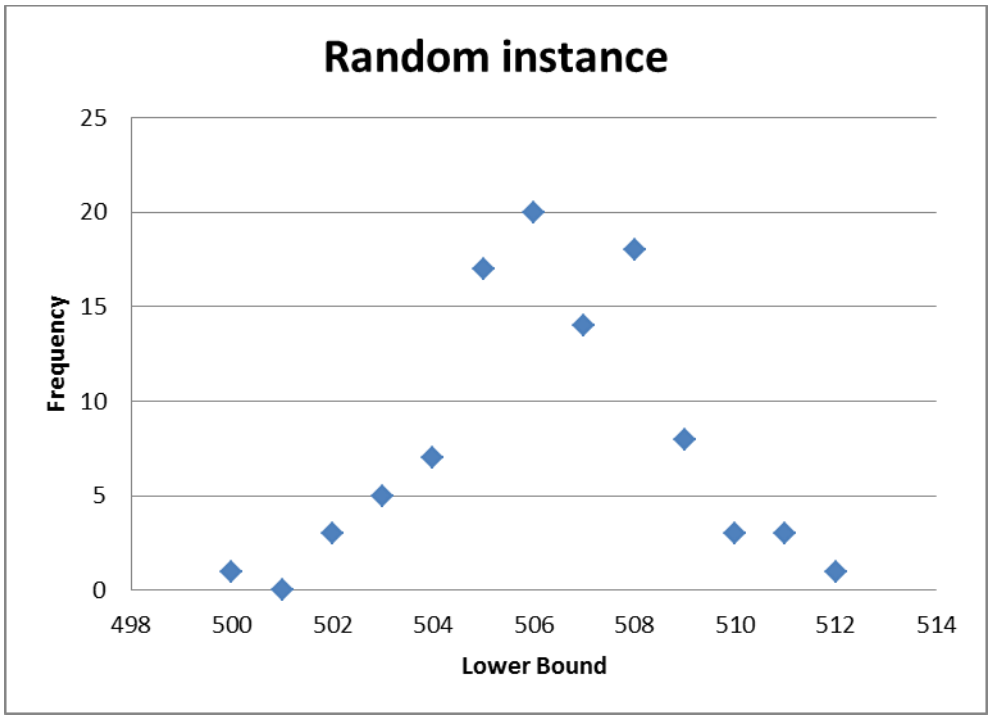


Figure A.1: Random problem N=60, D=10

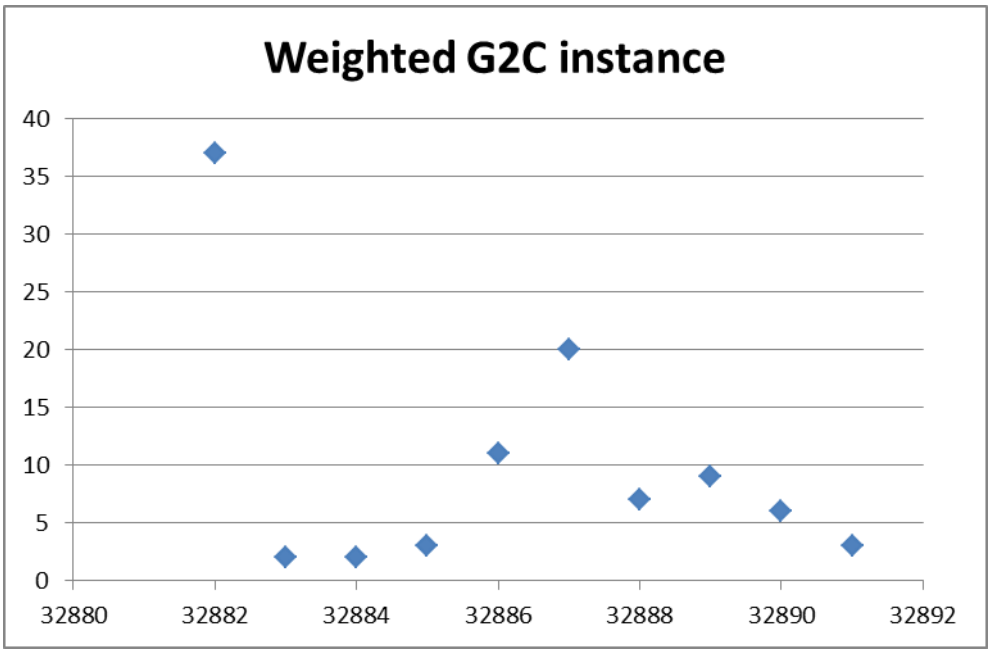
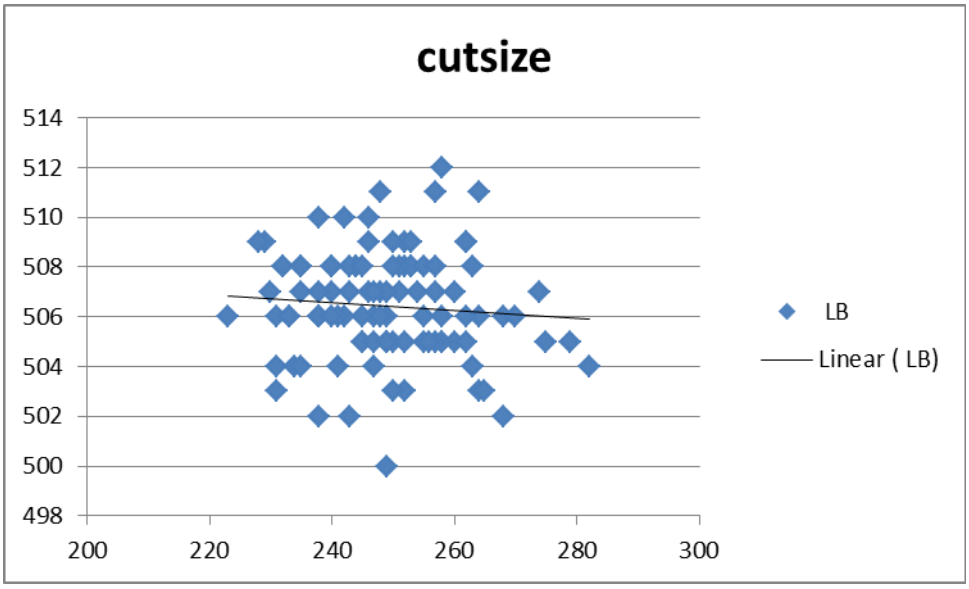


Figure A.2: Weighted Graph-2-Colorability instance. N=80, D=10

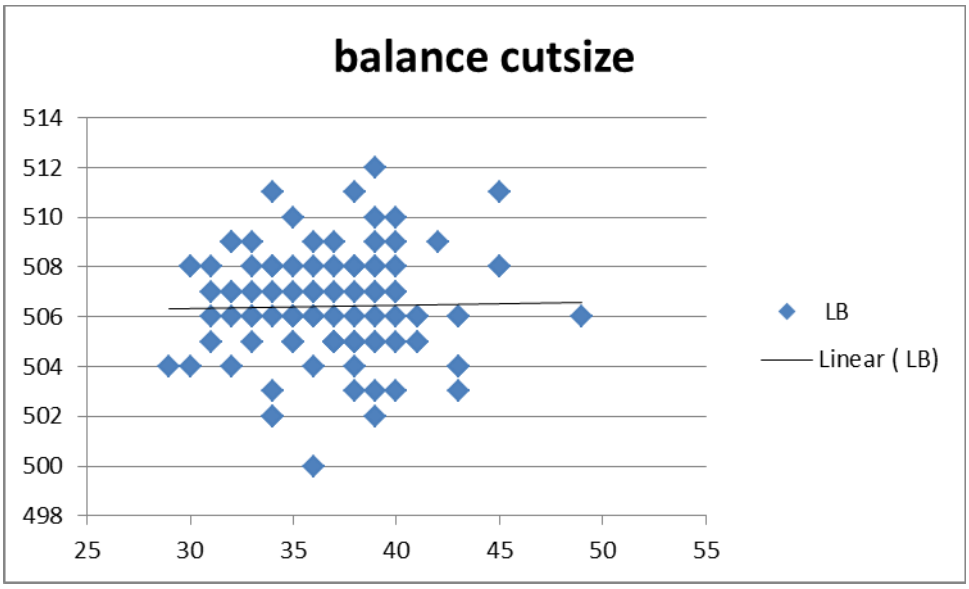
A.2 Solution Quality Characterizations

These results show a variety of metrics used to identify random decompositions, plotted against the lower bounds achieved using those decompositions. The purpose of these graphs was to find a correlation between these metrics and the quality of the lower bounds, but none of these metrics proved useful for developing a heuristic to decompose problems effectively.

Decomposition $M1 \mid M2$, where $|M1|, |M2| = 50\%$. Let $V = v_1 \ldots v_n$ be the variables, and $C = c_1 \ldots c_m$ the clauses. Let $t_i = v_i$ *or* $\overline{v_i}$. Let w_i be the weight of clause c_i.



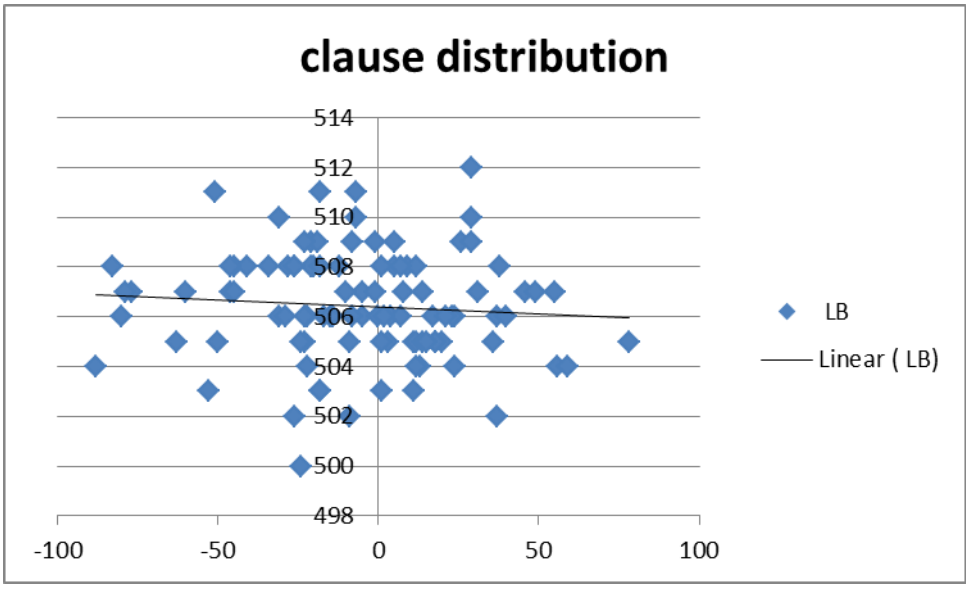
$$cutsize = \sum_{(t_i \vee t_j) \in C,\ v_i \in M1, v_j \in M2} w_i$$



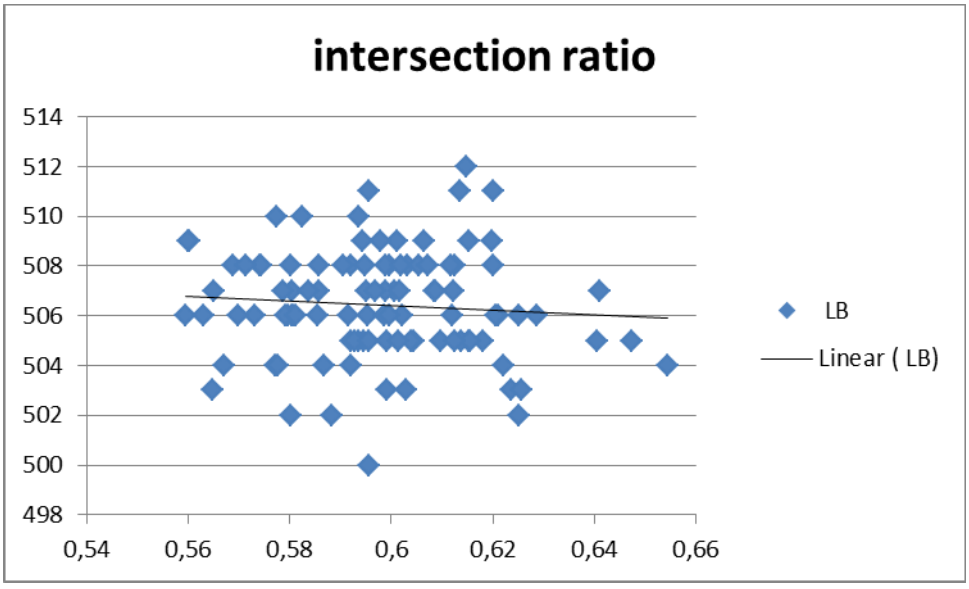
$$sign(v_i) = \sum_{(t_i \vee t_j) \in C} w_i * \begin{cases} +1 \; if \; t_i = v_i \\ -1 \; if \; t_i = \overline{v_i} \end{cases}$$

$$contraSignClause(v_i,\ (t_i \vee t_j)) = true \; iff \; sign(v_i) \neq 0 \; AND \; sign(v_i) > 0 \; \neq \; sign(t_i) > 0$$

$$balance\ cutsize = \sum_{c=(t_i \vee t_j) \in C,\ v_i \in M1, v_j \in M2} \begin{cases} 1 \; if \; contraSignClause(v_i, c) \; AND \; contraSignClause(v_j, c) \\ 0 \; otherwise \end{cases}$$



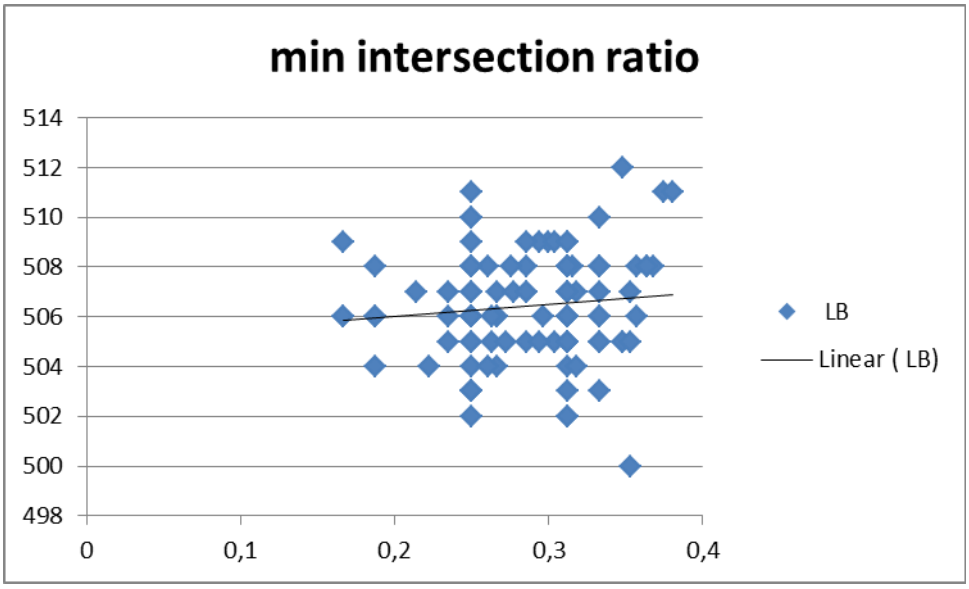
$$clause\ distribution = \left(\sum_{(t_i \vee t_j) \in C,\ \ v_i \in M1} w_i \right) - \left(\sum_{(t_i \vee t_j) \in C,\ \ v_i \in M2} w_i \right)$$



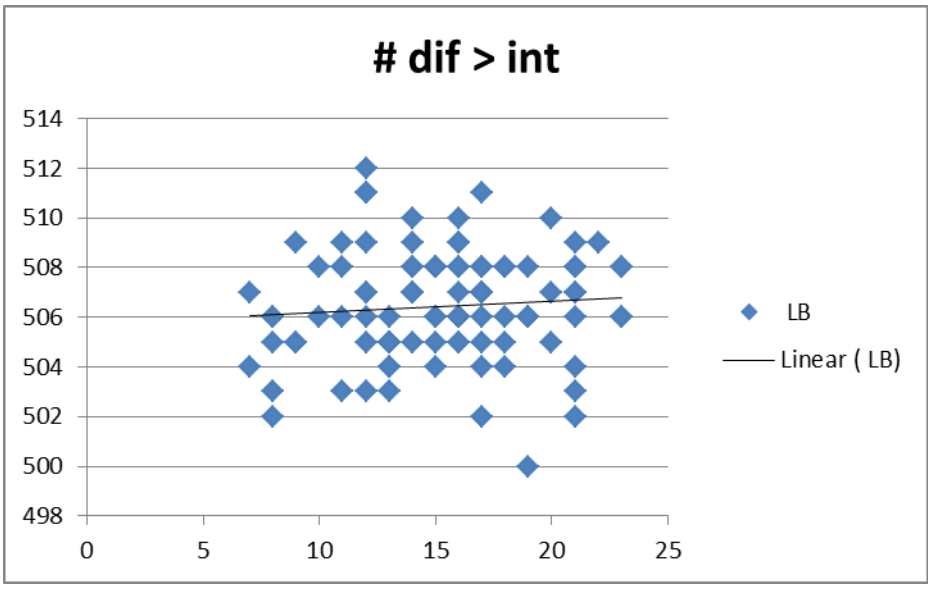
$$intersection(v_i) = \{c | c = (t_i \vee t_j) \in C,\ v_i \in M_x, v_j \in M_x\}$$

$$difference(v_i) = \{c | c = (t_i \vee t_j) \in C,\ v_i \in M_x, v_j \notin M_x\}$$

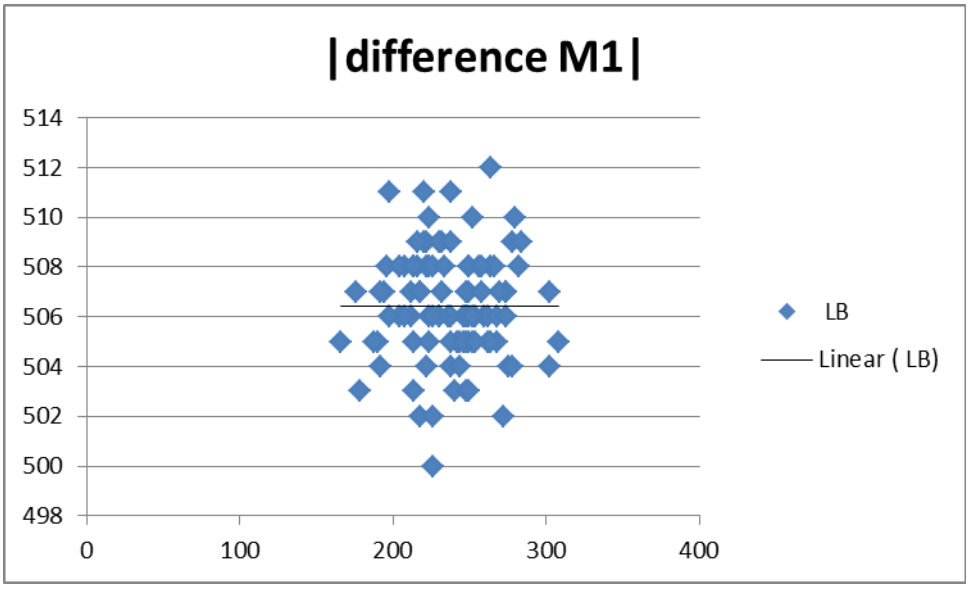
$$intersection\ ratio = \frac{1}{n} \sum_{v_i \in V} \frac{|intersection(v_i)|}{|intersection(v_i)| + |difference(v_i)|}$$



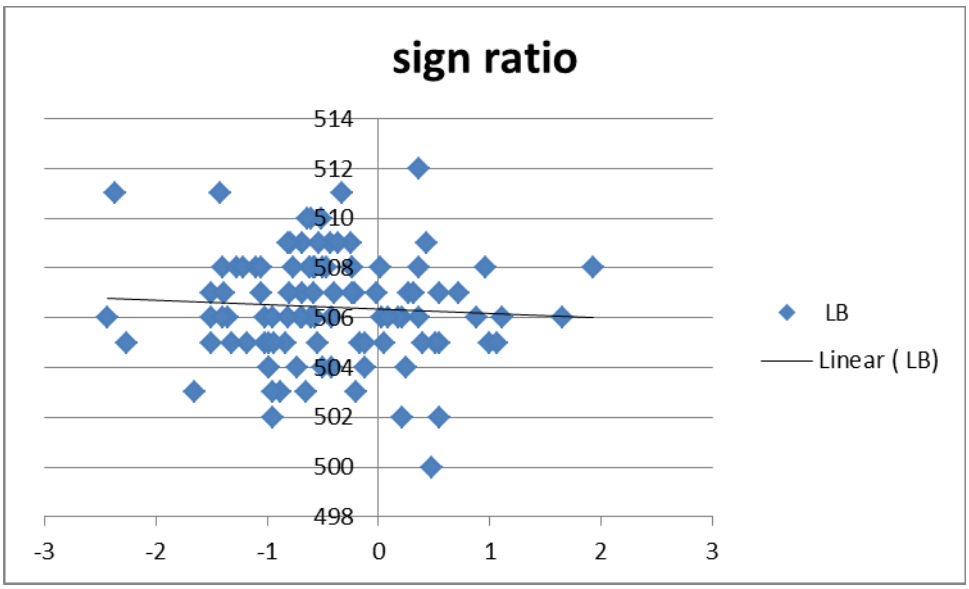
$$\min intersection\ ratio = \min_{v_i \in V} \frac{|intersection(v_i)|}{|intersection(v_i)| + |difference(v_i)|}$$



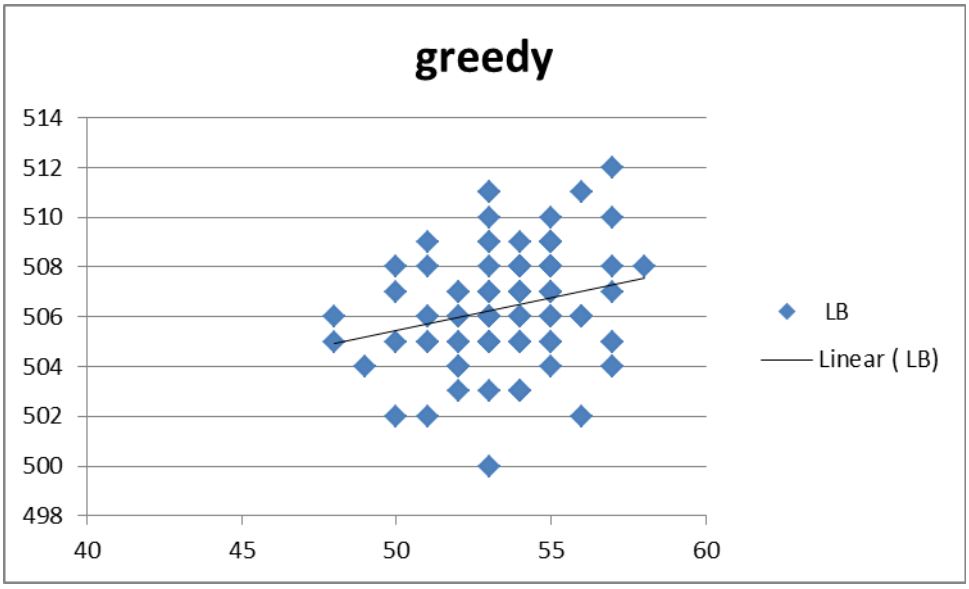
$$"\#dif > int" = |\{v_i \mid difference(v_i) > intersection(v_i)\}|$$



$$|difference\ M1| = \sum_{v_i \in M1} |difference(v_i)|$$



$$sign\ ratio = \frac{1}{n} \sum_{v \in V} sign(intersection(v)) * sign(difference(v))$$

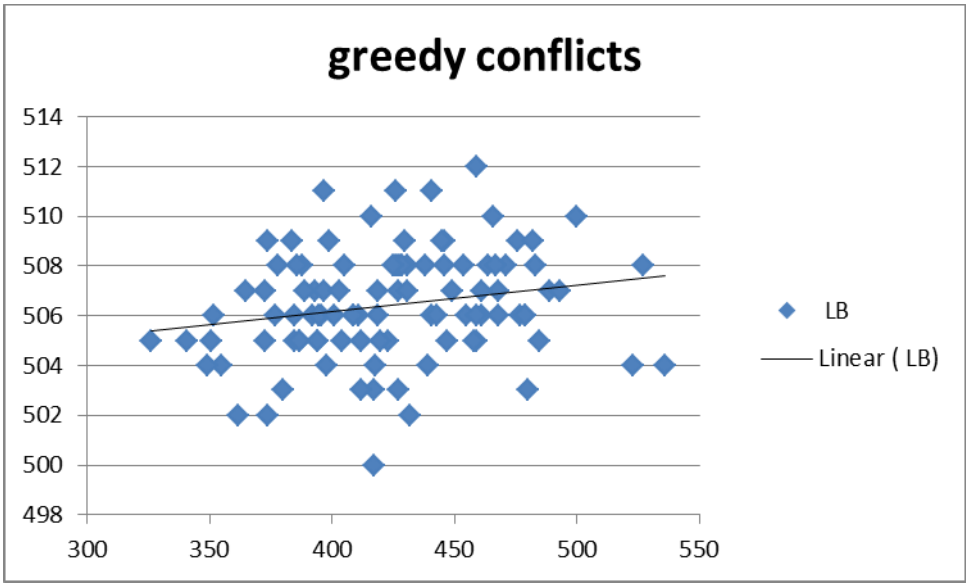


$$pos\left(intersection\left(v_i\right)\right) = \sum_{c=\left(t_i \vee t_j\right), c \in intersection(v_i),\ t_i = v_i} w_i$$

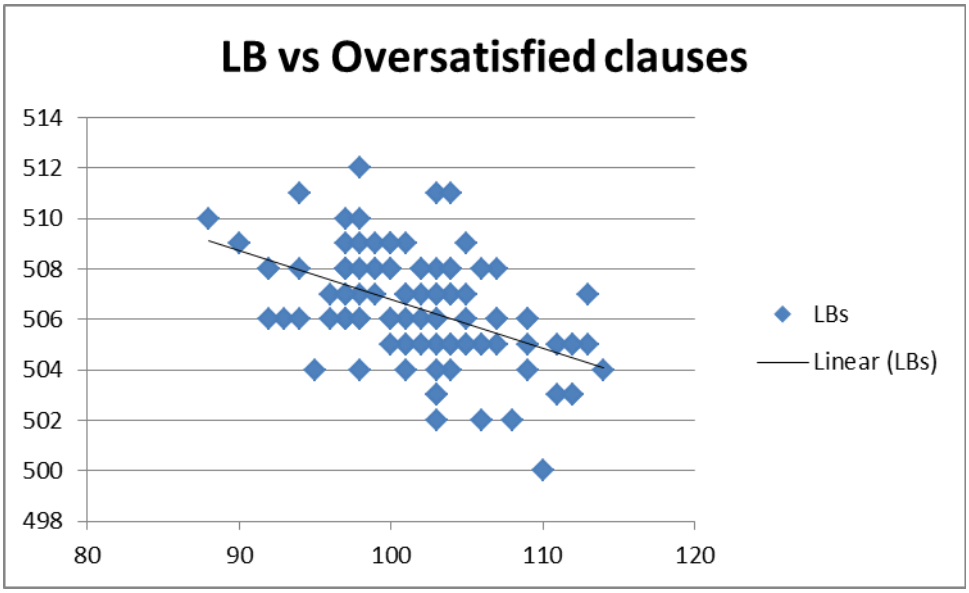
$$neg\left(intersection\left(v_i\right)\right) = \sum_{c=\left(t_i \vee t_j\right), c \in intersection(v_i),\ t_i = \overline{v_i}} w_i$$

$$greedy(v) = \begin{array}{l} pos\left(difference\left(v\right)\right) + \frac{pos(intersection(v))}{2} > neg\left(difference\left(v\right)\right) + \frac{neg(intersection(v))}{2} \; OR \\ neg\left(difference\left(v\right)\right) + \frac{neg(intersection(v))}{2} > pos\left(difference\left(v\right)\right) + \frac{pos(intersection(v))}{2} \end{array}$$

$$greedy\ intersection = \left|\left\{v \mid v \in V,\ greedyIntersection\left(v\right)\right\}\right|$$



$$greedy\ conflicts = \sum_{v_i \in V,\ greedy(v_i)} \left|\left\{v_j \;\middle|\; \left(t_i \vee t_j\right) \in C,\ greedy\left(v_j\right)\right\}\right|$$



$$oversatisfied\ clauses = \sum_{c_i\ satisfied\ by\ solution\ M1} w_i + \sum_{c_j\ satisfied\ by\ solution\ M2} w_j - \sum_{c_k\ satisfied\ by\ solution\ M1 \cup M2} w_k$$